Generating incremental attribute evaluators

Het genereren van incrementele attribuut evaluatoren

(met een samenvatting in het Nederlands)

Proefschrift

ter verkrijging van de graad van doctor aan de Universiteit Utrecht op gezag van de Rector Magnificus, Prof. dr. J.A. van Ginkel, ingevolge het besluit van het College van Decanen in het openbaar te verdedigen op vrijdag 25 november 1994 des middags te 12:45 uur

door

Maarten Christiaan Pennings

geboren op 28 december 1965
te Utrecht

Promotor: Prof. dr. S.D. Swierstra
Co-promotor: Dr. M.F. Kuiper

Faculteit Wiskunde en Informatica

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Pennings, Maarten Christiaan

Generating incremental attribute evaluators / Maarten
Christiaan Pennings. - Utrecht : Universiteit Utrecht,
Faculteit Wiskunde en Informatica
Proefschrift Universiteit Utrecht. - Met index, lit. opg.
- Met samenvatting in het Nederlands.
ISBN 90-393-0665-6
Trefw.: attributen (computerprogramma's) /
programmeertalen / compilers.

NWO

These investigations were supported by the Netherlands Computer Science Research Foundation (SION) with the financial support from the Netherlands Organization for Scientific Research (NWO).

At the back cover: the initial abstract syntax tree (left) of the VARUSE grammar defined in this thesis and the same tree (right) after augmenting, splitting and elimination.

Preface

In the passed four years, I was often asked by friends and relatives what my research was about. If the question was raised by an insider of the world of computing science I would drop the keyword "compiler generator", and we would talk shop for a while. The other scenario, the question coming from an outsider, was perhaps even more appealing. I always felt challenged to explain specialized matters to others, and I still do.

With outsiders, I started talking about software familiar to them—a wordprocessor whose brand I shall not name here. I then explained that such software is written by programmers in programming languages like "Pascal" or "c". The next point in my exposition appeared critical quite often: a program written by a programmer needs to be translated by a "compiler" before a computer can actually execute it. When I then finally told that a compiler is also a program that can be generated by yet another program "my research topic", I usually lost my audience allthough they were mostly too polite to admit it.

In short, my thesis is about a program that generates compilers, or, using insiders jargon, about a program that generates attribute evaluators. I have done my best, joyfully, to write an understandable thesis. I have included lots of pictures to enlighten the subject and lots of marginal notes that informally elaborate on the matters in the main text. However, it is the destiny of computing scientist that their research is for insiders. I hope that if you are one, you do not get lost in my thesis.

I would like to thank Doaitse Swierstra, my promotor, for offering me an OIO position and introducing me to the field of compiler *generators*. His ideas, often wild but always brought with enthusiasm, are stimulating. I enjoyed the various discussions with him, including the ones about computing science. In the same breath I must mention Matthijs Kuiper, who is my co-promotor. Together, we spend numerous hours on writing the generator and identifying research problems. He also tried to teach me to write concise. Wherever this thesis is not concise, I am to blame.

Furthermore, I would like to thank the reading-committee, dr. H. Alblas, dr. M. Jourdan, prof. dr. F.E.J. Kruseman Aretz, prof. L.G.L.T. Meertens, and prof. dr. ir. M.J. Plasmeijer for reviewing my thesis. Chritiene Aarts meticulously worked his way through a draft version of my thesis finding many errors. I would also like to thank Harald Vogt, who was a pleasant roommate for a too short period, for commenting on previous versions of this text.

Numerous years of research in computing science have gone by, and still there is no document preparation system that one can operate without getting a headache now and then. I thank Chritiene Aarts and Rob Udink who were always willing to discuss LaTeX-quirks. By the way, the previewer with encapsulated postscript support has become a real paper-saver now.

The Vakgroep has always been a pleasant place to work. I need only to bring into mind the (too) few sailing days, the "film-list" and "skwosj-list".

Finally, I thank my family and friends for their interest and moral support. In particular I want to thank my parents for stimulating my curiosity. And I thank Debbie, for being there.

C

Contents

List of Figures

The VARUSE example

The box example

The binding example

Algorithms

Grammar class examples

Other figures

Chapter 1

Introduction

This thesis is concerned with several aspects of generators for incremental attribute evaluators. We introduce evaluators that are purely functional. Functional evaluators allow for several optimizations aimed at improving incremental behavior. This thesis also presents a new algorithm for scheduling the work in evaluators. To examine the feasibility of our approach, we have implemented an evaluator generator.

Attribute grammars are a formalism for describing formal languages such as programming languages. An attribute grammar describes the computation of attributes, objects attached to tree nodes. An attribute grammar defines the structure of trees, associates attributes with tree nodes and specifies how attributes of adjacent nodes depend on each other. An attribute evaluator takes as input a tree and computes the values of all attributes attached to the nodes of that tree, a process known as decoration. After an edit action that changes the input tree, an *incremental* attribute evaluator does not redecorate the new tree from scratch; it uses the previous results to speed up decoration.

Because (incremental) attribute evaluators can be constructed automatically from an attribute grammar, attribute grammars are not just definitions. Software tools can be defined by an attribute grammar. An implementation of such a tool can be obtained by generating an attribute evaluator from its attribute grammar.

We discuss the implementation of generators for incremental attribute evaluators. In particular, we discuss a new approach that uses functions to decorate trees. These so-called visit-functions are memoized to obtain incremental evaluation. This thesis also addresses the feasibility of evaluation with memoized visit-functions: a generator has been implemented.

1.1 Motivation

The simple mechanism of computing the values of attributes associated with tree nodes has a wide range of applications [DJL88]. With an attribute grammar one can compute something as simple as the minimal value in a tree. But attribute grammars are also used to describe compilers [KHZ82]. Other application of attribute evaluators are language based editors, program transformation systems, document preparation systems and verification tools [RT88, DJL88].

For example, generators have been used to create compilers for different kind of languages with varying structures: plain programming languages, specification languages, grammar definitions. Such compilers feature type checking, detecting programming anomalies and code generation. Furthermore, transformation systems for functional programming languages have been implemented. The generated system checks the correctness of transformations. Document preparation tools have been constructed for plain text, mathematical formulas, dictionary entries, or tax forms to name but a few. Such systems are mainly used for pretty printing, that is, creating consistent, high-quality typeset output.

A language based editor can be applied in each of these examples. Such editors consist of two parts: a user interface that lets the user manipulate the "program" under construction and an attribute evaluator that is applied after each change to the program. As a result of attribute evaluation, a language based editor may detect errors in the program against the static semantics of the language it is constructed for. Furthermore, a language based editor may be queried by the user about properties of the program under construction. These two features make that language based editors increase the productivity of its users.

A language based editor is an editor with knowledge about the language it is constructed for ("Which variables are declared here?").

Incremental evaluators can be used wherever plain evaluators are applied; in particular, they can be used in interactive systems such as language based editors.

The delay between user input and system response is an extremely important aspect of interactive systems. Drawing programs, desk top publishing systems and spreadsheets are famous examples of non grammar based interactive tools that use incremental techniques to decrease system response time. Incremental algorithms are hard to write by hand. On the other hand, we observe a still growing need for interactive software. Generated incremental language based editors alleviate this friction.

Many drawing programs have a button [REDRAW SCREEN] to be used when the incremental refresh algorithm leaves some pixels in the wrong color.

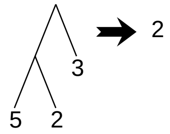
The use of generators reduces the production time for an evaluator considerably. This makes generators especially useful for prototyping or for writing evaluators for specialized languages. A disadvantage of generated evaluators is that their efficiency is lower than that of hand written ones. Our work is an approach to overcome the time penalty for generated evaluators by using incremental techniques. Observe that improvements in the generator improve any generated evaluator.

1.2 A formal motivation

This section gives an abstract motivation for research in incremental attribute evaluation. It explains that computations described by attribute grammars are easily made incremental. This section also shows that the class of problems that can be specified by an attribute grammar is large. Non-interested readers may skip this section. The next one informally introduces attribute grammars.

Many problems require an incremental solution. For *compositional* problems an incremental solution is easily obtained. Compositional problems are problems defined as homomorphisms on an inductive structure (lists, trees). A homomorphism h is defined on the "basis" $\langle\cdot\rangle$ of the inductive structure $h\ \langle i\rangle = \langle\langle i\rangle\rangle$ and on the "step" $\bullet$ of the inductive structure $h\ (a \bullet b) = h\ a \bullet\!\bullet\ h\ b$, where $\langle\langle\cdot\rangle\rangle$ is some unary and $\cdot \bullet\!\bullet \cdot$ some binary operation. A change to one of the basic values $\langle i\rangle$ in such a term, requires only recomputation alongside the path from the tip $\langle i\rangle$ to the root of the term. For a balanced term of size n, this implies $\log n$ re-applications of $\bullet\!\bullet$. In other words, compositional problems are well suited for incremental evaluation.

As an example consider the problem of finding the minimal element in a binary tree of integers. Binary trees are defined by the following type definition. Note that **tip** is the basic constructor ($\langle\cdot\rangle$) and **fork** the step ($\bullet$).



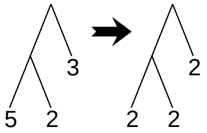
$$\begin{array}{lcl}\mathbf{type}\ \mathit{tree} & = & \mathbf{tip}(\mathit{int}) \\ & | & \mathbf{fork}(\mathit{tree}, \mathit{tree})\end{array}$$

Function f presented below, gives a solution to the posed problem.

$$\begin{array}{ll}f :: \mathit{tree} \longrightarrow \mathit{int} & \\ f\ \mathbf{tip}(i) & = i \\ f\ \mathbf{fork}(l, r) & = f\ l\ \mathbf{min}\ f\ r\end{array}$$

One may wonder whether the class of compositional problems is not too restricted: data flow is essentially bottom-up, which appears not very expressive. However, if we allow higher-order functions, top-down flow can be mimicked. Essentially, this means that $h\ a$ and $h\ b$ are functions instead of ordinary values—a higher order aspect. In that case $\bullet\!\bullet$ is a higher-order function, such as functional composition, so that the value for $h\ (a \bullet b)$ is a function too; let us call it g. By passing g an argument, we realize top-down flow.

We extend our previous example to illustrate this. We are required to transform a binary tree into one of the same shape, but with all the tips replaced by the minimal value. Function f applied to a tree t will return a tuple (v, g), whose first component v is the minimal tip value of t, and whose second component g is a higher-order function. When g is passed an integer m, it returns a tree of the same structure as t but with tip values m.



$$\begin{array}{ll}f :: \mathit{tree} \ \longrightarrow\ (\ \mathit{int}\ ,\ \mathit{int} \longrightarrow \mathit{tree}\ ) & \\ f\ \mathbf{tip}(i) & = (\ i\ ,\ [\lambda m : \mathbf{tip}(m)]\ ) \\ f\ \mathbf{fork}(l, r) & = f\ l \otimes f\ r \\ \quad\quad \mathbf{where}\ (v_l, g_l) \otimes (v_r, g_r) = (\ v_l\ \mathbf{min}\ v_r\ ,\ [\lambda m : \mathbf{fork}(g_l\ m, g_r\ m)]\ ) & \end{array}$$

The requested answer is then easily computed by the following function.

$$\begin{array}{l}\mathit{ans} :: \mathit{tree} \longrightarrow \mathit{tree} \\ \mathit{ans}\ s = t \\ \quad\quad \mathbf{where}\ \begin{array}{ll}(v, g) & = f\ s \\ t & = g\ v\end{array}\end{array}$$

Here, "strict" means that all the arguments of a function must be known in advance; more specifically they may not depend on the functions result. This is a liberal interpretation of the usual definition.

However, the above program can be shortened if we allow non-strict functions. In that case, function f has two arguments: the source tree and an integer that will replace all the tip values. Function f returns the minimal value of the source tree and the target tree.

$$\begin{array}{l}
f :: \mathit{tree} \longrightarrow \mathit{int} \longrightarrow (\,\mathit{int}\,,\,\mathit{tree}\,) \\
f\ \mathbf{tip}(i)\ m \qquad = (\ i\ ,\ \mathbf{tip}(m)\ ) \\
f\ \mathbf{fork}(l,r)\ m\ = f\ l\ m \otimes f\ r\ m \\
\qquad \mathbf{where}\ (v_l,t_l) \otimes (v_r,t_r) = (\ v_l\ \mathbf{min}\ v_r\ ,\ \mathbf{fork}(t_l,t_r)\ )
\end{array}$$

The ans function now uses that f is non-strict in its second argument.

$$\begin{array}{l}
ans :: \mathit{tree} \longrightarrow \mathit{tree} \\
ans\ s = t \\
\qquad \mathbf{where}\ (v,t) = f\ s\ v
\end{array}$$

The example in this section comes from Bird [Bir84]. In 3.4.1 it is treated as an attribute grammar problem.

We conclude with the observation that the class of compositional problems allows for good incremental update and is large enough for practical applications, provided that we allow non-strict, though not circular, homomorphisms. The attribute grammar formalism describes precisely this class of problems, and is thus an ideal framework to specify incremental computations.

1.3 Attribute grammars

A definition of a formal language usually consists of concrete syntax, abstract syntax and descriptions of static and dynamic semantics. Programming languages are a typical example of formal languages. Elements of a language shall be referred to as *programs*. The concrete syntax of a language definition specifies which sequences of characters establish a syntactically valid program. The abstract syntax defines trees; with each syntactically valid program an abstract syntax tree is associated. Not every abstract syntax tree describing a syntactically valid program corresponds to a semantically valid program. The abstract syntax tree might violate semantic properties, called context conditions, defined by the static semantics of the language. The dynamic semantics concerns the actual *meaning* of the program. It is usually described by defining a relation between the programs input and output.

Attribute grammars constitute a formalism for formal language definitions. An attribute grammar includes a context-free grammar describing the concrete and abstract syntax. Furthermore, an attribute grammar associates so-called attributes with tree nodes. Attributes of a node describe properties of the subtree rooted at that node and properties of the context of that node. Attributes of adjacent nodes functionally depend on each other. Static semantics is imposed by enforcing conditions on the attributes. Dynamic semantics can also be defined via attributes; for example, by associating an executable program describing the meaning of a tree with that tree.

We illustrate this with an example. Consider the assignment statement in a PASCAL like language. The context-free grammar will contain a production for the assignment statement that might look like

$$\mathit{assignment}\colon\ \mathit{statement} \longrightarrow \mathit{variable}\ \texttt{':='}\ \mathit{expression} \quad .$$

This production, named *assignment*, describes that a V ':=' E is a syntactically correct *statement* if string V is a syntactically correct *variable* and string E is a syntactically correct *expression*. Hence, the string ':=' must occur literally in assignments. The above production describes the concrete syntax. The associated abstract syntax, the constructor **assignment**, has the following definition. Note that does not mention literal strings.

$$\mathit{statement} = \mathbf{assignment}(\mathit{variable}, \mathit{expression})$$

When v is an abstract syntax tree for a syntactically correct *variable*, and e is an abstract syntax tree for a syntactically correct *expression*, then $\mathbf{assignment}(v, e)$ is an abstract syntax tree for a syntactically correct *statement*.

A typical context condition is the requirement that the type of the *expression* must be coercible to the type of the *variable*. In order to check for such a condition, attributes are introduced. A *variable* and an *expression* are both augmented with an attribute *type*. The condition can then be formulated as follows

$$\mathbf{condition}\ \mathit{coercible}(\mathit{variable.type}, \mathit{expression.type}) \quad ,$$

where the so-called semantic function *coercible* defines which types are coercible.

The value of the attribute *variable.type* denotes the type of the variable denoted by the subtree rooted at *variable*.

The type of a *variable* depends on its declaration elsewhere in the program. Similarly, the type of an *expression* depends on the types of its constituents which eventually might be variables. Usually, the declarations of a program induce a symbol table. The symbol table is passed to the *expression* by the following equation.

$$\mathit{expression.table} := \mathit{statement.table}$$

The dynamic semantics of an assignment statement, more informally known as its *code*, can also be characterized via attributes. An attribute *code* is associated with *variable*, *expression* and *statement*. The semantic function *code_assign* constructs the code for a statement out of the codes for the variable and the expression.

$$\mathit{statement.code} := \mathit{code\_assign}(\mathit{variable.code}, \mathit{expression.code})$$

The entities *variable*, *expression* and *statement* are known as the non-terminals of the attribute grammar. In production *assignment*, the non-terminal *statement* is known as the left-hand side non-terminal or parent. The other two non-terminals are known as right-hand side non-terminals or children of production $assignment$.

The attributes $type$ and $code$ are known as a synthesized attribute. Synthesized attributes contain information that is derived from the subtree to whose root they are attached. Inherited attributes, like $table$, contain information from the part of the tree surrounding the subtree to whose root they are attached.

1.4 Higher-order attribute grammars

The structure of an abstract syntax tree is fixed during decoration. This is a disadvantage. For example, conditions on attributes can be used to reject incorrect abstract syntax trees a posteriori, but they can not be used to guide construction of correct trees a priori. Higher-order attribute grammars [VSK89] were introduced to alleviate this shortcoming.

In higher-order attribute grammars, attributes that are tree valued may be decorated themselves. In other words, decoration of the abstract syntax tree computes a tree that may be instantiated—grafted into the abstract syntax tree—and decorated. Note the analogy with higher-order functions in functional languages.

Higher-order attribute grammars are interesting on two accounts.

First of all, it is advantageous that the abstract syntax tree, which guides decoration, can be computed. When a computation can not easily be expressed in terms of the inductive structure of the tree, a better suited structure can first be computed. For example, a multi-pass compiler can be modeled by computing an intermediate structure in an attribute upon which that attribute can be decorated, much like attribute coupled grammars [GG84].

The second reason is a fundamental one rather than a practical one. Every computation can be modeled through attribute evaluation in a higher-order attribute grammar. More specifically, execution of semantic functions can be replaced by decoration of higher-order attributes. Vogt has proven that higher-order attribute grammars without semantic functions (except identity functions and tree constructors) have the same expressive power as plain attribute grammars with semantic functions [Vog93]. In other words, semantic functions are redundant, which is interesting from an aesthetic point of view because the formalism for semantic functions is alien to the attribute grammar formalism.

From a practical point of view, it makes sense to promote using higher-order attributes instead of semantic functions. Functional programming is "harder" than writing attribute grammars since the latter are recursion free. Furthermore, during incremental evaluation a semantic function is either completely reevaluated or not at all. Thus, time consuming semantic functions cripple incremental attribute evaluation.

A typical application of higher-order attributes is to model symbol table lookups. The declarations in the program do not synthesize a symbol table as in a plain grammar setting, but a *tree* encoding the declared identifiers and their types. Symbol table lookup is achieved by first instantiating that tree, passing the variable under consideration as an inherited attribute to that tree and thirdly decorating that tree. The information associated with the variable will then be synthesized.

The following fragment illustrates symbol table lookup in the *assigment* production discussed in the previous section. The attributable attribute T computes the type of the *variable* used in the context condition.

```
**ata** $T : symbol\_table$;                         {declare attributable attribute $T$}
$T := statement.table$;                                              {instantiate $T$}
$T.id := variable.name$;                           {pass the name of the variable}
**condition** $coercible(T.type, expression.type)$          {use synthesized type}
```

Higher-order attributes are just a small enhancement to the attribute grammar formalism. As a result, standard evaluation techniques are also applicable to higher-order attribute grammars [VSK89]. However, the standard *incremental* techniques proved inefficient [TC90]. We developed a new approach based on memoized visit-functions [VSK91, PSV92a]. This thesis elaborates on that approach and discusses the the LRC processor that implements it.

1.5 Structure of this thesis

Chapter 2 introduces the attribute grammar formalism and the higher-order extension. Chapter 2 also introduces our notation with which we will denote grammar specifications. Furthermore, it defines a graphical notation for grammars that will be used extensively in this thesis.

Chapter 3 presents two methods for constructing incremental attribute evaluators. The first method is based on a visit-sequences driven tree walker. The second method maps visit-sequences to visit-functions that are memoized to obtain incremental behavior. Visit-functions based incremental evaluation handles higher-order attributes efficiently as opposed to the first method. However, visit-functions require bindings, a complex data structure. A large part of Chapter 3 contains a thorough definition of bindings.

With the introduction of bindings, decoration is purely functional. This allows for several optimizations, such as splitting, elimination, unification, folding, normalization and untyping. These optimizations are discussed in Chapter 4.

The third chapter only discusses how evaluators can be constructed from visit-sequences. Chapter 5 explains how the visit-sequences themselves are computed. We show how the ordered scheduling algorithm can be modified on two accounts. The first change (concerning step 4) causes a larger class of attribute grammars to be accepted, the so-called DAT grammars. Secondly, we have implemented a new scheduling algorithm, chained scheduling (new step 5), that optimizes visit-sequences with respect to incremental behavior.

Part of our research concerned the feasibility of incremental attribute evaluation with memoized visit-functions. To address this question, we have implemented our ideas in the LRC system. Implementation issues are discussed in Chapter 6, together with the first performance results.

Finally, Chapter 7 presents the conclusions.

1.6 Meta remarks

This thesis is intended to be rather self contained. That is why the VARUSE grammar, the running example of this thesis, is treated so thoroughly. This also explains the inclusion of the circularity test in Chapter 2 and Kastens' ordered scheduling algorithm in Chapter 5.

Pictures provide a good insight in dependency patterns. Static analysis of attribute grammars means analyzing dependencies. That is why pictures are useful in explaining attribute grammars which in turn accounts for the numerous pictures in this thesis.

It is thus also inevitable that references to figures occur frequently. To help the reader find the figure being referred to, page numbers are given as subscripts, except when the figure is placed at the page of reference or its facing page. For example, Figure 3.21 on page 67 will be referred to as Figure $3.21_{\triangleright 67}$. If only one page is to be turned, this is indicated with a single $\triangleleft$ or $\triangleright$ subscript.

The successive stages in the processing of the VARUSE grammar are illustrated with a figure. In order to be able to find related figures back easily, a list of figures is appended to the table of contents. The list of figures is subdivided by topic.

For the curious reader, a test run of the LRC processor—the evaluator generator written by us—is included. The VARUSE grammar has been used: Appendix A gives the source, Appendix B presents the generated C code and Appendix C lists the (diagnostic) output of the generated evaluator.

The notation for attribute grammars, defined in Chapter 2 is not standard. To facilitate the reading of subsequent chapters a glossary is appended to this thesis. It lists and shortly explains our notation.

A marginal note looks like this.

The margins contain side notes and small figures now and then. These marginal notes are intended to elaborate on the matter discussed in the main text. Usually they rephrase the main text in an informal way.

Chapter 2

The formalism

In this chapter we formally define context-free grammars, structure trees, attribute grammars and higher-order attribute grammars. Two formalisms are presented to specify (higher-order) attribute grammars: an algebraic and a graphical one. The notions of attributes, attribute occurrences, attribute instances and dependency graphs are introduced. The circularity test at the end of this chapter illustrates the definitions and gives a first taste of grammar analysis.

2.1 Notation

Compilers deal with languages and for each of these languages we need a notation. On top of that, we need a meta-notation to denote ordinary mathematical expressions and algorithms. However, we do not make a strict separation between the various languages in this thesis.

The *source* language for the evaluator generator will be defined in Section 2.3. For the *target* language, we will use a notation inspired by functional languages like GOFER [Jon91], a large subset of HASKELL [HF92, HPJW$^+$92]. Ordinary mathematics is also expressed in a functional fashion. Algorithms in this thesis are presented in a PASCAL like syntax.

A good introduction into functional programming is given by Bird and Wadler [BW88].

We denote functional application by juxtaposition and we use parentheses for grouping. For example, the application of f to x is denoted with $f\ x$. Conventional binary operators, like functional composition $\circ$ or addition $+$, will be written infix. Grammar related infix and prefix operators are written bold, as in for example $X\,\mathbf{S}\,v$ and $\mathbf{v}X$, or they are written with parentheses, as in $tds(X)$.

A *tuple* of comma separated elements is enclosed in parentheses. The empty tuple is denoted with $()$. The first respectively second element of a two-tuple or pair t is denoted with $fst\ t$ respectively $snd\ t$. In a context with many parentheses, straight parentheses $[\,]$ will be used for tuples.

Following mathematical conventions, a *set* is denoted by enclosing the comma separated elements in braces. The empty set is denoted by $\{\}$, union with $\cup$, disjunction with $\cap$ and element-of with $\in$. Sometimes, $+$ is used instead of $\cup$, to stress that the operands are disjoint. If $V = V_1 + V_2$ then V_1 and V_2 *partition* V. The crossproduct $V_1 \times V_2$ is the set of two-tuples $\{(v_1, v_2) \mid v_1 \in V_1 \wedge v_2 \in V_2\}$. *Types* are regarded as sets. Hence, a new type can be formed by applying the crossproduct to two existing types.

A *graph* is a pair (V, E) of vertices V and arcs E. If v and w are vertices, then $v \rightarrow w$ denotes an arc. In this thesis, the set of vertices is known and fixed for most graphs, which makes *dependency sets* more convenient. A dependency set is a set of arcs which represents a graph with implicit vertices. Standard set operations will be used for dependency sets. Transitive closure of a graph G (dependency set D) is denoted with G^+ (respectively D^+).

A *list* is a sequence of elements. A list is denoted by enclosing the comma separated sequence of its elements in square brackets. For example, $[4, 5, 2, 2, 9, 5]$ denotes a list of six natural numbers, $[\,]$ denotes the empty list and $[18]$ denotes a singleton list. For a set (type) V, the set $[V]$ consists of all lists that can be formed by concatenating zero or more elements from V.

The following operators on lists are defined: the $head$ of a non-empty list is its first element; the $tail$ of a non-empty list consists of all elements but the first, preserving the order. $head$ and $tail$ are *partial* functions since they do not take the empty list as argument. The len of a list is the number of elements in the sequence. If l is a list and i a natural number, and $0 \leq i < len\ l$, then $l{\cdot}i$ is the element in l with i predecessors. Concatenation of two lists is denoted with $+\!\!+$ and $x : l$ "cons" abbreviates $[x] +\!\!+ l$.

The final piece of notation for lists is called a *list comprehension*. It employs a syntax adapted from conventional mathematics for describing sets. A list $[E \mid P]$ contains the values of the expression E for which the predicate P holds. An order on the values is obtained by a special predicate of the form $x \leftarrow l$ which is the list-equivalent of the set predicate $a \in V$. The idea is that a so called *generator* $x \leftarrow l$ preserves the order of list l:

$$[\,2x \mid x \leftarrow [1, 2, 3, 4]\,] \;=\; [2, 4, 6, 8] \quad .$$

We will miss-use list comprehension to convert a set into a list in the following manner. The resulting order is arbitrary.

$[\,x \mid x \in \{1, 2, 3, 4\}\,]$ could be $[4, 1, 3, 2]$

A *term* is a recursive data structure. Terms are created by constructors, a special kind of function, denoted by bold symbols. For example, the declaration

$$\begin{array}{lll} \textbf{type}\ N & = & \textbf{zero}() \\ & \mid & \textbf{succ}(N) \end{array}$$

defines type N representing natural numbers, and constructors $\textbf{zero}$ and $\textbf{succ}$. The term $\textbf{succ}(n)$ is *constructed* by applying constructor $\textbf{succ}$ to the term n. *Destructing* a term means inspecting which constructor is applied to which subterms. In this thesis, we will often not distinguish between a term *type* and an *instance* of that type. For example, the following function that returns the half of a given natural number, uses pattern matching on terms. The third pattern uses N as a fresh variable of *type* N: symbol N is *overloaded*.

$$\begin{aligned}
&\mathit{half} :: N \longrightarrow N \\
&\mathit{half}\ \mathbf{zero}() &&= \mathbf{zero}() \\
&\mathit{half}\ \mathbf{succ}(\mathbf{zero}()) &&= \mathbf{zero}() \\
&\mathit{half}\ \mathbf{succ}(\mathbf{succ}(N)) &&= \mathbf{succ}(\mathit{half}\ N)
\end{aligned}$$

In the above example, the symbol :: denotes has-type. Note the difference in function application (denoted with white space) and constructor application (with brackets).

2.2 Grammars

In this section, we introduce attributegrammars [Knu68, Knu71] and the higher-order extension [VSK89, TC90, Vog93]. Our notation differs from other notations [Kas80, WG84, Kui89, Vog93] in order to avoid ambiguity, most notably between non-terminals and their occurrences.

See [WG84] for an introduction on grammars.

2.2.1 Context-free grammars

An attribute grammar is based on a context-free grammar.

Definition 1 *Context-free grammar.*
A context-free grammar is a triple $G = (V, P, S)$. V is a non-empty, finite set of symbols, partitioned into a non-empty set of non-terminal symbols N and a set of terminal symbols T. P is a finite set of productions. A production is a non-empty finite list of symbols whose first element is a non-terminal symbol. The start symbol S is a non-terminal symbol.
□

V is the vocabulary of grammar G. Non-terminal symbols are usually called *non-terminals* and terminal symbols are called *terminals*. Unless noted otherwise, we assume that $T = \{\}$, in which case, the context-free grammar is said to describe the *abstract syntax* as opposed to the *concrete syntax*.

Unlike most definitions of context-free grammars, we define productions as a non-empty list of non-terminals, as Kuiper did [Kui89]. A production $p = [X_0, X_1, X_2 \ldots, X_s] \in P$, $s \geq 0$, will be denoted either as $p : X_0 \longrightarrow X_1\ X_2 \ldots X_s$ or as $X_0 = \mathbf{p}(X_1, X_2, \ldots, X_s)$. The former denotation stresses the rewriting characteristics whereas the latter stresses the analogy with terms. Note that symbol p is overloaded. It is used to denote the entire production "$p \in P$" as well as its name "$p : \ldots$"; its bold version is reserved for the corresponding term constructor

"$\mathbf{p}(\ldots)$". Non-terminal X_0 is called the left-hand side non-terminal or *parent* of p and X_1, $X_2, \ldots$, X_s are the right-hand side non-terminals or *children* of p.

List selection allows us to refer to the non-terminals of a production: $p{\cdot}i$ refers to X_i. Furthermore, we use the unary operator $\mathbf{s}$ "size" to denote the number of children of a production $p \in P$: $\mathbf{s}p = \mathit{len}\ p - 1$. Note that $\mathbf{s}p \geq 0$ by definition of P. In other words, we have $\forall_{0 \leq i \leq \mathbf{s}p} p{\cdot}i = X_i$. We say that p is a production *on* X if $p{\cdot}0 = X$.

A production p is a *terminal production* if it has no (non-terminal) children: $\mathbf{s}p = 0$. In the concrete syntax such a production has only terminal symbols on the right-hand side. A context-free grammar can only derive strings of terminals only, if terminal productions exist.

A tuple (p, i), for production p and $0 \leq i \leq \mathbf{s}p$, denotes an occurrence, on position i, of non-terminal $p{\cdot}i$ in production p. We use a shorter notation for non-terminal occurrences, based on the binary infix operator $\mathbf{o}$ "occurrence". It is defined as $p\mathbf{o}i = (p, i)$, and has the highest precedence of all operators used in this thesis.

The '$\mathbf{o}$' is a big '$\cdot$'.

A *primitive value* or *pseudo terminal* is a non-terminal with implicit productions. Examples of primitive values are booleans, integers and strings. Primitive values abbreviate grammar specifications. Note that non-terminals and pseudo terminals induce a set of values (terms); they will therefore also be regarded as *types*.

A structure tree or abstract syntax tree of a context-free grammar is a finite ordered tree whose nodes are labeled with *productions* (abstract syntax trees used to be referred to as parse trees). For every node K of a structure tree $\mathit{prod}(K)$ is the production that labels K and $\mathit{nont}(K)$ is the left-hand side non-terminal of $\mathit{prod}(K)$. With $K{\cdot}i$ we denote the ith child of node K. Structure trees must be such that each node K has $\mathbf{s}\mathit{prod}(K)$ children, and $\mathit{nont}(K{\cdot}i) = \mathit{prod}(K){\cdot}i$. For a structure tree T with root K we define $\mathit{prod}(T)$, $\mathit{nont}(T)$ and $T{\cdot}i$ to mean respectively $\mathit{prod}(K)$, $\mathit{nont}(K)$ and the subtree rooted at $K{\cdot}i$. Furthermore, $\mathit{nont}(T)$ is known as the *root* or *type* of T.

A tree is *ordered* if, for every node, a linear order is defined on the children of that node [WG84].

The root of a structure tree does not have to be the start symbol.

An *instance* of a production p is a node in a structure tree labeled with p. To denote a node in a structure tree, we use a list of natural numbers that describes the path from the root of the tree to that node. The root is denoted by $[\,]$ and the ith child of a node denoted by l is denoted by $i : l$. From now on nodes are identified by lists of natural numbers.

The list denotation for a node is reversed compared to a UNIX path denotation.

An *instance* of a non-terminal X is a node K in a structure tree such that $\mathit{nont}(K) = X$. Note that node K will not only be used to refer to the production instance of $\mathit{prod}(K)$ but also to refer to the non-terminal instance of $\mathit{nont}(K)$.

2.2.2 Attribute grammars

An attribute grammar, sometimes referred to as a *plain* attribute grammar as opposed to a higher-order attribute grammar, is based on a context-free grammar which is augmented with attributes and attribute equations. Each attribute describes a property of an abstract syntax tree. The value of an attribute is defined by a composition of several equations in a context dependent manner.

Definition 2 *Attribute grammar.*
An attribute grammar is a triple $AG = (G, A, E)$. G is a context-free grammar (V, P, S). A is a finite set of attributes, partitioned into sets $A_{nont}(X)$ and $A_{loc}(p)$ for each $X \in N$ and $p \in P$. The sets $A_{nont}(X)$ are further partitioned into sets $A_{inh}(X)$ and $A_{syn}(X)$. E is a set of attribute equations, partitioned into sets $E(p)$ for each production $p \in P$.
□

A and E together are known as the attribution rules of an attribute grammar. Each attribute in an attribute grammar is associated with either a symbol $X \in N$ or a production $p \in P$. The attributes in $A_{loc}(p)$, $p \in P$ are known as *local* attributes, they are denoted by $p.l$, $p.k$,... The set $A_{nont}(X)$, which will be abbreviated to $A(X)$, consists of all attributes associated with non-terminal X. Its elements are denoted by $X.a$, $X.b$,... An attribute $X.a$ is either *inherited*, if $X.a \in A_{inh}(X)$, or *synthesized* if $X.a \in A_{syn}(X)$. It is convenient to let list selection bind stronger than the "attribute dot": with $p{\cdot}i.a$ we mean $(p{\cdot}i).a$.

Do not confuse $p.l$ (local attribute l of p) and $p{\cdot}i$ (the ith non-terminal of p).

For each occurrence $p\mathbf{o}i$ of non-terminal X, there is an attribute occurrence associated with production p, denoted by $p\mathbf{o}i.a$ for all attributes $X.a \in A(X)$. $O(p\mathbf{o}i)$ is the set of all attribute occurrences of $p\mathbf{o}i$. It is formally defined as $O(p\mathbf{o}i) = \{p\mathbf{o}i.a \mid p{\cdot}i.a \in A(p{\cdot}i)\}$. $O_{nont}(p)$ is the set of all the attribute occurrences associated with the non-terminal occurrences of p: $O_{nont}(p) = \bigcup_{0 \le i \le \mathbf{sp}} O(p\mathbf{o}i)$. Local attributes of p also induce attribute occurrences for p: $O_{loc}(p) = A_{loc}(p)$. $O(p)$ is the set of all attribute occurrences associated with production p: $O(p) = O_{loc}(p) \cup O_{nont}(p)$. Note that operator O is overloaded: it operates not only on non-terminal occurrences, but also on productions. The elements of $O(p)$ will usually be denoted with Greek letters from the beginning of the alphabet α, β,...

$O(\cdot)$ is for attribute occurrences and $\mathcal{O}(\cdot)$ for orders.

$$\begin{tikzcd}
X \arrow[r, leftrightarrow, "P"] \arrow[d, "A(\cdot)"'] & p\mathbf{o}i \arrow[d, "O(\cdot)"] \\
X.a \arrow[r, leftrightarrow, "P"'] & p\mathbf{o}i.a
\end{tikzcd}$$

The partitions $A_{inh}(X)$ and $A_{syn}(X)$ of the attributes of a non-terminal X induce the partitions $O_{inh}(p\mathbf{o}i)$ and $O_{syn}(p\mathbf{o}i)$ of the attribute occurrences $O(p\mathbf{o}i)$ of occurrence $p\mathbf{o}i$ of X. Hence, the non-local attribute occurrences of $O(p)$ can be partitioned in inherited and synthesized attribute occurrences. However, a more useful partition is into *input* and *output* occurrences. The input attributes occurrences are the inherited attribute occurrences of the parent and the synthesized attribute occurrences of the children. The set of output attribute occurrences consist of the complement with respect to the non-local attribute occurrences: the synthesized attribute occurrences of the parent and the inherited attribute occurrences of the children. Formally, the following three sets partition $O(p)$:

$O_{inp}(p) \cup O_{out}(p) = O_{nont}(p)$

$$\begin{aligned}
O_{inp}(p) &= O_{inh}(p\mathbf{o}0) \cup \textstyle\bigcup_{1 \le i \le \mathbf{sp}} O_{syn}(p\mathbf{o}i) \quad , \\
O_{out}(p) &= O_{syn}(p\mathbf{o}0) \cup \textstyle\bigcup_{1 \le i \le \mathbf{sp}} O_{inh}(p\mathbf{o}i) \quad , \\
O_{loc}(p) &= A_{loc}(p) \quad .
\end{aligned}$$

$E(p)$ is the set of attribute equations associated with production p. Each equation defines the value of an attribute occurrence in $O(p)$ in terms of other attribute occurrences in $O(p)$. An equation will be denoted by $(\alpha := f \ldots \beta \ldots)$. In this equation, f is the name of a *semantic function* and α and β are attribute occurrences of p. Attribute occurrence α is said to *depend* on β. $E(p)$ defines a dependency set $dpr(p)$ "dependencies in a production" on $O(p)$. There is an arc $\beta \rightarrow \alpha$ if α depends on β, $\alpha, \beta \in O(p)$. Formally $dpr(p) = \{\beta \rightarrow \alpha \mid (\alpha := f \ldots \beta \ldots) \in E(p)\}$.

Each attribute equation describes a dependency of one attribute occurrence on zero or more others. The set $O_{def}(p)$ denotes all attribute occurrences of production p for which there is a defining equation, that is to say $O_{def}(p) = \{\alpha \mid (\alpha := f \ldots \beta \ldots) \in E(p)\}$. Likewise, $O_{use}(p)$ denotes the set of used attribute occurrences. It is defined as $O_{use}(p) = \{\beta \mid (\alpha := f \ldots \beta \ldots) \in E(p)\}$.

An attribute grammar is in *Bochmann normal form* [Boc76] or *normalized* if equations do not *use* output attribute occurrences: $O_{use}(p) \cap O_{out}(p) = \{\}$. Every attribute grammar can be converted to Bochmann normal form by means of a simple transformation of the semantic functions provided the dpr sets are cycle free [Boc76].

Non-terminal occurrences may occur on the right-hand side of an attribute equation: $(\alpha := f \ldots p\mathbf{o}i \ldots) \in E(p)$. This is known as a *syntactic reference*. Syntactic elements can be regarded as constants because the abstract syntax tree is not subject to change during attribute evaluation. Primitive values in the abstract syntax tree can only be used via a syntactic reference.

Syntactic references can be avoided by defining a context-free grammar for primitive values and a $self$ attribute for each non-terminal.

The following two conditions guarantee that an attribute grammar description "makes sense". This is commonly known as the *completeness* property [WG84, Kui89, Vog93]. The first condition, $O_{def}(p) = O_{loc}(p) \cup O_{out}(p)$, guarantees that every local and every output attribute of p has at least one defining equation. Secondly we require that no two equations have the same target: $|O_{def}(p)| = |E(p)|$.

Definition 3 *Complete attribute grammar.*
An attribute grammar is complete, if for each production p the following two conditions hold:

- $O_{def}(p) = O_{loc}(p) \cup O_{out}(p)$,
- $|O_{def}(p)| = |E(p)|$.

□

Completeness alone does not guarantee that all attributes of an abstract syntax tree are effectively computable: circular dependencies may occur. If they do not occur for any derivable tree, the grammar is called *well-formed* [DJL88] or *well-defined* [Knu68, WG84].

Definition 4 *Well-defined attribute grammar.*
An attribute grammar is well-defined, if for each abstract syntax tree of the grammar all attribute instances are effectively computable.
□

The specifications of semantic functions and context conditions are absent from the given definition of attribute grammars. Only the names of the semantic functions occur in the attribute equations. Since our emphasis is on the analysis of dependencies between attribute (occurrences) the precise nature of semantic functions and conditions is irrelevant.

2.2.3 Higher-order attribute grammars

Higher-order attribute grammars are based on plain attribute grammars which are augmented with higher-order attributes. Higher-order attributes are attributes whose value is a tree *with which we associate attributes again*. Attributes of these so-called *higher-order trees*, may be higher-order attributes again.

Operationally, we associate the following behavior with higher-order attributes. Assignments to tree-valued attributes always assign an *undecorated* tree. In particular, this holds for higher-order attributes. After a higher-order attribute is *instantiated*, the inherited attributes of the root of the instantiated tree are assigned a value in the "normal" way. The synthesized attributes of the instantiated tree are obtained by decorating it. When decoration is finished, the synthesized attributes are ready for further "normal" processing.

Two kinds of higher-order attributes can be distinguished, namely *non-terminal attributes* and *attributable attributes*. Non-terminal attributes were introduced by Vogt [VSK89]. He proposed to interpret them as gaps in the abstract syntax tree that are filled by attribute evaluation. This computation model is very powerful but hard to implement efficiently and elegantly. Vogt proposed an evaluator for higher-order attribute grammars [VSK91] that uses *attributable attributes* instead. Attributable attributes offer the same expressive power as non-terminal attributes, but they force the grammar writer into a harness that allows for more elegant evaluators. Vogt probably was not aware of the difference between attributable and non-terminal attributes.

We first discuss attributable attributes, since that is how we model higher-order attributes. Next, we illustrate why we prefer attributable attributes to non-terminal attributes.

Attributable attributes

An attributable attribute $p.x$ is an attribute associated with a production p, much like a local attribute. An equation of that production should assign a tree t to $p.x$. The root non-terminal of t is fixed, or in other words, the *type* of $p.x$ is determined by the grammar specification. The inherited attributes of t should also be given a value so that its synthesized attributes may be used in other equations. The inherited and synthesized attributes of t are known as the *generated attributes* of $p.x$.

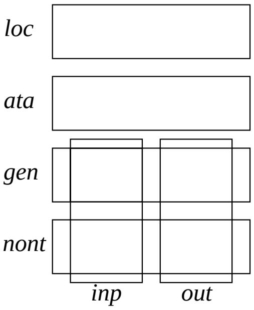
Definition 5 *Higher-order attribute grammar.*
A higher-order attribute grammar HAG is a quadruple $HAG = (G, A, E, R)$. G is a context-free grammar (V, P, S). A is a finite set of attributes, partitioned into sets $A_{nont}(X)$, $A_{loc}(p)$ and $A_{ata}(p)$ for $X \in N$ and $p \in P$. The sets $A_{nont}(X)$ are further partitioned into sets $A_{inh}(X)$ and $A_{syn}(X)$. E is a set of attribute equations, partitioned into sets $E(p)$ for each production $p \in P$. R is a partial function from attributes to non-terminals.
□

We distinguish three kinds of attributes in higher-order attribute grammars, each of which induces attribute occurrences in one or more productions.

The set $A_{nont}(X)$, which will be abbreviated to $A(X)$, consists of all attributes associated with non-terminal X. Its elements are denoted by $X.a$, $X.b$, ... An attribute $X.a$ is either *inherited*, if $X.a \in A_{inh}(X)$, or *synthesized* if $X.a \in A_{syn}(X)$. Each occurrence $p\mathbf{o}i$ of X induces the attribute occurrences $O_{inh}(p\mathbf{o}i)$, $O_{syn}(p\mathbf{o}i)$ and $O(p\mathbf{o}i)$ defined as $O_{inh}(p\mathbf{o}i) = \{p\mathbf{o}i.a \mid p{\cdot}i.a \in A_{inh}(p{\cdot}i)\}$ respectively $O_{syn}(p\mathbf{o}i) = \{p\mathbf{o}i.a \mid p{\cdot}i.a \in A_{syn}(p{\cdot}i)\}$ and $O(p\mathbf{o}i) = O_{inh}(p\mathbf{o}i) \cup O_{syn}(p\mathbf{o}i)$. Furthermore we define $O_{nont}(p) = \bigcup_{0 \le i \le \mathbf{sp}} O(p\mathbf{o}i)$.

Secondly, the attributes in $A_{loc}(p)$ are known as *local* attributes of $p \in P$, they are denoted by $p.l$, $p.k$, ... Each local attribute induces precisely one attribute occurrence: $O_{loc}(p) = A_{loc}(p)$.

Thirdly, $A_{ata}(p)$ is the set of *attributable attributes* associated to production $p \in P$. They are denoted by $p.x$, $p.y$, ... Each attributable attribute induces an attribute occurrence: $O_{ata}(p) = A_{ata}(p)$. The partial function R maps an attributable attribute $p.x$ to a non-terminal X: $X = R\ p.x$. Non-terminal X is the *root* non-terminal of the trees that may be instantiated at $p.x$; X is the *type* of $p.x$. Each attribute $X.a$ of X induces a *generated* attribute occurrence denoted by $p.x.a$. Operator O will be overloaded once more; it not only operates on non-terminal occurrences and productions, but also on attributable attributes. The sets $O_{inh}(p.x)$, $O_{syn}(p.x)$ and $O(p.x)$ are defined as $O_{inh}(p.x) = \{p.x.a \mid X = R\ p.x \wedge X.a \in A_{inh}(X)\}$ respectively $O_{syn}(p.x) = \{p.x.a \mid X = R\ p.x \wedge X.a \in A_{syn}(X)\}$ and $O(p.x) = O_{inh}(p.x) \cup O_{syn}(p.x)$. The generated attribute occurrences associated with a production are defined as $O_{gen}(p) = \bigcup_{p.x \in O_{ata}(p)} O(p.x)$.



For higher-order attribute grammars the set $O(p)$ is partitioned into four sets: $O_{nont}(p)$, $O_{loc}(p)$, $O_{ata}(p)$ and $O_{gen}(p)$. The attribute occurrences $O_{nont}(p)$ and $O_{gen}(p)$ can be partitioned into *input* and *output* occurrences as we did for plain attribute grammars:

$$\begin{aligned} O_{inp}(p) &= O_{inh}(p\mathbf{o}0) \cup \textstyle\bigcup_{1 \le i \le \mathbf{sp}} O_{syn}(p\mathbf{o}i) \cup \bigcup_{p.x \in O_{ata}(p)} O_{syn}(p.x) \quad , \\ O_{out}(p) &= O_{syn}(p\mathbf{o}0) \cup \textstyle\bigcup_{1 \le i \le \mathbf{sp}} O_{inh}(p\mathbf{o}i) \cup \bigcup_{p.x \in O_{ata}(p)} O_{inh}(p.x) \quad . \end{aligned}$$

We have now defined similar classifications for attribute occurrences of higher-order attribute grammars and plain attribute grammars. Therefore, the notions of semantic functions, $dpr(p)$, $O_{def}(p)$, $O_{use}(p)$ and Bochmann normal form, are also similar. We will not repeat them.

A higher-order attribute grammar is *complete* if every non-input attribute occurrence is defined $O_{def}(p) = O_{out}(p) \cup O_{loc}(p) \cup O_{ata}(p)$ and if no two equations have the same target: $|O_{def}(p)| = |E(p)|$. Furthermore, the semantic function for attributable attributes must compute trees with the right root non-terminal.

Definition 6 *Complete higher-order attribute grammar.*
A higher-order attribute grammar is complete, if for each production p the following three conditions hold:

- $O_{def}(p) = O_{out}(p) \cup O_{loc}(p) \cup O_{ata}(p) \quad ,$
- $|O_{def}(p)| = |E(p)|$ and
- for every $(p.x := f \ldots) \in E(p)$, the type of f must match $R\ p.x$.

□

Completeness alone does not guarantee that all attributes of an abstract syntax tree are effectively computable: circular dependencies may occur. If they do not occur for any derivable tree with appropriate instances of its attributable attributes, the grammar is called *well-defined* [Vog93].

Definition 7 *Well-defined higher-order attribute grammar.*
A higher-order attribute grammar is well-defined, if for each abstract syntax tree of the grammar all attribute instances are effectively computable.
□

Non-terminal attributes

The name "non-terminal attribute" stems from the fact that an occurrence of a non-terminal is computed by attribute evaluation as if it were an occurrence of an attribute. Vogt [VSK89] uses the following notation for a production to show that non-terminal occurrence $p\mathbf{o}i$ is a non-terminal attribute

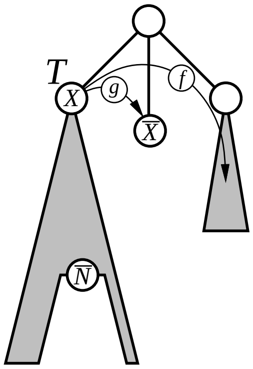
$$p : X_0 \longrightarrow X_1 \ldots \overline{X_i} \ldots X_s \quad .$$

Such a declaration does require an attribute equation $(p\mathbf{o}i := f \ldots)$ to be part of $E(p)$. Informally $\overline{p\mathbf{o}i}$ is a gap which is to be instantiated during attribute evaluation. Thus, for *constructing* a tree with root $\mathbf{p}$, no subtree for position i should be supplied, one must reserve a "joker" ($?$) for that purpose:

$$T_0 := \mathbf{p}(T_1, \ldots, T_{i-1}, ?, T_{i+1}, \ldots, T_s) \quad .$$

The *core* of a tree is the *unexpanded* tree; the tree without the jokers being instantiated by attribute evaluation. Instantiated non-terminal attributes really belong to the abstract syntax tree. Thus, when *destructing* (inspecting) a tree, the non-terminal attributes may be referenced. The value of $T_0{\cdot}i$ in the above example depends on the context of T_0, more specifically on the inherited attributes of T_0. They determine what will be instantiated at the gap.

Syntactic references do not mingle well with non-terminal attributes. We will shortly discuss two peculiarities.



Consider a subtree T rooted X with an instance $\overline{N}$ of a non-terminal attribute. Assume that T is referenced (a syntactic reference) by a semantic function g. It simply copies T into a non-terminal attribute $\overline{X}$ (thus $g = id$). Since the values of inherited attributes of $\overline{X}$ are possibly different from the ones of T the gap in the tree of $\overline{X}$ will probably be filled with a different instantiation of $\overline{N}$. Hence g copies only the core of T into $\overline{X}$. This gives rise to a peculiar situation: $\overline{X}$ is *assigned* the value of T, but is does not equal T.

To complicate the example, consider another semantic function f that also refers to T. Note that a "copy" of T can be passed around in the abstract syntax tree from attribute instance to attribute instance (where the semantic functions may add or delete some nodes). In this case, the core of T does *not* suffice since the formalism allows *inspection of the copied T which might include the instantiated non-terminal attribute*. If the result of f—and probably a chain of other semantic functions—happens to be assigned to a non-terminal attribute somewhere, it must still be *stripped*, leaving only the core. Note that if syntactic references occur, one must make sure that a tree is only referenced after all non-terminal attributes have been instantiated. This puts a heavy burden on the attribute evaluator.

· · ·

One of the applications of higher-order attributes in a grammar is to replace an inductive semantic function. Alternatively, higher-order attributes are used to decorate a subtree that is first massaged into a more easily decoratable tree. In such cases the higher-order attribute is not supposed to be visible to the outside world. Let alone that the outside world should be bothered by spurious jokers in constructor calls. Observe further that a sort-of $self$ attribute can be introduced by the grammar writer that computes an image of a tree—including possibly some higher-order attributes—in a way as the grammar writer wants the tree to be seen by the outside world. Therefore, syntactic references to expanded abstract syntax trees can be realized without non-terminal attributes.

Attributable attributes are absent from the context-free grammar, thus jokers are no longer needed in constructor calls. Attributes in general and specifically attributable attributes are not available while destructing an abstract syntax tree. Therefore, we need not worry about problems caused by syntactic references either. The notion of abstract syntax trees need not be refined. In this thesis, we model higher-order attributes with *attributable attributes*.

2.3 Grammar specifications

The previous section presented a mathematical definition of attribute grammars, suitable for analysis. In this section we will describe two formalisms to *write down* grammar specifications. The first *algebraic* formalism resembles SSL, the Synthesizer Specification Language, used by the Synthesizer Generator [RT88]. The second formalism is a *graphical* one. It has the same expressive power as the algebraic one except that the semantic functions are usually not specified. The former is better suited for computer processing; the latter gives more insight in dependencies.

This section also presents a running example, the VARUSE grammar. A small abstract syntax tree will accompany the VARUSE grammar. It will be referred to as the VARUSE application.

2.3.1 Algebraic formalism

The algebraic formalism will be presented by example. We will use a small grammar, the VARUSE grammar [PSV92a]. The VARUSE grammar defines a simple programming language that captures the essence of declaring and applied occurrences of variables.

A program is a list of "declarations" (such as `var x`) and "statements" (like `use x`). They may be mixed freely, and we do not require *definition before use*. In the translation process, each variable is mapped onto a number; its "address". Addresses are assigned in an order corresponding with the declarations. The outcome of the translation will be a list which contains the address for each applied occurrence, and the negation of the address for the defining occurrence. Hence

```
(use x;use y;var y;use x;use y;use z;var z;var x;use x;)
```

is mapped onto the list

```
3, 1, -1, 3, 1, 2, -2, -3, 3  .
```

We will now formally define an attribute grammar that describes this mapping. We start with a context-free grammar. Unlike the mathematical definition, we do not list the non-terminals separately; we only list the productions. Productions are labeled for future reference. Non-terminal S is the start symbol.

$$
\begin{array}{ll}
\mathit{root}: & S \longrightarrow \text{'('}\ L\ \text{')'} \\
\mathit{decl}: & L \longrightarrow \text{'}\texttt{var}\text{'}\ N\ \text{';'}\ L \\
\mathit{stat}: & L \longrightarrow \text{'}\texttt{use}\text{'}\ N\ \text{';'}\ L \\
\mathit{empty}: & L \longrightarrow \\
\mathit{name}: & N \longrightarrow \mathit{str}
\end{array}
$$

The non-terminal str is a pseudo terminal. This means that it is a non-terminal for which the productions are implicit; in this case the productions produce strings that we use as identifiers in our programming language.

Concrete syntax is for parsing; abstract syntax for attribute evaluation.

Concrete syntax is not of interest to us; for grammar analysis we focus on the abstract syntax. So, before we extend the grammar with attribution rules, we "extract" the abstract syntax by leaving out the terminals ('(', ')', '`var`', '`use`', and '`;`'). Furthermore, we prefer to cast the definition in a form that stresses the term-like nature of abstract syntax trees.

$$\begin{aligned} S &= \mathbf{root}(L) \\ L &= \mathbf{decl}(N, L) \\ &\mid\ \mathbf{stat}(N, L) \\ &\mid\ \mathbf{empty}() \\ N &= \mathbf{name}(str) \end{aligned}$$

The grammar formalism only deals with attribute dependencies, not with semantic functions, let alone types. In a grammar specification, types help understanding.

Now that the context-free grammar for the abstract syntax is defined, we focus on the attribution rules: the attributes and the equations. Firstly, we define the attributes associated with the non-terminals. Attributes have a type. Apart from the primitive value str, the type num representing natural numbers is assumed to exist. We define two new types: Env represents symbol tables (lists of defined identifiers) and $Code$ represents the final output.

$$\begin{aligned} &\mathbf{type}\ Env\ \ = [str] \\ &\mathbf{type}\ Code = [num] \end{aligned}$$

The start symbol S has a single (synthesized) attribute returning the code. List symbol L has three attributes: one that collects the declarations (synthesized), one that distributes the environment (inherited) and one that synthesizes the code. A name N only synthesizes the underlying identifier. The following declarations define the attributes for each non-terminal. Synthesized attributes are prefixed by "↑", inherited attributes by "↓".

$$\begin{aligned} &S{<}{\uparrow}code : Code{>} \\ &L{<}{\uparrow}decs : Env,\ {\downarrow}env : Env,\ {\uparrow}code : Code{>} \\ &N{<}{\uparrow}id : str{>} \end{aligned}$$

We use denotations like $p\mathbf{o}i$ when discussing grammars in general. For an grammar instance, like VARUSE, we use denotations like L_2.

The second part of the attribution rules defines the relationships between the attributes of various occurrences of non-terminals. For example, in a $decl$ production, the synthesized symbol table consists of the identifier declared by the N child consed to the symbol table of the L child. We will not use the $\mathbf{o}$ operator in grammar specifications: denotation $decl\mathbf{o}0$ shows no relation with non-terminal L. Instead, the two L occurrences in a $decl$ production are subscripted to distinguish them: $decl\mathbf{o}0$ is denoted with L_1 and $decl\mathbf{o}2$ with L_2. Since $decl\mathbf{o}1$ is the only occurrence of N in $decl$, we simply denote it by N. The afore mentioned relationship is thus described by the following equation

$$L_1.decs := N.id : L_2.decs \quad .$$

```
root S

S<↑code : Code>

S = root(L)
      L.env  := L.decs;
      S.code := L.code
```

```
type Env  = [str]
type Code = [num]

L<↑decs : Env, ↓env : Env, ↑code : Code>

L1 = decl(N, L2)
      L1.decs := N.id : L2.decs;
      L2.env  := L1.env;
      L1.code := (−lookup L1.env N.id) : L2.code
   |  stat(N, L2)
      L1.decs := L2.decs;
      L2.env  := L1.env;
      L1.code := (+lookup L1.env N.id) : L2.code
   |  empty()
      L1.decs := [ ];
      L1.code := [ ]
```

```
upstring : : str ⟶ str
lookup : : Env ⟶ str ⟶ num

N<↑id : str>

N = name(str)
      N.id := upstring str


func lookup (s : ss) i =
   if s = i
   then 1
   else  1 + lookup ss i
   fi
func lookup [ ] i =
   ⊥
```

Figure 2.1. The VARUSE grammar

```
root S

S<↑code : Code>

S = root(L)
      L.env  := L.decs;
      S.code := L.code
```

```
type Code = [num]

L<↑decs : E, ↓env : E, ↑code : Code>

L1 = decl(N, L2)
      L1.decs  := add(N.id, L2.decs);
      L2.env   := L1.env;
      ata table : E;
      table    := L1.env;
      table.id := N.id;
      L1.code  := −table.addr : L2.code
   |  stat(N, L2)
      L1.decs  := L2.decs;
      L2.env   := L1.env;
      ata table : E;
      table    := L1.env;
      table.id := N.id;
      L1.code  := +table.addr : L2.code
   |  empty()
      L1.decs  := none();
      L1.code  := [ ]
```

```
upstring : : str ⟶ str

N<↑id : str>

N = name(str)
      N.id := upstring str


E<↓id : str, ↑addr : num>

E1 = add(str, E2)
      E2.id   := E1.id;
      E1.addr := if str = E1.id
                 then 1
                 else  1 + E2.addr
                 fi
   |  none()
      E1.addr := ⊥
```

Figure 2.2. A higher-order variant for VARUSE

This equation uses the semantic function cons : , which is assumed to be built-in. To find an identifier in a symbol table, a *lookup* function must be written. For this reason, grammar specification languages include a functional language in which semantic functions can be specified. We will not define one, but merely use a standard functional programming language. Figure 2.1◁ presents the complete VARUSE grammar. The specification includes the *lookup* function which is a simple recursive function that returns a polymorphic "undefined" value ($\perp$) when the lookup fails (the given identifier does not occur in the symbol table). The semantic function *upstring* may be regarded as a standard function on the standard primitive type *str*. It converts all lowercase letters in a string to uppercase. The application of *upstring* in the VARUSE grammar makes the language case insensitive.

upstring 'Idx'
= 'IDX'

Note that about half the equations are so-called *copy-rules*: the semantic function is the identity function *id*. Copy-rules pass information around in abstract syntax trees. They make grammar descriptions long winded and generated evaluators inefficient.

Figure 2.2◁ lists an equivalent *higher-order* attribute grammar for the VARUSE language. An additional non-terminal E is defined. Type E is isomorphic with Env: **add** corresponds with : and **none** corresponds with [ ]. It is common in attribute grammars that abstract syntax is defined to describe data types. In this example, attribution rules are given for E which replace the *lookup* function. This allows us to use in production *stat* an attributable attribute $stat.table$ of type E to lookup the address of $N.id$. Note the analogy between function *lookup* in Figure 2.1◁ and the equations for E in Figure 2.2◁. This is an example where a higher-order attribute replaces a recursive function.

The next chapter (Section 3.2) introduces the BOX grammar. It will feature *local* attributes, *factorization of equations* common to several productions and *modularization*.

2.3.2 Graphical formalism

Attribute grammars are most often presented in a linear representation as the one just given. However, they could also be presented in a graphical way. Such pictures provide more insight in the attribute dependencies while preserving the formal semantics.

A graphical specification of a (higher-order) attribute grammar consists of a drawing for each production. A graphic describing a single production is known as a *production icon*. A specification in the graphical formalism lists the production icons.

A production is rendered by a *large box*. The name of the production is given in a "flag" emanating from the left border. Figure 2.3 shows a "general" production $p : X_0 \longrightarrow X_1 X_2 X_3 \ldots X_s$.

Non-terminals are rendered by labeled *discs*. The left-hand side non-terminal (Fig 2.3: X_0) is drawn at the top of the production box, the right-hand side non-terminals are drawn at the bottom of the production box in the order described by

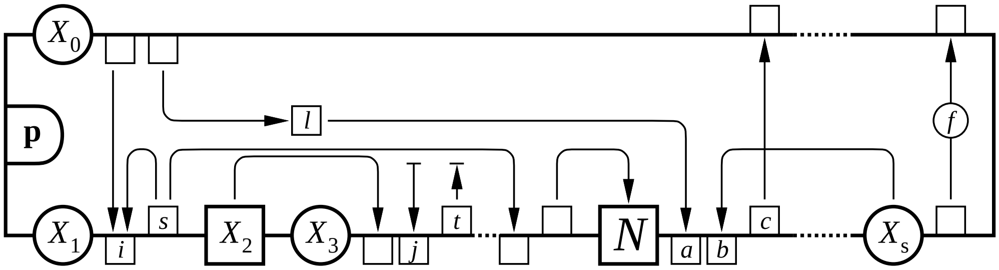


Figure 2.3. A production icon

the production.

Each non-terminal disc is followed by *small boxes* denoting the attributes. The boxes are labeled with names of the attributes—i instead of $X.i$. Inherited attributes (Fig 2.3: i) are lowered, synthesized attributes (Fig 2.3: s) raised. Attributes are usually ordered in such a way that the overall picture comes out nicely. Since the discs actually correspond to non-terminal occurrences, the small boxes correspond to attribute occurrences.

Output attribute occurrences “stick out” of a production icon; input occurrences “point in”.

Local attributes will also be rendered by small boxes labeled with the name of the attribute—l instead of $p.l$. Local attributes will roughly be placed in the center of the production box (Fig 2.3: l). Attributable attributes are drawn in a similar fashion as right-hand side non-terminals except that they will be rendered by a *box* instead of a disc. The box is labeled with the type of the attributable attribute. Attributable attributes are interspersed somewhere at the bottom of the production (Fig 2.3: N). The generated attributes associated to them (Fig 2.3: a, b and c) are drawn as if they were plain attributes. Pseudo terminals are also drawn as boxes at the bottom of the production (Fig 2.3: X_2).

The *arrows* describe the dependencies on the attribute occurrences. In other words they form the $dpr(p)$ graph. Arrows may emanate from syntactic elements as non-terminals (Fig 2.3: X_s) or pseudo terminals (Fig 2.3: X_2). The arrow then represents a syntactic reference. Arrows may be labeled with a *small disc* showing the semantic function associated with the arrow (Fig 2.3: f). But since we are mostly interested in dependencies, we usually leave out the labels. Sources in the $dpr(p)$ graph—attribute occurrences with a defining semantic function that is constant—are decorated with an in-arrow (Fig 2.3: j). For cosmetic reasons, sinks are supplied with an out-arrow (Fig 2.3: t).

The varuse grammar, algebraically specified in Figure 2.1$_\triangleleft$, is presented graphically in Figure 2.4$_\triangleright$. The higher-order variant, specified in Figure 2.2$_\triangleleft$, is presented graphically in Figure 2.5$_\triangleright$.

2.3.3 Discussing the varuse grammar

We will review the characteristics of the varuse grammar. First we discuss the plain attribute grammar, next we deal with the higher order variant.

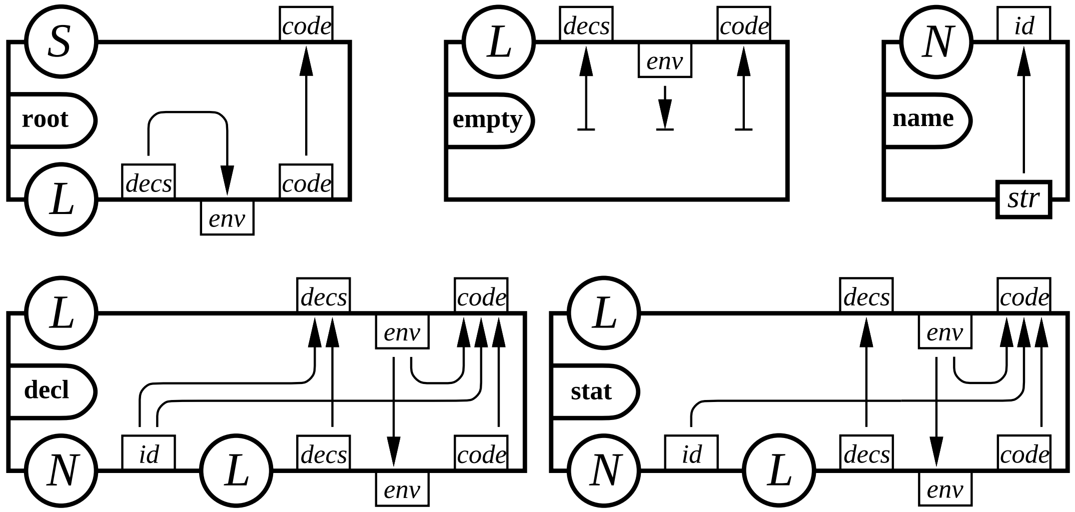


Figure 2.4. The production icons of the VARUSE grammar

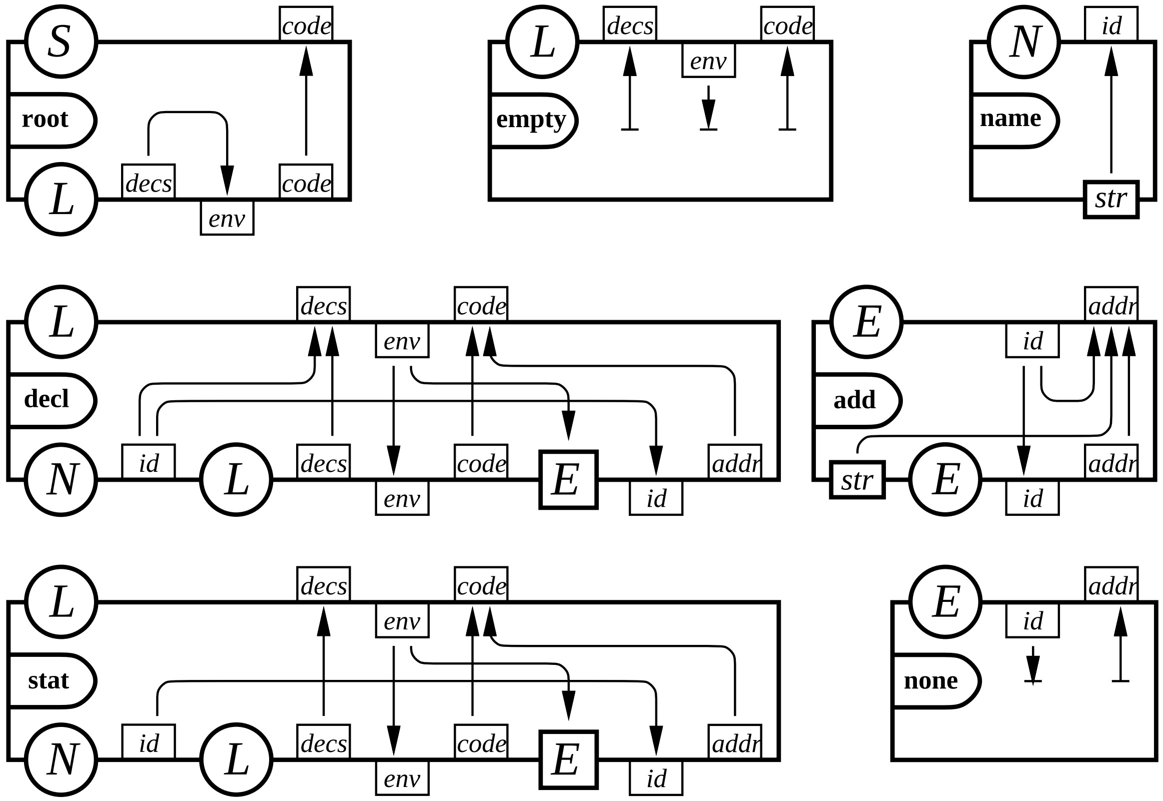


Figure 2.5. The production icons for higher-order variant of the VARUSE grammar

Production $decl : L \longrightarrow N\ L$ has size 2 $(\mathbf{s}decl = 2)$ which means that it has two children: $decl{\cdot}1 = N$ and $decl{\cdot}2 = L$. The parent is denoted with $decl{\cdot}0 = L$.

Non-terminal N has one attribute, namely $N.id$. Non-terminal L has three attributes: $A(L) = \{L.decs, L.env, L.code\}$, one inherited $A_{inh}(L) = \{L.env\}$ and two synthesized $A_{syn}(L) = \{L.decs, L.code\}$.

Consequently, occurrence $decl\mathbf{o}0$ of non-terminal L has three attribute occurrences namely $O(decl\mathbf{o}0) = \{decl\mathbf{o}0.decs, decl\mathbf{o}0.env, decl\mathbf{o}0.code\}$. One attribute occurrence is inherited $O_{inh}(decl\mathbf{o}0) = \{decl\mathbf{o}0.env\}$ and two are synthesized $O_{syn}(decl\mathbf{o}0) = \{decl\mathbf{o}0.decs, decl\mathbf{o}0.code\}$.

Similar inductions leads to the following partitions of the set $O_{nont}(decl)$

$$\begin{aligned} O(decl\mathbf{o}0) &= \{decl\mathbf{o}0.decs, decl\mathbf{o}0.env, decl\mathbf{o}0.code\} \quad , \\ O(decl\mathbf{o}1) &= \{decl\mathbf{o}1.id\} \quad \text{and} \\ O(decl\mathbf{o}2) &= \{decl\mathbf{o}2.decs, decl\mathbf{o}2.env, decl\mathbf{o}2.code\} \quad . \end{aligned}$$

Production $decl$ has no local attributes. The set of attribute occurrences $O(decl)$ thus equals $O_{nont}(decl)$. Non-terminal occurrences are usually denoted with the non-terminal subscripted with an index to distinguish multiple occurrences in a production. In other words, we write

$$O(decl) = \{L_1.decs, L_1.env, L_1.code,\ N.id,\ L_2.decs, L_2.env, L_2.code\} \quad .$$

The set of attribute occurrences of a production can either be partitioned into local attribute occurrences and attribute occurrences induced by non-terminal occurrences, or it can be partitioned in local, input and output attribute occurrences.

$$\begin{aligned} O_{loc}(decl) &= \{\} \\ O_{inp}(decl) &= \{L_1.env, N.id, L_2.decs, L_2.code\} \\ O_{out}(decl) &= \{L_1.decs, L_1.code, L_2.env\} \end{aligned}$$

. . .

Let us turn to the higher-order variant of the VARUSE grammar. It features two equivalent attributable attributes: $decl.table$ and $stat.table$, both of type E. Non-terminal E has two attributes: $A_{inh}(E) = \{E.id\}$ and $A_{syn}(E) = \{E.addr\}$. Consequently, $decl.table$ generates two attribute occurrences $O_{inh}(decl.table) = \{decl.table.id\}$ and $O_{syn}(decl.table) = \{decl.table.addr\}$ and thus we have

$$O(decl.table) = \{decl.table.id, decl.table.addr\} \quad .$$

Attributable attributes and their generated attribute occurrences (and local attribute occurrences for that matter) are usually denoted without the production name, if it is clear from the context which production is meant. In other words, in context $decl$ we usually write $O(table) = \{table.id, table.addr\}$.

Attribute $table$ is the only attributable attribute associated with production $decl$: $O_{ata}(decl) = \{table\}$. The set of generated attributes associated with $decl$

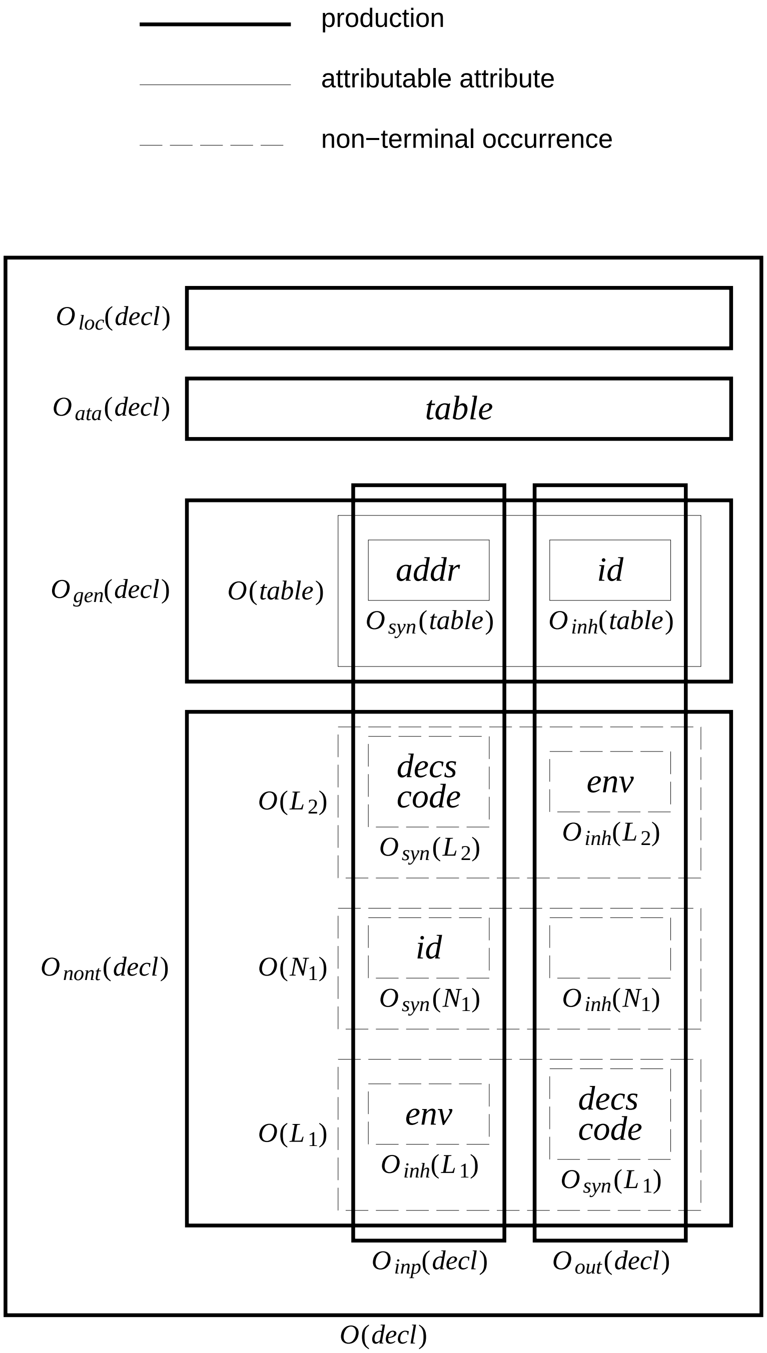


Figure 2.6. Venn-diagram of $O(decl)$; the higher-order variant

thus equals $O_{gen}(decl) = \{table.id, table.addr\}$. $O_{loc}(decl)$ and $O_{nont}(decl)$ are the same as in the plain attribute grammar. These four sets partition $O(decl)$. Another way to partition $O(decl)$ is as follows.

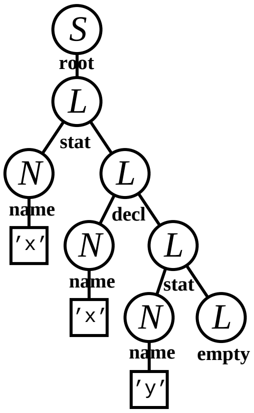
$$\begin{aligned}
O_{loc}(decl) &= \{\} \\
O_{ata}(decl) &= \{table\} \\
O_{inp}(decl) &= \{L_1.env, N.id, L_2.decs, L_2.code, table.addr\} \\
O_{out}(decl) &= \{L_1.decs, L_1.code, L_2.env, table.id\}
\end{aligned}$$

A Venn-diagram of all attribute occurrences of $decl$ is given in Figure 2.6. The borders of the sets associated with productions, attributable attributes respectively non-terminal occurrences are drawn in different line styles.

· · ·

In this section, we have defined the VARUSE grammar with two different specification methods namely an algebraic and a graphical formalism. The VARUSE grammar will serve as running example. The program `(use x;var x;use y;)` will figure as the (standard) VARUSE *application*. That variable 'y' is not declared is not crucial to the example. The following (linear rendering of the) abstract syntax corresponds with the given concrete syntax.

$$\mathbf{root}\Big(\mathbf{stat}\Big(\mathbf{name}(\text{'x'}), \mathbf{decl}\Big(\mathbf{name}(\text{'x'}), \mathbf{stat}\Big(\mathbf{name}(\text{'y'}), \mathbf{empty}()\Big)\Big)\Big)\Big)$$



Abstract syntax trees are best presented graphically. In traditional drawings of abstract syntax trees, tree nodes correspond to *non-terminal* instances. For an overview of the dependencies between the attribute instances it is better to use *production* instances as the tree nodes. Besides, that is how we defined abstract syntax trees in the first place.

The production icons can be pasted together to construct an abstract syntax tree with attribute instances. Figure 2.7$_\triangleright$ shows the abstract syntax tree for the VARUSE application. Adjacent non-terminal occurrences and their attributes unite nicely. Note that the abstract syntax tree induces a dependency graph on the attribute instances, the so-called dtr graph. The dtr "dependencies in a tree" graph appears clearly if the large discs and large rectangles are removed from Figure 2.7$_\triangleright$. The next section discusses dtr graphs.

2.4 Circularities

The completeness property alone does not guarantee that all attribute instances in a tree are effectively computable: cyclic dependencies will in general not be allowed. A circular attribute grammar is usually regarded as ill-defined. Nevertheless, viable semantics might be assigned. The equations of the grammar should not be considered as assignments—a view which is often silently adopted—but looked upon as *equations*. In that case, a least fixed-point can be defined, which

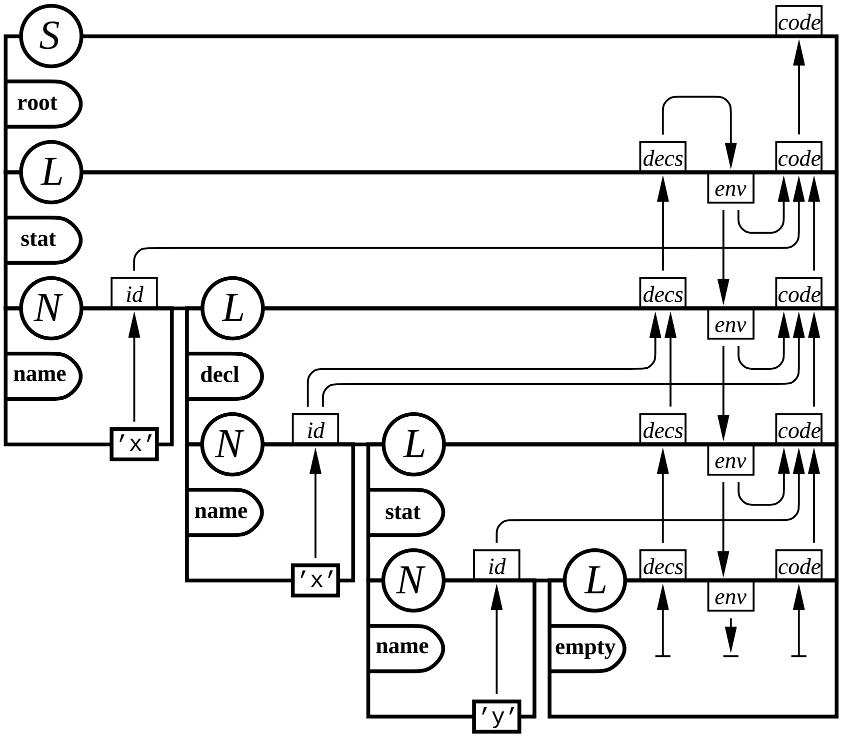


Figure 2.7. An abstract syntax tree "pasted together": the VARUSE application

can be approximated [Far86, Alb91a]. In this thesis, we only deal with non-circular attribute grammars.

This section presents two well-known algorithms that check for circularities. They are not only given for the sake of completeness, but also because they illustrate the concepts of static grammar analysis (bottom-up analysis, exponential versus polynomial approximation, infinite set represented by finite set, non-terminals and their occurrences, attributes versus attribute occurrences, pasting and projection). Familiarity with these concepts will aid in understanding the computation of visit-sequences discussed in Chapter 5. Furthermore, we will see how algorithms for plain attribute grammars can be adapted for higher-order attribute grammars.

dtr graphs is for plain grammars, not for higher-order grammars.

2.4.1 The dtr graph

An *instance* of attribute $X.a$ is an occurrence of $X.a$ in a structure tree. It is denoted by a tuple $(K, X.a)$ where K is an instance of non-terminal X. Likewise, an instance of an attribute $p.l$ is an occurrence of $p.l$ in a structure tree; it is denoted by a tuple $(K, p.l)$ where K is an instance of production p. We will use Greek letters near the end of the alphabet σ, τ, ... for attribute instances.

The function ins maps a tree node and an attribute occurrence onto an attribute instance. Note that the attribute occurrences of a right-hand side non-terminal occurrence $p\mathbf{o}i$ of production instance K are mapped onto attribute instances of node $i : K$.

$$\begin{aligned} ins(K,p\mathbf{o}0.a) &= (\,K\,,\,p{\cdot}0.a\,) \\ ins(K,p\mathbf{o}i.a) &= (\,i:K\,,\,p{\cdot}i.a\,), \quad i>0 \\ ins(K,p.l) &= (\,K\,,\,p.l\,) \end{aligned}$$

Recall that a tree node is denoted by a list of natural numbers.

Function ins is partial, since $ins(K,\alpha)$ is only defined if $\alpha \in O(prod(K))$.

Let T be a abstract syntax tree of some grammar G. The set $v_{dtr}(T)$ of all attribute instances is defined by the following recursive function

$$\begin{aligned} v_{dtr}(T) = v(T,[\,]) & \\ \textbf{where } v(t,K) &= \{ins(K,\alpha) \mid \alpha \in O(prod(t))\} \\ &\cup \textstyle\bigcup_{1\le i\le \mathbf{s}prod(t)} v(t{\cdot}i,\, i:K) \quad . \end{aligned}$$

$v(t,K)$ can equivalently be defined as $\{\ ins(K,\alpha)$ $\mid\ \alpha \in O(prod(t)\mathbf{o}0)$ $\vee\ \alpha \in O_{loc}(prod(t))$ $\}\ \cup \bigcup_i v(t{\cdot}i, i{:}K)$.

In other words, $v(t,K)$ consists of the attribute instances corresponding with the attribute occurrences of the root production of t (which is node K of the outermost tree T) and the attribute instances of the children $t{\cdot}i$ of t.

A point of confusion in this definition might be that every non-terminal instance (except for the root) is inspected twice: once as the parent of a production and once as a child of a production. Nevertheless, the attached attribute instances are not duplicated since ins generates the same tuple in both cases while building the set.

The set $e_{dtr}(T)$ contains the dependencies between the attribute instances. Function ins is used to map dependencies on attribute occurrences onto dependencies on attribute instances.

$$\begin{aligned} e_{dtr}(T) = e(T,[\,]) & \\ \textbf{where } e(t,K) &= \{ins(K,\alpha) \to ins(K,\beta) \mid \alpha \to \beta \in dpr(prod(t))\} \\ &\cup \textstyle\bigcup_{1\le i\le \mathbf{s}prod(t)} e(t{\cdot}i,\, i:K) \end{aligned}$$

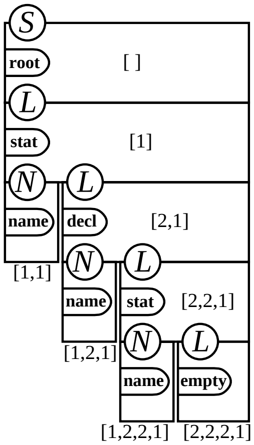
Essentially, dpr graphs are pasted together.

The attribute instances of an abstract syntax tree T induce a dependency graph $dtr(T)$ "dependencies in a tree" defined as $dtr(T) = (v_{dtr}(T), e_{dtr}(T))$. The base of the above recursive definitions is formed by trees T such that a terminal production is applied at the root of T. For such trees T, $dtr(T)$ is isomorphic with $(O(p), dpr(p))$. An attribute grammar AG is non-circular if the attribute dependency graph $dtr(T)$ for any tree T is a-cyclic.

Definition 8 *Non-circular attribute grammar.*
An attribute grammar is non-circular if for each structure tree T of the grammar, $dtr(T)$ is a-cyclic.
□

An attribute grammar is well-defined if and only if it is non-circular [WG84].

Figure 2.7 gives (the dtr graph of) the VARUSE application. Node $[2,2,2,1]$ is the only instance of $empty$. Its parent node, denoted by $[2,2,1]$, is an instance of production $stat$. Attribute instances $([2,2,1], L.code)$ and $([2,2,2,1], L.code)$ correspond with attribute occurrences of $stat$: the former with $stat\mathbf{o}0.code$ and the latter with $stat\mathbf{o}2.code$. The $dpr(stat)$ arc $stat\mathbf{o}2.code \to stat\mathbf{o}0.code$ is lifted to the attribute instance dependency $([2,2,2,1], L.code) \to ([2,2,1], L.code)$.



```
var
  $DS$ : array [non-terminal] of set of graph of attributes
  $d_0, \ldots, d_s$ : graph of attributes
  $d$ : graph of attribute occurrences
  $cycle$, $conv$ : bool
begin
  for each non-terminal $X$ do $DS[X] := \{\}$ rof
  $cycle := false$
  repeat
    $conv := true$
    for each production $p : X_0 \longrightarrow X_1 \ldots X_s$ do
      for each tuple $(d_1, \ldots, d_s) \in DS[X_1] \times \cdots \times DS[X_s]$ do
        $d := dpr(p)[\{\}, d_1, \ldots, d_s]^+$
        if $d$ is circular then $cycle := true$ fi
        $d_0 := d \upharpoonright 0$
        if $d_0 \notin DS[X_0]$ then $conv := false$; $DS[X_0] := DS[X_0] \cup \{d_0\}$ fi
      rof
    rof
  until $conv \vee cycle$
  { $cycle = AG$ is circular }
end
```

Algorithm 2.8. The circularity test

2.4.2 The circularity test

Now let us consider an algorithm which determines whether or not an attribute grammar is circular. We want to decide whether there exists an abstract syntax tree for which the attribute dependency graph is circular. Of course, we can not enumerate all abstract syntax trees, there are possibly infinitely many.

However, we are not interested in every conceivable *tree*. We are interested in its *attribute dependencies*. Note that only the (transitive) dependencies between the attributes of the root of a tree are visible to the "outside world". This leads to the key idea of representing the *infinite* set of abstract syntax trees with the same root X by the *finite* set of graphs on $A(X)$. The number of these so-called *is-graphs* [Alb91c] or *subordinate characteristic graphs* [RTD83] is finite because $|A(X)|$ is finite.

Algorithm 2.8 uses this finite representation to determine whether a cycle exists. It is basically Knuth's revised algorithm [Knu71]. How does it work?

Suppose there exist abstract syntax trees $T_1 \ldots T_s$ whose roots are labeled with the respective non-terminals $X_1 \ldots X_s$. Suppose further that tree T_i induces dependencies between the attributes of $A(X_i)$ that are recorded in the dependency set d_i. We would like to compute the dependency set on $A(X_0)$ reflecting the dependencies in the tree T_0 that is constructed by applying production $p : X_0 \longrightarrow X_1 \ldots X_s$ to the subtrees $T_1 \ldots T_s$. This is achieved by pasting the dependencies $d_1 \ldots d_s$ into $dpr(p)$ and taking a transitive closure. Pasting is defined as follows.

Definition 9 *Pasting attribute graphs into an attribute occurrence graph.*
Given a production $p : X_0 \longrightarrow X_1 \ldots X_s$, a dependency set d_p on $O(p)$ and the dependency sets d_i on $A(X_i)$, $0 \leq i \leq s$, the graph $d_p[d_0, \ldots, d_s]$ is defined as the dependency set $d_p \cup \bigcup_{0 \leq i \leq s} \{p\mathbf{o}i.a \rightarrow p\mathbf{o}i.b \mid p{\cdot}i.a \rightarrow p{\cdot}i.b \in d_i\}$ on $O(p)$.
□

The paste operator $d_p[\ldots]$ requires $1 + \mathrm{s}p$ arguments: the dependency graphs on $A(p{\cdot}i)$ for $0 \leq i \leq \mathrm{s}p$. If, as in the circularity test, one of the graphs needs not to be pasted, we supply the empty dependency set, for example, $d_p[\{\}, d_1, \ldots, d_s]$.

Pasting takes (dependencies on) attributes and converts them into (dependencies on) attribute occurrences. *Projection* does the opposite. Given a production p and a non-terminal occurrence index i, $0 \leq i \leq \mathrm{s}p$, it extracts the dependencies for $p\mathbf{o}i$ and converts them into dependencies for $p{\cdot}i$.

Definition 10 *Projection.*
Given a dependency set d on $O(p)$, and index i, $0 \leq i \leq \mathrm{s}p$, projection $\restriction$ is defined as follows $d{\restriction}i = \{p{\cdot}i.a \rightarrow p{\cdot}i.b \mid p\mathbf{o}i.a \rightarrow p\mathbf{o}i.b \in d\}$.
□

The circularity test algorithm computes a closure. The base is formed by terminal productions. For a terminal production p, the inner loop performs a single step with the empty tuple $()$, which is the only element in a cartesian product of zero sets. In that single step d is initialized to $dpr(p)$, enabling tuples for other productions. In other words, the is-graphs are built bottom-up.

2.4.3 Strongly non-circular

It is well-known that the circularity test is exponential [Jaz81, DJL84, Alb91c]. We will now present a polynomial algorithm that approximates the circularity test. It is safe in the sense that circular grammars will be flagged as such, but it is pessimistic in the sense that some non-circular grammars will also be flagged.

Algorithm 2.9$_{\triangleright}$, which is Knuth's first (too pessimistic) circularity test [Knu68], defines the class of *strongly non-circular attribute grammars* (SNC) [Jou84], also known as *absolutely non-circular attribute grammars* (ANC) [KW76]. The basic difference with the "real" circularity test is that during the computation it *merges* the graphs in $DS[X]$ into one dependency set $D[X]$. The dependency sets are initialized to $\{\}$: they contain no *arcs*.

As an example of the pessimistic nature consider the grammar in Figure 2.10$_{\triangleright}$. It defines two abstract syntax trees, $\mathbf{p}(\mathbf{q}())$ and $\mathbf{p}(\mathbf{r}())$. Both the induced dtrs are cycle free, hence the grammar is non-circular. The circularity test determines $DS[X]$ to be $\{\{X.i \rightarrow X.s\}, \{X.j \rightarrow X.t\}\}$. However, in the strongly non-circular test, these graphs are merged yielding $D[X] = \{X.i \rightarrow X.s\ ,\ X.j \rightarrow X.t\}$. Pasting this graph into $dpr(p)$ gives a cycle. Hence the grammar is not strongly non-circular.

```
var
  $D$ : array [non-terminal] of dependency_set of attributes
  $d_0$ : graph of attributes
  $d$ : graph of attribute occurrences
  $cycle$, $conv$, $snc$ : bool
begin
  for each non-terminal $X$ do $D[X] := \{\}$ rof
  $cycle := false$
  repeat
    $conv := true$
    for each production $p : X_0 \longrightarrow X_1 \ldots X_s$ do
      $d := dpr(p)[\{\}, D[X_1], \ldots, D[X_s]]^+$
      if $d$ is circular then $cycle := true$ fi
      $d_0 := d{\restriction}0$
      if $d_0 \backslash D[X_0] \neq \{\}$ then $conv := false$; $D[X_0] := D[X_0] \cup d_0$ fi
    rof
  until $conv \vee cycle$
  $snc := \neg cycle$
  { $snc = AG$ is strongly non-circular }
end
```

Algorithm 2.9. The strongly non-circular test

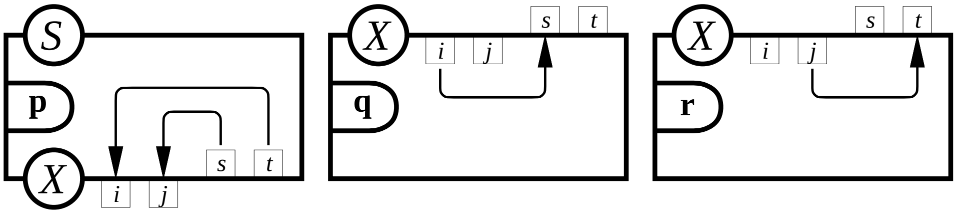


Figure 2.10. A non-circular grammar that is not strongly non-circular

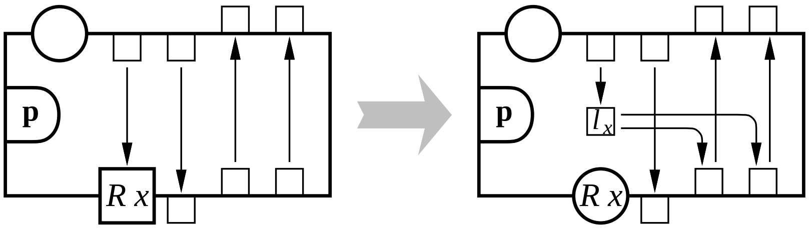


Figure 2.11. Reducing a production of a higher-order attribute grammar

2.4.4 Reducing higher-order attribute grammars

Most algorithms that analyze attribute grammars for attribute dependencies can be adapted to handle higher-order attribute grammars as well. The key idea is to transform the higher-order attribute grammar into a plain attribute grammar. An attributable attribute is thereby replaced by a non-terminal *and* a local attribute. The local attribute captures the dependencies of the attributable attribute itself. The attribute occurrences of the non-terminal capture the dependencies of the generated attribute occurrences. This transformation is known as *reduction* [Vog93]. Of course, reduction only makes sense if the information gathered by analyzing the reduced grammar can be interpreted for the underlying higher-order grammar.

Needless to say, a higher-order tree can only be decorated after it has been computed. Therefore, the generated synthesized attributes, which are determined by decorating the higher-order tree, should neither directly nor indirectly contribute to the computation of the higher-order tree. The fourth item in the following definition of reduction makes these dependencies explicit. Figure 2.11 illustrates reduction.

Definition 11 *Reducing a higher-order attribute grammar.*
Let H be a higher-order attribute grammar. The reduced attribute grammar H' is obtained by the following transformations on every production p of H:

- For each $p.x \in O_{ata}(p)$ a non-terminal $R\ p.x$ is appended to p, say at position i_x, and a fresh local variable $p.l_x$ is introduced.
- Every attributable attribute $p.x \in O_{ata}(p)$ is removed.
- Every attribute occurrence $p.x.a$ in any equation of $E(p)$ is replaced by $p\mathbf{o}i_x.a$.
- Every synthesized attribute occurrence $p\mathbf{o}i_x.s$ is explicitly made dependent on $p.l_x$; for example by adding $p.l_x \to p\mathbf{o}i_x.s$ to $dpr(p)$.

□

We loose information when reducing a higher-order grammar to a plain attribute grammar. We can no longer distinct between a "real" non-terminal and a non-terminal $p\mathbf{o}i_x$ that replaces attributable attribute $p.x$. As a consequence, attribute dependencies deduced for $A(p{\cdot}i_x)$ might be assumed to also occur at $p\mathbf{o}i_x$.

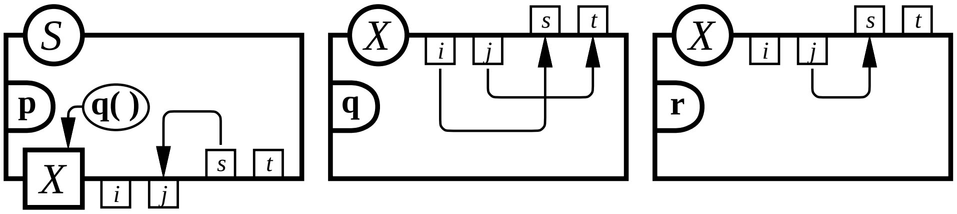


Figure 2.12. A well-defined higher-order grammar that is not reduced non-circular

However, in the underlying higher-order grammar, the instantiated tree for $p.x$ is fixed by attribute evaluation; it can certainly not have an arbitrary shape. Thus, an algorithm for plain attribute grammars that is applied to reduced higher-order grammars is generally pessimistic.

Testing whether a higher-order attribute grammar is non-circular is hard; it appears to be as hard as the halting problem.

For example, the circularity test can be used for reduced higher-order grammars as well. But some reduced higher-order grammars will fail the test although all attribute instances of every abstract syntax tree are effectively computable. A higher-order attribute grammar is said to be *reduced non-circular* if the reduced grammar is non-circular. Reduced non-circular is a sufficient condition for well-definedness of higher-order attribute grammars.

Definition 12 *Reduced non-circular higher-order attribute grammar.*
An higher-order attribute grammar is reduced non-circular if the reduced grammar is non-circular.
□

Figure 2.12 shows a higher-order attribute grammar that is not reduced non-circular although well-defined. This grammer defines only one abstract syntax tree, namely $\mathbf{p}()$. The equation for the attributable attribute $p.x$ of type X specifies that $\mathbf{q}()$ should be instantiated. Decoration of $\mathbf{q}()$ is possible in the given context of $\mathbf{p}()$. However, the reduced grammar defines two abstract syntax trees, namely $\mathbf{p}(\mathbf{q}())$ and $\mathbf{p}(\mathbf{r}())$. The latter generates a circularity, so that the reduced grammar is circular and hence the higher-order grammar not reduced non-circular.

Chapter 3

Using visit-sequences for incremental evaluation

We discuss how to obtain an elegant and efficient incremental attribute evaluator. For the subclass of *partitionable attribute grammars*, so-called *visit-sequences* can be statically determined. In Chapter 5 we will discuss how to *compute* visit-sub-sequences. In this chapter we discuss how visit-sequences are used for plain and incremental attribute evaluation. We also present a new approach that converts visit-sequences into *visit-functions*, which can be memoized in order to obtain incremental behavior.

3.1 Introduction

This section discusses attribute evaluation and introduces visit-sequences and visit-sequence related terminology and notations.

3.1.1 Plain and incremental evaluation

An attribute grammar specification consists of a context-free grammar and attribution rules. The context-free grammar defines a set of abstract syntax trees. The nodes of an abstract syntax tree are decorated with attribute instances. The attribution rules of the attribute grammar describe how attribute instances depend on each other. An *attribute evaluator* is an algorithm that, given an abstract syntax tree T, computes the values of the attribute instances associated with the nodes of T. The result of attribute evaluation is defined as the values of the synthesized attribute instances of the root of T.

In an interactive environment, a user slightly modifies a fully decorated tree T into tree T'. An *incremental* attribute evaluator uses T and its attribute instances to compute the attribute instances of T'. The underlying assumption is that decoration of T' from scratch is more expensive (in time) than an incremental update. This is motivated by the observation that changes from T to T' are often small so that the changes in the attribute instances of T to T' are frequently also small. Incremental evaluators are optimized for *time* efficiency; space consumption is not a major topic.

Compiler generators generate attribute evaluators.

An attribute grammar evaluator is generated from an attribute grammar specification at *generation time*. The time during which the generated evaluator itself is running will be referred to as *evaluation time*. Since both processes are a form of compilation we will avoid the term "compile time". The *grammar writer* writes the attribute grammar specification, and the *user* provides the evaluator with input.

Formal descriptions such as attribute grammars are well suited for various forms of generation time analysis or *static analysis*. These may result in particularly efficient evaluators [KW76, Boc76, Kas80], space optimized evaluators [Kas87, EdJ90, JP90, odAS91], parallel evaluators [Kui89, Zar90, Jou91, Kle91, Rep93] or incremental evaluators [DRT81, Rep82, Yeh83, RTD83, YK88, Alb91b].

Plain evaluation

Conceptually, attribute evaluation—*plain* attribute evaluation as opposed to incremental attribute evaluation—is not complex. The attribute instances of an abstract syntax tree induce a dependency graph dtr. The attribute instances must be computed in a *total order* that complies with the dtr graph. In essence, an attribute evaluator is an algorithm that determines such an order, and computes the attribute instances on the fly.

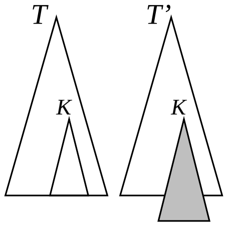
Visit-sequences are determined at generation time. They impose a total order on the attribute occurrences of a production. Furthermore, they describe how total orders associated to subtrees of a production instance K should be composed to describe a total order on the attribute instances of the entire tree rooted at K. In other words, visit-sequences statically induce a total order on the attribute instances of any abstract syntax tree.

Visit-sequence based evaluators are more efficient than evaluators that dynamically determine a total order. However, visit-sequence based evaluation is restricted to a subclass of attribute grammars called *partitionable grammars* (see Chapter 5). Generally, the more efficient an evaluation strategy, the smaller the class of grammars that can be evaluated by that strategy.

Incremental evaluation

An *incremental attribute evaluator* is an algorithm that recomputes a tree's attributes after an operation that slightly changes that tree. Although any non-incremental attribute evaluator can be applied to completely redecorate the tree, our goal is to minimize work by focusing on the attribute instances that have a new or changed value. The set of attribute instances that require a new value is referred to as Δ; it is commonly known as AFFECTED [DRT81, Yeh83].

To be more precise, let T' be an abstract syntax tree resulting from a subtree replacement at node K in tree T. The attribute instances in the new subtree at



K are called *newborn*, all other instances are called *retained*. When evaluating T' only retained attribute instances may have the same value as in T. Let us call the unchanged attribute instances *equal* as opposed to the *changed* instances. Needless to say, the subtree replacement causes these changes. An incremental attribute evaluator has to assign new values to the changed and newborn attribute instances. This subset is denoted by Δ. The following table sums up the relations.

attribute instances		
retained		newborn
equal	changed	
	Δ	

It is hard to give a good definition for Δ in the case of higher-order grammars, since the set of attribute instances grows during evaluation.

Bear in mind that it is not known in advance which attribute instances are members of Δ: the incremental update algorithm has to determine Δ itself. An incremental evaluator is called *asymptotically optimal* if its costs are $\mathcal{O}(|\Delta|)$. In measuring evaluation costs, we neglect the true cost of evaluating the semantic functions and regard them as having unit costs. In reality, such functions can be more expensive as is illustrated by the $lookup$ function in the VARUSE grammar.

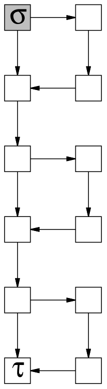
Incremental attribute evaluators must recompute a (small) superset of Δ in order to determine the edge of Δ: $\Delta \cup \{\tau \mid \exists_{\sigma \in \Delta} \sigma \rightarrow \tau\}$. This set, denoted by Δ^{+1}, extends Δ with those attribute instances that are reachable in a single step from the frontier of Δ. Due to the constant out-degree of attribute occurrences in dpr graphs (given an attribute grammar) $\mathcal{O}(|\Delta^{+1}|) = \mathcal{O}(|\Delta|)$ holds.

Some define double-bar delta $||\Delta|| := |\Delta^{+1}|$.

Incremental evaluation requires the storage of "old values" of the attribute instances. In this sense, incremental evaluation conflicts with storage optimizations (such as mapping several attribute instances to one global variable). In most approaches, the values of the attribute instances are stored in the abstract syntax tree. In other words, the abstract syntax tree serves as a very efficient attribute cache. It should combine this function with its primary role, the one of data flow driver. The functional approach presented in this chapter, separates the two functions of the abstract syntax tree: attribute instances will be stored in a cache tailored for incremental evaluation, while the abstract syntax tree is optimized for driving the evaluation (see Chapter 4).

A naive implementation of incremental attribute evaluation is *dynamic change propagation*. Attribute instances are marked if their value is possibly inconsistent. Initially, the attribute instances of the newly inserted subtree are marked. The dynamic change propagation algorithm selects a marked attribute instance that has no marked predecessors and computes its value (such instances must exist since $dtr(T)$ is non-circular). If the new value differs from the old one, all successors of the attribute instance are marked. When propagation has died out, all instances will be consistent again.

The dynamic change propagation is inefficient for two reasons. First of all, an attribute instance may be assigned a "new" value in the course of reevaluation, that is not its correct final value. For example, if an attribute instance τ depends



on a marked instance σ via eight paths, unfortunate scheduling may cause τ to be assigned eight times. The efficiency of dynamic change propagation may thus be as bad as exponential in the number of the marked attribute instances. Secondly, the set of marked attributes can be far larger than Δ. This is also due to multiple assignments: spurious changes are likely to propagate beyond Δ leading to suboptimal running time.

The obvious solution to the naive algorithm is to make sure that an attribute instance is only reevaluated after all its arguments have gotten their correct final values. In order to achieve this, the dependencies between the attribute instances—direct as well as indirect—have to be known. For arbitrary non-circular grammars, Reps presented an algorithm PROPAGATE that dynamically constructs a dependency graph that is topologically sorted on the fly [Rep82, RTD83]. It has an optimal running time, but its space consumption is considerable; it usually dominates the storage for attribute values [RT89].

Visit-sequences implicitly establish an order on the attribute instances of an abstract syntax tree. An incremental evaluator can make use of this order [Yeh83, YK88, RT89]. This eliminates the need for a dynamically constructed graph which makes the algorithm not only faster but also less memory consuming. Of course the applicability of this method is restricted to the class of partitionable attribute grammars.

3.1.2 Visit-sequences

A *tree-walk evaluator* traverses the abstract syntax tree. It starts at the root and moves from node to (adjacent) node. A subtree is *entered* when it is reached with a downward move and it is *exited* with an upward move from the root of the subtree. During a traversal attribute instances are computed. The decoration of a tree is completed when all its attribute instances are computed and the root node is exited.

A *visit* is a sequence of actions that starts with entering a subtree T, followed by a number of attribute computations during the traversal of T and then exiting T. In general multiple visits to T are necessary to compute *all* its attribute instances. During a specific visit to T, multiple visits to any subtree of T may take place and some subtree might not be visited at all.

A *visit-sequence evaluator* is a tree-walk evaluator that has associated a fixed sequence of moves, interspersed with attribute computations, with each production. Such a sequence is known as *visit-sequence* or *plan*. Each node of the abstract syntax tree is an *instance* of a production, and the instructions from the plan apply to the actual non-terminal and attribute instances of that node. Three kind of instructions are used, two transfer instructions and one evaluation instruction. The following table lists them and their semantics when executed in node K:

The plan for node K includes $eval$ instructions for attributes associated with node $i:K$.

instruction	**semantics in tree node K**
$eval(\alpha)$	compute attribute instance $ins(K, \alpha)$
$visit(i, v)$	descend to node $i : K$ for the vth time
$suspend(v)$	exit visit number v and ascend to $tail\ K$

In a visit-sequence evaluator, the number of visits to a non-terminal is fixed. Furthermore, each visit to a non-terminal has a fixed *interface* that consists of a set of inherited attributes of the root that may be used during the visit and a set of synthesized attributes of the root that are guaranteed to be computed by the visit (the yield).

Figure 3.1 shows the visit-sequences of the VARUSE grammar. Note that instead of writing $visit(2, 1)$, we write $visit(L_2, 1)$. If there is only one occurrence of a non-terminal in a production, we even leave out the subscript and write $visit(L, 1)$.

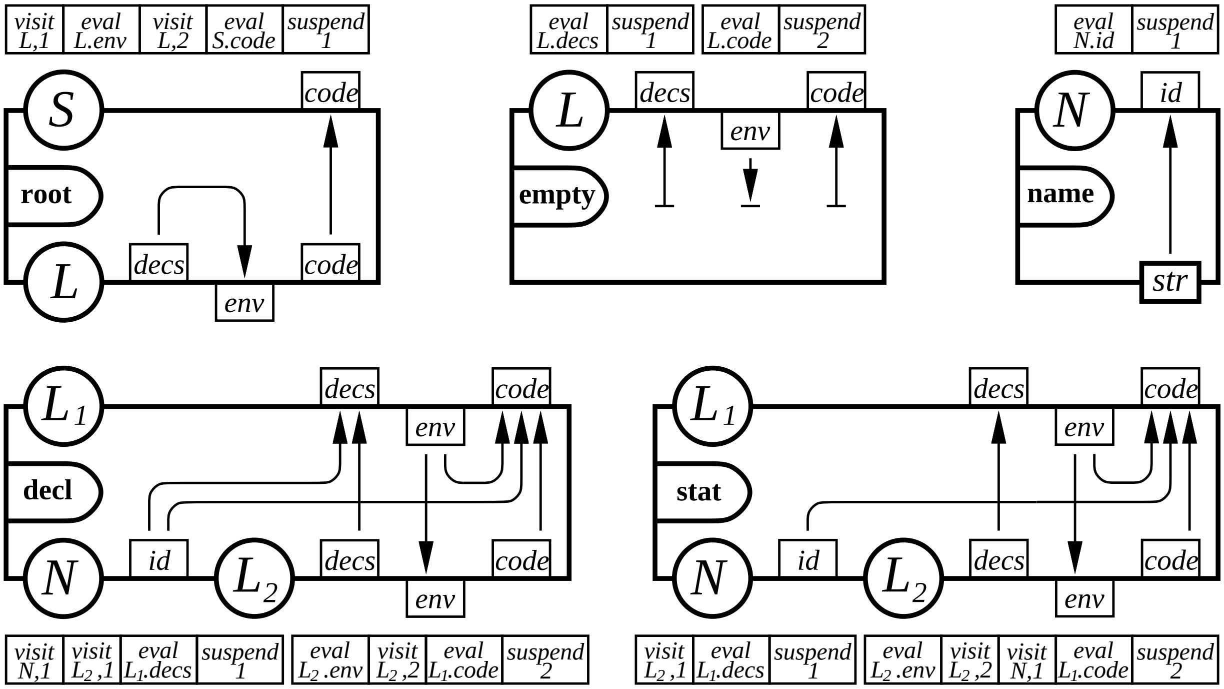

visit L,1 | eval L.env | visit L,2 | eval S.code | suspend 1
S
code
root
L
decs
env
code
eval L.decs | suspend 1
eval L.code | suspend 2
L
decs
env
code
empty
eval N.id | suspend 1
N
id
name
str
L1
decs
env
code
decl
N
id
L2
decs
code
env
visit N,1 | visit L2,1 | eval L1.decs | suspend 1
eval L2.env | visit L2,2 | eval L1.code | suspend 2
L1
decs
env
code
stat
N
id
L2
decs
code
env
visit L2,1 | eval L1.decs | suspend 1
eval L2.env | visit L2,2 | visit N,1 | eval L1.code | suspend 2

Figure 3.1. The production icons of the VARUSE grammar with their plans

Each L tree is visited twice. The interface for the first visit is $(\{\}, \{L.decs\})$ and for the second $(\{L.env\}, \{L.code\})$. The plans for a production on L are divided into two *sub*-sequences corresponding to the two visits to L. The plan for a $root$ node (which is not divided since S is only visited once), contains both visits to its L child. During the first visit to a $stat$ node, its L child is visited for the first time. The second visit to this child takes place during the second visit to $stat$. This "synchronization" need not be, it simply follows from the particular dependencies in $dpr(stat)$. See for example the visits to the N children: in a $decl$ node N is skipped during the second and in a $stat$ node during the first visit to L.

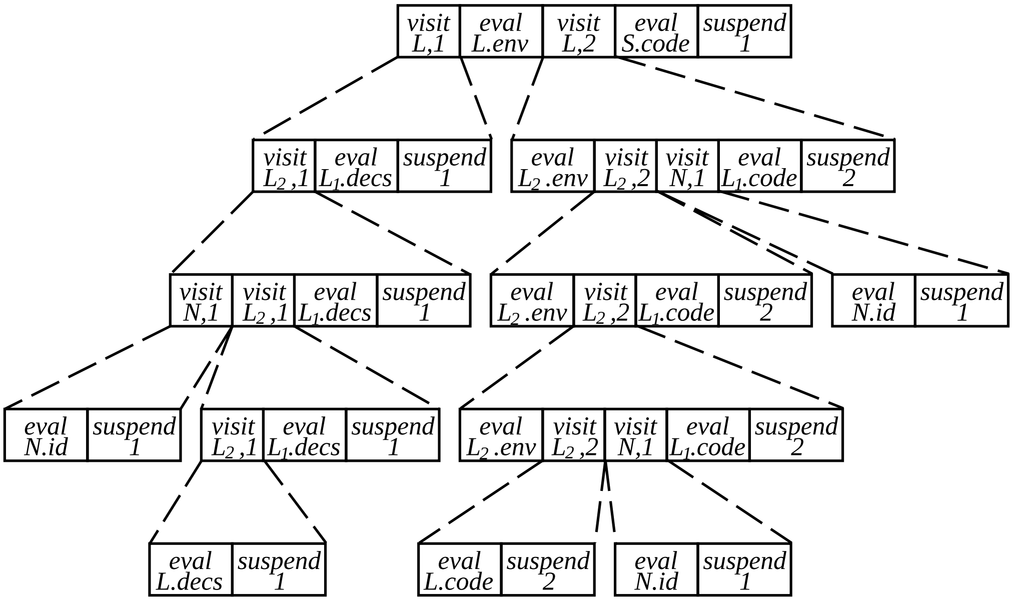


Figure 3.2. The plan tree for the VARUSE application

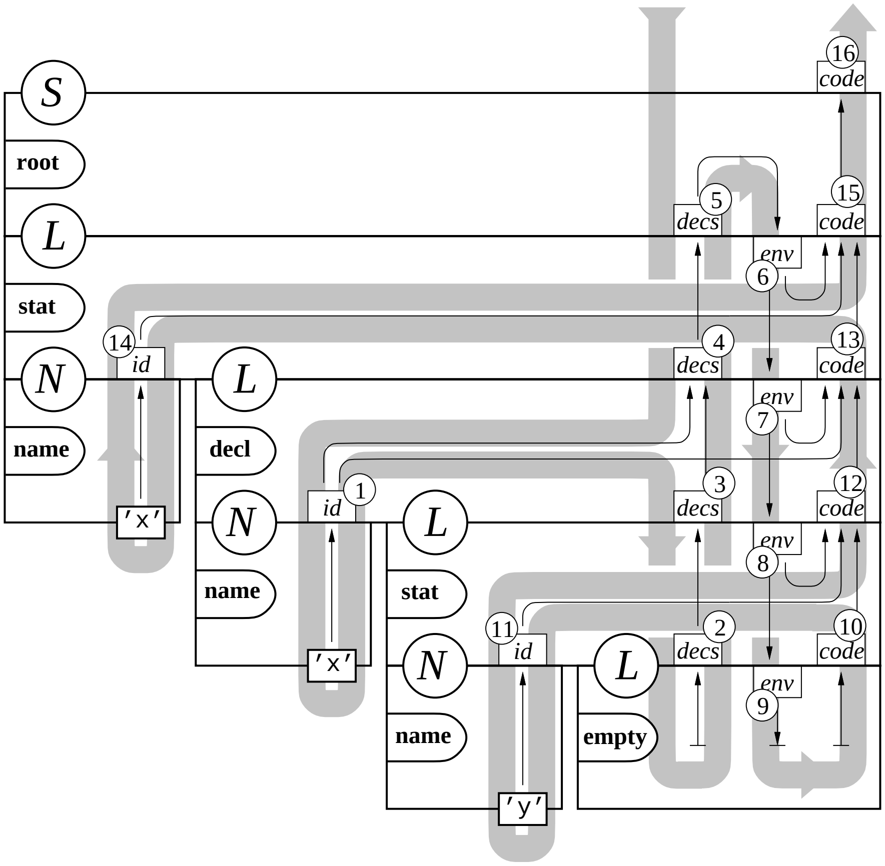


Figure 3.3. Total order on the attribute instances for the VARUSE application

```
plan root
begin 1
  visit($L$, 1);
  eval($L.env$);
  visit($L$, 2);
  eval($S.code$)
end 1

plan decl
begin 1
  visit($N$, 1);
  visit($L_2$, 1);
  eval($L_1.decs$)
end 1
begin 2
  eval($L_2.env$);
  visit($L_2$, 2);
  eval($L_1.code$)
end 2

plan stat
begin 1
  visit($L_2$, 1);
  eval($L_1.decs$)
end 1
begin 2
  eval($L_2.env$);
  visit($L_2$, 2);
  visit($N$, 1);
  eval($L_1.code$)
end 2

plan empty
begin 1
  eval($L.decs$)
end 1
begin 2
  eval($L.code$)
end 2

plan name
begin 1
  eval($N.id$)
end 1
```

Figure 3.4. The visit-*sub*-sequences of the VARUSE grammar "structured style"

Decoration proceeds in a coroutine like fashion. We imagine machines attached to the nodes of the trees. These machines execute the plan associated with the node. At any moment, only one of the machines is active. It stops execution by transferring control to the machine of the parent (*suspend*) or to the machine of one of its children (*visit*). The *eval* instructions perform the actual computation: evaluating the attribute instances.

The *eval* instructions in the visit-sequence of production p describe a total order on the attribute occurrences of p. The total orders on the attribute occurrences of the productions induce a total order on the attribute instances of an abstract syntax tree. By composing the plan induced total orders of the subtrees $T_1, \ldots, T_S$ with the total order described by p, the plan induced total order of the tree $\mathbf{p}(T_1, \ldots, T_s)$ is obtained. Terminal productions form the base of the recursion.

For example, Figure 3.2 shows how the plans of the productions of the VARUSE application are composed to form a so-called *plan tree*. A depth-first left-to-right traversal of the plan tree that takes only the *eval* instructions into account, defines the plan induced total order on the attribute instances of the VARUSE application. Figure 3.3 shows the VARUSE application with a gray line denoting a "time-axis" flowing through all the attribute instances which, for clarity, have been "timestamped".

The visit-sequences presented thus far have been formatted in a "classic style" used by many authors. That is to say, the three basic instructions are simply listed. Classic style visit-sequences are the *output* of plan generating algorithms.

However, the visit-sequences are also used as *input* for an algorithm that extracts visit-functions as discussed later in this chapter. It is then convenient to bring more structure into visit-sequences: they are divided into *sub*-sequences. The *suspend* instructions are used as breakpoints, and become obsolete as shown in Figure 3.4. The difference between classic and structured plans can be neglected, and we will use the two views interchangeably.

3.1.3 Interfaces and partitionable grammars

Visit-sequence based evaluators are particularly efficient. However, visit-sequences only exist if for every non-terminal in the grammar the evaluation order of its attributes can be fixed at generation time. To see why, consider the two machines attached to adjacent nodes in an abstract syntax tree. The actual productions applied at those nodes are not known to the respective machines at the other side; they are only joined by a non-terminal, say X. The inherited attribute instances of X are computed by the upper machine, the synthesized attribute instances by the lower. The interaction between the machines is as follows. The upper computes some subset of $A_{inh}(X)$ as indicated by the $eval$ instructions in its plan. It then passes control to the lower machine by executing a $visit$ instruction. The lower machine executes its plan, computing a subset of $A_{syn}(X)$. It transfers control to the upper machine by executing a $suspend$ instruction. Both machines could have evaluations and transfers to other nodes interspersed. This process of alternatingly computing inherited and synthesized attribute instances continues until all of X's attribute instances have been processed.

The attributes of X constitute the only interface between the upper and lower machine. When the upper machine transfers control to the lower it expects that a particular subset of $A_{syn}(X)$ will be computed before control is transferred back, *irrespective* of the subtree present. Likewise, whatever the current context of the lower machine, it expects a particular subset of $A_{inh}(X)$ to be available so that it may refer to them in order to compute the required synthesized attribute instances.

Definition 13 *Interface of a non-terminal.*
A list l of pairs of sets is an *interface* for non-terminal X if its sets form a partition of $A(X)$ in the following sense

$[\underbrace{(\{.\},\{.\})}_{i=0},\underbrace{(\{.\},\{.\})}_{i=1},..]$

$$\begin{aligned} A_{inh}(X) &= \mathop{+}_{0 \le i < len\ l} fst\ (l{\cdot}i) \quad , \\ A_{syn}(X) &= \mathop{+}_{0 \le i < len\ l} snd\ (l{\cdot}i) quad. \end{aligned}$$

□

The interface for non-terminal X, computed by an algorithm f, is denoted by $interface_f(X)$. In order to facilitate interface related discussions, we introduce some notation ($\mathbf{v}_f$, $\mathbf{I}_f$ and $\mathbf{S}_f$).

The number of visits to non-terminal X is $len\ interface_f(X)$. It will be denoted by $\mathbf{v}_f X$. We overload this operator: $\mathbf{v}_f p\mathbf{o}i = \mathbf{v}_f p{\cdot}i$ and we abbreviate $\mathbf{v}_f p = \mathbf{v}_f p{\cdot}0$ and $\mathbf{v}_f T = \mathbf{v}_f prod(T)$. Furthermore, for $1 \le v \le \mathbf{v}_f X$ we define $X\mathbf{I}_f\ v = I$ and $X\mathbf{S}_f\ v = S$ where tuple $(I, S) = interface_f(X){\cdot}(v-1)$. In other words, $(X\mathbf{I}_f\ v, X\mathbf{S}_f\ v)$ is the interface for the vth visit to X. The selectors $\mathbf{I}_f$ and $\mathbf{S}_f$ are overloaded; they operate on non-terminal occurrences and attributable attributes too: $p\mathbf{o}i\mathbf{I}_f\ v = \{p\mathbf{o}i.a \mid p{\cdot}i.a \in p{\cdot}i\mathbf{I}_f\ v\}$ and $p\mathbf{o}i\mathbf{S}_f\ v = \{p\mathbf{o}i.a \mid p{\cdot}i.a \in p{\cdot}i\mathbf{S}_f\ v\}$ respectively $p.x\mathbf{I}_f\ v = \{p.x.a \mid X = R\ p.x \wedge X.a \in X\mathbf{I}_f\ v\}$ and $p.x\mathbf{S}_f\ v = \{p.x.a \mid X = R\ p.x \wedge X.a \in X\mathbf{S}_f\ v\}$. In general, we drop the subscript f.

For historic reasons, visits are numbered starting from 1.

Interface $\mathit{interface}(X)$ induces a partial order on the attributes $A(X)$. To define this partial order, we first present the auxiliary notion of time slots. Time slots are associated with attribute sets from $\mathit{interface}(X)$. Slots are numbered from 1 to $2 \cdot \mathbf{v}X$ and map to the sets as follows: $\mathit{slot}(X\mathbf{I}\,v) = 2v - 1$ and $\mathit{slot}(X\mathbf{S}\,v) = 2v$. We also associate a time slot $\mathit{slot}(X.a)$ with an attribute $X.a$; either $\mathit{slot}(X.a) = \mathit{slot}(X\mathbf{I}\,v)$ or $\mathit{slot}(X.a) = \mathit{slot}(X\mathbf{S}\,v)$ depending on $X.a \in X\mathbf{I}\,v$ or $X.a \in X\mathbf{S}\,v$. The partial order induced by $\mathit{interface}(X)$ can now be characterized as follows: $X.a \to X.b$ if and only if $\mathit{slot}(X.a) < \mathit{slot}(X.b)$. Attributes within the same slot (interface set) are not ordered.

$[\underbrace{(\overset{1}{\{\cdot\}},\overset{2}{\{\cdot\}})}_{v=1},\underbrace{(\overset{3}{\{\cdot\}},\overset{4}{\{\cdot\}})}_{v=2},..]$

The largest class of attribute grammars for which visit-sequences can be computed is the class of *partitionable attribute grammars*. Informally, an attribute grammar is partitioned if for each non-terminal an interface exist, such that in any context of the non-terminal the attributes are computable in an order which is included in the partial order induced by the interface. Visit-sequences are not fixed by interfaces; they merely have to comply with them. Chapter 5 gives a formal definition of partitionable attribute grammars. It also presents two different algorithms to compute visit-sequences that comply with the same interface. In this chapter, we simply assume that plans exist.

3.1.4 Background

An attribute evaluator computes the attribute instances of an abstract syntax tree T in a *total order* that complies with $\mathit{dtr}(T)$. We immediately obtain two different classes of evaluators. *Demand driven* or top-down evaluators start with (part of) the sinks of the dtr graph and work their way downwards. *Data driven* or bottom-up evaluators start with the sources of the dtr graph and work their way up. The advantage of the demand driven approach lies in the ability to avoid useless computations caused by, for example, conditional expressions. Kaviani [Kav84] showed that the advantage of the top-down approach is not essential over bottom up.

An unrelated classification is based on the ordering strategy. The total order on dtr can be determined at evaluation time or at generation time. Evaluators of the former kind are know as *dynamic* evaluators, whereas evaluators of the latter kind are referred to as *static*. Intermediate forms also exist [KW76]: evaluation is mainly dynamic, but some decisions are made at generation time.

In general, dynamic evaluators [KR79, CH79] are slower and consume more memory than static evaluators since they construct dependency graphs at evaluation time. However, dynamic evaluators have a distinct advantage over static evaluators. In general, they work for arbitrary non-circular attribute grammars. A drawback of this feature is that, unless the grammar is checked for circularity, one can not be sure the evaluator always terminates. It is well know that this test is exponential [Jaz81, DJL84, Alb91c].

One may check for circularities at evaluation time: constant costs.

Numerous statically determined evaluators have been researched [Eng84]. Alblas [Alb91a] subdivides static evaluators into *flexible* and *rigid* evaluators. For evaluators that use a rigid strategy, the computation order is fixed a priori. An example is a multi-pass left-to-right strategy [Alb81]. If, on the other hand, the attribute grammar specification is analyzed and the computed dependencies are used to construct an evaluator, the evaluator uses a flexible strategy. For example, for a so-called *ordered attribute grammar* [Kas80] there exists a particularly efficient static evaluator whose strategy is determined at generation time.

3.2 The BOX grammar

We will now present a second running example, the BOX grammar. Unlike the VARUSE grammar it only distantly resembles a traditional compiler. It is inspired by Donald Knuth's TEX system [Knu91].

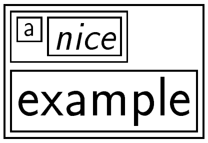
3.2.1 Single visit

The basic building block of the BOX grammar is a box containing text in some font. Boxes can be stacked on top of each other or placed side by side giving new boxes. A structure of boxes is to be translated into a display list containing for each text its position and font.

The following abstract syntax tree will figure as the (standard) application of the BOX grammar.

$$\begin{array}{lll} \textbf{stackl} & (& \textbf{sidet}(\textbf{box}(\texttt{'small'},\texttt{'a'}),\textbf{box}(\texttt{'emph'},\texttt{'nice'})) \\ & , & \textbf{box}(\texttt{'large'},\texttt{'example'}) \\ &) & \end{array}$$

The box application translates to the following display list, assuming that the upper left corner of the outer box is to be placed at $(0,0)$ and supposing some font metrics and mappings.



(0,0)	'helv-8pt'	'a'
(4,0)	'helv-italic-10pt'	'nice'
(0,11)	'helv-14pt'	'example'

Converting a display list into bitmaps of the right font and size is left to a separate interpreter. The translator of the BOX grammar only knows the font metrics so that the sizes of the boxes can be determined; just like TEX converts a `tex` file into a `dvi` file and an appropriate driver converts the `dvi` file into bitmaps for an output device.

The name display *list* is somewhat misleading since we use a more convenient display *tree* represented by D. Note that D is not part of the context-free grammar, although it uses the same syntax. It describes a type, just like the pseudo terminal $real$ describes real numbers, except that it is defined by the grammar writer.

$$\begin{aligned} D &= \mathbf{join}(D, D) \\ &\mid\ \mathbf{at}(\mathit{real}, \mathit{real}, \mathit{font}, \mathit{str}) \end{aligned}$$

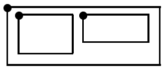
Primitive type *real* denotes floating point numbers and *str* represents strings. A *font* is a name, such as 'helv-italic-10pt', that is known to the display list interpreter. In the input specification, fonts are not referenced explicitly but rather by a symbolic name (*fname*) like 'emph' for emphasize. A font library or "style file" maps symbolic names to fonts: *Lib* is the type $\mathit{fname} \longrightarrow \mathit{font}$. In the translation of the box application above, we assumed a font library *flib* such that *findfont flib* 'emph' = 'helv-italic-10pt'. Two other font functions are available, namely *strwidth* and *strheight* that return the width respectively height of a string in a given font. For example *strheight* 'helv-italic-10pt' 'nice' = 11pt.

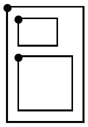
The context-free grammar for the BOX grammar is defined by the following declaration

$$\begin{aligned} B &= \mathbf{box}(\mathit{fname}, \mathit{str}) \\ &\mid\ \mathbf{sidet}(B, B) \\ &\mid\ \mathbf{stackl}(B, B) \quad . \end{aligned}$$

Non-terminal B has six attributes. Three attributes are inherited: *flib*, x and y supplying respectively the font library and the requested upper left corner. The other three are synthesized attributes. Attributes w and h report the width and height of the box and *dlist* represents its display list representation.

Figure 3.5$_\triangleright$ gives the specification of the BOX grammar. Two aspects are new. First of all, a *local* attribute is used in production *box*, namely $\mathit{box}.f$. By using it, we avoid writing, and probably evaluating, function *findfont* three times. Secondly, we *factorized* the common equations of *sidet* and *stackl*. The common parts include passing down the font library and the coordinates of the first child and synthesizing the joined display lists. Factorization allows for a modest form of modularization of attribute grammar specifications.





The equations for *sidet* define a box that consists of the box for the left child glued to the box for the right child. Hence the width is the sum of the widths and the height is the maximum of the heights. The boxes are top aligned. The *stackl* production puts the box for the left child above the box for the right child. These boxes are left aligned.

We will show in Chapter 5 that the BOX grammar is partitionable and that the following interface is valid in any context of B.

$$\Big[\ \Big(\{B.\mathit{flib}, B.x, B.y\}\ ,\ \{B.w, B.h, B.\mathit{dlist}\}\Big)\ \Big]$$

In other words, one visit suffices to decorate a B tree and the inherited attributes of the first visit are $B\mathbf{I}\,1 = \{B.\mathit{flib}, B.x, B.y\}$ whereas the synthesized attributes are $B\mathbf{S}\,1 = \{B.w, B.h, B.\mathit{dlist}\}$. Observe that $B.x$ is scheduled in slot 1 and $B.w$ is scheduled in slot 2. Thus, the partial order induced by this single visit interface includes the arc $B.x \rightarrow B.w$.

$$B{<}{\downarrow}\mathit{flib} : \mathit{Lib},\ {\uparrow}w, h : \mathit{real},\ {\downarrow}x, y : \mathit{real},\ {\uparrow}\mathit{dlist} : D{>}$$

$$\begin{array}{lll}
B_1 = & \mathbf{box}(\mathit{fname}, \mathit{str}) & \\
& \mathbf{local}\ f : \mathit{font}; & \\
& f & := \mathit{findfont}\ B_1.\mathit{flib}\ \mathit{fname}; \\
& B_1.w & := \mathit{strwidth}\ f\ \mathit{str}; \\
& B_1.h & := \mathit{strheight}\ f\ \mathit{str}; \\
& B_1.\mathit{dlist} & := \mathbf{at}(B_1.x, B_1.y, f, \mathit{str}) \\
\mid & \mathbf{sidet}(B_2, B_3),\ \mathbf{stackl}(B_2, B_3) & \\
& B_2.\mathit{flib} & := B_1.\mathit{flib}; \\
& B_3.\mathit{flib} & := B_1.\mathit{flib}; \\
& B_2.x & := B_1.x; \\
& B_2.y & := B_1.y; \\
& B_1.\mathit{dlist} & := \mathbf{join}(B_2.\mathit{dlist}, B_3.\mathit{dlist})
\end{array}$$

$$\begin{array}{lll}
\mid & \mathbf{sidet}(B_2, B_3) & \\
& B_1.w & := B_2.w + B_3.w; \\
& B_1.h & := B_2.h\ \mathbf{max}\ B_3.h; \\
& B_3.x & := B_1.x + B_2.w; \\
& B_3.y & := B_1.y \\
\mid & \mathbf{stackl}(B_2, B_3) & \\
& B_1.w & := B_2.w\ \mathbf{max}\ B_3.w; \\
& B_1.h & := B_2.h + B_3.h; \\
& B_3.x & := B_1.x; \\
& B_3.y & := B_1.y + B_2.h
\end{array}$$

Figure 3.5. The box grammar (part I)

$$\begin{array}{lll}
B_1 = & \mathbf{sideb}(B_2, B_3) & \\
& B_2.\mathit{flib} & := B_1.\mathit{flib}; \\
& B_3.\mathit{flib} & := B_1.\mathit{flib}; \\
& B_1.w & := B_2.w + B_3.w; \\
& B_1.h & := B_2.h\ \mathbf{max}\ B_3.h; \\
& \mathbf{local}\ b : \mathit{real}; & \\
& b & := B_1.y + B_1.h; \\
& B_2.x & := B_1.x; \\
& B_2.y & := b - B_2.h; \\
& B_3.x & := B_1.x + B_2.w; \\
& B_3.y & := b - B_3.h; \\
& B_1.\mathit{dlist} & := \mathbf{join}(B_2.\mathit{dlist}, B_3.\mathit{dlist})
\end{array}$$

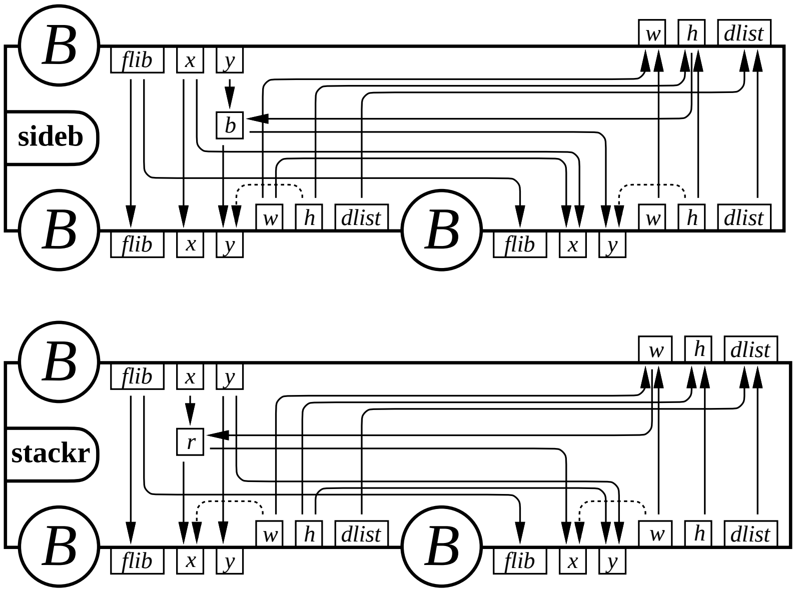
$$\begin{array}{lll}
\mid & \mathbf{stackr}(B_2, B_3) & \\
& B_2.\mathit{flib} & := B_1.\mathit{flib}; \\
& B_3.\mathit{flib} & := B_1.\mathit{flib}; \\
& B_1.w & := B_2.w\ \mathbf{max}\ B_3.w; \\
& B_1.h & := B_2.h + B_3.h; \\
& \mathbf{local}\ r : \mathit{real}; & \\
& r & := B_1.x + B_1.w; \\
& B_2.x & := r - B_2.w; \\
& B_2.y & := B_1.y; \\
& B_3.x & := r - B_3.w; \\
& B_3.y & := B_1.y + B_2.h; \\
& B_1.\mathit{dlist} & := \mathbf{join}(B_2.\mathit{dlist}, B_3.\mathit{dlist})
\end{array}$$

Figure 3.6. The box grammar (part II)

$$\begin{array}{l}
R{<}{\uparrow}\mathit{dlist} : D{>} \\
H{<}{\uparrow}\mathit{flib} : \mathit{Lib},\ {\uparrow}w, h : \mathit{real}{>}
\end{array}$$

$$\begin{array}{lll}
R = & \mathbf{root}(H, B) & \\
& B.\mathit{flib} & := H.\mathit{flib}; \\
& B.x & := H.w/2 - B.w/2; \\
& B.y & := H.h/2 - B.h/2; \\
& R.\mathit{dlist} & := B.\mathit{dlist}
\end{array}$$

Figure 3.7. The box grammar (part III)



The dashed arrows describe a dependency of an inherited attribute on a synthesized attribute of the same non-terminal.

Figure 3.8. The production icons of *sideb* and *stackr*

```
plan box
begin 1
  eval(f);
  eval(B1.w);
  eval(B1.h)
end 1
begin 2
  eval(B1.dlist)
end 2
```

```
plan sidet
begin 1
  eval(B2.flib);
  visit(B2, 1);
  eval(B3.flib);
  visit(B3, 1);
  eval(B1.w);
  eval(B1.h)
end 1
begin 2
  eval(B2.x);
  eval(B2.y);
  visit(B2, 2);
  eval(B3.x);
  eval(B3.y);
  visit(B3, 2);
  eval(B1.dlist)
end 2
```

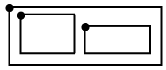
```
plan stackl
begin 1
  eval(B2.flib);
  visit(B2, 1);
  eval(B3.flib);
  visit(B3, 1);
  eval(B1.w);
  eval(B1.h)
end 1
begin 2
  eval(B2.x);
  eval(B2.y);
  visit(B2, 2);
  eval(B3.x);
  eval(B3.y);
  visit(B3, 2);
  eval(B1.dlist)
end 2
```

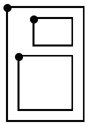
```
plan sideb
begin 1
  eval(B2.flib);
  visit(B2, 1);
  eval(B3.flib);
  visit(B3, 1);
  eval(B1.w);
  eval(B1.h)
end 1
begin 2
  eval(b);
  eval(B2.x);
  eval(B2.y);
  visit(B2, 2);
  eval(B3.x);
  eval(B3.y);
  visit(B3, 2);
  eval(B1.dlist)
end 2
```

```
plan stackr
begin 1
  eval(B2.flib);
  visit(B2, 1);
  eval(B3.flib);
  visit(B3, 1);
  eval(B1.w);
  eval(B1.h)
end 1
begin 2
  eval(r);
  eval(B2.x);
  eval(B2.y);
  visit(B2, 2);
  eval(B3.x);
  eval(B3.y);
  visit(B3, 2);
  eval(B1.dlist)
end 2
```

Figure 3.9. The visit-sub-sequences of the BOX grammar

3.2.2 Two visits





To complicate matters, we will now add two productions: *sideb* and *stackr*. They are analogue to *sidet* respectively *stackl*. However the *b*ottoms respectively *r*ight sides of the boxes are aligned instead of the *t*ops respectively *l*eft sides. See Figure 3.6$_\triangleleft$ for the description of these two productions. Notice that the specification in Figure 3.5$_\triangleleft$ is completely self-contained. The two productions in Figure 3.6$_\triangleleft$ can be regarded as a separate module that may or may not be "linked".

Equation $b := B_1.y + B_1.h$ in *sideb* (Figure 3.5$_\triangleleft$) is not in Bochmann normal form since it uses the output attribute occurrence $B_1.h$. As noted before, substitution of the output attribute occurrences by their defining equations normalizes the equation: $b := B_1.y + (B_2.h \ \mathbf{max}\ B_3.h)$.

Why is the extended BOX grammar more complex than the short one? The crux is in the dashed arrows in the production icons for *sideb* and *stackr* displayed in Figure 3.8$_\triangleleft$: an inherited attribute depends on a synthesized attribute of the same non-terminal. A *stackr* node introduces a context in which the partial order introduced by the single-visit interface given previously is in conflict with the actual dependencies. To be concrete, the dashed arrow in $dpr(stackr)$ denotes the dependency $B.w \rightarrow B.x$ which violates the interface.

Nevertheless, the extended BOX grammar is still partitionable. As will be shown in Chapter 5, the following interface is valid in any context. Possible visit-sequences adhering to this interface are presented in Figure 3.9$_\triangleleft$.

$$\Big[\ \big(\{B.\mathit{flib}\}\ ,\ \{B.w, B.h\}\big)\ ,\ \big(\{B.x, B.y\}\ ,\ \{B.\mathit{dlist}\}\big)\ \Big]$$

Note that $\mathbf{v}B = 2$. We observe that arcs from a synthesized attribute to an inherited attribute of the same non-terminal force additional visits.

3.2.3 The header

Finally, the BOX grammar is extended with two non-terminals. Non-terminal R is the root of the extended grammar. It derives a "header" H and a box B. Headers are not specified any further, but the idea is that they define a font library and a "paper size". The equations for the *root* production map the center of the box to the center of the paper. These ideas are formalized by the fragment in Figure 3.7$_\triangleleft$. For the remainder of this thesis, the BOX grammar comprises all three parts.

3.3 Visit-sequence based evaluators

This section discusses how visit-sequences can be used to construct a plain or incremental attribute evaluator. The BOX grammar illustrates the pitfalls of incremental evaluation.

3.3.1 Plain evaluation

Several methods exist to implement a visit-sequence evaluator. Algorithm 3.10 sketches one that is not obscured by an excess of tables. DECORATE is based on a single table $plan$ that encodes visit-sequences in the structured style. Visit-*sub*sequence number s of production p is retrieved by an expression of the form $plan[p, s]$. The number of instructions in this subsequence is determined via $length(plan[p, s])$, and the individual instructions are retrieved by indexing the subsequence: $plan[p, s][i]$.

Attribute instances are attached to tree nodes. To decorate an abstract syntax tree T, one must first set the inherited attributes of (the root of) T to the appropriate values, and then call $decorate(root(T), 1)$. When the procedure returns, all attribute instances of T have been assigned a correct value, including the synthesized attributes of (the root of) T.

```
proc $decorate(K : Node;\ segment : Integer)$
  var $index : Integer;$
  begin
    for $index := 1$ to $length(plan[prod(K), segment])$ do
      case $plan[prod(K), segment][index]$ of
        $eval(\alpha)$    : compute $ins(K, \alpha)$
        $visit(i, v)$ : $decorate(i : K, v)$
      esac
    rof
  end
```

Algorithm 3.10. Attribute evaluator DECORATE

3.3.2 Incremental evaluation

Plans induce a total order on the attribute instances of an abstract syntax tree. When a tree is evaluated the $eval$ instructions are executed respecting all direct and indirect dependencies. Therefore, the plans also represent an acceptable *re*evaluation order for the attribute instances of a production instance. The basic idea behind incremental evaluator REDECORATE (Algorithm 3.11$_{\triangleright}$) is to use the plans as during non-incremental evaluation except that as many instructions as possible should be skipped. In particular, skipping $visit$ and $suspend$ is profitable: in unit time arbitrarily many reevaluations can be avoided.

Incremental evaluator REDECORATE is based on the plain evaluator DECORATE. REDECORATE passes through three phases as recorded in variable $phase$. The first phase initializes the evaluator state, the second phase deals with the redecoration and in the third phase no changes need to be propagated anymore.

The first phase initializes the state of the evaluator. Let T and T' be two abstract syntax trees such that T' is obtained from T by subtree replacement at node S. The initialization phase brings the evaluator stack in the same configuration

```
var
  $phase : int = 1$
  $S : Node =$ “root of the replaced subtree”
proc $update(\sigma)$
  begin
    if $\sigma$ is newborn
      then compute and mark $\sigma$
      else if any predecessor of $\sigma$ is marked
        then compute $\sigma$ and mark it if it changes
        else skip
    fi fi
  end
proc $redecorate(K : Node;\ segment : Integer)$
  var $index : Integer$
  begin
    for $index := 1$ to $length(plan[prod(K), segment])$ do
      case $plan[prod(K), segment][index]$ of
        $eval(\alpha)$    : if $phase = 2$ then $update(ins(K, \alpha))$ fi
        $visit(i, v)$ : if $K = S$ then $phase := 2$ fi
                      $redecorate(i : K, v)$
      esac
    rof
  end
```

Algorithm 3.11. Incremental attribute evaluator REDECORATE

as DECORATE has when it first reaches node S. As pointed out by Yeh [Yeh83], this is required to start incremental decoration at node S. The configuration is reached by simply following the linear order, while skipping the *eval* instructions, until root S of the replaced subtree is reached. The time for sequence control (*visit*s) is insignificant compared with time for decoration (*eval*s) [YK88]. This explains why simply following the linear order is preferred to an optimal algorithm that sets up the stack [Yeh83].

During the second phase, the real work is done; the *eval* instructions are no longer skipped a priori. Attribute instances are marked to reflect the fact that their value in T' is different from their value in T. The decision to recompute an attribute instance is now as follows, either (i) one of the predecessors is marked (changed) or (ii) the instance is attached to the new subtree: it is *newborn*. The newly computed value of an attribute instance is compared with the old value in order to decide whether the instance should be marked.

A mark in the naive dynamic change propagation algorithm discussed earlier meant that the attribute instance needed to be evaluated. Here it means that it has been evaluated and found out to have changed.

The evaluator is in phase three once it is outside the area of marked attribute instances. Every *update* issued by an *eval* instruction will fall through to the *skip* branch. One may device all kinds of strategies [Yeh83] that deal with detecting the end of change propagation so that REDECORATE can be aborted. Informally this means setting variable *phase* to 3. No such strategies have been incorporated in REDECORATE.

3.3.3 Skipping visits

A interesting improvement for REDECORATE (Algorithm 3.11) is suggested by Reps and Teitelbaum [RT89] and Alblas [Alb89]. They aim at skipping *visit* and *suspend* instructions during phase two. In order to appreciate their ideas let us have a look at three examples *landscape*, *reduce* and *bold*. They are based on the BOX grammar.

Figure 3.12$_\triangleright$ shows the abstract syntax tree for the box application. We assume that the header in the box application specifies as paper size 100×200 and as font library the following mapping

'small' $\rightarrow$ 'helv-8pt'
'emph' $\rightarrow$ 'helv-italic-10pt'
'large' $\rightarrow$ 'helv-14pt' .

The plan tree associated with the box application is given in Figure 3.13$_\triangleright$. It is composed from the plans given in Figure 3.9$_{\triangleleft 47}$; the plan tree for the header is left unspecified (grey).

The three examples that will now be presented all deal with an edit operation on the box application. In each case, the header is modified, which is reflected by a change in the inherited attributes of B. We discuss which visits may be skipped.

Landscape

The first example of an edit operation illustrates that a (first) visit may be skipped if the inherited attributes for that visit are unchanged.

We assume a subtree replacement in the header that defines "landscape" paper 200×100. Consequently, redecoration changes phase from 1 to 2 while visiting the header. Since the display list must reflect the changed paper, redecoration (phase two) is still in progress when the plan for $root$ is reached again. Consequently, instruction $eval(B.flib)$ (labeled **a** in Figure 3.13$_\triangleright$) will issue an $update$. Since $H.flib$, the only predecessor of $B.flib$, is not changed by the assumed subtree replacement, $B.flib$ will not be (recomputed and thus not be) marked. Observe that since none of the inherited attribute instances for the first visit to child B is marked, $visit(B, 1)$ (labeled **b**) can be skipped! Redecoration proceeds by executing $eval(B.x)$ (labeled **c**). This attribute instance does change—it depends on the changed $H.h$—so that the second visit to B (labeled **e** in Figure 3.13$_\triangleright$) can not be skipped.

Visit 1 to B can be skipped because its inherited attribute instances (instances of $B\mathbf{I}\,1$) are unmarked. Unfortunately, this rule only applies to the *first* visit of a tree as is illustrated in the second example.

Reduce

The second example shows that visit 2 to B can not be skipped even though all its inherited attribute instances (instances of $B\mathbf{I}\,2$) are unchanged. The reason for

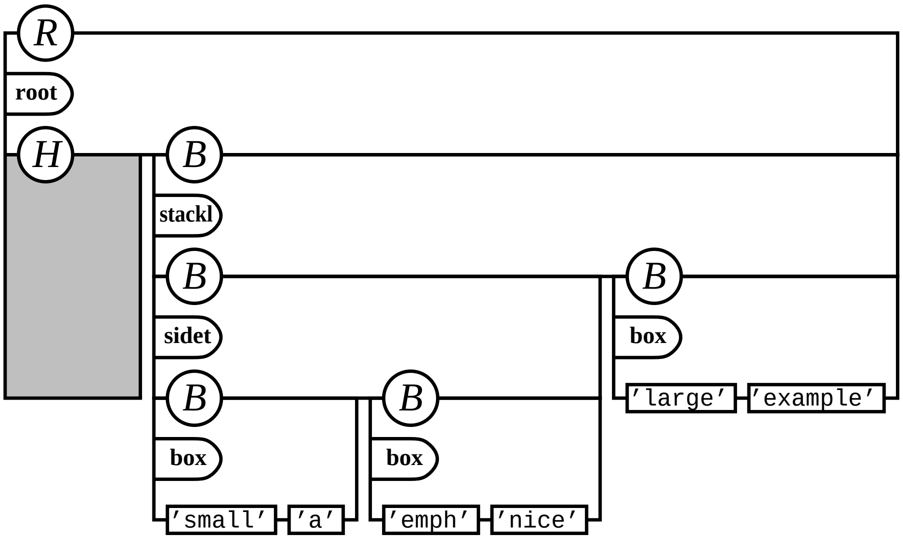
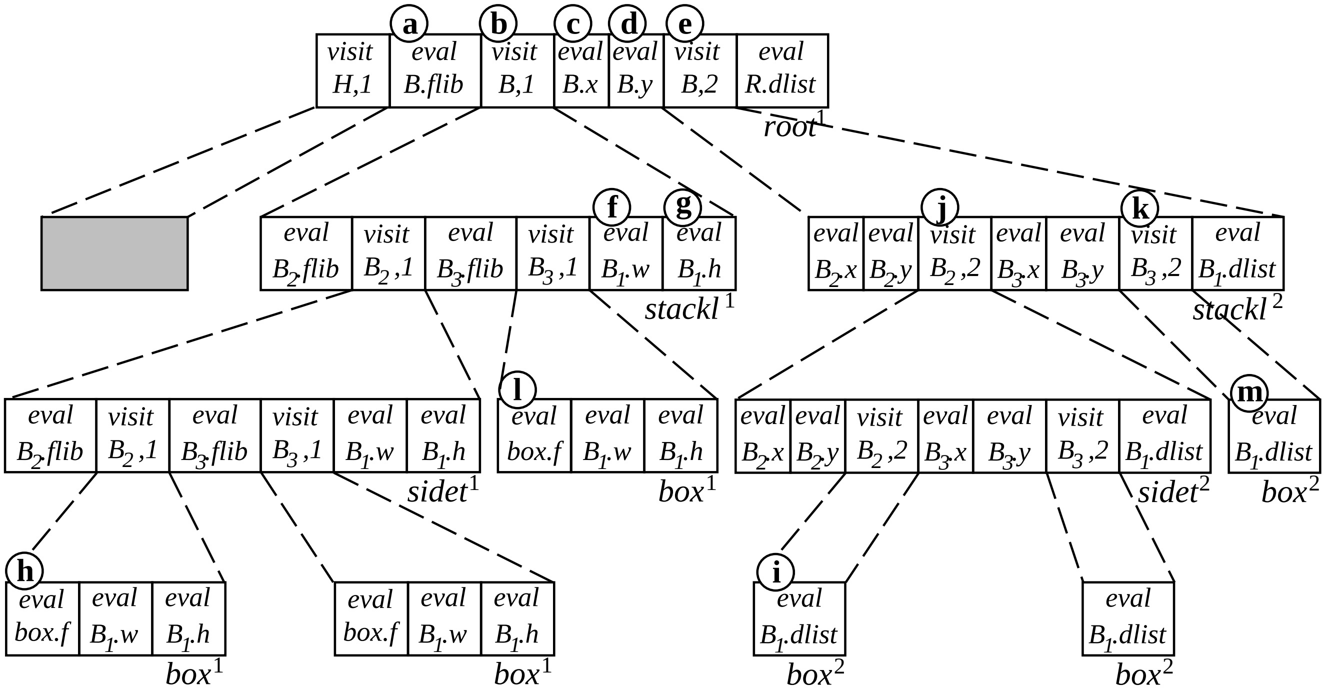


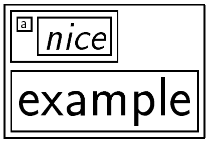
Figure 3.12. The abstract syntax tree for the box application



The **bold** labels under the lower right corner of each visit-sub-sequence identify the production instance; the superscript denotes the subsequence number. The circled letters label instructions that are referred to in the examples "landscape", "reduce" and "bold".

Figure 3.13. The plan tree for the box application

this are intra-visit-dependencies.



Instead of changing the paper size, we now assume a subtree replacement in the header that changes the font library. The symbolic name 'small' is no longer mapped to 'helv-8pt', we let it map to a reduced font, say 'helv-5pt'. Like before, redecoration is in phase two when the plan for $root$ is reached again. Unlike the previous example, $eval(B.flib)$ causes $B.flib$ to be computed and marked. The first visit to B can not be skipped.

Since the height of 'nice' still dominates the height of 'a', and the width of 'example' still dominates the width of the joined boxes 'a' and 'nice', the size of the entire structure does not change. Consequently, $B.h$ and $B.w$ in the $root$ production (labeled **f** and **g** in Figure 3.13) remain unchanged (are unmarked) so that $B.x$ and $B.y$ (labeled **c** and **d**) need not be recomputed. In other words, the inherited attribute instances for the second visit to B do not change value.

However, since the *changed* font library, passed down in the first visit, is also used in the second visit the second visit can not be skipped. More specific, the change in the font library causes instruction $eval(box.f)$ labeled **h** to be executed. The value of local attribute instance f changes from 'helv-8pt' to 'helv-5pt'. In the second visit, instruction $eval(B.dlist)$ labeled **i** refers to the new value and must therefore be executed. The dependency $box.f \rightarrow B_1.dlist$ is a so-called *intra-visit-dependency*. Intra-visit-dependencies play an important role in the next section where visit-sequences are converted into *visit-functions*.

Incremental evaluator VSPROPAGATE, introduced by Reps and Teitelbaum [RT89], avoids wasting $visit$s and $suspend$s by keeping track of the production instances that contain changed attribute instances. VSPROPAGATE uses an additional state variable "reactivated" that records those production instances. Whenever an attribute instance is marked, the associated production instance is reactivated. Transfers to reactivated nodes can never be skipped. In other words, visit v to a tree with root N may be skipped as long as *all* the inherited attribute instances used by visit v and all previous ones ($\bigcup_{1\leq w\leq v} N\mathbf{I}\,w$) are not marked. Once an inherited attribute for visit u changes, the attribute instance is marked and the tree node reactivated so that all visits v, $u \leq v \leq \mathbf{v}N$ must be executed.

Care must be taken with syntactic references.

Bold

The edit action described in the last—rather complex—example, changes the inherited attributes of the first visit, but not the inherited attributes of the second visit. Although the previous example showed that, in general, one may not skip a visit if a previous visit has been executed, in specific cases—like this one—one may. In other words, VSPROPAGATE is pessimistic and may be refined.

The current example strongly resembles the previous one; the symbolic name 'large' is remapped instead of 'small'. The subtree replacement changes the mapping for 'large' from 'helv-14pt' to 'helv-bold-14pt'. We suppose that the font metrics of 'helv-14pt' and 'helv-bold-14pt' do not differ.

The first visit to B (label **b** in Figure 3.13) can not be skipped since its inherited attribute, the font library $B.\mathit{flib}$ labeled **a**, is changed. Because we supposed equal font metrics for the old and the substituted font, the sizes of the boxes as computed by the first visit, do not change. So, like in the previous example, the inherited attribute instances (labels **c** and **d**) for the second visit do not change. The second visit to B (label **e**) can not be skipped since the change in the font library must still be induced in the display list.

The superscript 2 in stackl^2 denotes the second subsequence of stackl.

We have now traced incremental update up to the second subsequence of the stackl node which is labeled stackl^2 in Figure $3.13_{\triangleleft}$. The left and right subtree of this node, $\mathbf{sidet}(\mathbf{box}(\texttt{'small'}, \texttt{'a'}), \mathbf{box}(\texttt{'emph'}, \texttt{'nice'}))$ and $\mathbf{box}(\texttt{'large'}, \texttt{'example'})$, will be visited for the second time: $\mathit{visit}(B_2, 2)$ (label **j**) respectively $\mathit{visit}(B_3, 2)$ (label **k**). The inherited attributes for both visits, the origins $B_2.x, B_2.y$ respectively $B_3.x, B_3.y$, are unchanged since they depend on the unchanged origin of the parent. The previous example learned that this does not mean that we may skip the visits: intra-visit-dependencies with the first visit may exist.

An arbitrary large tree "B_2" can be skipped as long as it does not use the changed 'large' font.

Nevertheless, we observe that $\mathit{visit}(B_2, 2)$ can be skipped whereas $\mathit{visit}(B_3, 2)$ can not! The reason for this is that during the decoration of B_2 no eval instruction refers to a marked attribute instance so they all lead to the skip branch anyhow. However, during the decoration of B_3 the marked attribute instance $\mathit{box}.f$ labeled **l** is referenced by the $\mathit{eval}(B_1.\mathit{dlist})$ instruction labeled **m**. This is the only instruction that actually changes the display list (reflecting the changed mapping for '`large`'); it may certainly not be skipped.

. . .

The first example suggested the rough idea of skipping visits if the associated inherited attribute instances have not changed. The second example learned that the observation does not hold in general: previous visits may have changed attribute instances that are referenced by later visits (intra-visit-dependencies). It is safe to skip visit v if all inherited attribute instances for visits $1 \ldots v$ have not been changed. The third example showed that this is a safe but pessimistic strategy. Visits may be skipped as long as they only refer to unchanged attribute instances. The next section will show how to implement this approach. Basically, a visit computes an additional structure called *binding* that holds the values of precisely those attribute instances that are computed during that visit and that are referenced by a later visit. If the binding has not changed, the later visit may be skipped.

3.3.4 Evaluation of higher-order grammars

Visit-sequences for a higher-order grammar G can be obtained by first reducing G to a plain grammar G' then computing visit-sequences vss' for G' and finally translating vss' to visit-sequences vss for G.

When applying a plain-grammar algorithm f' to a reduced higher-order grammar two aspects must be investigated. First of all, the reduced grammar G' is a pessimistic representation of G: f' will assume that derived dependencies between attributes on $A(X)$ might also occur between the generated attribute occurrences

```
plan $root$
begin 1
    $visit(L, 1)$;
    $eval(L.env)$;
    $visit(L, 2)$;
    $eval(S.code)$
end 1

plan $decl$
begin 1
    $visit(N, 1)$;
    $visit(L_2, 1)$;
    $eval(L_1.decs)$
end 1
begin 2
    $eval(L_2.env)$;
    $visit(L_2, 2)$;
    $eval(table)$;
    $eval(table.id)$;
    $visit(table, 1)$;
    $eval(L_1.code)$
end 2

plan $stat$
begin 1
    $visit(L_2, 1)$;
    $eval(L_1.decs)$
end 1
begin 2
    $eval(L_2.env)$;
    $visit(L_2, 2)$;
    $visit(N, 1)$;
    $eval(table)$;
    $eval(table.id)$;
    $visit(table, 1)$;
    $eval(L_1.code)$
end 2

plan $empty$
begin 1
    $eval(L.decs)$
end 1
begin 2
    $eval(L.code)$
end 2

plan $none$
begin 1
    $eval(E.addr)$
end 1

plan $name$
begin 1
    $eval(N.id)$
end 1

plan $add$
begin 1
    $eval(E_2.id)$;
    $visit(E_2, 1)$;
    $eval(E_1.addr)$
end 1
```

Figure 3.14. The visit-sub-sequences of the higher-order variant of the VARUSE grammar

$O(p.x)$ if $R\ p.x = X$. Algorithm f' may reject G' on account of these induced dependencies. Since $p.x$ is defined by *computation* the higher-order trees that may be instantiated at $p.x$ are not arbitrary, nor are their dependencies. In other words, the reduction strategy rejects some higher-order grammars that would be accepted by a higher-order version f of f'. The question that must be answered is whether reduction does not narrow the class of higher-order grammars too much: whether

Assuming that f exists.

$$\{G \mid \text{reduced } G' \text{ is accepted by } f'\} \approx \{G \mid G \text{ is accepted by } f\} \quad .$$

The second question that must be raised is whether the results computed by the plain-grammar algorithm f' for the reduced grammar G' are applicable for the underlying higher-order grammar G. The first question is harder to answer than the second. The answer to the first question must be supported by experimental evidence; whereas the second problem requires mathematical reasoning.

Let us deal with these questions with respect to the computation of visit-sequences. The class of *ordered attribute grammars*, which is a subclass of partitionable attribute grammars (both are defined in Chapter 5), is a sufficiently large class for defining programming languages [Kas80]. There is an efficient algorithm, *Kastens ordered scheduling algorithm*, to compute visit-sequences for ordered attribute grammars. The class of *reduced ordered higher-order attribute grammars* contains those higher-order attribute grammars whose reduced grammar is an ordered grammar. Vogt expects that the reduced ordered higher-order grammars form a large enough class [VSK89]. Our experiments substantiate this expectation.

Higher-order attribute grammars are attribute grammars with attributable attributes. The higher-order variant of visit-sequences include an instruction to visit attributable attributes.

Before we address the second question, we define the instructions occurring in visit-sequences for higher-order attribute grammars.

instruction	**semantics**
$eval(\alpha)$	compute α and instantiate if it is an attributable attribute
$visit(i, v)$	visit $p\mathbf{o}i$ for the vth time
$visit(p.x, v)$	visit attributable attribute $p.x$ for the vth time
$suspend(v)$	exit visit v

We will now address the second question: can visit-sequences for a reduced grammar be converted into visit-sequences for the underlying higher-order grammar. Let G be a higher-order grammar and G' its reduced counterpart. Suppose production p of G contains an attributable attribute $p.x$ that is reduced to a local attribute $p.l_x$ and a non-terminal occurrence $p\mathbf{o}i_x$ in G'. The visit-sequence for p in G' will contain

- an equation for $p.l_x$
  $eval(p.l_x)$ is replaced by $eval(p.x)$;
- visits to $p\mathbf{o}i_x$
  every $visit(p\mathbf{o}i_x, v)$ is replaced by $visit(p.x, v)$;
- equations for the inherited attribute occurrences of $p\mathbf{o}i_x$
  every $eval(p\mathbf{o}i_x.a)$ is replaced by $eval(p.x.a)$;
- implicit uses of synthesized attribute occurrences of $p\mathbf{o}i_x$ in semantic functions every $\alpha := f \ldots p\mathbf{o}i_x.a \ldots$ is replaced by $\alpha := f \ldots p.x.a \ldots$.

Figure 3.14◁ shows the visit-sub-sequences of the higher-order variant of the VARUSE grammar obtained via reduction.

In the same way as visit-sequences for plain grammars can be used for incremental evaluation, visit-sequences for a higher-order attribute grammar form the basis of an incremental attribute evaluator for that higher-order grammar [VSK89]. However, multiple instances of a higher-order attribute are redecorated separately after a change to a higher-order attribute [TC90]. This leads to non-optimal incremental behavior. The next section discusses *visit-functions* that solve this problem.

3.3.5 An attribute grammar system

The key concepts for a classical attribute evaluator have been discussed. We will now synthesize the ingredients to obtain a *compiler generator*. A compiler generator reads an attribute grammars specification given in some formal language. There is not much consensus on the *source* language for compiler generators; many languages exist.

Once the grammar is read and parsed it must be checked: equations and semantic functions must be type correct; productions and non-terminals must be declared. Then, the grammar is analyzed and visit-sequences are generated.

Part 1: plans&driver

The generated visit-sequences, accompanied with an appropriate driver (DECORATE) form the main part of the generated compiler. Another important part deals with the data structures for storing the abstract syntax tree and structured

Part 2: data types

attributes. The generated compiler is either *batch*, or *interactive*. In the former case, a third module consists of a scanner and parser for the underlying context-free grammar. This module reads an application for the attribute grammar, constructs the associated abstract syntax tree and starts the attribute evaluator. The GAG system [KHZ82] is an example of this approach.

Part 3a: parser

In case of an interactive compiler, the third part of the generated system is a so-called *language based editor*. The editor module maintains an abstract syntax tree that is repeatedly edited by the user and then passed to the evaluator. Clearly, incremental evaluation makes sense here, so the plan module is extended with an incremental driver (REDECORATE).

Part 3b: editor

3.3.6 Background

Several methods exist to implement visit-sequence evaluators. Reps and Teitelbaum [RT89] use a complicated algorithm VSEVALUATE that requires two tables. The first table, indexed with a production, stores the plans. The second table maps the parameters from $visit$ and $suspend$ instructions to an index into the plan of a child respectively parent. In addition to this, each node records a reference to its parental tree and its child-index within that tree. The advantages of this scheme is two-fold. First of all, no state information needs to be stored at evaluation time. The (sizes of the) tables are fixed by the grammar. However, the storage space for nodes increases a little due to the parental reference. Moreover, this storage space does grow with the size of the abstract syntax tree. The second advantage of this scheme is that an incremental update can start anywhere in the tree; the tables will automatically follow the total order.

Yeh [Yeh83] uses an explicit stack to recapture the plan index after a visit to a child returns. Algorithm $3.10_{\triangleleft 49}$, DECORATE, is based on this approach. The advantage of DECORATE over VSEVALUATE is that the index table becomes redundant which makes the algorithm easier to understand. Furthermore, the memory occupied by the stack, which does grow with the depth of the abstract syntax tree, is less than the space occupied by the no longer needed parental references. On the other hand, it is harder to start an incremental update after subtree S of T has been changed: the stack must first be initialized to contain the path from the root of T to the root of S. An easy way to achieve this, is to start evaluation with the root of T thereby skipping $eval$ instructions until the root of S is reached, as in REDECORATE.

Vogts visit-functions [Vog93] resemble Yehs approach. However, instead of coding the visit-sequences in the $plan$ table, the instructions are inlined in the bodies of the visit-functions. Because the evaluation stack can not be controlled directly, incremental update can no longer be started in the middle of the tree. The stack is initialized by simulating plain evaluation. Where Yeh skips $eval$ instructions to reach the replaced subtree, Vogt skips $visit$ instructions. This is realized by memoizing the visit-functions. Vogts visit-functions will be described in the next section.

The incremental counterpart of VSEVALUATE in the Synthesizer Generator is VSPROPAGATE. It is based on the "reactivation" scheme discussed earlier. REDECORATE, the incremental version of DECORATE was introduced by Yeh and Kastens [Yeh83, YK88].

Incremental evaluation requires equality tests on attribute instances. The equality test may be expensive since attribute instances may be arbitrarily large structures such as symbol tables. In the Synthesizer Generator this has led to an optimization that eliminates the need for equality tests for a special class of semantic functions, the so-called one-to-one functions (most notably identity functions) [RT89] .

3.4 Visit-functions

The previous section described a classical attribute evaluator driven by visit-sequences. This section discusses a novel approach using *visit-functions* [VSK91, SV91, Vog93]. It is based on the following combination of ideas.

Visit-functions Attribute instances are computed by visit-functions. Such functions take as parameter a tree and a subset of inherited attributes of the root of that tree. They return a subset of the synthesized attributes. The entire evaluator consists of visit-functions that recursively call each other in order to decorate the tree.

Memoizing visit-functions In a conventional incremental treewalk evaluator a partially attributed tree can be considered as a very efficient cache—where caching is replaced by explicit navigation—for the *semantic* functions. The tree serves as data flow driver and as attribute cache. Pugh [Pug88] separates the two roles of the abstract syntax tree by introducing a separate cache for the semantic functions. Vogt [VSK91, Vog93] caches the visit-functions instead. This is efficient, since a cache hit for a visit-function means that an entire visit to an arbitrarily large tree can be skipped. Furthermore, a visit-function may return the results of several semantic functions at the same time. Observe that, except for the cache, no administration is needed to record whether attributes have changed and further visits are necessary.

Memoizing constructors Since attributes may be found in the cache, there is no longer need to store them in the tree. This allows us to share multiple instances of the same tree. Memoization of tree constructors [TC90] is an elegant way to achieve sharing. A memoized constructor is also known as a *hashing cons* [Hug85]. Sharing not only reduces space consumption, it also allows for fast equality tests between terms: a pointer comparison suffices. Attribute values may be large structures too (symbol tables). Therefore, the constructors for user defined types are also shared. This solves the problem of expensive attribute equality test during incremental evaluation and huge space consumption due to multiple instances of the same attribute value in an abstract syntax tree.

Bindings The above ideas are complicated by intra-visit-dependencies: attribute instances computed in an earlier visit may have to be preserved for use in later visits. For standard evaluators, this is no problem since attributes are stored in the tree. In a functional setting these values must be passed explicitly to the future visits. Each visit-function therefore not only computes synthesized attributes but also *bindings* for subsequent visits. Bindings computed by earlier visits are passed as parameters to later visits.

Vogt first used standard visit-sequences to evaluate higher order attribute grammars incrementally [VSK89]. However, a standard evaluator decorates each instance of a higher-order attribute separately, leading to non-optimal incremental behavior after a change to a higher-order attribute [TC90]. This problem can be solved by using memoized visit-functions instead of visit-sequences [VSK91, SV91, Vog93]. Observe that a changed higher-order attribute is redecorated only once because all other instances cause the visit-function to find its previously computed result in the cache (assuming that each instance has the same inherited attributes).

3.4.1 Mapping grammars to functional programs

Attribute grammars and functional languages [BW88] are closely related [Joh87]. Explicit sequencing is completely absent in both formalisms and the description of the data flow is purely functional. Garbage collection needs not to be made explicit as in more imperative settings (assignments).

On some accounts the formalisms differs. When compared to functional programs, attribute grammar specifications are "algorithm free"; they only specify attribute dependencies, and no recursive functions. It is "easier" to write a grammar specification. On the other hand, the attribute grammar formalism usually lacks the powerful mechanisms of polymorphism and higher order functions. Higher order attribute grammars [Vog93] are a first attempt to close this gap. Polymorphism in attribute grammars has not received much attention yet—the elegant system [Aug93] being an exception. Almost all typed functional languages (for example MIRANDA [Tur85], GOFER [Jon91], HASKELL [HF92, HPJW^{+}92]) support polymorphism and modularity supporting constructs like type and constructor classes. The impact of these innovations in functional languages is such, that similar extensions to the attribute grammar formalism deserve more attention.

We will introduce three mappings from attribute grammar specifications onto functional programs. Three small grammars will be used to illustrate the mappings. The first mapping, CIRC [KS87], requires lazy evaluation. It constructs definitions of the form $(\ldots, x, \ldots) = f\ \ldots\ x\ \ldots$ which appear to be circular. The second mapping, VSS, eliminates the need for this laziness and constructs strict functions by breaking up decorations into multiple visits. The functions created by VSS are

Here, "strict" means that all the arguments of a function must be known in advance; more specifically they may not depend on the functions result. This is a liberal interpretation of the usual definition.

generally not side-effect free due to intra-visit-dependencies. Functions with side-effects can not be memoized. The third mapping, FUN, which will be introduced later, uses bindings to eliminate the need for side-effects.

We will now present the examples, *minimum*, *replace* and *repmin*. The former two are the constituents of the latter, as the name suggest. Bird introduced the repmin example [Bir84].

Minimum

The first example introduces the concept of mapping a grammar to a function. The resulting functions are known as *visit-functions*.

A synthesized-only attribute grammar can easily be mapped onto a functional program. In synthesized-only grammars, the synthesized attributes are recursively defined over the abstract syntax tree. The grammar maps to a top-down recursive function with one argument, an abstract syntax tree T, whose result is the values of the synthesized attributes of the root of T. Such functions have a primitive recursive scheme; they correspond to homomorphisms from the term algebra to some other domain.

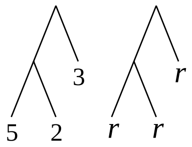
As an example, consider the grammar in Figure 3.15a that computes the minimum element of a simple binary tree labeled with integers. The corresponding homomorphism f is given in Figure 3.15b. Such a function is called a *visit-function* and since it is a visit-function on the data type T we usually write $visit_T$. Note that a visit-function for T uses *pattern matching* of constructors (productions) on T. The *pattern variables* correspond to the right-hand side non-terminals of p. We overload the symbols for the data types, such as T and int, to also denote pattern variables. Figure 3.15c shows the way we will present visit-functions.

f is also known as the *catamorphism* $(\![id, \mathbf{min}]\!)$.

Replace

The second example defines and illustrates the CIRC mapping, applicable to any grammar.

The mapping CIRC from grammars to functional programs [KS87] is as follows. For each non-terminal X a function $visit_X$ is written. It takes a tree of type X and an additional argument for each attribute in $A_{inh}(X)$. The result of $visit_X$ is a tuple of the synthesized attributes $A_{syn}(X)$. For each production p on X, the visit function on X has an alternative definition, trivially depending on the equations $E(p)$. The visit-functions generated by the CIRC mapping can handle arbitrarily non-circular grammars [Joh87, KS87].



We will examine a simple grammar to which we can apply CIRC. We are required to compute a tree with the same shape as the abstract syntax tree except that the tips are to be replaced with a given value r. Figure 3.16a gives the attribute grammar. Note that T has an inherited attribute r. For the visit-function, given in Figure 3.16b, this maps to an additional argument.

$$\begin{array}{l}
T\!<\uparrow \mathit{min} : \mathit{int}> \\[4pt]
T_1 = \mathbf{tip}(\mathit{int}) \\
\qquad T_1.\mathit{min} := \mathit{int} \\
\quad \mid \ \mathbf{fork}(T_2, T_3) \\
\qquad T_1.\mathit{min} := T_2.\mathit{min}\ \mathbf{min}\ T_3.\mathit{min}
\end{array}$$

a. The grammar

$$\begin{array}{lcl}
\multicolumn{3}{l}{f :: T \longrightarrow \mathit{int}} \\[4pt]
f\ \mathbf{tip}(i) & = & i \\
f\ \mathbf{fork}(l, r) & = & f\ l\ \mathbf{min}\ f\ r
\end{array}$$

b. The visit-function

$$\begin{array}{lcl}
\multicolumn{3}{l}{\mathit{visit}_T :: T \longrightarrow \mathit{int}} \\[4pt]
\mathit{visit}_T\ \mathbf{tip}(\mathit{int}) & = & \mathit{int} \\
\mathit{visit}_T\ \mathbf{fork}(T_2, T_3) & = & \mathit{visit}_T\ T_2\ \mathbf{min}\ \mathit{visit}_T\ T_3
\end{array}$$

c. Rewritten visit-function

Figure 3.15. The minimum example

$$\begin{array}{l}
T\!<\downarrow r : \mathit{int};\ \uparrow \mathit{new} : T> \\[4pt]
T_1 = \mathbf{tip}(\mathit{int}) \\
\qquad T_1.\mathit{new} := \mathbf{tip}(T.r) \\
\quad \mid \ \mathbf{fork}(T_2, T_3) \\
\qquad T_2.r \quad := T_1.r; \\
\qquad T_3.r \quad := T_1.r; \\
\qquad T_1.\mathit{new} := \mathbf{fork}(T_2.\mathit{new}, T_3.\mathit{new})
\end{array}$$

a. The grammar

$$\begin{array}{lcl}
\multicolumn{3}{l}{\mathit{visit}_T :: T \times \mathit{int} \longrightarrow T} \\[4pt]
\mathit{visit}_T\ \mathbf{tip}(\mathit{int})\ r & = & \mathbf{tip}(r) \\
\mathit{visit}_T\ \mathbf{fork}(T_2, T_3)\ r & = & \mathbf{fork}(\mathit{visit}_T\ T_2\ r, \mathit{visit}_T\ T_3\ r)
\end{array}$$

b. The visit-function

Figure 3.16. The replace example

$$\begin{array}{lll}
\multicolumn{3}{l}{R\!<\uparrow \mathit{new} : T>} \\
\multicolumn{3}{l}{T\!<\uparrow \mathit{min} : \mathit{int};\ \downarrow r : \mathit{int};\ \uparrow \mathit{new} : T>} \\[4pt]
\multicolumn{3}{l}{R\ = \mathbf{root}(T)} \\
\qquad & T.r & := T.\mathit{min}; \\
\qquad & R.\mathit{new} & := T.\mathit{new} \\
\multicolumn{3}{l}{T_1\ = \mathbf{tip}(\mathit{int})} \\
\qquad & T_1.\mathit{min} & := \mathit{int}; \\
\qquad & T_1.\mathit{new} & := \mathbf{tip}(T.r) \\
\multicolumn{3}{l}{\quad \mid \ \mathbf{fork}(T_2, T_3)} \\
\qquad & T_1.\mathit{min} & := T_2.\mathit{min}\ \mathbf{min}\ T_3.\mathit{min}; \\
\qquad & T_2.r & := T_1.r; \\
\qquad & T_3.r & := T_1.r; \\
\qquad & T_1.\mathit{new} & := \mathbf{fork}(T_2.\mathit{new}, T_3.\mathit{new})
\end{array}$$

a. The grammar

$$\begin{array}{lcl}
\multicolumn{3}{l}{\mathit{visit}_R :: R \longrightarrow T} \\
\multicolumn{3}{l}{\mathit{visit}_T :: T \times \mathit{int} \longrightarrow \mathit{int} \times T} \\[4pt]
\mathit{visit}_R\ \mathbf{root}(T) & = & T.\mathit{new} \\
 & & \quad \mathbf{where}\ (T.\mathit{min}, T.\mathit{new}) = \mathit{visit}_T\ T\ T.\mathit{min} \\[4pt]
\mathit{visit}_T\ \mathbf{tip}(\mathit{int})\ r & = & (\mathit{int}, \mathbf{tip}(r)) \\
\mathit{visit}_T\ \mathbf{fork}(T_2, T_3)\ r & = & (T_2.\mathit{min}\ \mathbf{min}\ T_3.\mathit{min}, \mathbf{fork}(T_2.\mathit{new}, T_3.\mathit{new})) \\
 & & \quad \mathbf{where}\ (T_2.\mathit{min}, T_2.\mathit{new}) = \mathit{visit}_T\ T_2\ r \\
 & & \qquad\qquad ;\ (T_3.\mathit{min}, T_3.\mathit{new}) = \mathit{visit}_T\ T_3\ r
\end{array}$$

b. The non-strict visit-functions

Figure 3.17. The repmin example

Repmin

Finally, the third example illustrates that the CIRC mapping generates seemingly circular functions. These kind of functions can not be evaluated in a conventional, call by value style.

The problem requires to compute a tree with the same shape as the abstract syntax tree except that the tips are to be replaced with the minimal tip value of the abstract syntax tree. This is a full blown example since it is essentially two pass. For an attribute grammar writer, this problem is not much harder than the previous two. In fact, the previous two grammars can simply be merged and extended with an additional $root$ production to specify the equation $T.r := T.min$, see (Figure 3.17a$_\triangleleft$).

This problem was originally introduced by Bird [Bir84] to illustrate how "one-touch" solutions can be obtained. The seemingly circular definition for $visit_R$ in Figure 3.17b$_\triangleleft$ is not straightforward; it makes essential use of lazy evaluation. Bird uses mathematically based rewrite techniques to obtain the given solution from an initial correct but less elegant set of functions. Kuiper and Swierstra [KS87] follow our approach: the functions in Figure 3.17b$_\triangleleft$ are obtained by applying CIRC to the grammar in Figure 3.17a$_\triangleleft$. The counterintuitive cyclic function is easy to understand since it is based on a simple decoration scheme, in which an inherited attribute occurrence happens to depend on a synthesized attribute occurrence.

. . .

The goal of Bird and Kuiper and Swierstra is to derive efficient and elegant functional programs. We, on the other hand, aim at mapping attribute grammars onto functional programs that can be executed efficiently and incrementally.

Strict functions are easily and efficiently implementable in imperative languages like C.

Lazy evaluation is needed to resolve the "circular" dependencies that arise when an inherited attribute depends on a synthesized attribute of the same non-terminal occurrence. We have encountered this situation before: when visit-sequences are generated such dependencies cause an additional visit. A former visit computes the synthesized attribute in question so that the depending inherited attribute can be computed before the second visit is started.

The VSS mapping maps *visit-sequences* to strict visit-functions. For each *visit* to a non-terminal, a separate visit-function is constructed. The VSS mapping is sketched in Figure 3.18. Figure 3.18a lists the visit-sub-sequences for the repmin grammar (given in Figure 3.17a$_\triangleleft$). Each *sub*-sequence induces an separate body for a visit-function. An $eval$ instruction maps to its associated equation and a $visit$ instruction (both plain and higher-order) maps to a recursive call. Figure 3.18b shows the resulting bodies. Reordering of these fragments yields the visit-functions given in Figure 3.18c. These functions are strict. The top level function $visit_R^1$ returns, for every abstract syntax tree R, the associated synthesized attribute $R.new$.

There is one major drawback in the VSS mapping: the resulting functions are generally not side-effect free. For some grammars, the constructed visit-functions are not functions in the mathematical sense; they rely on side-effects to resolve

```
plan root
begin 1
  visit(T, 1);
  eval(T.r);
  visit(T, 2);
  eval(R.new)
end 1

plan tip
begin 1
  eval($T_1$.min)
end 1
begin 2
  eval($T_1$.new)
end 2

plan fork
begin 1
  visit($T_2$, 1);
  visit($T_3$, 1);
  eval($T_1$.min)
end 1
begin 2
  eval($T_2$.r);
  visit($T_2$, 2);
  eval($T_3$.r);
  visit($T_3$, 2);
  eval($T_1$.new)
end 2
```

a. Sub-sequences

```
$visit_R^1$ root(T) = R.new
  where T.min  = $visit_T^1$ T
      ; T.r    = T.min
      ; T.new  = $visit_T^2$ T T.r
      ; R.new  = T.new

$visit_T^1$ tip(int) = $T_1$.min
  where $T_1$.min = int

$visit_T^2$ tip(int) $T_1$.r = $T_1$.new
  where $T_1$.new = tip($T_1$.r)

$visit_T^1$ fork($T_2$, $T_3$) = $T_1$.min
  where $T_2$.min = $visit_T^1$ $T_2$
      ; $T_3$.min = $visit_T^1$ $T_3$
      ; $T_1$.min = $T_2$.min min $T_3$.min

$visit_T^2$ fork($T_2$, $T_3$) $T_1$.r = $T_1$.new
  where $T_2$.r   = $T_1$.r
      ; $T_2$.new = $visit_T^2$ $T_2$ $T_2$.r
      ; $T_3$.r   = $T_1$.r
      ; $T_3$.new = $visit_T^2$ $T_3$ $T_3$.r
      ; $T_1$.new = fork($T_2$.new, $T_3$.new)
```

b. Sub-sequences ⇝ visit-functions

```
$visit_R^1$ :: R ⟶ T
$visit_R^1$ root(T) = R.new
  where T.min  = $visit_T^1$ T
      ; T.r    = T.min
      ; T.new  = $visit_T^2$ T T.r
      ; R.new  = T.new

$visit_T^1$ :: T ⟶ int
$visit_T^1$ tip(int) = $T_1$.min
  where $T_1$.min = int
$visit_T^1$ fork($T_2$, $T_3$) = $T_1$.min
  where $T_2$.min = $visit_T^1$ $T_2$
      ; $T_3$.min = $visit_T^1$ $T_3$
      ; $T_1$.min = $T_2$.min min $T_3$.min

$visit_T^2$ :: T × int ⟶ T
$visit_T^2$ tip(int) $T_1$.r = $T_1$.new
  where $T_1$.new = tip($T_1$.r)
$visit_T^2$ fork($T_2$, $T_3$) $T_1$.r = $T_1$.new
  where $T_2$.r   = $T_1$.r
      ; $T_2$.new = $visit_T^2$ $T_2$ $T_2$.r
      ; $T_3$.r   = $T_1$.r
      ; $T_3$.new = $visit_T^2$ $T_3$ $T_3$.r
      ; $T_1$.new = fork($T_2$.new, $T_3$.new)
```

c. Strict visit-functions

Figure 3.18. The VSS mapping: strict visit-functions for repmin

intra-visit-dependencies. Since functions with side-effects can not be memoized, a third mapping (using bindings) will be introduced: FUN. First we will investigate intra-visit-dependencies and bindings.

3.4.2 Intra-visit-dependencies and plan icons

The VARUSE grammar is an example of a grammar that VSS maps to visit-functions with side-effects. Note that the VARUSE grammar resembles the repmin grammar closely. Both synthesize tree-leaf information (taking the minimum in $T.min$ respectively concat in $L.decs$) on the first pass and distribute this information on the second pass (in $T.r$ respectively $L.env$). During the second pass, the "output" is generated at the leaves (using the distributed information) and combined and synthesized (in $T.new$ and $L.code$) by the internal nodes. However, there is one important aspect in the VARUSE grammar that causes side-effects in the visit-functions resulting from the VSS mapping: during the second visit to a $decl$ node, the attribute instance $N.id$—which is computed in the first visit—is referenced again. This is known as an *intra-visit-dependency*.

Definition 14 *Intra-visit-dependencies.*
An abstract syntax tree T has an intra-visit-dependency between visit v and w, $v < w$, if there is an attribute instance that is computed during visit v to T which is used during visit w to T.
□

Figure 3.19 gives the visit-functions obtained by applying the VSS mapping to the VARUSE plans (Figure 3.4$_{\triangleleft 41}$). The defining and a using occurrence of $N.id$ have been boxed. Note that the variable $N.id$ is not defined within the scope of $visit_L^2$; and thus it is a *free* variable. Some mechanism must make it accessible there so attribute instance $N.id$ in a $decl$ node can not be deleted when returning from the first visit. The conventional solution is to store the attribute instances in the tree, but this is a side-effect, frustrating the memoizability of the visit-functions. We will take a different approach. The later needed attribute instances are bundled and passed to the parent (via a newly introduced attribute) and we rely on the parent to pass them down (via another newly introduced attribute) for the next visit. The bundle of attribute instances passed from $visit_X^v$ to $visit_X^w$ via a parent is called a *binding* from v to w for non-terminal X. It is denoted by $X^{v \to w}$.

To help understand bindings, it is useful to present visit-sequences graphically with so-called *plan icons*. Plan icons should not be confused with production icons with which they have much in common. For example, compare the production icons in Figure 2.4$_{\triangleleft 24}$ with the plan icons in Figure 3.20. Production icons form a graphical specification of an attribute grammar. The use of plan icons lies mainly in the insight they provide in dependencies on attribute instances, most notably intra-visit-dependencies.

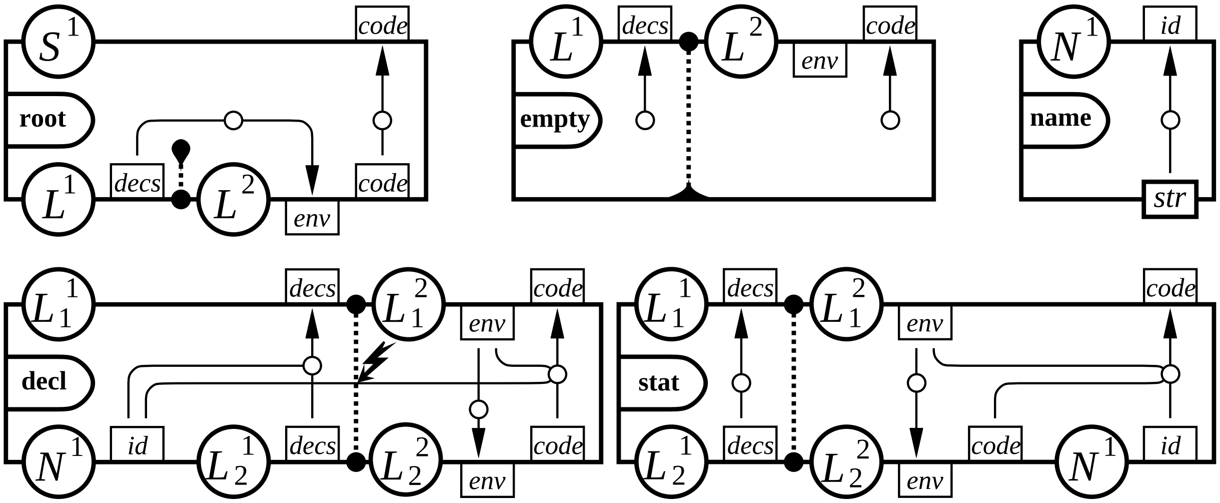
$$
\begin{aligned}
&\mathit{visit}_S^1 :: S \longrightarrow \mathit{Code}\\
&\mathit{visit}_S^1\ \mathbf{root}(L) = S.\mathit{code}\\
&\quad \mathbf{where}\ L.\mathit{decs} &&= \mathit{visit}_L^1\ L\\
&\qquad ;\ L.\mathit{env} &&= L.\mathit{decs}\\
&\qquad ;\ L.\mathit{code} &&= \mathit{visit}_L^2\ L\ L.\mathit{env}\\
&\qquad ;\ S.\mathit{code} &&= L.\mathit{code}\\
&\\
&\mathit{visit}_N^1 :: N \longrightarrow \mathit{str}\\
&\mathit{visit}_N^1\ \mathbf{name}(\mathit{str}) = N.\mathit{id}\\
&\quad \mathbf{where}\ N.\mathit{id} = \mathit{upstring}\ \mathit{str}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{visit}_L^1 :: L \longrightarrow \mathit{Env}\\
&\mathit{visit}_L^1\ \mathbf{decl}(N, L_2) = L_1.\mathit{decs}\\
&\quad \mathbf{where}\ \boxed{N.\mathit{id}} &&= \mathit{visit}_N^1\ N\\
&\qquad ;\ L_2.\mathit{decs} &&= \mathit{visit}_L^1\ L_2\\
&\qquad ;\ L_1.\mathit{decs} &&= N.\mathit{id} : L_2.\mathit{decs}\\
&\mathit{visit}_L^1\ \mathbf{stat}(N, L_2) = L_1.\mathit{decs}\\
&\quad \mathbf{where}\ L_2.\mathit{decs} &&= \mathit{visit}_L^1\ L_2\\
&\qquad ;\ L_1.\mathit{decs} &&= L_2.\mathit{decs}\\
&\mathit{visit}_L^1\ \mathbf{empty}() = L_1.\mathit{decs}\\
&\quad \mathbf{where}\ L_1.\mathit{decs} &&= [\,]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{visit}_L^2 :: L \times \mathit{Env} \longrightarrow \mathit{Code}\\
&\mathit{visit}_L^2\ \mathbf{decl}(N, L_2)\ L_1.\mathit{env} = L_1.\mathit{code}\\
&\quad \mathbf{where}\ L_2.\mathit{env} &&= L_1.\mathit{env}\\
&\qquad ;\ L_2.\mathit{code} &&= \mathit{visit}_L^2\ L_2\ L_2.\mathit{env}\\
&\qquad ;\ L_1.\mathit{code} &&= (-\mathit{lookup}\ L_1.\mathit{env}\ \boxed{N.\mathit{id}}\ )\\
&&&\quad : L_2.\mathit{code}\\
&\mathit{visit}_L^2\ \mathbf{stat}(N, L_2)\ L_1.\mathit{env} = L_1.\mathit{code}\\
&\quad \mathbf{where}\ L_2.\mathit{env} &&= L_1.\mathit{env}\\
&\qquad ;\ L_2.\mathit{code} &&= \mathit{visit}_L^2\ L_2\ L_2.\mathit{env}\\
&\qquad ;\ N.\mathit{id} &&= \mathit{visit}_N^1\ N\\
&\qquad ;\ L_1.\mathit{code} &&= (+\mathit{lookup}\ L_1.\mathit{env}\ N.\mathit{id})\\
&&&\quad : L_2.\mathit{code}\\
&\mathit{visit}_L^2\ \mathbf{empty}()\ L_1.\mathit{env} = L_1.\mathit{code}\\
&\quad \mathbf{where}\ L_1.\mathit{code} &&= [\,]
\end{aligned}
$$

Attribute occurrence $N.id$ (boxed) is *defined* in the first visit to a $decl$ instance and *used* in the second visit: a so-called intra-visit-dependency.

Figure 3.19. Visit-functions from the VSS mapping (with side-effects) for the VARUSE grammar



Attribute occurrence $N.id$ is *defined* in the first visit to a $decl$ instance and *used* in the second visit. The intra-visit-dependency is flagged with a lightning symbol.

Figure 3.20. The visit-sequences of the VARUSE grammar: plan icons

A plan icon is not simply a more detailed production icon: the lexical order of the children is lost; it is substituted by an order on the *visit* instructions.

Plan icons consist of several elements. The large rectangle denotes the production. Its name is given in the "flag" emanating from the left border. The small boxes denote the attribute occurrences, and the large boxes denote syntactic elements that are referred to by equations. These three elements have the same semantics as in production icons. Large and small discs and arrows also occur in production icons, but their semantics is slightly different. In plan icons, large discs denote *visits* to non-terminal occurrences instead of the non-terminal occurrences themselves. The arrows still denote dependencies, but in plan icons the emphasis is on the data flow of the *equations* which are denoted by small discs. A new element is a dashed vertical line enclosed by small black discs. It is called a *visit-border*. Visit-borders divide a production into *compartments*. Each compartment represents a visit-sub-sequence.

In roughly the same way that production icons can be pasted together to form an abstract syntax tree, plan icons can be pasted together. The resulting trees correspond to plan trees, although plan trees have been drawn differently until now. When combining plan icons into a plan tree, the small black discs hook visit-borders together, separating area's of the tree. Such a tree border starts with a black "head" (⧫) and ends with a black "anchor" (▲). Figure 3.21 shows the plan tree composed from plan icons for the VARUSE application. A time-axis flows through all attribute instances which have been "time-stamped". Compare this figure with Figures $3.2_{\triangleleft 40}$ and $3.3_{\triangleleft 40}$.

The following table lists the semantics of the various pictorial elements for production and plan icons.

element	**production icon**	**plan icon**
large rectangle	production	production
small box	attribute occurrence	attribute occurrence
large box	syntactic element	syntactic element
large disc	non-terminal occurrence	visit to non-terminal occurrence
arrow	dependency	source or destination of equation
small disc	semantic function	equation
dashed line	–	visit-border
black head	–	begin of border
black disc	–	border hook
black anchor	–	end of border

$L_2{}^1$ denotes the first visit to L_2. Therefore we do not write L_2^1.

Most of the elements in the plan icons are labeled. The name of the production is given in the flag. Attribute names are given in the respective boxes. A large disc is labeled with the non-terminal occurrence superscripted with the visit number. For example $L_2{}^1$ is the label corresponding with instruction $visit(L_2, 1)$. An equation is often not labeled, but easily identified since there is an arrow to its left-hand side attribute occurrence. Compartments (visit-sub-sequences) are also easily identified: each compartment has precisely one large parental disc associated with it. This disc represents a visit or rather $suspend$ to the parent. The

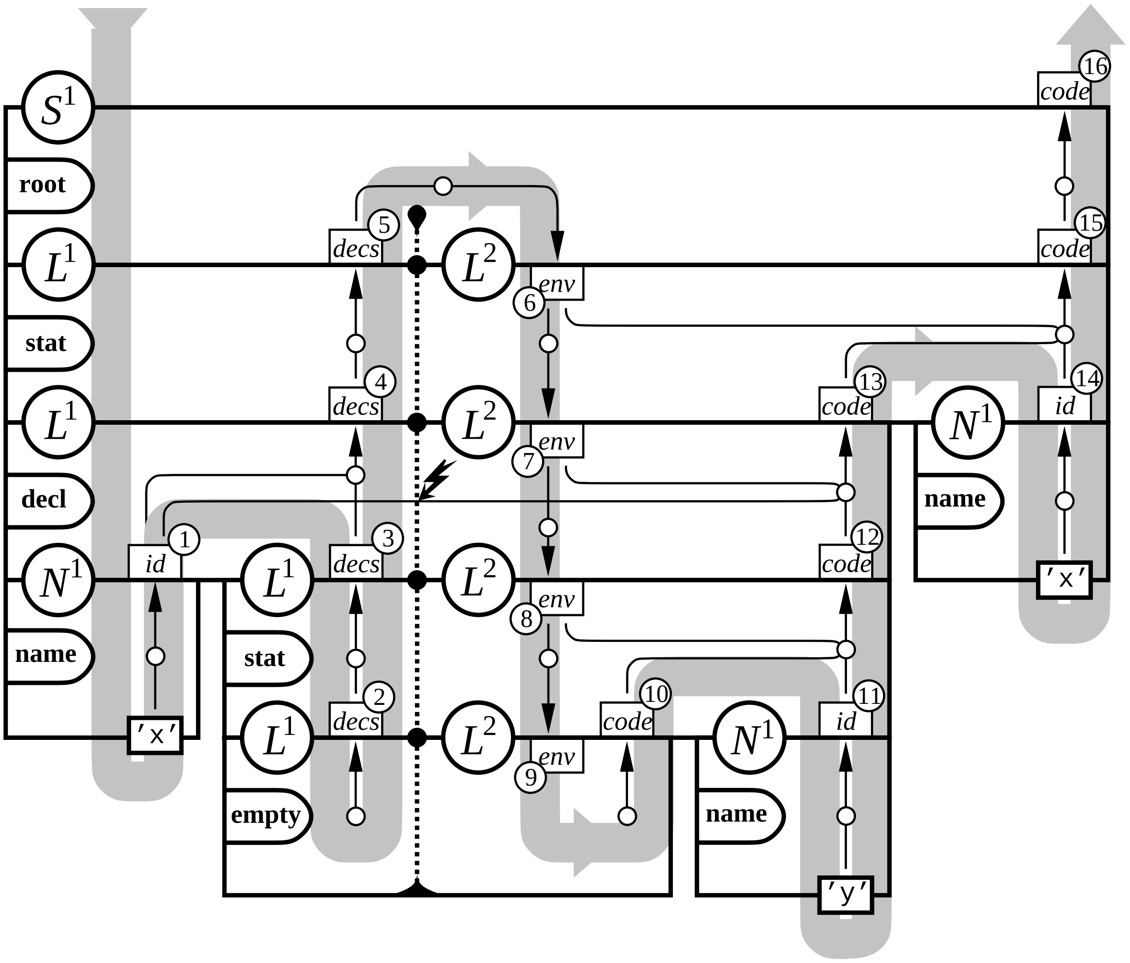
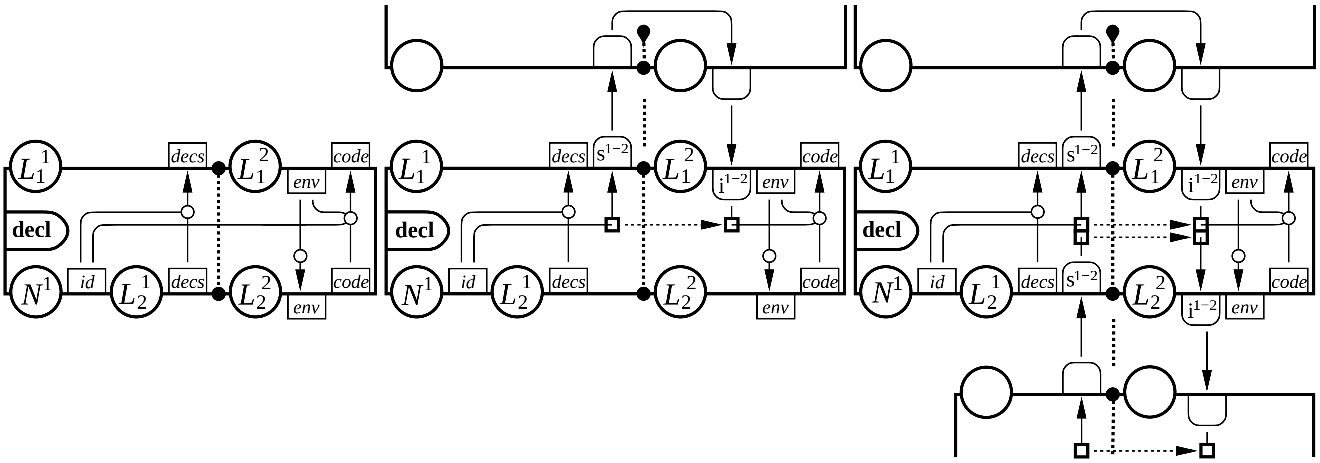


Figure 3.21. Plan tree for the VARUSE application composed from plan icons



a. Intra-visit-dependency **b.** Bypass visit-border **c.** Handle bypass for child

Border-crossers are bundled (stacks of small squares) and passed to the parent via the *binding attributes* denoted by the boxes with the round corners. The bypassed border-crossings are shown with dashed arrows.

Figure 3.22. Diverting intra-visit-dependencies

compartment number can thus be read from the superscript in the label of that disc.

Intra-visit-dependencies are easily recognized in plan icons: an arrow crosses a visit-border. Intra-visit-dependencies are therefore also referred to as *border-crossings*, their sources as *border-crossers*. The border-crossing in the VARUSE example is easily spotted. In Figures 3.20$_{\triangleleft 65}$ and 3.21$_{\triangleleft}$ border-crossings are marked with a lightning symbol (⚡) next to the border. It is immediately clear that the equation for $L_1.code$ in the second compartment of $decl$ uses border-crosser $N.id$ which is defined in the first compartment.

3.4.3 Bindings

When an abstract syntax tree is decorated, each node is visited several times. In a visit some attribute instances are computed, some children are visited and finally the visit exits. During a subsequent visit to the same node, attribute instances computed in earlier visits may be needed (border-crossers).

In the functional approach by Vogt [VSK91, SV91, Vog93] this is still feasible. There are two reasons for *not* following that direction. First, it is not purely functional: the tree is a parameter of the visit-functions and functions should not change arguments. Secondly, if visit-functions are purely functional (without side-effects), they can be memoized. Memoized visit-functions implement an elegant incremental evaluator.

In a purely functional setting, a visit not only computes the required synthesized attributes, but also the border-crossers that need to be transported to a later visit. A bundle is passed to the parent and we rely on him to pass it back again. Figure 3.22b illustrates how a bundle with only attribute instance $N.id$ is transported to the parent via the fresh synthesized attribute $L_1.\mathrm{s}^{1\to2}$ and passed back via the fresh inherited attribute $L_1.\mathrm{i}^{1\to2}$. This mechanism solves the border-crossing for the $decl$ node shown in Figure 3.22a. A bundle is known as *binding* [VSK91, Pen93]; in this example the binding is denoted by $L^{1\to2}$. The newly introduced attributes $L_1.\mathrm{s}^{1\to2}$ and $L_1.\mathrm{i}^{1\to2}$ are known as *binding attributes*.

If a $decl$ node may request its parent to pass a binding back on the second visit, then a child of $decl$ may ask the same of $decl$. So, by induction, child L_2 of a $decl$ node also introduces two new attributes: a synthesized attribute $L_2.\mathrm{s}^{1\to2}$ and an inherited attribute $L_2.\mathrm{i}^{1\to2}$ (see Figure 3.22c). The first visit to L_2 (which synthesizes $L_2.\mathrm{s}^{1\to2}$) takes place in the first compartment of $decl$ whereas the second visit to L_2 (which uses $L_2.\mathrm{i}^{1\to2}$) takes place in the second compartment of $decl$. As a consequence, the dependency $L_2.\mathrm{s}^{1\to2} \to L_2.\mathrm{i}^{1\to2}$ is an intra-visit-dependency for $decl$. Consequently, border-crossers $L_2.\mathrm{s}^{1\to2}$ and $N.id$ are bundled together, creating a binding that is passed to the parent and received back in the second compartment. There the binding is unpacked, making $N.id$ available for the $lookup$ and $L_2.\mathrm{s}^{1\to2}$ available for copying into $L_2.\mathrm{i}^{1\to2}$.

$L_2.\mathrm{s}^{1\to2}$ is a binding attribute occurrence for child L_2 from its first visit to its second. It should be denoted with $p\mathbf{o}2.\mathrm{s}^{1\to2}$ since $p\mathbf{o}2$ is the correct denotation for L_2.

Factorized versus accumulated bindings

The VARUSE example is relatively simple in that the maximum number of visits of any non-terminal is two. A non-terminal with only one visit needs no bindings, and a non-terminal with two visits needs at most a binding from visit one to two. If a grammar has a non-terminal with more than two visits, we are confronted with a choice. Suppose X has three visits. The first visit might compute values needed in the second and third visit, and the second visit might compute values that are needed in the third. So, we could either define bindings $X^{1\to2}$ and $X^{2\to3}$, accumulating (threading) the border-crossings, or we could define bindings $X^{1\to2}$, $X^{1\to3}$ and $X^{2\to3}$, factorizing the border-crossings. The advantage of the former is that fewer bindings are required. The number of bindings is linear in the number of visits, namely $\mathbf{v}X-1$. In the latter approach the number of bindings is quadratic in the number of visits, namely $\frac{1}{2}\cdot\mathbf{v}X\cdot(\mathbf{v}X-1)$.

$\mathbf{v}X$	acc	fac
1	0	0
2	1	1
3	2	3
4	3	6
5	4	10
6	5	15

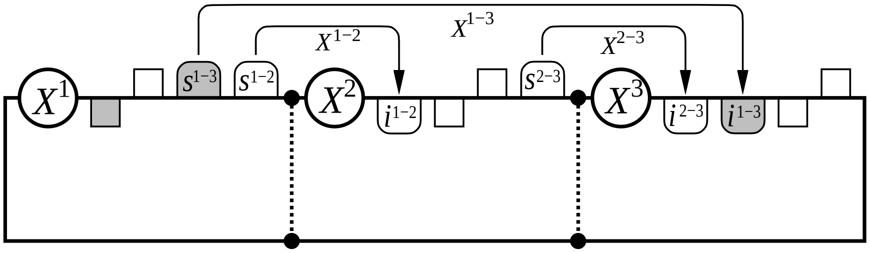


Figure 3.23. Factorization gives better incremental behavior than accumulation

The factorized solution gives a better incremental behavior though. For example, take the three-visit node in Figure 3.23$_{\triangleleft}$, which uses factorized bindings, and assume that the grey attributes change due to an edit action. Since the arguments for the second visit do not change it can be skipped. If we had used accumulated bindings instead, the border-crossers that were formerly contained in $X^{1\to 2}$ and $X^{1\to 3}$ would be merged into one binding from 1 to 2. As a consequence, this binding would change since it incorporates the changing border-crosser formerly contained in $X^{1\to 3}$. This leads to a redundant second visit.

Because we aim at time-efficiency and not so much at space-efficiency, factorized bindings have our preference. Although the static analysis for bindings is cumbersome, (factorized) bindings have a considerable advantage: they contain precisely the information needed for a particular visit and nothing more. Factorized bindings accurately indicate whether a visit may be skipped.

As a result of introducing bindings—factorized or accumulated—the free variables of the visit-functions are bound. Visit-functions then correspond to supercombinators [Hug82].

The bind grammar

A binding from visit v to visit w $(v < w)$ contains the border-crossers computed in visit v and used in visit w in an organized manner. More precisely, a binding is a term whose structure will be defined using a context-free grammar, known as the *bind grammar*. As a preliminary example, consider the binding computed for the VARUSE application given in Figure 3.31$_{\triangleright 83}$.

The visit-sequences will be augmented with *binding attributes* that pass around bindings in the abstract syntax tree. The bind grammar describes the types of and operators on the binding attributes. In order to distinguish between trees, productions and non-terminals of the underlying context-free grammar on one hand and of the bind grammar on the other hand, we will introduce different names for the respective elements of the bind grammar.

Bindings are constructed using *wrappers*, which are the equivalent of productions. A production constructs a tree for the left-hand side non-terminal out of the trees for the right-hand side non-terminals and zero or more terminals. Likewise, a wrapper constructs a binding for the left-hand side *parcel* out of the bindings for the right-hand side parcels and zero or more attribute instances (coming from the production instance the wrapper is associated with). The following table summarizes the terminology and illustrates the analogy between algebras, context-free-grammars and bindings.

algebra	**context-free grammar**	**bind grammar**
term	tree	binding
constructor	production	wrapper
sort	non-terminal	parcel

A bind grammar is induced by an attribute grammar and its visit-sub-sequences. Non-terminal X induces the parcels $X^{v\to w}$, where $1 \leq v < w \leq \mathbf{v}X$. Note that the number of parcels induced by X is $\frac{1}{2}\cdot\mathbf{v}X\cdot(\mathbf{v}X-1)$. Parcels denote the elementary *types* of a binding and thus of the binding attributes in the augmented visit-sequences.

Production p induces the wrappers $p^{v\to w}$, where $1 \leq v < w \leq \mathbf{v}p$. The left-hand side parcel of $p^{v\to w}$ is $p{\cdot}0^{v\to w}$. The right-hand side of $p^{v\to w}$ consists of parcels and attributes. Informally, $p^{v\to w}$ bundles the occurrences of binding attributes and plain attributes that are defined in compartment v of p and used in compartment w of p. For a formal definition, we must first augment the visit-sequences with binding attributes and then perform a life-time analysis on plain and binding attributes.

The binding attributes

In order to pass around bindings in an abstract syntax tree, we associate fresh attributes to non-terminals, the so-called *binding attributes*. The visit-sequences will be augmented with equations to reflect the passing of bindings.

The set $BA_{inh}(X)$ of inherited binding attributes associated with non-terminal X is defined as $BA_{inh}(X) = \{X.\mathrm{i}^{v\to w} \mid 1 \leq v < w \leq \mathbf{v}X\}$. Likewise, $BA_{syn}(X) = \{X.\mathrm{s}^{v\to w} \mid 1 \leq v < w \leq \mathbf{v}X\}$ is the set of synthesized binding attributes. $BA(X)$ denotes the set of all binding attributes associated with X. It is defined as $BA(X) = BA_{inh}(X) \cup BA_{syn}(X)$. Elements of $BA(X)$ will usually be denoted with $X.\mathrm{a}^{v\to w}$ unless we want to stress that the binding attribute is inherited or synthesized. The type of binding attribute $X.\mathrm{a}^{v\to w}$ is parcel $X^{v\to w}$.

We assume that attribute names like $X.\mathrm{i}^{v\to w}$ and $X.\mathrm{s}^{v\to w}$ do not occur in $A(X)$. Note that i and s stand for inherited respectively synthesized.

For each occurrence $p\mathbf{o}i$ of non-terminal X, there is an occurrence denoted by $p\mathbf{o}i.\mathrm{a}^{v\to w}$ for every binding attribute $X.\mathrm{a}^{v\to w} \in BA(X)$. Occurrences of binding attributes will be referred to as *binding occurrences*. $BO(p\mathbf{o}i)$ is the set of all binding occurrences of $p\mathbf{o}i$ defined as $BO(p\mathbf{o}i) = \{p\mathbf{o}i.\mathrm{a}^{v\to w} \mid p{\cdot}i.\mathrm{a}^{v\to w} \in BA(p{\cdot}i)\}$. $BO(p)$ is the set of all binding occurrences associated with production p. It consists of the binding occurrences associated with every non-terminal occurrence of p: $BO(p) = \bigcup_{0\leq i\leq \mathbf{s}p} BO(p\mathbf{o}i)$.

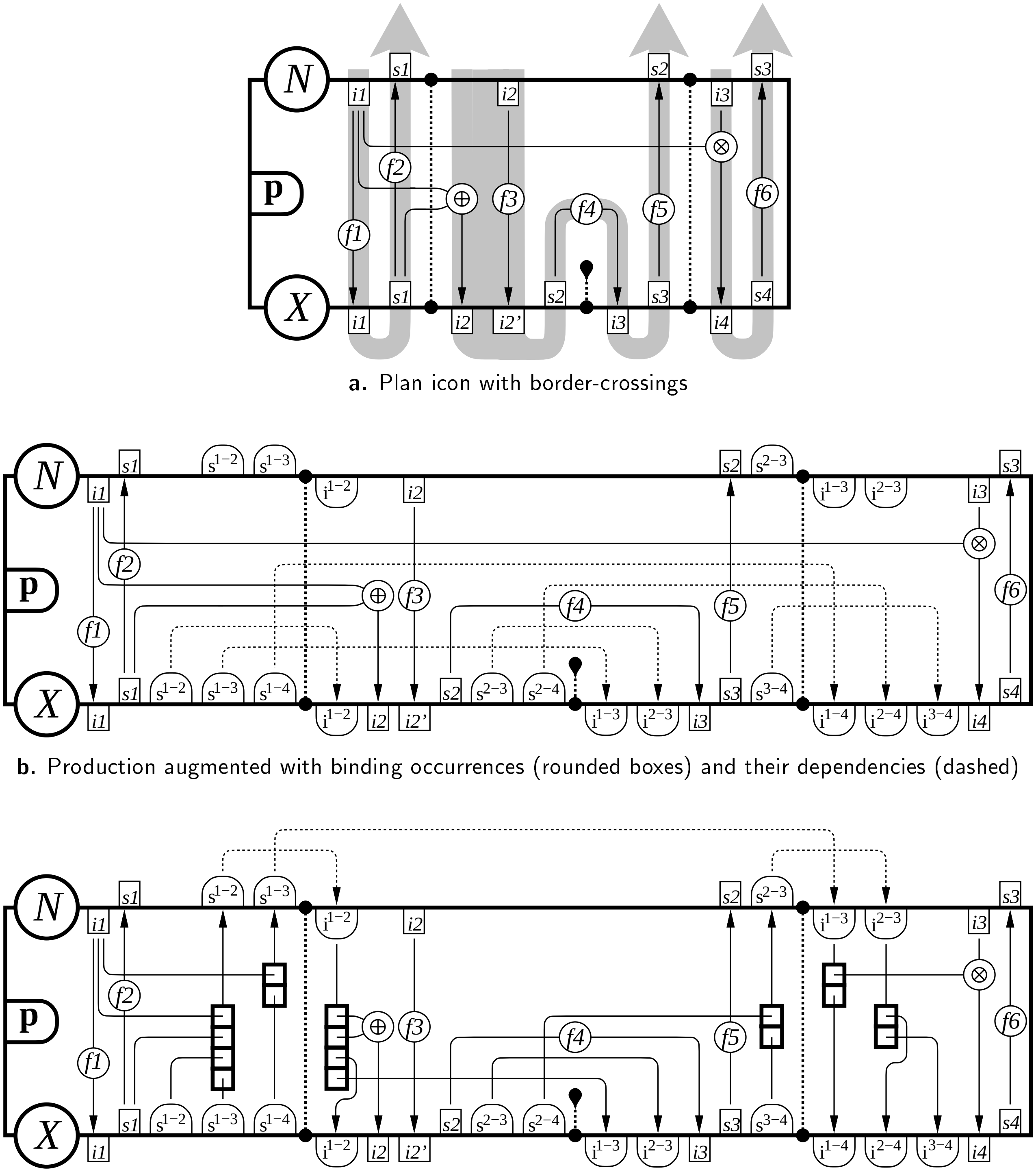
$$\begin{tikzcd}
X \arrow[r, leftrightarrow, "P"] \arrow[d, "BA(\cdot)"'] & p\mathbf{o}i \arrow[d, "BO(\cdot)"] \\
X.\mathrm{a}^{v\to w} \arrow[r, leftrightarrow, "P"'] & p\mathbf{o}i.\mathrm{a}^{v\to w}
\end{tikzcd}$$

Let f be an algorithm that computes visit-sequences. The set of synthesized binding attributes of visit v to X $(1 \leq v \leq \mathbf{v}_f X)$ is defined as $X\mathbf{BS}_f\ v = \{X.\mathrm{s}^{v\to w} \in BA_{syn}(X) \mid v < w \leq \mathbf{v}_f X\}$. The set of inherited binding attributes for visit v to X is defined by $X\mathbf{BI}_f\ v = \{X.\mathrm{i}^{u\to v} \in BA_{inh}(X) \mid 1 \leq u < v\}$. The selectors $\mathbf{BS}_f$ and $\mathbf{BI}_f$ are overloaded: $p\mathbf{o}i\mathbf{BS}_f\ v = \{p\mathbf{o}i.\mathrm{s}^{v\to w} \mid p{\cdot}i.\mathrm{s}^{v\to w} \in p{\cdot}i\mathbf{BS}_f\ v\}$ respectively $p\mathbf{o}i\mathbf{BI}_f\ v = \{p\mathbf{o}i.\mathrm{i}^{v\to w} \mid p{\cdot}i.\mathrm{i}^{v\to w} \in p{\cdot}i\mathbf{BI}_f\ v\}$. From now on, we assume the existence of visit-sequences so we drop the subscript f. Note the (notational) analogy with the operators for attributes.

attr	bind
A	BA
O	BO
$\mathbf{S}$	$\mathbf{BS}$
$\mathbf{I}$	$\mathbf{BI}$

Recall that in the VARUSE grammar $decl\mathbf{o}2$ is usually written as L_2 and $root\mathbf{o}1$ even as L. Similarly, binding occurrence $decl\mathbf{o}2.\mathrm{a}^{1\to 2}$ is usually rendered as $L_2.\mathrm{a}^{1\to 2}$ and $root\mathbf{o}1.\mathrm{a}^{1\to 2}$ as $L.\mathrm{a}^{1\to 2}$. Both have the parcel $L^{1\to 2}$ as type.

We shall now define the dependencies between the binding attributes. For each



a. Plan icon with border-crossings

b. Production augmented with binding occurrences (rounded boxes) and their dependencies (dashed)

c. Border-crossings diverted

Figure 3.24. Diversion of border-crossings with bindings in production p of the BINDING grammar

child poi of production p, the binding attribute occurrence $poi.\mathrm{i}^{v\rightarrow w}$ is a copy of $poi.\mathrm{s}^{v\rightarrow w}$. The set $BE(poi)$ consists of these (binding) equations:

$$BE(poi) = \{(poi.\mathrm{i}^{v\rightarrow w} := id\ poi.\mathrm{s}^{v\rightarrow w}) \mid poi.\mathrm{i}^{v\rightarrow w} \in BO(poi)\} \quad (i \neq 0).$$

The equations for the synthesized binding occurrences of the parent $po0$ are explicitly excluded from the above definition. Binding analysis, which will be described next, determines which attribute or binding occurrences are border-crossers. Let $bind(p, v, w)$ be the set of all border-crossers defined in compartment v and used in compartment w of p. The equation for $po0.\mathrm{s}^{v\rightarrow w}$ creates a bundle of all border-crossers in $bind(p, v, w)$. It uses wrapper $p^{v\rightarrow w}$ for that.

$bind(p, v, w)$ is determined by *binding analysis*, which will be discussed next.

$$BE(po0) = \{(po0.\mathrm{s}^{v\rightarrow w} := \mathbf{p}^{v\rightarrow w}(\text{“}bind(p, v, w)\text{”})) \mid po0.\mathrm{s}^{v\rightarrow w} \in BO(po0)\}$$

Figure 3.24a shows the plan icon of an artificial production $N = \mathbf{p}(X)$ that we will use in this section to illustrate bindings. This production will be referred to as production p of the BINDING grammar. For future reference, the semantic functions in Figure 3.24a have been given a name. Note also that the visit discs are replaced by a single non-terminal disc to “abbreviate” the plan icon. In the BINDING grammar, non-terminal N has three visits and X has four visits. Thus, three parcels (six binding attributes) are associated with N and six parcels (twelve binding attributes) with X. Figure 3.24b shows the plan icon augmented with the inherited and synthesized binding occurrences. Furthermore, it shows the dependencies for the binding occurrences of child X with dashed arrows.

3.4.4 Binding analysis

Binding analysis determines border-crossers and thereby the right-hand side of the wrappers. Border-crossers in the augmented visit-sequences are either occurrences of plain attributes or occurrences of binding attributes of right-hand side non-terminals. For each occurrence of either kind the defining compartment and all using compartments will be determined. This is known as *lifetime analysis*. For example, attribute occurrence $X.s1$ in Figure 3.24b is *defined* in the first compartment (or subsequence) and *used* in the first and second compartment. Binding occurrence $X.\mathrm{s}^{1\rightarrow 3}$ (more correctly denoted with $po1.\mathrm{s}^{1\rightarrow 3}$) is also defined in the first compartment; it is only used in the second.

First, the visit-sequences will be augmented with instructions associated with the computation of binding occurrences. Secondly, the augmented visit-sequences will be annotated with define and usage directives to facilitate lifetime analysis, which is the third step of binding analysis. Observe that binding analysis is completely static.

Roughly sketched, binding analysis determines which arrows must be “cut and diverted” in Figure 3.24b to obtain Figure 3.24c. Wrappers construct the bundles denoted by the stacks of small squares in Figure 3.24c.

Augmented visit-sub-sequences

Lifetime analysis is based upon visit-sub-sequences. However, the visit-sequences of the attribute grammar should first be augmented with the binding attributes and their equations described above.

Each $visit(i,v)$ instruction in the plan of p is prefixed with the instructions $eval(\beta)$ for each $\beta \in p\mathbf{o}i\mathbf{BI}\,v$. The associated equation is the copy equation (because $visit$ instructions apply to children, i is guaranteed to be non-zero).

The equations for the synthesized binding attributes of the left-hand side non-terminal pose a small problem. They should be inserted at the end of each sub-sequence. To be more precise an instruction $eval(p\mathbf{o}0.\mathrm{s}^{v \to w})$ should be appended for each binding occurrence $p\mathbf{o}0.\mathrm{s}^{v \to w} \in p\mathbf{o}0\mathbf{BS}\,v$. The associated equation is $\mathbf{p}^{v \to w}(\text{“}bind(p,v,w)\text{”})$, where $bind(p,v,w)$ is defined by binding analysis. However, binding analysis would be frustrated by such instructions because it uses $eval$ instructions to *determine* $bind(p,v,w)$. Therefore, we propose the alternative name $pass$ for $eval$ in these cases.

Annotated visit-sub-sequences

Lifetime analysis uses augmented visit-sub-sequences as input. However, the sub-sequences will first be annotated with define and usage directives to make the analysis easier. We distinguish between the three kinds of instructions that occur in the plan for production p.

The $sref$ directives become important in the next chapter.

- An $eval(\alpha)$ instructions shows that α is defined. We add the directive $uses(\beta)$ for every attribute occurrence β that α depends on. We add a directive $sref(p\mathbf{o}i)$ for every non-terminal that is syntactically referenced, thus $(\alpha := f \ldots p\mathbf{o}i \ldots) \in E(p)$.
- A $visit(i,v)$ causes non-terminal occurrence $p\mathbf{o}i$ to be visited for the vth time. Hence the attribute occurrences of $p\mathbf{o}i\mathbf{I}\,v$ and the binding occurrences of $p\mathbf{o}i\mathbf{BI}\,v$ are used as arguments and the attribute occurrences of $p\mathbf{o}i\mathbf{S}\,v$ and binding occurrences of $p\mathbf{o}i\mathbf{BS}\,v$ are returned. Thus we add $inp(\alpha)$ for every $\alpha \in p\mathbf{o}i\mathbf{I}\,v \cup p\mathbf{o}i\mathbf{BI}\,v$ and $out(\alpha)$ for every $\alpha \in p\mathbf{o}i\mathbf{S}\,v \cup \beta \in p\mathbf{o}i\mathbf{BS}\,v$.

The lifetime of the binding occurrences associated to the left-hand side non-terminal need not be analyzed.

- Analogous annotation is derived for a $suspend(v)$ instruction. Subsequence v computes the attribute occurrences of $p\mathbf{o}0\mathbf{S}\,v$ using the attribute occurrences of $p\mathbf{o}0\mathbf{I}\,v$. Therefore we add $inh(\alpha)$ for every $\alpha \in p\mathbf{o}0\mathbf{I}\,v$ and $syn(\alpha)$ for every $\alpha \in p\mathbf{o}0\mathbf{S}\,v$.
- Each $pass(p\mathbf{o}0.\mathrm{s}^{v \to w})$ instruction is left untouched.

The interfaces for the three visits to non-terminal N of the BINDING grammar are $(\{N.i1\}, \{N.s1\})$ respectively $(\{N.i2\}, \{N.s2\})$ and $(\{N.i3\}, \{N.s3\})$. Non-terminal X has four visits with interfaces $(\{X.i1\}, \{X.s1\})$ respectively $(\{X.i2, X.i2'\}, \{X.s2\})$, $(\{X.i3\}, \{X.s3\})$ and $(\{X.i4\}, \{X.s4\})$. Note that visit 2 and 3 to X both fall in the second compartment of p. The annotated visit-sequences for production p are given in Figure 3.25$_{\triangleright}$. Observe that they are much better understandable for the human reader than plain visit-sequences.

Lifetime analysis

Annotated visit-sub-sequences specify *when* an attribute or binding occurrence is *defined* or *used*. Let us denote the annotated visit-sub-sequence number w of production p with $vss(p, w)$. The set $def(p, w)$ denotes the attribute and binding occurrences that are defined in $vss(p, w)$. Likewise, $use(p, w)$ denotes the set of occurrences that are used in $vss(p, w)$. They are defined as follows.

$def(p, w)$ contains all occurrences that become available in compartment w of p. Inherited attributes of the parent "inh" and synthesized attributes of a child "out" belong to this set.

$$\begin{aligned} use(p, w) &= \{\alpha \mid syn(\alpha),\ uses(\alpha) \text{ or } inp(\alpha) \text{ occurs in } vss(p, w)\} \\ def(p, w) &= \{\alpha \mid inh(\alpha),\ eval(\alpha) \text{ or } out(\alpha) \text{ occurs in } vss(p, w)\} \end{aligned}$$

Occurrences that are used but not defined in a compartment are called free.

$$free(p, w) = use(p, w) \setminus def(p, w)$$

A free occurrence α in compartment w corresponds to a border-crossing. If α is defined in compartment v, then α should be transported from v to w. The set $bind(p, v, w)$ consists of all attribute and binding occurrences that should be transported from v to w. It is defined as

$$bind(p, v, w) = free(p, w) \cap def(p, v) \quad .$$

Let us go back to the BINDING grammar. The following occurrences are used in the second compartment of p.

$$\begin{aligned} use(p, 2) = \{\ & N.s2 && \text{(syn)} \\ ,\ & N.i1,\ X.s1,\ N.i2,\ X.\mathrm{s}^{1\to 2},\ X.s2,\ X.\mathrm{s}^{1\to 3},\ X.\mathrm{s}^{2\to 3},\ X.s3 && \text{(uses)} \\ ,\ & X.\mathrm{i}^{1\to 2},\ X.i2,\ X.i2',\ X.\mathrm{i}^{1\to 3},\ X.\mathrm{i}^{2\to 3},\ X.i3 && \text{(inp)} \\ \}\ & \end{aligned}$$

Many of these are also defined in the second compartment, such as attribute occurrence $X.i2$ ($eval$) and binding occurrence $X.\mathrm{s}^{2\to 3}$ (out).

$$\begin{aligned} def(p, 2) = \{\ & N.i2 && \text{(inh)} \\ ,\ & X.i2,\ X.i2',\ X.\mathrm{i}^{1\to 2},\ X.i3,\ X.\mathrm{i}^{1\to 3},\ X.\mathrm{i}^{2\to 3},\ N.s2 && \text{(eval)} \\ ,\ & X.s2,\ X.\mathrm{s}^{2\to 3},\ X.\mathrm{s}^{2\to 4},\ X.s3,\ X.\mathrm{s}^{3\to 4} && \text{(out)} \\ \}\ & \end{aligned}$$

Consequently, the following occurrences should be bound for the second compartment.

$$free(p, 2) = \{N.i1, X.s1, X.\mathrm{s}^{1\to 2}, X.\mathrm{s}^{1\to 3}\}$$

```
begin 1
  inh(N.i1)
  syn(N.s1)
  eval(X.i1)
    uses(N.i1);
  visit(X, 1)
    inp(X.i1)
    out(X.s1)
    out($X.\mathrm{s}^{1\to 2}$)
    out($X.\mathrm{s}^{1\to 3}$)
    out($X.\mathrm{s}^{1\to 4}$);
  eval(N.s1)
    uses(X.s1);
  pass($N.\mathrm{s}^{1\to 2}$);
  pass($N.\mathrm{s}^{1\to 3}$)
end 1
```

```
begin 2
  inh(N.i2)
  syn(N.s2)
  eval(X.i2)
    uses(N.i1)
    uses(X.s1);
  eval($X.i2'$)
    uses(N.i2);
  eval($X.\mathrm{i}^{1\to 2}$)
    uses($X.\mathrm{s}^{1\to 2}$);
  visit(X, 2)
    inp($X.\mathrm{i}^{1\to 2}$)
    inp(X.i2)
    inp($X.i2'$)
    out(X.s2)
    out($X.\mathrm{s}^{2\to 3}$)
    out($X.\mathrm{s}^{2\to 4}$);
  eval(X.i3)
    uses(X.s2);
  eval($X.\mathrm{i}^{1\to 3}$)
    uses($X.\mathrm{s}^{1\to 3}$);
  eval($X.\mathrm{i}^{2\to 3}$)
    uses($X.\mathrm{s}^{2\to 3}$);
  visit(X, 3)
    inp($X.\mathrm{i}^{1\to 3}$)
    inp($X.\mathrm{i}^{2\to 3}$)
    inp(X.i3)
    out(X.s3)
    out($X.\mathrm{s}^{3\to 4}$);
  eval(N.s2)
    uses(X.s3);
  pass($N.\mathrm{s}^{2\to 3}$)
end 2
```

```
begin 3
  inh(N.i3)
  syn(N.s3)
  eval(X.i4)
    uses(N.i1)
    uses(N.i3);
  eval($X.\mathrm{i}^{1\to 4}$)
    uses($X.\mathrm{s}^{1\to 4}$);
  eval($X.\mathrm{i}^{2\to 4}$)
    uses($X.\mathrm{s}^{2\to 4}$);
  eval($X.\mathrm{i}^{3\to 4}$)
    uses($X.\mathrm{s}^{3\to 4}$);
  visit(X, 4)
    inp($X.\mathrm{i}^{1\to 4}$)
    inp($X.\mathrm{i}^{2\to 4}$)
    inp($X.\mathrm{i}^{3\to 4}$)
    inp(X.i4)
    out(X.s4);
  eval(N.s3)
    uses(X.s4)
end 3
```

Figure 3.25. Annotated visit-sub-sequences for production p of the BINDING grammar

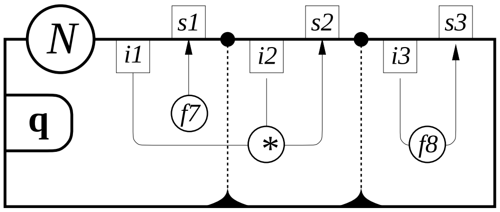


Figure 3.26. The (only) other production on N in the BINDING grammar

By the nature of visit-sequences, these occurrences are all defined in the first compartment. Therefore we have

$$bind(p,1,2) = \{N.i1, X.s1, X.\mathrm{s}^{1\to 2}, X.\mathrm{s}^{1\to 3}\} \quad .$$

In the first compartment of p in Figure 3.24c$_{\triangleleft 72}$, we see a stack of four small squares. They correspond with the four elements in $bind(p,1,2)$.

The attribute occurrences $N.i1$ and $X.s1$ are easily spotted as border-crossers in Figure 3.24b$_{\triangleleft 72}$. Whether $X.\mathrm{s}^{1\to 2}$ and $X.\mathrm{s}^{1\to 3}$ are really needed depends on the (visit-sequences associated to the) productions on X. If none of them have border-crossings, the bindings may be omitted. This will be discussed later.

Similar analysis yields

$$\begin{aligned} bind(p,1,3) &= \{N.i1, X.\mathrm{s}^{1\to 4}\} \quad , \\ bind(p,2,3) &= \{X.\mathrm{s}^{2\to 4}, X.\mathrm{s}^{3\to 4}\} \quad . \end{aligned}$$

Wrappers

Wrapper $p^{v\to w}$ creates a bundle containing the border-crossers in $bind(p,v,w)$. For a formal definition of wrappers, we use a similar denotation as for productions, namely a list consisting of the left-hand side and right-hand side parcels. Formally, a wrapper is defined as follows

Set to list conversion: just pick an order.

$$p^{v\to w} = p{\cdot}0^{v\to w} : [\hat{\alpha} \mid \alpha \in bind(p,v,w)] \quad .$$

This definition consists of two parts. The first part, $p{\cdot}0^{v\to w}$, is the left-hand side parcel. The second part is a list comprehension that defines the right-hand side. Every border-crosser α is mapped onto its type $\hat{\alpha}$. For an attribute occurrence $\mathbf{po}i.a$, $\widehat{\mathbf{po}i.a}$ maps to the type of attribute $p{\cdot}i.a$. For a binding occurrence, $\widehat{\mathbf{po}i.\mathrm{s}}^{t\to u}$ maps to the parcel $p{\cdot}i^{t\to u}$.

Let us illustrate wrappers with the BINDING grammar. Suppose that in addition to p there is one other production on N, namely $N = \mathbf{q}()$. The plan icon of q is given in Figure 3.26. Lifetime analysis of production q yields $bind(q,1,2) = \{N.i1\}$ as opposed to $bind(p,1,2) = \{N.i1, X.s1, X.\mathrm{s}^{1\to 2}, X.\mathrm{s}^{1\to 3}\}$. Thus we obtain

$$\begin{aligned} p^{1\to2} &= N^{1\to2} : [\widehat{N.i1}, \widehat{X.s1}, \widehat{X.\mathrm{s}^{1\to2}}, \widehat{X.\mathrm{s}^{1\to3}}] \\ q^{1\to2} &= N^{1\to2} : [\widehat{N.i1}] \quad . \end{aligned}$$

$\widehat{X.\mathrm{s}^{1\to2}} = X^{1\to2}$

$\widehat{X.\mathrm{s}^{1\to3}} = X^{1\to3}$

The following fragment defines the type $N^{1\to2}$ using the inferred wrapper definitions. Since p and q are the only productions on N, $p^{1\to2}$ and $q^{1\to2}$ are the only wrappers on $N^{1\to2}$.

$$\begin{aligned} N^{1\to2} &= \mathbf{p}^{1\to2}(\widehat{N.i1}, \widehat{X.s1}, X^{1\to2}, X^{1\to3}) \\ &\mid \ \mathbf{q}^{1\to2}(\widehat{N.i1}) \end{aligned}$$

A wrapper $p^{v\to w}$ is said to be *terminal* if $bind(p,v,w) \cap BO(p) = \{\}$. In that case, the wrapper has no children. A wrapper is *empty* if $bind(p,v,w) = \{\}$. For example, all wrappers induced by production q in of the BINDING grammar are terminal since q itself is terminal. Wrappers $\mathbf{q}^{1\to3}$ and $\mathbf{q}^{2\to3}$ are even empty.

The FUN mapping

The VSS mapping discussed earlier has one serious flaw. Intra-visit-dependencies in an attribute grammar map to side-effects of the visit-functions. However, by augmenting the visit-sequences with bindings, intra-visit-dependencies are diverted. As a result, the augmented visit-sequences are *free* of intra-visit-dependencies so that a VSS-like mapping will map them to pure visit-functions.

The FUN mapping maps visit-sequences to pure visit-functions. It is defined as the VSS mapping on the augmented visit-sequences, where $pass$ instructions are now interpreted as $eval$ instructions.

The order of the arguments—the inherited (binding) attributes—can be freely chosen (see also page 116).

As a result of the FUN mapping, a visit-function $visit_X^v$ is constructed for every non-terminal X and every visit v such that $1 \le v \le \mathbf{v}X$. It has the following signature

$$visit_X^v :: X \times \text{“}X\mathbf{BI}\,v\text{”} \times \text{“}X\mathbf{I}\,v\text{”} \longrightarrow \text{“}X\mathbf{S}\,v\text{”} \times \text{“}X\mathbf{BS}\,v\text{”} \quad .$$

The quotes around a set belong to the meta-language. “V” should be interpreted as (the Cartesian product of) *the types of the elements of* V.

$\mathbf{p}^{v\to w}(\cdot)$ abbreviates $\mathbf{p}^{v\to w}(\text{‘}bind(p,v,w)\text{’})$

For each production p on X an alternative body for $visit_X^v$ is generated. It is selected by using pattern matching on p in the following manner

$$\begin{aligned} visit_X^v \quad & \mathbf{p}(\cdot) \quad \mathbf{p}^{1\to v}(\cdot) \quad \ldots \quad \mathbf{p}^{(v-1)\to v}(\cdot) \quad \text{‘}X\mathbf{I}\,v\text{’} \\ &= (\ \text{‘}X\mathbf{S}\,v\text{’}\ ,\ \mathbf{p}^{v\to(v+1)}(\cdot)\ ,\ \ldots\ ,\ \mathbf{p}^{v\to\mathbf{v}X}(\cdot)\ ) \\ & \quad \textbf{where } body \quad . \end{aligned}$$

The single quotes around a set also belong to the meta-language. ‘V’ should be interpreted as *the elements of* V. Since the bindings follow the structure of the abstract syntax tree, the multiple pattern matching on the binding constructors (wrappers) will never fail. The $body$ is obtained from visit-sub-sequence v of

$$
\begin{array}{lll}
\multicolumn{3}{l}{visit_N^2 :: N \times N^{1\to 2} \times \widehat{N.i2} \longrightarrow \widehat{N.s2} \times N^{2\to 3}} \\
\multicolumn{3}{l}{visit_N^2\ \mathbf{p}(X)\ \mathbf{p}^{1\to 2}(N.i1, X.s1, X.\mathrm{s}^{1\to 2}, X.\mathrm{s}^{1\to 3})\ N.i2 = (N.s2, N.\mathrm{s}^{2\to 3})} \\
\quad \mathbf{where} & X.i2 & = N.i1 \oplus X.s1 \\
 & ;\ X.i2' & = f3\ N.i2 \\
 & ;\ X.\mathrm{i}^{1\to 2} & = id\ X.\mathrm{s}^{1\to 2} \\
 & ;\ (X.s2, X.\mathrm{s}^{2\to 3}, X.\mathrm{s}^{2\to 4}) & = visit_X^2\ X\ X.\mathrm{i}^{1\to 2}\ X.i2\ X.i2' \\
 & ;\ X.i3 & = f4\ X.s2 \\
 & ;\ X.\mathrm{i}^{1\to 3} & = id\ X.\mathrm{s}^{1\to 3} \\
 & ;\ X.\mathrm{i}^{2\to 3} & = id\ X.\mathrm{s}^{2\to 3} \\
 & ;\ (X.s3, X.\mathrm{s}^{3\to 4}) & = visit_X^3\ X\ X.\mathrm{i}^{1\to 3}\ X.\mathrm{i}^{2\to 3}\ X.i3 \\
 & ;\ N.s2 & = f5\ X.s3 \\
 & ;\ N.\mathrm{s}^{2\to 3} & = \mathbf{p}^{2\to 3}(X.\mathrm{s}^{2\to 4}, X.\mathrm{s}^{3\to 4}) \\
\multicolumn{3}{l}{visit_N^2\ \mathbf{q}()\ \mathbf{q}^{1\to 2}(N.i1)\ N.i2 = (N.s2, N.\mathrm{s}^{2\to 3})} \\
\quad \mathbf{where} & N.s2 & = N.i1 * N.i2 \\
 & ;\ N.\mathrm{s}^{2\to 3} & = \mathbf{q}^{2\to 3}()
\end{array}
$$

Figure 3.27. Visit-functions using bindings

production p. Each $eval$ instruction maps to its associated equation. A visit instruction of the form $visit(p\mathbf{o}i, w)$ maps to a recursive call of the form

$$(\text{`}p\mathbf{o}i\mathbf{S}\,w\text{'}\ ,\ \text{`}p\mathbf{o}i\mathbf{BS}\,w\text{'}\ ) = visit^{w}_{p\cdot i}\quad p\mathbf{o}i\quad \text{`}p\mathbf{o}i\mathbf{BI}\,w\text{'}\quad \text{`}p\mathbf{o}i\mathbf{I}\,w\text{'}\quad .$$

As an example, consider the function $visit_N^2$ given in Figure 3.27.

Let S be the start symbol of an attribute grammar. An abstract syntax tree T rooted S is evaluated by successively calling the visit-functions $visit_S^v$, for $1 \leq v \leq \mathbf{v}S$. It is convenient to have a single visit root $\mathbf{v}S = 1$ since then only one visit-function has to be called, and no bindings have to be passed from visit to visit "manually". Fortunately, it is always possible to transform an attribute grammar for which the start symbol has more than one visit into an attribute grammar for which the start symbol has precisely one visit. The set of non-terminals is extended with a symbol S' that takes over the role of start symbol. Furthermore, a production $S' = \mathbf{root}'(S)$ is added. Symbol S' has only one inherited attribute and one synthesized attribute. Both are tuples of respectively all inherited and all synthesized attributes of S. The equations associated with $root'$ destruct the inherited and construct the synthesized tuple. From now on, we will assume that the root symbol of an attribute grammar is single visit.

Higher-order attribute grammars

Computing pure visit-functions for a higher-order attribute grammars proceeds in much the same way: the higher-order visit-sequences are augmented in analogy with plain visit-sequences. Then, the VSS mapping is applied to the higher-order visit-sequences.

When augmenting the higher-order visit-sequences, attributable attributes need special treatment. Like non-terminals, attributable attributes induce binding occurrences. The following set of *generated binding occurrences* associated with attributable attribute $p.x$ should be added to $BO(p)$:

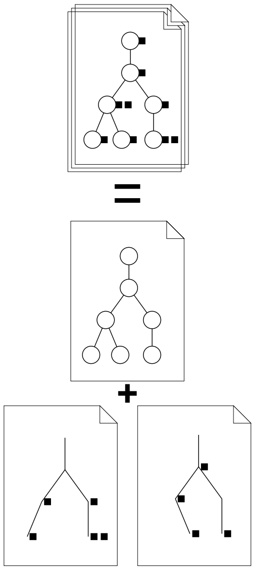
$$BO(p.x) = \{p.x.a^{v\to w} \mid X = R\ p.x \wedge X.\mathrm{a}^{v\to w} \in BA(X)\} \quad .$$

The associated equations are

$$BE(p.x) = \{\ (p.x.\mathrm{i}^{v\to w} := id\ p.x.\mathrm{s}^{v\to w})\ \mid p.x.\mathrm{i}^{v\to w} \in BO(p.x)\} \quad .$$

When augmenting a higher-order visit-sequence, the associated $eval(\beta)$ instructions (for each $\beta \in p.x\mathbf{BI}\,v$) are to be inserted just before the $visit(p.x, v)$ instruction in the plan of p.

There is one aspect that might easily be forgotten. An attributable attribute $p.x$ computed in compartment v might be visited in compartment w. This means of course, that $p.x$ should be transported form v to w. In other words, a visit to $p.x$ is a usage of $p.x$. As a consequence the set $use(p, w)$ should be extended with $\{p.x \mid visit(p.x, u)$ occurs in $vss(p, w)\}$ for every attributable attribute $p.x$.



3.4.5 Emptiness test for bindings

Given visit-sequences, an abstract syntax tree T induces the bindings $T^{v\to w}$, $1 \leq v < w \leq \mathbf{v}nont(T)$. A binding $T^{v\to w}$ is a tree with attribute instances, computed during visit v and used during visit w, as leaves. Bindings closely resemble the abstract syntax tree we pretended not to need anymore for storing attribute instances. For example, Figure 3.31$_{\triangleright}$ shows the binding that is constructed during the decorating of the VARUSE application. Bindings can be thought of as extracts of the tree that contain precisely those attribute instances needed for a later visit. These extracts can be "peeled" of, binding by binding.

If no intra-visit-dependencies occur between visit v and w of an abstract syntax tree T, then the induced *binding* $T^{v\to w}$ is said to be *empty*. An empty binding is an arbitrarily large term that contains no attribute instances. If, for every abstract syntax tree T with root X, the induced binding $T^{v\to w}$ is empty, then the *parcel* $X^{v\to w}$ is said to be *empty*.

Definition 15 *Empty parcel.*
A parcel $X^{v\to w}$ is empty if for every structure tree T with root X no intra-visit-dependency between visit v and w occurs.
□

Emptiness is an important property of parcels. Empty parcels, and their wrappers can be removed from the augmented grammar without altering the meaning with respect to plain attributes.

For example, the wrapper $p^{2\to 3}$ induced by production p in BINDING grammar contains the binding occurrences $X.\mathrm{s}^{2\to 4}$ and $X.\mathrm{s}^{3\to 4}$. Assuming that the parcels $X^{2\to 4}$ and $X^{3\to 4}$ are empty, these binding occurrences may be removed, making

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{visit}_N^2 :: N \times N^{1\to 2} \times \widehat{N.i2} \longrightarrow \widehat{N.s2}} \\
\multicolumn{2}{l}{\mathit{visit}_N^2\ \mathbf{p}(X)\ \mathbf{p}^{1\to 2}(N.i1, X.s1, X.\mathrm{s}^{1\to 2}, X.\mathrm{s}^{1\to 3})\ N.i2 = N.s2} \\
\quad \textbf{where}\ X.i2 & = N.i1 \oplus X.s1 \\
\qquad\quad ;\ X.i2' & = f3\ N.i2 \\
\qquad\quad ;\ X.\mathrm{i}^{1\to 2} & = \mathit{id}\ X.\mathrm{s}^{1\to 2} \\
\qquad\quad ;\ (X.s2, X.\mathrm{s}^{2\to 3}) & = \mathit{visit}_X^2\ X\ X.\mathrm{i}^{1\to 2}\ X.i2\ X.i2' \\
\qquad\quad ;\ X.i3 & = f4\ X.s2 \\
\qquad\quad ;\ X.\mathrm{i}^{1\to 3} & = \mathit{id}\ X.\mathrm{s}^{1\to 3} \\
\qquad\quad ;\ X.\mathrm{i}^{2\to 3} & = \mathit{id}\ X.\mathrm{s}^{2\to 3} \\
\qquad\quad ;\ X.s3 & = \mathit{visit}_X^3\ X\ X.\mathrm{i}^{1\to 3}\ X.\mathrm{i}^{2\to 3}\ X.i3 \\
\qquad\quad ;\ N.s2 & = f5\ X.s3 \\
\multicolumn{2}{l}{\mathit{visit}_N^2\ \mathbf{q}()\ \mathbf{q}^{1\to 2}(N.i1)\ N.i2 = N.s2} \\
\quad \textbf{where}\ N.s2 & = N.i1 * N.i2
\end{array}
$$

It is assumed that the parcels $X^{2\to 4}$ and $X^{3\to 4}$ are empty, and that p and q are the only productions on N. In that case, parcel $N^{2\to 3}$ is empty.

Figure 3.28. Visit-functions with only non-empty parcels

$p^{2\to 3}$ an empty wrapper. Assume further that q in the BINDING grammar is the only other production on N. Since $q^{2\to 3}$ is also an empty wrapper, all wrappers on the parcel $N^{2\to 3}$ are empty which makes the parcel $N^{2\to 3}$ empty. Removing the empty bindings from the BINDING grammar yields the visit-function given in Figure 3.28.

It is not known a priori whether an intra-visit-dependency exits between visit v and w of any X tree. Even if none of the productions on X have a border-crossing on their attribute occurrences, there might still be a child Y of such a production whose binding occurrences induce a border-crossing. That is, if these bindings occurrences are not occurrences of an empty binding.

We will now discuss an algorithm for determining the emptiness of parcels [Vog93]. By definition, a binding $X^{v\to w}$ is constructed by applying some wrapper $p^{v\to w}$ (where $p{\cdot}0 = X$) to attribute occurrences of p and binding occurrences of children of p. If an attribute occurrences is bound, then $p^{v\to w}$ and hence $X^{v\to w}$ is clearly not empty. If binding occurrence $p\mathbf{o}i.\mathrm{s}^{t\to u}$ is bound and the underlying parcel $p{\cdot}i^{t\to u}$ is not empty, then parcel $X^{v\to w}$ is not empty either.

These observations lead to Algorithm 3.29$_{\triangleright}$ that basically performs a transitive closure on a parcel dependency graph. The vertices of a parcel dependency graph are either parcels or attribute occurrences. The attribute occurrences are condensed into a single vertex $\perp$. The wrappers define the dependencies; binding occurrences are thereby converted to their underlying parcels. A binding $X^{v\to w}$ is empty if Algorithm 3.29$_{\triangleright}$ sets $\mathit{empty}(X^{v\to w})$ to true.

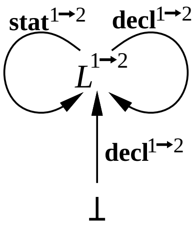
```
$G := \{\}$
for each production $p$ do
  for each pair $v, w$ such that $1 \le v < w \le \mathbf{v}p$ do
    if $O(p) \cap bind(p,v,w) \ne \{\}$ then $G := G \cup \{\bot \to p{\cdot}0^{v\to w}\}$
    for each binding occurrence $p\mathbf{o}i.\mathrm{a}^{t\to u} \in bind(p,v,w)$ do $G := G \cup \{p{\cdot}i^{t\to u} \to p{\cdot}0^{v\to w}\}$
$G := G^+$
for each non-terminal $X$ do
  for each binding $X^{v\to w} \in BA(X)$ do
    $empty(X^{v\to w}) := \bot \to X^{v\to w} \notin G$
```

Algorithm 3.29. The emptiness test for parcels

3.4.6 Reviewing the varuse and box grammar

In the varuse grammar, non-terminal L is visited twice which induces one binding $L^{1\to2}$. Three productions on L exist, each of which induces a wrapper: $\mathbf{decl}^{1\to2}$, $\mathbf{stat}^{1\to2}$ and $\mathbf{empty}^{1\to2}$. Only production $decl$ has an intra-visit-dependency, thus only $\mathbf{decl}^{1\to2}$ binds an attribute instance. The bind grammar has the following form:

$$\begin{aligned} L^{1\to2} &= \mathbf{decl}^{1\to2}(str, L^{1\to2}) \\ &\mid\ \mathbf{stat}^{1\to2}(L^{1\to2}) \\ &\mid\ \mathbf{empty}^{1\to2}() \quad . \end{aligned}$$



The parcel dependency graph associated with the bind grammar induced by the varuse grammar has only two vertices: parcel $L^{1\to2}$ and the attribute occurrence $decl\mathbf{o}1.id$. The single attribute occurrence is "condensed" into the vertex $\bot$. Each right-hand side symbol in any wrapper induces an arc in the parcel dependency graph; in this example we thus have three arcs: $\bot \xrightarrow{\mathbf{decl}^{1\to2}} L^{1\to2}$, $L^{1\to2} \xrightarrow{\mathbf{decl}^{1\to2}} L^{1\to2}$ and $L^{1\to2} \xrightarrow{\mathbf{stat}^{1\to2}} L^{1\to2}$. Since $\bot$ is a predecessor of the parcel $L^{1\to2}$, it is not empty.

The plan icons associated with augmented visit-sequences are known as augmented plan icons. When pasted together, they form an augmented plan tree. The augmented plan tree associated with the varuse application is given in Figure 3.30. The difference with the plain plan tree in Figure 3.21$_{\triangleleft 67}$ are the binding attributes denoted with boxes with rounded corners.

See the marginal note on page 29 for node $[1,2,1]$.

Figure 3.31 presents the binding computed during the decoration of the standard varuse application. Only one attribute instance is stored, namely $([1,2,1], N.id)$. For completeness and future reference, the visit-functions and data types for the varuse grammar are given in Figure 3.32$_{\triangleright}$.

· · ·

The plan icons of the box grammar are given in Figure 3.33$_{\triangleright 86}$. With the plan icons given, it is fairly simple to determine the following bind grammar.

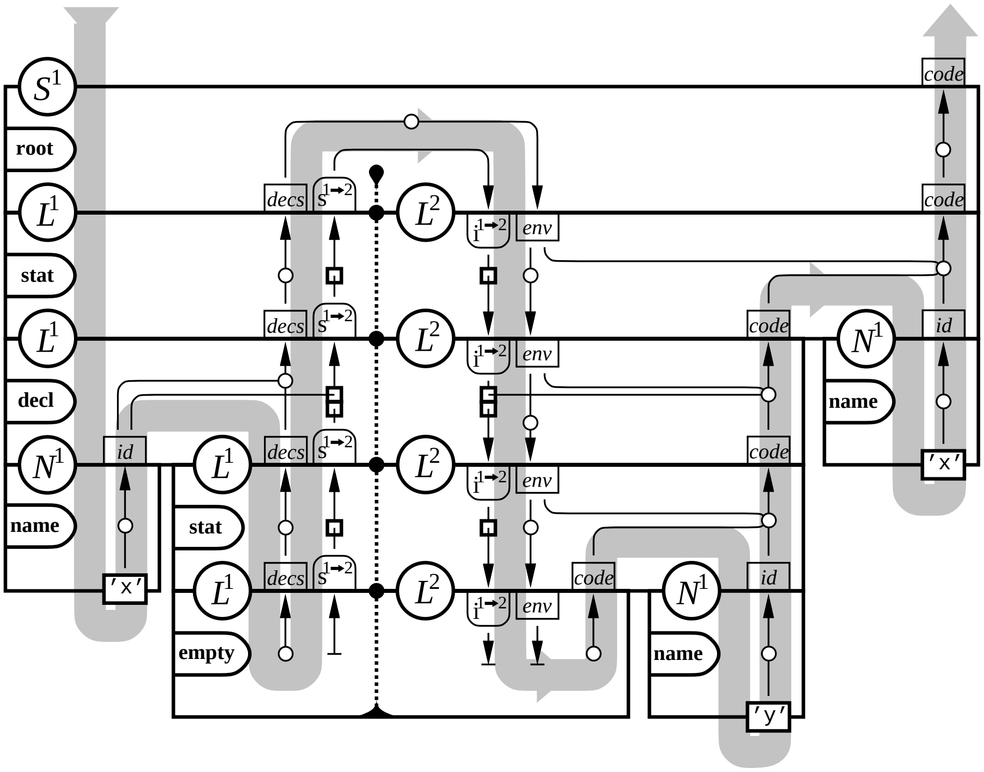


Figure 3.30. The augmented plan tree for the VARUSE application

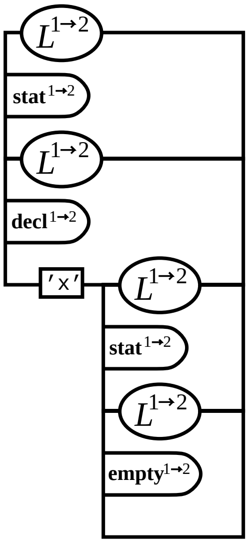


Figure 3.31. Binding

```
$\textbf{type}\ S = \textbf{root}(L)$
$\textbf{type}\ L = \textbf{decl}(N, L)$
        $|\ \textbf{stat}(N, L)$
        $|\ \textbf{empty}()$
$\textbf{type}\ N = \textbf{name}(str)$

$visit^1_S :: S \longrightarrow Code$
$visit^1_S\ \textbf{root}(L) = S.code$
  $\textbf{where}\ (L.decs, L.\mathrm{s}^{1\to2}) = visit^1_L\ L$
        $;\ L.env = L.decs$
        $;\ L.\mathrm{i}^{1\to2} = L.\mathrm{s}^{1\to2}$
        $;\ L.code = visit^2_L\ L\ L.\mathrm{i}^{1\to2}\ L.env$
        $;\ S.code = L.code$

$visit^1_L :: L \longrightarrow Env \times L^{1\to2}$
$visit^1_L\ \textbf{decl}(N, L_2) = (L_1.decs, L_1.\mathrm{s}^{1\to2})$
  $\textbf{where}\ N.id = visit^1_N\ N$
        $;\ (L_2.decs, L_2.\mathrm{s}^{1\to2}) = visit^1_L\ L_2$
        $;\ L_1.decs = N.id : L_2.decs$
        $;\ L_1.\mathrm{s}^{1\to2} = \textbf{decl}^{1\to2}(N.id, L_2.\mathrm{s}^{1\to2})$
$visit^1_L\ \textbf{stat}(N, L_2) = (L_1.decs, L_1.\mathrm{s}^{1\to2})$
  $\textbf{where}\ (L_2.decs, L_2.\mathrm{s}^{1\to2}) = visit^1_L\ L_2$
        $;\ L_1.decs = L_2.decs$
        $;\ L_1.\mathrm{s}^{1\to2} = \textbf{stat}^{1\to2}(L_2.\mathrm{s}^{1\to2})$

$visit^1_L\ \textbf{empty}() = (L_1.decs, L_1.\mathrm{s}^{1\to2})$
  $\textbf{where}\ L_1.decs = [\,]$
        $;\ L_1.\mathrm{s}^{1\to2} = \textbf{empty}^{1\to2}()$
```

```
$\textbf{type}\ L^{1\to2} = \textbf{decl}^{1\to2}(str, L^{1\to2})$
        $|\ \textbf{stat}^{1\to2}(L^{1\to2})$
        $|\ \textbf{empty}^{1\to2}()$

$visit^1_N :: N \longrightarrow str$
$visit^1_N\ \textbf{name}(str) = N.id$
  $\textbf{where}\ N.id = upstring\ str$

$visit^2_L :: L \times L^{1\to2} \times Env \longrightarrow Code$
$visit^2_L\ \textbf{decl}(N, L_2)\ \textbf{decl}^{1\to2}(N.id, L_2.\mathrm{s}^{1\to2})\ L_1.env = L_1.code$
  $\textbf{where}\ L_2.env = L_1.env$
        $;\ L_2.\mathrm{i}^{1\to2} = L_2.\mathrm{s}^{1\to2}$
        $;\ L_2.code = visit^2_L\ L_2\ L_2.\mathrm{i}^{1\to2}\ L_2.env$
        $;\ L_1.code = (-lookup\ L_1.env\ N.id) : L_2.code$
$visit^2_L\ \textbf{stat}(N, L_2)\ \textbf{stat}^{1\to2}(L_2.\mathrm{s}^{1\to2})\ L_1.env = L_1.code$
  $\textbf{where}\ L_2.env = L_1.env$
        $;\ L_2.\mathrm{i}^{1\to2} = L_2.\mathrm{s}^{1\to2}$
        $;\ L_2.code = visit^2_L\ L_2\ L_2.\mathrm{i}^{1\to2}\ L_2.env$
        $;\ N.id = visit^1_N\ N$
        $;\ L_1.code = (+lookup\ L_1.env\ N.id) : L_2.code$
$visit^2_L\ \textbf{empty}()\ \textbf{empty}^{1\to2}()\ L_1.env = L_1.code$
  $\textbf{where}\ L_1.code = [\,]$
```

Figure 3.32. Pure visit-functions and data types for the VARUSE grammar

$$\begin{array}{rcl}
B^{1\to 2} & = & \mathbf{box}^{1\to 2}(\mathit{font}) \\
& | & \mathbf{sidet}^{1\to 2}(\mathit{real}, B^{1\to 2}, B^{1\to 2}) \\
& | & \mathbf{stackl}^{1\to 2}(\mathit{real}, B^{1\to 2}, B^{1\to 2}) \\
& | & \mathbf{sideb}^{1\to 2}(\mathit{real}, \mathit{real}, \mathit{real}, \mathit{real}, B^{1\to 2}, B^{1\to 2}) \\
& | & \mathbf{stackr}^{1\to 2}(\mathit{real}, \mathit{real}, \mathit{real}, \mathit{real}, B^{1\to 2}, B^{1\to 2})
\end{array}$$

Note that in productions $\mathbf{sidet}$, $\mathbf{stackl}$, $\mathbf{sideb}$ and $\mathbf{stackr}$ both children induce an occurrence of binding $B^{1\to 2}$ that must be transported from the first compartment to the second. Thus, the "first" $B^{1\to 2}$ binds $B_2.\mathrm{s}^{1\to 2}$ and the "second" binds $B_3.\mathrm{s}^{1\to 2}$ (or rather $\mathit{sidet}\mathbf{o}2.\mathrm{s}^{1\to 2}$ respectively $\mathit{sidet}\mathbf{o}3.\mathrm{s}^{1\to 2}$).

A minor flaw in our drawing formalism can be observed in the plan icon of the box production. The arc $\mathit{str} \to \mathit{dlist}$ appears to cross a visit-border so that one would expect a lightning symbol. However, str is not an attribute occurrence computed in the first compartment; it is a syntactic element available in all compartments.

3.4.7 Memoization

Visit-functions have become pure functions due to the introduction of bindings. Static analysis for bindings is cumbersome. However, due to the pure nature of the visit-functions, incremental evaluation can now be obtained by memoizing the visit-functions. This moves the bookkeeping for incremental evaluation into a separate module for implementing function caching.

Memoization is implemented by associating a table (cache) with each function that records argument/result pairs. Each time a memoized function is called, the computed result together with the argument is stored in the table. A memoized function is said to *hit* when it is called with an argument that happens to occur in the table. Then, instead of actually evaluating the function, the associated result can be retrieved from the table. A memoized function is said to *miss* when its argument is not yet stored in the table.

One of the major problems of visit-sequence based incremental evaluation, namely when to skip a visit instruction (see Subsection 3.3.3), is implicitly solved by the introduction of bindings. The binding occurrence $\mathbf{po}i\mathbf{BI}\,v$ contain precisely the attribute instances computed by earlier visits that will be used by $\mathit{visit}(i, v)$. Thus visit v to $\mathbf{po}i$ can be skipped only if the bindings $\mathbf{po}i\mathbf{BI}\,v$—and the standard inherited attributes $\mathbf{po}i\mathbf{I}\,v$—have not changed, and the replaced subtree is not a subtree of $\mathbf{po}i$. A visit-functions has precisly these three arguments (the abstract syntax tree, the bindings and the inherited attributes). They all contribute to the decision whether the visit-function "hits" (is found in the cache). Therefore, memoized visit-functions implement an optimal scheme for skipping visits. Further reducing the number of visits is only possible in two directions: by taking the *semantics* of the dpr arcs into account, or by optimizing the role of data flow driver of the abstract syntax tree.

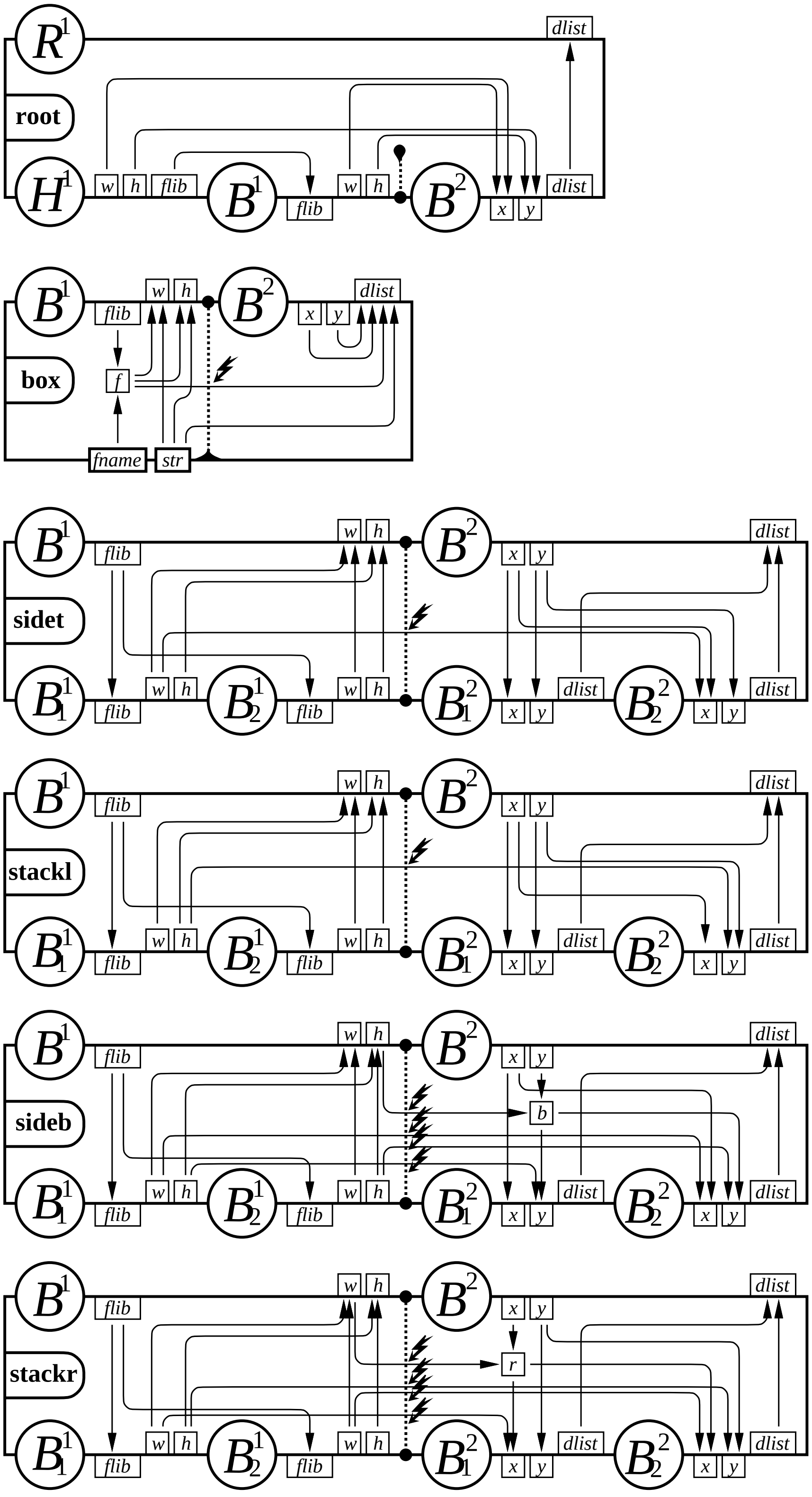
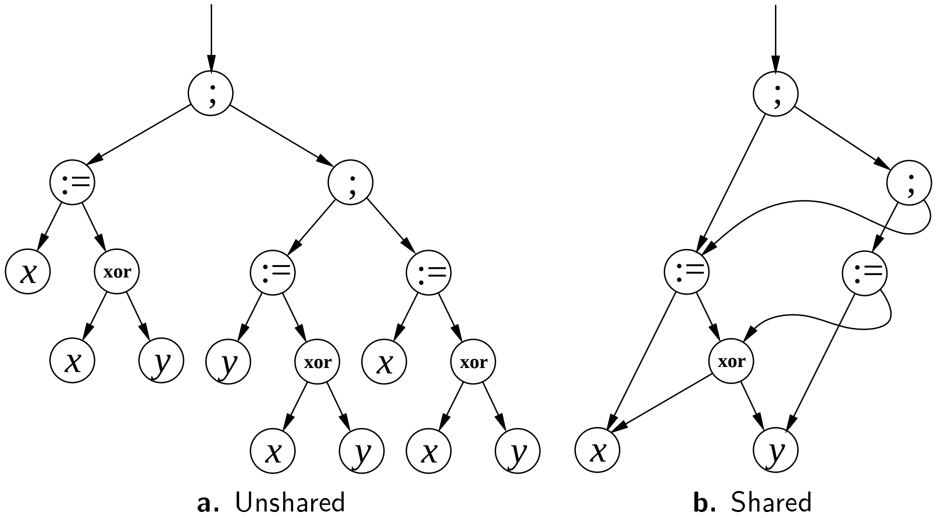


Figure 3.33. The plan icons of the BOX grammar



a. Unshared **b.** Shared

Figure 3.34. Terms for the fragment "$x := x$ **xor** y; $y := x$ **xor** y; $x := x$ **xor** y"

The use of bindings for optimal visit skipping is not a privilege of functional evaluators. Traditional evaluators may incorporate bindings for incremental evaluation. In fact, in traditional evaluators, the bindings need not actually be computed since the attribute instances may all be found in the tree. It suffices to synthesize a boolean for every binding that indicates whether the binding would have changed, had it been computed.

A second problem concerning incremental evaluation has not been discussed yet. Efficient incremental computation relies heavily on efficient equality test of attribute instances. For simple instances like integers equality can be efficiently decided. For structured attribute instances like symbol tables, the equality test can be extremely inefficient. Note that bindings are treated as generated attributes that must also be compared, and that, in general, bindings are large term-like structures. Furthermore, the abstract syntax tree itself is a parameter for the visit-functions. It must also be checked for equality. Consequently, the equality problem is even more salient in a functional setting. Fortunately, the purely functional approach allows us to *share* values. If we make sure that equal terms are always represented by the *same* "internal" structure, then term comparison can be implemented by pointer comparison.

For example, consider the term in Figure 3.34a. It represents the PASCAL fragment $x := x$ **xor** y; $y := x$ **xor** y; $x := x$ **xor** y that swaps the values of x and y. Sharing of subtrees will yield the structure given in Figure 3.34b. Sharing is obtained by *memoizing the constructors*. Shared terms may also be viewed as *directed acyclic graphs*, DAGs for short.

In a standard attribute avaluator, the abstract syntax tree has a double role. It serves as a data flow driver selecting which (plan-)instructions are to be executed and it serves as a cache for the semantic functions. In a functional setting the abstract tree only serves as data flow driver. It can be optimized for this function as will be illustrated in the next chapter. The role of cache is taken over by the memo table. It can be optimized separately. For example, the size of the cache tunes the time-space trade-off for an evaluator. Observe that the cache not only records the previous state but, theoretically, *all* prevous states. An edit action $T \longrightarrow T' \longrightarrow T$ can be very inefficient in a conventional setting. In our functional setting, the second edit transition takes only $\mathcal{O}(1)$, provided the cache is large enough to keep both T and T'.

3.4.8 An attribute grammar system

The concepts for functional evaluation have now been discussed. We will sketch how to obtain a compiler generator that generates a functional evaluator.

The compiler generator reads, parses and checks an attribute grammar specification. Next, visit-sub-sequences are generated. The lifetime of attribute occurrences and binding occurrences is analyzed defining pessimistic wrappers. The emptyness test eliminates occurrences from empty bindings from the wrappers.

Part 1: visit-functions

Part 2: data types

Part 3a: parser

Part 3b: editor

The generated visit-functions form the main part of the generated compiler. The second module deals with the data structures and constructors for implementing the abstract syntax tree, the bindings and structured attributes. A third module implements either the parser (for a batch compiler) or a language based editor (for an interactive compiler).

Part 4: caches

When the generated compiler has to be incremental, which is desirable if it is interactive, a fourth module must be added. It implements the memoization of the constructors for the abstract syntax tree, bindings and structured attributes. Furthermore, it implements a cache for the visit-functions.

Chapter 4

New techniques for incremental evaluation

This chapter discusses several ways of transforming an abstract syntax tree while preserving the functional dependency between the inherited and synthesized attributes of the root of that tree. The role of tree as data flow driver is optimized.

First, *splitting* is introduced: the abstract syntax tree is split into different representations for different visits with the aim to improve incremental behavior. Then, *encapsulators* are introduced. Encapsulators replace the actual tree constructors. They construct an alternative abstract syntax tree that can be evaluated more efficiently. Several encapsulator applications will be discussed: *elimination*, *unification*, *folding* and *normalization*.

4.1 Introduction

In this chapter we discuss techniques that aim at increasing the chance on visit-function cache hits. The *arguments* of a function play a key role during memoization; they determine whether the function call hits or misses.

Non-injective functions allow for cache-optimizations. For example, suppose that $square$ is memoized. We obtain a more efficient squaring function if we precede $square$ by abs, which returns the absolute value of its argument, in the following manner (functional composition)

$$square' = square \circ abs \quad .$$

The definition of $square'$ utilizes the property that $square$ is not injective: $\forall_x square\ -x = square\ +x$. Assuming that abs is more efficient then $square$, this definition saves computation time by increasing the change on cache hits.

Visit-functions are in general not injective. For example consider the VARUSE grammar. Whether we apply $visit_L^1$ to the argument $\mathbf{stat}(\mathbf{name}(\text{'x'}), \mathbf{empty}())$ or to $\mathbf{stat}(\mathbf{name}(\text{'y'}), \mathbf{empty}())$ the result remains $([\,], \mathbf{stat}^{1\to 2}(\mathbf{empty}^{1\to 2}()))$, namely an empty symbol table and an empty binding. Since visit-functions are not injective, there is a possibility to increase the chance on cache hits by mapping different arguments with the same results to equal arguments.

For visit-functions, we distinguish the following arguments: the abstract syntax tree, bindings and the inherited attributes. A visit-function can be non-injective in each of these arguments. However, in order to make use of this property for inherited attributes and bindings we need to investigate the nature of the semantic functions. Such analysis is beyond the scope of this thesis. We will make one exception though. Section 4.3 discusses optimizations applicable when a semantic function happens to be the identity function.

The identity function id is defined as $id\,x = x$ for x of any type (id is polymorphic).

In Section 4.2 we focus on the first argument of a visit-function: the abstract syntax tree. As we illustrated above with the VARUSE grammar, visit-functions are non-injective in the tree argument. The reason for this is that in general, a specific visit to an abstract syntax tree does not inspect every node of that tree. In the VARUSE grammar, $name$ nodes are not inspected during the first visit to $stat$ nodes. This observation is especially fruitful in incremental evaluation: a change to an uninspected node would cause the visit-function with otherwise unchanged arguments to miss if no special precautions were taken.

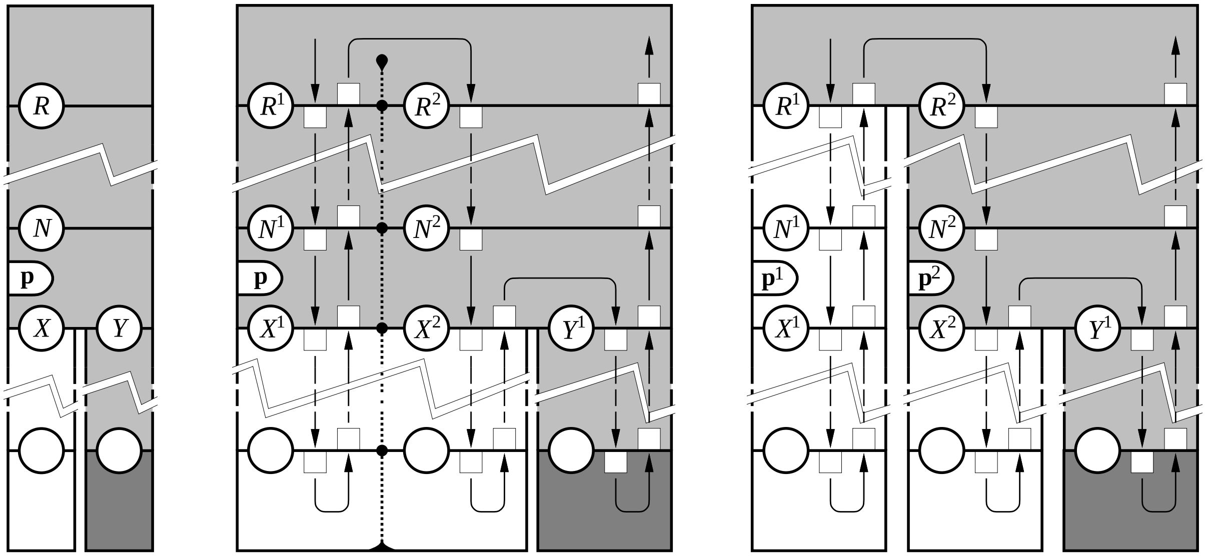
4.2 Splitting

A *split* abstract syntax tree T is a tuple $[\,T^1, \ldots, T^{\mathrm{v}\,T}\,]$ of terms [PSV92a]. Term T^v includes only those parts of T that are inspected during visit v of T. The underlying principle is that by including only those parts that are actually used during a visit, unnecessary cache misses can be avoided.

We will illustrate the process of splitting with the abstract syntax tree given in Figure 4.1a. The grammar itself remains anonymous; all important details can be read from the figure. We focus on production $N = \mathbf{p}(X, Y)$ and non-terminal R. Decoration occurs according to the plan tree given in Figure 4.1b. Observe that non-terminals R, N and X are visited twice, whereas non-terminal Y is visited only once. During the first visit to a p node the X child is visited for the first time; during the second visit to that p node, the X child is visited for the second time upon which Y is visited. In other words, the subtree rooted Y does not play a role during the first visit to p.

We assume the abstract syntax tree is decorated using the memoized visit-functions described in the previous chapter. As a result of decoration, the visit-function cache is filled with among others, entries for the following visit-functions: $visit^1_R$, $visit^1_N$, $visit^1_X$, $visit^2_R$, $visit^2_N$, $visit^2_X$ and $visit^1_Y$.

Suppose, an edit operation issued by the user changes a subtree of the Y node that is marked dark grey in Figure 4.1a. The nodes of the abstract syntax tree from the changed subtree to the root is marked light grey as is the corresponding part in Figure 4.1b. Redecoration using memoized visit-functions will reevaluate every equation in the grey part of the plan tree. The reason for this is as follows: the dark grey part changes the Y tree, which in turn changes the N tree (since



a. Abstract syntax tree **b.** Plan tree **c.** Split abstract syntax tree

The dark grey part of the tree is changed by an edit action of the user. The light grey part of the tree includes the dark part and is therefore also subject to reevaluation.

Figure 4.1. Splitting the abstract syntax tree

Y is a child of N) which changes the R tree. So for each of the visit-functions $visit^1_R$, $visit^1_N$, $visit^2_R$, $visit^2_N$ and $visit^1_Y$, one of the arguments (namely the tree) has changed which requires the recomputation of the visit-function. Only the visit-functions $visit^1_X$ and $visit^2_X$ will lead to a cache hit. The nodes associated with these visits are kept white in Figure 4.1.

Closer observation however, reveals that we may skip recomputation of $visit^1_R$. As the plan tree shows, the first visit to R does not inspect the changed subtree Y. It is only during the second visit to R that the second visit to N is executed which includes a visit to the changed subtree. Consequently, if only we could detect that Y is not included in the first visit to R, we could reuse the previously computed results stored in the cache.

We propose to *split* an abstract syntax tree T, that is visited v times (rephrased $v = \mathbf{v}\,T$) into trees $T^1 \dots T^v$ that each include only those parts of T that are actually inspected by the respective visits $visit^1_T \dots visit^v_T$.

Figure 4.1c shows the split abstract syntax tree, corresponding to the abstract syntax tree given in Figure 4.1a. Observe that the trees rooted R, N and X are each represented by two trees, rooted R^1, R^2 respectively N^1, N^2 and X^1, X^2. The Y tree is visited only once, and is thus represented by a single tree rooted Y^1.

Do not confuse the plan tree, which shows data flow, with the split abstract syntax tree, a data structure which is visited (a data flow *driver*). Whereas in the former N^2 denotes visit 2 to N, in the latter N^2 is a term constructed by applying *split constructors* like $\mathbf{p}^2$, to terms rooted with *split non-terminals* like X^2 and Y^1.

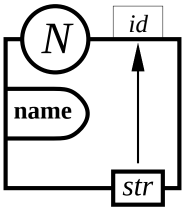
The crux of splitting in this example is that the *split tree* rooted R^1 does not include the changed subtree Y^1. In other words, a change to Y^1 has no repercussions for the R^1 tree: $visit^1_R$ will hit. Note that the subtree rooted R^1 can be arbitrarily large.

4.2.1 The split grammar, $split$ function and SPLIT mapping

Splitting requires four steps. First, we must perform a "split analysis" on the visit-sequences to determine which part of the abstract syntax tree is actually inspected during a visit. Next, we define a grammar for split trees. Thirdly, we define a function that converts an abstract syntax tree into a split tree. Finally, we define the mapping SPLIT from visit-functions to split visit-functions. Split visit-functions operate on split trees instead of plain abstract syntax trees.

Split analysis

In the introduction, we have not been precise. We informally defined a split tree T^v as a term that only includes those parts of the abstract syntax tree T that are *inspected* during visit v to T. Although we left out to define "inspected", we implied that it means "visited". This is only half true.



Another way to inspect parts of the abstract syntax tree is via a syntactic reference. A syntactic reference is a dependency of an attribute occurrence on a syntactic element, that is to say a non-terminal or pseudo terminal. For example, in the VARUSE grammar, the set $E(name)$ contains the equation $N.id := upstring\ str$ which should formally be rendered as $name\mathbf{o}0.id := upstring\ name\mathbf{o}1$. The pseudo terminal $name\mathbf{o}1$ is inspected.

During decoration of node K, a syntactic reference like $p\mathbf{o}i.a := p\mathbf{o}j$ equates the entire *subtree* $K{\cdot}j$ with attribute instance $ins(K, p\mathbf{o}i.a)$. Tree values may be destructed by semantic functions or they may be assigned to attributable attributes. In either case, the *entire* subtree needs to be available. As a consequence, the terminals of the split grammar correspond to subtrees of the underlying abstract syntax tree.

Annotated visit-sequences are defined on page 74. Amongst others, they include $sref$ directives flagging syntactic references. The set $inspect(p, v)$ contains all split non-terminal occurrences that are visited and all syntactic elements that are referenced in visit-sub-sequence v of production p. It is defined as follows.

$p\mathbf{o}i^w$ denotes visit w to non-terminal occurrence $p\mathbf{o}i$.

$$\begin{aligned} inspect(p, v) \;&=\; \{p\mathbf{o}i^w \mid visit(i, w) \text{ occurs in } vss(p, v)\} \\ &\cup\; \{p\mathbf{o}i \mid sref(p\mathbf{o}i) \text{ occurs in } vss(p, v)\} \end{aligned}$$

Production p of the grammar in Figure 4.1$_\triangleleft$ has no syntactic references, so only the $visit$ instructions matter. Since the first compartment of p only contains the first visit to X and the second compartment of p contains the second visit to X and the first visit to Y, we have the following $inspect$ sets

Recall that Y^1 is informal notation for $p\mathbf{o}2^1$.

$$
\begin{aligned}
\mathit{inspect}(p,1) &= \{X^1\} \\
\mathit{inspect}(p,2) &= \{X^2, Y^1\} \quad .
\end{aligned}
$$

The split grammar

In the previous chapter bindings were introduced. Bindings are described by a context-free grammar, the bind grammar, induced by the attribute grammar and its visit-sequences. In a similar fashion an attribute grammar induces a split grammar. However, where the bind grammar defines an additional data structure passed around in binding attributes, the split grammar defines split trees that replace the abstract syntax tree. The following table shows the relation between the attribute grammar, the bind grammar and the split grammar.

attribute grammar	**bind grammar**	**split grammar**
	$1 \leq v < w \leq \mathbf{v}\{T,p,X\}$	$1 \leq v \leq \mathbf{v}\{T,p,X\}$
tree T	bindings $T^{v\rightarrow w}$	split trees T^v
production p	wrapper $p^{v\rightarrow w}$	split productions p^v
non-terminal X	parcels $X^{v\rightarrow w}$	split non-terminals X^v
terminals	attribute instances	abstract syntax trees

We will define the split grammar in analogy with the bind grammar. The split representation $\underline{T}$ of an abstract syntax tree T is a $\mathbf{v}T$-tuple of terms $\lgroup T^1, \ldots, T^{\mathbf{v}T} \rgroup$. A term T^v, with $1 \leq v \leq \mathbf{v}T$, is known as a *split tree* (for visit v). Each non-terminal X induces a type $\underline{X}$ defined as $\lgroup X^1, \ldots, X^{\mathbf{v}X} \rgroup$; X^v with $1 \leq v \leq \mathbf{v}X$ is known as a *split non-terminal*. The root of a split tree is a split non-terminal. A pseudo-terminal is not split. For each occurrence $p\mathbf{o}i$ of non-terminal X, there is a *split non-terminal occurrence* denoted $p\mathbf{o}i^v$ for every $1 \leq v \leq \mathbf{v}X$.

We will use straight parenthesis $\lgroup\,\rgroup$ for tuples in the split grammar.

In order to construct split trees from the split non-terminals, we need *split productions*. In a similar fashion as the set $\mathit{bind}(p,v,w)$ defines the right-hand side of the wrapper $p^{v\rightarrow w}$ in the bind grammar, the set $\mathit{inspect}(p,v)$ defines the right-hand side of the split production p^v $(1 \leq v \leq \mathbf{v}p)$:

Set to list conversion: just pick an order.

$$
p^v = p{\cdot}0^v : [\hat{\eta} \mid \eta \in \mathit{inspect}(p,v)] \quad .
$$

This definition consists of two parts. The first part, $p{\cdot}0^v$, is the left-hand side split non-terminal. The second part is a list comprehension that defines the right-hand side. Every element of $\mathit{inspect}$ is mapped onto its type: $\hat{\eta}$ denotes the type of η. For example, for a split non-terminal occurrence $p\mathbf{o}i^v$, $\widehat{p\mathbf{o}i}^v$ maps to the split non-terminal $p{\cdot}i^v$. For a non-terminal occurrence, $\widehat{p\mathbf{o}i}$ maps to $p{\cdot}i$.

For the grammar in Figure 4.1$_{\triangleleft}$ we obtain the following definitions. Non-terminals R, N and X are visited twice, thus their split representations are two-tuples: $\underline{R} = \lgroup R^1, R^2 \rgroup$, $\underline{N} = \lgroup N^1, N^2 \rgroup$ and $\underline{X} = \lgroup X^1, X^2 \rgroup$. The split representation for the single visit non-terminal Y is a one-tuple $\underline{Y} = \lgroup Y^1 \rgroup$. Production p induces two split production p^1 and p^2 since $\mathbf{v}p = 2$. Using the equalities $inspect(p, 1) = \{X^1\}$ and $inspect(p, 2) = \{X^2, Y^1\}$, we infer that the split productions induced by p have the form $N^1 = \mathbf{p}^1(X^1)$ respectively $N^2 = \mathbf{p}^2(X^2, Y^1)$.

The *split* function

We will define functions that split an abstract syntax tree. For each non-terminal X, a function $split_X :: X \longrightarrow \underline{X}$ is defined. Each production p on X induces an alternative definition for $split_X$. The alternative definitions are selected using pattern matching on constructor $\mathbf{p}$. The alternative for production p has the following form

$split_X$ is a *homomorphism* mapping X values to $\underline{X}$ values. Observe that $split_X$ is half-bijective in the sense that there exists a function f such that $X = f\ (split_X\ X)$ for all X.

$$split_X\ \mathbf{p}(\text{“children”}) = [\mathbf{p}^1(\text{“params 1”}), \ldots, \mathbf{p}^{\mathbf{v}X}(\text{“params }\mathbf{v}\text{X”})]$$
$$\quad \mathbf{where}\ \text{“recurse for children”} \quad .$$

For each non-terminal on the right-hand side of p a *pattern variable* $p\mathbf{o}i$ occurs in “children”. Furthermore, each $p\mathbf{o}i$ is split with a recursive call in the where-clause “recurse for children”: $[p\mathbf{o}i^1, \ldots, p\mathbf{o}i^{\mathbf{v}p\mathbf{o}i}] = split_{p\cdot i}\ p\mathbf{o}i$.

The fragment “params v” is determined by $inspect(p, v)$: each element of the set $inspect(p, v)$ maps to a parameter for “params v”. An element $p\mathbf{o}i^v$ refers to a split tree computed by the recursive call in “recurse for children”. An element $p\mathbf{o}i$ refers to a subtree of the abstract syntax tree that matched with the pattern $\mathbf{p}$(“children”).

Let us return to the example in Figure 4.1$_{\triangleleft 91}$ and derive the $split_N$ function. Only one production on N, namely p, is known, so we obtain only one alternative for $split_N$.

$$\begin{aligned}
&split_N :: N \longrightarrow [N^1, N^2] \\
&split_N\ \mathbf{p}(X, Y) = [\ \mathbf{p}^1(X^1)\ ,\ \mathbf{p}^2(X^2, Y^1)\ ] \\
&\quad \mathbf{where}\ [X^1, X^2] = split_X\ X \\
&\qquad\qquad ;\ [Y^1] \qquad = split_Y\ Y
\end{aligned}$$

The SPLIT mapping

The FUN mapping discussed in the previous chapter generates a visit-function $visit_X^v$ to handle visit v of an abstract syntax tree T rooted X. The function $split_X$ applied to T returns a tuple $[T^1, \ldots, T^v, \ldots, T^{\mathbf{v}X}]$. Split tree T^v rooted X^v contains precisely those parts of T that are inspected by visit v. Therefore, we would like to use T^v instead of T as first parameter for $visit_X^v$. Consequently, the signature of $visit_X^v$ should be changed so that not X but X^v is the type of the first parameter.

The SPLIT mapping maps pure visit-functions, as generated by the FUN mapping, onto *split visit-functions*. Split visit-functions are visit-functions whose first argument is a split tree instead of a plain abstract syntax tree. In other words, the visit-function $visit_X^v$ will be changed into $visit_{X^v}$ with signature

$$visit_{X^v} :: X^v \times \text{“}X\mathbf{BI}\,v\text{”} \times \text{“}X\mathbf{I}\,v\text{”} \longrightarrow \text{“}X\mathbf{S}\,v\text{”} \times \text{“}X\mathbf{BS}\,v\text{”} \quad .$$

A pattern $\mathbf{p}(\ldots)$ selecting an alternative for $visit_X^v$ is mapped to the pattern $\mathbf{p}^v($"params v"$)$ for $visit_{X^v}$. Each element of the set $inspect(p, v)$ maps to a parameter for "params v".

Equations ($eval$ instructions) in the body of a visit-function remain unchanged. If they refer to a syntactic element $\mathbf{po}i$ then $\mathbf{po}i \in inspect(p, v)$, so that $\mathbf{po}i$ occurs in "params v". Furthermore, a recursive call ($visit$ instruction) $visit^w_{p \cdot i}\ \mathbf{po}i \ldots$ in the body of a visit-function is mapped onto $visit_{p \cdot i^w}\ \mathbf{po}i^w \ldots$

. . .

The split grammar describes the abstract syntax of split trees. The $split$ function actually constructs a split tree, that is destructed by a split visit-function. Splitting will be illustrated shortly using the VARUSE grammar which includes a syntactic reference.

The elements of $inspect(p, v)$ play a recurring role in the three topics discussed. In the split grammar, a $visit(i, w)$ instruction induces a right-hand side split non-terminal $p \cdot i^w$ in the split production p^v. In the $split$ function, it maps to a variable $\mathbf{po}i^w$ that holds a split tree rooted $p \cdot i^w$. In the split visit-function the corresponding pattern variable $\mathbf{po}i^w$ is introduced to match that split tree rooted $p \cdot i^w$. Likewise, a syntactic reference to $\mathbf{po}i$ induces a right-hand side non-terminal $p \cdot i$ in the split production p^v. In the $split$ function, it maps to a variable $\mathbf{po}i$ that holds a subtree of the underlying abstract syntax tree rooted $p \cdot i$. In the split visit-function the corresponding pattern variable $\mathbf{po}i$ is introduced to match that subtree rooted $p \cdot i$.

Since every $visit(i, w)$ instruction ($1 \le i \le \mathbf{s}p$ and $1 \le w \le \mathbf{v}p \cdot i$) occurs precisely once in the plan of p, every split non-terminal occurrence $\mathbf{po}i^w$ occurs only once in $\bigcup_{1 \le v \le \mathbf{v}p} inspect(p, v)$. In other words, for every production p, the sets $V_v = \{\mathbf{po}i^w \mid \mathbf{po}i^w \in inspect(p, v)\}$ partition $\{\mathbf{po}i^w \mid 1 \le i \le \mathbf{s}p \wedge 1 \le w \le \mathbf{v}p \cdot i\}$.

The syntactic references that are inspected might have multiple occurrences. For every production p, the sets $W_v = \{\mathbf{po}i \mid \mathbf{po}i \in inspect(p, v)\}$ are generally not disjunct, nor do they unite to $\{\mathbf{po}i \mid 1 \le i \le \mathbf{s}p \wedge 1 \le w \le \mathbf{v}p \cdot i\}$.

Recall that we assume the start symbol S of an attribute grammar to be single visit. Thus, in order to decorate an abstract syntax tree T rooted S using the split visit-functions, the function $visit_S^1\ (split_S\ T)$ must be called with the appropriate inherited attributes.

A *decorator* is a context-free grammar G and a function f that takes as argument a tree T rooted with the start symbol S of G. Decorator (G', f') is a *refinement* of decorator (G, f) if there exists an *abstraction* ϕ from G trees to G' trees such that $f\ T = f'\ T'$, where $T' = \phi\ T$.

$$\begin{tikzcd}
T \text{ of } G \arrow[r, "f"] \arrow[d, "\phi"'] & (S\mathbf{I}1 \rightarrow S\mathbf{S}1) \arrow[d, "id"] \\
T' \text{ of } G' \arrow[r, "f'"'] & (S\mathbf{I}1 \rightarrow S\mathbf{S}1)
\end{tikzcd}$$

For example, the underlying context-free grammar of an attribute grammar, and the computed visit-functions form a decorator. The induced split grammar and the split visit-functions also form a decorator. The function $split_S$ (where S is the root symbol of the context-free grammar) is an abstraction. As a consequence, the split decorator is a refinement of the plain decorator.

$$\begin{aligned}
&\mathit{inspect}(\mathit{root},1) &&= \{L^1, L^2\}\\
&\mathit{inspect}(\mathit{decl},1) &&= \{N^1, L^1\}\\
&\mathit{inspect}(\mathit{decl},2) &&= \{L^2\}\\
&\mathit{inspect}(\mathit{stat},1) &&= \{L^1\}\\
&\mathit{inspect}(\mathit{stat},2) &&= \{L^2, N^1\}\\
&\mathit{inspect}(\mathit{empty},1) &&= \{\}\\
&\mathit{inspect}(\mathit{empty},2) &&= \{\}\\
&\mathit{inspect}(\mathit{name},1) &&= \{\mathit{str}\}
\end{aligned}$$

a. The *inspect* sets

$$\begin{aligned}
S^1 &= \mathbf{root}^1(L^1, L^2)\\
L^1 &= \mathbf{decl}^1(N^1, L^1)\\
&\mid \mathbf{stat}^1(L^1)\\
&\mid \mathbf{empty}^1()\\
L^2 &= \mathbf{decl}^2(L^2)\\
&\mid \mathbf{stat}^2(L^2, N^1)\\
&\mid \mathbf{empty}^2()\\
N^1 &= \mathbf{name}^1(\mathit{str})
\end{aligned}$$

b. The split grammar

$$\begin{aligned}
&\mathit{split}_S :: S \longrightarrow \underline{S}\\
&\mathit{split}_S\ \mathbf{root}(L) = [\mathbf{root}^1(L^1, L^2)]\\
&\quad \mathbf{where}\ [L^1, L^2] = \mathit{split}_L\ L\\
&\mathit{split}_L :: L \longrightarrow \underline{L}\\
&\mathit{split}_L\ \mathbf{decl}(N, L) = [\mathbf{decl}^1(N^1, L^1), \mathbf{decl}^2(L^2)]\\
&\quad \mathbf{where}\ [N^1] \quad = \mathit{split}_N\ N\\
&\qquad\quad ;\ [L^1, L^2] = \mathit{split}_L\ L\\
&\mathit{split}_L\ \mathbf{stat}(N, L) = [\mathbf{stat}^1(L^1), \mathbf{stat}^2(L^2, N^1)]\\
&\quad \mathbf{where}\ [N^1] \quad = \mathit{split}_N\ N\\
&\qquad\quad ;\ [L^1, L^2] = \mathit{split}_L\ L\\
&\mathit{split}_L\ \mathbf{empty}() = [\mathbf{empty}^1(), \mathbf{empty}^2()]\\
&\mathit{split}_N :: N \longrightarrow \underline{N}\\
&\mathit{split}_N\ \mathbf{name}(\mathit{str}) = [\mathbf{name}^1(\mathit{str})]
\end{aligned}$$

c. The *split* functions

Figure 4.2. Split grammar and *split* functions associated with the VARUSE grammar

4.2.2 Splitting the VARUSE grammar

As an example, we will now split the VARUSE grammar. It features a syntactic reference. First, split analysis determines the *inspect* sets. Secondly, we will define the split grammar induced by the VARUSE grammar. Next, we will present the *split* functions. Finally, we will give the visit-functions obtained by the SPLIT mapping from the visit-functions presented in the previous chapter (Figure 3.32$_{\triangleleft 84}$).

The root symbol S has one visit inducing $\underline{S} = [S^1]$. Likewise, N will induce $\underline{N} = [N^1]$. Non-terminal L has two visits, so it induces the two-tuple $\underline{L} = [L^1, L^2]$. str is a pseudo-terminal, so it will not be split.

Split analysis yields the *inspect* sets given in Figure 4.2a. For example, since the N child is visited in the first compartment of a decl node and the L child is visited for the first time in the first compartment and for the second time in the second compartment, $\mathit{inspect}(\mathit{decl},1) = \{N^1, L^1\}$ and $\mathit{inspect}(\mathit{decl},2) = \{L^2\}$. The visit-sequence for the root production contains the first as well as the second visit to L. Hence, $\mathit{inspect}(\mathit{root},1) = \{L^1, L^2\}$. Since the visit-sequence for the production name contains a syntactic reference (but no visits), $\mathit{inspect}(\mathit{name},1)$ is the singleton $\{\mathit{str}\}$.

The split non-terminal occurrences ${L_2}^1$ and ${L_2}^2$ are abbreviated to L^1 respectively L^2. There can be no confusion with ${L_1}^1$ respectively ${L_1}^2$ since the parent (L_1) can not be a member of an *inspect* set.

The split grammar, given in Figure 4.2b, is straightforwardly obtained from the *inspect* sets of Figure 4.2a.

Figure 4.2c gives the split_S, split_L and split_N functions. They form a homomorphism that map a plain abstract syntax tree to a split tree. Observe that in the split_N function the subtree that matches the pattern variable str is included (unsplit) in the split production name^1.

Figure 4.3 shows the split tree obtained by applying split_S to the standard VARUSE application. It closely resembles the augmented plan tree given in Figure 3.30$_{\triangleleft 83}$. The visit-functions for the VARUSE grammar obtained by the SPLIT mapping are given in Figure 4.4$_{\triangleright}$.

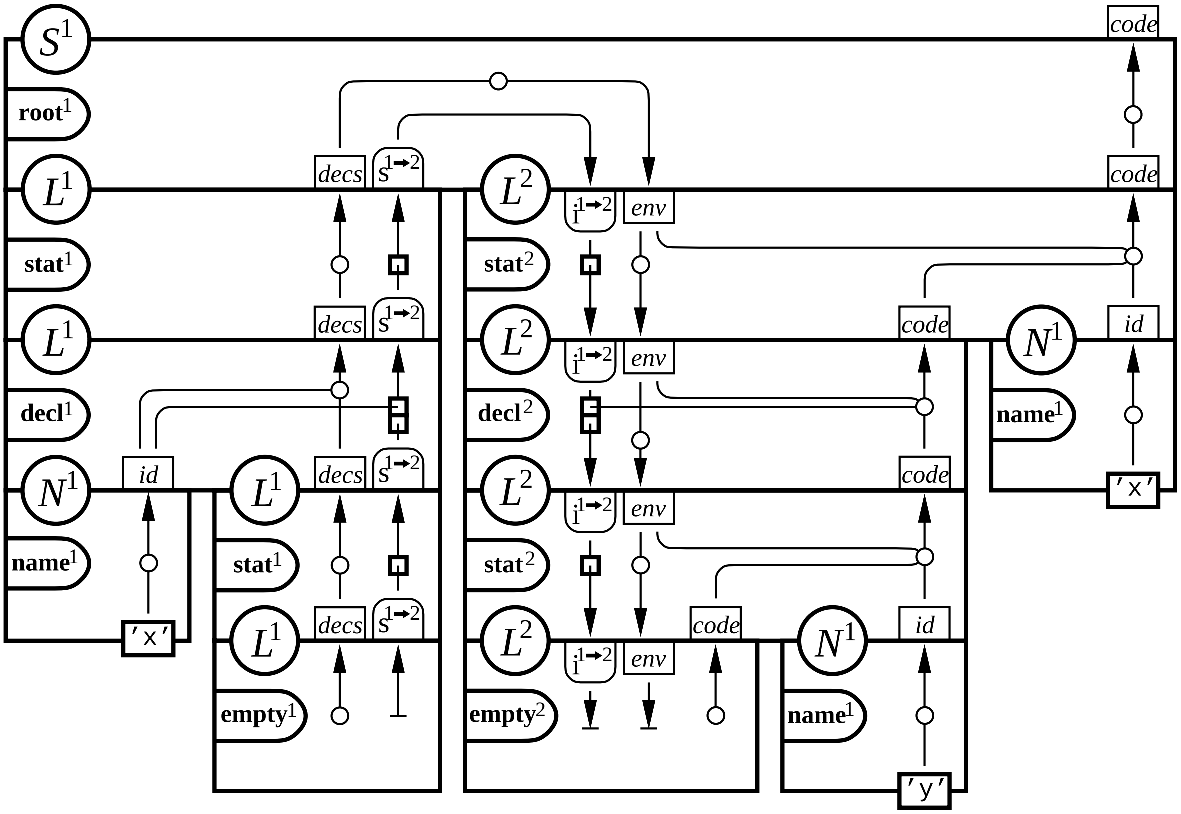


Figure 4.3. The split tree associated with the standard VARUSE application

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{visit}_{S^1} :: S^1 \longrightarrow \mathit{Code}} \\
\multicolumn{3}{l}{\mathit{visit}_{S^1}\ \mathbf{root}^1(L^1, L^2) = S.\mathit{code}} \\
\quad \mathbf{where} & (L.\mathit{decs}, L.\mathrm{s}^{1\to 2}) & = \mathit{visit}_{L^1}\ L^1 \\
 & ;\ L.\mathit{env} & = L.\mathit{decs} \\
 & ;\ L.\mathrm{i}^{1\to 2} & = L.\mathrm{s}^{1\to 2} \\
 & ;\ L.\mathit{code} & = \mathit{visit}_{L^2}\ L^2\ L.\mathrm{i}^{1\to 2}\ L.\mathit{env} \\
 & ;\ S.\mathit{code} & = L.\mathit{code}
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{visit}_{L^1} :: L^1 \longrightarrow \mathit{Env} \times L^{1\to 2}} \\
\multicolumn{3}{l}{\mathit{visit}_{L^1}\ \mathbf{decl}^1(N^1, {L_2}^1) = (L_1.\mathit{decs}, L_1.\mathrm{s}^{1\to 2})} \\
\quad \mathbf{where} & N.\mathit{id} & = \mathit{visit}_{N^1}\ N^1 \\
 & ;\ (L_2.\mathit{decs}, L_2.\mathrm{s}^{1\to 2}) & = \mathit{visit}_{L^1}\ {L_2}^1 \\
 & ;\ L_1.\mathit{decs} & = N.\mathit{id} : L_2.\mathit{decs} \\
 & ;\ L_1.\mathrm{s}^{1\to 2} & = \mathbf{decl}^{1\to 2}(N.\mathit{id}, L_2.\mathrm{s}^{1\to 2}) \\
\multicolumn{3}{l}{\mathit{visit}_{L^1}\ \mathbf{stat}^1({L_2}^1) = (L_1.\mathit{decs}, L_1.\mathrm{s}^{1\to 2})} \\
\quad \mathbf{where} & (L_2.\mathit{decs}, L_2.\mathrm{s}^{1\to 2}) & = \mathit{visit}_{L^1}\ {L_2}^1 \\
 & ;\ L_1.\mathit{decs} & = L_2.\mathit{decs} \\
 & ;\ L_1.\mathrm{s}^{1\to 2} & = \mathbf{stat}^{1\to 2}(L_2.\mathrm{s}^{1\to 2}) \\
\multicolumn{3}{l}{\mathit{visit}_{L^1}\ \mathbf{empty}^1() = (L_1.\mathit{decs}, L_1.\mathrm{s}^{1\to 2})} \\
\quad \mathbf{where} & L_1.\mathit{decs} & = [\,] \\
 & ;\ L_1.\mathrm{s}^{1\to 2} & = \mathbf{empty}^{1\to 2}()
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{visit}_{N^1} :: N^1 \longrightarrow \mathit{str}} \\
\multicolumn{2}{l}{\mathit{visit}_{N^1}\ \mathbf{name}^1(\mathit{str}) = N.\mathit{id}} \\
\quad \mathbf{where} & N.\mathit{id} = \mathit{upstring}\ \mathit{str}
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{visit}_{L^2} :: L^2 \times L^{1\to 2} \times \mathit{Env} \longrightarrow \mathit{Code}} \\
\multicolumn{3}{l}{\mathit{visit}_{L^2}\ \mathbf{decl}^2({L_2}^2)\ \mathbf{decl}^{1\to 2}(N.\mathit{id}, L_2.\mathrm{s}^{1\to 2})\ L_1.\mathit{env} = L_1.\mathit{code}} \\
\quad \mathbf{where} & L_2.\mathit{env} & = L_1.\mathit{env} \\
 & ;\ L_2.\mathrm{i}^{1\to 2} & = L_2.\mathrm{s}^{1\to 2} \\
 & ;\ L_2.\mathit{code} & = \mathit{visit}_{L^2}\ {L_2}^2\ L_2.\mathrm{i}^{1\to 2}\ L_2.\mathit{env} \\
 & ;\ L_1.\mathit{code} & = (-\mathit{lookup}\ L_1.\mathit{env}\ N.\mathit{id}) : L_2.\mathit{code} \\
\multicolumn{3}{l}{\mathit{visit}_{L^2}\ \mathbf{stat}^2({L_2}^2, N^1)\ \mathbf{stat}^{1\to 2}(L_2.\mathrm{s}^{1\to 2})\ L_1.\mathit{env} = L_1.\mathit{code}} \\
\quad \mathbf{where} & L_2.\mathit{env} & = L_1.\mathit{env} \\
 & ;\ L_2.\mathrm{i}^{1\to 2} & = L_2.\mathrm{s}^{1\to 2} \\
 & ;\ L_2.\mathit{code} & = \mathit{visit}_{L^2}\ {L_2}^2\ L_2.\mathrm{i}^{1\to 2}\ L_2.\mathit{env} \\
 & ;\ N.\mathit{id} & = \mathit{visit}_{N^1}\ N^1 \\
 & ;\ L_1.\mathit{code} & = (+\mathit{lookup}\ L_1.\mathit{env}\ N.\mathit{id}) : L_2.\mathit{code} \\
\multicolumn{3}{l}{\mathit{visit}_{L^2}\ \mathbf{empty}^2()\ \mathbf{empty}^{1\to 2}()\ L_1.\mathit{env} = L_1.\mathit{code}} \\
\quad \mathbf{where} & L_1.\mathit{code} & = [\,]
\end{array}
$$

Figure 4.4. Split visit-functions for the VARUSE grammar

Does the VARUSE grammar benefit from splitting? As the split tree given in Figure 4.3◁ clearly shows, a change to a *name* node that is a child of a *stat* node has no effect on the L^1 tree. In other words, suppose that we edit the standard application `(use x;var x;use y;)` into `(use x;var x;use x;)` correcting the obvious mistake of using the undeclared variable `y`. The standard application maps to the split tree given in Figure 4.5a, whereas the corrected version maps to the split tree given in Figure 4.5b.

Observe that both versions share the split tree for the first visit to L, which collects the declarations. As a result, incremental decoration of the corrected version will be substantially faster since

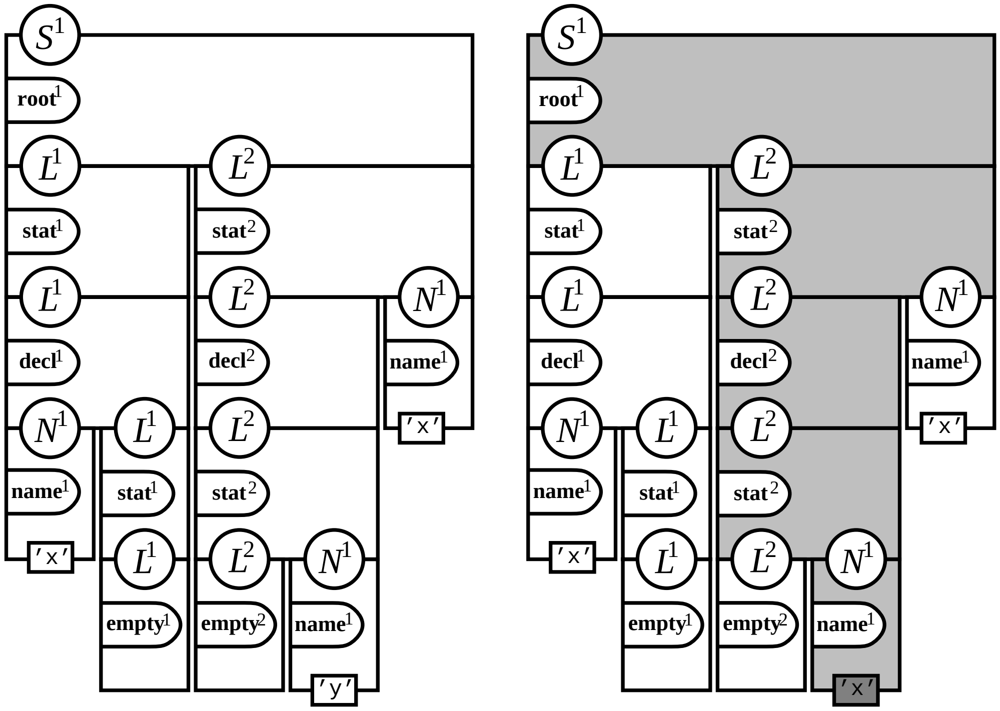
$$\mathit{visit}_{L^1}\ \mathbf{stat}^1(\mathbf{decl}^1(\mathbf{name}^1(\text{'x'}), \mathbf{stat}^1(\mathbf{empty}^1())))$$

will hit.

We conclude that we have reached the goal set in the introduction. The *split* functions preprocess the input for the visit-functions so that some unequal trees yielding the same results are mapped to the same split tree.

4.2.3 Higher-order attribute grammars

Trees assigned to an attributable attribute should also be split. Let $p.x$ be an attributable attribute of type X (thus $X = R\ p.x$). Let $p.x := E$ be the equation



a. Before **b.** After

The using occurrence of y is changed to x (dark grey). The light grey nodes belong to the changed tree.

Figure 4.5. An edit operation on the standard VARUSE application

associated with the instruction $eval(p.x)$ in the higher-order visit-sequence. The type of E is X. In other words, $p.x$ is assigned an unsplit tree. If it were split, we could use the split visit-functions to decorate the higher-order tree, which is potentially more efficient.

The unsplit $p.x$ must remain available since it could be used in some equation.

Of course, the solution is to apply $split_X$ to E. Attributable attribute $p.x$ becomes a local attribute and $\mathbf{v}X$ fresh attributable attributes $p.x^1 \ldots p.x^{\mathbf{v}X}$ are introduced. Instruction $[p.x^1, \ldots, p.x^{\mathbf{v}X}] = split_X\ p.x$ is inserted immediately after instruction $p.x := E$. Each $visit(p.x, v)$ instruction $(1 \leq v \leq \mathbf{v}X)$ must be mapped onto $visit_{X^v}\ p.x^v$.

An attributable attribute $p.x$ might also be a border-crosser. In the unsplit variant, this results in $p.x$ being transported, as a whole, to a successive compartment of p. However, when generating split visit-functions, binding analysis should be modified somewhat: *not* $\{p.x \mid visit(p.x, u) \text{ occurs in } vss(p, w)\}$ should be added to $use(p, w)$ but $\{p.x^u \mid visit(p.x, u) \text{ occurs in } vss(p, w)\}$. As a result, only those split components of $p.x$ will be bound by wrapper $p^{v \to w}$ that are actually visited in compartment w of p.

4.2.4 Flow trees and data trees

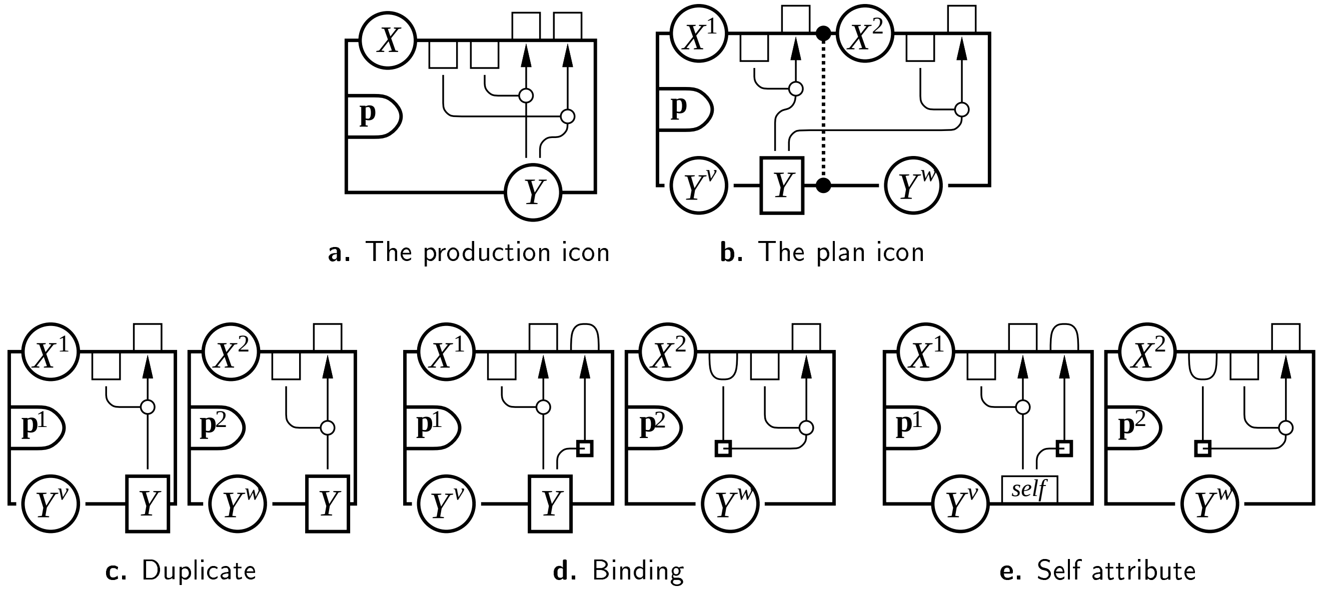
In an attribute grammar system, we can distinct two different kind of terms, both of which are described by a context-free grammar defined by the grammar writer. The first kind of term is known as *flow* tree. Informally formulated, a flow tree is a tree that is passed as first argument to a visit-function. The abstract syntax tree which is decorated by the visit-functions is a flow tree. The other kind of term is known as *data* tree. Data trees are data structures defined by the grammar writer. Such structures are passed around in attributes and they are destructed by semantic functions.

This distinction is important, because we could use more efficient representations for one or the other. For example, flow trees could be represented as ordinary terms, but as we have seen above, splitting is a transformation that optimizes their role as data flow driver. Other optimizations will be discussed in the rest of this chapter.

Attributable attribute:
data ⟶ flow;
syntactic reference:
flow ⟶ data.

Unfortunately, there are a two places were the distinction between flow trees and data trees is blurred. Those places are the "higher-order" constructs in our attribute grammar formalism: attributable attributes and syntactic references. In the former case a data tree is assigned to an attributable attribute that is then visited. In the latter case a part of the abstract syntax tree, a flow tree, is referred to as a data tree.

The problem with data trees being assigned to attributable attributes was solved for splitting by dynamically converting the data tree to a flow tree by means of the $split$ function. The problem with flow trees being syntactically referenced is solved for splitting by statically including a syntactically referenced tree as terminal in a split production.



a. The production icon

b. The plan icon

c. Duplicate

d. Binding

e. Self attribute

Figure 4.6. Syntactic references in a split production

The next section introduces encapsulators. Encapsulators are like tree constructors, except that they construct split trees instead of plain terms. In other words, by statically replacing constructors by encapsulators, the dynamic conversion from data trees to flow trees is eliminated.

Syntactic references

A syntactic reference to an unsplit abstract syntax tree T is resolved by including T in the split production. We discuss two other possible solutions.

Production p in Figure 4.6a has only one child, namely Y. Suppose that Y is a "real" non-terminal, that is visited several times. For brevity, the attributes of Y have been left out. Note, that Y is referenced twice by an equation.

Suppose further the plan icon associated with Y is given in Figure 4.6b. Non-terminal X has two visits, so that p has two compartments. There appears to be a border crossing , but there is none: since Y is a syntactical element and not an attribute, it is available in both compartments. For completeness, two visits to Y, v and w, have been sketched. Again, we left out Ys attributes.

For identical reasons, the *box* production of the BOX grammar (Fig 3.33$_{\triangleleft 86}$) also appears to have a border-crossing.

If we split p, we obtain the production icons in Figure 4.6c. The subtree rooted Y is inspected twice: once in the first compartment of p and once in the second compartment of p. As a result, Y is included in both split productions. Since terms are shared in our grammar system, such duplications are not inefficient. Of course, a change in Y changes both X^1 and X^2 so that splitting seems pointless. However, the fact that Y is used in both compartments is an intrinsic property of production p.

A different way of dealing with several syntactic references to the same tree is presented in Figure 4.6d$_{\triangleleft}$. Subtree Y is only included once, and it is passed via a binding to the second compartment of p. A change to Y might keep X^2 unchanged, however binding $X^{1\rightarrow 2}$ has changed so that $visit_X^2$ misses nevertheless. If parcel $X^{1\rightarrow 2}$ were empty, we are worse of with this solution since it no longer is empty.

The third solution, presented in Figure 4.6e$_{\triangleleft}$, is based on a $self$ attribute. Syntactic references can be avoided altogether by introducing an attribute $self$ with every non-terminal that synthesized the underlying abstract syntax tree. Decoration of the *flow* tree Y yields as attribute a *data* tree isomorphic with Y in attribute $Y.self$. In Figure 4.6e$_{\triangleleft}$, we assumed that $Y.self \in Y\,\mathbf{S}\,v$. Observe that $self$ attributes eliminate the need of incorporating tree fragments in split productions. However the disadvantages of the previous approach still holds: the second visit still misses since the binding has changed and the parcel $X^{1\rightarrow 2}$ is no longer empty.

4.2.5 Discussion

As a result of splitting, an abstract syntax tree is represented by a tuple of split trees. Like plain abstract syntax trees in the unsplit approach, split trees are arguments of cached functions. This implies that split trees should be shared for fast comparison.

Since we assume that the start symbol is visited only once, the split representation of an abstract syntax tree rooted by the start symbol is a single term. Each split tree is tailored for a specific visit. It only includes those parts of the underlying abstract syntax tree that are actually inspected during that visit. Therefore, the split abstract syntax tree resembles the (augmented) plan tree closely.

As a consequence, the plan tree may be superimposed on the split tree. The left to right depth first order on the attribute instances (the grey line in Figure 3.30$_{\triangleleft 83}$) in the plan tree may also be superimposed on the attribute instances in the split tree. Consequently, the split grammar can be evaluated in one left to right depth first pass. Thus, splitting may be viewed as a grammar transformation that maps arbitrary partitionable grammars to one pass left to right depth first grammars.

However, it is not completely fair to see the transformation as a *grammar* transformation. Splitting depends heavily on the algorithm that computes visit-sequences, which offers a lot of freedom. It might happen for example, that if one would transform a grammar G into a grammar G' using splitting, that a split non-terminal of G' is assigned two visits by the algorithm that computes visit-sequences. In other words, the visit-sequences *computed* for G', might not coincide with the visit-sequences obtained by *splitting* the visit-sequences computed for G (which is in essence the approach advocated in this section).

One must realize that splitting is only feasible if there are no implicit intra-visit-dependencies. There is no way of storing the value of a border-crosser in a tree node in order to access it in a later visit. Due to splitting, the later visit refers to another node. As a consequence, splitting is not applicable in standard attribute grammar setting where attribute instances are stored in tree nodes, unless the visit-sequences happen to be free of intra-visit-dependencies.

In a split tree, we can not really distinguish between two different *children* of a tree node and two different *visits* to the same child. The only distinction is that successive visits to the same child are linked with bindings. However, since bindings may be empty, these links may be absent too. As a result, a split grammar is more "general" and may therefore be well suited for further analysis. For example, parallelism detection might benefit from such a generalization.

Another advantage of the split grammar is that the dpr graphs associated with each split production are smaller. The $dpr(p)$ graph is unraveled into subgraphs $dpr(p^v)$. The benefit of having simpler dpr graphs is that they are easier to analyze. In the next sections we will encounter examples exploiting this fact.

4.3 Encapsulators

Encapsulators are functions associated with constructors, that hide the application of those constructors. As such, encapsulators are not interesting. However, by hiding the actual tree construction, the encapsulators may construct a tree in a different representation.

As a first example, encapsulators will be used to construct a *split tree* instead of a plain abstract syntax tree. In effect, the encapsulators hide the immediate application of $split$.

Furthermore, since encapsulators hide tree construction, they may as well construct a differently shaped tree. The only restriction is that the semantics in the sense of the relation between the inherited and synthesized attributes of the root of the tree do not change.

As a second usage of encapsulators we will show that we may statically decide not to include certain tree nodes. As a result, the tree is not only smaller, which saves space and evaluation time, it also increases the chance on cache hits of the visit-functions, since more trees are likely to be the same.

A third application of encapsulators is *unification*. Static analysis may infer that different tree nodes have the same semantics. In such a case, every node of one kind may be replaced by a node of another kind. The advantage of such an optimization is that more trees are likely to be the same which increases the chance on cache hits.

Fourthly, we discuss folding and normalization. These optimizations are also realized by encapsulators, however they bring dynamic costs.

4.3.1 Definitions

Let (G', f') be a refinement of (G, f), and let ϕ be the corresponding abstraction. Let $\mathbf{p} :: X_1 \times \ldots \times X_s \longrightarrow X_0$ be a constructor corresponding with production p in G. For clarity, the signature of constructor $\mathbf{p}$ is denoted in a functional manner. Finally, let $T_1 \ldots T_s$ be terms of type $X_1 \ldots X_s$. The function p' is an *encapsulator* of $\mathbf{p}$ if

$$p' \; (\phi \; T_1) \; \ldots \; (\phi \; T_s) = \phi \; \mathbf{p}(T_1, \ldots, T_s) \quad .$$

The type X' is an *implementation* of X. It is defined as

$$X' = \{\phi \; T \mid T \text{ has root } X\} \quad .$$

Thus, the signature of encapsulator p' is

$$p' :: X'_1 \times \ldots \times X'_s \longrightarrow X'_0 \quad .$$

In other words, the encapsulators p' and types X' *behave* as the productions p and non-terminals X of G. Nevertheless, they construct terms described by grammar G'. Abstraction ϕ guarantees that a tree T' constructed by encapsulators is a refinement of tree T constructed by the corresponding productions, in the sense that $f' \; T' = f \; T$. Observe that by using encapsulators to construct T' there is no longer a need to dynamically convert T to T' with ϕ; ϕ is encapsulated.

4.3.2 Encapsulating *split*

The split grammar and split visit-functions are a refinement of the underlying context-free grammar and the computed visit-sequences. The corresponding abstraction is the *split* function. We will now define encapsulators that hide the construction of split trees.

As a first attempt, we take $\underline{X}$ as an implementation for X. A constructor $\mathbf{p} :: X_1 \times \ldots \times X_s \longrightarrow X_0$ induces an encapsulator $\underline{p} :: \underline{X}_1 \times \ldots \times \underline{X}_s \longrightarrow \underline{X}_0$. The definition of $\underline{p}$ has the following form where each element $p\mathbf{o}i^w$ of $inspect(p, v)$ maps to a parameter X_i^w for 'prms v'.

$$\underline{p} \; [X_1^1 \ldots X_1^{\mathbf{v}X_1}] \; \ldots \; [X_s^1 \ldots X_s^{\mathbf{v}X_s}] = [\mathbf{p}^1(\text{'prms } 1\text{'}), \ldots, \mathbf{p}^{\mathbf{v}X_0}(\text{'prms } \mathbf{v}X_0\text{'})]$$

The tuple for the parent is assembled from the tuples of its children.

Paraphrased, the above definition says that the $\mathbf{v}X_0$-tuple representing the tree $\mathbf{p}(X_1, \ldots, X_s)$, is obtained by applying split productions $\mathbf{p}^1 \ldots \mathbf{p}^{\mathbf{v}X_0}$ to appropriate split trees X_i^w which are components of the tuples $[X_1^1 \ldots X_1^{\mathbf{v}X_1}] \ldots [X_s^1 \ldots X_s^{\mathbf{v}X_s}]$ that represent the respective subtrees $X_1, \ldots, X_s$.

However, the above definition exhibits a flaw when syntactic references occur in p. In the definition of the split grammar given in the previous section, a syntactic reference to X_i recorded in $inspect(p, v)$ maps to a parameter for the split production $\mathbf{p}^v$. The subtree corresponding with X_i could be obtained by applying $split_{X_i}^{-1}$ to the tuple $[X_i^1 \ldots X_i^{\mathbf{v}X_i}]$. However, there are two reasons for not doing that.

Firstly, one of the reasons for introducing encapsulators, is that we wish to eliminate dynamic conversions, in this case from plain trees to split trees. If we were to use $split^{-1}$, we would still be faced with a dynamic conversion, albeit the other way round.

$$\underline{root} :: \underline{L} \longrightarrow \underline{S}$$
$$\underline{root}\ [\ L^1\ ,\ L^2\ ] = [\ \mathbf{root}^1(L^1, L^2)\ ]$$
$$\underline{decl} :: \underline{N} \times \underline{L} \longrightarrow \underline{L}$$
$$\underline{decl}\ [\ N^1\ ]\ [\ L^1\ ,\ L^2\ ] = [\ \mathbf{decl}^1(N^1, L^1)\ ,\ \mathbf{decl}^2(L^2)\ ]$$
$$\underline{stat} :: \underline{N} \times \underline{L} \longrightarrow \underline{L}$$
$$\underline{stat}\ [\ N^1\ ]\ [\ L^1\ ,\ L^2\ ] = [\ \mathbf{stat}^1(L^1)\ ,\ \mathbf{stat}^2(L^2, N^1)\ ]$$
$$\underline{empty} :: \underline{L}$$
$$\underline{empty} = [\ \mathbf{empty}^1()\ ,\ \mathbf{empty}^2()\ ]$$
$$\underline{name} :: str \longrightarrow \underline{N}$$
$$\underline{name}\ str = [\ \mathbf{name}^1(str)\ ]$$

a. The shells

$$\underline{\underline{root}} :: \underline{\underline{L}} \longrightarrow \underline{\underline{S}}$$
$$\underline{\underline{root}}\ (L, \underline{L}) = (\mathbf{root}(L), \underline{root}\ \underline{L})$$
$$\underline{\underline{decl}} :: \underline{\underline{N}} \times \underline{\underline{L}} \longrightarrow \underline{\underline{L}}$$
$$\underline{\underline{decl}}\ (N, \underline{N})\ (L, \underline{L}) = (\mathbf{decl}(N, L), \underline{decl}\ \underline{N}\ \underline{L})$$
$$\underline{\underline{stat}} :: \underline{\underline{N}} \times \underline{\underline{L}} \longrightarrow \underline{\underline{L}}$$
$$\underline{\underline{stat}}\ (N, \underline{N})\ (L, \underline{L}) = (\mathbf{stat}(N, L), \underline{stat}\ \underline{N}\ \underline{L})$$
$$\underline{\underline{empty}} :: \underline{\underline{L}}$$
$$\underline{\underline{empty}} = (\mathbf{empty}(), \underline{empty})$$
$$\underline{\underline{name}} :: str \longrightarrow \underline{\underline{N}}$$
$$\underline{\underline{name}}\ str = (\mathbf{name}(str), \underline{name}\ str)$$

b. The encapsulators

Figure 4.7. The encapsulators

Secondly, although the *split* function is invertible, other optimizations as elimination, unification, normalization and folding which are to be discussed later, massage the split tree into more efficient forms which are no longer invertible; the *split* function is injective, but it is not surjective.

The solution we have chosen is to pass the syntactically referenced trees $X_i \ldots X_j$ as additional parameters to the function $\underline{p}$. Each element $p\mathbf{o}i^w$ respectively $p\mathbf{o}i$ of $inspect(p, v)$ maps to a parameter X_i^w respectively X_i for 'parms v'.

$$\underline{p}\ \underline{X}_1\ \ldots\ \underline{X}_s\ X_i\ \ldots\ X_j = [\mathbf{p}^1(\text{'parms 1'}), \ldots, \mathbf{p}^{\mathbf{v}X_0}(\text{'parms }\mathbf{v}X_0\text{'})]$$

A function like $\underline{p}$ is not an *encapsulator* since its signature no longer matches the signature of p. We will refer to these functions as *shells*. As an example, consider the shells in Figure 4.7a. They hide the construction of a split tree for the VARUSE grammar. The shell $\underline{name}$ has a parameter str for passing the syntactic reference.

Shells are not encapsulators according to our definition, since their signatures specify, in addition to the implementations of the right hand side symbols, the syntactic references. To obtain encapsulators, we will construct a plain tree in synchronization with the split tree, so that fragments of the plain tree are available for referencing.

The proposed implementation for a non-terminal X becomes $\underline{\underline{X}}$ defined as the two tuple $(X, \underline{X})$. In essence, the encapsulators construct a flow tree as well as a data tree. A constructor $\mathbf{p} :: X_1 \times \ldots \times X_s \longrightarrow X_0$ induces an encapsulator $\underline{p} :: \underline{\underline{X}}_1 \times \ldots \times \underline{\underline{X}}_s \longrightarrow \underline{\underline{X}}_0$; it has the following definition.

Encapsulator in general: the implementation of the parent is assembled from the implementation of its children.

$$\underline{p}\ (X_1, \underline{X}_1)\ \ldots\ (X_s, \underline{X}_s) \quad = \quad (\ \mathbf{p}(X_1, \ldots, X_s)\ ,\ \underline{p}\ \underline{X}_1\ \ldots\ \underline{X}_s\ X_i\ \ldots\ X_j\ )$$

The implementation of $\underline{\underline{X}}$ adheres to the following type invariant

$$\underline{\underline{X}} = (X, \underline{X}) \quad \equiv \quad split_X\ X = \underline{X} \quad .$$

Observe that the $split$ functions become obsolete when using encapsulators. The implementation of a pseudo-terminal is that pseudo-terminal itself.

Figure 4.7b$_\triangleleft$ gives the encapsulators for the VARUSE grammar. Observe that the $\underline{\underline{name}}$ encapsulator passes the syntactically referenced element str to the shell $\underline{name}$.

Every constructor in any equation in the attribute grammar should be replaced by an encapsulator. Then, every term is implemented as a two tuple consisting of a data tree and a flow tree. So, for destructing, the first component can be used, and for assignment to an attributable attribute, the second component can be used.

If syntactic references to non-terminals were forbidden—which does not diminish expressivity since syntactic references can be rewritten with $self$ attributes—encapsulators could be greatly simplified. The shells alone would suffice.

. . .

We assume $\mathbf{v}S = 1$.

Given a grammar G with start symbol S and visit-function $visit^1_S$. G induces a split grammar G' and split visit-function $visit_{S^1}$. With implementation $(X, \underline{X})$ for non-terminal X and encapsulator $\underline{\underline{p}}$ for production p as defined above, the decorator (G', f') is a refinement of $(G, visit^1_S)$, where $f'\ (S, [S^1]) = visit_{S^1}\ S^1$. The corresponding abstraction ϕ is defined as $\phi\ S = (S, split_S\ S)$.

4.3.3 Elimination and unification

The refinement defined above is based on splitting. Split trees are a better implementation than plain terms because they increase the chance on visit-function cache hits. We will now discuss two optimizations, *elimination* and *unification*, that are based on the *same implementation* for trees, while employing *different encapsulators*. In effect, the encapsulators restructure the flow tree to obtain better (incremental) evaluation performance while preserving the semantics.

In short, elimination removes tree nodes that do not contribute in an essential way to the data flow. Unification maps tree nodes of one kind to tree nodes of another kind if their dpr graphs coincide.

Elimination and unification are tree-restructuring optimizations effected at *generation* time. Static analysis of dpr graphs determines which tree nodes will be eliminated or unified.

Analyzing dpr graphs

The performance of any program can be enhanced by detailed analysis of data flow. In its generality data flow analysis is beyond the scope of this thesis. We limit ourselves to data flow analysis in which the semantic functions remain anonymous; we do not investigate the *semantics* of the arcs of the dpr graphs.

We make one exception though, namely for identity functions. The identity function id occurs frequently in attribute grammar specification, most notably in

$$\begin{array}{l} L^{1\to2}{=}\mathbf{decl}^{1\to2}(\mathit{str},L^{1\to2}) \\ \quad \mid \ \mathbf{empty}^{1\to2}() \end{array}$$

a. Declaration

$$\begin{array}{l} \mathit{visit}_{L^1} :: L^1 \longrightarrow \mathit{Env} \times L^{1\to2} \\ \mathit{visit}_{L^1}\ \mathbf{stat}^1({L_2}^1) = (L_1.\mathit{decs}, L_1.\mathrm{s}^{1\to2}) \\ \quad \mathbf{where}\ (L_2.\mathit{decs}{,}L_2.\mathrm{s}^{1\to2}){=}\mathit{visit}_{L^1}\ {L_2}^1 \\ \qquad\quad ;\ L_1.\mathit{decs} \qquad\qquad {=}L_2.\mathit{decs} \\ \qquad\quad ;\ L_1.\mathrm{s}^{1\to2} \qquad\qquad {=}L_2.\mathrm{s}^{1\to2} \end{array}$$

b. Construction

$$\begin{array}{l} \mathit{visit}_{L^2} :: L^2 \times L^{1\to2} \times \mathit{Env} \longrightarrow \mathit{Code} \\ \mathit{visit}_{L^2}\ \mathbf{stat}^2({L_2}^2{,}N^1)\ L_2.\mathrm{s}^{1\to2}\ L_1.\mathit{env}{=}L_1.\mathit{code} \\ \quad \mathbf{where}\ L_2.\mathit{env}\ = L_1.\mathit{env} \\ \qquad\qquad ;\ L_2.\mathrm{i}^{1\to2} = L_2.\mathrm{s}^{1\to2} \\ \qquad\qquad ;\ L_2.\mathit{code} = \mathit{visit}_{L^2}\ {L_2}^2\ L_2.\mathrm{i}^{1\to2}\ L_2.\mathit{env} \\ \qquad\qquad ;\ N.\mathit{id} \quad\ = \mathit{visit}_{N^1}\ N^1 \\ \qquad\qquad ;\ L_1.\mathit{code} = (+\mathit{lookup}\ L_1.\mathit{env}\ N.\mathit{id}) \\ \qquad\qquad\qquad\qquad :\ L_2.\mathit{code} \end{array}$$

c. Destruction

Figure 4.8. Eliminating the copy-wrapper $\mathbf{stat}^{1\to2}$ of the VARUSE grammar

so-called *copy-rules*. Copy-rules are equations of the form $\alpha := id\ \beta$. Copy-rules make grammar specifications long winded and evaluators inefficient. Not surprisingly, the elimination of copy rules is a major topic in attribute grammar research, ranging from syntactic abbreviations for remote access to attribute instances in grammar specifications [Kas91, RT88] to bypassing copy-rules in evaluators [Hoo86, RT89].

Wrappers are sometimes copy-rules in disguise. For example, the VARUSE grammar, contains the wrapper $L^{1\to2} = \mathbf{stat}^{1\to2}(L^{1\to2})$ which is a copy-wrapper. The first step in data flow analysis removes these wrappers. Such a transformation has three aspects: the *definition* of the wrapper must be removed from the parcel definition, the wrapper must be removed from the equation were it is used to *construct* a binding, and thirdly, the wrapper must be removed from the equation that *destructs* that binding.

As an example, consider how the parcel definition for $L^{1\to2}$ and the visit-functions visit_{L^1} and visit_{L^2} change when the copy-wrapper $\mathbf{stat}^{1\to2}$ is removed. Figure 4.8a shows the parcel definition after removal of $\mathbf{stat}^{1\to2}$. Figure 4.8b shows the visit-function body that constructed a binding with $\mathbf{stat}^{1\to2}$ and Figure 4.8c shows the visit-function for the second traversal that destructed the binding (using pattern matching on $\mathbf{stat}^{1\to2}$). With the removal of $\mathbf{stat}^{1\to2}$, the binding computed during decoration of the standard VARUSE application is reduced in size: see Figure 4.11b$_{\triangleright 110}$.

In the rest of this section, we use a liberal interpretation of dpr graphs. A dpr graph is to be understood as the *augmented* dependency graph—thus including the equations for the binding occurrences—associated with a *split* production—thus only considering the occurrences of a single compartment. Copy-wrappers are assumed to have been eliminated.

Elimination

Tree nodes, whose associated dpr graph consists of copy-rules only, do not contribute in an essential way to the dtr graph. Therefore, such node may be *eliminated*.

According to Definition 16, a (split) production $X^v = \mathbf{p}^v(X^v)$ is redundant if its associated $dpr(p^v)$ graph consists of non-crossing copy-rules only. Such production are characterized by the icon in Figure 4.9. The requirement that the copy-rules are non-crossing is important. If the arrows did cross, the tree node would contribute to the dtr graph: it would swap the values of some attributes.

Definition 16 *Redundant production.*
A split production p^v is redundant if

- its only child has the same type as the parent: $inspect(p, v) = \{p\mathbf{o}1^v\}$ and $p{\cdot}0 = p{\cdot}1$;
- the equations are non-crossing copy-rules only: $\forall_{p\mathbf{o}1.i \in p\mathbf{o}1\mathbf{I}^v}(p\mathbf{o}1.i = id\ p\mathbf{o}0.i) \in E(p)$ and $\forall_{p\mathbf{o}0.s \in p\mathbf{o}0\mathbf{S}^v}(p\mathbf{o}0.s = id\ p\mathbf{o}1.s) \in E(p)$.

□

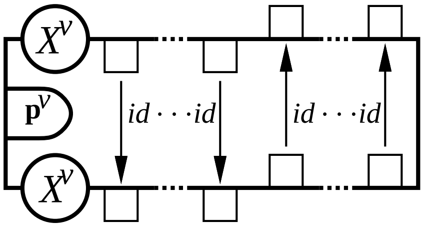
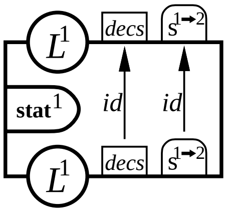


Figure 4.9. Non crossing copy-rules only

Nodes that are instances of redundant productions may be eliminated from any tree T without altering the semantics of T. Elimination of p^v is easily implemented in the encapsulator $\underline{p}$. The shell $\underline{p}$ constructs a tuple with $\mathbf{p}^v(X^v)$ as vth component. This component should be transformed into X^v alone, leaving out the application of $\mathbf{p}^v$. Since shell $\underline{p}$ is the *only* function applying $\mathbf{p}^v$, this transformation makes constructor $\mathbf{p}^v$ superfluous. As a result, $\mathbf{p}^v$ can be removed from the split grammar and the visit-function alternative $visit_{p\cdot0^v}\ \mathbf{p}^v(\ldots)$ may also be removed.



Production $stat^1$ in the (split) VARUSE grammar is an example of a redundant production (once the copy-wrapper $\mathbf{stat}^{1\to2}$ is removed). To eliminate $stat^1$, the shell $\underline{stat}$ is transformed from

$$\underline{stat}\ [N^1]\ [L^1, L^2] = [\mathbf{stat}^1(L^1), \mathbf{stat}^2(L^2, N^1)]$$

into

$$\underline{stat}\ [N^1]\ [L^1, L^2] = [\qquad L^1\ \ , \mathbf{stat}^2(L^2, N^1)]\quad .$$

Figure 4.10 shows the data structures and visit-functions after elimination of $stat^1$. First of all, note that $stat^1$ is removed from the term definition (upper

$$\begin{array}{lcl} S^1 &=& \mathbf{root}^1(L^1,L^2)\\ L^1 &=& \mathbf{decl}^1(N^1,L^1)\\ &|& \mathbf{empty}^1()\\ L^2 &=& \mathbf{decl}^2(L^2)\\ &|& \mathbf{stat}^2(L^2,N^1)\\ &|& \mathbf{empty}^2()\\ N^1 &=& \mathbf{name}^1(str) \end{array}$$

$$\begin{array}{l} visit_{S^1} :: S^1 \longrightarrow Code\\ visit_{S^1}\ \mathbf{root}^1(L^1,L^2) = S.code\\ \quad \begin{array}{rll} \mathbf{where} & (L.decs, L.\mathrm{s}^{1\to2}) & = visit_{L^1}\ L^1\\ ; & L.env & = L.decs\\ ; & L.\mathrm{i}^{1\to2} & = L.\mathrm{s}^{1\to2}\\ ; & L.code & = visit_{L^2}\ L^2\ L.\mathrm{i}^{1\to2}\ L.env\\ ; & S.code & = L.code \end{array} \end{array}$$

$$\begin{array}{l} visit_{L^1} :: L^1 \longrightarrow Env \times L^{1\to2}\\ visit_{L^1}\ \mathbf{decl}^1(N^1,{L_2}^1) = (L_1.decs, L_1.\mathrm{s}^{1\to2})\\ \quad \begin{array}{rll} \mathbf{where} & N.id & = visit_{N^1}\ N^1\\ ; & (L_2.decs, L_2.\mathrm{s}^{1\to2}) & = visit_{L^1}\ {L_2}^1\\ ; & L_1.decs & = N.id : L_2.decs\\ ; & L_1.\mathrm{s}^{1\to2} & = \mathbf{decl}^{1\to2}(N.id,L_2.\mathrm{s}^{1\to2}) \end{array}\\ visit_{L^1}\ \mathbf{empty}^1() = (L_1.decs, L_1.\mathrm{s}^{1\to2})\\ \quad \begin{array}{rll} \mathbf{where} & L_1.decs & = [\,]\\ ; & L_1.\mathrm{s}^{1\to2} & = \mathbf{empty}^{1\to2}() \end{array} \end{array}$$

$$\begin{array}{l} \underline{root}\ [L^1,L^2] = [\mathbf{root}^1(L^1,L^2)]\\ \underline{decl}\ [N^1]\ [L^1,L^2] = [\mathbf{decl}^1(N^1,L^1), \mathbf{decl}^2(L^2)]\\ \underline{stat}\ [N^1]\ [L^1,L^2] = [L^1, \mathbf{stat}^2(L^2,N^1)]\\ \underline{empty} = [\mathbf{empty}^1(), \mathbf{empty}^2()]\\ \underline{name}\ str = [\mathbf{name}^1(str)] \end{array}$$

$$\begin{array}{lcl} L^{1\to2} &=& \mathbf{decl}^{1\to2}(str, L^{1\to2})\\ &|& \mathbf{empty}^{1\to2}() \end{array}$$

$$\begin{array}{l} visit_{N^1} :: N^1 \longrightarrow str\\ visit_{N^1}\ \mathbf{name}^1(str) = N.id\\ \quad \mathbf{where}\ N.id = upstring\ str \end{array}$$

$$\begin{array}{l} visit_{L^2} :: L^2 \times L^{1\to2} \times Env \longrightarrow Code\\ visit_{L^2}\ \mathbf{decl}^2({L_2}^2)\ \mathbf{decl}^{1\to2}(N.id,L_2.\mathrm{s}^{1\to2})\ L_1.env = L_1.code\\ \quad \begin{array}{rll} \mathbf{where} & L_2.env & = L_1.env\\ ; & L_2.\mathrm{i}^{1\to2} & = L_2.\mathrm{s}^{1\to2}\\ ; & L_2.code & = visit_{L^2}\ {L_2}^2\ L_2.\mathrm{i}^{1\to2}\ L_2.env\\ ; & L_1.code & = (-lookup\ L_1.env\ N.id) : L_2.code \end{array}\\ visit_{L^2}\ \mathbf{stat}^2({L_2}^2,N^1)\ L_2.\mathrm{s}^{1\to2}\ L_1.env = L_1.code\\ \quad \begin{array}{rll} \mathbf{where} & L_2.env & = L_1.env\\ ; & L_2.\mathrm{i}^{1\to2} & = L_2.\mathrm{s}^{1\to2}\\ ; & L_2.code & = visit_{L^2}\ {L_2}^2\ L_2.\mathrm{i}^{1\to2}\ L_2.env\\ ; & N.id & = visit_{N^1}\ N^1\\ ; & L_1.code & = (+lookup\ L_1.env\ N.id) : L_2.code \end{array}\\ visit_{L^2}\ \mathbf{empty}^2()\ \mathbf{empty}^{1\to2}()\ L_1.env = L_1.code\\ \quad \mathbf{where}\ L_1.code\ = [\,] \end{array}$$

Figure 4.10. Split visit-functions for the VARUSE grammar with $stat^1$ eliminated

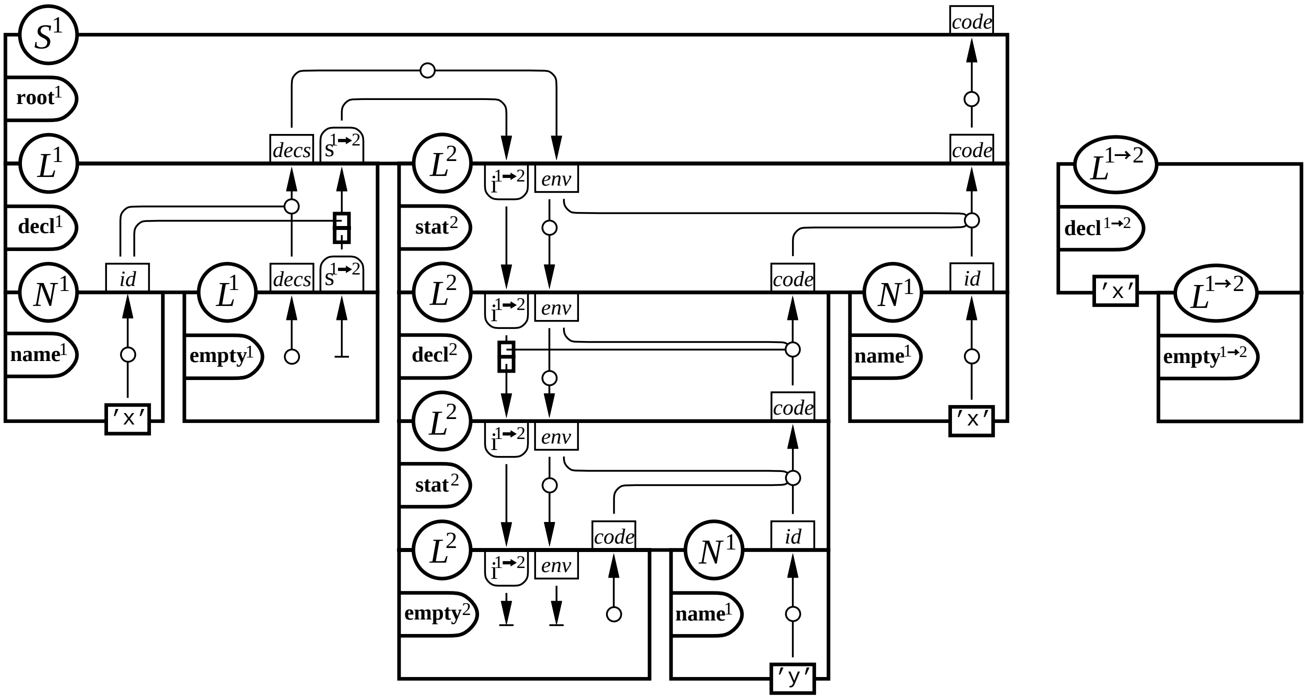
left). The shell $\underline{stat}$ no longer applies $\mathbf{stat}^1$ as just explained (upper right); the encapsulators themselves do not change. Visit-function $visit_{L^1}$ (lower left) is changed: it no longer has an alternative for $\mathbf{stat}^1$ nodes.

Figure 4.11a$_{\triangleright}$ gives the split tree constructed by the shells in Figure 4.10. The main difference with Figure 4.3$_{\triangleleft 97}$ is, of course, that $\mathbf{stat}^1$ icons have been short circuited. Note also that the $\mathbf{stat}^2$ nodes use copy rules for the binding attributes instead of the copy-wrapper $\mathbf{stat}^{1\to2}$. As a result, the binding computed during decoration of the standard application is reduced in size; compare Figure 3.31$_{\triangleleft 83}$ with Figure 4.11b$_{\triangleright}$.

Incremental decoration of VARUSE grammar trees also gains from elimination of $stat^1$. When a $stat$ node is inserted or deleted in an abstract syntax tree, the encapsulator does not insert or delete a $\mathbf{stat}^1$ constructor in the split tree. As a result, the L^1 tree does not change, resulting in a cache hit for $visit_{L^1}$.

For example, suppose that an edit action transforms the standard tree (leaving out the $root$ node for brevity)

$$(L, [L^1, L^2]) = \underline{\underline{stat}}\ (\underline{\underline{name}}\ \texttt{'x'})\ \Big(\underline{\underline{decl}}\ (\underline{\underline{name}}\ \texttt{'x'})\ (\underline{\underline{stat}}\ (\underline{\underline{name}}\ \texttt{'y'})\ \underline{\underline{empty}})\Big)$$



a. $stat^1$ eliminated **b.** $\mathbf{stat}^{1\to2}$ eliminated

Figure 4.11. The standard VARUSE application with elimination in effect

into

$$(L, [L^1, L^2]) = \underline{\underline{stat}}\ (\underline{\underline{name}}\ \text{'x'})\ \left(\underline{\underline{decl}}\ (\underline{\underline{name}}\ \text{'x'})\ \underline{\underline{empty}}\right)$$

simply deleting the mistakenly used variable 'y'. In both cases, the split tree for the first visit to L are equal

$$L^1 = L'^1 = \mathbf{decl}^1(\mathbf{name}^1(\text{'x'}), \mathbf{empty}^1()) \quad .$$

Unification

Tree nodes that are isomorphic, may be unified. According to Definition 17 two (split) productions $X^v = \mathbf{p}^v(Y_1, \ldots, Y_s)$ and $X^v = \mathbf{q}^v(Z_1, \ldots, Z_s)$ are isomorphic if the children $Y_1 \ldots Y_s$ form a permutation of the children $Z_1 \ldots Z_s$ and the equation on the associated attribute occurrences are subject to the same permutation (with similar requirements for local, attributable and generated attribute occurrences).

Factorization of common equations for different productions, an abbreviation mechanism in some grammar specification languages, promotes unification.

Definition 17 *Isomorphic productions.*
Two split productions p^v and q^v are isomorphic if there exist bijections

- π from $[0..\mathbf{s}p^v]$ to $[0..\mathbf{s}q^v]$ with $\pi\ 0 = 0$;
- λ from $O_{loc}(p^v)$ to $O_{loc}(q^v)$;
- η from $O_{ata}(p^v)$ to $O_{ata}(q^v)$

inducing a bijection Π from $O(p^v)$ to $O(q^v)$ defined as follows

$$\begin{array}{lcl} \Pi\ p\mathbf{o}i.a & = & q\mathbf{o}(\pi\ i).a \\ \Pi\ p.l & = & \lambda\ p.l \\ \Pi\ p.x & = & \eta\ p.x \\ \Pi\ p.x.a & = & (\eta\ p.x).a \end{array}$$

such that

- the symbols of p^v form a permutation of those of q^v:
  $\forall_{0 \leq i \leq \mathbf{s}p^v} p^v{\cdot}i = q^v{\cdot}(\pi\ i)$;
- the corresponding attributable attributes have corresponding types:
  $\forall_{p.x \in O_{ata}(p^v)} R\ p.x = R\ (\eta\ p.x)$;
- the corresponding equations are equal:
  $(\ \alpha := f\ \ldots \beta \ldots\ ) \in E(p^v)\ \equiv\ (\ (\Pi\ \alpha) := f\ \ldots (\Pi.\beta) \ldots\ ) \in E(q^v)$.

□

Suppose to p^v and q^v are isomorphic, and that π is the corresponding bijection from $[0..\mathrm{s}p^v]$ to $[0..\mathrm{s}q^v]$. Unification of p^v and q^v is easily implemented in the encapsulator $\underline{\underline{p}}$. The shell $\underline{p}$ constructs a tuple with $\mathbf{p}^v(Y_1, \ldots, Y_s)$ as vth component. This component should be transformed into $\mathbf{q}^v(Y_{\pi\,1}, \ldots, Y_{\pi\,s})$, replacing constructor $\mathbf{p}^v$ with $\mathbf{q}^v$. Since shell $\underline{p}$ is the *only* function applying $\mathbf{p}^v$, this transformation makes $\mathbf{p}^v$ superfluous. As a result, $\mathbf{p}^v$ can be removed from the split grammar and the visit-function alternative $visit_{p\cdot 0^v}\ \mathbf{p}^v(\ldots)$ may also be removed.

The purpose of unification is the same as it is for splitting and elimination: to map differently shaped trees with the same semantics to the same tree. The result of these strategies is more identical trees and thus, due to sharing, less space consumption. But above all, more identical trees lead to more cache hits.

4.3.4 Folding and normalization

Elimination and unification are static optimizations. We will now discuss optimizations that are partly static and partly dynamic. For example, we might consider a form of elimination that is only applicable in certain contexts. Static analysis determines which contexts make a specific constructor superfluous. The encapsulator for that constructor bases its behavior on the context, a dynamic decision. Such partly static, partly dynamic elimination will be referred to as *folding*.

Another kind of partly static partly dynamic analysis is *normalization*. Static analysis determines contexts which allow for some kind of tree restructuring. Encapsulators dynamically decide whether such contexts emerge, and if so, they restructure accordingly. For example, one might think about balancing binary trees or, on the contrary, linearizing trees.

For folding the emphasis is on restructuring the tree in such a way that fewer attribute instances have to be evaluated. For normalization on the other hand, the emphasis is on restructuring the tree in such a way that it has a normalized form, which hopefully improves sharing (and thus evaluation).

Folding may be thought of as a form of partial evaluation during tree construction.

Folding and normalization are optimizations that are not fully understood yet. Further research is needed to reveal recurring patterns that are amenable for restructuring. However, unlike elimination and unification, folding and normalization have dynamic costs. Therefore, such optimizations should only be applied after careful considerations.

Analysis of semantic functions broadens the possibilities.

Folding

Folding is a strategy that can not easily be characterized in general. Informally, an encapsulator is rewritten to dynamically check for conditions upon which some restructuring may be applied. There exist many ways to restructure trees, with as many conditions to check, but we will only give two examples.

An often recurring technique in attribute grammars is *threading*. With threading, the data flows top-down either from left to right or vice versa. The typical

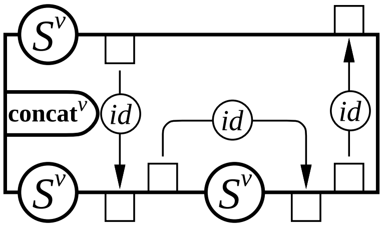


Figure 4.12. Typical threading productions

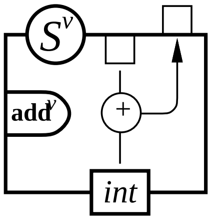


Figure 4.13. Generated

productions for left to right threading are given in Figure 4.12: a *compose* constructor corresponding to functional composition ($\mathbf{concat}^v$), an *identity* constructor corresponding to the identity function id ($\mathbf{goto}^v$) and the *unit* constructor that actually transforms the input ($\mathbf{label}^v$). Threading is used for gathering sequence dependent tree information. In this example, label numbers for goto's are distributed in left to right order.

Other "statement" (S) constructors do not influence label counting and are thus eliminated.

The first example of folding is *compose-identity-folding*. Let $\mathbf{concat}^v$ be a compose production. The shell $\underline{concat}$ associated with it has the form

$$\underline{concat}\ [\ldots, S_1^v, \ldots]\ [\ldots, S_2^v, \ldots] = [\ldots, \mathbf{concat}^v(S_1^v, S_2^v), \ldots] \quad .$$

Furthermore, let $\mathbf{goto}^v$ be an identity production. Compose-identity-folding transforms the vth component of the tuple created by $\underline{concat}$ into

```
if  $S_1^v = \mathbf{goto}^v() \wedge S_2^v = \mathbf{goto}^v() \longrightarrow \mathbf{goto}^v()$
□  $S_1^v = \mathbf{goto}^v() \wedge S_2^v \neq \mathbf{goto}^v() \longrightarrow S_2^v$
□  $S_1^v \neq \mathbf{goto}^v() \wedge S_2^v = \mathbf{goto}^v() \longrightarrow S_1^v$
□  $S_1^v \neq \mathbf{goto}^v() \wedge S_2^v \neq \mathbf{goto}^v() \longrightarrow \mathbf{concat}^v(S_1^v, S_2^v)$
fi
```

Compose-identity-folding effectively reduces a split tree rooted S^v to the smallest possible tree: a tree with at most one $\mathbf{goto}^v()$ node. This example shows that folding is a form of partial attribute evaluation.

The only tree with a $\mathbf{goto}^v()$ node is the "empty" tree $\mathbf{goto}^v()$ itself.

The second example, *constant-folding*, rigorously folds the S^v tree. Partial evaluation during construction yields a tree that may be attributed in $\mathcal{O}(1)$ time. All that is required is a production like $\mathbf{add}^v$ given in Figure 4.13. The shell for $\underline{label}$ may now be transformed from

$$\underline{label} = [\ldots, \mathbf{label}^v(), \ldots]$$

into

$$\underline{label} = [\ldots, \mathbf{add}^v(1), \ldots] \quad .$$

Similarly, the shell for $goto$ becomes

$$\underline{goto} = [\ldots, \mathbf{add}^v(0), \ldots] \quad .$$

However, the work horse is the shell for $concat$

$$\underline{concat}\ [\ldots, \mathbf{add}^v(i_1), \ldots]\ [\ldots, \mathbf{add}^v(i_2), \ldots] = [\ldots, \mathbf{add}^v(i_1 + i_2), \ldots] \quad .$$

Normalization

The purpose of normalization is to restructure the tree into some normal form so that the chance on cache hits increases [Pug88]. Many schemes are possible, we present balancing and linearization.

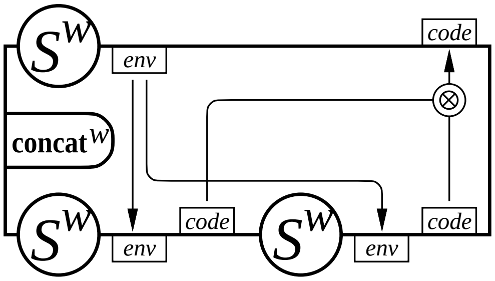
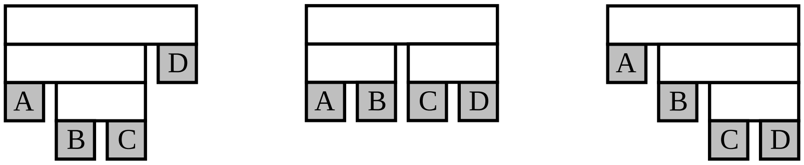


Figure 4.14. Divide and conquer production

Like threading, *divide and conquer* is an often recurring technique in attribute grammars. With divide and conquer, data is first distributed to all tree leafs, where it is processed, upon which the transformed data is synthesized and merged at every node. Figure 4.14 shows a typical example of a divide and conquer constructor: the symbol table env is passed unmodified to both children, and the $code$ computed by the children is merged ($\otimes$) and synthesized.

If the merge operator $\otimes$ of the divide and conquer constructor $\mathbf{concat}^w$ is *associative*, then the data flow associated with instances of that constructor is associative. As a result, trees constructed with $\mathbf{concat}^w$ are amenable for *associative-normalization*: balancing or linearization (see Figure 4.15). Linearization yields normal forms which are more likely to be reconstructed. Balancing produces trees with a more evenly distributed evaluation workload for the left and right child of a $\mathbf{concat}^w$ node. Balanced trees might prove useful in parallel evaluators. If the merge operator is *commutative* in addition to associative, the restructuring possibilities for normalization grow even further.



a. Before **b.** After balancing **c.** After linearization

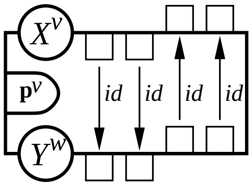
Figure 4.15. Associative-normalization

The crux of the divide and conquer constructors with associative merge operators is that their data flow graph is associative. The graph associated with $\mathbf{concat}^v$ in Figure 4.12$_\triangleleft$ is also associative. Consequently, trees constructed with $\mathbf{concat}^v$ are also amenable for associative-normalization in addition to folding.

Divide and conquer constructors arise naturally. For example, the **sidet**, **sideb**, **stackl** and **stackr** constructors of the BOX grammar are all of that kind. On first glance, the constructors even appear to have associative data flow graphs associated with them. For example, $dpr(sidet^1)$ (Figure 3.33$_{\triangleleft 86}$) distributes the font library *flib* to its children, and it synthesizes the width w and height h. The width is computed as the *sum* of the widths of its children and the height is computed as the *maximum* of the heights of its children. Addition and maximum are both associative. However, the binding attributes must also be synthesized. And a constructor $B^{1\to 2} = \mathbf{sidet}^{1\to 2}(real, B^{1\to 2}, B^{1\to 2})$ is, by definition, not associative.

4.4 Untyping

Closer examination of the optimizations discussed in the previous section reveals that they are more restricted than they need to be. For example, elimination requires a production of the form $X^v = \mathbf{p}^v(X^v)$ with non-crossing copy-rules only. But, a production like $X^v = \mathbf{p}^v(Y^w)$ with non-crossing copy-rules only (implying equal attribute types) could also be eliminated, with respect to the induced dtr graph. However, if we were to eliminate root constructor $\mathbf{p}^v$ from $\mathbf{p}^v(T)$, then the *type of the tree* changes from X^v to Y^w. This type mismatch can be remedied with *untyping*.



Untyping is a type transformation, that maps every split non-terminal X^v to one single data type $\mathcal{N}$. The split representation $\underline{X} = [X^1, \ldots, X^{\mathbf{v}X}]$ thereby becomes $\underline{X} = [\mathcal{N}, \ldots, \mathcal{N}]$ which means that $\underline{\underline{X}} = (X, [\mathcal{N}, \ldots, \mathcal{N}])$. Neither the encapsulators, nor the shells have to be transformed. Only the data-type for terms has to be changed. For example, for the VARUSE grammar we get

$$\begin{aligned}
\mathcal{N} = \;& \mathbf{root}^1(\mathcal{N}, \mathcal{N}) \\
\mid\;& \mathbf{decl}^1(\mathcal{N}, \mathcal{N}) \\
\mid\;& \mathbf{stat}^1(\mathcal{N}) \\
\mid\;& \mathbf{empty}^1() \\
\mid\;& \mathbf{decl}^2(\mathcal{N}) \\
\mid\;& \mathbf{stat}^2(\mathcal{N}, \mathcal{N}) \\
\mid\;& \mathbf{empty}^2() \\
\mid\;& \mathbf{name}^1(str) \quad .
\end{aligned}$$

All optimizations discussed benefit from untyping. Where elimination, unification, compose-identity-folding, constant-folding and associative-normalization were severely limited by the restrictions on the abstract syntax, after untyping, only the types of the associated attributes matter.

optimization	**drop requirement**
elimination	$p{\cdot}0 = p{\cdot}1$
unification	$\forall_{0\le i\le \mathbf{s}p^v} p^v{\cdot}i = q^v{\cdot}(\pi\ i)$
compose-identity-folding	form $X^v = \mathbf{p}(X^v, X^v)$
associative-normalization	form $X^v = \mathbf{p}(X^v, X^v)$

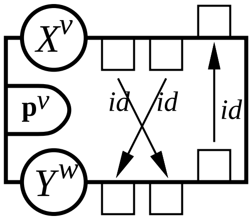
The table above shows the syntactic requirements that may be dropped for the various optimizations, after untyping is applied. They are replaced by restrictions on the attributes. These restrictions are hard to formalize algebraically, since they do not only depend on the *type* of the attributes but also on the *positions* of attributes in the plan icon.

Until now, we have not paid much attention to the positions of the attributes in the plan icons. A disc labeled $p\mathbf{o}i^v$ denoting visit v to non-terminal occurrence $p\mathbf{o}i$ is followed by boxes denoting the inherited attributes $p\mathbf{o}i\mathbf{I}\,v$ which are followed by boxes denoting the synthesized attributes $p\mathbf{o}i\mathbf{S}\,v$. The actual *order* of the inherited (respectively synthesized) attributes does matter: it corresponds with the order on the parameters of the visit-function $visit_{p \cdot i^v}$. Therefore, the *algebraic* requirement $[\forall_{p\mathbf{o}1.i \in p\mathbf{o}1\mathbf{I}v}(p\mathbf{o}1.i = id\ p\mathbf{o}0.i) \in E(p)] \wedge [\forall_{p\mathbf{o}0.s \in p\mathbf{o}0\mathbf{S}v}(p\mathbf{o}0.s = id\ p\mathbf{o}1.s) \in E(p)]$ for elimination is well captured by "non-crossing copy-rules" in our graphical formalism.

Wherever one of the discussed optimizations has a syntactic requirement like $p^v{\cdot}i = q^w{\cdot}j$, in some form or another, this is replaced by the following requirements on the associated attributes:

- the number of associated attributes are equal $|p\mathbf{o}i\mathbf{I}\,v| = |q\mathbf{o}j\mathbf{I}\,w|$;
- the associated attributes are ordered such that if α is the kth attribute of $p\mathbf{o}i\mathbf{I}\,v$ and β is the kth attribute of $q\mathbf{o}j\mathbf{I}\,w$ than the types of α and β are equal;
- equivalent requirements for the synthesized attributes hold.

These requirements boil down to the following: the signatures of the partially parameterized functions $visit_{p \cdot i^v}\ T_1$ and $visit_{q \cdot j^w}\ T_2$ (with T_1 of type $p{\cdot}i^v$ and T_2 of type $q{\cdot}j^w$) must be equal. Pennings, Vogt and Swierstra represent a tree by a tuple of *functions* instead of a tuple of split trees [PSV92b]. In essence, these functions are the split visit-functions partially parameterized with the split trees.



Observe that the order on the attributes of $p\mathbf{o}i\mathbf{I}\,v$ can be chosen arbitrarily. These orders should be chosen with care, if one wishes to apply certain optimizations. For example, if one tries to eliminate production $X^v = \mathbf{p}^v(Y^w)$, one should not order the attributes of $X\mathbf{I}\,v$ and $Y\mathbf{I}\,w$ in such a way that the copy-rules cross in $\mathbf{p}^v$. Naturally, such decisions may conflict with the same wish for $X^v = \mathbf{q}^v(Y^w)$ so that elaborate analysis with an appropriate cost scheme is required.

4.5 Discussion

This chapter discussed optimizations of the visit-function based attribute evaluator of the previous chapter. The optimizations aim at improving the incremental behavior. The term "optimize" is somewhat confusing since an "optimized evaluator" is not optimal (in some sense), it is *better*. It is customary in the field of compiler research to use "optimize" in that way.

The optimizations fall into three categories. The first category optimizes the *implementation*. We have presented two variants, namely splitting and untyping. Splitting enhances incremental behavior and reduces the complexity of the associated dpr graphs so that other optimizations are more likely to be applicable. Untyping is also an implementation optimization intended to improve the applicability of other optimizations.

The second class of optimizations are the *static optimizations*. Elimination removes productions and unification removes productions by substituting them by others. Both increase the chance that different abstract syntax trees map to the same implementation.

Finally, we discussed *dynamic optimizations*. Folding is a form of evaluation during tree construction. It not only saves evaluation time by reducing the tree size, it also increases the chance on cache hits for that same reason. Normalization restructures the tree, fine-tuning the workload during evaluation.

implementation	**static**	**dynamic**
splitting	elimination	folding
untyping	unification	normalization

The above table lists the optimizations discussed in this chapter. The list is certainly not exhaustive.

Encapsulators construct an abstract syntax tree with an alternative implementation. The optimizations incorporated in the encapsulators improve the way trees are stored. The effects of the optimizations are intertwined very closely. Trees will occupy less space and they will share larger parts. Both aspects also lower the evaluation time: smaller trees imply fewer function calls and enhanced sharing means more cache hits.

However, the optimizations are rather unlikely to occur massively in attribute grammars. What is the chance that dependency graphs have a useful form for unification or folding? We hope, however, that the representation optimizations, splitting and untyping, will increase our chances. Note that all optimizations are also possible in plain attribute grammar evaluators, where attribute instances are stored in tree nodes. However, due to the complexity of the unsplit dpr graphs, optimizable patterns are less likely to occur. On the other hand, in our functional setting, we need bindings to resolve intra-visit-dependencies for split visit-functions. Bindings typically complicate the dpr graphs, thereby also reducing the chances on applicable optimizations.

A grammar specification might include equations that use constructors. These equations map to instructions using encapsulators. As a result, a tree computed during attribute evaluation is available either as data tree or as flow tree optimized for visiting. The former representation is needed when the tree is destructed by semantic functions, the latter representation is needed when the tree is assigned to an attributable attribute.

When the attribute grammar evaluator is driven by a language based editor, the editor should use the encapsulators instead of the plain constructors, since an optimized flow tree should be passed to the evaluator. Partial evaluation, performed by folding, is not foreseen to be cached by the visit-function cache. However, since split trees are shared (split constructors are memoized), incremental evaluation is not crippled by folding, or any of the other optimizations.

5

Chapter 5

Computing visit-sequences

Chapter 3 discusses how visit-sequences are *used* to construct a functional evaluator. This chapter discusses how visit-sequences can be *computed* from a grammar specification. We discuss two algorithms that compute visit-sequences, namely Kastens' ordered scheduling and a new algorithm named *chained scheduling*.

Chained scheduling is a variant of Kastens' algorithm that uses a priority-queue driven topological sort that results in visit-sequences with lower cost. In this case, the visit-sequences are tailored for fast incremental evaluation based on cached visit-functions.

The priority-queue driven topological sort performs best, if the graph which is sorted contains the least number of arcs. We will therefore first define dat graphs that replace Kastens' tdp graphs. The dat graphs contain no other arcs than strictly necessary. Therefore, dat graphs are better suited than tdp graphs to extract visit-sequences from. We preset one algorithm, chained scheduling, which is basically a priority based topological sort of the dat graph.

Visit-sequences can not be computed for every attribute grammar, but only for grammars from the class of *partitionable attribute grammars*. The algorithms discussed in this chapter accept only a subclass of partitionable grammars: the so-called *ordered attribute grammars* respectively *dat attribute grammars*. The latter class is slightly larger than the former, as will be proven.

5.1 Grammar classes

Recall from 3.1.3 that visit-sequences only exist if for every non-terminal in the grammar *interfaces* (see Definition 13) can be fixed at generation time. Recall also that an interface for X induces a partial order on the attributes $A(X)$ of X.

Visit-sequences are easily formed once the interfaces for the non-terminals are established. Let $PO(X)$ be the partial orders induced by $interface_f(X)$ computed by some algorithm f. The interfaces are only *compatible* if for every production $p : X_0 \longrightarrow X_1 \ldots X_s$ the graph $dpr(p)[PO(X_0), \ldots, PO(X_s)]$ is non-circular. The reason for this is as follows. A node labeled with p must, due to the plan-based evaluation scheme, compute the attribute instances for each adjacent node

Recal that pasting is denoted with $d_p[d_0 \ldots d_{\mathbf{s}p}]$.

X_i in an order matching its interface. Only if these orders comply with the actual dependencies as described by $dpr(p)$, all instances can successfully be computed in orders matching $PO(X_i)$.

In other words the non-circular graph $dpr(p)[PO(X_0), \ldots, PO(X_s)]$ describes every dependency that has to be taken into account: semantic dependencies and interface dependencies. A total order of this graph yields a visit-sequence for p. Consequently, compatible interfaces for the non-terminals supply us with the visit-sequences for the productions.

Grammars for which interfaces can be determined are important since we can avoid the dynamic construction of the visit-sequences. Several classes of such grammars are defined. In general we observe: the larger the class, the worse the time complexity for determining the interfaces. The simplest class is probably the class of *synthesized-only attribute grammars*.

The UNIX tool YACC is based on synthesized-only attribute grammars.

Definition 18 *Synthesized-only attribute grammar.*
An attribute grammar is synthesized-only if $\forall_{X \in N} A_{inh}(X) = \{\}$.
□

The interfaces for a synthesized-only grammar are trivial: every non-terminal X has one visit and the interface for this visit is $(\{\}, A_{syn}(X))$.

The *largest* class of attribute grammars for which interfaces can be determined is, by definition, the class of *partitionable* [WG84] grammars, also known as *arranged orderly* [Kas80] grammars.

Definition 19 *Partitionable attribute grammar.*
An attribute grammar is partitionable if for every non-terminal X an interface $interface(X)$ exists, such that for every production $p : X_0 \longrightarrow X_1 \ldots X_s$, the graph $dpr(p)[PO(X_0), \ldots, PO(X_s)]$ is non-circular, where $PO(X_i)$ is the partial order induced by $interface(X_i)$.
□

Partitionable grammars form a proper subclass of the strongly non-circular grammars. Nevertheless, the class of partitionable grammars is sufficiently large: all grammars of practical importance turn out to be partitionable [Kas80]. Figure 5.1 presents a (non-practical) grammar that is SNC but not partitionable. In tree $\mathbf{p}(\mathbf{r}())$ the order of evaluation of X's attributes is $X.j \rightarrow X.s \rightarrow X.i \rightarrow X.t$ whereas the order in $\mathbf{q}(\mathbf{r}())$ would be $X.i \rightarrow X.t \rightarrow X.j \rightarrow X.s$. Note that the order on $A(X)$ depends on its context, which is explicitly ruled out in the definition of partitionable grammars.

In the literature [DJL88, Alb91a, EF82] one encounters *l-ordered* grammars more often then partitionable grammars. The two classes are different, although not fundamentally.

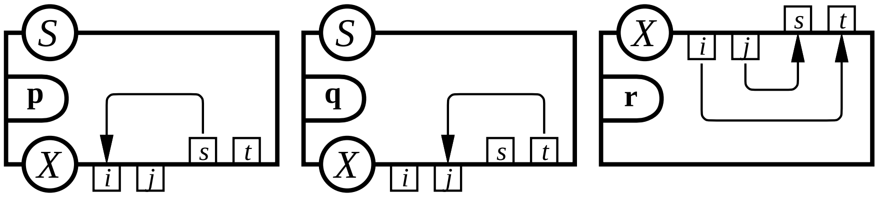


Figure 5.1. A (strongly) non-circular grammar that is not partitionable

Definition 20 *l-ordered attribute grammar.*
An attribute grammar is l-ordered if there exist *total orders* $TO(X)$ for every non-terminal X such that for every production $p : X_0 \longrightarrow X_1 \ldots X_s$ the graph $dpr(p)[TO(X_0), \ldots, TO(X_s)]$ is non-circular.
□

Such total orders $TO(X)$ on $A(X)$ are easily converted into interfaces: cut them into maximal segments of inherited and synthesized attributes. Henceforth, an l-ordered grammar is partitionable.

However, partitionable grammars are not always l-ordered. This is illustrated by Figure 5.2. Due to $\mathbf{r}$ two total orders on $A(X)$ are possible $X.i \rightarrow X.j \rightarrow X.s$ and $X.j \rightarrow X.i \rightarrow X.s$. The former causes a cycle in $\mathbf{p}$ and the latter in $\mathbf{q}$, hence the grammar is not l-ordered. On the other hand $(\{X.i, X.j\}, \{X.s\})$ induces partial orders that fit in all three productions so that the grammar is partitionable.

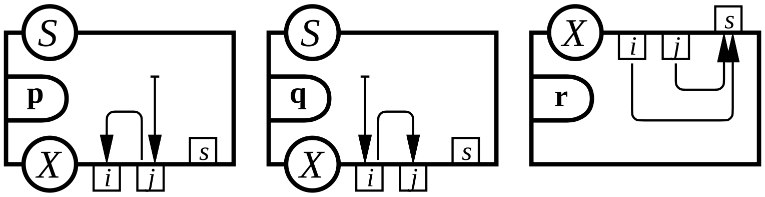


Figure 5.2. A partitionable grammar that is not l-ordered

The claim that the two classes do not *fundamentally* differ is based on the observation that they are equal when we only consider grammars in Bochmann normal form. For such grammars, a dpr graph can not have (induced) dependencies between output attribute occurrences belonging to a single non-terminal occurrence, other than via input attribute occurrences of the same non-terminal occurrence. This means that any two attributes in the same interface set or time slot can be linearly ordered arbitrarily. In other words, a partitionable grammar in Bochmann normal form is l-ordered.

Deciding whether an attribute grammar is l-ordered means finding a total order on $A(X)$ for every non-terminal X. Engelfriet and Filè [EF82] proved that this problem is NP-complete. Faced with this complexity, Kastens [Kas80] exhibited a subclass that can be tested in polynomial time called the *ordered attribute grammars* (OAG). The definition of ordered attribute grammars consists of a particular

The class of ordered attribute grammars is defined by an algorithm.

attribute grammars (page 13)
↓
complete attribute grammars (page 14)
↓
non-circular attribute grammars (page 30)
↓ Fig 2.10$_{\triangleleft 32}$
strongly non-circular attribute grammars (page 32)
↓ Fig 5.1$_{\triangleleft}$
partitionable attribute grammars (page 120)
↓ Fig 5.2$_{\triangleleft}$
l-ordered attribute grammars (page 121)
↓ Fig 5.7a$_{\triangleright 126}$
dat attribute grammars (page 134)
↓ Fig 5.9$_{\triangleright 134}$
ordered attribute grammars (page 121)
↓
synthesized-only attribute grammars (page 120)

Each class is labeled with a page number where a definition for that class is given. Some arrows are labeled with references to counterexamples: grammars that belong to the class just above the arrow but do not belong to the class just below.

Figure 5.3. Grammar hierarchy

procedure that tries to construct the orders. If the construction succeeds, the grammar is ordered.

In the next section Kastens' five step ordered scheduling algorithm will be presented. In section 5.4 we will present a variation on ordered scheduling that allows for optimizations of the visit-sequences. This so-called class of *dat grammars* is larger than the class of ordered grammars, but they appear to be equal if one considers normalized grammars only. Figure 5.3 shows the hierarchy of attribute grammar classes discussed in this thesis.

5.2 Ordered scheduling

In this section, we discuss Kastens' five step ordered scheduling algorithm [Kas80, RT89] in detail. From a practical point of view, the purpose of this algorithm is to compute the visit-sequences. From a more theoretical point, this algorithm computes *orders* on attributes.

Kastens distinguishes two kinds of orders. The first three steps of his algorithm determine a partial order (interface) on the *attributes of each non-terminal*. These orders are deduced from the so-called $tds(X)$ relations "transitive dependencies of a symbol" on $A(X)$. The last two steps determine a total order (visit-sequence) on the *attribute occurrences of each production*. They are determined by a topological sort of the so-called $tdp(p)$ relations "the transitive dependencies of a production" on $O(p)$.

Steps 1 and 2 of Kastens' algorithm compute the $tds(X)$ relations and step 3 extracts the interfaces. The third step is the crucial step that makes the algorithm polynomial. Many partial orders comply with a tds relation, but step 3 fixes a particular choice, which may be an unfortunate one. Step 4 pastes the chosen orders into the $tdp(p)$s to find out whether the choice was lucky or not. If it was, the grammar is said to be ordered. Step 5 performs a topological sort of the thus constructed $tdp(p)$ graphs, creating the visit-sequences.

Step 4 performs the class test.

5.2.1 Ordering attributes of a non-terminal

Ordered scheduling makes a worst case assumption by merging all (indirect) dependencies on attributes of a non-terminal in any context into one dependency graph. In other words, for each non-terminal X the computed relation $tds(X)$ on the attributes of X *covers* the actual dependencies on the attributes of any instance of X in any structure tree. The pessimistic approach in computing the dependencies is crucial for l-ordered grammars: it must always be possible to compute the attributes of X in the order specified by $tds(X)$ irrespective of the actual context. Hence $tds(X)$ must describe all dependencies simultaneously.

$tds(X)$ is much like the $D[X]$ in the SNC test (Figure 2.9$_{\triangleleft 32}$). However, $D[X]$ records only the bottom-up dependencies, whereas $tds(X)$ also records top-down dependencies.

Steps 1 and 2 of the algorithm determine the tds relations. Initially, the tds graphs are empty. They are constructed using an auxiliary relation tdp that records the transitive indirect dependencies in a production. The key operation during the construction is the procedure $add\_arc\_induce(tdp(p),\ \alpha,\ \beta)$ given below. It inserts the arc $\alpha \rightarrow \beta$ into $tdp(p)$ and adds arcs needed to transitively reclose $tdp(p)$. Any added arc may be induced in a $tds(X)$ relation: if $p\mathbf{o}i.a \rightarrow p\mathbf{o}i.b$ is added, where $p\mathbf{o}i$ is an occurrence of X, then $X.a \rightarrow X.b$ is added to $tds(X)$.

```
**proc** $add\_arc\_induce$(**inout** $tdp(p)$ : **graph of** $attrocc$; **in** $\alpha,\ \beta : attrocc$)
$G := tdp(p)$
$tdp(p) := (tdp(p) \cup \{\alpha \rightarrow \beta\})^+$
**for** each (added) arc of the form $p\mathbf{o}i.a \rightarrow p\mathbf{o}i.b \in tdp(p) \backslash G$ **do**
  **if** $p{\cdot}i.a \rightarrow p{\cdot}i.b \notin tds(p{\cdot}i)$ **then** $tds(p{\cdot}i) := tds(p{\cdot}i) \cup \{p{\cdot}i.a \rightarrow p{\cdot}i.b\}$ **fi**
**rof**
```

Step 1 (see Algorithm 5.4$_{\triangleright}$) initializes the $tdp(p)$ graphs with all direct dependencies between the attribute occurrences of p as described by $dpr(p)$. Because add_arc_induce is used, step 1 also initializes the tds relations with all direct dependencies occurring in the grammar.

Step 2 of the algorithm performs a closure operation of the tdp and tds graphs. Every arc in any of the $tds(X)$ relations is induced in every $tdp(p)$ with an occurrence $p\mathbf{o}i$ of X. This closure is computed using a marking scheme for tds arcs. The arcs added by add_arc_induce to the tds relations are *unmarked*. Each *unmarked* arc is induced by step 2 in all relevant tdp relations upon which it is marked (see Algorithm 5.4$_{\triangleright}$). When all tds arcs are marked, the effects of all direct dependencies are induced in the tdp and tds relations.

```
proc $step_0$
  for each non-terminal $X$ do $tds(X) := \{\}$ rof
  for each production $p$ do $tdp(p) := \{\}$ rof

proc $step_1$
  for each production $p$ do
    for each arc $\alpha \to \beta \in dpr(p)$ do
      $add\_arc\_induce(tdp(p),\ \alpha,\ \beta)$
    rof
  rof

proc $step_2$
  while there is an unmarked arc $X.a \to X.b$ in any $tds(X)$ do
    $mark(tds(X),\ X.a,\ X.b)$
    for each occurrence $p\mathbf{o}i$ of $X$ in any production $p$ do
      $add\_arc\_induce(tdp(p),\ p\mathbf{o}i.a,\ p\mathbf{o}i.b)$
    rof
  end

proc $step_3$
  for each non-terminal $X$ do
    $G := tds(X);\ parts(X) := [\,]$
    $sinks(G, syn\_work, inh\_work)$
    while $syn\_work \neq \{\} \vee inh\_work \neq \{\}$ do
      $syn\_part := \{\}$
      while $syn\_work \neq \{\}$ do
        select and remove $X.b$ from $syn\_work$
        $syn\_part := syn\_part \cup \{X.b\}$
        $preds(G, X.b, syn\_work, inh\_work)$
      end
      $inh\_part := \{\}$
      while $inh\_work \neq \{\}$ do
        select and remove $X.b$ from $inh\_work$
        $inh\_part := inh\_part \cup \{X.b\}$
        $preds(G, X.b, syn\_work, inh\_work)$
      end
      $parts(X) = (inh\_part, syn\_part) : parts(X)$
    end
  rof
```

Algorithm 5.4. Ordering the attributes of a non-terminal: step 1–3

```
proc $sinks$ (in $G$ : graph of $attr$
             ; out $syns, inhs$ : set of $attr$)
  $syns := \{\};\ inhs := \{\}$
  for each attribute $X.a$ in $G$ do
    if $is\_sink(G, X.a)$ then
      if $X.a$ is a synthesized attribute of $X$
        then $syns := syns \cup \{X.a\}$
        else  $inhs := inhs \cup \{X.a\}$
  fi fi rof

proc $preds$ (inout $G$ : graph of $attr$; in $X.b$ : $attr$
             ; inout $syns, inhs$ : set of $attr$)
  for each predecessor $X.a$ of $X.b$ in $G$ do
    remove arc $X.a \to X.b$ from $G$
    if $is\_sink(G, X.a)$ then
      if $X.a$ is a synthesized attribute of $X$
        then $syns := syns \cup \{X.a\}$
        else  $inhs := inhs \cup \{X.a\}$
  fi fi rof
```

Algorithm 5.5. Graph operations for step 3

```
proc $step_4$
  for each production $p$ do
    for each occurrence $p\mathbf{o}i$ of non-terminal $X$ do
      for each pair $p\mathbf{o}i.a$ and $p\mathbf{o}i.b$ do
        if $X.a < X.b$ in the order induced by $parts(X)$
          then $tdp(p) := tdp(p) \cup \{p\mathbf{o}i.a \to p\mathbf{o}i.b\}$
        fi
  rof rof rof

proc $condense$(inout $G$ : graph of $attrocc$
               ; in $p$ : $production$)
  for $i := 0$ to $\mathbf{s}p$ do for $v := 1$ to $\mathbf{v}p\mathbf{o}i$ do if $i = 0$
    then condense $p\mathbf{o}0\mathbf{I}\,v$ in $G$ to $V(0, v)$
    else  condense $p\mathbf{o}i\mathbf{S}\,v$ in $G$ to $V(i, v)$
  fi rof rof

proc $step_5$
  for each production $p$ do
    $G := tdp(p);\ code := [\,]$
    $condense(G, p)$
    remove $V(0,1)$ from $G$
    $sources(G, work)$
    while $work \neq \{\}$ do
      select and remove vertex $\alpha$
      case $\alpha$ of
        $V(0, v)$   : $code := code + [suspend(v-1)]$
        $V(i{>}0, v)$: $code := code + [visit(i, v)]$
        else     : $code := code + [eval(\alpha)]$
      esac
      $succs(G, \alpha, work)$
    end
    $code := code + [suspend\ \mathbf{v}p\mathbf{o}0]$
  rof
```

Algorithm 5.6. Ordering the attributes occurrences of a production: step 4 and 5

If any of the tdp relations is circular, the grammar is not *ordered*. A circularity in step 1 indicates an error in the attribute grammar specification: a dpr graph is circular. A circularity during step 2 can indicate that the grammar is circular, but circularities also arise for some non-circular grammars. In any case, it is not l-ordered.

Step 3 (see Algorithm 5.4) determines the interfaces for the non-terminals. Several orders are possible, but instead of choosing the best one (in some sense) or even a compatible one, the choice is fixed a priori. Step 3 maximizes the sizes of interface sets so that the number of visits is minimized. It is basically an eager topological sort using two worklists; syn_work contains only synthesized attributes and inh_work only inherited attributes. Kastens uses a "backward" sort hence we start with the sinks of the graph and after dealing with them, continue with the predecessors (see Algorithm 5.5). The main result of step 3 is the $parts(X)$ data structure. We define $interface_{OAG}(X) = parts(X)$.

If non-terminal X has no attributes, then $interface(X) = [\,]$ or, formulated differently, $\mathbf{v}X = 0$. Unlike Kastens or Reps and Teitelbaum [Kas80, RT89] we feel that a non-terminal can indeed have no visits. The reason for this is that we have a different concept of tree decoration. For us, tree decoration means computing the synthesized attributes of the root. Others feel that the entire decorated tree is the result of decoration. This implies that a node without attributes must still be visited in order to enable decoration of its children.

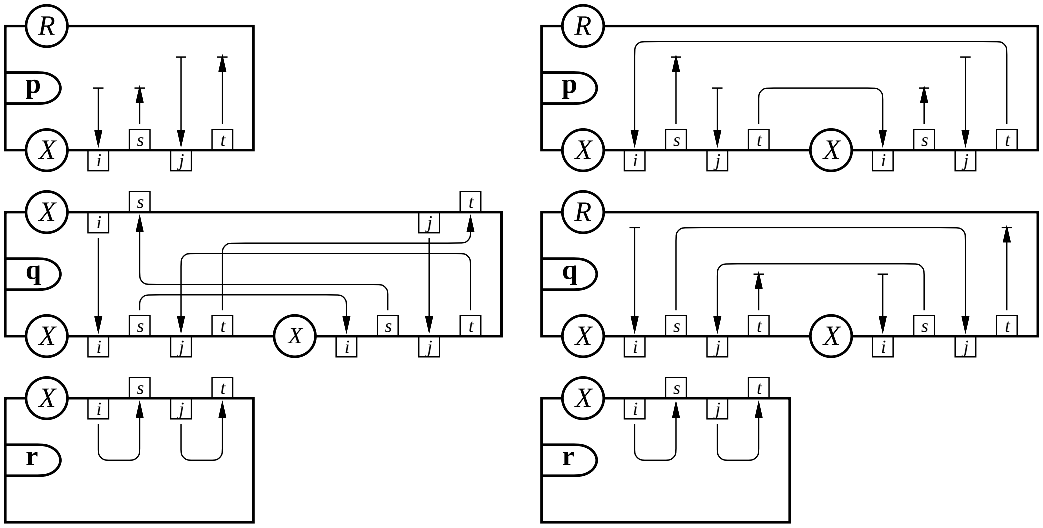
This difference in view has no severe implications for the algorithm; ours does not differ very much from the one presented by Reps and Teitelbaum [RT89]. It does allow us to formulate it in terms of the high level data structure $parts(X)$. Historically, step 3 computes partition sets $A_i(X)$ of attributes. The relation with our data type is as follows: $X\mathbf{I}\,v = A_{2\cdot\mathbf{v}X+1-slot(X\mathbf{I}v)}(X)$ and $X\mathbf{S}\,v = A_{2\cdot\mathbf{v}X+1-slot(X\mathbf{S}v)}(X)$ for $1 \leq v \leq \mathbf{v}X$.

The name $parts$ stems from "partitions", historically the result of step 3.

5.2.2 Ordering attribute occurrences of a production

The third step of ordered scheduling extracts interfaces from the dependencies described by tds relations. Step 4 checks whether the chosen interfaces are compatible. One would expect that, as suggested by the definitions of partitionable and l-ordered grammars, the partial orders $PO(X)$ induced by the interfaces would be pasted into the dpr graphs. If the thus constructed graphs $dpr(p)[PO(p{\cdot}0), \ldots, PO(p{\cdot}\mathrm{s}p)]$ are circular the grammar is rejected. If they are not circular, they can be topologically sorted, producing the plans.

The ordering algorithm for *chained attribute grammars* presented in this thesis follows that approach. Kastens algorithm is obscured by the usage of the auxiliary tdp graphs from the preceding steps instead of the dpr graphs. In other words step 4 computes $tdp(p)[PO(p{\cdot}0), \ldots, PO(p{\cdot}\mathrm{s}p)]$. If any of these graphs is circular the grammar is not ordered (OAG) by definition. If all are non-circular, step 5 performs a topological sort to determine the plans. See Algorithm 5.6 for steps 4 and 5 of ordered scheduling.



a. Not ordered **b.** Not l-ordered

Figure 5.7. Grammars passing steps 1–3 but failing step 4

Many are confused by this usage of the tdp graphs. The tdp graphs *seem* to comprise the interface orders ("the interfaces are extracted from them") so that step 4 appears to be redundant. But once again, the chosen interfaces might not be compatible as will be illustrated shortly.

A circularity in step 4 can originate from two sources. Either the grammar is not l-ordered (so that no interface exists), or it is, but step 3 selected incompatible interfaces. Figure 5.7 illustrates both cases. Both grammars have start symbol R and one extra non-terminal X. X has two inherited attributes $X.i$ and $X.j$ and two synthesized attributes $X.s$ and $X.t$. Both grammars have an identical terminal production t, but different productions p and q. However, in both cases $tds(X) = \{X.i \rightarrow X.s, X.j \rightarrow X.t\}$. With this restriction imposed, 3 interfaces are possible:

1. $\Big[\ \big(\{X.i, X.j\}, \{X.s, X.t\}\big)\ \Big]$,
2. $\Big[\ \big(\{X.i\}, \{X.s\}\big)\ ,\ \big(\{X.j\}, \{X.t\}\big)\ \Big]$ and
3. $\Big[\ \big(\{X.j\}, \{X.t\}\big)\ ,\ \big(\{X.i\}, \{X.s\}\big)\ \Big]$.

Step 3 maximizes interface sets, so it chooses the first one. For the grammar in Figure 5.7a, this partition does not comply with production q as the reader may infer. This means that the grammar is not ordered. The other two partition are both viable. A schoolbook 'trick' is to add an artificial dependency $X.s \rightarrow X.j$ to force Kastens algorithm to find the second partition. However, in the grammar from Figure 5.7b none of the interfaces comply, the grammar is not l-ordered, let alone ordered.

Finally, step 5 computes a total order on the attribute occurrences of a production. This order can be interpreted as the visit-sequence code for that production. Each *output* and *local* attribute occurrence correspond with an $eval$ instruction and each *input* occurrence corresponds with a transfer instruction. A transfer instruction can either be a $visit$ (in case of a synthesized attribute occurrence of a child) or $suspend$ (in case of a inherited attribute occurrences of the parent) instruction.

More precisely, a $visit(i, v)$ instruction in the plan for production p $(1 \leq i \leq \mathbf{s}p$ and $1 \leq v \leq \mathbf{v}p\mathbf{o}i)$ corresponds to the *entire* set $p\mathbf{o}i\mathbf{S}\,v$ and a $suspend(v)$ instruction in the visit-sequence of p corresponds to the *entire* set $p\mathbf{o}0\mathbf{I}\,v$. Because it is an entire interface set that corresponds to a transfer instruction, rather than an individual attribute occurrence, each set of input attribute occurrences is condensed to a single vertex $V(i, v)$ where i is the non-terminal occurrence number and v the visit number. By "condensing set W in graph G to x" we mean the operation of creating a vertex x in G and deleting all vertices in W from G, where x becomes the source of all arcs that formerly emanated from members of W as well as the destination for all arcs that formerly reached members of W. Reflexive arcs on x are discarded.

Observe that $\mathbf{v}p\mathbf{o}i = len\ part(p{\cdot}i)$.

There are two exceptions concerning the transfer of control. The initial transfer is implicit so that no $suspend$ should be generated. This explains the removal of $V(0, 1)$. Furthermore, the final exit must be made explicit. Therefore, an additional $suspend$ is generated.

Algorithm 5.6$_\triangleleft$ implements step 5. The auxiliary functions $sources$ and $succs$ move the sources of the given graph to the worklist respectively move the successors of a given vertex in the given graph to the worklist. They are much like the graph operations for step 3 (see Algorithm 5.5$_\triangleleft$) and are not specified further.

5.3 Dat grammars

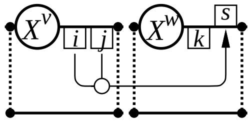
The previous section discussed Kastens' ordered scheduling. The last step of the algorithm schedules the attribute occurrences of each production in a total order complying with the tdp graphs. This total order is chosen rather arbitrarily from a large set of possible orders. We will formulate a cost criterion, and then select the order with minimal costs: chained scheduling computes visit-sequences specially tailored for fast incremental evaluation based on cached visit-functions.

However, closer investigation of the tdp graphs reveals that they contain more edges than necessary for the last step. This means that if the topological sort were applied to a less restrictive graph, the costs could be reduced even further. This section defines dat graphs which are meant to replace Kastens' tdp graphs. The dat graphs contain no other arcs than strictly necessary.

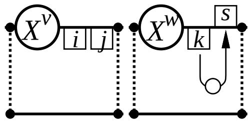
As explained in the previous section, a grammar is *ordered* if there is no cycle in any of the tdp graphs. *Dat grammars* are attribute grammars whose dat graphs are cycle free. The class of dat grammars is slightly larger than the class of ordered grammars.

5.3.1 Arc surplus in the tdp graphs

The tdp graphs that are topologically sorted in step 5 of ordered scheduling have a surplus of arcs. How these spurious arcs are added by step 2 as well as step 4, as we illustrate.



Let p be a production on X where a synthesized attribute s of visit w depends only on the inherited attributes i and j of visit v, with $v < w$. It might be wise to compute s as soon as i and j are available, which means for this example in compartment v. However, since step 4 of ordered scheduling adds arcs between the attributes of the respective partitions, s is explicitly made to depend on the inherited attribute k of visit w. Thus, the arcs added in step 4 to the tdp graph inhibit the *computation* of s in compartment v.



Step 2 also introduces spurious arcs in the tdp graph. In the example in the previous paragraph, there is a reason for s being scheduled in a different compartment than i and j. Probably, there exists a production q on X whose inherited attribute k of visit w is used in the defining equation for s in q. Arc $k \to s$ is added to $tds(X)$ to correctly infer the partitioning information. Arcs in $tds(X)$ are in turn induced in tdp graphs, amongst others in $tdp(p)$. The arc $k \to s$ induced by step 2 in $tdp(p)$ prevents, once again, the scheduling of the computation of s in compartment w.

5.3.2 Definition of dat graphs

The use of tdp graphs in step 5 is based on the overly strict interpretation of the definition of ordered attribute grammars. This definition states that there is a total order in which attributes of a non-terminal *can* be computed. The evaluator however has the freedom to compute attributes earlier or later than is dictated by these orders. The only important aspect is that visit-sequences adhere to the interfaces.

In order to exploit the freedom in scheduling computations, it is better to use graphs that contain no spurious arcs. In the following we define dat "dependencies of attributes and transfers" graphs that are better suited as input to step 5 than the tdp graphs.

The vertices of the dat graph

We start with defining the set of vertices. We distinguish two kinds. Firstly, we associate a vertex with each attribute occurrence. Secondly, we associate a vertex with each visit to a non-terminal occurrence. The former is known as *attribute* vertex and the latter as *transfer* vertex.

Attribute vertices correspond to instructions to compute attributes, and transfer vertices correspond instructions that transfer control. This distinction is natural: step 5 orders pieces of code, so the vertices should be code.

Higher-order grammars are first reduced.

Visits to the *parent* are treated different from visits to *children*. In case of a transfer to the parent, also known as suspend, synthesized attributes are *input* values and inherited attributes are *output* values which is just the inverse of "normal" visits.

As an example take a (two visit) left-hand side with the following interface: $[\ (\{i\},\{s\})\ ,\ (\{j\},\{t\})\ ]$. In this example the first suspend synthesizes s and then inherits j. The second suspend synthesizes t and exits. One could argue that this last suspend is only "half"; it computes t, but does not need any results. What we have brushed under the carpet is that a similar situation arises at the beginning except that it is precisely the inverse: without giving output we get input i.

To model these half visits, we create an additional transfer vertex 0 for the parent that represents half the initial visit. The last visit of the parent also acts as half a visit. The set of vertices $v_{dat}(p)$ of the $dat(p)$ graph is thus defined as

$$v_{dat}(p) = O(p) \cup \{(p\mathbf{o}i, v) \mid 0 \le i \le \mathbf{s}p \wedge 1 \le v \le \mathbf{v}p\mathbf{o}i\} \cup \{(p\mathbf{o}0, 0)\} \quad .$$

The arcs of the dat graph

The arcs in the dat graph represent semantic dependencies and interface requirements.

As an example of a semantic dependency, consider the attribute vertices α and β. Attribute vertices stand for the defining equation of that attribute. Thus, if the equation for β refers to α, then β should be scheduled *after* α. This is forced by an arc $\alpha \rightarrow \beta$.

As an example of an interface requirement, consider the attribute vertices $p\mathbf{o}i.j$ and $p\mathbf{o}i.t$ and the transfer vertex $(p\mathbf{o}i, v)$. Suppose that $p\mathbf{o}i$ is a right-hand side symbol and that $p\mathbf{o}i.j \in p\mathbf{o}i\mathbf{I}\,v$ and $p\mathbf{o}i.t \in p\mathbf{o}i\mathbf{S}\,v$. Visit v to $p\mathbf{o}i$ may only be scheduled after $p\mathbf{o}i.j$ has been computed, which is represented by an arc $p\mathbf{o}i.j \rightarrow (p\mathbf{o}i, v)$. Likewise, $p\mathbf{o}i.t$ is computed by visit v. This relation is captured by the arc $(p\mathbf{o}i, v) \rightarrow p\mathbf{o}i.t$.

As a second example of an interface requirement consider the two transfer vertices $(p\mathbf{o}i, v)$ and $(p\mathbf{o}i, w)$, with $v < w$. Naturally, visit v to $p\mathbf{o}i$ must be scheduled before visit w. Such a dependency is forced with $(p\mathbf{o}i, v) \rightarrow (p\mathbf{o}i, w)$.

The set $e_{dat}(p)$ is partitioned into the above three kinds of arcs: arcs from attribute vertices to attribute vertices $e_{aa}(p)$, transfer–transfer arcs $e_{tt}(p)$ and mixed arcs $e_{mix}(p)$.

attribute–attribute All arcs from the dpr graph will occur in the dat graph. These arcs define the semantic dependencies that have to be obeyed. Since the dpr graph describes dependencies between attribute occurrences, these arcs are indeed attribute–attribute arcs.

$$e_{aa}(p) = dpr(p)$$

transfer–transfer For each non-terminal occurrence in p, $e_{tt}(p)$ contains an arc from visit v to visit $v+1$, including the artificial transfer vertex 0 of the parent. These arcs force an order on visits.

$$\begin{array}{rcl} e_{tt}(p) & = & \{(p\mathbf{o}i, v) \rightarrow (p\mathbf{o}i, v+1) \mid 0 \leq i \leq \mathbf{s}p \wedge 1 \leq v < \mathbf{v}p\mathbf{o}i\} \\ & \cup & \{(p\mathbf{o}0, 0) \rightarrow (p\mathbf{o}0, 1)\} \end{array}$$

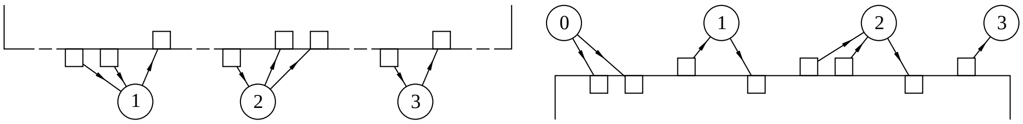
The set $e_{mix}(p)$ contains attribute–transfer and transfer–attribute arcs.

mixed Finally $e_{dat}(p)$ contains arcs to obey the interface as inferred during step 3. We distinguish between arcs describing the interface for the parent, and arcs describing the interface for the children.

$$e_{mix}(p) = e_{mix0}(p) \cup e_{mix1}(p)$$

For each transfer vertex that belongs to a visit v of a *child*, $e_{mix1}(p)$ contains arcs from the inherited attributes of visit v and the outgoing arcs to synthesized attributes of visit v, as illustrated in Figure 5.8a.

$$\begin{array}{rcl} e_{mix1}(p) & = & \{p\mathbf{o}i.j \rightarrow (p\mathbf{o}i, v) \mid 1 \leq i \leq \mathbf{s}p \wedge 1 \leq v \leq \mathbf{v}p\mathbf{o}i \wedge p\mathbf{o}i.j \in p\mathbf{o}i\mathbf{I}\, v\} \\ & \cup & \{(p\mathbf{o}i, v) \rightarrow p\mathbf{o}i.s \mid 1 \leq i \leq \mathbf{s}p \wedge 1 \leq v \leq \mathbf{v}p\mathbf{o}i \wedge p\mathbf{o}i.s \in p\mathbf{o}i\mathbf{S}\, v\} \end{array}$$



a. *mixed* for child ($mix1$ arcs) **b.** *mixed* for parent ($mix0$ arcs)

Figure 5.8. The *mixed* arcs for a three-visit non-terminal

For the vertex associated with suspend v of the parent $e_{mix0}(p)$ contains incoming arcs from the *synthesized* attributes of visit v and outgoing arcs to the inherited attributes of visit $v+1$. This is illustrated in Figure 5.8b. Thus, transfer vertex 0 has no incoming arcs, and transfer vertex $\mathbf{v}p$ has no outgoing arcs.

$$\begin{array}{rcl} e_{mix0}(p) & = & \{p\mathbf{o}0.s \rightarrow (p\mathbf{o}0, v) \mid 1 \leq v \leq \mathbf{v}p\mathbf{o}0 \wedge p\mathbf{o}0.s \in p\mathbf{o}0\mathbf{S}\, v\} \\ & \cup & \{(p\mathbf{o}0, v-1) \rightarrow p\mathbf{o}0.i \mid 1 \leq v \leq \mathbf{v}p\mathbf{o}0 \wedge p\mathbf{o}0.i \in p\mathbf{o}0\mathbf{I}\, v\} \end{array}$$

. . .

The dependencies between attributes and transfers in a production p are described by the graph $dat(p)$ defined as $(v_{dat}(p), e_{dat}(p))$. This graph normally has one source namely the vertex associated with transfer 0 of the parent $(p\mathbf{o}0, 0)$. However, each semantic function that is constant introduces another source. Furthermore, a synthesized only visit, such as $visit_{N^1}$ in the VARUSE grammar, to a child is also a source.

On the other hand, the last transfer vertex of the parent $(p\mathbf{o}0, \mathbf{v}p)$ is the general sink of a dat graph. Each other sink is the ultimate end of a dead-end.

5.3.3 New steps 4 and 5

The new step 4 sets up a dat graph as defined above. If one of the constructed dat graphs is cyclic, the grammar is not a *dat* grammar. In other words, step 4 performs the *member-ship* test for dat grammars.

Visit-sequences are determined in step 5 by a topological sort of the dat graphs. Any total order that complies with the dat graph of production p, describes a valid visit-sequence for p. In the next section, we present an algorithm that selects the "optimal" total order for incremental attribute evaluation based on memoized visit-functions.

Optimal is to be understood as with the lowest costs in some cost scheme.

Although we only distinguish two kinds of vertices in a dat graph, we do distinguish four *categories* of vertices, each of which maps to a different visit-sequence instruction. The mapping of dat vertices to visit-sequence instructions is given below.

- *suspend vertex*
  A vertex of the form $(p\mathbf{o}0, v)$.
  For such a vertex step 5 emits a $suspend(v)$ instruction.
- *visit vertex*
  A vertex of the form $(p\mathbf{o}i, v)$ with $i > 0$.
  Step 5 emits a $visit(i, v)$.
- *eval vertex*
  A vertex of the form $p\mathbf{o}i.a \in O_{out}(p)$ or $p.l \in O_{loc}(p)$.
  Step 5 emits an $eval(p\mathbf{o}i.a)$ respectively $eval(p.l)$.
- *auxiliary vertex*
  A vertex of the form $p\mathbf{o}i.a$ such that $p\mathbf{o}i.a \in O_{inp}(p)$.
  Step 5 emits nothing.

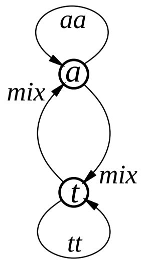
5.3.4 Comparing dat grammars with ordered grammars

We address the question of how dat grammars relate to ordered grammars. We prove that the class of dat grammars is a superset of the class of ordered grammars. In order to prove this, we first show that any path from attribute vertex to attribute vertex in the dat graph has a corresponding path in the tdp graph. In particular, this holds for circular paths which completes the proof.

Lemma *Path in a dat graph.*
Each segment of any element in the set of arc-sequences described by the following regular expression describes a path in a dat graph.

$$\Big(\big(\xrightarrow{aa}\big)^* \xrightarrow{mix} \big(\xrightarrow{tt}\big)^* \xrightarrow{mix} \Big)^*$$

□



proof
A *pawn* traversing a dat graph by hopping from vertex to vertex can be in two modes: attribute-mode or transfer-mode, depending on the kind of vertex the pawn is placed on. The pawn remains in attribute-mode while traversing $\xrightarrow{aa}$ arcs, it switches from mode when traversing a $\xrightarrow{mix}$ arc and it remains in transfer-mode while traversing $\xrightarrow{tt}$ arcs; all by definition of dat graphs. As a result the pawn path is a $\xrightarrow{mix}$ separated sequence of $\xrightarrow{aa}$ arcs and $\xrightarrow{tt}$ arcs.
□

A path $\xrightarrow{mix} \left(\xrightarrow{tt}\right)^* \xrightarrow{mix}$ in a dat graph is called an *interface segment*. A pawn traversing a interface segment starts at an attribute vertex which is followed by one or more transfer vertices and ends at an attribute vertex.

Lemma *Interface segment.*
An interface segment has the following form

$$p\mathbf{o}i.a \xrightarrow{mix} (p\mathbf{o}i, v) \xrightarrow{tt} (p\mathbf{o}i, v+1) \xrightarrow{tt} \ldots \xrightarrow{tt} (p\mathbf{o}i, w) \xrightarrow{mix} p\mathbf{o}i.b$$

where $slot(p\mathbf{o}i.a) < slot(p\mathbf{o}i.b)$.
□

proof
By definition, only a mixed arc $\xrightarrow{mix}$ connects an attribute vertex $p\mathbf{o}i.a$ with a transfer vertex $(p\mathbf{o}i, v)$ or vice versa. Likewise, only a transfer–transfer arc $\xrightarrow{tt}$ connects a transfer vertex $(p\mathbf{o}i, v)$ with a transfer vertex $(p\mathbf{o}i, v+1)$. This explains the form of an interface segment.
The slot associated with $p\mathbf{o}i.a$ is smaller than the slot associated with $p\mathbf{o}i.b$. For $i > 0$ this holds because $p\mathbf{o}i.a$ is an inherited attribute for visit v and $p\mathbf{o}i.b$ is an synthesized attribute of visit w, with $v \leq w$. For $i = 0$ this holds because $p\mathbf{o}0.a$ is a synthesized attribute of visit v and $p\mathbf{o}i.b$ is an inherited attribute of visit $w+1$ (by definition of $e_{mix0}(p)$), with $v \leq w$.
□

The above lemma shows that an interface segment in $dat(p)$ connects two attribute vertices $p\mathbf{o}i.a$ and $p\mathbf{o}i.b$ of the same non-terminal. It represents the interface requirement for $p{\cdot}i$ that $p{\cdot}i.a$ is computed before $p{\cdot}i.b$. Kastens ensures this requirement by pasting the arc $p\mathbf{o}i.a \rightarrow p\mathbf{o}i.b$ into $tdp(p)$ in $step_4$. The crux of the following lemma is precisely the fact that $dat(p)$ as well as $tdp(p)$ enforce this interface requirement.

Lemma *Path correspondence.*
If there is a path from attribute vertex α to attribute vertex β in the $dat(p)$ graph, then there is a path from α to β in $tdp(p)$.
□

proof
According to the first lemma, a path in the dat graph is a segment from the following sequence, where V_i denote vertices.

$$V_0 \xrightarrow{aa} \ldots \xrightarrow{aa} V_n \xrightarrow{mix} V_{n+1} \xrightarrow{tt} \ldots \xrightarrow{tt} V_{m-1} \xrightarrow{mix} V_m \xrightarrow{aa} \ldots$$

According to the second lemma, each interface segment can be short circuited by a single arc $\xrightarrow{is}$.

$$V_0 \xrightarrow{aa} \ldots \xrightarrow{aa} V_n \xrightarrow{\qquad\qquad\qquad\qquad is \qquad\qquad\qquad\qquad} V_m \xrightarrow{aa} \ldots$$

Every vertex V_i in this sequence is an attribute vertex. Each arc $\alpha \xrightarrow{aa} \beta$ is an attribute–attribute arc induced by $dpr(p)$. It is also present in $tdp(p)$ since $step_1$ induces $dpr(p)$ arcs in $tdp(p)$. Each arc $p\mathbf{o}i.a \xrightarrow{is} p\mathbf{o}i.b$ represents an interface requirement. Because $slot(p\mathbf{o}i.a) < slot(p\mathbf{o}i.b)$ this arc is induced in $tdp(p)$ by $step_4$. In other words, every arc of the short circuited path occurs in $tdp(p)$.
□

A cycle is a path. Thus, a cycle in the $dat(p)$ graph can be mapped onto a cycle in the $tdp(p)$ graph. This completes our theorem.

Theorem *Dat grammar.*
Every ordered grammar is also a dat grammar.
□

proof
We have to prove that every grammar that passes $step_4$ of Kastens' ordered scheduling, also passes dat_step_4. This is equivalent to proving that if a grammar fails dat_step_4, it must also fail $step_4$. Formally, this means that if the $dat(p)$ graph contains a cycle, the $tdp(p)$ graph is also circular.

Assume that there is a cycle in $dat(p)$. This cycle must contain an *attribute* vertex α since transfer vertices are totally ordered according to $e_{tt}(p)$. Since α is part of a cycle, there is a path from α to α in $dat(p)$. According to the above lemma, this means that there is a path from α to α in $tdp(p)$.

In other words, every cycle in $dat(p)$ has a corresponding cycle in $tdp(p)$.
□

The class of dat grammars is strictly larger than the class of ordered grammars. This is illustrated by the grammar of Figure 5.9▹. The interface of X is $[\ (\{\},\{s\})\ ,\ (\{i\},\{t\})\ ]$; S and N are single visit. The crux of the example is in the computation of attribute $X.t$ in production q. Since "all" the predecessors of $X.t$ are known in the first compartment of q, $X.t$ might be computed there.

This is true since $X.t$ has no predecessors.

The grammar in Figure 5.9▹ is a dat grammar because of the non-circular $dat(q)$ graph in Figure 5.10a▹. When $X.t$ is computed in the first compartment,

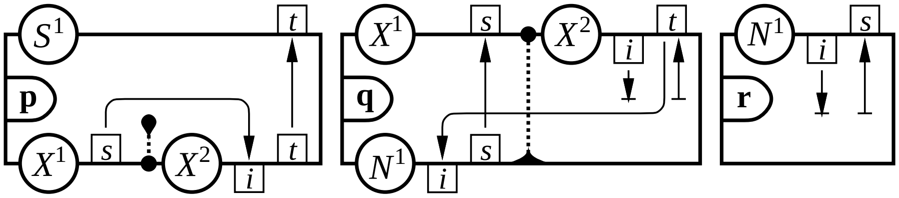
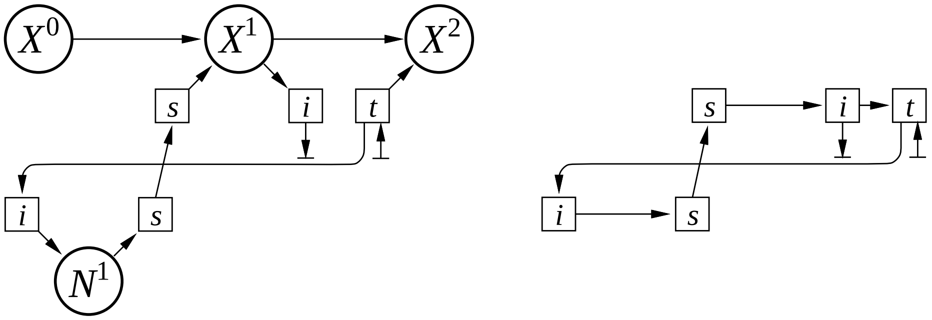


Figure 5.9. A dat grammar that is not ordered

$N.i$ might be scheduled next, giving the *impression* of an illegal back-dependency. This allows the scheduling of visit 1 to N and thus the computation of $N.s$ in the first compartment so that $X.s$ can be synthesized, adhering to the interface.

The grammar is not ordered as is illustrated with the (abridged) $tdp(q)$ in Figure 5.10b. The interface arcs close the cycle set up by the $dpr(q)$ arcs. The arc $X.i \rightarrow X.t$ reflects the misconception that an attribute scheduled to be synthesized by visit v must be computed *during* visit v.



a. The $dat(q)$ graph **b.** The $tdp(q)$ graph

Figure 5.10. The $dat(q)$ and $tdp(q)$ graphs

Note however, that the grammar in Figure 5.9 is not in Bochmann normal form [Boc76]. Synthesized attribute $X.t$ is *used* for computing $N.i$. It is this dependency that introduces the "back-arrow" that creates the cycle in the $tdp(q)$ graph. We have not yet found a dat grammar in Bochmann normal form that is not ordered. Our conjecture is that for grammars in normal form, the class of dat grammars coincides with the class of ordered grammars.

For grammars in Bochmann normal form, the class of partitionable grammars coincides with the class of l-ordered grammars (see page 121).

The applicability of dat grammars is not the larger class. It is the greater freedom of dat graphs over tdp graphs that makes them interesting. The next section gives an example of how to exploit this freedom.

5.4 Chained scheduling

In a traditional setting, where attribute instances are stored at tree nodes, it makes little difference in what order the attributes are computed. In a functional approach, there is a difference. Values computed in previous visits that are referenced in later visits have to be passed explicitly in *bindings*. Changing the scheduling order, alters the life-time of the attribute occurrences and thus the intra-visit-dependencies, which in turn influence the number and contents of bindings. This results in different incremental behavior. Choices made by Kastens for an imperative setting, are not always appropriate in a functional setting. We investigate a different scheduling strategy, named *chained scheduling*.

Chained scheduling is based on Kastens' algorithm. It uses the same first three steps to determine (possible) interfaces. However, the computation of the visit-sequences from the interfaces is different. Step 4 of chained scheduling sets up the dat graphs as described in the previous section. Step 5 is a topological sort of the dat graphs. Many total orders comply with a given dat graph. In this section we present a priority queue based topological sort that optimizes visit-sequences that form the base of a functional attribute evaluator.

5.4.1 Introduction to scheduling costs

The scheduling strategy during step 5 influences incremental behavior of the generated visit-functions based evaluator. We illustrate this with four examples.

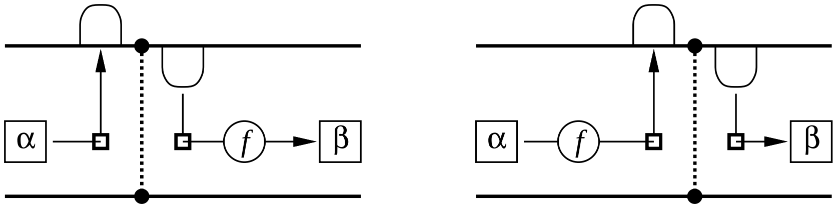
The question in Figure 5.11$_\triangleright$ is where to schedule the computation of $\beta := f\ \alpha$. There are two possibilities. First, α can be put in a binding from the first compartment to the second. In that case β is computed in the *second* compartment. This is illustrated in Figure 5.11a$_\triangleright$. Secondly, $f\ \alpha$ can be computed in the *first* compartment upon which the result is bound for the second compartment. This is illustrated in Figure 5.11b$_\triangleright$. Function f may destroy information. This means that $f\ \alpha$ and $f\ \alpha'$ may be equal even if α and α' differ. Thus, the so-called *eager scheduling* strategy of Figure 5.11b$_\triangleright$ is preferred.

Computations destroy information: functions are not necessarily injective. For example, *addition* destroys information: $3+5$ equals 8, but so does $2+6$.

For functions with more arguments, like f in Figure 5.12$_\triangleright$, eager scheduling is also preferred. Instead of binding a symbol table α and a symbol β for the next visit (Figure 5.12a$_\triangleright$), it is preferred to schedule the lookup in the first compartment and bind only the address γ for the next (Figure 5.12b$_\triangleright$). This means that if, due to an edit action of the user, the symbol β is renamed but still mapped to the same address, the binding remains unchanged. An additional advantage of eager scheduling of a function with multiple arguments is that only the *result* of the function is bound instead of all its *arguments*. As a result, the binding is reduced in size.

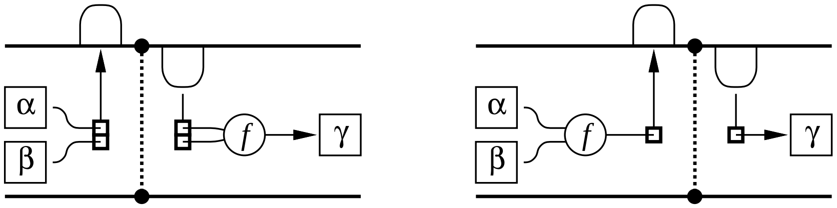
However, reducing the size of bindings is not our main objective; we aim at *optimal incremental behavior*. For example, in case of the production icons in Figure 5.13$_\triangleright$, we still prefer the eager scheduling variant of Figure 5.13b$_\triangleright$, although the variant of Figure 5.13a$_\triangleright$ induces a smaller binding.

There is one situation, where we do not opt for eager scheduling: a function with no arguments, a constant expression. Indeed, eager scheduling (see Figure 5.14b$_\triangleright$) induces a binding with good incremental behavior: it remains unchanged, more precisely, it is constant. Avoiding the binding (as in Figure 5.14a$_\triangleright$)



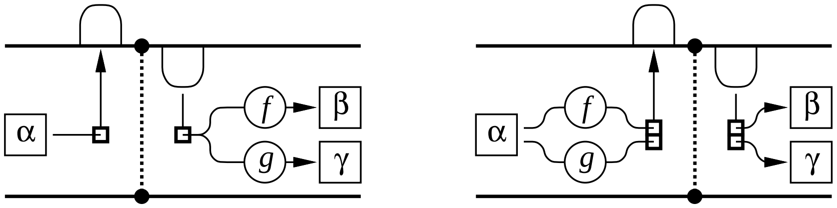
a. Behind the border **b.** Before the border

Figure 5.11. Scheduling $\beta := f\ \alpha$



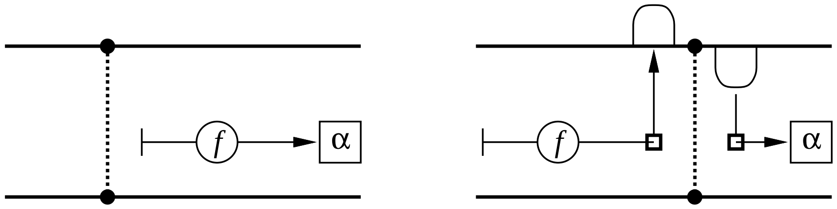
a. Behind the border **b.** Before the border

Figure 5.12. Scheduling $\gamma := f\ \alpha\ \beta$



a. Behind the border **b.** Before the border

Figure 5.13. Scheduling $\beta := f\ \alpha$ and $\gamma := g\ \alpha$



a. Behind the border **b.** Before the border

Figure 5.14. Scheduling $\alpha := f$

does not worsen incremental behavior. On top of that a border crossing is avoided.

We conclude with the formulation of a scheduling strategy. *Attribute vertices of the* dat *graph should be scheduled as soon as possible except for the cases where the semantic functions computing the underlying attribute is a constant.*

5.4.2 Computation classes

We consider a path in a dat graph to be a *computation*. We distinguish four classes of computations within a production, namely *converging*, *diverging*, *unconnected* and *chained* computations. Two criteria determine the class of a computation: its dependency on an inherited attribute of the parent, and secondly its contribution to any of the synthesized attributes of the parent.

A computation that neither depends on nor contributes to an attribute of the parent is called an *unconnected* computation. Since the class of a computation is a *local* property of a production, an unconnected computation is probably a programming error. If a computation does depend on an inherited attribute, but it does not contribute to any result, it is called a *diverging* computation. Diverging computations may indicate programming errors too. However, it is quite natural not to inspect all input attributes of a production, so diverging computations of a single vertex need not indicate an error. Unconnected computations and diverging computations are sometimes called *dead-ends*. Dead-ends are easily detected and eliminated by an evaluator generator.

Converging computations are computations that do contribute to any synthesized attribute of the parent but that do not depend on any of its inherited attributes. Such computations yield a constant value. Constant expressions and syntactic references are examples of converging computations. The last kind of computations is the class of *chained computations*. Chained computations do depend on inherited attributes of the parent and they do contribute to the output. Chained computations are the most common ones.

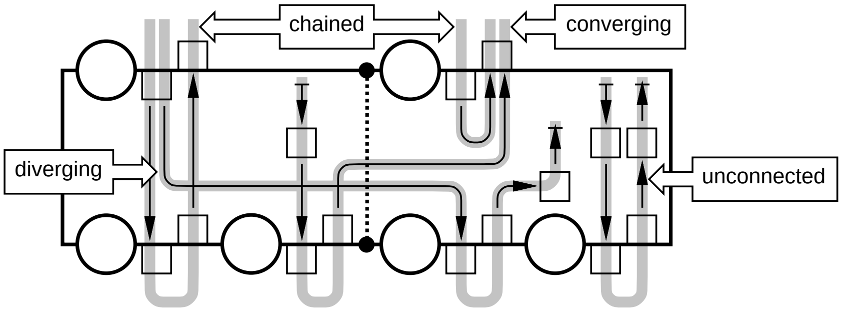


Figure 5.15. The four classes of computations

In the plan icon in Figure 5.15 all computations are marked with a grey curve. Two chained computations, one diverging, one converging and one unconnected computation occur in that production. The converging and diverging computations are not optimally scheduled because both induce an unnecessary intra-visit-dependency.

5.4.3 Priority worklist

The chained scheduling algorithm performs a topological sort of the dat graph. At any moment a worklist contains a set of vertices all of which may be scheduled next. Instead of selecting a random element from the worklist, the "best" one is retrieved. We next describe what constitutes the best element.

Recall that we distinguish four categories of vertices in the dat graph. Transfer vertices are partitioned into suspend and visit vertices and attribute vertices are partitioned into eval and auxiliary vertices. Suspend vertices play a crucial role, since they determine the visit-borders.

When we introduced scheduling costs we observed that the main objective is eager scheduling. In terms of priority, this means that suspend vertices have a *lower* priority than every other kind of vertex. All vertices in a *diverging* (and *unconnected*) computation—if not eliminated after detection—will be placed in a single compartment with eager scheduling: no border-crossing is created.

However, the introduction also learned that eager scheduling should not be applied to *converging* computations since this would introduce unnecessary border-crossings. In terms of priority, this means that the sources of converging computations have a *lower* priority than suspend vertices. Once the source of a converging computation is scheduled, all it successors should be scheduled eagerly since possible border-crossings can no longer be avoided.

priority	**vertices**
low	sources of converging computations
medium	suspend vertices
high	all other vertices

Within the high priority section, we do not distinct between vertices. The medium section contains only suspend vertices. Since suspend vertices are totally ordered by $\xrightarrow{tt}$ arcs, the medium section contains at most one element. There is no need for sub-priorities. However, the low priority section is subdivided.

The dat graph may have various converging computations, each with its own source in the low priority section. Some of these computations may converge to the "main-stream" computation in the first compartment, others may converge later. Suppose that scheduling of the dat graph is proceeded so far that the first suspend vertex yet to be scheduled closes compartment v. Suppose also that the high and medium sections are empty. In that case, the next vertex to schedule must be chosen from the low priority section. It makes a difference which one is selected; we must take one that contributes to the synthesized attributes of compartment v. Such a synthesized attribute exists, since otherwise the suspend vertex that closes compartment v would be in the medium section. We shouldn't take another one into account, since this would introduce an unnecessary binding.

We see that the suspend vertices form the intermediate goals of the computations. Suspend vertices are totally ordered, and the last one is the ultimate goal of the computations in the visit-sequence.

We say that a vertex a has a *goal* g if there is a, possibly empty, path from a to g in the dat graph, and g is a suspend vertex. Note that a vertex has multiple goals in general. We will be interested in a smallest goal with respect to the total order on suspend vertices. Observe that not every vertex has a goal: vertices in diverging and unconnected computations do not contribute to any of the synthesized attributes of the parent and are thus not connected with suspend vertices.

The function $goal$ (Algorithm $5.16_{\triangleright}$) returns, given a dat graph G and a vertex α of G, the transfer number of α's nearest goal or $+\infty$ in case α has no goal. The function $goal(dat(p), \alpha)$ returns an $integer$ in the range $0 \ldots \mathbf{v}p$ or $+\infty$. There is only one vertex, namely $(p\mathbf{o}0, 0)$, that has goal 0.

The procedure $retrieve$ (Algorithm $5.17_{\triangleright}$) retrieves the cheapest element α with respect to graph G from a priority worklist $work$ taking the priorities into account as described above.

5.4.4 Ordering attribute occurrences of a production

We will now present $chained\_step_5$ that replaces $step_5$ of Kastens ordered scheduling. $chained\_step_5$ performs a topological sort of the dat. The topological sort is based on the priority scheme devised in 5.4.1.

The topological sort of chained scheduling (see Algorithm $5.18_{\triangleright}$) requires less pre- and postprocessing than Kastens' topological sort. Most notably, condensing is no longer needed since the transfer vertices in the dat graph represent visits and suspends. As a result, the input attributes of a production become auxiliary vertices in the dat graph. To assure that the final $suspend$ is indeed scheduled last, we make $(p\mathbf{o}0, \mathbf{v}p)$ the ultimate sink of the graph by adding arcs from all other vertices.

Procedure $sources(G, w)$ called from $chained\_step_5$ determines the sources of graph G and initializes worklist w with them. In Algorithm $5.18_{\triangleright}$ the low section of the worklist is initialized that way. It thus contains the sources of all converging computations and the special transfer vertex $(p\mathbf{o}0, 0)$ of the parent. Vertex $(p\mathbf{o}0, 0)$ is retrieved first, since $work.high = \{\}$, $work.medium = \{\}$ and the goal of $(p\mathbf{o}0, 0)$ is the lowest possible: 0. Therefore, instruction $suspend(0)$ is scheduled first. If compatibility with Kastens' visit-sub-sequences is desired, it must be deleted.

```
func goal(in $G$ : graph of vertex; in $\alpha$ : vertex) : integer
  $g := +\infty$
  for each vertex $\beta$ of $G$ do
    if $\beta$ is a suspend vertex $(p\mathbf{o}0, v)$ of $G$ then
      if there is a path from $\alpha$ to $\beta$ in $G$ then $g := g$ min $v$ fi
    fi
  rof
  return g
```

Algorithm 5.16. The *goal* function

```
type worklist = record low,medium,high: set of vertex end
proc retrieve(in $G$ : graph of vertex; inout work : worklist; out $a$ : vertex)
  if $work.high \neq \{\}$
    then select and remove any vertex $\alpha$ from $work.high$
    else if $work.medium \neq \{\}$
      then select and remove any vertex $\alpha$ from $work.medium$
      else  select and remove a vertex $\alpha$ from $work.low$
            such that $goal(G, \alpha)$ is minimal
    fi
  fi
```

Algorithm 5.17. The *retrieve* procedure

```
proc chained_step$_5$
  for each production $p$ do
    $G := dat(p) \cup \{\alpha \rightarrow (p\mathbf{o}0, \mathbf{v}p\mathbf{o}0) \mid \alpha \neq (p\mathbf{o}0, \mathbf{v}p\mathbf{o}0)\}$ ; $code := [\,]$
    $work.high := \{\}$; $work.medium := \{\}$; $sources(G, work.low)$
    while $work.high \cup work.medium \cup work.low \neq \{\}$ do
      $retrieve(G, work, \alpha)$
      case $\alpha$ of
        $(p\mathbf{o}0, v)$        : $code := code +\!\!+ [suspend(v)]$
        $(p\mathbf{o}i > 0, v)$ : $code := code +\!\!+ [visit(i, v)]$
        $\alpha \in O_{out}(p)$ : $code := code +\!\!+ [eval(\alpha)]$
        $\alpha \in O_{inp}(p)$ : $skip$
      esac
      $psuccs(G, \alpha, work)$
    end
  rof
```

Algorithm 5.18. New step 5: chained scheduling

```
proc $psuccs$(inout $G$ : graph of $vertex$; in $\alpha$ : $vertex$; inout $work$ : $worklist$)
  for each successor $\beta$ of $\alpha$ in $G$ do
    remove arc $\alpha \rightarrow \beta$ from $G$
    if $is\_source(G, \beta)$ then
      if $\beta$ is suspend vertex (it has the form $(po0, v)$)
        then $work.medium := work.medium \cup \{\beta\}$
        else $work.high := work.high \cup \{\beta\}$
      fi
    fi
  rof
```

The procedure $psuccs$ removes α and determines the new sources of G after α is removed. It updates the worklist $work$ accordingly, respecting the type invariant of the priority worklist.

The while loop in $chained\_step_5$ has the following invariant. The vertices of the dat graph are partitioned into two sets A and B. The vertices in A are linearly ordered as described by $code$. Every vertex in B is larger than or incomparable with any vertex in A. Initially, A is empty and B contains all vertices of the dat graph. The algorithm ends when B is empty; $code$ then holds the total order. Worklist $work$ is a subset of B containing the smallest elements of B. In other words, it contains the sources of the subgraph of dat induced by B. $work.low$ contains all vertices that have no predecessors in A. $work.medium$ contains "all" suspend vertices and $work.high$ contains the rest.

5.4.5 Optimizing bindings

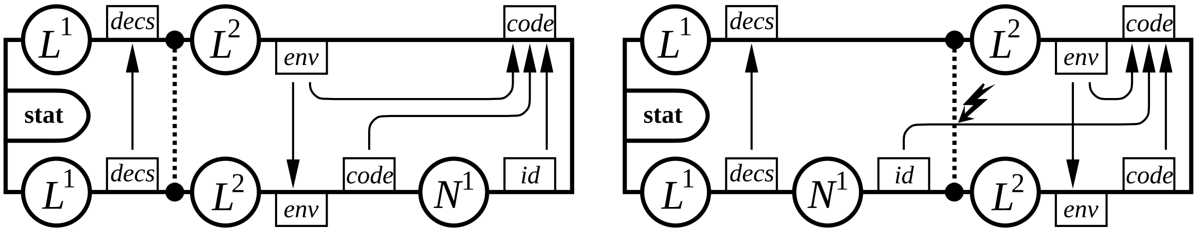
Figures $5.11_{\triangleleft 136}$–$5.14_{\triangleleft 136}$ illustrate that the *contents* of bindings is influenced by scheduling. Figures $5.12_{\triangleleft 136}$ and $5.13_{\triangleleft 136}$ show that the *size* of the binding may be reduced. In the extreme case presented in Figure $5.14_{\triangleleft 136}$ a binding is eliminated altogether. By deferring the computation of $\alpha := f$ till the second compartment, a border-crossing is avoided.

A more elaborate example is presented in Figure $5.19_{\triangleright}$ which shows two ways of scheduling the dat graph of the $stat$ production in the VARUSE grammar. By delaying the converging computation that determines $N.id$, which includes the first visit to N, the binding $L^{1 \rightarrow 2}$ may be kept empty.

We mean the computation consisting of the vertices corresponding with $visit(N,1)$ and $eval(N.id)$.

Scheduling does influence the life-time of attribute occurrences and thus the number of bindings. It is not possible however, to minimize the number of non-empty bindings in an attribute grammar on account of the local characteristics of the productions only. A minimal number of bindings can not be determined by a appropriate priority scheme alone.

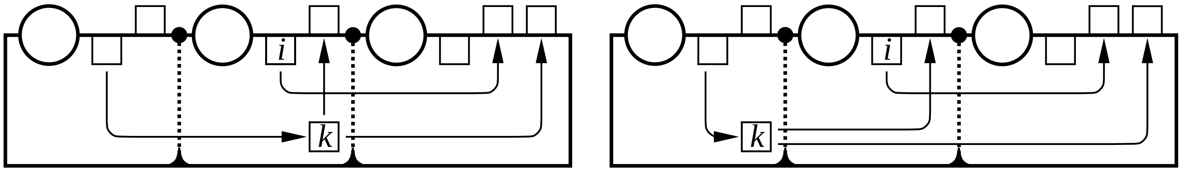
For example, consider the production in Figure $5.20_{\triangleright}$. The difference between Figure $5.20a_{\triangleright}$ and Figure $5.20b_{\triangleright}$ lies in the local attribute k which is scheduled in the first respectively second compartment. As a result, the scheduling in Figure $5.20a_{\triangleright}$ induces only two bindings: on from 1 to 2 containing the (source for) k and one binding from 2 to 3 containing both i and k. Figure $5.20b_{\triangleright}$ on the



a. Deferring converging computations **b.** Eager scheduling for converging computations

Figure **b** shows that eager scheduling introduces a binding. The priority scheme for chained scheduling as discussed in 5.4.3 makes an exception for converging computations, essentially delaying them until necessary. This results in the bindingless order of Figure **a**.

Figure 5.19. Chained scheduling for *stat* of the VARUSE grammar



a. 3 crossings, 2 bindings **b.** 3 crossings, 3 bindings

Figure 5.20. Minimizing the number of bindings

other hand induces three bindings: one from 1 to 2 containing k, one from 1 to 3 also containing k and one from 2 to 3 containing i.

Although the number of *border-crossings* does not depend on the scheduling strategy (it only does so for converging computations), the number of *bindings* does. The reason is that multiple border-crossing may be merged into a single binding. This is what happens in Figure 5.20: because there is a binding from compartment 2 to 3 (containing i), the border-crossing for k in Figure 5.20a gets a free ride. If the intra-visit-dependency using i were not present, both cases would show two border-crossings resolved with two bindings.

This observation is important because it shows that for reducing the number of bindings all border-crossings must be known. In particular, It must be known whether the *bindings for a child* are empty or not. This means that the border-crossings for all productions that may occur as a child influence the number of non-empty wrappers for the production under consideration.

The naive algorithm given below determines a combination of visit-sequences such that the number of non-empty parcels is minimal. It is assumed that there are n productions $p_1 \ldots p_n$ in the grammar, and that the visit-sequences corresponding with the various scheduling orders for production p_i are collected in the set $VSS(p_i)$ (thus $VSS(p_i)$ contains all possible plans for p_i).

```
$b := +\infty$
for all $(v_1, v_2, \ldots, v_n) \in VSS(p_1) \times VSS(p_2) \times \ldots \times VSS(p_n)$ do
  perform emptiness test on visit-sequences $v_1, v_2, \ldots, v_n$
  $bind$ := number of non-empty parcels
  if $bind < b$ then $bind := b$; $(vss_1, vss_2, \ldots, vss_n) := (v_1, v_2, \ldots, v_n)$ fi
rof
{ $(vss_1, vss_2, \ldots, vss_n)$ is a combination of visit sequences
  with the minimal number of parcels $b$.
}
```

The number of total orders complying with a partial order on v vertices can be exponential in v. This means that the number of visit-sequences in $VSS(p_i)$ that can be derived from $dat(p_i)$ may be exponential in the number of instructions. The number of elements in the Cartesian product $VSS(p_1) \times \ldots \times VSS(p_n)$ is already exponential in the number of productions (n) if each of the sets $VSS(p_i)$ contains more than one element. The emptiness test itself is a transitive closure on the parcel dependency graph which has $1 + \sum_{X \in N} \frac{1}{2} \cdot \mathbf{v}X \cdot (\mathbf{v}X - 1)$ vertices. The naive approach is extremely inefficient, but the problem itself seems intrinsically hard.

5.5 Review

Step 5 of Kastens' ordered scheduling algorithm [Kas80] constructs an arbitrary order that complies with the tdp graphs. We associated qualitative *costs* with scheduling orders (visit-sequences) so that the *best* order can be selected. However, the tdp graphs contain too many arcs so that not every visit-sequence that is compatible with the interfaces of the symbols complies with the tdp graph.

This was remedied by defining new graphs, the dat graphs, that contain only those dependencies that describe interface requirements or dpr relations. As expected, the class of dat grammars is bigger, albeit not essentially, than the class of ordered grammars.

We presented one application of cost driven scheduling: chained scheduling. Chained scheduling chooses that order such that every (non converging) computation is scheduled as early as possible. This results in better incremental behavior. Vogt already tried two different scheduling strategies [Vog93], which he named *greedy* and *just in time* evaluation. Greedy evaluation equals eager scheduling as discussed in this chapter. Converging computations are scheduled too early. Just in time evaluation does the opposite. It avoids the bindings for converging evaluation but also postpones all other computations which has a negative effect on incremental behavior. Besides, Vogt did not use dat graphs so scheduling freedom was not optimal.

Note that just in time evaluation does *introduce* bindings for diverging computations.

Steps 1–3 of Kastens' algorithm are also biased against traditional evaluators: the partitions are made as large as possible, making the number of required visits as small as possible. In our setting, incremental evaluation is obtained by memoization of visits. More visits mean *finer grain* and thus better incremental behavior.

The finest grain incremental evaluation is obtained by using one visit-function per synthesized attribute. However, a one-visit-per-attribute scheme creates the largest memoization overhead. Overhead can be reduced by grouping dependent synthesized attributes in one partition. An interesting problem is to devise a new algorithm for computing interfaces coarser than one-visit-per-attribute but finer than Kastens' interfaces.

Storage optimizations in traditional evaluators heavily depend on the life-time of the attribute occurrences [Kas87]. Therefore, it seems that storage optimization might benefit from dat graphs with an appropriate priority driven topological sort.

6

Chapter 6

An implementation

Previous chapters exhibited a novel approach in incremental attribute evaluation, namely memoized visit-functions. It was first devised by Vogt, Swierstra and Kuiper [VSK91] to solve the problems of incremental evaluation of higher-order attribute grammars. Vogt conducted the first implementation experiments [Vog93].

This thesis continues the work of Vogt. The current chapter describes the project of *implementing* a generator for attribute grammar evaluators based on memoized visit-functions. It discusses the architecture of the generator and the generated evaluators and includes some test results. Furthermore, we present a new edit model. We conclude this chapter with a discussion of open problems.

6.1 Structure of the generator

As part of our research on attribute grammars, we have written a generator for attribute grammar evaluators, the LRC processor. The LRC processor consists of a *front-end* which processes a grammar specification into intermediate code and a *back-end* which translates intermediate code to executable code. We have also written a tool, that generates production and plan icons.

6.1.1 Front-end

The front-end of the LRC processor is based on the LRC kernel, written by Matthijs Kuiper for his experiments with parallel attribute evaluation [Kui89]. The LRC kernel consists of a lexical scanner, parser and attribute evaluator. It reads a grammar specification written in SSL, the Synthesizer Specification Language [RT88], and constructs a parse tree. An attribute evaluator traverses the parse tree, checking the types of productions, attributes, equations and semantic functions while constructing the *agstore*. The agstore is a collection of data structures representing the attribute grammar. More specifically it stores lists of productions, equations, non-terminal and attributes, and occurrences of the latter two. These lists are cross-referenced where appropriate.

The Synthesizer Specification Language has been chosen as source language for the LRC processor because the Synthesizer Generator is a well known evaluator generator for which many syntax based editors have been written. By choosing such a wide-spread language, we assure ourself of many grammar specifications to test the LRC processor with.

The initial version of the LRC kernel served as starting point for Vogts experiments. Our project is also based on the kernal albeit a newer version. We have added several modules that together implement a generator for attribute evaluators based on memoized split visit-functions. Each module implements a phase of static analysis corresponding with the algorithms discussed in the previous chapters.

Phase 0

As an initialization, referred to as phase zero, the LRC kernel is activated to read an SSL specification and construct the corresponding agstore.

Phase 1

Phase 1 constructs the visit-sequences. We have implemented both algorithms, Kastens' ordered scheduling as well as chained scheduling. For comparison, generalizations of Vogts greedy and just-in-time scheduling [Vog93] have also been implemented. The first phase includes an algorithm to strip local dead-ends (diverging computations) from the visit-sequences.

Phase 2

The second phase first determines the life-time of attribute and binding occurrences. Furthermore, the bind grammar is induced and the empty parcels are determined.

Phase 3

The third phase of the generator induces the split grammar and the associated encapsulators. A first attempt to elimination has been made, but the other optimizations discussed in Section 4.3 have not yet been implemented.

Each of the four phases of the front-end generates *intermediate code* as well as internal tables for diagnostics and debugging. Phase zero generates intermediate code for equations, semantic functions and the constructors of the abstract syntax tree. Phase one generates the visit-sequences, and phase two the augmented visit-sequences and the bindings. The third phase generates the split grammar and the encapsulators. For an overview, see the following table.

phase	**intermediate code**
0	abstract syntax, equations, semantic functions
1	visit-sequences
2	bindings, visit-functions
3	split syntax, encapsulators

6.1.2 Back-end

The back-end of the LRC processor reads the intermediate code and data generated by the front-end. It generates an attribute evaluator. Three orthogonal characteristics determine the kind of evaluator: the architecture (visit-sequences, VSS visit-functions, FUN visit-functions, SPLIT visit-functions), the model (plain or incremental evaluation) and the target language (PASCAL or C).

Different architectures require different parts of the intermediate code. A standard visit-sequence evaluator may be generated, requiring only the intermediate code of phases zero and one. Alternatively, the intermediate code of phase zero and one may be mapped to visit-functions using the VSS mapping. In that case, no bindings are generated since attribute instances are stored in the abstract syntax tree. Thirdly, pure visit-functions (FUN mapping) may be generated using the intermediate code from phase two as well. The back-end has not yet been extended to generate an evaluator using split visit-functions (SPLIT mapping).

Not every architecture can be combined with every target language or evaluation model. The following table lists valid ($\surd$) combinations.

architecture	**language and model**			
	PASCAL		C	
	plain	incr.	plain	incr.
visit-sequences	$\surd$			
visit-functions from VSS	$\surd$		$\surd$	
visit-functions from FUN			$\surd$	$\surd$
visit-functions from SPLIT				

PASCAL is supported as target language for plain visit-sequence based evaluators because part of the LRC kernel—which is written in PASCAL—is generated by the LRC processor itself; LRC is partly *bootstrapped*. C is chosen as main target language since it is more or less (hardware) platform independent. We have not taken the trouble to write a run-time library for every combination in the above table. This explains for the missing $\surd$s. Unfortunately, this prevents us from comparing architectures, most notably incremental evaluation with visit-sequences versus incremental evaluation with memoized visit-functions.

6.1.3 Rundraw

Plan icons provide for an excellent overview of the attribute dependencies in a production. We have written a tool called RUNDRAW that reads LRC output and writes a POSTSCRIPT file containing the (production or plan) icons of requested productions. The POSTSCRIPT file conforms to a special syntax so that it may be edited by an interactive drawing program.

The use of RUNDRAW is twofold. Grammar writers sometimes use grammar constructions that take the grammar out of the class of ordered or dat grammars: a cycle emerges in one of the tdp or dat graphs. Such cycles are much more easily studied in a graphical representation.

RUNDRAW proved also useful while trying out new techniques. Especially in the cause of writing new scheduling algorithms with different priority schemes (such as chained scheduling) or optimizations (such as elimination), plan icons provide quick insight in the obtained results.

The icons generated by RUNDRAW are in essence the same as the ones appearing in this thesis. However, they are not of *presentation* quality, because RUNDRAW is rather rigid in laying out arrows.

6.2 Structure of generated evaluators

In this section we will describe the structure of a generated evaluator using an incremental model, C as target language and an architecture based on FUN visit-functions.

A generated evaluator consists of two parts: a grammar dependent part and a fixed part, the so-called library. The library can be further subdivided into a library for plain evaluation and a library for incremental evaluation.

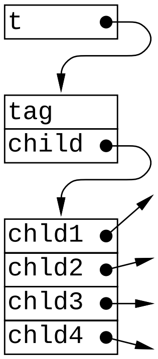
The grammar dependent code is emitted by the backend. It includes code for trees, bindings; semantic functions and equations; the visit-sub-sequences and some tables for diagnostics. The library code for incremental evaluation implements term sharing and visit-function caching. The non-incremental library code implements terms and primitive values and the operations on them.

6.2.1 Library for plain evaluation

The major part of the library implements the primitive values like booleans, integers and strings and the operations on them like logical and ($\wedge$), addition ($+$) and uppercase conversion ($\mathit{upstring}$). Without much trouble, additional operations or even new primitive values can be added to the library.

In addition to primitive values, the library also implements structured terms. To simplify implementation, *terms* are untyped and thus independent of the grammar.

The library exports the type `term`—corresponding to $\mathcal{N}$ from Section 4.4—that implements *any* term: flow trees, data trees as well as bindings. In a future version, `term` would also implement split trees. One would expect the following definition for `term`



```
typedef
  struct termnode
  {
    production_tag tag;
    struct termnode **child;
  } * term;
```

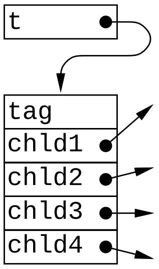
which can be paraphrased as: "a `termnode` consists of a production tag `tag` and a `child` array (first $*$) which holds pointers (second $*$) to nodes".

We assume that `sizeof(int)` $\leq$ `sizeof(void*)`.

The above definition is correct. However, we want to eliminate the extra indirection from the `child` field to the array of children. The idea is to "misuse" a pointer field as tag field, leading to a definition like the following.

```
typedef
  struct termnode
  {
    void * tag;
    struct termnode **child;
  } * term;
```

The child array consists of pointers, and the tag now is a pointer —which must be coerced to integer now and then. The representation we have chosen abbreviates the above definition to the following.



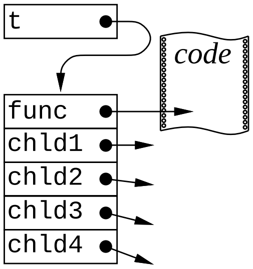
```
typedef
  void * ptr;
typedef
  ptr * term;
```

These declarations can be paraphrased as: "a `term` is an array of pointers, where the first pointer is a coerced production tag".

Let us shortly discuss the properties of the chosen representation. We distinguish two frequently occurring operations on (flow) trees, namely plan selection and destruction (child access). Retrieving a child from a term in the chosen implementation requires indexing, an little amount of work that can not be avoided. Selecting a plan requires two steps: an indexation to extract the tag field and a lookup of the start address of the associated visit-function. The latter is usually implemented as a switch statement. The lookup can be avoided by storing the start addresses of the visit-functions in the tree node in addition to the tag field. For example, a **decl** node is visited twice, so the associated fields in the array would be as follows.

tag value for **decl**
pointer to $visit_L^1$
pointer to $visit_L^2$
pointer to N child
pointer to L child

Such implementation is especially fruitful for split trees because in that case each tree has only *one* visit-function associated with it. Since there is a one-to-one correspondence between the (address of) this visit-function and the tag field, the tag field may be removed at the cost of a lookup when the term is a data tree being destructed. When the tag field is replaced by a pointer to a visit-function, an instance of `term` could be viewed as a *function* that can be "called". In this respect, such a function corresponds to a partially parameterized visit-function as described in Section 4.4.



6.2.2 Library for incremental evaluation

Incremental evaluation is achieved through two mechanisms that are completely separated from plain evaluation: sharing of terms and caching of visit-functions. In order for function caching to be efficient, fast argument comparison is required. Terms are represented uniquely, so that term equality can be realized by pointer equality. In other words, term sharing forms the bases of visit-function caching.

The library for incremental evaluation implements term sharing and visit-function caching. We will also discuss garbage collection.

Memoizing constructors

Term sharing is essentially achieved by caching calls to malloc, the C primitive for dynamic memory allocation. For example, assume a constructor **fork** is applied to the two uniquely represented subtrees T_1 and T_2. In light of the definition of terms given in 6.2.1 this requires a node of three fields: the tag for $fork$, a pointer to T_1 and a pointer to T_2. However, instead of *allocating* this three-field node right away, it is first checked to see whether such a node already exists. If so, that node is returned. If not, the node is allocated.

The following C function C_fork implements the constructor **fork**. Note that a three-field local variable node is temporarily constructed. The constant T_fork denotes the tag for a **fork** node; it must be cast to term.

To avoid name clashes between the tag and the constructor associated with a production, the former is prefixed with T_ and the latter with C_.

```
term  C_fork(term T1,term T2) {
  term node[3];
  node[0]=(term)T_fork;
  node[1]=T1;
  node[2]=T2;
  return SHALLOC(node);
}
```

The macro SHALLOC calls the library routine shared_alloc that we will discuss next. Observe that the first parameter for shared_alloc denotes the size of the node in bytes; divided by sizeof(ptr) it denotes the number of fields, 3 in the **fork** example.

```
#define  SHALLOC(n) shared_alloc(sizeof(n),(term)&n)
```

The constructor cache is a hash table. It is implemented as an array of lists of term nodes. We distinguish three stages in shared_alloc. First, the hash value associated with the given node is computed in i: the hash value of the tag initializes i, and each childs hash value rehashes i. Secondly, entry i of the hash table is searched: the list is traversed (t=NEXT(t)) until the end of the list is reached (t==FAIL) or the node matches the list entry (memcmp(t,node,size)==0). The third step actually adds the node to the hash table if it did not already occur in the list. The function hash_table_add eventually calls malloc.

This is known as "hashing with collision resolution by chaining" [CLR92].

The hash functions HASH_TAG and REHASH must be carefully chosen.

```
term  shared_alloc(int size,term t1) {
  term t2; int num=size/sizeof(ptr); int i;

  i=HASH_TAG(LRC_TAG(t1));
  { int n; for(n=1; n<num; n++) i=REHASH(i,HASH(t1[n])); }

  t2=hash_table[i];
  while (t2!=FAIL && memcmp(t1,t2,size)!=0) t2=NEXT(t2);

  if (t2==FAIL) t2=hash_table_add(i,size,t1);
  return t2;
}
```

The above function does *not* distinguish between tree nodes and elements in a list of the hash table. The truth is, that they are the same. Each `term` is represented by a memory block with the following layout.

-5	-4	-3	-2	-1	0	
next	hash	hits	birth	size	tag	pointers to children

A pointer to such a block is supposed to point to the tag field. The elements to the left of the tag field are for internal administration and diagnostics only. For example, the "hash" field records the hash value as computed in `i` by `shared_alloc`. All nodes with the same hash value are linked via the "next" field. The `HASH` and `NEXT` macro access these fields.

"hits" records the number of hits on the entry, and "birth" records the creation time.

. . .

For primitive values, we distinguish two representations. Firstly, we use a plain C implementation: `int` for integers and `char*` for strings and so forth. Secondly, we use a representation similar to the one for real terms. The diagram below illustrates the memory block layout. Note that in this case, the size of the block needs not be a multiple of `sizeof(ptr)`.

-5	-4	-3	-2	-1	0	
next	hash	hits	birth	size	tag	primitive value

The reason for the second representation of primitive values is uniformity. For example, we need not make a distinction between primitives and non-primitives in `shared_alloc` where we refer to the hash value of the children (`HASH(node[n])`). The uniform approach also allows for a more elegant framework for memoizing the visit-functions.

For each primitive type, two functions are available that coerce one representation into the other. An integer `t` in "shared" implementation is converted to a C integer with the aid of the macro `VALUE(t,int)`.

```
#define  VALUE(t,type)  (*(type*)((term)(t)+1))
```

The conversion the other way round is more complex. Each primitive type requires a library function that creates a shared instance; the function is known as *the unique constructor*. The unique integer constructor is named `C_int`. Unique constructors hide two primitive dependent characteristics: equality (`eq_int`) and hash value (`hash_int(i)`).

The function `C_int` *constructs* an integer term. Hence the constructor like name.

```
term  C_int(int i) {
  struct {production_tag tag; int val;} node;
  node.tag=T_int;
  node.val=i;
  return shared_alloc_prim
    (hash_int(i),  eq_int,  sizeof(node),  (term)&node);
}
```

Equal primitive values must be mapped to the same term. The equality function passed to shared_alloc_prim implements the knowledge of when two primitives are equal. For integer terms, equality is straightforward.

```
bool  eq_int(term t1,term t2) {
  return TAG(t1)==TAG(t2) && VALUE(t1,int)==VALUE(t2,int);
}
```

Furthermore, a hash value is passed to shared_alloc_prim. The hash function maps the primitive value to a hash value in the correct domain. Only the programmer knows the structure and distribution of instances of the primitive type, so he must write a homogeneous distributed hash function. For integers the following hash function usually does a good job [CLR92].

```
hash_value  hash_int(int i) {
  return abs(i) % hash_table_size;
}
```

The library function shared_alloc_prim is used to implement primitive types as is demonstrated in the C_int function above. It has a definition similar to shared_alloc.

```
term shared_alloc_prim(hash_value i,eq_func eq,int size,term t1) {
  term t2;

  t2=table[i];
  while  (t2!=FAIL && !eq(t1,t2))  t2=NEXT(t2);

  if (t2==FAIL) t2=hash_table_add(i,size,t1);
  return t2;
}
```

· · ·

The *correctness* of the evaluator is not in danger when we relax the uniqueness requirement. Any two terms may be *declared different*, that is, represented by a different set of nodes. The only effect is less efficient evaluation since fewer visit-functions are likely to hit. We may decide *not* to memoize certain constructors in order to save space occupied by the hash table. This might lead to more efficient cache lookups. We must realize however, that when an exact copy of a term is constructed all copied nodes will be found in the cache until a non-memoized constructor is applied. Every ancestor of that newly allocated node will create new nodes.

In addition to the equality of visit-function arguments, semantic functions may use *equality on primitive values*. The latter can no longer be implemented with pointer equality, when the uniqueness requirement is dropped.

If two terms are *declared equal*—they are represented by the same set of nodes—they must have the same *meaning*. Non-primitive terms are semantically equivalent if they have the same structure. Primitive terms are semantically equivalent if they denote the same value. For example, when we use integers to implement booleans, we could agree that any non-zero value corresponds with $true$.

Two equal associative arrays may be represented by different AVL trees.

```
bool  eq_bool(term t1,term t2) {
  return TAG(t1)==TAG(t2) &&
         (  VALUE(t1,int)==0 && VALUE(t2,int)==0
         || VALUE(t1,int)!=0 && VALUE(t2,int)!=0
         );
}
```

Suppose we incorporate the above equality function in C_bool. Suppose also that we were to execute the following fragment t1=C_bool(3); t2=C_bool(4);. Then, the pointers t1 and t2 would be equal, that is to say, t2 would also point to the node storing 3, because 3 has the same meaning as 4, namely $true$.

For efficiency reasons, the uniqueness constructors for some primitives are supplied with a sort of cache. For example, for the integer primitive, we prerecord the terms associated with integers in the range $-100\ldots+100$ in a local array. The function C_int either returns an array entry or it falls through to the original routine discussed above, depending on its argument falling in the domain $-100\ldots+100$ or not.

Memoizing visit-functions

Visit-function memoization is achieved by caching function applications. For example, assume a function f is applied to an argument x. Instead of calling $f\ x$ it is first checked whether f was previously called with the argument x. If it was, the previously computed and stored result is returned. If not, the function is called and the result stored for future reference.

The macro MEMO implements cached function application. To cache the application of the C function f to argument x call MEMO(f,x,y), which sets y to the result.

In order for MEMO to be generally applicable, the functions to be cached must conform to a uniform format. The format we have chosen is as follows. A function has *one* input parameter called the *input packet* and *one* result called the *output packet*. Both packets are an array of terms, in effect allowing multiple arguments and results. The "term arrays" are passed by reference. The following two declarations define the type packet and the type visfun. The later describes the signature of functions that can be memoized via MEMO.

```
typedef  term  * packet;
typedef  void  visfun(packet,packet);
```

A packet consists of *terms*. Thus, if an integer is to be passed as argument, it must first be converted to shared representation (using C_int) before it can be put in a packet. This accounts for the term representation for primitives introduced previously.

Let us take the VARUSE grammar as an example, more specifically, the second visit to an L node. The output packet O_L_2_tp for $visit_L^2$ is a structure recording the results of that function. In other words, it consists of $L\mathbf{S}\,2$ and $L\mathbf{BS}\,2$. Since the latter is empty it consists of the $code$ attribute only.

```
typedef struct {
  term code;
} O_L_2_tp;
```

The input packet I_L_2_tp contains not only $L\mathbf{I}\,2$ and $L\mathbf{BI}\,2$ but also the *input tree*, in this case an L tree represented by term. The inherited binding attribute $L^{1\to 2}$ is also stored as a term.

```
typedef struct {
  term tree;
  term P_L_1_2;
  term env;
} I_L_2_tp;
```

Function $visit_L^2$ is implemented by the C function VNT_2_L. VNT functions use a switch statement on the production tag to select the appropriate VPR visit-subsequence.

VNT stands for "visit to non-terminal". VPR stands for "visit to production".

```
void VNT_2_L(I_L_2_tp *ip, O_L_2_tp *op) {
  switch (TAG(ip->tree)) {
    case T_empty : MEMO(VPR_2_empty,ip,op); return;
    case T_decl  : MEMO(VPR_2_decl ,ip,op); return;
    case T_stat  : MEMO(VPR_2_stat ,ip,op); return;
  }
}
```

The type of a visit-function as VPR_2_stat nearly matches visfun: it has two parameters, an input packet ip passed by reference and an output packet op passed by reference. The only problem still left is that the type of the packets is too strong: I_L_2_tp and O_L_2_tp instead of packet. That is easily remedied by type casting the parameters to the right type. The sizes of the packets must remain known for the library routine memo discussed below.

A safe implementation for MEMO(f,ip,op) is f(ip,op). The result is a non-incremental evaluator.

```
#define MEMO(f,ip,op) memo( (visfun)(f)
                          , sizeof(*ip),(packet)(ip)
                          , sizeof(*op),(packet)(op)
                          )
```

The VPR functions are of type visfun. As an example, the second subsequence for $stat$ is given below. It consists of four instruction: an $eval$ instruction to compute the environment for L_2, a $visit$ instruction for the second visit to L_2, a $visit$ instruction for the first visit to N and finally an $eval$ instruction to compute the result $L_1.code$. The four instructions are easily identified in the following C fragment; they immediately follow the declarations.

```
void VPR_2_stat(packet ip, packet op) {
#define IP (*(I_L_2_tp*)ip)
#define OP (*(O_L_2_tp*)op)
  I_L_2_tp I_L_2_2;
  O_L_2_tp O_L_2_2;
  I_N_1_tp I_N_1_1;
  O_N_1_tp O_N_1_1;

  I_L_2_2.env = IP.env;

  I_L_2_2.tree = CHILD(IP.tree,2);
  I_L_2_2.P_L_1_2 = W_stat_1_2_CAST(IP.P_L_1_2).P_L_2_1_2;
  VNT_2_L(&I_L_2_2,&O_L_2_2);

  I_N_1_1.tree = CHILD(IP.tree,1);
  VNT_1_N(&I_N_1_1,&O_N_1_1);

  OP.code = equation_7(O_L_2_2.code, O_N_1_1.id, IP.env);

  return;
#undef IP
#undef OP
}
```

```
plan $stat$
begin 2
  $eval(L_2.env)$;
  $visit(L_2, 2)$;
  $visit(N, 1)$;
  $eval(L_1.code)$
end 2
```

The declarations within a VPR function consist of two parts: two macros and a number of packets. The two macros define IP and OP that are used instead of the arguments ip and op. This allows for clear expressions as IP.env which relies on a type cast of ip to the (intended) type I_L_2_tp. The second part of the declarations consists of an input and output packet for every visit occurring in the subsequence. If instruction $visit(X_i, v)$ occurs, then the packets I_X_i_v and O_X_i_v are declared. Their respective types are I_X_v_tp and O_X_v_tp.

Each visit instruction is preceded by assignments that set the non attribute fields of the associated input packet. Two kinds of assignments are distinguished: assignments to tree parameters (as in I_L_2_2.tree=CHILD(IP.tree,2)) and assignments to binding attributes. Observe also that the input and output packets are indeed passed by reference (as in VNT_1_N(&I_N_1_1,&O_N_1_1)).

Each *eval* instruction in a plan has an equation associated with it. An equation is either implemented as a C function (see for example equation_7), or it is inlined. In the LRC processor, only copy-rules have been inlined. See for example I_L_2_2.env=IP.env.

· · ·

Let us now discuss to the library function memo. It is basically the same as the shared_alloc library routine discussed previously. In essence, it manages a hash table where the cells have the following layout. The sizes of the "input packet" and "output packet" fields depend on the signature of the visit-function "func", but they are a multiple of sizeof(ptr).

-3	-2	-1	0		
next	hits	birth	func	input packet	output packet

`THASH` could look at the address of *term* `ip[n]` or at the hash value recorded for that term. `FHASH` can only use the address of the passed *function*.

In `memo` we distinguish the same three steps as in `shared_alloc`: the hash value is computed, the hash table is searched and depending on the outcome of the search, the visit-function is called and the result stored, or a previously computed result is returned.

```
void memo(visfun f,int ipsize,packet ip,int opsize,packet op) {
  cell c; int ipnum=ipsize/sizeof(ptr); int i;

  i=FHASH(f);
  { int n; for(n=0; n<ipnum; n++) i=REHASH(i,THASH(ip[n])); }

  c=table[i];
  while (c!=FAIL && !EQ(f,ip,ipsize,c)) c=NEXT(c);

  if (c==FAIL) {
    f(ip,op);
    hash_table_add(i,f,ipsize,ip,opsize,op);
  } else {
    memcpy(op,c+1+ipnum,opsize);
  }
  return;
}
```

As with constructor sharing, *correctness* is not in danger when we do not memoize each function call. The effect is less efficient evaluation since functions that are not memoized can not hit. Relaxing visit-function memoization is a space time trade-off: less entries in the function cache, but less function hits.

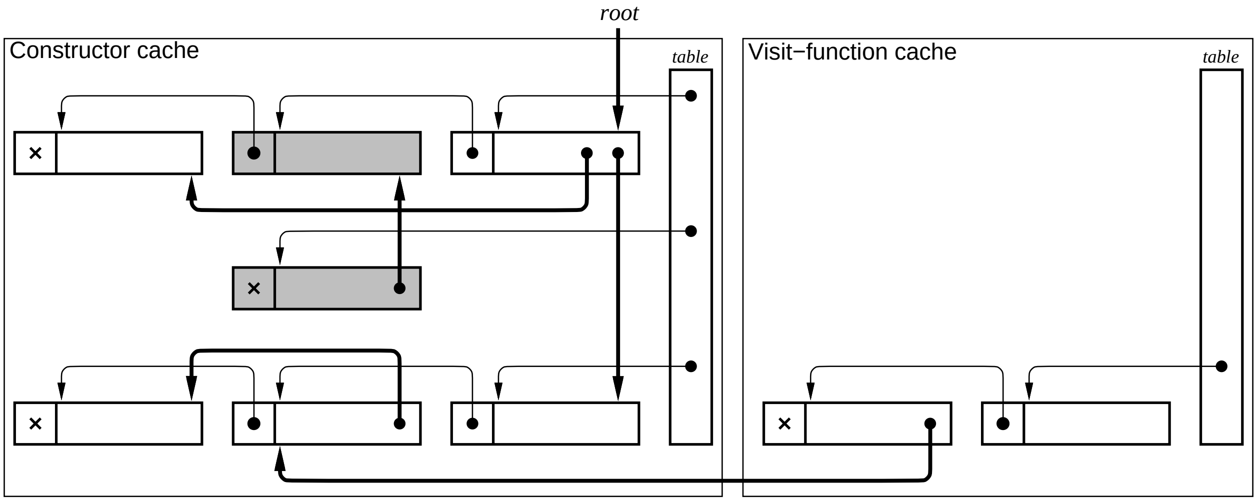
In our implementation, the hash table for visit-function memoization and the hash table for constructor sharing are separate.

Garbage collection

During an edit session, the abstract syntax tree is changed constantly, causing the constructor cache to grow. Visits to a changed tree induce new entries in the visit-function cache. In other words, both caches keep growing, and need to be purged frequently.

Entries from the visit-function cache may be removed at any time. Which purging strategy performs well deserves further investigation.

Entries from the constructor cache can not be removed arbitrarily since they may be referenced from outside the cache. Figure 6.1 presents a snapshot of the two caches. It is assumed that the "main program" has a variable $root$ pointing to the root of an abstract syntax tree. Of course, nodes belonging to the abstract syntax tree can not be deleted. Entries in the visit-function cache also refer to



Thin arrows link the cache entries. Thick arrows either denote parent-child relations (constructor cache) or they link a visit-function argument or result to the underlying term (from visit-function cache to constructor cache).

Entries from the visit-function cache may be deleted without any restriction. Entries from the constructor cache may be deleted when they are not visible from the outside (that is from $root$ or the visit-function cache) via *thick* arrows only. Given the state of the visit-function cache, the grey entries in the constructor cache are garbage.

Figure 6.1. Garbage in the constructor cache

terms (packets contain terms). If a term that is referenced by a visit-function cache entry were to be removed, a so-called *dangling pointer* would be created, making the visit-function cache entry invalid. Hence, terms referenced from the visit-function cache can not be deleted either.

In Figure 6.1, the constructor cache nodes that are not referenced from outside the cache (either by $root$ or by the visit-function cache) are colored grey. They constitute the *garbage* amenable for purging.

The snapshot in Figure 6.1 is simplified with respect to references from the main program. It suggests that only *one* arrow (from $root$) links the constructor cache with the main program. This is only true before and after evaluation.

During evaluation the evaluator creates a stack of (VNT and) VPR frames. Such frames include input and output packets and local variables which are also packets. Packets are arrays of terms. This means that the entire stack contains references to entries in the constructor cache. Thus, when the constructor cache is purged *during* evaluation, all arrows emanating from the stack should also be taken into account.

6.2.3 Generated code

The generated code for an attribute evaluator is grammar dependent code. As an example, we included the code generated for the VARUSE grammar in Appendix B. It is generated from the source given in Appendix A. The output from the generator is given in Appendix C.

Generated code may be subdivided into four groups.

The first group implements terms. It includes code for plain trees: tags and constructors, and code for bindings: tags, types and constructors. Observe from Appendix B that the tags for the plain terms and wrappers are numbered consecutively. The tags do not start at zero because several tags have been reserved by the library for the unique constructors for primitives. The types for the wrappers are mainly used in the `CAST` macros which in turn are used in the visit-sequences to destruct bindings.

The second group implements semantic functions written by the programmer, like $lookup$, and the attribute equations. Equations that are copy-rules are inlined and thus not included. Absent from the code in Appendix B are constant expressions. The LRC processor isolates all constant expressions, merges identical ones and generates an initialization routine. The initialization routine is called upon start up. It computes the values of the constants once, so that repeated access by the equations and semantic functions brings no additional costs.

The third group implements the visit-sub-sequences. It includes the type definition for packets, forward declarations (prototypes) of `VNT` functions, the `VPR` functions that implement the visit-sub-sequences and the `VNT` functions themselves.

The fourth group defines tables for diagnostics. They need only be included when the hash tables for constructor sharing or visit-function caching will be dumped.

The main program in Appendix B is hand written. In a complete system it would be replaced by a language based editor.

6.3 Performance results

Unfortunately, we have not yet extensively profiled the LRC processor. A good test set is hard to define, and on top of that, it is unclear what to measure. In short, we lack a benchmark for incremental attribute evaluation.

6.3.1 Test set

To conduct meaningful tests, we need realistic input. But what is realistic? Observe that four choices have to be made.

First of all we have to choose a "*programming language*". By this choice alone, we restrict ourselves tremendously. Most languages do not feature every interesting construct. Furthermore, most existing languages are essentially one-pass (requiring definition before use) so that efficient compilers can be hand written.

Secondly, we need a *grammar specification* suitable for the generator. Observe that different grammar specifications may describe the same semantics. Consequently, "bad" specification may be written obscuring test results. If a suitable grammar specification in a processable source language is not available in the public domain, one has to be written—a laborious task.

These two choices fix an evaluator. In order to test the efficiency of the evaluator, two more choices have to be made: the input for the evaluator and the edit actions.

The third choice concerns the *input* for the evaluator. Had we generated a C compiler, then we would need for example the C source of a spreadsheet. Had we generated a proof editor, then we would need a derivation for some tautology. Observe that this choice might restrict the test even more. For example, the source of the spreadsheet might expose a typical use of variables like over-usage of global variables, or it might not use `typedefs`. Writing an elaborate input is a laborious task.

The final choice we have to make is how we *edit* the input. *Incremental* behavior is extremely sensitive for the kind of edit action performed: in general global changes are handled poorly, local changes fairly.

6.3.2 What to measure?

When the test set is determined, the tests can be conducted. We are then faced with the question what to measure.

One of the problems is that "objective characteristics" like the size of the set Δ of affected attribute instances are tailored for standard evaluators. In a functional evaluator there are, in addition to attribute instances, binding instances. Should they be regarded as plain attributes while determining Δ?

It would be relatively easy to compare one functional evaluator with another, for example to find out the benefits of chained scheduling, empty bindings, splitting and elimination.

Similar questions are raised when comparing space consumption. In a functional setting tree nodes require far less space than in a traditional setting: no links to attributes or parental nodes nor flags for administration during incremental evaluation. This is compensated by large cache structures.

Two valid characteristics seem to be the number of $eval$ and/or $visit$ instructions executed in the course of (incremental) evaluation. Note that if the number of $eval$ instructions executed equals the size of Δ, the evaluator is optimal.

6.3.3 Results

The test described in this section shows that incremental evaluation with memoized visit-functions is feasible. We have selected a *real* programming language, a *real* program and *real* edit actions.

We have chosen a test set which is stable, documented and readily available. The programming language is PASCAL. As grammar specification we have taken the SSL source that is part of the Synthesizer Generator distribution. As input for the generated PASCAL compiler we have taken a PASCAL program that implements a TROFF-like text formatting program. This format program is described in chapter 7 of the book on software tools in PASCAL by Kernighan and Plaugher [KP81]. An implementation of the format program comes also with the Synthesizer Generator distribution. It is nearly 500 lines long.

As edit actions, we have changed the format program in five different ways. We have (i) kept the program unchanged, (ii) added a statement to the body of a procedure, (iii) deleted a global procedure, (iv) changed the name of a global procedure and (v) we have added a global variable. To measure the effects of incremental evaluation we conducted the following tests. Each variant is decorated three times: (i) once without memoizing the visit-functions, (ii) once with memoized visit-function starting from scratch, that is with an empty cache and (iii) once with memoized visit-functions starting with a cache filled by decorating the original format program.

edit action		**test**		**%**	
	nomemo	scratch	incr.	2/1	3/2
none	4819	3497	0	73	0
	22932	19043	0	83	0
	0.58	0.61	0.00	105	0
add a statement to a procedure	4834	3499	47	72	1
	23008	19056	344	83	2
	0.57	0.61	0.02	107	3
delete a global procedure	4584	3312	599	72	18
	21886	18141	4143	83	23
	0.56	0.60	0.08	107	13
change the name of a global procedure	4819	3497	742	73	21
	22932	19043	4916	83	26
	0.56	0.60	0.09	107	15
add a global variable	4827	3503	2394	73	68
	22974	19083	15942	83	84
	0.57	0.61	0.46	107	75

number of executed *visit*s
number of executed *eval*s
execution time in seconds

The above table shows the test results. We have measured the number of *visit* and the number of *eval* instructions executed during decoration. Furthermore, we have clocked the execution time on a plain Silicon Graphics workstation. The results give us a clue of how much work is saved.

The ratio of the second and the first test (see column labeled 2/1) quantifies the costs of memoization overhead. In the first test (nomemo) memoization of visit-functions is switched off and the second test (scratch) it is switched on. We observe a considerable drop in the number of visits (nearly 30%) and in the number of evaluations (nearly 20%). This reduction is explained by the fact that parts of the tree are shared and decorated similarly. For example, each occurrence of a PASCAL

By setting the macro `MEMO(f,ip,op)` to `f(ip,op)` we effectively switch off memoization.

variable i in the same scope is decorated equally; the same holds for the larger subtree i:=i+1. Execution time however, *increases*. This is due to the overhead costs for memoization: each visit-function call must first be looked up in the cache. Observe that the costs of cache overhead are practically compensated by the decrease of work. The memoized version is only 7% slower than the unmemoized version.

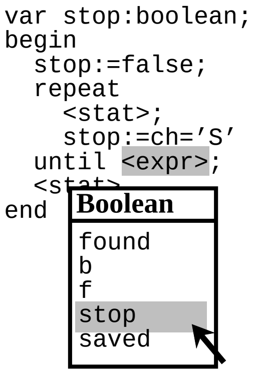
The column labeled 3/2 shows the difference between memoized evaluation from scratch respectively from a cache filled by decorating the original format program. For example, with no change applied, the *top level* visit-function hits, so no *visit* or *eval* instruction is executed; execution time is 0 seconds (see "none" row). As was expected, local changes are handled extremely well incrementally. Adding a statement to the body of a procedure shows a drop of 99% in the number of visits. A non-local change as adding a global variable performs rather poor in our evaluator; 68% of the visits has to be redone. However, it is expected that non-local changes are intrinsically hard problems giving poor results in other approaches as well.

Note that the ratio of the number of executed *eval* instructions and the number of executed *visit* instructions is roughly constant. The constant is grammar dependent, in our tests it is about 6. This means that on the average every plan in the PASCAL grammar has 6 times as many *eval* as *visit* instructions. Depending on the mixture of nodes in the abstract syntax tree associated with the format variants, the constant may vary.

The average execution time is 0.6 seconds, which means that the generated evaluator processes the format program (500 lines) at a speed of 50k lines per minut. Processing does not include parsing, only decoration. However, processing does include administration for diagnostics. The speed-up for incremental evaluation ranges from a factor of 1.3 for global changes (65k lines per minut) to a factor of 30 for local changes (1500k lines per minut). In short, speed is of production quality.

6.4 Towards a user-interface

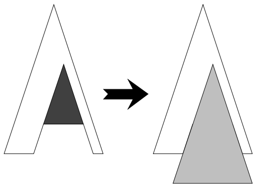
A language based editor has a certain knowledge about the (program-)text under construction. Due to incremental compilation, semantic errors such as type mismatches and undeclared variables are detected and announced to the user instantly. Moreover, the user may *query* the editor about the program under construction. For example, when the editor associates a symbol table with every block in the program, the user may ask for a list of all boolean variables—global as well as local—that are visible at the selected point.



```
var stop:boolean;
begin
  stop:=false;
  repeat
    <stat>;
    stop:=ch='S'
  until <expr>;
  <stat>
end
```

We feel that the strong point of language based systems is the fact that they are better equipped to assist the user. Consequently, a good, powerful and flexible user interface forms the basis of a successful editor. In this section, we sketch a novel approach to model the interaction between the editor and the user. First, we discuss editing.

6.4.1 Editing



Edit operations on trees are modeled through *subtree replacement*. This means that a subtree of the abstract syntax tree can be replaced by another tree with the same root.

To facilitate editing, the notion of "gaps" in the abstract syntax tree is often introduced. Gaps mimic incomplete subtrees. They are easily incorporated by adding a *terminal* production for every non-terminal. For example, in the BOX grammar a production $B = \mathbf{Bgap}()$ could be added. Such an empty production does require equations so that the grammar writer can assign adequate meaning to gaps.

Subtree replacement may be interfaced to the user user via "keyboard driven editing" or by selecting *templates* with a pointing device. A *hybrid* editor supports both methods.

A template is a labeled pair of terms denoted by $l : t \Longrightarrow t'$, in which l is the label, t is a term pattern and t' a term expression. Term pattern t may contain *template variables*. The free variables that occur in term expression t' must also occur in t. Assume that the user selects a subtree T of the abstract syntax tree; T is then known as the *current selection*. When the current selection matches t, a button labeled l appears on the screen. When the user presses the button, subtree T is replaced by T', where T' is obtained from t' by substituting subterms for the free variables as implied by the unification of T with t. For example, a template $swap : \mathbf{stackl}(l, r) \Longrightarrow \mathbf{stackl}(r, l)$ in the BOX grammar allows the user the swap the left and right subtree of a $stackl$ node. The template $ins\_stackl : \mathbf{Bgap}() \Longrightarrow \mathbf{stackl}(\mathbf{Bgap}(), \mathbf{Bgap}())$ inserts a $\mathbf{stackl}$ node at a gap.

l and r are the template variables for template $swap$. They are free variables bound by unifying a template and a tree.

Template editing is less convenient for "smaller" trees such as expressions (as opposed to statements). Typing, as with ordinary editors, seems more appropriate. Of course, this requires a scanner/parser to convert the typed input to an abstract syntax tree. Note that keyboard editing is especially desirable for the primitive values such as integers.

Template editing is not very powerful; it is based on *syntax* only. *Conditional* templates extend the formalism in two ways. First, conditions are associated with templates. The template button only appears on the screen when the selected tree matches the left-hand side of the template *and the condition holds*. Secondly, attributes may be used in the condition as well as in the right-hand side of the template. The following example illustrates both extensions. The template prefixes a statement "`use Some;`" in the VARUSE grammar with a declaration "`var SOME;`", that is, if `SOME` was indeed not defined yet.

Recall that declarations are converted to uppercase.

$$\begin{aligned} add\_var : \mathbf{stat}(N, L) &\Longrightarrow \mathbf{decl}(\mathbf{name}(N.id), \mathbf{stat}(N, L)) \\ &\quad \mathbf{if}\ lookup\ L.env\ N.id == \bot \end{aligned}$$

We have not discussed yet how the abstract syntax tree is presented to the user. The rendering of the abstract syntax tree is known as *unparsing*. One possibility is to render the abstract syntax tree as linearized term, perhaps formatted via indentation. This solution is not attractive for two reasons. Firstly, the solution is not flexible. The grammar writer needs direct control over how to format the tree, which nodes to suppress, where to insert reserved words, where to use fancy fonts and where to break lines. Secondly, decoration determines attribute values, which record properties of the nodes they are attached to. These properties include diagnostic information ("type mismatch") that must somehow be added to the displayed tree at the appropriate node.

A better solution is to *compute*, via decoration, the unparsing in a string representing the tree. For each non-terminal X the set $A(X)$ of associated attributes is extended with a synthesized attribute $\mathit{display}$ of type string. For each node, the display strings of the children are concatenated, possibly interspersed with local strings, and synthesized as the display string of the parent. For a fancy output, *format strings* may be inserted in the display string. Format strings include font switching or layout commands. Note that the display string is computed just as any attribute. In other words, it is also incrementally updated. The Synthesizer Generator uses this technique.

6.4.2 Display tree editing

Template editing is not yet powerful and flexible enough to our taste. We advocate another model that generalizes template editing. It allows for arbitrary tree transformations, even at places other than the current selection.

We construct an object that contains the unparsing as well as tree transformations. This object is called a display tree. The unparsing defined by a display trees describe a two dimensional layout much like the BOX grammar. The transformations defined by a display tree must be selected and activated by the user. When activated, a template is applied to some subtree of the underlying abstract syntax tree. Which subtree that is, is also specified by the display tree.

For example, assume that an integer is assigned to an undeclared variable. A display tree may include a transformation that inserts a declaration of type integer for that variable. Observe that the actual transformation (adding a declaration) is not done at the current selection (erroneous assignment). This is why such transformations are sometimes referred to as *remote editing*.

Display trees

Display trees have a BOX grammar like definition. The basic box is a txt node, containing a string. The hor and ver productions create a box by concatenation respectively stacking two boxes. The sel production associates a menu of transformations to a part of the unparsing.

We could also add a font field to txt.

$$\begin{aligned} disp &= \mathbf{txt}(str) \\ &\mid \mathbf{hor}(disp, disp) \\ &\mid \mathbf{ver}(disp, disp) \\ &\mid \mathbf{sel}(disp, menu) \end{aligned}$$

Display trees are more general than display strings. With the *txt* and *hor* productions, we can emulate display strings. The *ver* and *sel* productions are redundant when emulating display strings.

We require that for each non-terminal X the set $A_{syn}(X)$ is extended with an attribute *out* of type *disp*. By making unparsing explicit, we implicitly allow the description of non-trivial data flow. The computation of the display string in the Synthesizer Generator is strictly bottom-up and strictly child order preserving. We advocate reordering the display trees of the children, massaging display trees via semantic function and passing parts of display trees down via inherited attributes as well. The display tree synthesized by an abstract syntax tree, needs not to bear any resemblance with the underlying abstract syntax tree.

This flexibility is a disadvantage as well. A display tree determines what is shown to the user. In general, a subtree (subbox) selection in the display tree can not straightforwardly be associated with a subtree of the abstract syntax tree. That is why ordinary template editing can not be combined with display trees.

Editing

The *sel* production associates an edit menu of tree transformations with a part of the display tree. When the user selects that part, the menu is shown. Each menu entry consists of a template and a node in the abstract syntax tree to which the template will be applied when the user selects that menu entry.

$$\begin{aligned} menu &= \mathbf{done}() \\ &\mid \mathbf{choice}(template, node, menu) \end{aligned}$$

A *node* uniquely specifies a subtree of the abstract syntax tree. For example, when trees are not shared, a pointer to a node uniquely characterizes that node. When trees are shared, we could take integer lists and use a path denotation as in this thesis. For each non-terminal X the set $A(X)$ of associated attributes should then be extended with an inherited attribute *path* of type $[int]$ (where $[int] = node$).

Example

We will now define the attribute equations for the *stat* production in the VARUSE grammar that compute an elaborate display tree. We assume that shared abstract syntax trees are uniquely identified with *path* attributes. The following fragment shows how the *path*s are inherited, and the display tree *out* is synthesized.

$$\begin{array}{l} L_1 \;=\; \mathbf{stat}(N, L_2) \\ \qquad \begin{array}{ll} N.path & := 1 : L_1.path; \\ L_2.path & := 2 : L_1.path; \\ L_1.out & := f_{\texttt{use}}\; N.out \oplus L_2.out \end{array} \end{array}$$

path is a normal attribute. We could store the path to a declaring occurrence of a variable in the symbol table record for that variable. This allows for remote editing if an applied occurrence of that variable causes an error.

The semantic function f_s constructs a display tree from a display tree by prefixing it with the string 's'. The semantic function b_s suffixes its argument with string 's'. It is used in the definition of the semantic function $\oplus$ that joins two display trees and places the string ';' in between.

$$\begin{array}{ll} \mathbf{func}\; f_s\; d & = \mathbf{hor}(\mathbf{txt}('s'), d) \\ \mathbf{func}\; b_s\; d & = \mathbf{hor}(d, \mathbf{txt}('s')) \\ \mathbf{func}\; d_1 \oplus d_2 & = \mathbf{ver}(b_{;}\; d_1, d_2) \end{array}$$

A more elaborate display tree is constructed in the following fragment. When an identifier is not in the symbol table, the warning "error" shows up, right after the violating using occurrence. Furthermore some menu m, which is still to be defined, will be made available.

$$\begin{array}{l} L_1 \;=\; \mathbf{stat}(N, L_2) \\ \qquad \begin{array}{ll} \multicolumn{2}{l}{\mathbf{local}\; d_{ok}, d_{err} : disp;} \\ d_{ok} & := f_{\texttt{use}}\; N.out \oplus L_2.out; \\ d_{err} & := f_{\texttt{use}}\; (b_{\texttt{error}}\; N.out) \oplus L_2.out; \\ \multicolumn{2}{l}{\mathbf{local}\; m : menu;} \\ m & := \ldots; \\ L_1.out & := \mathbf{if}\; lookup\; L.env\; N.id = \bot\; \mathbf{then}\; \mathbf{sel}(d_{err}, m)\; \mathbf{else}\; d_{ok}\; \mathbf{fi} \end{array} \end{array}$$

When an identifier is not declared, we can either insert its declaration or we can delete the identifier. For deletion, template *Delete* is applied to the node denoted by $N.path$. We define two possible places to insert a declaration. Just above the using occurrence (template *Insert*) or at the beginning of the program (template *InsTop*). Note that *InsTop* is applied to the root ([]) of the abstract syntax tree.

Recall the syntax of a template: $l : t \Longrightarrow t'$. Pattern t contains free variables that are bound by unifying t with the subtree denoted by the *path*.

$$\begin{array}{lllll} m := & \mathbf{choice}(Delete : & \mathbf{name}(n) & \Longrightarrow \mathbf{Ngap}(), & N.path, \\ & \mathbf{choice}(Insert : & l & \Longrightarrow \mathbf{decl}(\mathbf{name}(N.id), l), & L_1.path, \\ & \mathbf{choice}(InsTop : & \mathbf{root}(l) & \Longrightarrow \mathbf{decl}(\mathbf{name}(N.id), l), & [\,], \\ & \mathbf{done}())))) & & & \end{array}$$

Observations

This section only presented preliminary thoughts on how to model a state-of-the-art user interface. We will conclude it with some observations.

- A *node* in a menu denotes a subtree in the abstract syntax tree. Could we perform some kind of static analysis to determine whether the template for that *node* has the same type as the specified subtree?
- Observe that attribute *path* is an *inherited* attribute that uniquely addresses a node. As such, it cripples memoization of the visit-functions. It seems wise to add an additional visit $(path, out)$ to each interface, as a last visit, so that incremental evaluation of "normal" visits is not influenced.
- Higher-order attributes are *computed* during tree decoration. It does not make sense to allow edit actions that change such attributes.
- It would be nice if we had "templates" that would allow for tree directed movements. Suppose that the user has selected an applied occurrence. It seems an interesting possibility if a button appeared that, when clicked, moved the current selection to the defining occurrence.
- Templates are in fact functions. When a menu entry is selected, the subtree specified by *node* is traced and the template function is applied to it. We could image other functions, most notably *interactive functions*, for example to enter primitive values.
- Other primitives are imaginable for display trees besides *txt* leafs. We could image icons or graphs or interactive widgets like buttons, dials, scroll bars or tick boxes....

6.5 Open problems

We conclude this chapter with an overview of the problems encountered in the course of writing the LRC processor. Some of the problems are relatively easy to solve, they merely require some long hours of programming. Other problems still require a lot of research respectively engineering.

In the current system the split grammar is computed after the bind grammar. As explained in 4.2.3, wrappers should contain split attributable attributes. It would have been better if phases 2 and 3 were swapped.

The algorithm in the LRC processor that removes diverging and unconnected computations (dead-ends) performs a *local* test: the synthesized attributes of the parent are regarded as useful, and any attribute not contributing to any of them are discarded. However, *global* analysis should be added to check which of the synthesized attributes of the parent eventually contributes to the synthesized attributes of the start symbol.

Another form of extensive analysis deals with *conditional dead-ends*. In Figure 3.14◁55 the visit-sequence for *add* of the higher-order variant of the VARUSE grammar is given. Let us repeat (and expand) that fragment.

```
plan $add$
begin
  $eval(E_2.id)$;
  $visit(E_2, 1)$;
  $eval(E_1.addr)$
end
```

$$E_2.id := E_1.id;$$
$$E_2.addr := visit^1_E\ E_2\ E_2.id;$$
$$E_1.addr := \textbf{if}\ str = E_1.id\ \textbf{then 1 else}\ 1 + E_2.addr\ \textbf{fi}$$

We see that if $str = E_1.id$, the first two instructions are dead. In other words, if $str = E_1.id$ holds, we may ignore the $visit$ instruction skipping the decoration of the arbitrarily large subtree rooted at E_2. It would be interesting if the generator could optimize the evaluator for such conditional dead-ends. This would require branching visit-sequences.

We have changed Kastens' step 4 and 5 since we felt that Kastens' algorithm is biased towards standard evaluation. However, we think that the computation of the interfaces is also biased. Kastens' algorithm maximizes the partitions so that the number of visits is minimized, since this saves time-consuming context switches. However, in our functional setting, it might be better to have more, smaller visit-functions in order to obtain better caching results.

We have not yet investigated the consequences of not caching every constructor or every visit-function. It may be the case that a lot of cache entries can be spared, leading to faster cache lookup and thus more efficient evaluation, without causing too many misses.

Vogt has done some research on cache purging strategies in his implementation of a functional evaluator in GOFER [Vog93]. We have not yet considered garbage collection for the evaluators generated by the LRC processor.

Although the front-end of the LRC processor computes the split grammar, the back-end does not yet generated target code for that. The optimizations discussed in Chapter 4 are not yet incorporated in LRC either. We did some preliminary experiments with display trees, but a lot of work remains to be done.

Incremental evaluation can be enhanced greatly by writing "smarter" grammar specifications. For example, a block of statements usually inherits a symbol table. One small change in that symbol tables requires the recompilation of that block, even if the change is confined to a symbol that is not used in the entire block. If a block synthesizes a list of *used* variables, the inherited symbol table could be *projected* on that list yielding greater incremental efficiency. However, we feel that the grammar writer should not be bothered by such implementation considerations. The generator should analyze the grammar and apply such optimizations automatically.

7

Chapter 7

Final remarks

In this final chapter we draw some conclusions about the work presented in this thesis and we discuss how to extend our work. We start with a review.

7.1 Review

This thesis discussed the construction of a generator for incremental attribute evaluator. Chapter 5 described how to compute interfaces for non-terminals and how chained scheduling produces plans well suited for incremental evaluation based on memoized visit-functions.

Chapter 3 showed how visit-sequences are mapped to pure visit-functions. Intra-visit-dependencies are resolved with bindings. Bindings accurately indicate when a visits may be skipped during incremental evaluation. In other words visit-functions can be memoized to obtain incremental evaluation.

The construction of the LRC processor is the proof of the feasibility of our functional approach. The first performance results presented in Section 6.3 are encouraging, but we do not know how evaluators generated by the LRC processor compare with evaluators from other systems.

Appendices A, B and C present a sample run of the LRC processor.

Chapter 4 deals with several optimizations aimed at improving the incremental behavior of the visit-functions. Splitting appears to improve incremental behavior considerably. Encapsulators allow for many interesting tree modifications at construction time. Especially elimination seems promising.

7.2 Completing the generator

The LRC processor is not yet of production quality. It consists of many tangled tools and has many unlisted options and parameters. The implementation of the fron-end is well documented, but there is no user manual. A major flaw is the lack of an integrated interactive editor. This is the single most important feature missing in LRC. Other features that would enhance LRC are listed in Section 6.5.

The advantages of LRC are its relatively small size and the elegance of the generated evaluators. Furthermore, LRC uses SSL as input language. This language is well documented and many grammars have been written in it. The LRC processor is written in PASCAL, a fairly abstract language. With a PASCAL to C translator it becomes easily portable. Finally, the evaluators generated by LRC appear to be bug-free. Large parts of LRC are generated by LRC itself.

7.3 Parallel attribute evaluation

In this thesis we studied sequential attribute evaluation. How to use parallelism to speed up incremental evaluation is still an open problem. Pure functions seem amenable to parallel computation. However, memoization of visit-functions and term sharing rely on large global data structures which are not straightforwardly implemented on a distributed memory machine. Shared memory machines on the other hand do not scale well. It is therefore not clear how memoized visit-functions could be parallelized.

7.4 Conclusions

In this thesis we have discussed how to construct a generator for incremental attribute evaluators. We have presented ways to improve incremental behavior. It is also shown that incremental evaluation with memoized visit-functions is a good approach. Generated editors have an simple, elegant and robust architecture. Decoration speed of 50k lines per minut is of production quality.

The advent of caching in attribute grammar evaluation as well as in other areas of incremental computation might prove to be as useful as virtual memory is today.

We are convinced that this thesis also proves that the plan and production icons, plan trees and other pictures are useful to convey ideas.

Incremental attribute evaluation is a new technique for obtaining incremental implementations of programs. We hope that with the continuing development of incremental techniques editors will be generated that make the task of writing programs even more enjoyable.

Appendix A

Ssl source

A.1 Comments

This appendix contains a specification of the VARUSE grammar. As specification language we have chosen SSL [RT88], the source language of the Synthesizer Generator and the LRC processor.

This source is different on two accounts compared with the grammar specification in Figure 2.1$_{\triangleleft 21}$. First of all, SSL lacks a list primitive. Lists must be explicitly coded as was done with the non-terminals `ENV` and `CODE`. Secondly, SSL lacks a polymorphic $\perp$ value. The *lookup* function returned this value when the identifier was not contained in the environment. In order not to complicate the example, we have used a large integer (9999) instead.

Note that L_1 is denoted with `L$1`. Since `root` is a reserved word specifying the start symbol of the grammar, the *root* production on S has been renamed to `Root` with an uppercase `R`.

A.2 Listing

```
/********************************************\
**                                          **
** "varuse.ssl"                             **
** Demonstration grammar                    **
**                                          **
\********************************************/

root S;

/* Abstract syntax **************************/

S : Root(L)
  ;
L : empty()
  | decl(N L)
  | stat(N L)
  ;
N : name(STR)
  ;

ENV  : emptyenv()
     | consenv(STR ENV)
     ;
CODE : emptycode()
     | conscode(INT CODE)
     ;

/* Attributes *******************************/

S { synthesized CODE code;
  };
L { synthesized ENV  decs;
    inherited   ENV  env;
    synthesized CODE code;
  };
N { synthesized STR  id;
  };
```

```
/* Equations ********************************/

S : Root {
      L.env    = L.decs;
      S.code   = L.code;
    }
  ;
L : decl {
      L$1.decs = consenv(N.id,L$2.decs);
      L$2.env  = L$1.env;
      L$1.code = conscode(
                   -lookup(L$1.env,N.id),
                   L$2.code);
    }
  | stat {
      L$1.decs = L$2.decs;
      L$2.env  = L$1.env;
      L$1.code = conscode(
                   +lookup(L$1.env,N.id),
                   L$2.code);
    }
  | empty {
      L$1.decs = emptyenv();
      L$1.code = emptycode();
    }
  ;
N : name {
      N.id     = STRtoupper(STR);
    }
  ;

/* Semantic functions ***********************/

INT lookup(ENV env,STR id){
  with(env)(emptyenv()   : 9999
           ,consenv(s,e) : ( s==id
                           ? 1
                           : 1+lookup(e,id)
                           )
           )
};
```

Appendix B

Generated C program

B.1 Comments

When the SSL source of the VARUSE grammar, as given in Appendix A is passed through the LRC processor, the C program listed next is generated.

The major concepts of the generated C target are discussed in Chapter 6. A minor difference is that all external names declared by the library are prefixed with LRC_ in either upper or lower case.

Equations have not been discussed in Chapter 6. There is one interesting point to make. Recall that attributes are shared. The arguments and result of the functions implementing the equations are therefore *terms*. However, the semantic functions, like the user defined *lookup* or the library function *upstring*, are defined on the underlying *primitive values*. As a consequence, the functions implementing the equations cast values from and to terms.

The main program refers to the library function `lrc_term_print_long` that prints a term. Of course, the names of the productions are grammar dependent; they must be installed. The code under the heading `/* shalloc tables */` defines the name tables, and the library routine `lrc_install_defaults` installs them.

Likewise, the library routine `lrc_cell_print_fipop` prints entries from the visit-function cache. This routine prints the *f*unction, the *i*nput *p*acket and the *o*utput *p*acket stored in the cell. The required tables are defined under the heading `/* memo tables */`. These tables are also installed by `lrc_install_defaults`.

The main program makes two runs. After the first run, the entire visit-function cache is dumped with the library function `lrc_memo_dump`. It needs two parameters, namely a filter function and a print function. For the latter, we use the above mentioned `lrc_cell_print_fipop`.

The function `lrc_memo_dump(f,p)` calls `p(c)` for every cell for which `f(c)` holds. Furthermore, the cells birth-stamp must exceed a certain threshold. Each newly created cell is time-stamped upon which the "clock" is stepped. When a cell hits, it is restamped without stepping the clock. The current "time" can be obtained with `lrc_memo_time()`. Initially, the threshold is set to 0, the start-up

Every memo cell must thus pass 2 filters: f and the birth threshold.

value of the clock. For the second run, the threshold is set to the time just after evaluating the first tree. This ensures that the second dump only shows the new cache entries.

B.2 Listing

```
/********************************************\
**                                          **
** "varuse.c"                               **
** Generated by the LRC processor from      **
** source "varuse.ssl".                     **
** main() handwritten.                      **
**                                          **
\********************************************/

#define LRC_TERM_FAILURE(s) printf("Fatal")

#include <stdio.h>
#include <lrclib.h>

/* tags *************************************/

typedef enum  {
  T_Root=9,
  T_empty=10,
  T_decl=11,
  T_stat=12,
  T_name=13,
  T_emptyenv=14,
  T_consenv=15,
  T_emptycode=16,
  T_conscode=17} production_tag_tp;

/* constructors *****************************/

lrc_term  C_Root(lrc_term s1) {
  lrc_term res[2];
  res[0]=(lrc_term)T_Root;
  res[1]=s1;
  return LRC_SHALLOC(res);
}

lrc_term  C_empty() {
  lrc_term res[1];
  res[0]=(lrc_term)T_empty;
  return LRC_SHALLOC(res);
}

lrc_term  C_decl(lrc_term s1, lrc_term s2) {
  lrc_term res[3];
  res[0]=(lrc_term)T_decl;
  res[1]=s1;
  res[2]=s2;
  return LRC_SHALLOC(res);
}

lrc_term  C_stat(lrc_term s1, lrc_term s2) {
  lrc_term res[3];
  res[0]=(lrc_term)T_stat;
  res[1]=s1;
  res[2]=s2;
  return LRC_SHALLOC(res);
}

lrc_term  C_name(lrc_string s1) {
  lrc_term res[2];
  res[0]=(lrc_term)T_name;
  res[1]=C_Str(s1);
  return LRC_SHALLOC(res);
}

lrc_term  C_emptyenv() {
  lrc_term res[1];
  res[0]=(lrc_term)T_emptyenv;
  return LRC_SHALLOC(res);
}

lrc_term C_consenv(lrc_string s1,lrc_term s2){
  lrc_term res[3];
  res[0]=(lrc_term)T_consenv;
  res[1]=C_Str(s1);
  res[2]=s2;
  return LRC_SHALLOC(res);
}

lrc_term  C_emptycode() {
  lrc_term res[1];
  res[0]=(lrc_term)T_emptycode;
  return LRC_SHALLOC(res);
}

lrc_term  C_conscode(int s1, lrc_term s2) {
  lrc_term res[3];
  res[0]=(lrc_term)T_conscode;
  res[1]=C_Int(s1);
  res[2]=s2;
  return LRC_SHALLOC(res);
}
```

```
/* wrapper tags ****************************/

typedef enum {
  W_empty_1_2_tag=18,
  W_decl_1_2_tag=19,
  W_stat_1_2_tag=20
} wrapper_tag_tp;

/* wrapper types ***************************/

typedef struct {
  wrapper_tag_tp tag;
} W_empty_1_2_tp;
#define W_empty_1_2_CAST(t) \
  (*(W_empty_1_2_tp*)(t))

typedef struct {
  wrapper_tag_tp tag;
  lrc_term A0_decl_N_1_id;
  lrc_term P_L_2_1_2;
} W_decl_1_2_tp;
#define W_decl_1_2_CAST(t) \
  (*(W_decl_1_2_tp*)(t))

typedef struct {
  wrapper_tag_tp tag;
  lrc_term P_L_2_1_2;
} W_stat_1_2_tp;
#define W_stat_1_2_CAST(t) \
  (*(W_stat_1_2_tp*)(t))

/* wrapper constructors ********************/

lrc_term CW_empty_1_2() {
  W_empty_1_2_tp res;
  res.tag=W_empty_1_2_tag;
  return LRC_SHALLOC(res);
}

lrc_term CW_decl_1_2(lrc_term a0,lrc_term b0){
  W_decl_1_2_tp res;
  res.tag=W_decl_1_2_tag;
  res.A0_decl_N_1_id=a0;
  res.P_L_2_1_2=b0;
  return LRC_SHALLOC(res);
}

lrc_term CW_stat_1_2(lrc_term b0) {
  W_stat_1_2_tp res;
  res.tag=W_stat_1_2_tag;
  res.P_L_2_1_2=b0;
  return LRC_SHALLOC(res);
}
```

```
/* semantic functions **********************/

int lookup(lrc_term p1,lrc_string p2) {
  int cell_1;
  int cell_2;

  if ((LRC_TAG(p1)==T_emptyenv)) {
    cell_2 = 9999;
  } else {
    if (lrc_string_equal(
        LRC_VALUE(LRC_CHILD(p1,1),lrc_string),
        p2))
    {
      cell_1 = 1;
    } else {
      cell_1 = 1+(lookup(LRC_CHILD(p1,2),p2));
    }
    cell_2 = cell_1;
  }
  return cell_2;
}

/* equations *******************************/

lrc_term eq_9(){
  return C_emptycode();
}

lrc_term eq_8(){
  return C_emptyenv();
}

lrc_term eq_4(lrc_term p0, lrc_term p1,
           lrc_term p2){
  return C_conscode(
    -lookup(p2, LRC_VALUE(p1,lrc_string)),p0);
}

lrc_term eq_2(lrc_term p0, lrc_term p1){
  return C_consenv(
    LRC_VALUE(p1,lrc_string), p0);
}

lrc_term eq_7(lrc_term p0, lrc_term p1,
           lrc_term p2){
  return C_conscode(
    lookup(p2, LRC_VALUE(p1,lrc_string)),p0);
}

lrc_term eq_10(lrc_term p0){
  return C_Str(
    STRtoupper(LRC_VALUE(p0,lrc_string)));
}
```

```
/* packets **********************************/

typedef struct {
  lrc_term tree;
} I_S_1_tp;

typedef struct {
  lrc_term code;
} O_S_1_tp;

typedef struct {
  lrc_term tree;
} I_L_1_tp;

typedef struct {
  lrc_term P_L_1_2;
  lrc_term decs;
} O_L_1_tp;

typedef struct {
  lrc_term tree;
  lrc_term P_L_1_2;
  lrc_term env;
} I_L_2_tp;

typedef struct {
  lrc_term code;
} O_L_2_tp;

typedef struct {
  lrc_term tree;
} I_N_1_tp;

typedef struct {
  lrc_term id;
} O_N_1_tp;



/* VNT forward ******************************/

void VNT_1_S(I_S_1_tp *ip, O_S_1_tp *op);
void VNT_1_L(I_L_1_tp *ip, O_L_1_tp *op);
void VNT_2_L(I_L_2_tp *ip, O_L_2_tp *op);
void VNT_1_N(I_N_1_tp *ip, O_N_1_tp *op);
```

```
/* VPR functions ****************************/

void VPR_1_Root(lrc_packet ip,lrc_packet op){
#define IP (*(I_S_1_tp*)ip)
#define OP (*(O_S_1_tp*)op)
  I_L_1_tp I_L_1_1;
  O_L_1_tp O_L_1_1;
  I_L_2_tp I_L_1_2;
  O_L_2_tp O_L_1_2;

  I_L_1_1.tree=LRC_CHILD(IP.tree,1);
  VNT_1_L(&I_L_1_1,&O_L_1_1);

  I_L_1_2.env= O_L_1_1.decs;

  I_L_1_2.tree=LRC_CHILD(IP.tree,1);
  I_L_1_2.P_L_1_2=O_L_1_1.P_L_1_2;
  VNT_2_L(&I_L_1_2,&O_L_1_2);

  OP.code= O_L_1_2.code;

  return;
#undef IP
#undef OP
}

void VPR_1_empty(lrc_packet ip,lrc_packet op){
#define IP (*(I_L_1_tp*)ip)
#define OP (*(O_L_1_tp*)op)

  OP.decs=eq_8();

  OP.P_L_1_2=CW_empty_1_2();
  return;
#undef IP
#undef OP
}

void VPR_2_empty(lrc_packet ip,lrc_packet op){
#define IP (*(I_L_2_tp*)ip)
#define OP (*(O_L_2_tp*)op)

  OP.code=eq_9();

  return;
#undef IP
#undef OP
}
```

```
void VPR_1_decl(lrc_packet ip,lrc_packet op){
#define IP (*(I_L_1_tp*)ip)
#define OP (*(O_L_1_tp*)op)
  I_L_1_tp I_L_2_1;
  O_L_1_tp O_L_2_1;
  I_N_1_tp I_N_1_1;
  O_N_1_tp O_N_1_1;

  I_N_1_1.tree=LRC_CHILD(IP.tree,1);
  VNT_1_N(&I_N_1_1,&O_N_1_1);

  I_L_2_1.tree=LRC_CHILD(IP.tree,2);
  VNT_1_L(&I_L_2_1,&O_L_2_1);

  OP.decs=eq_2( O_L_2_1.decs, O_N_1_1.id);

  OP.P_L_1_2=CW_decl_1_2
    (O_N_1_1.id,O_L_2_1.P_L_1_2);
  return;
#undef IP
#undef OP
}

void VPR_2_decl(lrc_packet ip,lrc_packet op){
#define IP (*(I_L_2_tp*)ip)
#define OP (*(O_L_2_tp*)op)
  I_L_2_tp I_L_2_2;
  O_L_2_tp O_L_2_2;

  I_L_2_2.env= IP.env;

  I_L_2_2.tree=LRC_CHILD(IP.tree,2);
  I_L_2_2.P_L_1_2=W_decl_1_2_CAST
    (IP.P_L_1_2).P_L_2_1_2;
  VNT_2_L(&I_L_2_2,&O_L_2_2);

  OP.code=eq_4(O_L_2_2.code,
    W_decl_1_2_CAST(IP.P_L_1_2).AO_decl_N_1_id
    ,IP.env);

  return;
#undef IP
#undef OP
}

void VPR_1_stat(lrc_packet ip,lrc_packet op){
#define IP (*(I_L_1_tp*)ip)
#define OP (*(O_L_1_tp*)op)
  I_L_1_tp I_L_2_1;
  O_L_1_tp O_L_2_1;

  I_L_2_1.tree=LRC_CHILD(IP.tree,2);
  VNT_1_L(&I_L_2_1,&O_L_2_1);

  OP.decs= O_L_2_1.decs;

  OP.P_L_1_2=CW_stat_1_2(
    O_L_2_1.P_L_1_2);
  return;
#undef IP
#undef OP
}

void VPR_2_stat(lrc_packet ip,lrc_packet op){
#define IP (*(I_L_2_tp*)ip)
#define OP (*(O_L_2_tp*)op)
  I_L_2_tp I_L_2_2;
  O_L_2_tp O_L_2_2;
  I_N_1_tp I_N_1_1;
  O_N_1_tp O_N_1_1;

  I_L_2_2.env= IP.env;

  I_L_2_2.tree=LRC_CHILD(IP.tree,2);
  I_L_2_2.P_L_1_2=W_stat_1_2_CAST
    (IP.P_L_1_2).P_L_2_1_2;
  VNT_2_L(&I_L_2_2,&O_L_2_2);

  I_N_1_1.tree=LRC_CHILD(IP.tree,1);
  VNT_1_N(&I_N_1_1,&O_N_1_1);

  OP.code=eq_7(O_L_2_2.code,
    O_N_1_1.id,IP.env);

  return;
#undef IP
#undef OP
}

void VPR_1_name(lrc_packet ip,lrc_packet op){
#define IP (*(I_N_1_tp*)ip)
#define OP (*(O_N_1_tp*)op)

  OP.id=eq_10( LRC_CHILD(IP.tree,1));

  return;
#undef IP
#undef OP
}
```

```
/* VNT functions ****************************/

int VNT_count;

void VNT_1_S(I_S_1_tp *ip, O_S_1_tp *op) {
  VNT_count++;
  switch (LRC_TAG(ip->tree)) {
    case T_Root :
      LRC_MEMOP(VPR_1_Root,ip,op); return;
    default     :
      LRC_TERM_FAILURE("VNT_1_S");
  }
}

void VNT_1_L(I_L_1_tp *ip, O_L_1_tp *op) {
  VNT_count++;
  switch (LRC_TAG(ip->tree)) {
    case T_empty :
      LRC_MEMOP(VPR_1_empty,ip,op); return;
    case T_decl  :
      LRC_MEMOP(VPR_1_decl,ip,op); return;
    case T_stat  :
      LRC_MEMOP(VPR_1_stat,ip,op); return;
    default :
      LRC_TERM_FAILURE("VNT_1_L");
  }
}

void VNT_2_L(I_L_2_tp *ip, O_L_2_tp *op) {
  VNT_count++;
  switch (LRC_TAG(ip->tree)) {
    case T_empty :
      LRC_MEMOP(VPR_2_empty,ip,op); return;
    case T_decl  :
      LRC_MEMOP(VPR_2_decl,ip,op); return;
    case T_stat  :
      LRC_MEMOP(VPR_2_stat,ip,op); return;
    default      :
      LRC_TERM_FAILURE("VNT_2_L");
  }
}

void VNT_1_N(I_N_1_tp *ip, O_N_1_tp *op) {
  VNT_count++;
  switch (LRC_TAG(ip->tree)) {
    case T_name :
      LRC_MEMOP(VPR_1_name,ip,op); return;
    default     :
      LRC_TERM_FAILURE("VNT_1_N");
  }
}
```

```
/* shalloc tables ***************************/


char * lrc_tag_names_table[21] = {
  "_Int","_Bool","_Char","_Real","_Str",
  "_Tok","_Ptr","_Map","_Attr","Root",
  "empty","decl","stat","name","emptyenv",
  "consenv","emptycode","conscode",
  "empty_1_2","decl_1_2","stat_1_2"
};

int lrc_num_sons_table[21] = {
 0, 0, 0, 0, 0,
 0, 0, 0, 0, 1,
 0, 2, 2, 1, 0,
 2, 0, 2,
 0, 2, 1
};



/* memo tables *****************************/


int lrc_vpr_index(lrc_ptr f) {
  if (f==(lrc_ptr)&VPR_1_Root) return 0;
  if (f==(lrc_ptr)&VPR_1_empty) return 1;
  if (f==(lrc_ptr)&VPR_2_empty) return 2;
  if (f==(lrc_ptr)&VPR_1_decl) return 3;
  if (f==(lrc_ptr)&VPR_2_decl) return 4;
  if (f==(lrc_ptr)&VPR_1_stat) return 5;
  if (f==(lrc_ptr)&VPR_2_stat) return 6;
  if (f==(lrc_ptr)&VPR_1_name) return 7;
  exit(1);
}

char * lrc_vpr_name_table[8] = {
 "Root\\1", "empty\\1", "empty\\2", "decl\\1",
 "decl\\2", "stat\\1", "stat\\2", "name\\1"
};

int lrc_vpr_num_ip_table[8] = {
 0, 0, 2, 0, 2, 0, 2, 0
};

int lrc_vpr_num_op_table[8] = {
 1, 2, 1, 2, 1, 2, 1, 1
};
```

```
/* main *************************************/

int main() {
  lrc_term t;
  I_S_1_tp ip;
  O_S_1_tp op;

  /* init library */
  lrc_lib_init();
  /* installs shalloc and memo tables */
  lrc_install_default;

  printf("----------------------------");
  printf("----------------------------\n");
  printf("Compile ");
  printf("'(use x; var x; use y;)'");
  printf("from scratch.\n\n");
  VNT_count=0;
  t=C_Root(C_stat(C_name("x")
          ,C_decl(C_name("x")
          ,C_stat(C_name("y")
          ,C_empty()))));
  ip.tree=t;
  VNT_1_S(&ip,&op);
  lrc_term_print_long(op.code);
  printf("\nNum of visits=%d\n\n",VNT_count);

  lrc_memo_dump(lrc_filter_true,
    lrc_cell_print_fipop);
  lrc_memo_dump_set_birth_time(
    lrc_memo_time());


  printf("----------------------------");
  printf("----------------------------\n");
  printf("Compile ");
  printf("'(use x; use x; var x; use y;)'");
  printf("incrementally.\n\n");
  VNT_count=0;
  t=C_Root(C_stat(C_name("x")
          ,C_stat(C_name("x")
          ,C_decl(C_name("x")
          ,C_stat(C_name("y")
          ,C_empty())))));
  ip.tree=t;
  VNT_1_S(&ip,&op);
  lrc_term_print_long(op.code);
  printf("\nNum of visits=%d\n\n",VNT_count);

  lrc_memo_dump(lrc_filter_true,
    lrc_cell_print_fipop);


  printf("----------------------------");
  printf("----------------------------\n");
  return 0;
}
```

Appendix C

Evaluator output

C.1 Comments

This section contains the output of the C program listed in Appendix B, which implements an incremental evaluator for the VARUSE grammar.

The output consists of two parts. The first part corresponds with the first run, an evaluation from scratch for the following program.

```
(use x; var x; use y;)
```

The second part corresponds with the second run, an incremental evaluation of a changed program.

```
(use x; use x; var x; use y;)
```

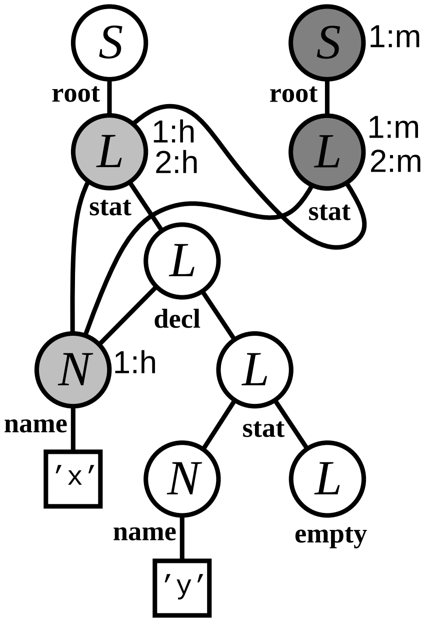
The output in both cases is formatted identically. The first four lines identify the source program, the computed translation and the number of visits required. Then follows a dump of the visit-function cache.

A dump consists of a header showing the birth-stamp threshold (0 respectively 11) and a list of buckets. The library uses a hash table with about 10000 buckets, only the non-empty ones are dumped. A bucket, consists of a list of entries followed by a banner denoted with `BUCKET`. The banner shows the sequence number of the bucket, the number of entries listed, the number of entries that are hidden (due to filtering as described in Appendix B) and the total number of entries.

An entries of a bucket is prefixed with `MEMO`. The first number following `MEMO` is the number of hits on that entry. The next number gives the birth-stamp. Then follow the visit-function that causes the entry, together with the arguments and results.

Observe that in both runs each bucket only contains one entry; the hash functions performed well.

During the first run, one visit hits, namely $visit_N^1\ \mathbf{name}(\text{'x'})$. In Figure C.1▷ we have sketched the contents of the constructor cache for the second run. The nodes that are visited are colored grey and annotated with strings like `1:m`. The `m` stands for miss, an `h` stands for hit. The number `1` refers to the visit. This information may be deduced from the dump: each annotation corresponds with one `MEMO` entry, 6 in total.



The white and light grey nodes belong to the abstract syntax tree of the first run. The dark grey nodes are the additional nodes allocated for the second (incremental) run. The nodes that are *visited* during the second run are grey. Grey nodes are annotated with strings like `#:$`, where `#` denotes the visit number and `$` is either 'h' for hit or 'm' for miss.

Figure C.1. The shared terms during the second run

C.2 Listing

```
---------------------------------------------------------
Compile '(use x; var x; use y;)' from scratch.

conscode(1,conscode(-1,conscode(10000,emptycode())))
Num of visits=12

DUMP MEMO base= 0

  MEMO 0 10 Root\1
   (Root(stat(name("x"),decl(name("x"),stat(name("y"),empty()))))
   )=(conscode(1,conscode(-1,conscode(10000,emptycode()))))
BUCKET 4775  list= 1 hide= 0 TOTAL= 1
```

```
  MEMO 0 3 decl\1
   (decl(name("x"),stat(name("y"),empty()))
   )=(decl_1_2("x",stat_1_2(empty_1_2())),consenv("x",emptyenv()))
BUCKET 5311  list= 1 hide= 0 TOTAL= 1

  MEMO 0 2 stat\1
   (stat(name("y"),empty()))=(stat_1_2(empty_1_2()),emptyenv())
BUCKET 5943  list= 1 hide= 0 TOTAL= 1

  MEMO 0 4 stat\1
   (stat(name("x"),decl(name("x"),stat(name("y"),empty())))
   )=(stat_1_2(decl_1_2("x",stat_1_2(empty_1_2())))
   ,consenv("x",emptyenv()))
BUCKET 6023  list= 1 hide= 0 TOTAL= 1

  MEMO 1 9 name\1(name("x"))=("x")
BUCKET 6447  list= 1 hide= 0 TOTAL= 1

  MEMO 0 6 name\1(name("y"))=("y")
BUCKET 6527  list= 1 hide= 0 TOTAL= 1

  MEMO 0 5 empty\2
   (empty(),empty_1_2(),consenv("x",emptyenv()))=(emptycode())
BUCKET 6588  list= 1 hide= 0 TOTAL= 1

  MEMO 0 8 decl\2
   (decl(name("x"),stat(name("y"),empty()))
   ,decl_1_2("x",stat_1_2(empty_1_2()))
   ,consenv("x",emptyenv())
   )=(conscode(-1,conscode(10000,emptycode())))
BUCKET 6931  list= 1 hide= 0 TOTAL= 1

  MEMO 0 1 empty\1(empty())=(empty_1_2(),emptyenv())
BUCKET 6999  list= 1 hide= 0 TOTAL= 1

  MEMO 0 7 stat\2
   (stat(name("y"),empty()),stat_1_2(empty_1_2()),consenv("x",emptyenv())
   )=(conscode(10000,emptycode()))
BUCKET 8771  list= 1 hide= 0 TOTAL= 1

  MEMO 0 9 stat\2
   (stat(name("x"),decl(name("x"),stat(name("y"),empty())))
   ,stat_1_2(decl_1_2("x",stat_1_2(empty_1_2())))
   ,consenv("x",emptyenv())
   )=(conscode(1,conscode(-1,conscode(10000,emptycode()))))
BUCKET 9491  list= 1 hide= 0 TOTAL= 1


--------------------------------------------------------
Compile '(use x; use x; var x; use y;)' incrementally.

conscode(1,conscode(1,conscode(-1,conscode(10000,emptycode()))))
Num of visits=6
```

```
DUMP MEMO base= 11

  MEMO 0 12 stat\2
   (stat(name("x"),stat(name("x"),decl(name("x"),stat(name("y"),empty()))))
   ,stat_1_2(stat_1_2(decl_1_2("x",stat_1_2(empty_1_2()))))
   ,consenv("x",emptyenv())
   )=(conscode(1,conscode(1,conscode(-1,conscode(10000,emptycode())))))
BUCKET 3324  list= 1 hide= 0 TOTAL= 1

  MEMO 0 13 Root\1
   (Root(stat(name("x")
           ,stat(name("x"),decl(name("x"),stat(name("y"),empty()))))
           )
   )=(conscode(1,conscode(1,conscode(-1,conscode(10000,emptycode())))))
BUCKET 5495  list= 1 hide= 0 TOTAL= 1

  MEMO 1 11 stat\1
   (stat(name("x"),decl(name("x"),stat(name("y"),empty())))
   )=
   (stat_1_2(decl_1_2("x",stat_1_2(empty_1_2())))
   ,consenv("x",emptyenv())
   )
BUCKET 6023  list= 1 hide= 0 TOTAL= 1

  MEMO 2 12 name\1(name("x"))=("x")
BUCKET 6447  list= 1 hide= 0 TOTAL= 1

  MEMO 0 11 stat\1
   (stat(name("x"),stat(name("x"),decl(name("x"),stat(name("y"),empty()))))
   )=
   (stat_1_2(stat_1_2(decl_1_2("x",stat_1_2(empty_1_2()))))
   ,consenv("x",emptyenv())
   )
BUCKET 6743  list= 1 hide= 0 TOTAL= 1

  MEMO 1 12 stat\2
   (stat(name("x"),decl(name("x"),stat(name("y"),empty())))
   ,stat_1_2(decl_1_2("x",stat_1_2(empty_1_2())))
   ,consenv("x",emptyenv())
   )=(conscode(1,conscode(-1,conscode(10000,emptycode()))))
BUCKET 9491  list= 1 hide= 0 TOTAL= 1

--------------------------------------------------------
```

B

Bibliography

[Alb81] H. Alblas. A characterization of attribute evalution in passes. *Acta Informatica*, **16**:427–464, 1981.

[Alb89] Henk Alblas. Optimal incremental simple multi-pass attribute evaluation. *Information Processing Letters*, **32**:289–295, 1989.

[Alb91a] Henk Alblas. Attribute evaluation methods. In H. Alblas and B. Melichar, editors, *Proceedings of the International Summer School SAGA '91 on Attribute Grammars, Applications and Systems*, volume **545** of Lecture Notes in Computer Science, pages 48–113. Springer-Verlag, 1991.

[Alb91b] Henk Alblas. Incremental attribute evaluation. In H. Alblas and B. Melichar, editors, *Proceedings of the International Summer School SAGA '91 on Attribute Grammars, Applications and Systems*, volume **545** of Lecture Notes in Computer Science, pages 215–233. Springer-Verlag, 1991.

[Alb91c] Henk Alblas. Introduction to attribute grammars. In H. Alblas and B. Melichar, editors, *Proceedings of the International Summer School SAGA '91 on Attribute Grammars, Applications and Systems*, volume **545** of Lecture Notes in Computer Science, pages 1–15. Springer-Verlag, 1991.

[Aug93] Lex Augusteijn. *Functional Programming, Program Transformations and Compiler Construction*. PhD thesis, Technische Universiteit Eindhoven, Eindhoven, The Netherlands, October 1993.

[Bir84] R.S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, **21**:239–250, 1984.

[Boc76] Gregor V. Bochmann. Semantic evaluation from left to right. *Communications of the ACM*, **19**(2):55–62, 1976.

[BW88] Richard Bird and Philip Wadler. *Introduction to Functional Programming*. International series in Computer Science. Prentice Hall, 1988.

[CH79] R. COHEN AND E. HARRY. Automatic generation of near-optimal linear-time translators for non-circular attribute grammars. *Proceedings of the 6th ACM Symposium on Principles of Programming Languages*, pages 121–134, 1979.

[CLR92] THOMAS H. CORMEN, CHARLES E. LEISERSON, AND RONALD L. RIVEST. *Introduction to Algorithms*. McGraw-Hill Book Company, 1992.

[DJL84] PIERRE DERANSART, MARTIN JOURDAN, AND BERNARD LORHO. Speeding up circularity tests for attribute grammars. *Acta Informatica*, **21**:375–391, 1984.

[DJL88] PIERRE DERANSART, MARTIN JOURDAN, AND BERNARD LORHO. *Attribute Grammars. Definitions, Systems and Bibliography*, volume **323** of Lecture Notes in Computer Science. Springer-Verlag, 1988.

[DRT81] A. DEMERS, T. REPS, AND T. TEITELBAUM. Incremental evaluation for attribute grammars with applications to syntax-directed editors. In *Conference records of the eight ACM Symposium on Principles of Programming Languages*, pages 105–116, January 1981.

[EdJ90] JOOST ENGELFRIET AND WILLEM DE JONG. Attribute storage optimization by stacks. *Acta Informatica*, **27**:567–581, 1990.

[EF82] JOOST ENGELFRIET AND GILBERTO FILÈ. Simple multi-visit attribute grammars. *Journal of Computer and System Sciences*, **24**:283–314, 1982.

[Eng84] J. ENGELFRIET. Attribute grammars: Attribute evaluation methods. In B. Lorho, editor, *Methods and Tools For Compiler Construction*, pages 103–138. Cambridge Press, 1984.

[Far86] R. FARROW. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 85–98, 1986.

[GG84] HARALD GANZINGER AND ROBERT GIEGERICH. Attribute coupled grammars. *ACM SIGPLAN notices*, **19**(6):157–170, June 1984.

[HF92] PAUL HUDAK AND JOSEPH H. FASEL. A gentle introduction to haskell. *ACM SIGPLAN Notices, Haskell special issue*, **27**(5), may 1992.

[Hoo86] ROGER HOOVER. Dynamically bypassing copy rule chains in attribute grammars. In *Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 14–25. ACM, January 1986.

[HPJW+92] P. Hudak, S. Peyton-Jones, P. Wadler, et al. Report on the programming language haskell, a non-strict purely functional language (version 1.2). *ACM SIGPLAN Notices, Haskell special issue*, **27**(5), may 1992.

[Hug82] R.J.M. Hughes. Super combinators — a new implementation method for applicative languages. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 1–10, Pittsburgh, 1982.

[Hug85] John Hughes. Lazy memo-functions. In Jean-Pierre Jouannaud, editor, *Proceedings of the IFIP Conference on Functional Programming Languages and Computer Architecture*, volume **201** of Lecture Notes in Computer Science, pages 129–146. Springer-Verlag, September 1985.

[Jaz81] Mehdi Jazayeri. A simpler construction for showing the intrinsically exponential complexity of the circularity problem for attribute grammars. *Journal of the Association for Computing Machinery*, **28**(4):715–720, October 1981.

[Joh87] Thomas Johnnson. Attribute grammars as a functional programming paradigm. In G. Kahn, editor, *Functional programming languages and computer architecture*, volume **274** of Lecture Notes in Computer Science, pages 154–173, Portland, September 1987. Springer-Verlag.

[Jon91] Mark P. Jones. *Introduction to Gofer 2.20.* Oxford Programming Research Group, November 1991.

[Jou84] M. Jourdan. Strongly non-circular attribute grammars and their recursive evaluation. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 81–93, 1984.

[Jou91] Martin Jourdan. A survey of parallel attribute evaluation methods. In H. Alblas and B. Melichar, editors, *Proceedings of the International Summer School SAGA '91 on Attribute Grammars, Applications and Systems*, volume **545** of Lecture Notes in Computer Science, pages 234–253. Springer-Verlag, 1991.

[JP90] C. Julié and D. Parigot. Space optimization in the fnc-2 attribute grammar system. In *Attribute grammars and their applications*, volume **461** of Lecture Notes in Computer Science, pages 29–45. Springer-Verlag, 1990.

[Kas80] Uwe Kastens. Ordered attribute grammars. *Acta Informatica*, **13**:229–256, 1980.

[Kas87] Uwe Kastens. Lifetime analysis for attributes. *Acta Informatica*, **24**:633–652, 1987.

[Kas91] Uwe Kastens. Attribute grammars as a specification method. In H. Alblas and B. Melichar, editors, *Proceedings of the International Summer School SAGA '91 on Attribute Grammars, Applications and Systems*, volume **545** of Lecture Notes in Computer Science, pages 16–47. Springer-Verlag, 1991.

[Kav84] C. Kaviani. *Comparaison de deux méthode d'évaluation d'attributs sémantiques: GAG et FNC*. PhD thesis, Université de Paris VI, January 1984.

[KHZ82] Uwe Kastens, Brigitte Hutt, and Erich Zimmermann. *GAG: A practical compiler generator*, volume **141** of Lecture Notes in Computer Science. Springer-Verlag, 1982.

[Kle91] Eduard Klein. *Ein Modell zur Generierung paralleler Attributauswerter*. PhD thesis, Gesellschaft für Mathematik und Datenverarbeitung mbH., München, Germany, May 1991.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, **2**(2):127–145, 1968.

[Knu71] Donald E. Knuth. Semantics of context-free languages (correction). *Mathematical Systems Theory*, **5**(1):95–96, 1971.

[Knu91] Donald E. Knuth. *The TeXbook, computers & typesetting / A*. Addison-Wesley, Reading, Massachusets, 1991.

[KP81] B.W. Kernighan and P.J. Plauger. *Software tools in pascal*. Addison-Wesley, 1981.

[KR79] Ken Kennedy and Jayashree Ramanathan. A deterministic attribute grammar evaluator based on dynamic sequencing. *ACM Transactions on Programming Languages and Systems*, **1**(1):142–160, July 1979.

[KS87] M.F. Kuiper and S.D. Swierstra. Using attribute grammars to derive efficient functional programs. *Computing Science in the Netherlands CSN '87*, November 1987.

[Kui89] Matthijs F. Kuiper. *Parallel Attribute Evaluation*. PhD thesis, Utrecht University, Department of Computer Science, Utrecht, The Netherlands, November 1989.

[KW76] K. Kennedy and S. Warren. Automatic generation of efficient evaluators for attribute grammars. In *Conference Record of the Third ACM Symposium on Principles of Programming Languages*, pages 32–49, New York, 1976. ACM.

[odAS91] Rieks op den Akker and Erik Sluiman. Storage allocation for attribute evaluators using stacks and queues. In H. Alblas and B. Melichar, editors, *Proceedings of the International Summer School SAGA '91 on Attribute Grammars, Applications and Systems*, volume **545** of Lecture Notes in Computer Science, pages 140–150. Springer-Verlag, 1991.

[Pen93] MAARTEN PENNINGS. Multi-traversal tree decoration in a functional setting: monads versus bindings. Technical Report RUU-CS-93-46, Utrecht University, Department of Computer Science, PO Box 80.089, 3508 TB Utrecht, The Netherlands, December 1993.

[PSV92a] MAARTEN PENNINGS, DOAITSE SWIERSTRA, AND HARALD VOGT. Using cached functions and constructors for incremental attribute evaluation. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the 4th International Symposium PLILP '92 on Programming Implementation and Logic Programming*, volume **631** of Lecture Notes in Computer Science, pages 130–144. Springer-Verlag, 1992.

[PSV92b] MAARTEN PENNINGS, DOAITSE SWIERSTRA, AND HARALD VOGT. Using cached functions and constructors for incremental attribute evaluation. Technical Report RUU-CS-92-11, Utrecht University, Department of Computer Science, PO Box 80.089, 3508 TB Utrecht, The Netherlands, December 1992.

[Pug88] W.W. PUGH. *Incremental Computation and the Incremental Evaluation of Functional Programs*. PhD thesis, Cornell University, Department of Computer Science, Ithaca, New York, August 1988.

[Rep82] T. REPS. Optimal-time incremental semantic analysis for syntax-directed editors. In *Conference Record of the Ninth ACM Symposium on Principles of Programming Languages*, pages 169–176, New York, 1982. ACM.

[Rep93] THOMAS REPS. Scan grammars: Parallel attribute evaluation via data-parallelism. In *5th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 367–376. ACM, July 1993.

[RT88] T. REPS AND T. TEITELBAUM. *The synthesizer generator reference manual*. Springer-Verlag, New York, third edition, 1988.

[RT89] THOMAS W. REPS AND TIM TEITELBAUM. *The Synthesizer Generator: a system for constructing language-based editors*. Springer-Verlag, 1989.

[RTD83] T. REPS, T. TEITELBAUM, AND A. DEMERS. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems*, **5**(3):449–477, July 1983.

[SV91] S.D. Swierstra and H.H. Vogt. Higher order attribute grammars. In H. Alblas and B. Melichar, editors, *Proceedings of the International Summer School SAGA '91 on Attribute Grammars, Applications and Systems*, volume **545** of Lecture Notes in Computer Science, pages 256–296. Springer-Verlag, 1991.

[TC90] Tim Teitelbaum and Richard Chapman. Higher-order attribute grammars and editing environments. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, **25**(6):197–208, june 1990.

[Tur85] D.A. Turner. A non-strict functional language with polymorphic types. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume **201** of Lecture Notes in Computer Science, pages 1–16. Springer-Verlag, 1985.

[Vog93] Harald Vogt. *Higher order Attribute Grammars*. PhD thesis, Utrecht University, Department of Computer Science, Utrecht, The Netherlands, February 1993.

[VSK89] H.H. Vogt, S.D. Swierstra, and M.F. Kuiper. Higher order attribute grammars. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, **24**(7):131–145, july 1989.

[VSK91] Harald Vogt, Doaitse Swierstra, and Matthijs Kuiper. Efficient incremental evaluation of higher order attribute grammars. In J. Maluszyński and M. Wirsing, editors, *Proceedings of the 3th International Symposium PLILP 91 on Programming Language Implementation and Logic Programming*, volume **528** of Lecture Notes in Computer Science, pages 231–242. Springer-Verlag, 1991.

[WG84] W.M. Waite and G. Goos. *Compiler Construction*. Springer-Verlag, 1984.

[Yeh83] D. Yeh. On incremental evaluation of ordered attribute grammars. *BIT*, **23**:308–320, 1983.

[YK88] Dashing Yeh and Uwe Kastens. Improvements of an incremental evaluation algorithm for ordered attribute grammars. *SIGPLAN notices*, **23**(12):45–50, December 1988.

[Zar90] Alan K. Zaring. *Parallel Evaluation in Attribute Grammar-Based Systems*. PhD thesis, Cornell University, Department of Computer Science, Ithaca, New York, August 1990.

S

Samenvatting

Bij het schrijven van deze samenvatting hinkte ik op twee gedachten. Enerzijds is de "samenvatting in het Nederlands" bij uitstek een onderdeel dat door familie en vrienden gelezen wordt. Het merendeel van deze mensen is buitenstaander wat betreft het vakgebied van de Informatica. Voor hen zou dit een informele inleiding moeten zijn. Anderzijds moet deze samenvatting een korte beschrijving van mijn proefschrift zijn, zodat ingewijden snel een overzicht kunnen krijgen. Ik heb dit dilemma opgelost door niet te kiezen. Eerst geef ik een informele inleiding en daarna een kort overzicht.

Informele inleiding

Compilers

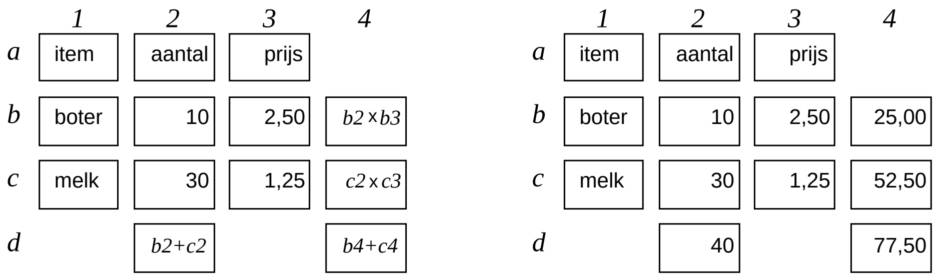
In onze maatschappij wordt tegenwoordig veel met software, dat wil zeggen computerprogramma's, gewerkt. Denk daarbij niet alleen aan tekstverwerkers, maar ook aan software in geldautomaten, videorecorders en het administratiesysteem van de fiscus.

Software moet *geschreven* worden. Programmeurs schrijven software in zogeheten hogere programmeertalen, waarvan Pascal, C, Basic, Cobol en Fortran de bekenste zijn. Zulke programmeertalen nemen tijdrovende, saaie en administratieve taken verbonden aan het programmeren, uit handen.

Echter, computers "begrijpen" hogere programmeertalen niet, zij moeten geïnstrueerd worden in een zogeheten machinetaal. Om een hogere programmeertaal naar machinetaal te vertalen zijn er speciale computerprogramma's in omloop, zogeheten *compilers*. Met andere woorden, compilers vormen een vitale schakel in het productieproces van software.

Een compiler zet een zogeheten *bron*programma, geschreven in een specifieke hogere programmeertaal, om naar een zogeheten *doel*programma in een specifieke machinetaal. Men spreek dan bijvoorbeeld ook van een "Pascal compiler voor een DOS computer". De kwaliteit van het doelprogramma is niet alleen afhankelijk van de kwaliteit van het bronprogramma, maar ook van de kwaliteit van de gebruikte compiler.

Compilers zijn grote, ingewikkelde programma's, die ook geschreven moeten worden. Dit proces wordt ondersteund door *compiler-generatoren*. Tijdens mijn



	1	2	3	4
a	item	aantal	prijs	
b	boter	10	2,50	$b2 \times b3$
c	melk	30	1,25	$c2 \times c3$
d		$b2+c2$		$b4+c4$

a. Invoer

	1	2	3	4
a	item	aantal	prijs	
b	boter	10	2,50	25,00
c	melk	30	1,25	52,50
d		40		77,50

b. Uitvoer

Figuur 2. Een rekenblad voor de melkboer

vierjarig onderzoek heb ik een compiler-generator gemaakt. Een compiler-generator is een computerprogramma dat, na invoer van een minutieuze beschrijving van een hogere programmeertaal en de gewenste machinetaal, een compiler voor die programmeertaal genereert.

De voordelen van een compiler-generator zijn legio. Elke verbetering aan de generator heeft tot gevolg dat elke gegenereerde compiler verbetert. Bovendien worden de produktiekosten en de produktietijd verbonden aan het maken van een compiler, aanzienlijk verminderd. Immers, in plaats van een compiler voor een hogere programmeertaal hoeft alleen nog maar een *beschrijving* van die hogere programmeertaal geschreven te worden. Een student informatica kan in een paar maanden tijd een compiler genereren, terwijl een groep beroeps programmeurs daar eerder vele maanden tot jaren mee bezig is.

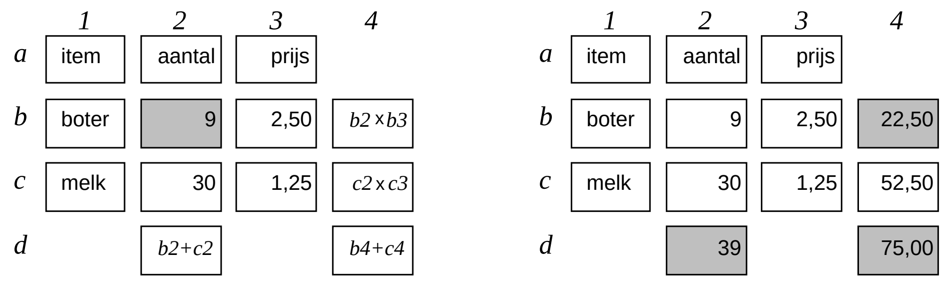
Incrementele berekeningen

De kracht van de compilers gegenereert door de door mij ontwikkelde generator is voornamelijk gelegen in het feit dat ze *incrementeel* zijn. Dat betekent dat na een kleine verandering in het bronprogramma, de compiler de corresponderende *veranderingen* in het doelprogramma aanbrengt—dit in tegenstelling tot het simpelweg vertalen van het gehele (gewijzigde) bronprogramma. Dit laatste zou veel meer tijd zou kosten.

Wat een incrementele berekening is wordt elegant geïllustreerd aan de hand van een compiler-verwante toepassing: een rekenblad of *spreadsheet*. Een rekenblad bestaat uit cellen in een rechthoekig patroon, zie Figuur 2a. Elke cel wordt aangeduid met een letter-cijfer combinatie. Zo wordt de cel links-boven aangeduid met $a1$. Een cel kan tekst, getallen of formules bevatten: cel $a1$ bevat de tekst 'item', cel $b2$ bevat het getal 10 en cel $b4$ bevat de formule $b2 \times b3$.

Een rekenblad-evaluator berekent de waarde van de cellen waar formules in staan. Dus, na invoer van het rekenblad in Figuur 2a wordt de uitvoer van Figuur 2b geproduceerd.

Stel dat we een verandering aanbrengen in het rekenblad; we voeren een zogeheten *edit*-operatie uit. We verlagen het getal in cel $b2$ van 10 naar 9, zie



	1	2	3	4
a	item	aantal	prijs	
b	boter	9	2,50	$b2 \times b3$
c	melk	30	1,25	$c2 \times c3$
d		$b2+c2$		$b4+c4$

a. Wijziging in de invoer

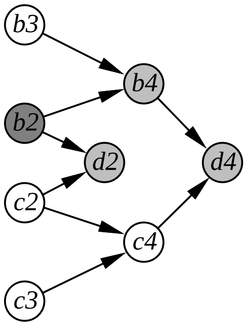
	1	2	3	4
a	item	aantal	prijs	
b	boter	9	2,50	22,50
c	melk	30	1,25	52,50
d		39		75,00

b. Zuinige herberekening

Figuur 3. Wijziging aan een rekenblad

Figuur 3a. De rekenblad-evaluator zou *alle* cellen met formules opnieuw kunnen uitrekenen. Een *incrementele* evaluator berekent alleen de grijze cellen in Figuur 3b. In dit geval spaart dat de berekening van cel $c4$ uit.

Het is overigens niet eenvoudig om in te zien welke cellen herberekend moeten worden. Om dat te bepalen stelt de rekenblad-evaluator een zogeheten *graaf* op die de cel-afhankelijkheden weergeeft. Een knoop in de graaf representeert een cel. Er loopt een pijl van knoop $b2$ naar knoop $b4$ omdat de formule van cel $b4$ refereert aan cel $b2$. Zo'n pijl impliceert dat cel $b4$ ná cel $b2$ uitgerekend moet worden.



Bij een incrementele evaluatie moet *elke* cel wiens knoop in de graaf een opvolger is van $b2$, herberekend worden, tenzij veranderingen eerder uitdoven. Als voorbeeld van dat laatste, beschouwen we de volgende verandering: de prijs van boter gaat naar 0,25 en de hoeveelheid boter naar 100. In dat geval blijft de waarde van cel $b4$ 25,00 zodat $d4$ niet herberekend hoeft te worden. Grafen spelen bij incrementele berekeningen een belangrijke rol. Ze komen dan ook veelvuldig in dit proefschrift voor.

Compilers worden ook vaak toegepast op slechts weinig veranderde invoer. Hoe vaak wordt een computerprogramma niet gecompileerd om er vervolgens achter te komen dat er een kleine fout in staat? Het is daarom jammer dat er nog zo weinig incrementele compilers zijn.

`i:=1` had natuurlijk `i:=0` moeten zijn.

Taal-specifieke editors

Een *editor* is een computerprogramma waarmee gegevens in een computer ingevoerd kunnen worden. Bovendien kunnen met een editor bestaande gegevens gewijzigd worden. Meestal verstaat men onder een editor een *algemeen* programma waarmee gegevens voor uiteenlopende *applicaties* (toepassings programma's zoals compilers, rekenbladen, tekstverwerkers) ingevoerd kunnen worden.

Een applicatie-specifieke editor is een invoer-en-wijzig programma dat "kennis" heeft van een applicatie. Alle huis-tuin-en-keuken applicaties zoals tekstverwerkers en rekenbladen zijn uitgerust met applicatie-specifieke editors. Het grootste voordeel van zulke editors is dat ze de gebruiker kunnen sturen: er kan geen al te grote onzin in worden gevoerd.

LaTeX is geen huis-tuin-en-keuken applicatie.

Compilers zouden ook uitgerust kunnen worden met een brontaal-specifieke editor. Dat is met name interessant als er een *incrementele* compiler beschikbaar is die tijdens het intypen van het bronprogramma voortdurend het bijbehorende doelprogramma berekend. Zulke systemen zouden de arbeidsproduktiviteit van programmeurs enorm verhogen. Immers, fouten in het bronprogramma worden direct als zodanig herkend door het system, en meegedeeld aan de gebruiker. Het grootste voordeel is echter dat de editor "ondervraagd" kan worden over bepaalde kenmerken van het bronprogramma. Voor ingewijden: als de editor constateert dat `i:=1` incorrect is omdat `i` niet gedeclareerd is kan het een menu aanbieden met alle integer variabelen in de huidige scoop en een optie om de declaratie `i:integer` toe te voegen aan de declaraties.

Kort overzicht

Dit proefschrift beschrijft de generatie van incrementele compilers, met name als onderdeel van een taal-specifieke "*language based*" editor.

Omdat we uitgaan van een taal-specifieke editor, kunnen we het ontleedtraject (*lexical scanning* en *parsing*) overslaan: de editor onderhoudt een boomrepresentatie van het ingevoerde programma. Dat betekent dat compileren niets anders is dan het attribueren van de boom; de compiler is een (incrementele) *attribute evaluator*.

De titel: *Generating incremental attribute evaluators*.

Het formalisme waarmee we bomen en attributen beschrijven is het attribuut grammatica formalisme.

We geven twee methodes om incrementele attribuut evaluatoren te construeren. De eerste methode is een bestaande: *visit-sequences* sturen boom decoratie. De tweede methode is nieuw: *visit-sequences* worden afgebeeld op *visit-functies* die *gecached* worden om incrementeel gedrag te verkrijgen. Een complicatie bij deze aanpak vormen zogeheten *intra-visit-dependencies*. Om dat probleem op te lossen worden bindingen geïntroduceerd.

Visit-functies zijn functies zonder zij-effecten. Dit opent de mogelijkheid tot allerhanden optimalizaties. Aan de orde komen *splitting*, *elimination*, *unification*, *folding*, *normalization* en *untyping*.

We bespreken hoe visit-sequences berekend worden aan de hand van een grammatica. We wijzigen Kastens' *ordered scheduling* op twee vlakken. Om te beginnen voeren we dat grafen in (in stap 4) waardoor een grotere klasse van gramatica's geaccepteerd wordt. Ten tweede voeren we een nieuw orderings algorithme in (*chained scheduling*) dat visit-sequences berekent die beter geschikt zijn voor omzetting naar visit-functies.

Een groot deel van het onderzoek heeft zich toegespitst op de haalbaarheid van een functionele aanpak: we hebben een generator geschreven. De gegenereerde evaluatoren zijn eenvoudig, elegant en robuust. Bovenal zijn ze snel. De generator is onder andere gebruikt om een deel van zichzelf te genereren.

C

Curriculum vitae

Maarten Christiaan Pennings

28 december 1965

Geboren te Utrecht.

augustus 1978 – juli 1982

Atheneum B op het Augustinianum te Eindhoven.

augustus 1982 – juli 1984

Atheneum B op het Sint Maartenscollege te Maastricht.
Atheneum diploma 4 juni 1984.

september 1984 – juli 1985

Studie werktuigbouwkunde aan de Technische Universiteit Eindhoven.
Propaedeuse werktuigbouwkunde 16 september 1985.

september 1985 – oktober 1990

Studie informatica aan de Technische Universiteit Eindhoven.
Propaedeuse informatica 10 juli 1986.
Doctoraal 1^{e} deel informatica 2 juli 1987.
Doctoraal informatica 25 oktober 1990.
Aanvullend examen informatica 25 oktober 1990.

november 1990 – november 1994

Onderzoeker in opleiding in dienst bij de Stichting informatica-onderzoek Nederland (SION) van de Nederlandse Organisatie voor Wetenschappelijk Onderzoek (NWO) gedetacheerd bij de Universiteit Utrecht.

januari 1995 –

Wetenschappelijk medewerker bij het Philips Natuurkundig Laboratorium te Eindhoven.

G

Glossary

This glossary is meant as a quick reference to all known and not so well-known symbols contained in this thesis: it gives a short description and a page number (after ▷) that refers to a page where the symbol is defined.

Variables

This section lists the variables that occur in this thesis. Some variables are compound. For example $p^{v \to w}$ consists of three components, namely p, v and w (the $\to$ is syntactic sugar). To facilitate lookup, variables are subdivided according to the number of components they are made up of.

One component

a, b, c	(non-terminal suffix for) attributes ▷13
α, β, γ	attribute occurrences ▷13
D	dependency set ▷10
f, g	semantic functions ▷14
G	graph or grammar ▷10
i, j	non-terminal indices in production ▷11
K	node of an abstract syntax tree ▷12
l, k	(production suffix for) local attributes ▷13
p, q, r	productions ▷11
$\mathbf{p}$, $\mathbf{q}$, $\mathbf{r}$	production constructors ▷11
$\underline{p}$, $\underline{q}$, $\underline{r}$	shells ▷105
$\underline{\underline{p}}$, $\underline{\underline{q}}$, $\underline{\underline{r}}$	encapsulators ▷104
R, S	root and start non-terminals ▷11
σ, τ	attribute instances ▷28
T	abstract syntax tree ▷12
v, w, t, u	visit indices ▷38
X, Y, N	non-terminals (and trees of that type) ▷11
x, y	(production suffix for) attributable attributes ▷16

Two components

p^v, q^w	split productions ▷93
$\mathbf{p}^v$, $\mathbf{q}^w$	split production constructors ▷93
$p.l$, $q.k$	local attributes (and occurrences thereof) ▷13
$p.x$, $q.y$	attributable attributes (and occurrences thereof) ▷16
$p\mathbf{o}i$, $q\mathbf{o}j$	non-terminal occurrences (with pairing via $\mathbf{o}$) ▷12
T^v	split tree ▷90
X^v, Y^w	split non-terminals (and trees of that type) ▷93 also visit to non-terminal in plan icons ▷66
$X.a$, $Y.b$	attributes ▷13

$p{\cdot}i$ is the ith non-terminal of p (list indexing).

Since $p{\cdot}i$ is a non-terminal, $p{\cdot}i.a$ denotes an attribute too.

Three components

$(K, X.a)$	non-terminal attribute instance ▷28
$(K, p.l)$	local attribute instance ▷28
$p^{v\to w}$, $q^{v\to w}$	wrappers ▷77
$\mathbf{p}^{v\to w}$, $\mathbf{q}^{v\to w}$	wrapper constructors ▷77
$p.x.a$, $q.y.b$	generated attribute occurrences ▷16
$p\mathbf{o}i.a$, $q\mathbf{o}j.b$	attribute occurrences (with pairing via $\mathbf{o}$) ▷13
$p\mathbf{o}i^v$, $q\mathbf{o}j^w$	split non-terminal occurrences ▷92, 93
$T^{v\to w}$	binding ▷69
$X^{v\to w}$, $Y^{v\to w}$	parcels (and bindings of that type) ▷70
$X.\mathrm{a}^{v\to w}$	binding attribute ▷71
$X.\mathrm{i}^{v\to w}$	inherited binding attribute ▷71
$X.\mathrm{s}^{v\to w}$	synthesized binding attribute ▷71

The a, i and s are part of fixed syntactic sugar (note the font).

Four components

$p\mathbf{o}i.\mathrm{a}^{v\to w}$	binding occurrence ▷71
$p\mathbf{o}i.\mathrm{i}^{v\to w}$	inherited binding occurrence ▷71
$p\mathbf{o}i.\mathrm{s}^{v\to w}$	synthesized binding occurrence ▷71
$p.x.\mathrm{a}^{v\to w}$	generated binding occurrence ▷80
$p.x.\mathrm{i}^{v\to w}$	generated inherited binding occurrence ▷80
$p.x.\mathrm{s}^{v\to w}$	generated synthesized binding occurrence ▷80

Symbols

Symbols are subdivided into two lists. The first list contains constants, functions and operators known from mathematics and functional programming; the second list contains grammar related symbols. In both lists, $\Box$ is used as placeholder for arguments.

Ordinary math symbols

$\Box\ \Box$	function application ▷9
$\Box{\cdot}\Box$	list (and tree) indexing ▷10
$\Box^{+}$	transclose (graphs, dependency sets) ▷10
$\Box + \Box$	addition (numbers), disjunct union (sets) ▷9
$\Box \cup \Box$	set union ▷9
$\Box \in \Box$	set membership ▷9
$\Box \circ \Box$	functional composition ▷9
$\Box \longrightarrow \Box$	arc in graphs and dependency sets ▷10
$\Box \setminus \Box$	set difference ▷9
$\Box \times \Box$	cross product of sets ▷9
$\Box : \Box$	cons of element to list ▷10
$\Box +\!\!\!+ \Box$	concatenation of lists ▷10
$\Box :: \Box$	has type ▷11
$(\Box)$	tuple; also $\lgroup \Box \rgroup$ ▷9
$[\Box]$	list; $[\,]$ empty list ▷10
$\{\Box\}$	set; $\{\}$ empty set ▷9
$[\Box \mid \Box \leftarrow \Box]$	comprehension; also $[\Box \mid \Box \in \Box]$ ▷10
$\Box\ \mathbf{div}\ \Box$	integer division
$\mathit{fst}\ \Box$	first component of a tuple ▷9
$\mathit{head}\ \Box$	head of a list ▷10
$\mathit{len}\ \Box$	length of a list ▷10
$\Box\ \mathbf{max}\ \Box$	maximum of two numbers
$\Box\ \mathbf{min}\ \Box$	minimum of two numbers
$\mathcal{O}(\Box)$	order; not to be confused with $O(\Box)$
$\mathit{snd}\ \Box$	second component of a tuple ▷9
$\mathit{tail}\ \Box$	tail of a list ▷10

Grammar related symbols

$\square.\square$	attribute selection: $X.a$, $p.l$, $p.x$, $p\mathbf{o}i.a$, $\ldots$ ▷13, 16, 71, 80
$\square{\cdot}\square$	$p{\cdot}i$ denotes the ith non-terminal of p; list indexing ▷10
$\widehat{\square}$	meta language type-of operator ▷77, 93
$\overline{\square}$	$\overline{X}$ denotes a non-terminal attribute of type X ▷17
$\underline{\square}$	$\underline{X}$ denotes the split representation $\llparenthesis X^1, \ldots, X^{\mathrm{v}X} \rrparenthesis$ ▷93 $\underline{p}$ denotes a shell for production p ▷105
$\underline{\underline{\square}}$	$\underline{\underline{X}}$ denotes the representation $(X , \llparenthesis X^1, \ldots, X^{\mathrm{v}X} \rrparenthesis)$ ▷105 $\underline{\underline{p}}$ denotes an encapsulator for production p ▷104
$?$	joker (for non-terminal attributes) ▷17
$\square \upharpoonright \square$	projection (attribute occurrence to attribute) ▷31
$\llparenthesis \square \rrparenthesis$	tuple of split trees; also $()$ ▷9, 93
$\square[\square]$	paste (attributes into attribute occurrence dependency) ▷31
$A(\square)$	set of attributes ▷13, 16
$A(X)$	attributes of non-terminal X; officially $A_{nont}(X)$ ▷13, 16
$A_{inh}(X)$	inherited attributes of non-terminal X ▷13, 16
$A_{syn}(X)$	inherited attributes of non-terminal X ▷13, 16
$A_{ata}(p)$	attributable attributes of production p ▷16
$A_{loc}(p)$	local attributes of production p ▷13, 16
$BA(\square)$	set of binding attributes ▷71
$BA(X)$	binding attributes of non-terminal X ▷71
$BA_{inh}(X)$	inherited binding attributes of non-terminal X ▷71
$BA_{syn}(X)$	inherited synthesized attributes of non-terminal X ▷71
$\square\,\mathbf{BI}\,\square$	$X\,\mathbf{BI}\,v$ denotes the inherited binding attributes for visit v to X ▷71
$bind(\square, \square, \square)$	$bind(p, v, w)$ denotes the attribute occurrences defined in visit-sub-sequence v of p that are used visit-sub-sequence w of p ▷75
$BO(\square)$	set of binding occurrences ▷71, 80
$BO(p\mathbf{o}i)$	binding occurrences of $p\mathbf{o}i$ ▷71
$BO(p.x)$	binding occurrences of $p.x$ ▷80
$BO(p)$	all binding occurrences of production p ▷71, 80
$\square\,\mathbf{BS}\,\square$	$X\,\mathbf{BS}\,v$ denotes the synthesized binding attributes for visit v to X ▷71
Δ	set of AFFECTED attribute instances ▷36
$dat(\square)$	dependencies of attribute and visit vertices ▷128
$def(\square, \square)$	$def(p, v)$ denotes the defined attribute occurrences of visit-sub-sequence v of p ▷75

$\mathbf{s}\Box$	$\mathbf{s}p$ denotes the size of a production p ▷12
$\Box\mathbf{S}\,\Box$	$X\mathbf{S}\,v$ denotes synthesized attributes for visit v to X ▷42
$slot(\Box)$	the slot number of an interface set $X\mathbf{I}\,v$, $X\mathbf{S}\,v$ or attribute $X.a$ ▷42
$split$	$split_X$ denotes a function that converts an abstract syntax tree of type X into a split representation $\underline{X}$ ▷94
$sref(\Box)$	annotation for $eval$ instruction listing a syntactic reference ▷74
$suspend(\Box)$	$suspend(v)$ denotes the plan instruction to exit visit v ▷38, 55
$syn(\Box)$	annotation for a visit-sub-sequence listing a synthesized occurrence ▷74
$tdp(\Box)$	transitive (attribute occurrence) dependencies of a production ▷122
$tds(\Box)$	transitive (attribute) dependencies of a symbol (non-terminal) ▷122
$use(\Box, \Box)$	$use(p, v)$ denotes the set of used attribute occurrences of visit-sub-sequence v of p ▷75
$uses(\Box)$	annotation for $eval$ instruction listing an used occurrence ▷74
$\mathbf{v}\Box$	$\mathbf{v}X$ denotes the number of visits of non-terminal X ▷42
$visit$	$visit_{X^v}$ or $visit_X^v$ denotes visit-function for visit v to X ▷78, 94
$visit(\Box, \Box)$	$visit(i, v)$ denotes the plan instruction to visit $p\mathbf{o}i$ for the vth time ▷38, 55
$vss(\Box, \Box)$	$vss(p, v)$ denotes visit-sub-sequence v of production p ▷75

Index

The index does not contain references to symbols; see the glossary for that. A pagenumber followed by an 'm' refers to a marginal note on that page. Similarly, an 'f' suffix refers to a figure. An underlined page number refers to a definition of the index entry.

legend
□f : figure
□m : margin
$\underline{\square}$: def

legend

□f : figure

□m : margin

□ : def

legend

□f : figure

□m : margin

□ : def

legend

□f : figure

□m : margin

□ : def

legend
□f : figure
□m : margin
□ : def

legend

□f : figure

□m : margin

□ : def