Modular Grammar Specification

Adrian Johnstone and Elizabeth Scott

Department of Computer Science,
Royal Holloway, University of London,
Egham, Surrey, United Kingdom

Mark van den Brand

Eindhoven University of Technology, Mathematics and Computer Science,
Den Dolech 2, NL-5612 AZ Eindhoven, The Netherlands

Abstract

We establish a semantics for building grammars from a modularised specification in which modules are able to delete productions from imported nonterminals. Modules have import lists of nonterminals; some or all of an imported nonterminal's productions may be suppressed at import time. There are two basic import mechanisms which (a) reference or (b) clone an imported nonterminal's productions. One of our goals is to allow a precise answer to the question: 'what character level language does this grammar generate' in the face of difficult issues such as the mutual embedding of languages that have different whitespace and commenting conventions. Our technique is to automatically generate a character level grammar from grammars written at token level in the conventional way; the grammar is constructed from modules each of which may have its own whitespace convention.

Keywords: Context Free Grammar, Modularity, Whitespace processing

This paper concerns modularity constructs and their applicability to context free grammar specifications. We advocate principles which are intended to provide extensibility whilst ensuring that it is easy to reason about the effects of the composition mechanisms. We propose a mechanism in which:

1. unless otherwise specified, modular composition preserves *name hy-*

Email addresses: e.scott@rhul.ac.uk, a.johnstone@rhul.ac.uk (Adrian Johnstone and Elizabeth Scott), m.g.j.v.d.brand@tue.nl (Mark van den Brand)

giene;

2. individual productions can be suppressed on import by pattern matching *deleters* against disallowed right-hand sides;
3. there is sophisticated support for automatic whitespace handling via whitespace module imports.

Name hygiene implies that after import, the productions of some nonterminal X in module L remain distinct from the productions of nonterminal X from module M. We provide left hand side renaming to allow controlled merging of productions into existing or new nonterminals.

Deleters implemented via pattern matching of right hand sides allow the user to specify 'unwanted' productions independently of the (possibly complicated) chain of imports that a production may be involved in.

Localised whitespace handling allows principled language embedding and overcomes problems with, for instance, the use of braces `{ }` to delimit comments in SQL being in direct conflict with the use of braces to delimit code blocks in C family languages.

This paper is in three main parts. We first discuss the general principles which motivated our approach, and then we describe some related work in grammar composition and refactoring, and discuss the capabilities of some existing tools' modularisation facilities.

The second part concerns our proposal for a *Modularised Grammar Specification* and defines formally how a context free grammar with hidden whitespace handling is induced from an MGS. We deliberately focus on the semantics of the mechanism without specifying, for instance, whether the base productions for a module are even held in the same file or syntactic unit: our intention is to allow MGS semantics to be added to existing tools in the style that best fits their existing practice.

The final part shows examples of the utility and of the limitations of our present proposal using the concrete syntax from the ART GLL parser generator tool with a series of use-case studies.

1. Motivation

Modularity and associated concepts such as inheritance and overriding are routinely used in the software engineering domain to support (a) the decomposition of a large application into small subsections that can be worked on separately without disturbing the contents, and behaviour, of other modules; (b) the facilitation of incremental builds, in which only the

changed module and modules dependent on those changes need to be rebuilt; (c) information hiding (abstraction) that allows internal details to be suppressed as an aid to comprehension; (d) name-space management to allow the same identifier to be used for different purposes in different modules; and (e) reuse of pre-existing fragments with customisation, say, by overriding internal methods.

In principle, all of these concerns arise in grammar engineering too; however grammarware does not have the complexity of general software — the semantics of context free grammar notations are extremely simple compared to the semantics of a programming language: grammars are essentially expressions over language constants (terminals) and variables (nonterminals) with two operators: concatenation into a sequence (a production) and alternation over productions. A thoughtful analysis of the challenges implicit in grammar engineering and reuse has been given by Klint, Lämmel and Verhoef [15]. In the rest of this section we examine several facets of our proposed scheme, and their rationale.

1.1. Granularity

Modularisation schemes are mechanisms for combining fragments of specifications, and the granularity of these fragments is key: we would not normally expect a software modularisation scheme to allow one particular expression in a function to be changed; instead such schemes usually compose over named pieces of code. In object oriented languages the units of combination are usually the class member and the class itself and it is common to provide both inheritance at the level of members, and traditional modularity at the level of classes. In Java, the package mechanism ensures name hygiene so that the name spaces of individual packages cannot interfere. Name space management is the central topic of this paper and will be discussed in the following sections.

We could choose to allow modularity operations over, for instance,

1. the set of all productions for a set of nonterminals;
2. the set of all productions for a given nonterminal;
3. an individual production;
4. the set of productions reachable from a given nonterminal;
5. the elements of a particular production, allowing users to specify things like *'replace the third element of the fourth production of nonterminal X.*

This last is clearly very unwieldy, and in any case productions themselves are usually short compared to functions in programming languages. Thus

there seems to be no justification for allowing module composition at the level of subsequences.

1.2. Name hygiene

In an earlier, limited, version of this work [14], we describe the import-by-reference mechanism (the 'single-arrow' operator $\leftarrow$); the deletion of productions on import; and the machinery for handling whitespace in embedded languages. In this paper we extend our proposal to allow mutation of recursive sub-grammars by providing an import-by-clone mechanism (the 'double-arrow' operator $\Leftarrow$) which copies a production into the namespace of an importing module with appropriate renaming of nonterminal instances.

As a notational convenience, we propose that any concrete syntax includes notations for importing all of the productions within a complete module (which corresponds to option 1), importing all of the productions reachable from a left hand side nonterminal (option 4), and importing all of the productions in modules containing productions reachable from a left hand side nonterminal (option 5). All of these may be viewed as macros that expand to sets of our basic operations.

1.3. Deleters

In our theoretical treatment in Section 4 we consider the import of a single nonterminal's productions (corresponding to option 2 above) and deleters (which act as filters during imports, causing some productions to be suppressed) that work at the level of individual productions (option 3 above).

1.4. Whitespace handling

We aim to provide a general mechanism that allows users to build both token level and character level tools in a way that seems natural to users of existing tools, and in particular to describe the mutual embedding of languages with different whitespace and commenting conventions.

Conventionally, programming language grammars are described at the level of tokens. For free format languages (such as Java and others in the Algol family) whitespace and comments are understood to be suppressed and discarded before token level parsing occurs. Usually, therefore, the grammar writer makes a global specification of a whitespace/comment pattern which is understood to be consumed before and after each token.

Of course, the grammar writer could specify a character level context free grammar in which the whitespace is explicitly specified by writing an instance of a whitespace-matching nonterminal on either side of the character level specification of the tokens. In practice, such grammars are verbose

and unwieldy, since the syntax of the token level language is obscured by the whitespace details. In addition, at the level of characters (as opposed to tokens) very few languages have grammars which are admissible by near-deterministic parsing algorithms such as LALR(1) or LL(1). Indeed, one view of token-level parsing is that by segmenting the input character string into tokens we separate out different semantic concepts. Consider, for instance, the Pascal substrings `for x := 3 downto 1 do` and `for x := 3 do` and a grammar with production

```
forStatement ::= 'for' id ':=' expression
                 ( 'to' expression | 'downto' expression )?
                 'do' ;
```

At character level, the `()?` optional bracket has the character `d` in both its first and follow set, and so a character level grammar is not LL(1)-deterministic. If we use a separate lexical analyser, then the first and follow token sets are disjoint, and so the token level grammar is LL(1).

The conventional token level description of languages with global whitespace does not extend well to mutually embedded languages with differing whitespace and commenting conventions. If we make a single global whitespace convention which describes the union of the embedded whitespace conventions then the resulting language will in general be larger than strictly necessary, and may display unnecessary ambiguities — the use of braces `{ }` to delimit comments in SQL being in direct conflict with the use of braces to delimit code blocks in C family languages, for instance.

In our scheme there is a whitespace convention associated with each module, and care is taken when combining modules with different whitespace conventions to ensure that the resulting grammar describes the language embedding correctly, whilst reducing the potential for ambiguity.

1.5. Separate compilation

Although in this paper we focus in composability over grammars, composability over parsers is sometimes desirable and in principle modules could be used to specify parser composition as well as grammar composition. This becomes important when developing large grammars: if parser generation time for a complete grammar is excessive, then separate generation and compilation of parser components allows small fragments of grammars to be efficiently reworked without needing to rebuild the whole parser.

The magnitude of the problem is parser technology dependent. For instance, it is well known that the computation of even LR(0) parse tables

is exponential in the size of the grammar and although this worst case is not usually encountered with conventional programming language grammars, user experiences of LR-table based parsers is that build times can sometimes be disappointing. In response, several authors have proposed schemes for composing LR tables, leading to incremental LR parser generators [13, 12, 5][1]. In practice, Bravenboer *et al* report order of magnitude speed improvements, though Horspool notes that the addition of a single production can introduce an exponential number of new states in an LR(0) automaton.

Recursive Descent (RD) parsers naturally lend themselves to composability, since there is one parse function per nonterminal in the grammar, and no co-dependencies: changing one production changes only one parse function. In principle, each parse function could be separately compiled and linked, though in practice RD parser generation is so fast that this is generally deemed unnecessary. The GLL algorithm [20, 21] used in our ART tool displays similar attributes; parsers can be easily composed from precompiled fragments. However, when running ART under Windows-7 on a Toshiba Tecra R840 laptop, the generation time for a Java parser is 0.13 CPU seconds and the full compilation time of a test program incorporating the resulting 16,289 line parser is 0.39 CPU seconds. Hence, complete regeneration of a running Java parser after a change requires only around 0.5 seconds, and so we have not yet needed to implement separate compilation of modules.

1.6. Approaches to modularity

Language composition, and by extension grammar composition, is an active area of research driven by the desire to embed domain specific languages such as SQL in more general programming languages, and to add extension semantics to old languages without disturbing the previously defined core.

Van Wyk's Silver system [24] provides extensible attribute grammars, and works with a parser generator called Copper which supports language composition. Copper is an LALR parser generator, and LALR grammars are not closed under union. In [19], Schwerdfeger and Van Wyk describe conservative constraints on composable sub-grammars that ensure an LALR composed grammar; this allows Copper to reject non-composable sub-grammars with useful error messages.

[1] There is a broader literature concerning incremental *parsing*, which concerns derivation tree rework in response to changes in a string being parsed against a *fixed* grammar.

The JastAdd modular semantics system [8] is a modular system for extending abstract syntax and attribute-based semantics. Parsing of concrete grammars may be performed by any parser generator that allows semantic actions to be executed as part of a derivation traversal; these actions construct the base abstract syntax tree (AST) which is then transformed and interpreted by JastAdd. Extensibility of ASTs is provided using static aspects, but this capability naturally does not extend to the concrete parser.

The ANTLR4 [18] parser generator has an import mechanism that processes a list of imports in a depth first fashion, adopting the first definition of some rule X that it encounters and ignoring subsequent instances [16]. Any imported rule is then discarded in favour of locally defined rules and there is no rule merging or renaming. This mechanism is essentially that of inherited overloading with multiple inheritance. It has the virtue of being easy to use and familiar for software engineers, but is essentially unable to perform grammar composition below the level of an individual module.

Rats! is a Parsing Expression Grammar [9] (PEG) based parser generator with extensive modularisation features. The basic model is similar to class overloading, but overridden nonterminals may be accessed using a fully qualified name. PEG's use ordered grammars, and so the order of individual productions in composed grammars is significant; hence when forming the union of a production and an imported nonterminal, the insertion point within the imported productions must be specified. The related complications of deciding on an ordering when importing nonterminal X from more than one module (called an *ambiguous* import) leads to the constraint that instances of such nonterminals must be fully qualified when used; in effect all but simple imports require knowledge of the global name space over all modules.

Algebraic approaches to syntax extension have been explored by Andersen and Brabrand [2, 3] using their so-called Banana Algebra which provides operators for transforming named productions within grammars; effectively languages can be transformed by incrementally transforming their grammars. This approach is powerful, and includes an analysis that allows transformations that produce 'nonsensical' grammars to be detected.

The original SDF general context free parser generator [11] provided an unparameterised import mechanism that merged together productions from identically named nonterminals. SDF2 [23] has a richer import mechanism that allows parameterisation of modules with associated renaming on import, in addition to explicit renaming which may be used as part of an import expression to rename nonterminals that have not been nominated as parameters on the exporting module. By renaming an imported nonterminal X to

some non-reachable nonterminal, the productions for X may effectively be deleted. There is no pattern-based specification of deleters. We discuss aspects of the SDF2 mechanism in detail in section 8.1. A related mechanism for SDF is discussed by de Jonge [7].

We now introduce notation, and give some informal examples of our modularity operators before giving a formal definition of our import and whitespace-handling mechanisms.

2. Notation

Context free grammars

A *Context Free Grammar* (CFG) is a 4-tuple $(\mathbb{N}, \mathbb{T}, \mathbb{P}, S)$ where $\mathbb{N}$ is a finite set of *nonterminal* symbols, $\mathbb{T}$ is a finite set of *terminal* symbols with $\mathbb{T} \cap \mathbb{N} = \emptyset$, $\mathbb{P}$ is a finite set of *productions* of the form $A ::= \alpha$ where $\alpha \in (\mathbb{N} \cup \mathbb{T})^*$, α is an *alternate* of A, and $S \in \mathbb{N}$ is the *start* symbol.

The relation *derives* written $\Rightarrow$ is defined as follows: if $\alpha B \gamma \in (\mathbb{N} \cup \mathbb{T})^*$ and $B ::= \beta \in \mathbb{P}$, then $\alpha B \gamma \Rightarrow \alpha \beta \gamma$. We write $\Rightarrow^+$ for the transitive closure of $\Rightarrow$, and $\Rightarrow^*$ for the reflexive and transitive closure of $\Rightarrow$. We say that α is a *sentential form* of the grammar, Γ, if $S \Rightarrow^* \alpha$ and the language of Γ, $L(\Gamma)$, is defined as $L(\Gamma) = \{w \in \mathbb{T}^* \mid S \Rightarrow^* w\}$. We say that $x \in (\mathbb{N} \cup \mathbb{T})$ is *reachable* if x appears in some sentential form of Γ.

(We note that there may be two or more different grammar productions with the same left hand side and right hand side, often referred to as 'repeated rules'. Grammars constructed, as described below, from a modular specification will not contain repeated rules. If we wished to permit this then we could formally assert that each alternate had an associated integer whose sole purpose is to allow the identification of the production and use this in the grammar construction procedure.)

In [14] we defined the set of nonterminals in a modular grammar specification as a partition, expressing the fact that nonterminal names in distinct modules were distinct. In our extended modularised grammar specification we need to have a correspondence between nonterminal names in each module and, to achieve this, new nonterminal names may be created as part of the grammar induction process. Thus we do not require nonterminals in distinct modules to have distinct names, we shall create uniqueness in the corresponding grammar by prepending module names to nonterminal names.

Grammar Specification Modules (GSM)

We begin with a set $\mathbb{M}$ of module names, a set $\mathbb{N}$ of nonterminals, and a set $\mathbb{T}$ of terminals. We require that the sets $\mathbb{M}$, $\mathbb{N}$ and $\mathbb{T}$ are pairwise disjoint.

A *grammar specification module* (GSM) over $(\mathbb{M}, \mathbb{N}, \mathbb{T})$ consists of

1. a *module name*, $M \in \mathbb{M}$
2. an optional whitespace convention $(W_M, L.V)$, where $L \in \mathbb{M}$ and $V, W_M \in \mathbb{N}$, which may be written $\mathbb{W}_M$,
3. a finite set $\mathbb{I}_M$ of *import by reference expressions* of the form $X \leftarrow H.Y$, and *import by clone expressions* of the form $X \Leftarrow H.Y$, where $X, Y \in \mathbb{N}$ $H \in \mathbb{M}$, and $H \neq M$,
4. a finite set $\mathbb{D}_M$ of *deleters* of the form $X :/= \delta$, where $X \in \mathbb{N}$ and $\delta \in (\mathbb{N} \cup \mathbb{T})^*$,
5. a finite set $\mathbb{P}_M$ of *module productions* of the form $X ::= \delta$, where $X \in \mathbb{N}$ and $\delta \in (\mathbb{N} \cup \mathbb{T})^*$. ($\delta$ is an *alternate* of X in M.)

Where no confusion results we shall adopt the standard convention of grouping sets of module productions (and sets of grammar productions) with the same left hand sides together. So $X ::= \alpha|\beta$ is an abbreviation of two productions $X ::= \alpha$, $X ::= \beta$.

We shall require that there is at most one GSM for each module name in $\mathbb{M}$ and thus we can, and we shall, refer to a GSM by its module name.

We say that $A \in \mathbb{N}$ is *nullable in* M if $(A ::= \epsilon) \in \mathbb{P}_M$ or if $(A ::= x_1 \ldots x_n) \in \mathbb{P}_M$ where each x_i is nullable in M, $1 \leq i \leq n$.

A *Modularised Grammar Specification* (MGS) over $(\mathbb{M}, \mathbb{N}, \mathbb{T})$ is a set Π of grammar module specifications over $(\mathbb{M}, \mathbb{N}, \mathbb{T})$ in which each element of $\mathbb{M}$ is the module name of at most one grammar module specification, one module M_0 is specified to be the *start* or *main module*, and one nonterminal $S \in \mathbb{N}$ is the *start nonterminal*.

3. Grammar induction - some examples

We now informally discuss, via examples, the basic way in which a modularised grammar specification over $(\mathbb{M}, \mathbb{N}, \mathbb{T})$ specifies a context free grammar whose nonterminal names are of the form $K.A$, where $K \in \mathbb{M}$ and $A \in \mathbb{N}$, and whose terminals are in $\mathbb{T}$. In our examples we shall assume that
MB,MF,MG,ME,MI,MW $\in \mathbb{M}$,
$X, S, E, G, B \in \mathbb{N}$, and
`&`, `t`, `f`, `+`, `-`, `0`, `1`, `while`, `do`, `until`, `if`, `id` $\in \mathbb{T}$.

A module which does not import anything is simply a name, say MB, and a list of productions, $\mathbb{P}_{MB}$:

```
MB
X ::= X & X  |  t  |  f
```

This specification induces the following context free grammar whose sole nonterminal is $MB.X$,

```
MB.X ::= MB.X & MB.X  |  t  |  f
```

Two or more modules together induce a set of grammar rules which are not initially connected. For example, the following two MGSs

```
MF                              MG
X ::= X + X                     G ::= 0 | 1
```

together induce the grammar productions

```
MF.X ::= MF.X + MF.X
MG.G ::=  0  |  1
```

To use the rules from MG to complete the language generated by X we can use the import statement X<-MG.G. With ME as main module, the modularised grammar specification

```
ME                               MG
X <- MG.G                        G ::= 0 | 1
X ::= X + X
```

induces the grammar

```
ME.X ::= ME.X + ME.X  |  0  |  1
MG.G ::= 0  |  1
```

The import X<-MG.G causes ME.X to acquire the right hand sides of MG.G.

One side-effect of the grammar induction process is that there will in general be many unreachable nonterminals; in this example MG.G is unreachable. We assume that all unreachable nonterminals which do not come from the main module are suppressed from the final grammar.

We now discuss several small examples which illustrate the basic import mechanisms.

Example 1 Using $\leftarrow$
In the module *M1* below we define a language which contains a while construct and an assignment statement, and uses the Boolean and expression

languages defined in *MB* and *ME*. Since we do not want to change the productions defined in *MB* and *ME* we use import-by-reference ← to create grammar rules corresponding to `E` and `B`. We write the following GSM *M1*

```
M1
E <- ME.X
B <- MB.X
S ::= while B do S  |  id := E
```

The modularised grammar specification {*MG*, *ME*, *MB*, *M1*}, with *M1* as the main module, induces, after reachability analysis and suppression of unreachable nonterminals, the following grammar

```
M1.S  ::=  while M1.B do M1.S |  id := M1.E
ME.X  ::=  ME.X + ME.X  |  0  |  1
MB.X  ::=  MB.X & MB.X  |  t  |  f
M1.E  ::=  ME.X + ME.X  |  0  |  1
M1.B  ::=  MB.X & MB.X  |  t  |  f
```

with start symbol `M1.S`. The module name extensions `ME.X` and `MB.X` ensure that the different uses of the nonterminal name `X` are kept separate.

Note: the order in which the grammar rules, and the alternates within a rule, are written is not significant, except for the convention that unless otherwise stated the rule for the start nonterminal is listed first. Our ART implementation adopts this convention.

Example 2 Modifying imported modules
Now suppose that we wish the expression sublanguage in the final grammar also to contain a subtraction operator. For this we need to extend the language defined in *ME*. In *M2* below, the import-by-clone statement `X<=ME.X` causes `M2.X` to acquire the right hand sides of `ME.X` but with the occurrences of the module extension *ME* replaced with *M2*.

```
M2
X <= ME.X
B <- MB.X
S ::= while B do S  |  id := X
X ::= X - X
```

The modularised grammar specification {*MG*, *ME*, *MB*, *M2*}, with *M2* as the main module, then induces the following grammar

```
M2.S  ::=  while M2.B do M2.S  |  id := M2.X
M2.X  ::=  M2.X - M2.X  |  M2.X + M2.X  |  0  |  1
M2.B  ::=  MB.X & MB.X  |  t  |  f
MB.X  ::=  MB.X & MB.X  |  t  |  f
```

The statement `X <= ME.X` has created a copy of the rules in *ME* with which the new rule `M2.X ::= M2.X - M2.X` are combined. The nonterminals `MG.G` and `ME.X` were unreachable and their productions were removed from the final grammar, which has start symbol `M2.S`.

(In a concrete syntax we would expect the module extensions of nonterminal names to be suppressed where no name clashes result.)

Example 3 Chains of imports
Now we suppose that the modules *MG*, *MB*, *ME* and *M2* from Examples 1 and 2 pre-exist and are not to be changed, but that we wish to add an if-statement to the language specified by *M2*. This example illustrates a chain of imports. We create a GSM *M3*, below, which imports *M2*, and as we want the while- and if-statements to nest correctly we use $\Leftarrow$. This will create nonterminals `M3.X` and `M3.B` whose productions we also need to import.

```
M3
S <= M2.S
X <= M2.X
B <= M2.B
S ::= if B then S
```

The import `X<=M2.X` assigns to `M3.X` productions both from the module productions in *M2* and the productions created by *M2*'s import of *ME*. With the additional imports to `X` and `B`, the grammar induced by {*MG*, *ME*, *MB*, *M2*, *M3*}, with *M3* as the main module, is

```
M3.S  ::=  if M3.B then M3.S  |  while M3.B do M3.S  |  id := M3.X
M3.X  ::=  M3.X - M3.X  |  M3.X + M3.X  |  0  |  1
M3.B  ::=  MB.X & MB.X  |  t  |  f
MB.X  ::=  MB.X & MB.X  |  t  |  f
```

4. Formal definition of the grammar induced by a GSM

In this section we give a formal definition of the raw CFG induced from a modularised grammar specification. We call this grammar the raw grammar because it takes no account of whitespace and comment layout considerations.

Notation For a string $\rho \in (\mathbb{T} \cup \mathbb{N})^*$ and $M \in \mathbb{M}$ define $M.\rho$ to be the string obtained from ρ by replacing each nonterminal A in ρ with $M.A$.

Definition A set Π of grammar module specifications over $(\mathbb{M}, \mathbb{N}, \mathbb{T})$, with start module M_0 and start symbol S, induces a *raw context free grammar* $\Gamma(\Pi, M_0, S)$ with start symbol $M_0.S$ and the set of grammar productions $\mathbf{Q}$ defined as follows.

First define $\mathbf{Q}'$ to be the smallest set of productions for which

1. if $(X ::= \rho) \in \mathbb{P}_M$, where $M \in \Pi$ then $(M.X ::= M.\rho) \in \mathbf{Q}'$,
2. if $(Y \leftarrow K.X) \in \mathbb{I}_M$, $(K.X ::= H.\rho) \in \mathbf{Q}'$, and $(Y :/= \rho) \notin \mathbb{D}_M$, where $M \in \Pi$, then $(M.Y ::= H.\rho) \in \mathbf{Q}'$,
3. if $(Y \Leftarrow K.X) \in \mathbb{I}_M$, $(K.X ::= H.\rho) \in \mathbf{Q}'$, where $H \neq K$, and $Y :/= \rho \notin \mathbb{D}_M$, where $M \in \Pi$, then $(M.Y ::= H.\rho) \in \mathbf{Q}'$.
4. if $(Y \Leftarrow K.X) \in \mathbb{I}_M$, $(K.X ::= K.\rho) \in \mathbf{Q}'$, and $Y :/= \rho \notin \mathbb{D}_M$, where $M \in \Pi$, then $(M.Y ::= M.\rho) \in \mathbf{Q}'$.

The set $\mathbf{Q}$ is obtained from $\mathbf{Q}'$ by removing all productions of the form $(H.Y ::= K.\rho)$ where $H \neq M_0$ and $H.Y$ is not reachable from $M_0.S$.

The production set $\mathbf{Q}$ may be constructed using the following algorithm, in which the set $\mathbf{R}$ is used to hold productions pending resolution.

1. *Initialisation*
   Set $\mathbf{Q}'$, $\mathbf{Q}$ and $\mathbf{R}$ to $\emptyset$
   for each $M \in \Pi$ and $(Y ::= M.\rho) \in \mathbb{P}_M$
   add an element $M.Y ::= M.\rho$ to $\mathbf{R}$
2. *Resolution*
   repeat
   remove $K.X ::= H.\rho$ from $\mathbf{R}$ and add it to $\mathbf{Q}'$
   for each $M \in \Pi$ and $Y \in \mathbb{N}$ such that $(Y :/= \rho) \notin \mathbb{D}_M$
   if $(Y \leftarrow K.X) \in \mathbb{I}_M$ and $(M.Y ::= H.\rho) \notin \mathbf{Q}'$
   add $M.Y ::= H.\rho$ to $\mathbf{R}$
   if $(Y \Leftarrow K.X) \in \mathbb{I}_M$
   if $H \neq K$ and $(M.Y ::= H.\rho) \notin \mathbf{Q}'$
   add $M.Y ::= H.\rho$ to $\mathbf{R}$
   if $H = K$ and $(M.Y ::= M.\rho) \notin \mathbf{Q}'$
   add $M.Y ::= M.\rho$ to $\mathbf{R}$
   until $\mathbf{R} = \emptyset$
3. *Reaching*
   for each $(M.Y ::= H.\rho) \in \mathbf{Q}'$
   if $M.Y$ is reachable from $M_0.S$ or if $M = M_0$
   add $M.Y ::= H.\rho$ to $\mathbf{Q}$

Example 4
The following example assumes that there exists a GSM module $M4$ which generates a grammar including a while-do statement and an expression subgrammar which enforces a particular operator associativity and priority. The aim is to use this as the basis for the specification of a related grammar in which the while-do statement is replaced with a do-until statement and the expression grammar is 'flattened', perhaps to simplify semantic specification. A deleter is used to remove the while-do statement. As the whole expression subgrammar is to be changed the productions for E are not imported, they are defined anew.

```
M5                          M4
S <= M4.S                   S ::= if B then S  |
B <= M4.B                         while B do S  |  id := E
S :/= while B do S          E ::= E + T | T
S ::= do S until B          T ::= T * F | F
E ::= E + E |               F ::= 0 | 1
       E * E | 0 | 1        B ::= t | f
```

The module $M5$ specifies, after unreachable productions are removed, the desired modified grammar.

```
M5.S ::= if M5.B then M5.S  |  id := M5.E  |  do M5.S until M5.B
M5.B ::= t | f
M5.E ::= M5.E + M5.E | M5.E * M5.E | 0 | 1
```

Notice that we could not have achieved the same result using import-by-reference. The specification

```
M6                          M4
S <- M4.S                   S ::= if B then S  |
B <- M4.B                         while B do S  |  id := E
S :/= while B do S          E ::= E + T | T
S ::= do S until B          T ::= T * F | F
E ::= E + E |               F ::= 0 | 1
       E * E | 0 | 1        B ::= t | f
```

specifies the grammar

```
M6.S ::= if M4.B then M4.S  |  id := M4.E  |  do M6.S until M6.B
M6.B ::= t | f
M6.E ::= M6.E + M6.E | M6.E * M6.E | 0 | 1
```

```
M4.S ::= if M4.B then M4.S  |  while M4.B do M4.S  |  id := M4.E
M4.E ::= M4.E + M4.T | M4.T
M4.T ::= M4.T * M4.F | M4.F
M4.F ::= 0 | 1
M4.B ::= t | f
```

This does not have the required effect, the do-until statement does not nest inside the if-statement and the while-do statement can still appear nested inside a do-until or if statement.

5. Recursive import-by-clone

In general the import-by-clone mechanism will create new non-terminals, for which grammar productions will need to be specified. The generality of the mechanism allows the user to specify such productions. For example, in the MGS $\{M5, M4\}$ in the previous section, the productions for `M5.E` are defined in $M5$. However, it is likely in many cases that the specifier will want the productions for the new nonterminals to include those from the original module, as is the case for `M5.B` in the above example. Of course, the user can achieve this by explicitly writing out the required import rules, as in nonterminal $M3$ in Example 3 above. However, we would expect a concrete modularity implementation to provide an automatic import rule creation mechanism. We now describe such a mechanism, $\Leftarrow^*$.

We modify the definition of $\mathbb{I}_M$ in a GSM M as follows:

- a finite set $\mathbb{I}_M$ of *import by reference expressions* of the form $X \leftarrow H.Y$, *import by clone expressions* of the form $X \Leftarrow H.Y$, where $X, Y \in \mathbb{N}$, $H \in \mathbb{M}$, and $H \neq M$, and *recursive import by clone expressions* of the form $X \Leftarrow^* H.Y$, where $X, Y \in \mathbb{N}$, $H \in \mathbb{M}$, and $H \neq M$,

We extend the definition of the grammar defined by a set of GMSs so that, conceptually, as the recursive import-by-clone expressions are dealt with, new import-by-clone expressions are created for the nonterminals on the right hand sides of productions.

Formally we define the set of grammar productions $\mathbf{Q}$ as follows.

First define $\mathbf{Q}'$ and $\mathbb{I}'_M$ to be the smallest sets of productions and import expressions for which

1. if $(X ::= \rho) \in \mathbb{P}_M$, where $M \in \Pi$ then $(M.X ::= M.\rho) \in \mathbf{Q}'$, and $\mathbb{I}_M \subseteq \mathbb{I}'_M$, where $M \in \Pi$,

2. if $(Y \leftarrow K.X) \in \mathbb{I}_M$, $(K.X ::= H.\rho) \in \mathbf{Q}'$, and $(Y :/= \rho) \notin \mathbb{D}_M$, where $M \in \Pi$, then $(M.Y ::= H.\rho) \in \mathbf{Q}'$,
3. if $(Y \Leftarrow K.X) \in \mathbb{I}_M$, $(K.X ::= H.\rho) \in \mathbf{Q}'$, and $Y :/= \rho \notin \mathbb{D}_M$, where $M \in \Pi$, then $(M.Y ::= M.\rho) \in \mathbf{Q}'$, if $H = K$ and $(M.Y ::= H.\rho) \in \mathbf{Q}'$, if $H \neq K$.
4. if $(Y \Leftarrow^* K.X) \in \mathbb{I}'_M$, $(K.X ::= K.\rho) \in \mathbf{Q}'$, and $Y :/= \rho \notin \mathbb{D}_M$, where $M \in \Pi$, then $(M.Y ::= M.\rho) \in \mathbf{Q}'$, and $(Z \Leftarrow^* K.Z) \in \mathbb{I}'_M$, for each Z in ρ.
5. if $(Y \Leftarrow^* K.X) \in \mathbb{I}'_M$, $(K.X ::= H.\rho) \in \mathbf{Q}'$, where $H \neq K$, and $Y :/= \rho \notin \mathbb{D}_M$, where $M \in \Pi$, then $(M.Y ::= H.\rho) \in \mathbf{Q}'$.

The set $\mathbf{Q}$ is obtained from $\mathbf{Q}'$ by removing all productions of the form $(H.Y ::= K.\rho)$ where $H \neq M_0$ and $H.Y$ is not reachable from $M_0.S$.

The following set of modules, with main module $M3$, specify the same grammar as that in Example 3 above.

```
M3                              M2
S <=* M2.S                      X <= ME1.X
S ::= if B then S               B <- MB.X
                                S ::= while B do S  |  id := X
                                X ::= X - X

ME1                             MB
X ::= X + X  |  0  |  1         X ::= X & X  |  t  |  f
```

Notice that imports of the form $M.A \Leftarrow^* N.B$ only impact on productions of the form $N.B ::= N.\gamma$. In the above example, the import $M3.S \Leftarrow^* M2.S$ creates imports $M3.X \Leftarrow^* M2.X$ and $M3.B \Leftarrow^* M2.B$. However, because the import to B in $M2$ is by reference, the productions form $M2.B$ are of all the form $M2.B ::= MB.\gamma$ and so the recursive by clone import terminates at this point.

6. Automatic whitespace insertion

As we have already mentioned in Section 1, compilers often include an initial lexical phase in which an input sequence of characters is transformed into a sequence of grammar terminals. Each terminal $t \in \mathbb{T}$ has an associated set of strings of characters, its *pattern*. The strings in the pattern of a terminal are called its *lexemes*. Typically the character set is a standard character set such as ASCII or Unicode but in principle it could be, for

instance, a collection of pixels, a sequence of mouse actions, or some output from a voice recognition system.

The patterns are usually regular languages over the set of characters and thus are commonly specified using regular expressions. For some applications this separation between the character strings and the grammar is too restrictive. The core problem is the situation in which the patterns of two different terminals have lexemes in common. For example, in a language which is the union of two languages such as Cobol and SQL, a lexeme may belong to a keyword terminal in one part of the grammar and the identifier terminal in the other. One solution is do away with the separate lexical phase, make the character set the terminals of the grammar and use grammar productions to specify the patterns. The terminals of the original grammar become nonterminals in the new grammar, which we shall refer to as a character level grammar. This is the approach taken in SDF [11, 22] and a detailed discussion can be found in [23].

When a separate lexical phase is employed, it is used to resolve certain ambiguities and to remove layout and comments. A character level grammar thus has to include whitespace characters in its terminal set, and the positions in the grammar productions where whitespace is allowed have to have an instance of a nonterminal which generates the corresponding whitespace language. Adding these instances by hand is tedious and makes the grammar hard to read. Our modularised grammar specification includes an automatic whitespace insertion mechanism, specified by the GSM whitespace conventions.

6.1. Positioning whitespace nonterminals

If M has whitespace convention $(W_M, L.V) = \mathbb{W}_M$ we call W_M the *whitespace nonterminal of* M and $L.V$ the *imported whitespace nonterminal* of M. Automatic whitespace insertion is the insertion of nonterminals $M.W_M$ into the induced grammar rules. The assumption is that W_M derives the desired language of whitespace elements. The instances of W_M are left out of the GSM and automatically inserted into the grammar during the resolution process. Before formally defining our whitespace insertion mechanism we look at some examples.

Example 7
The following simple example illustrates the basic approach. We have $\mathbb{W}_{M7} = (W, MW1.X)$, the terminals of $MW1$ are enclosed in single quotes to make them clear and comment bodies are reduced to strings of x's and y's for simplicity.

```
M7
 (W, MW1.X)
 S ::= a S b | c

MW1
 X ::= T X | ϵ
 T ::= '\t' | '\nl' | ' ' | '/*' Z '*/'
 Z ::= 'y' Z | 'x' Z | 'y' | 'x'
```

The grammar specified by $M7$ is

```
 M7.S' ::= M7.W M7.S
 M7.S ::= a M7.W M7.S b M7.W  |  c M7.W
 M7.W ::= MW1.T MW1.X | ϵ
 MW1.X ::= MW1.T MW1.X | ϵ
 MW1.T ::= '\t' | '\nl' | ' ' | '/*' MW1.Z '*/'
 MW1.Z ::= 'y' MW1.Z | 'x' MW1.Z | 'y' | 'x'
```

Here `M7.S'` is a new augmented start symbol introduced to permit whitespace at the start of an input string. Notice there is no instance of `M7.W` immediately to the right of `M7.S` in the second production. This is not required and if inserted would create unnecessary ambiguity.

Example 8
In this example we consider a module which has an import by reference from module $M7$ of Example 7 but which uses a slightly different whitespace convention, specified in module $MW2$.

```
M8
 (V, MW2.X)
 S <- M7.S
 A ::= d A | d | S

MW2
 X ::= T X | ϵ
 T ::= '\nl' | ' ' | '//' Z '\nl'
 Z ::= 'y' Z | 'q' Z | 'y' | 'q'
```

The grammar specified by $M8$ has productions

```
 M8.A' ::= M8.V M8.A
 M8.A ::= d M8.V M8.A  |  d M8.V  |  M8.S
 M8.S ::= M7.W a M7.W M7.S b M7.W M8.V  |  M7.W c M7.W M8.V
 M8.V ::= MW2.T MW2.X | ϵ
```

```
M7.S ::= a M7.W M7.S b M7.W  |  c M7.W
M7.W ::= MW1.T MW1.X | ϵ
```

together with the productions for `MW2.V` and `MW1.W`, which are omitted here.

The effect is that substrings derived from `M7.S` will have whitespace and commenting styles defined by $M7$. Instances of `d` in an input string can be surrounded by whitespace and comment styles defined by $MW2$.

Example 9
The case of a module containing an import by clone from module $M7$ is more interesting. Consider the following module $M9$.

```
M9
 (V, MW2.X)
 S <= M7.S
 A ::= d A | d | S
```

The grammar specified by $M9$ has productions

```
M9.A' ::= M9.V M9.A
M9.A ::= d M9.V M9.A  |  d M9.V  |  M9.S
M9.V ::= MW2.T MW2.X | ϵ
M9.S ::= M9.W a M9.W M9.S b M9.W M9.V  |  M9.W c M9.W M9.V
```

together with the productions for `MW2.V`.

The import-by-clone creates a new nonterminal `M9.W` which, in this case, has no production rules and will thus not generate any sentences. The writer of $M9$ would be expected to specify these rules. If, for example, they want the whitespace associated with the sublanguage defined by `M9.S` to be that of $M9$ they can add a production `W ::= V` or the import `W <- MW2.X` to $M9$. If they want the whitespace associated with the sublanguage defined by `M9.S` to be that of $M7$ they can add an import `W<-M7.W` to $M9$. If they want the union of both conventions then they can add both the production and the import.

We now describe formally the mechanisms that we used implicitly in the above examples.

We say that two modules K, H have *equivalent whitespace conventions*, written $\mathbb{W}_K \equiv \mathbb{W}_H$, if they both have the same imported whitespace nonterminal or if they both have empty whitespace import, and if W_K (W_H) does not appear on the left hand side of any element in $\mathbb{I}_K$ ($\mathbb{I}_H$) or $\mathbb{P}_K$ ($\mathbb{P}_H$).

In general we want to allow optional whitespace, thus, if W_M is not nullable, we create a new nonterminal W'_M whose productions are $W'_M ::= \epsilon \mid W_M$ and insert W'_M. To avoid the clerical overhead of separate special

cases, if the whitespace convention for M is empty we define $W'_M = \epsilon$ and $K.W'_M = \epsilon$, for all $K \in \mathbb{M}$, and, if W_M is nullable, W'_M is defined to be W_M.

Naïve automatic whitespace insertion processes can generate significant ambiguity in the grammar. In [14] we gave an extensive discussion of the alternative approaches and their disadvantages. We shall not repeat that discussion here, we shall simply use the approach arrived at from that discussion.

- Before any grammar rules are constructed, for each GSM, M, which has a whitespace convention, for each module production in $\mathbb{P}_M$, W'_M is inserted after each terminal instance, e.g.
  $M.A ::= a\ M.W'_M\ A\ b\ M.W'_M$,

- For imports $A \leftarrow K.B$ and $A \Leftarrow K.B$ in $\mathbb{I}_M$, if M has a whitespace convention and the whitespace conventions of K and M are not equivalent, then $M.W'_M$ is added at the end of each alternate of $M.A$ that is imported from $K.B$.

- For imports $A \leftarrow K.B$ in $\mathbb{I}_M$, if K has a whitespace convention and $\mathbb{W}_K \not\equiv \mathbb{W}_M$, then $K.W'_K$ is added at the start of each alternate of $M.A$ that is imported from $K.B$.

- For imports $A \Leftarrow K.B$ in $\mathbb{I}_M$, if K has a whitespace convention and $\mathbb{W}_K \not\equiv \mathbb{W}_M$, then $M.W'_K$ is added at the start of each alternate of $M.A$ that was imported from $K.B$.

- If the start module M_0 has a whitespace convention, a new augmented start rule $S' ::= W'_{M_0}\ S$ is created, where S is the original start symbol.

6.2. Grammar induction with whitespace insertion

We now formally define the grammar specified by a MGS with whitespace with non-empty whitespace conventions.

Because alternates will be imported with inserted whitespace nonterminals cyclic import dependencies will result in the lengths of alternates increasing without limit. Thus we place a non-cyclic condition on the modularised grammar specification.

A set Π of GSMs displays *whitespace sensitive cyclic module dependency* if there is a sequence of elements $K_1, \ldots, K_n, K_{n+1} = K_1$ from Π, with $\mathbb{W}_{K_j} \not\equiv \mathbb{W}_{K_p}$ for some j, p, but for which there exists imports $X_i \leftarrow K_{i+1}.X_{i+1}$ or$X_i \Leftarrow K_{i+1}.X_{i+1}$ in $\mathbb{I}_{K_i}$, $1 \leq i \leq n$.

For a string $\tau \in (\mathbb{T} \cup \mathbb{M}.\mathbb{N})^*$ define $\overline{\tau}$ to be the string in $(\mathbb{T} \cup \mathbb{N})^*$ obtained by replacing nonterminals of the form $H.A$ with A and then removing

instances of the nonterminals W'_M from τ. Define $\tau_{K/H}$ to be the string obtained by replacing each nonterminal, including whitespace nonterminals, of the form $H.A$ with $K.A$

For a string $\rho \in (\mathbb{T} \cup \mathbb{N})^*$ and $K \in \mathbb{M}$ with nonempty whitespace convention, define ρ_K to be the string in $(\mathbb{T} \cup \mathbb{N})^*$ obtained by inserting an instance of W'_K, after each instance of each element of $\mathbb{T}$ in ρ. If K has empty whitespace convention, $\rho_K = \rho$. Also, as above, define $K.\rho$ to be the string obtained from ρ by replacing each nonterminal A in ρ, including W_K, with $K.A$.

Definition A set Π of grammar module specifications over $(\mathbb{M}, \mathbb{N}, \mathbb{T})$, with start module M_0 and start symbol S, which does not display whitespace sensitive cyclic modular dependency, induces a *whitespace expanded context free grammar* $\Gamma(\Pi', M_0, S)$ with start symbol $M_0.S'$ and the set of grammar productions $\mathbf{Q}$ defined as follows.

First define $\mathbf{Q'}$ to be the smallest set of productions for which

1. if $(X ::= \rho) \in \mathbb{P}_M$, where $M \in \Pi$, then $(M.X ::= M.\rho_M) \in \mathbf{Q'}$,
2. if $(Y \leftarrow K.X) \in \mathbb{I}_M$, $(K.X ::= \tau) \in \mathbf{Q'}$, and $(Y :/= \overline{\tau}) \notin \mathbb{D}_M$, where $M \in \Pi$, then
   (a) $(M.Y ::= \tau) \in \mathbf{Q'}$, if $\mathbb{W}_M \equiv \mathbb{W}_K$,
   (b) $(M.Y ::= K.W'_K \tau M.W'_M) \in \mathbf{Q'}$ if $\mathbb{W}_M \not\equiv \mathbb{W}_K$,
3. if $(Y \Leftarrow K.X) \in \mathbb{I}_M$, $(K.X ::= \tau) \in \mathbf{Q'}$, and $(Y :/= \overline{\tau}) \notin \mathbb{D}_M$, where $M \in \mathbb{M}$, then
   (a) $(M.Y ::= \tau_{M/K}) \in \mathbf{Q'}$, if $\mathbb{W}_M \equiv \mathbb{W}_K$,
   (b) $(M.Y ::= M.W'_K \tau_{M/K} M.W'_M) \in \mathbf{Q'}$ if $\mathbb{W}_M \not\equiv \mathbb{W}_K$,
4. for each $M \in \Pi$ such that M has a nonempty whitespace convention $(W_M, L.Y)$, if $(L.Y ::= \gamma) \in \mathbf{Q'}$ then $(M.W_M ::= \gamma) \in \mathbf{Q'}$.

The set $\mathbf{Q}$ is obtained from $\mathbf{Q'}$ by removing all productions of the form $(H.Y ::= \tau)$ where $H \neq M_0$ and $H.Y$ is not reachable from $M_0.S$, adding $W'_M ::= W_M | \epsilon$ for each non-nullable whitespace nonterminal W_M, and finally, if W_{M_0} is not empty, adding a new production $S' ::= W'_{M_0}\ S$, where S' is a new nonterminal. If $\mathbb{W}_{M_0} = \emptyset$ then we define $S' = S$.

We can give an algorithm for computing $\mathbf{Q}$ as follows.

1. *Initialisation and whitespace after terminals*
   Set $\mathbf{Q'} = \mathbf{Q} = \mathbf{R} = \emptyset$
   for each $M \in \Pi$ with nonempty $\mathbb{W}_M = (W_M, L.V)$
      if W_M is not nullable add $W'_M ::= \epsilon \mid W_M$ to $\mathbf{Q}$
   for each $(X \to \rho) \in \mathbb{P}_M$, where $M \in \Pi$

add $M.X ::= M.\rho_M$ to $\mathbf{R}$,

2. *Resolution*

repeat

remove $K.X ::= \tau$ from $\mathbf{R}$ and add it to $\mathbf{Q}'$

for each $M \in \Pi$ and $(Y \leftarrow K.X) \in \mathbb{I}_M$ such that $Y :/= \overline{\tau} \notin \mathbb{D}_M$

if $\mathbb{W}_K \equiv \mathbb{W}_M$ and $M.Y ::= \tau \notin \mathbf{Q}'$

add $M.Y ::= \tau$ to $\mathbf{R}$

else add $M.Y ::= K.W'_K \tau M.W'_M$ to $\mathbf{R}$

for each $M \in \Pi$ and $(Y \Leftarrow K.X) \in \mathbb{I}_M$ such that $Y :/= \overline{\tau} \notin \mathbb{D}_M$

if $\mathbb{W}_K \equiv \mathbb{W}_M$ and $M.Y ::= \tau_{M/K} \notin \mathbf{Q}'$

add $M.Y ::= \tau_{M/K}$ to $\mathbf{R}$

else add $M.Y ::= M.W'_K \tau_{M/K} M.W'_M$ to $\mathbf{R}$

for each nonempty whitespace convention $(W_M, K.X)$, $M \in \Pi$

if $(M.W_M ::= \tau) \notin \mathbf{Q}'$ add $M.W_M ::= \tau$ to $\mathbf{R}$

until $\mathbf{R} = \emptyset$

3. *Reaching*

for each $(M.Y ::= \tau) \in \mathbf{Q}'$

if $M.Y$ is reachable from $M_0.S$ or $M = M_0$

add $M.Y ::= \tau$ to $\mathbf{Q}$

4. *Initial whitespace*

if $\mathbb{W}_{M_0}$ is not empty

let S' be a new nonterminal not in

$\mathbb{N} \cup \mathbb{T} \cup \mathbb{M}$ and add $S' ::= W'_{M_0}\ S$ to $\mathbf{Q}$

else set $S' = S$

6.3. Whitespace good practice

We have deliberately made our whitespace specification mechanism liberal in order to permit a wide variety of applications. For example, it is possible for specifiers to write module production rules whose left hand side is the whitespace nonterminal. As written, deleters will not be applied to whitespace productions, but this can easily be changed if experience shows it to be desirable. It is also possible for modules from which whitespace is imported to themselves have a whitespace convention. The theoretical underpinnings given in this paper ensure that such uses will be robust and their behaviours can be formally established. However, we believe that using such features will make the specification hard to reason about in practice.

We think it is good practice that whitespace be defined in a separate module which itself has an empty whitespace convention, and writing a production or import statement in M whose left hand side is W_M (or W'_M)

is deprecated. The latter is important for ease of reasoning about whitespace equivalence.

We have noted that for import-by-clone the specifier has significant control over the associated whitespace convention. However, with import-by-reference the whitespace convention associated with the imported productions will be that associated with the source module. It is possible to impose a different whitespace convention in this case by using an intermediate module.

For example, consider the MGS $\{M12, M11, MW3, MW4\}$

```
M12                              M11
(W2, MW4.Y)                      (W1, MW3.Y)
E <- M11.S                        S ::= S + S  |  1  |  0
S ::= id := E

MW3                              MW4
Y ::= '\t' Y | ϵ                 Y ::= '\nl' Y | ' ' Y | ϵ
```

The sublanguage specified by `M12.E` has the whitespace from $MW3$. To use the whitespace from $MW4$ but retain $\leftarrow$ to avoid name clashing we can use an intermediate module $M13$ to change the whitespace convention, then import this to $M12$

```
M12                     M13
(W2, MW4.Y)             S <= M11.S
E <- M13.S               W1 <=* MW4.Y
S ::= id := E
```

Finally we note that a user who wishes to parse a token stream generated by some external lexer can define the MGS at token level and leave the whitespace conventions empty. In this case none of the mechanisms described in this section will be invoked.

7. Modularity in use

In this section we consider several use cases. We take as our base examples two small languages C and P which display some of the syntactic characteristics of the ANSI-C and ISO-Pascal languages, respectively. Specifically:

1. C is completely case sensitive (including keywords like `while`); P is mostly case sensitive but keywords are case-insensitive so `BeGiN` and `begin` are equivalent. Type names in P are identifiers, not keywords, so they are case sensitive.

2. C uses ; as a statement terminator whereas P uses ; as a statement separator.
3. C comments are line oriented and introduced by //; P has nestable comments delimited by { and }.
4. C identifiers may begin with an underscore; P identifiers may not. Underscores are allowed within both languages' identifiers.

We present the examples using the concrete syntax for our GLL parser generator ART, version 2.6. ART specifications are a sequence of explicit productions and *module headers*. The grammar induction process in general creates new (non-explicit) productions.

A module header comprises an identifier, which is the module name, followed by an optional whitespace import, a parenthesised list of import statements and an optional parenthesized start symbol:

```
P w<-PLex.ws (PLex.intLiteral PLex.id)(program)
```

The syntax of imports is an ASCII version of the abstract syntax used in the rest of this paper with some shorthands: a module header of the form `M(L.Y)` is shorthand for `M(Y<-L.Y)` and a header of the form `M(<=L.Y)` is shorthand for `M(Y<=L.Y)`.

An explicit production comprises an identifier, the name of a nonterminal, followed by `::=` ρ where ρ is a regular expression over the terminals and nonterminals. In this paper, we restrict our examples to simple BNF. The basic terminal designation in ART is `` `a ``, that is a single character preceded by a back-quote. Non-printing characters are represented using ANSI-C like escape sequences. We also provide case-sensitive terminals `'thus'` which is a monolithic shorthand for `` `t`h`u`s ``, and case-insensitive terminals `"thus"` which is a shorthand for `(`t|`T)(`h|`H)(`u|`U)(`s|`S)`. These shorthands are monolithic in the sense that automatic whitespace insertion only occurs after instances; no whitespace is inserted between the individual characters of a `''` or `""` style terminal.

The token level syntax of P is specified by:

```
P w<-PLex.ws (PLex.intLiteral PLex.id)(program)

program ::= "program" id "begin" statDecs "end" ".";
statDecs ::= statDec | statDec ';' statDec ;
statDec ::=  statement | declaration ;

statement ::= "repeat" statement "until" expression |
              "if" expression "then" statement |
              id ':=' expression ;

declaration ::= "var" id ':' 'int' ;

expression ::= term | expression '+' term | expression '-' term ;
term ::= factor | term '*' factor | term '/' factor ;
factor ::= intLiteral | id | '(' expression ')' ;
```

Note the use of `""` terminals for keywords, and `'int'` for the type name, which is a case-sensitive identifier. As is traditional in programming language descriptions, the character level lexical rules are omitted at this level; simply being imported from a separate module. The whitespace convention is specified as `P.w`.

The lexical rules are as follows: ART requires no specific lexical section; we simply use a module without a whitespace convention.

```
PLex()
intLiteral ::= digit | digit intLiteral ;

id ::= alpha idTail ;
idTail ::= alphaNumU | alphaNumU idTail ;

alpha ::= `a .. `z | `A .. `Z ;
digit ::= `0 .. `9;
alphaNumU ::= alpha | digit | `_;

ws ::= wsElement | wsElement ws;
wsElement ::= ` | `\t | comment;
comment ::= `{ commentBody `};
commentBody ::= / commentDelim | comment;
commentDelim ::= `{ | `};
```

Internally, a production whose alternates are individual characters (such as `commentDelim` above) is represented as a character set and we provide some shorthands which are valid for character level operands — the expression `` `a .. `z `` is shorthand for `` `a | `b | ... | `z ``; the `|` operator between character sets naturally forms a character set containing the union of the operands; and a set-difference expression `a / b` forms a character set containing those elements that are in `a` but not in `b`. The monadic set difference expression `/ commentDelim` specifies the set with every element from the character-level alphabet except for those in set `commentDelim`.

The primary goal of our modularity constructs is to allow the user to write grammar specifications in a conventional style, but for the tools to display an exact character level context free grammar including all details of whitespace handling. This is the grammar induced from `P()` and `PLex()`:

```
P._program ::= P.w P.program ;

P.declaration ::= "var" P.w P.id ':' P.w 'int' P.w ;
P.expression ::= P.term | P.expression '+' P.w P.term | P.expression '-' P.w P.term;
P.factor ::= P.intLiteral | P.id | '(' P.w P.expression ')' P.w ;
P.id ::= PLex.alpha PLex.idTail P.w ;
P.intLiteral ::= PLex.digit P.w | PLex.digit PLex.intLiteral P.w ;
P.program ::= "program" P.w P.id "begin" P.w P.statDecs "end" P.w "." P.w ;
P.statDec ::= P.statement | P.declaration ;
P.statDecs ::= P.statDec | P.statDec ';' P.w P.statDec ;
P.statement ::= "repeat" P.w P.statement "until" P.w P.expression |
                "if" P.w P.expression "then" P.w P.statement |
                P.id ':=' P.w P.expression ;
P.term ::= P.factor | P.term '*' P.w P.factor | P.term '/' P.w P.factor ;

P.w ::= PLex.wsElement | PLex.wsElement PLex.ws | # ;

PLex.alpha ::= `a..`z | `A..`Z ;
PLex.alphaNumU ::= PLex.alpha | PLex.digit | `_ ;
PLex.comment ::= `{ PLex.commentBody `} ;
PLex.commentBody ::= / PLex.commentDelim | PLex.comment ;
PLex.commentDelim ::= `{ | `} ;
PLex.digit ::= `0..`9 ;
PLex.id ::= PLex.alpha PLex.idTail ;
PLex.idTail ::= PLex.alphaNumU | PLex.alphaNumU PLex.idTail ;
PLex.intLiteral ::= PLex.digit | PLex.digit PLex.intLiteral ;
PLex.ws ::= PLex.wsElement | PLex.wsElement PLex.ws ;
PLex.wsElement ::= `  | `\t | PLex.comment ;
```

The nonterminals `P.w, P.id and P.intLiteral` form the interface between the token level grammar and the lexical specification. Note how the rules for `P.id` and `P.intLiteral` are appended with an instance of `P.w` so that whitespace is automatically consumed by a parser whenever an identifier or literal is matched. An instance of `P.w` is also inserted after each explicit token, and a new augmented start symbol is created which matches any initial whitespace in the input string. Note also that the whitespace nonterminal `P.w` has automatically had an epsilon rule added to it. (In ART `#` denotes the empty string, which is denoted by ϵ in the theoretical sections.)

The induced grammar makes use of the `..` and `/` operators for conciseness, but `PLex.alpha`, `PLex.digit` and `PLex.commentBody` can be easily rendered in expanded form if a pure BNF grammar is preferred.

The token level and lexical grammars for our example `C` language follow the general pattern of that for `P`; in particular the expression grammars are

identical except for the new % operator.

```
C w<-CLex.ws (CLex.intLiteral CLex.id)(translationUnit)

translationUnit ::= statDecs ;
statDecs ::= statement | declaration | statement statDecs | declaration statDecs;

statement ::= 'while' '(' expression ')' statement ';' |
              'if' '(' expression ')' statement |
              id '=' expression ;

declaration ::= 'int' decName ;
decName ::= id | id optInitialiser ;
optInitialiser ::= '=' expression ;

expression ::= term | expression '+' term | expression '-' term ;
term ::= factor | term '*' factor | term '%' factor | term '/' factor ;
factor ::= intLiteral | id | '(' expression ')' ;
```

At lexical level, identifiers allow a leading underscore, and the commenting convention is changed.

```
CLex()
intLiteral ::= digit | digit intLiteral ;
id ::= alphaNumU idTail ;                    (* Different to PLex.id *)
idTail ::= alphaNumU | alphaNumU idTail ;

alpha ::= `a .. `z | `A .. `Z ;
digit ::= `0 .. `9;

alphaNumU ::= alpha | digit | `_;

ws ::= wsElement | wsElement ws;
wsElement ::= ` | `\t | comment;
comment ::= `/ `/ commentBody `\n;           (* Different to PLex.id *)
commentBody ::= / `\n |  / `\n commentBody;
```

7.1. Embedding

Language embedding is straightforward because the automatic whitespace handling takes care of synchronising the lexical level tokens. Consider the simplest case: the embedding of an entire P program at statement level within C. If we add the import `statement<-P.program` to the module header for `C` then one extra production is induced:

```
C.statement ::= P.w "program" P.w P.id "begin" P.w P.statDecs "end" P.w "." P.w C.w ;
```

The full grammar for `C` with embedded `P` is the concatenation of the induced `P` grammar above and:

```
C._translationUnit ::= C.w C.translationUnit ;
C.decName ::= C.id | C.id C.optInitialiser ;
C.declaration ::= 'int' C.w C.decName ;
C.expression ::= C.term | C.expression '+' C.w C.term | C.expression '-' C.w C.term;
C.factor ::= C.intLiteral | C.id | '(' C.w C.expression ')' C.w ;
C.id ::= CLex.alphaNumU CLex.idTail C.w ;
C.intLiteral ::= CLex.digit C.w | CLex.digit CLex.intLiteral C.w ;
C.optInitialiser ::= '=' C.w C.expression ;
C.statDecs ::=  C.statement | C.declaration |
                C.statement C.statDecs | C.declaration C.statDecs ;
C.statement ::= 'while' C.w '(' C.w C.expression ')' C.w C.statement ';' C.w |
                'if' C.w '(' C.w C.expression ')' C.w C.statement |
                C.id '=' C.w C.expression |
                P.w "program" P.w P.id "begin" P.w P.statDecs "end" P.w "." P.w C.w;
C.term ::= C.factor | C.term '*' C.w C.factor |
           C.term '%' C.w C.factor | C.term '/' C.w C.factor ;
C.translationUnit ::= C.statDecs ;

C.w ::= CLex.wsElement | CLex.wsElement CLex.ws | # ;

CLex.alpha ::= `a..`z | `A..`Z ;
CLex.alphaNumU ::= CLex.alpha | CLex.digit | `_ ;
CLex.comment ::= `/ `/ CLex.commentBody `\n ;
CLex.commentBody ::= / `\n | / `\n CLex.commentBody ;
CLex.digit ::= `0..`9 ;
CLex.id ::= CLex.alphaNumU CLex.idTail ;
CLex.idTail ::= CLex.alphaNumU |  CLex.alphaNumU CLex.idTail ;
CLex.intLiteral ::= CLex.digit | CLex.digit CLex.intLiteral ;
CLex.ws ::= CLex.wsElement | CLex.wsElement CLex.ws ;
CLex.wsElement ::= `  | `\t | CLex.comment ;
```

Of course, there will almost always be some ambiguity at the boundary between languages since nonprinting characters such as SPACE and TAB will be in both `P.w` and `C.w` and so the consecutive instances of nullable nonterminals `P.w` and `C.w` will both match any whitespace following the embedded P program; typically lexical level ambiguities are resolved using longest match.

In practice, embedding an entire program is unlikely to be required. More usually, the statements, declarations and expressions will all individually be intermingled with separate import expressions. As described earlier, this problem area is also addressed by Van Wyk and co-workers in the Copper parser generator [19].

7.2. Writing modules for reuse

The following module is an extract from the `C` module which only describes expressions at token level.

```
Expr ew<-Expr.ws ()
expression ::= term | expression '+' term | expression '-' term ;
term ::= factor | term '*' factor | term '%' factor | term '/' factor ;
factor ::= intLiteral | id | '(' expression ')' ;
```

The nonterminals `Expr.intLiteral`, `Expr.id` and `Expr.w` are all undefined.

If we import from this module using the by-clone import operator `<=`, then all nonterminals will be rewritten so that their module names match the importing module. There is a sense in which this module is 'parameterised' via these three undefined nonterminals which become hooks by which the importing module can change the language of expressions insofar as factors and whitespace are concerned.

Here is a modified version of `P` which imports its expression sub-language from the above module.

```
P w<-PLex.ws (<=Expr.expression <=Expr.term <=Expr.factor
              <=Expr.intLiteral <=Expr.id
              PLex.intLiteral PLex.id )(program)

program ::= "program" id "begin" statDecs "end" ".";
statDecs ::= statDec | statDec ';' statDec ;
statDec ::=  statement | declaration ;

statement ::= "repeat" statement "until" expression |
              "if" expression "then" statement |
              id ':=' expression ;

declaration ::= "var" id ':' 'int' ;
```

This induces (amongst others) the following rules

```
P.expression ::=
   P.ew P.term P.w |
   P.ew P.expression '+' P.ew P.term P.w |
   P.ew P.expression '-' P.ew P.term P.w ;

P.factor ::=
   P.ew P.intLiteral P.w |
   P.ew P.id P.w |
   P.ew '(' P.ew P.expression ')' P.ew P.w ;

P.term ::=
   P.ew P.factor P.w |
   P.ew P.term '*' P.ew P.factor P.w |
   P.ew P.term '%' P.ew P.factor P.w |
   P.ew P.term '/' P.ew P.factor P.w ;
```

The `P.ew` nonterminal has no productions defined. A unit rule `P.ew ::= P.w` serves to connect the whitespace nonterminals but at the cost of some extra ambiguity, and thus an increase in the size of parse forests.

7.3. Decomposition and recomposition

One area in which grammarware differs significantly from software is that we may wish to take a monolithic grammar and extract from it reusable portions. Of course, the traditional tool for this is the text editor, but our goal is to use the modularity constructs to document and specify the transformations so that we can reason about the resulting grammars and languages.

The expression sub-grammars for `C` and `P` are quite similar. Imagine that C pre-exists and we are building P *ab initio*. In the previous example we used a pre-written expression module. If we replace that with the following module header

```
Expr (<=C.expression <= C.term <=C.factor)
```

then the productions for `Expr` will be induced from the C grammar. Since there is no whitespace convention specified for `Expr` we get the whitespace from module `C` renamed to `Expr.w`.

```
Expr.expression ::=
   Expr.w Expr.term |
   Expr.w Expr.expression '+' Expr.w Expr.term |
   Expr.w Expr.expression '-' Expr.w Expr.term ;

Expr.factor ::=
   Expr.w Expr.intLiteral |
   Expr.w Expr.id |
   Expr.w '(' Expr.w Expr.expression ')' Expr.w ;

Expr.term ::=
   Expr.w Expr.factor |
   Expr.w Expr.term '*' Expr.w Expr.factor |
   Expr.w Expr.term '%' Expr.w Expr.factor |
   Expr.w Expr.term '/' Expr.w Expr.factor ;

P.expression ::=
   P.w P.term P.w |
   P.w P.expression '+' P.w P.term P.w |
   P.w P.expression '-' P.w P.term P.w ;

P.factor ::=
   P.w P.intLiteral P.w |
   P.w P.id P.w |
   P.w '(' P.w P.expression ')' P.w P.w ;


P.term ::=
   P.w P.factor P.w |
   P.w P.term '*' P.w P.factor P.w |
   P.w P.term '%' P.w P.factor P.w |
   P.w P.term '/' P.w P.factor P.w ;
```

The productions for `term` in C should not include a `%` operator. We can suppress that from the `Expr module` by adding the deleter

```
term :/= term '%' factor ;
```

7.4. Extension

Extension of non-recursive rules using import-by-reference is straightforward. For recursive rules, we usually want to use import-by-clone so that the right hand side instances refer to nonterminals in the importing module.

Consider this specification:

```
NewExpr(Expr)(expression)
term ::= term '%' factor;

Expr(Lex.intLiteral Lex.id)
expression ::= term | expression '+' term | expression '-' term ;
term ::= factor | term '*' factor | term '/' factor ;
factor ::= intLiteral | id | '(' expression ')' ;
```

`NewExpr` imports all of the nonterminals in `Expr` by reference, yielding the following productions for `NewExpr.term`.

```
NewExpr.term ::=
   NewExpr.term '%' NewExpr.factor |
   Expr.factor |
   Expr.term '*' Expr.factor |
   Expr.term '/' Expr.factor ;
```

We see a mix of nonterminals from `Expr` and from `NewExpr`, the effect of which is that an expression such as `a * b % c` is not in the language of `NewExpr.expression` — as soon as we encounter one of the operators from the `Expr` language, we are locked into the `Expr` sublanguage because the recursive loop of productions does not include the new production for `NewExpr.term`.

The solution is to clone-import the rules for `Expr` into `NewExpr` using a module header like `NewExpr(<=Expr)(expression)` which yields the following productions for `NewExpr.term`.

```
NewExpr.term ::=
   NewExpr.term '%' NewExpr.factor |
   NewExpr.factor |
   NewExpr.term '*' NewExpr.factor |
   NewExpr.term '/' NewExpr.factor ;
```

Adding a new operator priority level requires a little more work. We need some way to 'split open' the productions in the expression grammar so that we can insert a new nonterminal. To insert a new operator `@` of priority

above + and * we can use nonterminal renaming imports, and create a new version of the rule for `factor`:

```
NewExpr(<=Expr.expression <=Expr.term <=Expr.id <= Expr.intLiteral
        oldFactor<=Expr.factor )(expression)
factor ::= oldFactor | factor '@' oldFactor;
```

which induces the following rules:

```
NewExpr.expression ::=
   NewExpr.term |
   NewExpr.expression '+' NewExpr.term |
   NewExpr.expression '-' NewExpr.term ;

NewExpr.term ::=
   NewExpr.factor |
   NewExpr.term '*' NewExpr.factor |
   NewExpr.term '/' NewExpr.factor ;

NewExpr.factor ::=
   NewExpr.oldFactor |
   NewExpr.factor '@' NewExpr.oldFactor ;

NewExpr.oldFactor ::=
   NewExpr.intLiteral |
   NewExpr.id |
   '(' NewExpr.expression ')' ;
```

8. Related work

In this section we look at the modularity features of some well known grammar based tools.

Most versions of the LL parser generator ANTLR [17] do not support modularity at the grammar level although if the target language is Java it is possible to import other Java modules; this mechanism is not powerful enough to facilitate grammar modularity. Since LL is not a general parsing technique true modularity would in any case be difficult to sustain. ANTLR4 has some inheritance-oriented modularity mechanisms, as described earlier.

TXL [6] offers a very primitive grammar composition mechanism based on the standard include mechanism. It is possible to split up the grammar rules over multiple files, but there is only one start nonterminal: `program`. On the other hand a `redefine` mechanism allows replacement of the alternates for a specific nonterminal, or extension of the alternates of an existing nonterminal. Automatic whitespace handling in TXL may be disabled *via* a switch in which case whitespace must be explicitly handled within the grammar.

As discussed above, Rats! [10] is a parser generator that generates Java parsers based on parsing expression grammars (PEGs). PEGs are a recent reintroduction of Aho and Ullman's TDPL formalism [1, chapter 6]. In Rats!, PEGs are closed under set difference, which eases the deleter problem, although the need to take care over ordering (since the language of a PEG is not independent of rule ordering) complicates matters. The tool allows adding, overriding and removing of individual alternates of grammar rules. In order to perform these modifications, Rats! provides module parameters. Rats! provides operators to add `+=` a new alternate before or after an existing one, to remove `-=` an alternate, and to override `:=` a specific alternate or an entire production.

8.1. SDF2 module mechanisms

SDF2 module specifications induce grammars in which the nonterminal names do not retain their module origins; in our terminology SDF2 imports by clone not by reference. When processing a top level module M, SDF2 recursively constructs a set R of modules reachable via chains of import statements. For each module $R_i \in R$, and for each nonterminal $N \in R_i$, SDF2 performs the import $M.N \Leftarrow R_i.N$.

SDF2 allows renaming of symbols on import *via* renaming and parameterisation [11, 23]; both terminals and nonterminals may be renamed. For example, consider a module to manipulate tables with two parameters, key and value of the type Key and Value respectively. When importing this generic module the type Key can be bound to the nonterminal Identifier and the type Value can be bound to the nonterminal Type, in order to create a type environment. The parameterised modules are mainly employed when SDF2 is used to describe the signature of ASF functions rather than for grammar induction, although the AspectJ grammar described in [4] uses module parameterisation for the composition of the syntax of Java with the syntax of aspects.

Parameterisation and renaming in SDF2 are effectively two different concrete syntaxes which access the same underlying semantics; when nonterminal $P \in R_i$ is renamed to Q on import to M, SDF2 uses the import $M.Q \Leftarrow R_i.P$ rather than the default import $M.P \Leftarrow R_i.P$.

The parameterisation mechanism provides named formal parameters for modules which must be bound to symbols in the importing module; this example would be expressed as

```
module Ri[P]
exports sorts P
```

```
...

module M
imports Ri[Q]
```

The renaming mechanism allows any nonterminal to be directly renamed and thus explicit parameters are not required.

```
module Ri
exports sorts P
...

module M
imports Ri[P => Q]
```

SDF2 also provides an alias mechanism which may be used to create non-parameterised single level macros across all inputs. The main application is to construct single-symbol abbreviations; but since an alias defined in one module is applied across all modules used in inducing a grammar, complex effects can be generated. The maintenance issues with macro languages are well understood, and although the non-recursive application of SDF2 aliases mitigates their complexity, accidental name clashes arising from the global scope of aliases can generate unexpected behaviours. Our modularisation mechanisms do not extend to macro use, however where macros are desirable, processors such as M4 or the C preprocessor can be used to emulate SDF2's alias mechanism.

8.2. SDF2 whitespace management

SDF2 provides a single whitespace nonterminal which is inserted between all symbols in context free rules; this induces additional ambiguity compared to our mechanism.

In detail, SDF2 distinguishes three different types of syntax sections: lexical syntax, context-free syntax, and core syntax. The lexical syntax and context-free syntax sections are specializations of the latter one. The grammar rules in both sections are translated to core syntax grammar rules when the grammar is processed by the parse table generator. A number of grammar transformations will take place, we discuss only the transformation rules related to the processing of whitespace, new lines and comments. For the other grammar transformation rules we refer to [23].

In the grammar rules of the core syntax the grammar developer has to indicate explicitly where layout has to be inserted in left hand side

of a grammar rules (note that in SDF2 the left and right hand side are swapped with respect to (E)BNF rules). This has to be done via the nonterminal `LAYOUT?`, where the `?` indicates that layout is optional. There may be layout tokens inserted, but it is not strictly necessary. The grammar transformation that takes place when translating a lexical syntax section into a core syntax section does not insert this `LAYOUT?` between the members in the left-hand side of the lexical grammar rules. For instance, `[a-z][a-zA-Z0-9]* -> Id` is translated into `[a-z][a-zA-Z0-9]* -> Id`. The grammar transformation that takes place when translating a context-free syntax section into a core syntax section inserts the `LAYOUT?` nonterminal between all members in the left-hand side of the context-free grammar rules. For instance, `"begin" Decls Stats "end" -> Program` is translated into `"begin" LAYOUT? Decls LAYOUT? Stats LAYOUT? "end" -> Program`. There is no `LAYOUT?` nonterminal inserted before the first member or appended after last member, except for the rules that have the predefined nonterminal `START` in the right hand side, for instance, `LAYOUT? Program LAYOUT? -> START`. If a member in the left hand side of a context-free grammar rule is nullable, this may lead to ambiguities. This problem is solved by explicitly restricting the recognition of layout characters after a `LAYOUT?`. This is for instance achieved in the following way, under the assumption that a whitespace, newline and tabular are the layout characters:

```
context-free restrictions
  LAYOUT? -/- [\ \n\t]
```

SDF2 provides, like TXL, only one `LAYOUT` nonterminal. The layout can be defined by the grammar writer, but holds for the entire grammar definition so in the case of language embedding, the `LAYOUT` nonterminal would normally be set to the union of the individual whitespace languages. Module-specific whitespace conventions are an active development topic in SDF2.

9. Conclusions

We have described a modularisation technique for context free grammars in which productions are copied into importing modules, with deleters allowing individual productions to be suppressed from the copying process. Since removing a production from a context free grammar yields a context free grammar, this method for removing parts of a context free language cannot accidentally yield a context sensitive language. We have also developed a semantics for automatic whitespace insertion which supports the traditional

conventions used in programming language standards but which allows accurate embedding of languages with differing whitespace conventions.

There remain significant open questions concerning both the human factors and theoretical underpinnings of these techniques which we shall continue to investigate. We note that if no whitespace mechanism is defined then we have a traditional token level grammar which would specify parsers that operated on token streams, not characters. In such a case, we assume that an externally generated lexical analyser mediates the parser's access to the input string.

There are some technical challenges in the present scheme whose significance to users can only be evaluated with experience.

1. In the current scheme an import cannot add whitespace if the imported module has an empty whitespace convention.
   One alternative proposal that we shall investigate is that import-by-clone should suppress all whitespace from the exporting module, and instantiate the whitespace convention of the importing module. A less restrictive possibility could be that clone imports allow explicit specification of the whitespace convention to use.
2. An important aspect of disambiguation in real programming languages is the suppression of reserved words from the pattern of the identifier symbol. Our scheme presently offers no special features to support this, though in a generalised parsing environment local nondeterminism can often be resolved at phrase level.

Our main goals are to allow users to write grammar specifications in a conventional style, but for the tools to display an exact character level context free grammar including all details of whitespace handling, and to promote reuse. We are using the techniques described here to generate parsers for tools which aim to provide a modular library of reusable formal semantics components. Reusability of syntactic components would be highly desirable, and the project provides an opportunity to evaluate our proposals in the context of real-world general purpose languages. A particularly challenging case study has involved the lexical structure of the highly ambiguous OCaml reference grammar in which we have used our modularity constructs to 'mix and match' a variety of lexical models. This is necessary because the obvious character level grammar generates so much ambiguity that we wish to use longest match tokenisation as we would with conventional languages; however OCaml allows user defined operators in which the initial character has significance within the prioritisation scheme so a hybrid approach is required. Using ART's modularity constructs allows alterate partitionings to

be simply specified.

9.1. Acknowledgements

This work was partially funded under UK EPSRC grant EP/I032509/1 *PLanCompS: Programming Language Components and Specifications*. We are grateful for the helpful suggestions of the anonymous reviewers and further comments from Ana-Maria Farcasi, Robert Walsh, Joseph Reddington and the editors.

References

[1] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation and Compiling*, volume 1 — Parsing of *Series in Automatic Computation*. Prentice-Hall, 1972.

[2] J. Andersen and C. Brabrand. Syntactic language extension via an algebra of languages and transformations. *Electr. Notes Theor. Comput. Sci.*, 253(7):19–35, 2010.

[3] J. Andersen, C. Brabrand, and D. R. Christiansen:. Banana algebra: Compositional syntactic language extension. *Science of Computer Programming, to appear*, 2013.

[4] M. Bravenboer, E. Tanter, and E. Visser. Declarative, formal, and extensible syntax definition for AspectJ. OOPSLA 2006, pages 209–228, New York, NY, USA, 2006. ACM.

[5] M. Bravenboer and E. Visser. Parse table composition. In D. Gašević, R. Lämmel, and E. Wyk, editors, *Lecture Notes in Computer Science*, pages 74–94. Springer-Verlag, Berlin, Heidelberg, 2009.

[6] J. R. Cordy. The TXL source transformation language. *Sci. Comput. Program.*, 61:190–210, 2006.

[7] M. de Jonge. Reuse of ASF+SDF specifications by means of renaming. Technical Report P9718, University of Amsterdam, 1997.

[8] T. Ekman and G. Hedin. The JastAdd system – modular extensible compiler construction. *Science of Computer Programming*, 69(13):14 – 26, 2007.

[9] B. Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pages 111–122. ACM, 2004.

[10] R. Grimm. Better extensibility through modular syntax. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 38–51, New York, NY, USA, 2006. ACM.

[11] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF reference manual. *SIGPLAN Not.*, 24:43–75, November 1989.

[12] J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. *IEEE Trans. Software Eng.*, 16(12):1344–1351, 1990.

[13] R. N. Horspool. Incremental generation of LR parsers. *Comput. Lang.*, 15(4):205–223, 1990.

[14] A. Johnstone, E. Scott, and M. van den Brand. LDT: a language definition technique. In C. Brabrand and E. van Wyk, editors, *LDTA'11 11th Workshop on Language Descriptions, Tools and Applications*, ACM Digital Library. ACM, 2011.

[15] P. Klint, R. Lämmel, and C. Verhoef. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering Methodology*, 14(3):331–380, 2005.

[16] T. Parr. ANTLR4 grammar structure. `http://www.antlr.org/wiki/display/ANTLR4/Grammar+Structure`. Accessed: June 28 2013.

[17] T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. Pragmatic Bookshelf, 2007.

[18] T. Parr and K. Fisher. LL(*): the foundation of the ANTLR parser generator. In *PLDI*, pages 425–436, 2011.

[19] A. Schwerdfeger and E. Van Wyk. Verifiable composition of deterministic grammars. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 199–210, 2009.

[20] E. Scott and A. Johnstone. GLL parsing. *Electr. Notes Theor. Comput. Sci.*, 253(7):177–189, 2010.

[21] E. Scott and A. Johnstone. GLL parse-tree generation. *Science of Computer Programming, in press*, 2012.

[22] M. van den Brand, J. Heering, P. Klint, and P. Olivier. Compiling language definitions: the ASF+SDF compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, 2002.

[23] E. Visser. *Syntax Definition for Language Prototyping.* PhD thesis, University of Amsterdam, September 1997.

[24] E. V. Wyk, D. Bodin, J. Gao, and L. Krishnan. Silver: An extensible attribute grammar system. *Science of Computer Programming*, 75(1V2):39 – 54, 2010.