TECHNISCHE
UNIVERSITÄT
DRESDEN

Fakultät Informatik - Institut für Software- und Multimediatechnik

TECHNICAL REPORTS

ISSN 1430-211X

TUD-FI12-09 Dezember 2012

Christoff Bürger
Fakultät Informatik, Lehrstuhl Softwaretechnologie

RACR: A Scheme Library for Reference Attribute Grammar Controlled Rewriting

Technische Universität Dresden
Fakultät Informatik
Institut für Software- und Multimediatechnik
Lehrstuhl Softwaretechnologie
01062 Dresden
Germany

DRESDEN
concept
Exzellenz aus
Wissenschaft
und Kultur

Technische Universität Dresden
Institut für Software- und Multimediatechnik
Lehrstuhl Softwaretechnologie

Developer Manual

RACR

A *Scheme* Library for Reference Attribute Grammar Controlled Rewriting

Christoff Bürger
`Christoff.Buerger@gmx.net`

December 20, 2012

v0.4.3

Developer Manual for RACR v0.4.3
RACR download and homepage: `https://code.google.com/p/racr/`

Contents

List of Figures

1. Introduction

RACR is a reference attribute grammar library for the programming language *Scheme* supporting incremental attribute evaluation in the presence of abstract syntax tree (AST) rewrites. It provides a set of functions that can be used to specify AST schemes and their attribution and construct respective ASTs, query their attributes and node information and annotate and rewrite them. Three main characteristics distinguish *RACR* from other attribute grammar and term rewriting tools:

- **Library Approach** Attribute grammar specifications, applications and AST rewrites can be embedded into ordinary *Scheme* programs; Attribute equations can be implemented using arbitrary *Scheme* code; AST and attribute queries can depend on runtime information permitting dynamic AST and attribute dispatches.
- **Incremental Evaluation based on Dynamic Attribute Dependencies** Attribute evaluation is demand-driven and incremental, incorporating the actual execution paths selected at runtime for control-flows within attribute equations.
- **Reference Attribute Grammar Controlled Rewriting** AST rewrites can depend on attributes and automatically mark the attributes they influence for reevaluation.

Combined, these characteristics permit the expressive and elegant specification of highly flexible but still efficient language processors. The reference attribute grammar facilities can be used to realise complicated analyses, e.g., name, type, control- or data-flow analysis. The rewrite facilities can be used to realise transformations typically performed on the results of such analyses like code generation, optimisation or refinement. Thereby, both, reference attribute grammars and rewriting, are seamlessly integrated, such that rewrites can reuse attributes (in particular the rewrites to apply can be selected and derived using attributes and therefore depend on and are controlled by attributes) and attribute values change depending on performed rewrites. Figure 1.1 illustrates this analyse-synthesize cycle that is at the heart of reference attribute grammar controlled rewriting.

In the rest of the introduction we discuss why reference attribute grammar controlled rewriting is indeed expressive, elegant and efficient and why *RACR* additionally is flexible and reliable.

1.1. *RACR* is Expressive, Elegant, Efficient, Flexible and Reliable

Expressive The specification of language processors using *RACR* is convenient, because reference attribute grammars and rewriting are well-known techniques for the specification

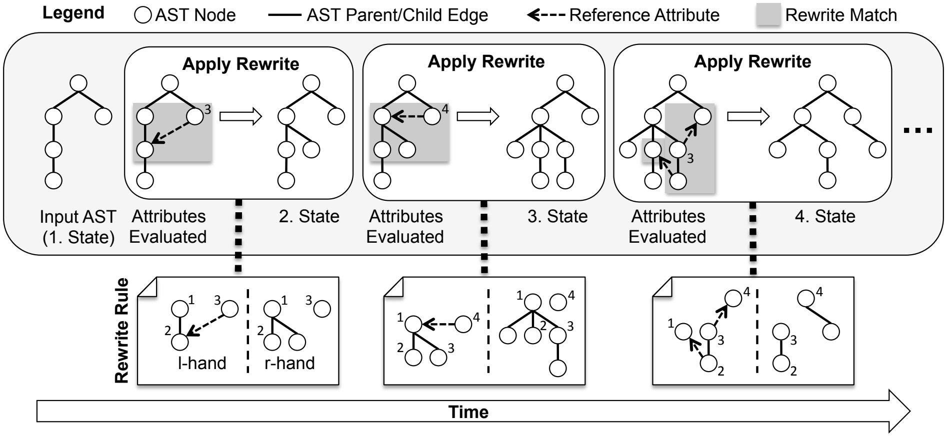

Figure 1.1.: Analyse-Synthesize Cycle of RAG Controlled Rewriting

of static semantic analyses and code transformations. Further, reference attributes extend ASTs to graphs by introducing additional edges connecting remote AST nodes. The reference attributes induce an overlay graph on top of the AST. Since *RACR* rewrites can be applied depending on attribute values, including the special case of dependencies on reference attributes, users can match arbitrary graphs and not only term structures for rewriting. Moreover, attributes can be used to realise complex analyses for graph matching and rewrite application (i.e., to control rewriting).

Example: *Figure 1.2 presents a set of rewrite rules realising a typical compiler construction task: The implicit coercion of integer typed expressions to real. Many statically typed programming languages permit the provision of integer values in places where real values are expected for which reason their compilers must automatically insert real casts that preserve the type correctness of programs. The RACR rewrite rules given in Figure 1.2 specify such coercions for three common cases: (1) Binary expressions, where the first operand is a real and the second an integer value, (2) the assignment of an integer value to a variable of type real and (3) returning an integer value as result of a procedure that is declared to return real values. In all three cases, a real cast must be inserted before the expression of type integer. Note, that the actual transformation (i.e., the insertion of a real cast before an expression) is trivial. The tricky part is to decide for every expression, if it must be casted. The specification of respective rewrite conditions is straightforward however, if name and type analysis can be reused like in our reference attribute grammar controlled rewriting solution. In the binary expression case (1), just the types of the two operands have to be constrained. In case of assignments (2), the name analysis can be used to find the declaration of the assignment's left-hand. Based on the declaration, just its type and the type of the assignment's right-hand expression have to be constrained. In case of procedure returns (3), an inherited reference attribute can be used to distribute to every statement the innermost procedure declaration it is part of. The actual rewrite condition then just has to constraint the return type of the innermost procedure declaration of the return statement and the type of its expression. Note, how the name analyses required in cases (2) and (3)*

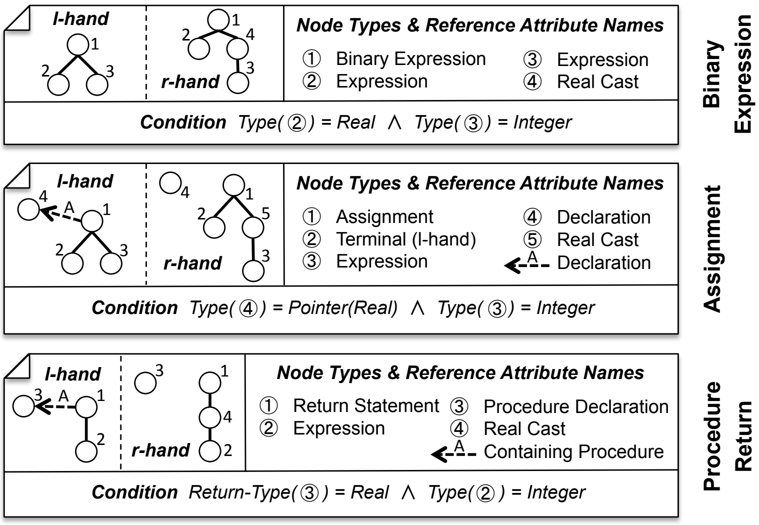

Figure 1.2.: Rewrite Rules for Integer to Real Type Coercion of a Programming Language

naturally correspond to reference edges within left-hand sides of rewrite rules. Also note, that rewrites can only transform AST fragments. The specification of references within right-hand sides of rewrite rules is not permitted.

Elegant Even if only ASTs can be rewritten, the analyse synthesise cycle ensures, that attributes influenced by rewrites are automatically reevaluated by the attribute grammar which specifies them, including the special case of reference attributes. Thus, the overlay graph is automatically transformed by AST rewrites whereby these transformations are consistent with existing language semantics (the existing reference attribute grammar). In consequence, developers can focus on the actual AST transformations and are exempt from maintaining semantic information throughout rewriting. The reimplementation of semantic analyses in rewrites, which is often paralleled by cumbersome techniques like blocking or marker nodes and edges, can be avoided.

Example: *Assume the name analysis of a programming language is implemented using reference attributes and we like to develop a code transformation which reuses existing or introduces new variables. In RACR it is sufficient to apply rewrites that just add the new or reused variables and their respective declarations if necessary; the name resolution edges of the variables will be transparently added by the existing name analysis.*

A very nice consequence of reference attribute grammar controlled rewriting is, that rewriting benefits from any attribute grammar improvements, including additional or improved attribute specifications or evaluation time optimisations.

Efficient Rewriting To combine reference attribute grammars and rewriting to reference attribute grammar controlled rewriting is also reasonable considering rewrite performance. The main complexity issue of rewriting is to decide for a rewrite rule if and where it can be applied on a given graph (matching problem). In general, matching is NP-complete for arbitrary rules and graphs and polynomial if rules have a finite left-hand size. In reference attribute grammar controlled rewriting, matching performance can be improved by exploiting the AST and overlay graph structure induced by the reference attribute grammar. It is well-known from mathematics, that for finite, directed, ordered, labeled tress, like ASTs, matching is linear. Starting from mapping an arbitrary node of the left-hand side on an arbitrary node of the host graph, the decision, whether the rest of the left-hand also matches or not, requires no backtracking; It can be performed in constant time (the pattern size). Likewise, there is no need for backtracking to match reference attributes, because every AST node has at most one reference attribute of a certain name and every reference attribute points to exactly one (other) AST node. The only remaining source for backtracking are left-hand sides with several unconnected AST fragments, where, even if some fragment has been matched, still several different alternatives have to be tested for the remaining ones. If we restrict, that left-hand sides must have a distinguished node from which all other nodes are reachable (with non-directed AST child/parent edges and directed reference edges), also this source for backtracking is eliminated, such that matching is super-linear if, and only if, the complexity of involved attributes is. In other words, the problem of efficient matching is reduced to the problem of efficient attribute evaluation.

Efficient Attribute Evaluation A common technique to improve attribute evaluation efficiency is the caching of evaluated attribute instances. If several attribute instances depend on the value of a certain instance `a`, it is sufficient to evaluate `a` only once, memorise the result and reuse it for the evaluation of the depending instances. In case of reference attribute grammar controlled rewriting however, caching is complicated because of the analyse-synthesise cycle. Two main issues arise if attributes are queried in-between AST transformations: First, rewrites only depend on certain attribute instances for which reason it is disproportionate to use (static) attribute evaluation strategies that evaluate all instances; Second, rewrites can change AST information contributing to the value of cached attribute instances for which reason the respective caches must be flushed after their application. In *RACR*, the former is solved by using a demand-driven evaluation strategy that only evaluates the attribute instances required to decide matching, and the latter by tracking dependencies throughout attribute evaluation, such that it can be decided which attribute instances applied rewrites influenced and incremental attribute evaluation can be achieved. In combination, demand-driven, incremental attribute evaluation enables attribute caching – and therefore efficient attribute evaluation – for reference attribute grammar controlled rewriting. Moreover, because dependencies are tracked throughout attribute evaluation, the actual execution paths selected at runtime for control-flows within attribute equations can be incorporated. In the end, the demand-driven evaluator of *RACR* uses runtime information to construct an AST specific dynamic attribute dependency graph that permits more precise attribute cache flushing than a static dependency analysis.

Example: *Let* `att-value` *be a function, that given the name of an attribute and an AST node evaluates the respective attribute instance at the given node. Let* `n1`*,...,*`n4` *be arbitrary AST nodes, each with an attribute instance* `i1`*,...,*`i4` *named* `a1`*,...,*`a4` *respectively. Assume, the equation of the attribute instance* `i1` *for* `a1` *at* `n1` *is:*

```
(if (att-value a2 n2)
    (att-value a3 n3)
    (att-value a4 n4))
```

Obviously, `i1` *always depends on* `i2`*, but only on either,* `i3` *or* `i4`*. On which of both depends on the actual value of* `i2`*, i.e., the execution path selected at runtime for the* `if` *control-flow statement. If some rewrite changes an AST information that influences the value of* `i4`*, the cache of* `i1` *only has to be flushed if the value of* `i2` *was* `#f`*.*

Besides automatic caching, a major strong point of attribute grammars, compared to other declarative formalisms for semantic analyses, always has been their easy adaptation for present programming techniques. Although attribute grammars are declarative, their attribute equation concept based on semantic functions provides sufficient opportunities for tailoring and fine tuning. In particular developers can optimise the efficiency of attribute evaluation by varying attributions and semantic function implementations. *RACR* even improves in that direction. Because of its tight integration with *Scheme* in the form of a library, developers are more encouraged to *"just program"* efficient semantic functions. They benefit from both, the freedom and efficiency of a real programming language and the more abstract attribute grammar concepts. Moreover, *RACR* uses *Scheme's* advanced macro- and meta-programming facilities to still retain the attribute evaluation efficiency that is rather typical for compilation- than for library-based approaches.

Flexible *RACR* is a *Scheme* library. Its AST, attribute and rewrite facilities are ordinary functions or macros. Their application can be controlled by complex *Scheme* programs that compute, or are used within, attribute specifications and rewrites. In particular, *RACR* specifications themselves can be derived using *RACR*. There are no limitations on the interactions between different language processors or the number of meta levels. Moreover, all library functions are parameterised with an actual application context. The function for querying attribute values uses a name and node argument to dispatch for a certain attribute instance and the functions to query AST information or perform rewrites expect node arguments designating the nodes to query or rewrite respectively. Since such contexts can be computed using attributes and AST information, dynamic – i.e., input dependent – AST and attribute dispatches within attribute equations and rewrite applications are possible. For example, the name and node arguments of an attribute query within some attribute equation can be the values of other attributes or even terminal nodes. In the end, *RACR's* library approach and support for dynamic AST and attribute dispatches eases the development and combination of language product lines, metacompilers and highly adaptive language processors.

Reliable *RACR* specified language processors that interact with each other to realise a stacked metaarchitecture consisting of several levels of language abstraction can become very complicated. Also dynamic attribute dispatches or user developed *Scheme* programs applying *RACR* can result in complex attribute and rewrite interactions. Nevertheless, *RACR* ensures that only valid specifications and transformations are performed and never outdated attribute values are used, no matter of application context, macros and continuations. In case of incomplete or inconsistent specifications, unspecified AST or attribute queries or transformations yielding invalid ASTs, *RACR* throws appropriate runtime exceptions to indicate program errors. In case of transformations influencing an AST information that has been used to evaluate some attribute instance, the caches of the instance and all instances depending on it are automatically flushed, such that they are reevaluated if queried later on. The required bookkeeping is transparently performed and cannot be bypassed or disturbed by user code (in particular ASTs can only be queried and manipulated using library functions provided by *RACR*). There is only one restriction developers have to pay attention for: To ensure declarative attribute specifications, attribute equations must be side effect free. If equations only depend on attributes, attribute parameters and AST information and changes of stateful terminal values are always performed by respective terminal value rewrites, this restriction is satisfied.

1.2. Structure of the Manual

The next chapter finishes the just presented motivation, application and feature overview of this introduction. It gives an overview about the general architecture of *RACR*, i.e., its embedding into *Scheme*, its library functions and their usage. Chapters 2-6 then present the library functions in detail: Chapter 2 the functions for the specification, construction and querying of ASTs; Chapter 3 the functions for the specification and querying of attributes; Chapter 4 the functions for rewriting ASTs; Chapter 5 the functions for associating and querying entities associated with AST nodes (so called AST annotations); and finally Chapter 6 the functions that ease development for common cases like the configuration of a default *RACR* language processor. The following appendix presents *RACR's* complete implementation. The implementation is well documented. All algorithms, including attribute evaluation, dependency graph maintenance and the attribute cache flushing of rewrites, are stepwise commented and therefore provide a good foundation for readers interested into the details of reference attribute grammar controlled rewriting. Finally, an API index eases the look-up of library functions within the manual.

2. Library Overview

2.1. Architecture

To use *RACR* within *Scheme* programs, it must be imported via `(import (racr))`. The imported library provides a set of functions for the specification of AST schemes, their attribution and the construction of respective ASTs, to query their information (e.g., for AST traversal or node type comparison), to evaluate their attributes and to rewrite and annotate them.

Every AST scheme and its attribution define a language – they are a ***RACR* specification**. Every *RACR* specification can be compiled to construct the ***RACR* language processor** it defines. Every *RACR* AST is one word in evaluation by a certain *RACR* language processor, i.e., a runtime snapshot of a word in compilation w.r.t. a certain *RACR* specification. Thus, *Scheme* programs using *RACR* can specify arbitrary many *RACR* specifications and for every *RACR* specification arbitrary many ASTs (i.e., words in compilation) can be instantiated and evaluated. Thereby, every AST has its own **evaluation state**, such that incremental attribute evaluation can be automatically maintained in the presence of rewrites. Figure 2.1 summarises the architecture of *RACR* applications. Note, that specification, compilation and evaluation are realised by ordinary *Scheme* function applications embedded within a single *Scheme* program, for which reason they are just-in-time and on demand.

The relationships between AST rules and attribute definitions and ASTs consisting of nodes and attribute instances are as used to. *RACR* specifications consist of a set of **AST rules**, whereby for every AST rule arbitrary many **attribute definitions** can be specified. ASTs

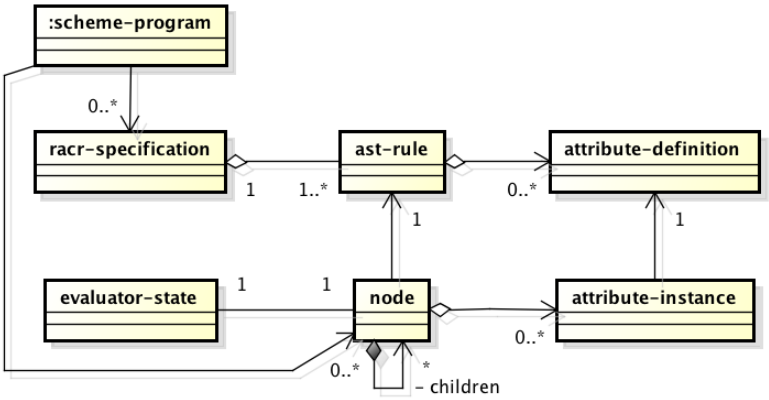

Figure 2.1.: Architecture of RACR Applications

consist of arbitrary many **nodes** with associated **attribute instances**. Each node represents a context w.r.t. an AST rule and its respective attributes.

2.2. Instantiation

Three different language specification and application phases are distinguished in *RACR*:

- AST Specification Phase
- AG Specification Phase
- AST construction, query, evaluation, rewriting and annotation phase (Evaluation Phase)

The three phases must be processed in sequence. E.g., if a *Scheme* program tries to construct an AST w.r.t. a *RACR* specification before finishing its AST and AG specification phase, *RACR* will abort with an exception of type `racr-exception` incorporating an appropriate error message. The respective tasks that can be performed in each of the three specification phases are:

- **AST Specification Phase** Specification of AST schemes
- **AG Specification Phase** Definition of attributes
- **Evaluation Phase** One of the following actions:
  - Construction of ASTs
  - Querying AST information
  - Querying the values of attributes
  - Rewriting ASTs
  - Weaving and querying AST annotations

The AST query and attribute evaluation functions are not only used to interact with ASTs but also in attribute equations to query AST nodes and attributes local within the context of the respective equation.

Users can start the next specification phase by special compilation functions, which check the consistency of the specification, throw proper exceptions in case of errors and derive an optimised internal representation of the specified language (thus, compile the specification). The respective compilation functions are:

- `compile-ast-specifications`: AST => AG specification phase
- `compile-ag-specifications`: AG specification => Evaluation phase

To construct a new specification the `create-specification` function is used. Its application yields a new internal record representing a *RACR* specification, i.e., a language. Such records are needed by any of the AST and AG specification functions to associate the specified AST rule or attribute with a certain language.

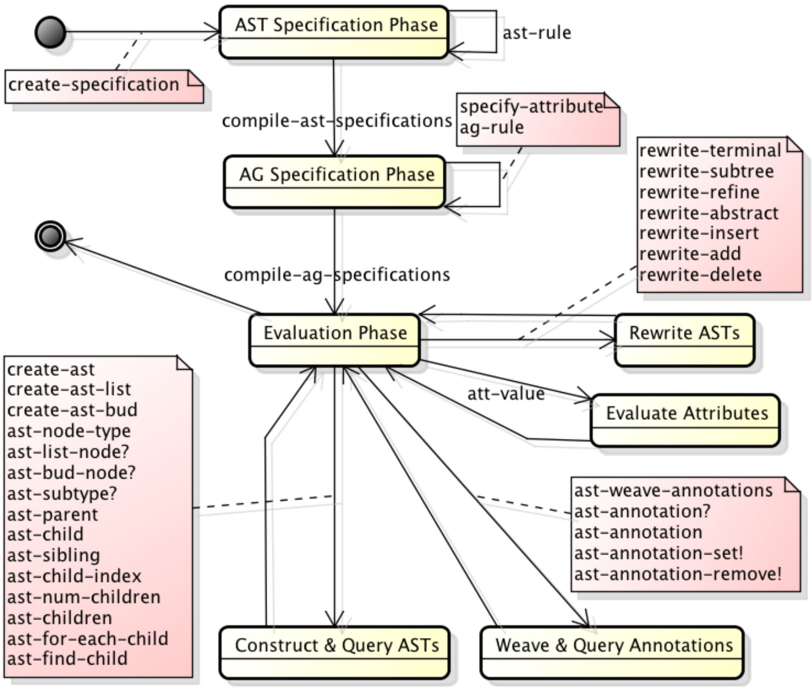

Figure 2.2.: RACR API

2.3. API

The state chart of Figure 2.2 summarises the specification and AST and attribute query, rewrite and annotation API of *RACR*. The API functions of a certain specification phase are denoted by labels of edges originating from the respective phase. Transitions between different specification phases represent the compilation of specifications of the source phase, which finishes the respective phase such that now tasks of the destination phase can be performed.

Remember, that *RACR* maintains for every *RACR* specification (i.e., specified language) its specification phase. Different *RACR* specifications can coexist within the same *Scheme* program and each can be in a different phase.

3. Abstract Syntax Trees

This chapter presents *RACR's* abstract syntax tree (AST) API, which provides functions for the specification of AST schemes, the construction of respective ASTs and the querying of ASTs for structural and node information. *RACR* ASTs are based on the following context-free grammar (CFG), Extended Backus-Naur Form (EBNF) and object-oriented concepts:

- **CFG** Non-terminals, terminals, productions, total order of production symbols
- **EBNF** Unbounded repetition (Kleene Star)
- **Object-Oriented Programming** Inheritance, named fields

RACR ASTs are directed, typed, ordered trees. Every AST node has a type, called its node type, and a finite number of children. Every child has a name and is either, another AST node (i.e., non-terminal) or a terminal. Non-terminal children can represent unbounded repetitions. Given a node, the number, order, types, names and information, whether they are unbounded repetitions, of its children are induced by its type. The children of a node type must have different names; children of different node types can have equal names. We call names defined for children context names and a node with type `T` an instance of `T`.

Node types can inherit from each other. If a node type `A` inherits from another type `B`, `A` is called direct subtype of `B` and `B` direct supertype of `A`. The transitive closure of direct sub- and supertype are called a node type's sub- and supertypes, i.e., a node type `A` is a sub-/supertype of a type `B`, if `A` is a direct sub-/supertype of `B` or `A` is a direct sub-/supertype of a type `C` that is a sub-/supertype of `B`. Node types can inherit from atmost one other type and must not be subtypes of themselves. If a node type is subtype of another one, its instances can be used anywhere an instance of its supertype is expected, i.e., if `A` is a subtype of `B`, every AST node of type `A` also is of type `B`. The children of a node type are the ones of its direct supertype, if it has any, followed by the ones specified for itself.

Node types are specified using AST rules. Every AST rule specifies one node type of a certain name. The set of all AST rules of a *RACR* specification are called an AST scheme.

In terms of object-oriented programming, every node type corresponds to a class; its children are fields. In CFG terms, it corresponds to a production; its name is the left-hand non-terminal and its children are the right-hand symbols. However, in opposite to CFGs, where several productions can be given for a non-terminal, the node types of a *RACR* specification must be unique (i.e., must have different names). To simulate alternative productions, node type inheritance can be used.

RACR supports two special node types besides user specified ones: list-nodes and bud-nodes. Bud-nodes are used to represent still missing AST parts. Whenever a node of some type is expected, a bud-node can be used instead. They are typically used to decompose and reuse decomposed AST fragments using rewrites. List-nodes are used to represent unbounded repetitions. If a child of type `T` with name `c` of a node type `N` is defined to be an unbounded repetition, all `c` children of instances of `N` will be either, a list-node with arbitrary many children of type `T` or a bud-node. Even if list- and bud-nodes are non-terminals, their type is undefined. It is not permitted to query such nodes for their type, including sub- and supertype comparisons. And although bud-nodes never have children, it is not permitted to query them for children related information (e.g., their number of children). After all, bud-nodes represent still missing, i.e., unspecified, AST parts.

3.1. Specification

```
(ast-rule spec symbol-encoding-rule)
```

Calling this function adds to the given *RACR* specification the AST rule encoded in the given symbol. To this end, the symbol is parsed. The function aborts with an exception, if the symbol encodes no valid AST rule, there already exists a definition for the l-hand of the rule or the specification is not in the AST specification phase. The grammar used to encode AST rules in symbols is (note, that the grammar has no whitespace):

```
Rule ::= NonTerminal [":" NonTerminal] "−>" [ProductionElement {"−" ProductionElement}];
ProductionElement := NonTerminal [*] [< ContextName] | Terminal;
NonTerminal ::= UppercaseLetter {Letter} {Number};
Terminal ::= LowercaseLetter {LowercaseLetter} {Number};
ContextName ::= Letter {Letter} {Number};
Letter ::= LowercaseLetter | UppercaseLetter;
LowercaseLetter ::= "a" | "b" | ... | "z";
UppercaseLetter ::= "A" | "B" | ... | "Z";
Number ::= "0" | "1" | ... | "9";
```

Every AST rule starts with a non-terminal (the l-hand), followed by an optional supertype and the actual r-hand consisting of arbitrary many non-terminals and terminals. Every non-terminal of the r-hand can be followed by an optional *Kleene star*, denoting an unbounded repetition (i.e., a list with arbitrary many nodes of the respective non-terminal). Further, r-hand non-terminals can have an explicit context name. Context names can be used to select the respective child for example in attribute definitions (`specify-attribute`, `ag-rule`) or AST traversals (e.g., `ast-child` or `ast-sibling`). If no explicit context name is given, the non-terminal type and optional *Kleene star* are the respective context name. E.g., for a list of non-terminals of type `N` without explicit context name the context name is `'N*`. For terminals, explicit context names are not permitted. Their name also always is their context name. For every AST rule the context names of its children (including inherited ones) must be unique. Otherwise a later compilation of the AST specification will throw an exception.

Note: *AST rules, and in particular AST rule inheritance, are object-oriented concepts. The l-hand is the class defined by a rule (i.e., a node type) and the r-hand symbols are its fields, each named like the context name of the respective symbol. Compared to common* *object-oriented languages however, r-hand symbols, including inherited ones, are ordered and represent compositions rather than arbitrary relations, such that it is valid to index them and call them child. The order of children is the order of the respective r-hand symbols and, in case of inheritance, "inherited r-hand first".*

```
(ast-rule spec 'N->A-terminal-A*)
(ast-rule spec 'Na:N->A<A2-A<A3) ; Context-names 4'th & 5'th child: A2 and A3
(ast-rule spec 'Nb:N->)
(ast-rule spec 'Procedure->name-Declaration*<Parameters-Block<Body)
```

```
(compile-ast-specifications spec start-symbol)
```

Calling this function finishes the AST specification phase of the given *RACR* specification, whereby the given symbol becomes the start symbol. The AST specification is checked for completeness and correctness, i.e., (1) all non-terminals are defined, (2) rule inheritance is cycle-free, (3) the start symbol is defined, (4) the start symbol is start separated, (5) no non-terminal inherits from the start symbol, (6) the start symbol does not inherit from any non-terminal and (7) all non-terminals are reachable and (8) productive. Further, it is ensured, that (9) for every rule the context names of its children are unique. In case of any violation, an exception is thrown. An exception is also thrown, if the given specification is not in the AST specification phase. After executing `compile-ast-specifications` the given specification is in the AG specification phase, such that attributes now can be defined using `specify-attribute` and `ag-rule`.

3.2. Construction

```
(ast-node? scheme-entity)
```

Given an arbitrary *Scheme* entity return `#t` if it is an AST node, otherwise `#f`.

```
(create-ast spec non-terminal list-of-children)
```

Function for the construction of non-terminal nodes. Given a *RACR* specification, the name of a non-terminal to construct (i.e., an AST rule to apply) and a list of children, the function constructs and returns a parentless AST node (i.e., a root) whose type and children are the given ones. Thereby, it is checked, that (1) the given children are of the correct type for the fragment to construct, (2) enough and not to many children are given, (3) every child is a root (i.e., the children do not already belong to/are not already part of another AST) and (4) no attributes of any of the children are in evaluation. In case of any violation an exception is thrown.

Note: *Returned fragments do not use the `list-of-children` argument to administer their actual children. Thus, any change to the given list of children (e.g., using `set-car!` or `set-cdr!`) after applying `create-ast` does not change the children of the constructed fragment.*

```
(create-ast spec 'N
  ; List of children:
  (list
    ...
    ; For non-terminal children an AST node is expected:
    (create-ast ...)
    ...
    ; For terminals, not an AST node, but their value is expected:
    "value for a terminal"
    ...
    ; For non-terminal children with unbounded cardinality (Kleene closure)
    ; a list-node containing their elements is expected:
    (create-ast-list ...)
    ...))
```

`(create-ast-list list-of-children)`

Given a list `l` of non-terminal nodes that are not AST list-nodes construct an AST list-node whose elements are the elements of `l`. An exception is thrown, if an element of `l` is not an AST node, is a list-node, already belongs to another AST, has attributes in evaluation or at least two elements of `l` are instances of different *RACR* specifications.

Note: *It is not possible to construct AST list-nodes containing terminal nodes. Instead however, terminals can be ordinary Scheme lists, such that there is no need for special AST terminal lists.*

`(create-ast-bud)`

Construct a new AST bud-node, that can be used as placeholder within an AST fragment to designate a subtree still to provide. Bud-nodes are valid substitutions for any kind of expected non-terminal child, i.e., whenever a non-terminal node of some type is expected, a bud node can be used instead (e.g., when constructing AST fragments via `create-ast` or `create-ast-list` or when adding another element to a list-node via `rewrite-add`). Since bud-nodes are placeholders, any query for non-terminal node specific information of a bud-node throws an exception (e.g., bud-nodes have no type or attributes and their number of children is not specified etc.).

Note: *There exist two main use cases for incomplete ASTs which have "holes" within their subtrees that denote places where appropriate replacements still have to be provided: (1) when constructing ASTs but required parts are not yet known and (2) for the deconstruction and reuse of existing subtrees, i.e., to remove AST parts such that they can be reused for insertion into other places and ASTs. The later use case can be generalised as the reuse of AST fragments within rewrites. The idea thereby is, to use* `rewrite-subtree` *to insert bud-nodes and extract the subtree replaced.*

3.3. Traversal

```
(ast-parent n)
```

Given a node, return its parent if it has any, otherwise thrown an exception.

```
(ast-child index-or-context-name n)
```

Given a node, return one of its children selected by context name or child index. If the queried child is a terminal node, not the node itself but its value is returned. An exception is thrown, if the child does not exist.

Note: *In opposite to many common programming languages where array or list indices start with `0`, in RACR the index of the first child is `1`, of the second `2` and so on.*

Note: *Because element nodes within AST list-nodes have no context name, they must be queried by index.*

```
(let ((ast
       (with-specification
        (create-specification)
        (ast-rule 'S->A-A*-A<MyContextName)
        (ast-rule 'A->)
        (compile-ast-specifications 'S)
        (compile-ag-specifications)
        (create-ast
         'S
         (list
          (create-ast
           'A
           (list))
          (create-ast-list
           (list))
          (create-ast
           'A
           (list)))))))
  (assert (eq? (ast-child 'A ast) (ast-child 1 ast)))
  (assert (eq? (ast-child 'A* ast) (ast-child 2 ast)))
  (assert (eq? (ast-child 'MyContextName ast) (ast-child 3 ast))))
```

```
(ast-sibling index-or-context-name n)
```

Given a node `n` which is child of another node `p`, return a certain child `s` of `p` selected by context name or index (thus, `s` is a sibling of `n` or `n`). Similar to `ast-child`, the value of `s`, and not `s` itself, is returned if it is a terminal node. An exception is thrown, if `n` is a root or the sibling does not exist.

```
(ast-children n . b1 b2 ... bm)
```

Given a node `n` and arbitrary many child intervals `b1,b2,...,bm` (each a pair consisting of a lower bound `lb` and an upper bound `ub`), return a *Scheme* list that contains for each child interval `bi = (lb ub)` the children of `n` whose index is within the given interval (i.e., `lb <= child index <= ub`). The elements of the result list are ordered w.r.t. the order of the child intervals `b1,b2,...,bm` and the children of `n`. I.e.:

- The result lists returned by the child intervals are appended in the order of the intervals.
- The children of the list computed for a child interval are in increasing index order.

If no child interval is given, a list containing all children of `n` in increasing index order is returned. A child interval with unbounded upper bound (specified using `'*` as upper bound) means "select all children with index >= the interval's lower bound". The returned list is a copy – any change of it (e.g., using `set-car!` or `set-cdr!`) does not change the AST! An exception is thrown, if a child interval queries for a non existent child or `n` is a bud-node.

```
(let ((ast
       (with-specification
        (create-specification)
        (ast-rule 'S->t1-t2-t3-t4-t5)
        (compile-ast-specifications 'S)
        (compile-ag-specifications)
        (create-ast 'S (list 1 2 3 4 5)))))
  (assert
   (equal?
    (ast-children ast (cons 2 2) (cons 2 4) (cons 3 '*))
    (list 2 2 3 4 3 4 5)))
  (assert
   (equal?
    (ast-children ast)
    (list 1 2 3 4 5))))
```

```
(ast-for-each-child f n . b1 b2 ... bm)
; f: Processing function of arity two: (1) Index of current child, (2) Current child
; n: Node whose children within the given child intervals will be processed in sequence
; b1 b2 ... bm: Lower-bound/upper-bound pairs (child intervals)
```

Given a function `f`, a node `n` and arbitrary many child intervals `b1,b2,...,bm` (each a pair consisting of a lower bound `lb` and an upper bound `ub`), apply for each child interval `bi = (lb ub)` the function `f` to each child `c` with index `i` with `lb <= i <= ub`, taking into account the order of child intervals and children. Thereby, `f` must be of arity two; Each time `f` is called, its arguments are an index `i` and the respective `i`'th child of `n`. If no child interval is given, `f` is applied to each child once. A child interval with unbounded upper bound (specified using `'*` as upper bound) means "apply `f` to every child with index >= the interval's lower bound". An exception is thrown, if a child interval queries for a non existent child or `n` is a bud-node.

Note: *Like all RACR API functions also* `ast-for-each-child` *is continuation safe, i.e., it is alright to apply continuations within* `f`*, such that the execution of* `f` *is terminated abnormal.*

```
(ast-find-child f n . b1 b2 ... bm)
; f: Search function of arity two: (1) Index of current child, (2) Current child
```

```
; n: Node whose children within the given child intervals will be tested in sequence
; b1 b2 ... bm: Lower−bound/upper−bound pairs (child intervals)
```

Given a search function `f`, a node `n` and arbitrary many child intervals `b1, b2,...,bm`, find the first child of `n` within the given intervals which satisfies `f`. Thereby, the children of `n` are tested in the order specified by the child intervals. The search function must accept two parameters – (1) a child index and (2) the actual child – and return a truth value telling whether the actual child is the one searched for or not. If no child within the given intervals, which satisfies the search function, exists, `#f` is returned, otherwise the child found. An exception is thrown, if a child interval queries for a non existent child or `n` is a bud-node.

Note: *The syntax and semantics of child intervals is the one of `ast-for-each-child`, except the search is aborted as soon as a child satisfying the search condition encoded in `f` is found.*

```
(let ((ast
       (with-specification
        (create-specification)

        ; A program consists of declaration and reference statements:
        (ast-rule ’Program->Statement*)
        (ast-rule ’Statement->)
        ; A declaration declares an entity of a certain name:
        (ast-rule ’Declaration:Statement->name)
        ; A reference refers to an entity of a certain name:
        (ast-rule ’Reference:Statement->name)

        (compile-ast-specifications ’Program)

        (ag-rule
         lookup
         ((Program Statement*)
          (lambda (n name)
            (ast-find-child
             (lambda (i child)
               (and
                (ast-subtype? child ’Declaration)
                (string=? (ast-child ’name child) name)))
             (ast-parent n)
             ; Child interval enforcing declare before use rule:
             (cons 1 (ast-child-index n))))))

        (ag-rule
         correct
         ; A program is correct , if its statements are correct :
         (Program
          (lambda (n)
            (not
             (ast-find-child
              (lambda (i child)
                (not (att-value ’correct child)))
              (ast-child ’Statement* n)))))
```

```
      ; A reference is correct, if it is declared:
      (Reference
       (lambda (n)
         (att-value 'lookup n (ast-child 'name n))))
      ; A declaration is correct, if it is no redeclaration:
      (Declaration
       (lambda (n)
         (eq?
          (att-value 'lookup n (ast-child 'name n))
          n))))

     (compile-ag-specifications)

     (create-ast
      'Program
      (list
       (create-ast-list
        (list
         (create-ast 'Declaration (list "var1"))
         ; First undeclared error:
         (create-ast 'Reference (list "var3"))
         (create-ast 'Declaration (list "var2"))
         (create-ast 'Declaration (list "var3"))
         ; Second undeclared error:
         (create-ast 'Reference (list "undeclared-var")))))))))
(assert (not (att-value 'correct ast)))
; Resolve first undeclared error:
(rewrite-terminal 'name (ast-child 2 (ast-child 'Statement* ast)) "var1")
(assert (not (att-value 'correct ast)))
; Resolve second undeclared error:
(rewrite-terminal 'name (ast-child 5 (ast-child 'Statement* ast)) "var2")
(assert (att-value 'correct ast))
; Introduce redeclaration error:
(rewrite-terminal 'name (ast-child 1 (ast-child 'Statement* ast)) "var2")
(assert (not (att-value 'correct ast))))
```

3.4. Node Information

`(ast-child-index n)`

Given a node, return its position within the list of children of its parent. If the node is a root, an exception is thrown.

`(ast-num-children n)`

Given a node, return its number of children. If the node is a bud-node an exception is thrown.

`(ast-node-type n)`

Given a node, return its type, i.e., the non-terminal it is an instance of. If the node is a list- or bud-node an exception is thrown.

`(ast-list-node? n)`

Given a node, return whether it represents a list of children, i.e., is a list-node, or not. If the node is a bud-node an exception is thrown.

`(ast-bud-node? n)`

Given a node, return whether is is a bud-node or not.

`(ast-subtype? a1 a2)`

Given at least one node and another node or non-terminal symbol, return if the first argument is a subtype of the second. The considered subtype relationship is reflexive, i.e., every type is a subtype of itself. An exception is thrown, if non of the arguments is an AST node, any of the arguments is a list- or bud-node or a given non-terminal argument is not defined (the grammar used to decide whether a symbol is a valid non-terminal or not is the one of the node argument).

```
; Let n, n1 and n2 be AST nodes and t a Scheme symbol encoding a non-terminal:
(ast-subtype? n1 n2) ; Is the type of node n1 a subtype of the type of node n2
(ast-subtype? t n) ; Is the type t a subtype of the type of node n
(ast-subtype? n t) ; Is the type of node n a subtype of the type t
```

4. Attribution

RACR supports synthesised and inherited attributes that can be parameterised, circular and references. Attribute definitions are inherited w.r.t. AST inheritance. Thereby, the subtypes of an AST node type can overwrite inherited definitions by providing their own definition. *RACR* also supports attribute broadcasting, such that there is no need to specify equations that just copy propagate attribute values from parent to child nodes. Some of these features differ from common attribute grammar systems however:

- **Broadcasting** Inherited *and* synthesised attributes are broadcasted *on demand*.
- **Shadowing** Synthesised attribute instances *dynamically* shadow inherited instances.
- **AST Fragment Evaluation** Attributes of incomplete ASTs can be evaluated.
- **Normal Form / AST Query Restrictions** Attribute equations can query AST information without restrictions because of attribute types or contexts.
- **Completeness** It is not checked if for all attribute contexts a definition exists.

Of course, RACR also differs in its automatic tracking of dynamic attribute dependencies and the incremental attribute evaluation based on it (cf. Chapter 1.1: Efficient Attribute Evaluation). Its differences regarding broadcasting, shadowing, AST fragment evaluation, AST query restrictions and completeness are discussed in the following.

Broadcasting If an attribute is queried at some AST node and there exists no definition for the context the node represents, the first successor node with a definition is queried instead. If such a node does not exist a runtime exception is thrown. In opposite to most broadcasting concepts however, *RACR* makes no difference between synthesised and inherited attributes, i.e., not only inherited attributes are broadcasted, but also synthesised. In combination with the absence of normal form or AST query restrictions, broadcasting of synthesised attributes eases attribute specifications. E.g., if some information has to be broadcasted to `n` children, a synthesised attribute definition computing the information is sufficient. There is no need to specify additional `n` inherited definitions for broadcasting.

Shadowing By default, attribute definitions are inherited w.r.t. AST inheritance. If an attribute definition is given for some node type, the definition also holds for all its subtypes. Of course, inherited definitions can be overwritten as used to from object-oriented programming in which case the definitions for subtypes are preferred to inherited ones. Further, the sets of synthesised and inherited attributes are not disjunct. An attribute of a certain name can be synthesised in one context and inherited in another one. If for some attribute instance a synthesised and inherited definition exists, the synthesised is preferred.

AST Fragment Evaluation Attribute instances of ASTs that contain bud-nodes or whose root does not represents a derivation w.r.t. the start symbol still can be evaluated if they are well-defined, i.e., do not depend on unspecified AST information. If an attribute instance depends on unspecified AST information, its evaluation throws a runtime exception.

Normal Form / AST Query Restrictions A major attribute grammar concept is the local definition of attributes. Given an equation for some attribute and context (i.e., attribute name, node type and children) it must only depend on attributes and AST information provided by the given context. Attribute grammar systems requiring normal form are even more restrictive by enforcing that the defined attributes of a context must only depend on its undefined. In practice, enforcing normal form has turned out to be inconvenient for developers, such that most attribute grammar systems abandoned it. Its main application area is to ease proofs in attribute grammar theories. Also recent research in reference attribute grammars demonstrated, that less restrictive locality requirements can considerably improve attribute grammar development. *RACR* even goes one step further, by enforcing no restrictions about attribute and AST queries within equations. Developers are free to query ASTs, in particular traverse them, however they like. *RACR's* leitmotif is, that users are experienced language developers that should not be restricted or patronised. For example, if a developer knows that for some attribute the information required to implement its equation is always located at a certain non-local but relative position from the node the attribute is associated with, he should be able to just retrieve it. And if a software project emphasises a certain architecture, the usage of *RACR* should not enforce any restrictions, even if "weird" attribute grammar designs may result. There are also theoretic and technical reasons why locality requirements are abandoned. Local dependencies are a prerequisite for static evaluation order and cycle test analyses. With the increasing popularity of demand-driven evaluation, because of much less memory restrictions than twenty years ago, combined with automatic caching and support for circular attributes, the reasons for such restrictions vanish.

Completeness Traditionally, attribute grammar systems exploit attribute locality to proof, that for every valid AST all its attribute instances are defined, i.e., an equation is specified for every context. Because of reference attributes and dynamic AST and attribute dispatches, such a static attribute grammar completeness check is impossible for *RACR*. In consequence, it is possible that throughout attribute evaluation an undefined or unknown attribute instance is queried, in which case *RACR* throws a runtime exception. On the other hand, *RACR* developers are never confronted with situations where artificial attribute definitions must be given for ASTs that, even they are valid w.r.t. their AST scheme, are never constructed, because of some reason unknown to the attribute grammar system. Such issues are very common, since parsers often only construct a subset of the permitted ASTs. For example, assume an imperative programming language with pointers. In this case, it is much more easy to model the left-hand side of assignments as ordinary expression instead of defining another special AST node type. A check, that left-hands are only dereference expressions or variables, can be realised within the concrete syntax used for parsing. If however, completeness is enforced and some expression that is not a dereference expression or variable has an inherited attribute, the attribute must be defined for the left-hand of assignments, although it will never occur in this context.

4.1. Specification

```
(specify-attribute spec att-name non-terminal index cached? equation circ-def)
; spec: RACR specification
; att-name: Scheme symbol
; non-terminal: AST rule R in whose context the attribute is defined.
; index: Index or Scheme symbol representing a context-name. Specifies the
;      non-terminal within the context of R for which the definition is.
; cached?: Boolean flag determining, whether the values of instances of
;      the attribute are cached or not.
; equation: Equation used to compute the value of instances of the attribute.
;      Equations have at least one parameter - the node the attribute instance
;      to evaluate is associated with (first parameter).
; circ-def: #f if not circular, otherwise bottom-value/equivalence-function pair
```

Calling this function adds to the given *RACR* specification the given attribute definition. To this end, it is checked, that the given definition is (1) properly encoded (syntax check), (2) its context is defined, (3) the context is a non-terminal position and (4) the definition is unique (no redefinition error). In case of any violation, an exception is thrown. To specify synthesised attributes the index 0 or the context name `'*` can be used.

Note: *There exist only few exceptions when attributes should not be cached. In general, parameterized attributes with parameters whose memoization (i.e., permanent storage in memory) might cause garbage collection problems should never be cached. E.g., when parameters are functions, callers of such attributes often construct the respective arguments – i.e., functions – on the fly as anonymous functions. In most Scheme systems every time an anonymous function is constructed it forms a new entity in memory, even if the same function constructing code is consecutively executed. Since attributes are cached w.r.t. their parameters, the cache of such attributes with anonymous function arguments might be cluttered up. If a piece of code constructing an anonymous function and using it as an argument for a cached attribute is executed several times, it might never have a cache hit and always store a cache entry for the function argument/attribute value pair. There is no guarantee that RACR handles this issue, because there is no guaranteed way in Scheme to decide if two anonymous function entities are actually the same function (RACR uses* `equal?` *for parameter comparison). A similar caching issue arises if attribute parameters can be AST nodes. Consider a node that has been argument of an attribute is deleted by a rewrite. Even the node is deleted, it and the AST it spans will still be stored as key in the cache of the attribute. It is only deleted from the cache of the attribute, if the cache of the attribute is flushed because of an AST rewrite influencing its value (including the special case, that the attribute is influenced by the deleted node).*

```
(specify-attribute spec
  'att ; Define the attribute att ...
  'N   ; in the context of N nodes their ...
  'B   ; B child (thus, the attribute is inherited). Further, the attribute is ...
  #f   ; not cached ,...
  (lambda (n para) ; parameterised (one parameter named para) and...
```

```
    ...)
   (cons ; circular .
     bottom-value
     equivalence-function)) ; E.g., equal?
 ; Meta specification : Specify an attribute using another attribute grammar:
 (apply
   specify-attribute
   (att-value 'attribute-computing-attribute-definition meta-compiler-ast))
```

```
(ag-rule
  attribute-name
  ; Arbitrary many, but at least one, definitions of any of the following forms:
  ((non-terminal context-name) equation) ; Default: cached and non-circular
  ((non-terminal context-name) cached? equation)
  ((non-terminal context-name) equation bottom equivalence-function)
  ((non-terminal context-name) cached? equation bottom equivalence-function)
  (non-terminal equation) ; No context name = synthesized attribute
  (non-terminal cached? equation)
  (non-terminal equation bottom equivalence-function)
  (non-terminal cached? equation bottom equivalence-function))
; attribute-name, non-terminal, context-name: Scheme identifiers , not symbols!
```

Syntax definition which eases the specification of attributes by:

- Permitting the specification of arbitrary many definitions for a certain attribute for different contexts without the need to repeat the attribute name several times
- Automatic quoting of attribute names (thus, the given name must be an ordinary identifier)
- Automatic quoting of non-terminals and context names (thus, contexts must be ordinary identifiers)
- Optional caching and circularity information (by default caching is enabled and attribute definitions are non-circular)
- Context names of synthesized attribute definitions can be left

The `ag-rule` form exists only for convenient reasons. All its functionalities can also be achieved using `specify-attribute`.

Note: *Sometimes attribute definitions shall be computed by a Scheme function rather than being statically defined. In such cases the* `ag-rule` *form is not appropriate, because it expects identifiers for the attribute name and contexts. Moreover, the automatic context name quoting prohibits the specification of contexts using child indices. The* `specify-attribute` *function must be used instead.*

```
(compile-ag-specifications spec)
```

Calling this function finishes the AG specification phase of the given *RACR* specification, such that it is now in the evaluation phase where ASTs can be instantiated, evaluated, annotated and rewritten. An exception is thrown, if the given specification is not in the AG specification phase.

4.2. Evaluation and Querying

`(att-value attribute-name node . arguments)`

Given a node, return the value of one of its attribute instances. In case no proper attribute instance is associated with the node itself, the search is extended to find a broadcast solution. If required, the found attribute instance is evaluated, whereupon all its meta-information like dependencies etc. are computed. The function has a variable number of arguments, whereas its optional parameters are the actual arguments for parameterized attributes. An exception is thrown, if the given node is a bud-node, no properly named attribute instance can be found, the wrong number of arguments is given, the attribute instance depends on itself but its definition is not declared to be circular or the attribute equation is erroneous (i.e., its evaluation aborts with an exception).

```
; Let n be an AST node:
(att-value 'att n) ; Query attribute instance of n that represents attribute att
(att-value 'lookup n "myVar") ; Query parameterised attribute with one argument
; Dynamic attribute dispatch:
(att-value
  (att-value 'attribute-computing-attribute-name n)
  (att-value 'reference-attribute-computing-AST-node n))
```

5. Rewriting

A very common compiler construction task is to incrementally change the structure of ASTs and evaluate some of their attributes in-between. Typical examples are interactive editors with static semantic analyses, code optimisations or incremental AST transformations. In such scenarios, some means to rewrite (partially) evaluated ASTs, without discarding already evaluated and still valid attribute values, is required. On the other hand, the caches of evaluated attributes, whose value can change because of an AST manipulation, must be flushed. Attribute grammar systems supporting such a behaviour are called incremental. *RACR* supports incremental attribute evaluation in the form of rewrite functions. The rewrite functions of *RACR* provide an advanced and convenient interface to perform complex AST manipulations and ensure optimal incremental attribute evaluation (i.e., rewrites only flush the caches of the attributes they influence).

Of course, rewrite functions can be arbitrary applied within complex *Scheme* programs. In particular, attribute values can be used to compute the rewrites to apply, e.g., rewrites may be only applied for certain program execution paths with the respective control-flow depending on attribute values. However, *RACR* does not permit rewrites throughout the evaluation of an attribute associated with the rewritten AST. The reason for this restriction is, that rewrites within attribute equations can easily yield unexpected results, because the final AST resulting after evaluating all attributes queried can depend on the order of queries (e.g., the order in which a user accesses attributes for their value). By prohibiting rewrites during attribute evaluation, *RACR* protects users before non-confluent behaviour.

Additionally, *RACR* ensures, that rewrites always yield valid ASTs. It is not permitted to insert an AST fragment into a context expecting a fragment of different type or to insert a single AST fragment into several different ASTs, into several places within the same AST or into its own subtree using rewrites. In case of violation, the respective rewrite throws a runtime exception. The reason for this restrictions are, that attribute grammars are not defined for arbitrary graphs but only for trees.

Figure 5.1 summarises the conditions under which *RACR's* rewrite functions throw runtime exceptions. Marks denote exception cases. E.g., applications of `rewrite-add` whereat the context `l` is not a list-node are not permitted. Rewrite exceptions are thrown at runtime, because in general it is impossible to check for proper rewriting using source code analyses. *Scheme* is Turing complete and ASTs, rewrite applications and their arguments can be computed by arbitrary *Scheme* programs.

5.1. Primitive Rewrite Functions

		`(rewrite-terminal n i v)`	`(rewrite-refine n t . c)`	`(rewrite-abstract n t)`	`(rewrite-add l e)`	`(rewrite-insert l i e)`	`(rewrite-delete n)`	`(rewrite-subtree n n2)`
Context	Not AST Node	×	×	×	×	×	×	×
	Bud-Node	×	×	×	×	×	×	
	List-Node	×	×	×			×	
	Not List-Node				×	×		
	Not Element of List-Node						×	
New Node(s)	Wrong Number		×					
	Do not fit		×		×	×		×
	No Root(s)		×		×	×		×
	Context is in Subtree		×		×	×		×
New Type	Not AST Node Type		×	×				
	Not Subtype of Context		×					
	Not Supertype of Context			×				
Attribute(s) in Evaluation		×	×	×	×	×	×	×
Child does not exist		×				×		
Child is AST Node		×						

Context: `n`, `l` New Nodes: `c`, `e`, `n2` New Type: `t`

Figure 5.1.: Runtime Exceptions of RACR's Primitive Rewrite Functions

`(rewrite-terminal i n new-value)`

Given a node `n`, a child index `i` and an arbitrary value `new-value`, change the value of `n`'s `i`'th child, which must be a terminal, to `new-value`. Thereby, the caches of any influenced attributes are flushed and dependencies are maintained. An exception is thrown, if `n` has no `i`'th child, `n`'s `i`'th child is no terminal or any attributes of the AST `n` is part of are in evaluation.

`(rewrite-refine n t . c)`

Given a node `n` of arbitrary type, a non-terminal type `t`, which is a subtype of `n`'s current type, and arbitrary many non-terminal nodes and terminal values `c`, rewrite the type of `n` to `t` and add `c` as children for the additional contexts `t` introduces compared to `n`'s current type. Thereby, the caches of any influenced attributes are flushed and dependencies are maintained. An exception is thrown, if `t` is no subtype of `n`, not enough or to much additional context children are given, any of the additional context children does not fit, any attributes of the AST `n` is part of or of any of the ASTs spaned by the additional children are in evaluation, any of the additional children already is part of another AST or `n` is within the AST of any of the additional children.

Note: *Since list-, bud- and terminal nodes have no type, they cannot be refined.*

```
(let* ((spec (create-specification))
       (A
        (with-specification
         spec
         (ast-rule 'S->A)
         (ast-rule 'A->a)
         (ast-rule 'Aa:A->b-c)
         (compile-ast-specifications 'S)
         (compile-ag-specifications)
         (ast-child 'A
          (create-ast
           'S
           (list
            (create-ast 'A (list 1))))))))
  (assert (= (ast-num-children A) 1))
  (assert (eq? (ast-node-type A) 'A))
  ; Refine an A node to an Aa node. Note, that Aa nodes have two
  ; additional child contexts beside the one they inherit:
  (rewrite-refine A 'Aa 2 3)
  (assert (= (ast-num-children A) 3))
  (assert (eq? (ast-node-type A) 'Aa))
  (assert (= (- (ast-child 'c A) (ast-child 'a A)) (ast-child 'b A))))
```

`(rewrite-abstract n t)`

Given a node `n` of arbitrary type and a non-terminal type `t`, which is a supertype of `n`'s current type, rewrite the type of `n` to `t`. Superfluous children of `n` representing child contexts not known anymore by `n`'s new type `t` are deleted. Further, the caches of any influenced attributes are flushed and dependencies are maintained. An exception is thrown, if `t` is not a supertype of `n`'s current type or any attributes of the AST `n` is part of are in evaluation. If rewriting succeeds, a list containing the deleted superfluous children in their original order is returned.

Note: *Since list-, bud- and terminal nodes have no type, they cannot be abstracted.*

```
(let* ((spec (create-specification))
       (A
        (with-specification
         spec
         (ast-rule 'S->A)
         (ast-rule 'A->a)
         (ast-rule 'Aa:A->b-c)
         (compile-ast-specifications 'S)
         (compile-ag-specifications)
         (ast-child 'A
          (create-ast
           'S
           (list
            (create-ast 'Aa (list 1 2 3))))))))
  (assert (= (ast-num-children A) 3))
```

```
(assert (eq? (ast-node-type A) 'Aa))
; Abstract an Aa node to an A node. Note, that A nodes have two
; less child contexts than Aa nodes:
(rewrite-abstract A 'A)
(assert (= (ast-num-children A) 1))
(assert (eq? (ast-node-type A) 'A)))
```

`(rewrite-subtree old-fragment new-fragment)`

Given an AST node to replace (`old-fragment`) and its replacement (`new-fragment`) replace `old-fragment` by `new-fragment`. Thereby, the caches of any influenced attributes are flushed and dependencies are maintained. An exception is thrown, if `new-fragment` does not fit, `old-fragment` is not part of an AST (i.e., has no parent node), any attributes of either fragment are in evaluation, `new-fragment` already is part of another AST or `old-fragment` is within the AST spaned by `new-fragment`. If rewriting succeeds, the removed `old-fragment` is returned.

Note: *Besides ordinary node replacement also list-node replacement is supported. In case of a list-node replacement* `rewrite-subtree` *checks, that the elements of the replacement list* `new-fragment` *fit w.r.t. their new context.*

`(rewrite-add l e)`

Given a list-node `l` and another node `e` add `e` to `l`'s list of children (i.e., e becomes an element of `l`). Thereby, the caches of any influenced attributes are flushed and dependencies are maintained. An exception is thrown, if `l` is not a list-node, `e` does not fit w.r.t. `l`'s context, any attributes of either `l` or `e` are in evaluation, `e` already is part of another AST or `l` is within the AST spaned by `e`.

`(rewrite-insert l i e)`

Given a list-node `l`, a child index `i` and an AST node `e`, insert `e` as `i`'th element into `l`. Thereby, the caches of any influenced attributes are flushed and dependencies are maintained. An exception is thrown, if `l` is no list-node, `e` does not fit w.r.t. `l`'s context, `l` has not enough elements, such that no `i`'th position exists, any attributes of either `l` or `e` are in evaluation, `e` already is part of another AST or `l` is within the AST spaned by `e`.

`(rewrite-delete n)`

Given a node `n`, which is element of a list-node (i.e., its parent node is a list-node), delete it within the list. Thereby, the caches of any influenced attributes are flushed and dependencies are maintained. An exception is thrown, if `n` is no list-node element or any attributes of the AST it is part of are in evaluation. If rewriting succeeds, the deleted list element `n` is returned.

5.2. Rewrite Strategies

```
(perform-rewrites n strategy . transformers)
```

Given an AST root `n`, a strategy for traversing the subtree spaned by `n` and a set of transformers, apply the transformers on the nodes visited by the given strategy until no further transformations are possible (i.e., a normal form is established). Each transformer is a function with a single parameter which is the node currently visited by the strategy. The visit strategy applies each transformer on the currently visited node until either, one matches (i.e., performs a rewrite) or all fail. Thereby, each transformer decides, if it performs any rewrite for the currently visited node. If it does, it performs the rewrite and returns a truth value equal to `#t`, otherwise `#f`. If all transformers failed (i.e., non performed any rewrite), the visit strategy selects the next node to visit. If any transformer matched (i.e., performed a rewrite), the visit strategy is reseted and starts all over again. If the visit strategy has no further node to visit (i.e., all nodes to visit have been visited and no transformer matched) `perform-rewrites` terminates.

`Perform-rewrites` supports two general visit strategies, both deduced form term rewriting: (1) outermost (leftmost redex) and (2) innermost (rightmost redex) rewriting. In terms of ASTs, outermost rewriting prefers to rewrite the node closest to the root (top-down rewriting), whereas innermost rewriting only rewrites nodes when there does not exist any applicable rewrite within their subtree (bottom-up rewriting). In case several topmost or bottommost rewritable nodes exist, the leftmost is preferred in both approaches. The strategies can be selected by using `'top-down` and `'bottom-up` respectively as strategy argument.

An exception is thrown by `perform-rewrites`, if the given node `n` is no AST root or any applied transformer changes its root status by inserting it into some AST. Exceptions are also thrown, if the given transformers are not functions of arity one or do not accept an AST node as argument.

When terminating, `perform-rewrites` returns a list containing the respective result returned by each applied transformer in the order of their application (thus, the length of the list is the total number of transformations performed).

Note: *Transformers must realise their actual rewrites using primitive rewrite functions; They are responsible to ensure all constraints of applied primitive rewrite functions are satisfied since the rewrite functions throw exceptions as usual in case of any violation.*

Note: *It is the responsibility of the user to ensure, that transformers are properly implemented, i.e., they return true if, and only if, they perform any rewrite and if they perform a rewrite the rewrite does not cause any exception. In particular,* `perform-rewrites` *has no control about performed rewrites for which reason it is possible to implement a transformer violating the intension of a rewrite strategy, e.g., a transformer traversing the AST on its own and thereby rewriting arbitrary parts.*

6. AST Annotations

Often, additional information or functionalities, which can arbitrarily change or whose value and behaviour depends on time, have to be supported by ASTs. Examples are special node markers denoting certain imperative actions or stateful functions for certain AST nodes. Attributes are not appropriate in such cases, since their intension is to be side-effect free, such that their value does not depend on their query order or if they are cached. Further, it is not possible to arbitrarily attach attributes to ASTs. Equal contexts will always use equal attribute definitions for their attribute instances. To realise stateful or side-effect causing node dependent functionalities, the annotation API of *RACR* can be used. AST annotations are named entities associated with AST nodes that can be arbitrarily attached, detached, changed and queried. Thereby, annotation names are ordinary *Scheme* symbols and their values are arbitrary *Scheme* entities. However, to protect users against misuse, *RACR* does not permit, throughout the evaluation of an attribute, the application of any annotation functionalities on (other) nodes within the same AST the attribute is associated with.

6.1. Attachment

```
(ast-annotation-set! n a v)
```

Given a node `n`, a *Scheme* symbol `a` representing an annotation name and an arbitrary value `v`, add an annotation with name `a` and value `v` to `n`. If `n` already has an annotation named `a`, set its value to `v`. If `v` is a function, the value of the annotation is a function calling `v` with the node the annotation is associated with (i.e., `n`) as first argument and arbitrary many further given arguments. An exception is thrown if any attributes of the AST `n` is part of are in evaluation.

Note: *Since terminal nodes as such cannot be retrieved (cf.* `ast-child`*), but only their value, the annotation of terminal nodes is not possible.*

```
(let ((n (function-returning-an-ast)))
  ; Attach annotations:
  (ast-annotation-set! n 'integer-value 3)
  (ast-annotation-set!
   n
   'function-value
   (lambda (associated-node integer-argument)
     integer-argument))
  ; Query annotations:
  (assert
```

```
(=
 (ast-annotation n 'integer-value)
 ; Apply the value of the 'function—value annotation. Note, that
 ; the returned function has one parameter (integer—argument). The
 ; associated—node parameter is automatically bound to n:
 ((ast-annotation n 'function-value) 3))))
```

`(ast-weave-annotations n t a v)`

Given a node `n` spanning an arbitrary AST fragment, a node type `t` and an annotation name `a` and value `v`, add to each node of type `t` of the fragment, which does not yet have an equally named annotation, the given annotation using `ast-annotation-set!`. An exception is thrown, if any attributes of the AST `n` is part of are in evaluation.

Note: *To annotate all list- or bud-nodes within ASTs,* `'list-node` *or* `'bud-node` *can be used as node type* `t` *respectively.*

`(ast-annotation-remove! n a)`

Given a node `n` and an annotation name `a`, remove any equally named annotation associated with `n`. An exception is thrown, if any attributes of the AST `n` is part of are in evaluation.

6.2. Querying

`(ast-annotation? n a)`

Given a node `n` and an annotation name `a`, return whether `n` has an annotation with name `a` or not. An exception is thrown, if any attributes of the AST `n` is part of are in evaluation.

`(ast-annotation n a)`

Given a node `n` and an annotation name `a`, return the value of the respective annotation of `n` (i.e., the value of the annotation with name `a` that is associated with the node `n`). An exception is thrown, if `n` has no such annotation or any attributes of the AST it is part of are in evaluation.

7. Support API

```
(with-specification
  expression-yielding-specification
  ; Arbitrary many further expressions:
  ...)
```

Syntax definition which eases the use of common *RACR* library functions by providing an environment where mandatory *RACR* specification parameters are already bound to a given specification. The `with-specification` form defines for every *RACR* function with a specification parameter an equally named version without the specification parameter and uses the value of its first expression argument as default specification for the newly defined functions (colloquially explained, it rebinds the *RACR* functions with specification parameters to simplified versions where the specification parameters are already bounded). The scope of the simplified functions are the expressions following the first one. Similarly to the `begin` form, `with-specification` evaluates each of its expression arguments in sequence and returns the value of its last argument. If the value of the last argument is not defined, also the value of `with-specification` is not defined.

```
(assert
 (=
  (att-value
   'length
   (with-specification
    (create-specification)

    (ast-rule 'S->List)
    (ast-rule 'List->)
    (ast-rule 'NonNil:List->elem-List<Rest)
    (ast-rule 'Nil:List->)
    (compile-ast-specifications 'S)

    (ag-rule
     length
     (S
      (lambda (n)
        (att-value 'length (ast-child 'List n))))
     (NonNil
      (lambda (n)
        (+ (att-value 'length (ast-child 'Rest n)) 1)))
     (Nil
      (lambda (n)
        0)))
```

```
     (compile-ag-specifications)

     (create-ast 'S (list
      (create-ast 'NonNil (list
       1
       (create-ast 'NonNil (list
        2
        (create-ast 'Nil (list)))))))))
   2))
```

`(specification-phase spec)`

Given a *RACR* specification, return in which specification phase it currently is. Possible return values are:

- AST specification phase: `1`
- AG specification phase: `2`
- Evaluation phase: `3`

```
(let ((spec (create-specification)))
  (assert (= (specification-phase spec) 1))
  (ast-rule spec 'S->)
  (compile-ast-specifications spec 'S)
  (assert (= (specification-phase spec) 2))
  (compile-ag-specifications spec)
  (assert (= (specification-phase spec) 3)))
```

Appendix

A. *RACR* Source Code

```
; This program and the accompanying materials are made available under the
; terms of the MIT license (X11 license) which accompanies this distribution.

; Author: C. Bürger

#!r6rs

(library
 (racr)
 (export
  ; Specification interface:
  (rename (make-racr-specification create-specification))
  (rename (racr-specification-specification-phase specification-phase))
  with-specification
  (rename (specify-ast-rule ast-rule))
  (rename (specify-ag-rule ag-rule))
  specify-attribute
  compile-ast-specifications
  compile-ag-specifications
  ; AST annotation interface:
  ast-weave-annotations
  ast-annotation?
  ast-annotation
  ast-annotation-set!
  ast-annotation-remove!
  ; AST & attribute query interface:
  create-ast
  create-ast-list
  create-ast-bud
  (rename (node? ast-node?))
  ast-node-type
  ast-list-node?
  (rename (node-bud-node? ast-bud-node?))
  ast-subtype?
  ast-parent
  ast-child
  ast-sibling
  ast-child-index
  ast-num-children
  ast-children
  ast-for-each-child
  ast-find-child
  att-value
  ; Rewrite interface:
  perform-rewrites
  rewrite-terminal
  rewrite-refine
  rewrite-abstract
  rewrite-subtree
  rewrite-add
  rewrite-insert
  rewrite-delete
  ; Utility interface:
  print-ast
  racr-exception?)
 (import (rnrs) (rnrs mutable-pairs))

 ;              ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 ;      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;      Internal Data Structures      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 ;              ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

 ; Constructor for unique entities internally used by the RACR system
 (define-record-type racr-nil-record (sealed #t) (opaque #t))
 (define racr-nil (make-racr-nil-record)) ; Unique value indicating undefined RACR entities

 ; Record type representing RACR compiler specifications. A compiler specification consists of arbitrary
 ; many AST rule, attribute and rewrite specifications, all aggregated into a set of rules stored in a
 ; non-terminal-symbol -> ast-rule hashtable, an actual compiler specification phase and a distinguished
 ; start symbol. The specification phase is an internal flag indicating the RACR system the compiler's
 ; specification progress. Possible phases are:
 ; 1 : AST specification
 ; 2 : AG specification
```

```
; 3 : Rewrite specification
; 4 : Specification finished
(define-record-type racr-specification
  (fields (mutable specification-phase) rules-table (mutable start-symbol))
  (protocol
   (lambda (new)
     (lambda ()
       (new 1 (make-eq-hashtable 50) racr-nil)))))

; INTERNAL FUNCTION: Given a RACR specification and a non−terminal, return the
; non−terminal's AST rule or #f if it is undefined.
(define racr-specification-find-rule
  (lambda (spec non-terminal)
    (hashtable-ref (racr-specification-rules-table spec) non-terminal #f)))

; INTERNAL FUNCTION: Given a RACR specification return a list of its AST rules.
(define racr-specification-rules-list
  (lambda (spec)
    (call-with-values
     (lambda () (hashtable-entries (racr-specification-rules-table spec)))
     (lambda (key-vector value-vector)
       (vector->list value-vector)))))

; Record type for AST rules; An AST rule has a reference to the RACR specification it belongs to and consist
; of its symbolic encoding, a production (i.e., a list of production−symbols) and an optional supertype.
(define-record-type ast-rule
  (fields specification as-symbol (mutable production) (mutable supertype)))

; INTERNAL FUNCTION: Given two rules r1 and r2, return whether r1 is a subtype of r2 or not. The subtype
; relationship is reflexive, i.e., every type is a subtype of itself.
(define ast-rule-subtype?
  (lambda (r1 r2)
    (and
     (eq? (ast-rule-specification r1) (ast-rule-specification r2))
     (let loop ((r1 r1))
       (cond
         ((eq? r1 r2) #t)
         ((ast-rule-supertype r1) (loop (ast-rule-supertype r1)))
         (else #f))))))

; INTERNAL FUNCTION: Given a rule, return a list containing all its subtypes except the rule itself.
(define ast-rule-subtypes
  (lambda (rule1)
    (filter
     (lambda (rule2)
       (and (not (eq? rule2 rule1)) (ast-rule-subtype? rule2 rule1)))
     (racr-specification-rules-list (ast-rule-specification rule1)))))

; Record type for production symbols; A production symbol has a name, a flag indicating whether it is a
; non−terminal or not (later resolved to the actual AST rule representing the respective non−terminal), a
; flag indicating whether it represents a Kleene closure (i.e., is a list of certain type) or not, a
; context−name unambiguously referencing it within the production it is part of and a list of attributes
; defined for it.
(define-record-type (symbol make-production-symbol production-symbol?)
  (fields name (mutable non-terminal?) kleene? context-name (mutable attributes)))

; Record type for attribute definitions. An attribute definition has a certain name, a definition context
; consisting of an AST rule and an attribute position (i.e., a (ast−rule position) pair), an equation, and
; an optional circularity−definition needed for circular attributes' fix−point computations. Further,
; attribute definitions specify whether the value of instances of the defined attribute are cached.
; Circularity−definitions are (bottom−value equivalence−function) pairs, whereby bottom−value is the value
; fix−point computations start with and equivalence−functions are used to decide whether a fix−point is
; reached or not (i.e., equivalence−functions are arbitrary functions of arity two computing whether two
; given arguments are equal or not).
(define-record-type attribute-definition
  (fields name context equation circularity-definition cached?))

; INTERNAL FUNCTION: Given an attribute definition, check if instances can depend on
; themself (i.e., be circular) or not.
(define attribute-definition-circular?
  (lambda (att)
    (attribute-definition-circularity-definition att)))

; INTERNAL FUNCTION: Given an attribute definition, return whether it specifies
; a synthesized attribute or not.
(define attribute-definition-synthesized?
  (lambda (att-def)
    (= (cdr (attribute-definition-context att-def)) 0)))

; INTERNAL FUNCTION: Given an attribute definition, return whether it specifies
; an inherited attribute or not.
(define attribute-definition-inherited?
  (lambda (att-def)
    (not (attribute-definition-synthesized? att-def))))

; Record type for AST nodes. AST nodes have a reference to the evaluator state used for evaluating their
```

```
; attributes and rewrites, the AST rule they represent a context of, their parent, children, attribute
; instances, attributes they influence and annotations.
(define-record-type node
  (fields
   (mutable evaluator-state)
   (mutable ast-rule)
   (mutable parent)
   (mutable children)
   (mutable attributes)
   (mutable attribute-influences)
   (mutable annotations))
  (protocol
   (lambda (new)
     (lambda (ast-rule parent children)
       (new
        #f
        ast-rule
        parent
        children
        (list)
        (list)
        (list))))))

; INTERNAL FUNCTION: Given a node, return whether it is a terminal or not.
(define node-terminal?
  (lambda (n)
    (eq? (node-ast-rule n) ’terminal)))

; INTERNAL FUNCTION: Given a node, return whether it is a non−terminal or not.
(define node-non-terminal?
  (lambda (n)
    (not (node-terminal? n))))

; INTERNAL FUNCTION: Given a node, return whether it represents a list of
; children, i.e., is a list−node, or not.
(define node-list-node?
  (lambda (n)
    (eq? (node-ast-rule n) ’list-node)))

; INTERNAL FUNCTION: Given a node, return whether is is a bud−node or not.
(define node-bud-node?
  (lambda (n)
    (eq? (node-ast-rule n) ’bud-node)))

; INTERNAL FUNCTION: Given a node, return its child−index. An exception is thrown,
; if the node has no parent (i.e., is a root).
(define node-child-index
  (lambda (n)
    (if (node-parent n)
        (let loop ((children (node-children (node-parent n)))
                   (pos 1))
          (if (eq? (car children) n)
              pos
              (loop (cdr children) (+ pos 1))))
        (throw-exception
         "Cannot access child-index; "
         "The node has no parent!"))))

; INTERNAL FUNCTION: Given a node find a certain child by name. If the node has
; no such child, return #f, otherwise the child.
(define node-find-child
  (lambda (n context-name)
    (and (not (node-list-node? n))
         (not (node-bud-node? n))
         (not (node-terminal? n))
         (let loop ((contexts (cdr (ast-rule-production (node-ast-rule n))))
                    (children (node-children n)))
           (if (null? contexts)
               #f
               (if (eq? (symbol-context-name (car contexts)) context-name)
                   (car children)
                   (loop (cdr contexts) (cdr children))))))))

; INTERNAL FUNCTION: Given a node find a certain attribute associated with it. If the node
; has no such attribute, return #f, otherwise the attribute.
(define node-find-attribute
  (lambda (n name)
    (find
     (lambda (att)
       (eq? (attribute-definition-name (attribute-instance-definition att)) name))
     (node-attributes n))))

; INTERNAL FUNCTION: Given two nodes n1 and n2, return whether n1 is within the subtree spaned by n2 or not.
(define node-inside-of?
  (lambda (n1 n2)
    (cond
```

```
      ((eq? n1 n2) #t)
      ((node-parent n1) (node-inside-of? (node-parent n1) n2))
      (else #f))))

; Record type for attribute instances of a certain attribute definition, associated with a certain
; node (context), dependencies, influences, a value cache, a cycle cache and an optional cache for the last
; arguments with which the attribute has been evaluated.
(define-record-type attribute-instance
  (fields
   (mutable definition)
   (mutable context)
   (mutable node-dependencies)
   (mutable attribute-dependencies)
   (mutable attribute-influences)
   value-cache
   cycle-cache
   (mutable args-cache))
  (protocol
   (lambda (new)
     (lambda (definition context)
       (new
        definition
        context
        (list)
        (list)
        (list)
        (make-hashtable equal-hash equal? 1)
        (make-hashtable equal-hash equal? 1)
        racr-nil)))))

; Record type representing the internal state of RACR systems throughout their execution, i.e., while
; evaluating attributes and rewriting ASTs. An evaluator state consists of a flag indicating if the AG
; currently performs a fix−point evaluation, a flag indicating if throughout a fix−point iteration the
; value of an attribute changed and an attribute evaluation stack used for dependency tracking.
(define-record-type evaluator-state
  (fields (mutable ag-in-cycle?) (mutable ag-cycle-change?) (mutable att-eval-stack))
  (protocol
   (lambda (new)
     (lambda ()
       (new #f #f (list))))))

; INTERNAL FUNCTION: Given an evaluator state, return whether it represents an evaluation in progress or
; not; If it represents an evaluation in progress return the current attribute in evaluation, otherwise #f.
(define evaluator-state-in-evaluation?
  (lambda (state)
    (and (not (null? (evaluator-state-att-eval-stack state))) (car (evaluator-state-att-eval-stack state)))))

;              ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;       ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;         Utility         ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;              ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; INTERNAL FUNCTION: Given an arbitrary Scheme entity, construct a string
; representation of it using display.
(define object->string
  (lambda (x)
    (call-with-string-output-port
     (lambda (port)
       (display x port)))))

(define-condition-type racr-exception &non-continuable make-racr-exception racr-exception?)

; INTERNAL FUNCTION: Given an arbitrary sequence of strings and other Scheme entities, concatenate them to
; form an error message and throw a special RACR exception with the constructed message. Any entity that is
; not a string is treated as error information embedded in the error message between [ and ] characters,
; whereby the actual string representation of the entity is obtained using object−>string.
(define-syntax throw-exception
  (syntax-rules ()
    ((_ m-part ...)
     (raise
      (condition
       (make-racr-exception)
       (make-message-condition
        (string-append
         "RACR exception: "
         (let ((m-part* m-part))
           (if (string? m-part*)
               m-part*
               (string-append "[" (object->string m-part*) "]"))) ...)))))))

; INTERNAL FUNCTION: Procedure sequentially applying a function on all the AST rules of a set of rules which
; inherit, whereby supertypes are processed before their subtypes.
(define apply-wrt-ast-inheritance
  (lambda (func rules)
    (let loop ((resolved ; The set of all AST rules that are already processed....
                (filter ; ...Initially it consists of all the rules that have no supertypes.
                 (lambda (rule)
```

```
                   (not (ast-rule-supertype rule)))
                 rules))
               (to-check ; The set of all AST rules that still must be processed....
                (filter ; ...Initially it consists of all the rules that have supertypes.
                 (lambda (rule)
                   (ast-rule-supertype rule))
                 rules)))
      (let ((to-resolve ; ...Find a rule that still must be processed and...
             (find
              (lambda (rule)
                (memq (ast-rule-supertype rule) resolved)) ; ...whose supertype already has been processed....
              to-check)))
        (when to-resolve ; ...If such a rule exists,...
          (func to-resolve) ; ...process it and...
          (loop (cons to-resolve resolved) (remq to-resolve to-check))))))) ; ...recur.

;              ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;       ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;       Support API       ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;              ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Given an AST, an association list L of attribute pretty−printers and an output port, print a
; human−readable ASCII representation of the AST on the output port. The elements of the association list
; L are (attribute−name pretty−printing−function) pairs. Every attribute for which L contains an entry is
; printed when the AST node it is associated with is printed. Thereby, the given pretty printing function
; is applied to the attribute’s value before printing it. Beware: The output port is never closed by this
; function − neither in case of an io−exception nor after finishing printing the AST.
(define print-ast
  (lambda (ast attribute-pretty-printer-list output-port)
    (letrec ((print-indentation
              (lambda (n)
                (if (> n 0)
                    (begin
                      (print-indentation (- n 1))
                      (my-display " |"))
                    (my-display #\newline))))
             (my-display
              (lambda (to-display)
                (display to-display output-port))))
      (let loop ((ast-depth 0)
                 (ast ast))
        (cond
          ((node-list-node? ast) ; Print list nodes
           (print-indentation ast-depth)
           (print-indentation ast-depth)
           (my-display "-* ")
           (my-display
            (symbol->string
             (symbol-name
              (list-ref
               (ast-rule-production (node-ast-rule (node-parent ast)))
               (ast-child-index ast)))))
           (for-each
            (lambda (element)
              (loop (+ ast-depth 1) element))
            (node-children ast)))
          ((node-bud-node? ast) ; Print bud nodes
           (print-indentation ast-depth)
           (print-indentation ast-depth)
           (my-display "-@ bud-node"))
          ((node-non-terminal? ast) ; Print non−terminal
           (print-indentation ast-depth)
           (print-indentation ast-depth)
           (my-display "-\\ ")
           (my-display (symbol->string (ast-node-type ast)))
           (for-each
            (lambda (att)
              (let* ((name (attribute-definition-name (attribute-instance-definition att)))
                     (pretty-printer-entry (assq name attribute-pretty-printer-list)))
                (when pretty-printer-entry
                  (print-indentation (+ ast-depth 1))
                  (my-display " <")
                  (my-display (symbol->string name))
                  (my-display "> ")
                  (my-display ((cdr pretty-printer-entry) (att-value name ast))))))
            (node-attributes ast))
           (for-each
            (lambda (child)
              (loop (+ ast-depth 1) child))
            (node-children ast)))
          (else ; Print terminal
           (print-indentation ast-depth)
           (my-display "- ")
           (my-display (node-children ast)))))
      (my-display #\newline))))

(define-syntax with-specification
```

```
  (lambda (x)
    (syntax-case x ()
      ((k spec body ...)
       #`(let* ((spec* spec))
                (#,(datum->syntax #'k 'ast-rule)
                 (lambda (rule)
                   (specify-ast-rule spec* rule)))
                (#,(datum->syntax #'k 'compile-ast-specifications)
                 (lambda (start-symbol)
                   (compile-ast-specifications spec* start-symbol)))
                (#,(datum->syntax #'k 'compile-ag-specifications)
                 (lambda ()
                   (compile-ag-specifications spec*)))
                (#,(datum->syntax #'k 'create-ast)
                 (lambda (rule children)
                   (create-ast spec* rule children)))
                (#,(datum->syntax #'k 'specification-phase)
                 (lambda ()
                   (racr-specification-specification-phase spec*)))
                (#,(datum->syntax #'k 'specify-attribute)
                 (lambda (att-name non-terminal index cached? equation circ-def)
                   (specify-attribute spec* att-name non-terminal index cached? equation circ-def))))
           (let-syntax ((#,(datum->syntax #'k 'ag-rule)
                         (syntax-rules ()
                           ((_ attribute-name definition (... ...))
                            (specify-ag-rule spec* attribute-name definition (... ...))))))
             body ...))))))

;             ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;      Abstract Syntax Tree Annotations   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;             ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define ast-weave-annotations
  (lambda (node type name value)
    (when (evaluator-state-in-evaluation? (node-evaluator-state node))
      (throw-exception
       "Cannot weave " name " annotation; "
       "There are attributes in evaluation."))
    (when (not (ast-annotation? node name))
      (cond
        ((and (not (node-list-node? node)) (not (node-bud-node? node)) (ast-subtype? node type))
         (ast-annotation-set! node name value))
        ((and (node-list-node? node) (eq? type 'list-node))
         (ast-annotation-set! node name value))
        ((and (node-bud-node? node) (eq? type 'bud-node))
         (ast-annotation-set! node name value))))
    (for-each
     (lambda (child)
       (unless (node-terminal? child)
         (ast-weave-annotations child type name value)))
     (node-children node))))

(define ast-annotation?
  (lambda (node name)
    (when (evaluator-state-in-evaluation? (node-evaluator-state node))
      (throw-exception
       "Cannot check for " name " annotation; "
       "There are attributes in evaluation."))
    (assq name (node-annotations node))))

(define ast-annotation
  (lambda (node name)
    (when (evaluator-state-in-evaluation? (node-evaluator-state node))
      (throw-exception
       "Cannot access " name " annotation; "
       "There are attributes in evaluation."))
    (let ((annotation (ast-annotation? node name)))
      (if annotation
          (cdr annotation)
          (throw-exception
           "Cannot access " name " annotation; "
           "The given node has no such annotation.")))))

(define ast-annotation-set!
  (lambda (node name value)
    (when (evaluator-state-in-evaluation? (node-evaluator-state node))
      (throw-exception
       "Cannot set " name " annotation; "
       "There are attributes in evaluation."))
    (when (not (symbol? name))
      (throw-exception
       "Cannot set " name " annotation; "
       "Annotation names must be Scheme symbols."))
    (let ((annotation (ast-annotation? node name))
          (value
           (if (procedure? value)
```

```
               (lambda args
                 (apply value node args))
               value)))
      (if annotation
          (set-cdr! annotation value)
          (node-annotations-set! node (cons (cons name value) (node-annotations node)))))))

(define ast-annotation-remove!
  (lambda (node name)
    (when (evaluator-state-in-evaluation? (node-evaluator-state node))
      (throw-exception
       "Cannot remove " name " annotation; "
       "There are attributes in evaluation."))
    (node-annotations-set!
     node
     (remp
      (lambda (entry)
        (eq? (car entry) name))
      (node-annotations node)))))

;             ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;       Abstract Syntax Tree Specifications    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;             ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define specify-ast-rule
  (lambda (spec rule)
    ;;; Ensure, that the RACR system is in the correct specification phase:
    (when (> (racr-specification-specification-phase spec) 1)
      (throw-exception
       "Unexpected AST rule " rule "; "
       "AST rules can only be defined in the AST specification phase."))
    (letrec* ((rule-string (symbol->string rule)) ; String representation of the encoded rule (used for parsing)
              (pos 0) ; The current parsing position
              ; Support function returning, whether the end of the parsing string is reached or not:
              (eos?
               (lambda ()
                 (= pos (string-length rule-string))))
              ; Support function returning the current character to parse:
              (my-peek-char
               (lambda ()
                 (string-ref rule-string pos)))
              ; Support function returning the current character to parse and incrementing the parsing position:
              (my-read-char
               (lambda ()
                 (let ((c (my-peek-char)))
                   (set! pos (+ pos 1))
                   c)))
              ; Support function matching a certain character:
              (match-char!
               (lambda (c)
                 (if (eos?)
                     (throw-exception
                      "Unexpected end of AST rule " rule ";"
                      "Expected " c " character.")
                     (if (char=? (my-peek-char) c)
                         (set! pos (+ pos 1))
                         (throw-exception
                          "Invalid AST rule " rule "; "
                          "Unexpected " (my-peek-char) " character.")))))
              ; Support function parsing a symbol, i.e., retrieving its name, type, if it is a list and optional context−name.
              ; It returns a (name−as−scheme−symbol terminal? klenee? context−name−as−scheme−symbol?) quadrupel:
              (parse-symbol
               (lambda (location) ; location: l−hand, r−hand
                 (let ((symbol-type (if (eq? location ’l-hand) "non-terminal" "terminal")))
                   (when (eos?)
                     (throw-exception
                      "Unexpected end of AST rule " rule "; "
                      "Expected " symbol-type "."))
                   (let* ((parse-name
                           (lambda (terminal?)
                             (let ((name
                                    (append
                                     (let loop ((chars (list)))
                                       (if (and (not (eos?)) (char-alphabetic? (my-peek-char)))
                                           (begin
                                             (when (and terminal? (not (char-lower-case? (my-peek-char))))
                                               (throw-exception
                                                "Invalid AST rule " rule "; "
                                                "Unexpected " (my-peek-char) " character."))
                                             (loop (cons (my-read-char) chars)))
                                           (reverse chars)))
                                     (let loop ((chars (list)))
                                       (if (and (not (eos?)) (char-numeric? (my-peek-char)))
                                           (loop (cons (my-read-char) chars))
                                           (reverse chars))))))
                               (when (null? name)
```

```
                              (throw-exception
                               "Unexpected " (my-peek-char) " character in AST rule " rule "; "
                               "Expected " symbol-type "."))
                             (unless (char-alphabetic? (car name))
                              (throw-exception
                               "Malformed name in AST rule " rule "; "
                               "Names must start with a letter."))
                             name)))
                        (terminal? (char-lower-case? (my-peek-char)))
                        (name (parse-name terminal?))
                        (klenee?
                         (and
                          (not terminal?)
                          (eq? location 'r-hand)
                          (not (eos?))
                          (char=? (my-peek-char) #\*)
                          (my-read-char)))
                        (context-name?
                         (and
                          (not terminal?)
                          (eq? location 'r-hand)
                          (not (eos?))
                          (char=? (my-peek-char) #\<)
                          (my-read-char)
                          (parse-name #f)))
                        (name-string (list->string name))
                        (name-symbol (string->symbol name-string)))
                    (when (and terminal? (eq? location 'l-hand))
                      (throw-exception
                       "Unexpected " name " terminal in AST rule " rule "; "
                       "Left hand side symbols must be non-terminals."))
                    (make-production-symbol
                     name-symbol
                     (not terminal?)
                     klenee?
                     (if context-name?
                         (string->symbol (list->string context-name?))
                         (if klenee?
                             (string->symbol (string-append name-string "*"))
                             name-symbol))
                     (list)))))
             (l-hand (parse-symbol 'l-hand)); The rule's l-hand
             (supertype ; The rule's super-type
              (and (not (eos?)) (char=? (my-peek-char) #\:) (my-read-char) (symbol-name (parse-symbol 'l-hand))))
             (rule* ; Representation of the parsed rule
              (begin
                (match-char! #\-)
                (match-char! #\>)
                (make-ast-rule
                 spec
                 rule
                 (append
                  (list l-hand)
                  (let loop ((r-hand
                              (if (not (eos?))
                                  (list (parse-symbol 'r-hand))
                                  (list))))
                    (if (eos?)
                        (reverse r-hand)
                        (begin
                          (match-char! #\-)
                          (loop (cons (parse-symbol 'r-hand) r-hand))))))
                 supertype))))
            ; Check, that the rule's l-hand is not already defined:
            (when (racr-specification-find-rule spec (symbol-name l-hand))
              (throw-exception
               "Invalid AST rule " rule "; "
               "Redefinition of " (symbol-name l-hand) "."))
            (hashtable-set! ; Add the rule to the RACR system.
             (racr-specification-rules-table spec)
             (symbol-name l-hand)
             rule*))))

(define compile-ast-specifications
  (lambda (spec start-symbol)
    ;;; Ensure, that the RACR system is in the correct specification phase and...
    (let ((current-phase (racr-specification-specification-phase spec)))
      (if (> current-phase 1)
          (throw-exception
           "Unexpected AST compilation; "
           "The AST specifications already have been compiled.")
          ; ... iff so proceed to the next specification phase:
          (racr-specification-specification-phase-set! spec (+ current-phase 1))))

    (racr-specification-start-symbol-set! spec start-symbol)
    (let* ((rules-list (racr-specification-rules-list spec))
```

```
     ; Support function, that given a rule R returns a list of all rules directly derivable from R:
     (derivable-rules
      (lambda (rule*)
        (fold-left
         (lambda (result symb*)
           (if (symbol-non-terminal? symb*)
               (append result (list (symbol-non-terminal? symb*)) (ast-rule-subtypes (symbol-non-terminal? symb*)))
               result))
         (list)
         (cdr (ast-rule-production rule*))))))

;;; Resolve supertypes and non-terminals occuring in productions and ensure all non-terminals are defined:
(for-each
 (lambda (rule*)
   (when (ast-rule-supertype rule*)
     (let ((supertype-entry (racr-specification-find-rule spec (ast-rule-supertype rule*))))
       (if (not supertype-entry)
           (throw-exception
            "Invalid AST rule " (ast-rule-as-symbol rule*) "; "
            "The supertype " (ast-rule-supertype rule*) " is not defined.")
           (ast-rule-supertype-set! rule* supertype-entry))))
   (for-each
    (lambda (symb*)
      (when (symbol-non-terminal? symb*)
        (let ((symb-definition (racr-specification-find-rule spec (symbol-name symb*))))
          (when (not symb-definition)
            (throw-exception
             "Invalid AST rule " (ast-rule-as-symbol rule*) "; "
             "Non-terminal " (symbol-name symb*) " is not defined."))
          (symbol-non-terminal?-set! symb* symb-definition))))
    (cdr (ast-rule-production rule*))))
 rules-list)

;;; Ensure, that inheritance is cycle-free:
(for-each
 (lambda (rule*)
   (when (memq rule* (ast-rule-subtypes rule*))
     (throw-exception
      "Invalid AST grammar; "
      "The definition of " (ast-rule-as-symbol rule*) " depends on itself (cyclic inheritance).")))
 rules-list)

;;; Ensure, that the start symbol is defined:
(unless (racr-specification-find-rule spec start-symbol)
  (throw-exception
   "Invalid AST grammar; "
   "The start symbol " start-symbol " is not defined."))

;;; Ensure, that the start symbol has no super- and subtype:
(let ((supertype (ast-rule-supertype (racr-specification-find-rule spec start-symbol))))
  (when supertype
    (throw-exception
     "Invalid AST grammar; "
     "The start symbol " start-symbol " inherits from " (ast-rule-as-symbol supertype) ".")))
(let ((subtypes (ast-rule-subtypes (racr-specification-find-rule spec start-symbol))))
  (unless (null? subtypes)
    (throw-exception
     "Invalid AST grammar; "
     "The rules " (map ast-rule-as-symbol subtypes) " inherit from the start symbol " start-symbol ".")))

;;; Ensure, that the CFG is start separated:
(let ((start-rule (racr-specification-find-rule spec start-symbol)))
  (for-each
   (lambda (rule*)
     (when (memq start-rule (derivable-rules rule*))
       (throw-exception
        "Invalid AST grammar; "
        "The start symbol " start-symbol " is not start separated because of rule " (ast-rule-as-symbol rule*) ".")))
   rules-list))

;;; Resolve inherited production symbols:
(apply-wrt-ast-inheritance
 (lambda (rule)
   (ast-rule-production-set!
    rule
    (append
     (list (car (ast-rule-production rule)))
     (map
      (lambda (symbol)
        (make-production-symbol
         (symbol-name symbol)
         (symbol-non-terminal? symbol)
         (symbol-kleene? symbol)
         (symbol-context-name symbol)
         (list)))
      (cdr (ast-rule-production (ast-rule-supertype rule))))
```

```
           (cdr (ast-rule-production rule)))))
       rules-list)

      ;;; Ensure context-names are unique:
      (for-each
       (lambda (rule*)
         (let loop ((rest-production (cdr (ast-rule-production rule*))))
           (unless (null? rest-production)
             (let ((current-context-name (symbol-context-name (car rest-production))))
               (when (find
                      (lambda (symb*)
                        (eq? (symbol-context-name symb*) current-context-name))
                      (cdr rest-production))
                 (throw-exception
                  "Invalid AST grammar; "
                  "The context-name " current-context-name " is not unique for rule " (ast-rule-as-symbol rule*) "."))
               (loop (cdr rest-production))))))
       rules-list)

      ;;; Ensure, that all non-terminals can be derived from the start symbol:
      (let* ((to-check (list (racr-specification-find-rule spec start-symbol)))
             (checked (list)))
        (let loop ()
          (unless (null? to-check)
            (let ((rule* (car to-check)))
              (set! to-check (cdr to-check))
              (set! checked (cons rule* checked))
              (for-each
               (lambda (derivable-rule)
                 (when (and
                        (not (memq derivable-rule checked))
                        (not (memq derivable-rule to-check)))
                   (set! to-check (cons derivable-rule to-check))))
               (derivable-rules rule*))
              (loop))))
        (let ((non-derivable-rules
               (filter
                (lambda (rule*)
                  (not (memq rule* checked)))
                rules-list)))
          (unless (null? non-derivable-rules)
            (throw-exception
             "Invalid AST grammar; "
             "The rules " (map ast-rule-as-symbol non-derivable-rules) " cannot be derived."))))

      ;;; Ensure, that all non-terminals are productive:
      (let* ((productive-rules (list))
             (to-check rules-list)
             (productive-rule?
              (lambda (rule*)
                (not (find
                      (lambda (symb*)
                        (and
                         (symbol-non-terminal? symb*)
                         (not (memq (symbol-non-terminal? symb*) productive-rules))))
                      (cdr (ast-rule-production rule*)))))))
        (let loop ()
          (let ((productive-rule
                 (find productive-rule? to-check)))
            (when productive-rule
              (set! to-check (remq productive-rule to-check))
              (set! productive-rules (cons productive-rule productive-rules))
              (loop))))
        (unless (null? to-check)
          (throw-exception
           "Invalid AST grammar; "
           "The rules " (map ast-rule-as-symbol to-check) " are not productive."))))))

;              ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;       Attribute Grammar Specifications     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;              ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-syntax specify-ag-rule
  (lambda (x)
    (syntax-case x ()
      ((_ spec att-name definition ...)
       (and (identifier? #’att-name) (not (null? #’(definition ...))))
       #’(let ((spec* spec)
               (att-name* ’att-name))
           (let-syntax
               ((specify-attribute*
                 (syntax-rules ()
                   ((_ spec* att-name* ((non-terminal index) equation))
                    (specify-attribute spec* att-name* ’non-terminal ’index #t equation #f))
                   ((_ spec* att-name* ((non-terminal index) cached? equation))
                    (specify-attribute spec* att-name* ’non-terminal ’index cached? equation #f))
```

```
                 ((_ spec* att-name* ((non-terminal index) equation bottom equivalence-function))
                  (specify-attribute spec* att-name* 'non-terminal 'index #t equation (cons bottom equivalence-function)))
                 ((_ spec* att-name* ((non-terminal index) cached? equation bottom equivalence-function))
                  (specify-attribute spec* att-name* 'non-terminal 'index cached? equation (cons bottom equivalence-function)))
                 ((_ spec* att-name* (non-terminal equation))
                  (specify-attribute spec* att-name* 'non-terminal 0 #t equation #f))
                 ((_ spec* att-name* (non-terminal cached? equation))
                  (specify-attribute spec* att-name* 'non-terminal 0 cached? equation #f))
                 ((_ spec* att-name* (non-terminal equation bottom equivalence-function))
                  (specify-attribute spec* att-name* 'non-terminal 0 #t equation (cons bottom equivalence-function)))
                 ((_ spec* att-name* (non-terminal cached? equation bottom equivalence-function))
                  (specify-attribute spec* att-name* 'non-terminal 0 cached? equation (cons bottom equivalence-function)))))
            (specify-attribute* spec* att-name* definition) ...))))))

(define specify-attribute
  (lambda (spec attribute-name non-terminal context-name-or-position cached? equation circularity-definition)
    ;;; Before adding the attribute definition, ensure...
    (let ((wrong-argument-type ; ...correct argument types,...
          (or
           (and (not (symbol? attribute-name))
               "Attribute name : symbol")
           (and (not (symbol? non-terminal))
               "AST rule : non-terminal")
           (and (not (symbol? context-name-or-position))
               (or (not (integer? context-name-or-position)) (< context-name-or-position 0))
               "Production position : index or context-name")
           (and (not (procedure? equation))
               "Attribute equation : function")
           (and circularity-definition
               (not (pair? circularity-definition))
               (not (procedure? (cdr circularity-definition)))
               "Circularity definition : #f or (bottom-value equivalence-function) pair"))))
     (when wrong-argument-type
       (throw-exception
        "Invalid attribute definition; "
        "Wrong argument type (" wrong-argument-type ").")))
    (unless (= (racr-specification-specification-phase spec) 2) ; ...that the RACR system is in the correct specification phase,...
     (throw-exception
      "Unexpected " attribute-name " attribute definition; "
      "Attributes can only be defined in the AG specification phase."))
    (let ((ast-rule (racr-specification-find-rule spec non-terminal)))
     (unless ast-rule ; ...the given AST rule is defined,...
       (throw-exception
        "Invalid attribute definition; "
        "The non-terminal " non-terminal " is not defined."))
     (let* ((position ; ...the given context exists,...
             (if (symbol? context-name-or-position)
                (if (eq? context-name-or-position '*)
                    0
                    (let loop ((pos 1)
                              (rest-production (cdr (ast-rule-production ast-rule))))
                      (if (null? rest-production)
                         (throw-exception
                          "Invalid attribute definition; "
                          "The non-terminal " non-terminal " has no " context-name-or-position " context.")
                         (if (eq? (symbol-context-name (car rest-production)) context-name-or-position)
                             pos
                             (loop (+ pos 1) (cdr rest-production))))))
                (if (>= context-name-or-position (length (ast-rule-production ast-rule)))
                    (throw-exception
                     "Invalid attribute definition; "
                     "There exists no " context-name-or-position "'th position in the context of " non-terminal ".")
                    context-name-or-position)))
            (context (list-ref (ast-rule-production ast-rule) position)))
       (unless (symbol-non-terminal? context) ; ...it is a non−terminal and...
         (throw-exception
          "Invalid attribute definition; "
          non-terminal context-name-or-position " is a terminal."))
       ; ...the attribute is not already defined for it:
       (when (memq attribute-name (map attribute-definition-name (symbol-attributes context)))
         (throw-exception
          "Invalid attribute definition; "
          "Redefinition of " attribute-name " for " non-terminal context-name-or-position "."))
       ;;; Everything is fine. Thus, add the definition to the AST rule's respective symbol:
       (symbol-attributes-set!
        context
        (cons
         (make-attribute-definition
          attribute-name
          (cons ast-rule position)
          equation
          circularity-definition
          cached?)
         (symbol-attributes context)))))))

(define compile-ag-specifications
```

```
  (lambda (spec)
    ;;; Ensure, that the RACR system is in the correct specification phase and...
    (let ((current-phase (racr-specification-specification-phase spec)))
      (when (< current-phase 2)
       (throw-exception
        "Unexpected AG compilation; "
        "The AST specifications are not yet compiled."))
      (if (> current-phase 2)
         (throw-exception
          "Unexpected AG compilation; "
          "The AG specifications already have been compiled.")
         (racr-specification-specification-phase-set! spec (+ current-phase 1)))) ; ...if so proceed to the next specification phase.

    ;;; Resolve attribute definitions inherited from a supertype. Thus,...
    (apply-wrt-ast-inheritance ; ...for every AST rule R which has a supertype...
     (lambda (rule)
      (let loop ((super-prod (ast-rule-production (ast-rule-supertype rule)))
                 (sub-prod (ast-rule-production rule)))
        (unless (null? super-prod)
          (for-each ; ...check for every attribute definition of R's supertype...
           (lambda (super-att-def)
             (unless (find ; ...if it is shadowed by an attribute definition of R....
                      (lambda (sub-att-def)
                        (eq? (attribute-definition-name sub-att-def) (attribute-definition-name super-att-def)))
                      (symbol-attributes (car sub-prod)))
               (symbol-attributes-set! ; ...If not, add...
                (car sub-prod)
                (cons
                 (make-attribute-definition ; ...a copy of the attribute definition inherited...
                  (attribute-definition-name super-att-def)
                  (cons rule (cdr (attribute-definition-context super-att-def))) ; ...to R.
                  (attribute-definition-equation super-att-def)
                  (attribute-definition-circularity-definition super-att-def)
                  (attribute-definition-cached? super-att-def))
                 (symbol-attributes (car sub-prod))))))
           (symbol-attributes (car super-prod)))
          (loop (cdr super-prod) (cdr sub-prod)))))
     (racr-specification-rules-list spec))))

;              ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;        Attribute Evaluator      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;              ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; INTERNAL FUNCTION: Given a node n find a certain attribute associated with it, whereas in case no proper
; attribute is associated with n itself the search is extended to find a broadcast solution. Iff the
; extended search finds a solution, appropriate copy propergation attributes (i.e., broadcasters) are added.
; Iff no attribute instance can be found or n is a bud node, an exception is thrown. Otherwise, the
; attribute or its respective last broadcaster is returned.
(define lookup-attribute
  (lambda (name n)
    (when (node-bud-node? n)
      (throw-exception
       "AG evaluator exception; "
       "Cannot access " name " attribute - the given node is a bud."))
    (let loop ((n n)) ; Recursively...
      (let ((att (node-find-attribute n name))) ; ...check if the current node has a proper attribute instance....
       (if att
           att ; ... Iff it has, return the found defining attribute instance.
           (let ((parent (node-parent n))) ; ...Iff no defining attribute instance can be found...
             (if (not parent) ; ...check if there exists a parent node that may provide a definition....
                 (throw-exception ; ...Iff not, throw an exception,...
                  "AG evaluator exception; "
                  "Cannot access unknown " name " attribute.")
                 (let* ((att (loop parent)) ; ...otherwise proceed the search at the parent node. Iff it succeeds...
                        (broadcaster ; ...construct a broadcasting attribute instance...
                         (make-attribute-instance
                          (make-attribute-definition ; ...whose definition context depends...
                           name
                           (if (eq? (node-ast-rule parent) 'list-node) ; ...if the parent node is a list-node or not....
                               (cons ; ... Iff it is a list-node the broadcaster's context is...
                                (node-ast-rule (node-parent parent)) ; ...the list-node's parent node and...
                                (node-child-index parent)) ; ...child position.
                               (cons ; ... Iff the parent node is not a list-node the broadcaster's context is...
                                (node-ast-rule parent) ; ...the parent node and...
                                (node-child-index n))) ; ...the current node's child position. Further,...
                           (lambda (n . args) ; ...the broadcaster's equation just calls the parent node's counterpart. Finally,...
                             (apply att-value name (ast-parent n) args))
                           (attribute-definition-circularity-definition (attribute-instance-definition att))
                           #f)
                          n)))
                   (node-attributes-set! n (cons broadcaster (node-attributes n))) ; ...add the constructed broadcaster and...
                   broadcaster)))))))) ; ...return it as the current node's look-up result.

(define att-value
  (lambda (name n . args)
    (let* (; The evaluator state used and changed throughout evaluation:
```

```
     (evaluator-state (node-evaluator-state n))
     ; The attribute instance to evaluate:
     (att (lookup-attribute name n))
     ; The attribute’s definition:
     (att-def (attribute-instance-definition att))
     ; The attribute’s value cache entry for the given arguments:
     (vc-hit
      (if (attribute-definition-cached? att-def)
          (hashtable-ref (attribute-instance-value-cache att) args racr-nil)
          racr-nil)))
(if (not (eq? vc-hit racr-nil)) ; First, check if the attribute’s value is cached....
    (begin ; ... Iff it is ,...
      ; maintaine attribute dependencies, i.e., iff this attribute is evaluated throughout the evaluation
      ; of another attribute, the other attribute depends on this one. Afterwards,...
      (add-dependency:att->att att)
      vc-hit) ; ...return the attribute’s cached value.
    ; ... Iff the attribute is not cached it must be evaluated. Therefore, prepare a few support values and functions:
    (let* (; The attribute’s computed value to return:
           (result racr-nil)
           ; The attribute’s cycle cache entry for the given arguments:
           (cc-hit (hashtable-ref (attribute-instance-cycle-cache att) args #f))
           ; Boolean value; #t iff the attribute already is in evaluation for the given arguments:
           (entered? (and cc-hit (cdr cc-hit)))
           ; Boolean value; #t iff the attribute is declared to be circular:
           (circular? (attribute-definition-circular? att-def))
           ; Boolean value; #t iff the attribute is declared to be circular and is the starting point for a
           ; fix−point evaluation:
           (start-fixpoint-computation? (and circular? (not (evaluator-state-ag-in-cycle? evaluator-state))))
           ; Support function that checks if the attribute’s value changed throughout fix−point evaluation and
           ; updates its and the evaluator’s state accordingly:
           (update-cycle-cache
            (lambda ()
              (attribute-instance-args-cache-set! att args)
              (unless ((cdr (attribute-definition-circularity-definition att-def))
                       result
                       (car cc-hit))
                (set-car! cc-hit result)
                (evaluator-state-ag-cycle-change?-set! evaluator-state #t)))))
      ; Now, decide how to evaluate the attribute dependening on whether the attribute is circular, already in evaluation
      ; or starting point for a fix−point evaluation:
      (cond
        ; EVALUATION−CASE (1): Circular attribute starting point for a fix−point evaluation:
        (start-fixpoint-computation?
         (let (; Flag indicating abnormal termination of the fix−point evaluation (e.g., by implementation
               ; errors within applied attribute equations and respective exceptions or the application of
               ; a continuation outside the fix−point evaluation’s scope):
               (abnormal-termination? #t))
           (dynamic-wind
            (lambda ()
              ; Maintaine attribute dependencies, i.e., iff this attribute is evaluated throughout the evaluation
              ; of another attribute, the other attribute depends on this one and this attribute must depend on
              ; any other attributes that will be evaluated through its own evaluation. Further,..
              (add-dependency:att->att att)
              (evaluator-state-att-eval-stack-set! evaluator-state (cons att (evaluator-state-att-eval-stack evaluator-state)))
              ; ... update the evaluator state that we are about to start a fix−point evaluation and...
              (evaluator-state-ag-in-cycle?-set! evaluator-state #t)
              ; ... mark, that the attribute is in evaluation and construct an appropriate cycle−cache entry.
              (set! cc-hit (cons (car (attribute-definition-circularity-definition att-def)) #t))
              (hashtable-set! (attribute-instance-cycle-cache att) args cc-hit))
            (lambda ()
              (let loop () ; Start fix−point evaluation. Thus, as long as...
                (evaluator-state-ag-cycle-change?-set! evaluator-state #f) ; ...an attribute’s value changes...
                (set! result (apply (attribute-definition-equation att-def) n args)) ; ...evaluate the attribute,...
                (update-cycle-cache) ; ...update its cycle cache and...
                ; ... check if throughout its evaluation the value of any attribute it depends on changed....
                (when (evaluator-state-ag-cycle-change? evaluator-state) ; ...Iff a value changed,
                  (loop)) ; ...trigger the attribute’s evaluation once more, until a fix−point is reached. Finally,...
                (set! abnormal-termination? #f))) ; ...indicate that the fix−point evaluation terminated normal.
            (lambda ()
              ; Mark that the fix−point evaluation is finished and...
              (evaluator-state-ag-in-cycle?-set! evaluator-state #f)
              ; ... update the caches of all circular attributes evaluated throughout it. To do so,...
              (let loop ((att att))
                (if (not (attribute-definition-circular? (attribute-instance-definition att)))
                    ; ... ignore non−circular attributes and just proceed with the attributes they depend on (to
                    ; ensure all strongly connected components within a weakly connected one are updated)....
                    (for-each
                     loop
                     (attribute-instance-attribute-dependencies att))
                    ; ... In case of circular attributes not yet updated,...
                    (when (> (hashtable-size (attribute-instance-cycle-cache att)) 0)
                      (when (and ; ...check...
                             (not abnormal-termination?) ; ...if the fix−point evaluation terminated normal and...
                             (attribute-definition-cached? (attribute-instance-definition att))) ; ...caching is enabled....
                        (hashtable-set! ; ...Iff so...
                         (attribute-instance-value-cache att) ; ...each such attribute’s fix−point value to cache...
```

```
                            (attribute-instance-args-cache att) ; ...is the value computed during its last invocation. Further,...
                            (car (hashtable-ref (attribute-instance-cycle-cache att) (attribute-instance-args-cache att) #f))))
                         (hashtable-clear! (attribute-instance-cycle-cache att)) ; ...ALWAYS clear the attribute's cycle and...
                         (attribute-instance-args-cache-set! att racr-nil) ; ...most recent arguments cache....
                         (for-each ; ...Then proceed with the attributes the circular attribute depends on....
                          loop
                          (attribute-instance-attribute-dependencies att)))))
                  ; ... Finally, pop the attribute from the attribute evaluation stack.
                  (evaluator-state-att-eval-stack-set! evaluator-state (cdr (evaluator-state-att-eval-stack evaluator-state)))))))

             ; EVALUATION−CASE (2): Circular attribute, already in evaluation for the given arguments:
             ((and circular? entered?)
              ; Maintaine attribute dependencies, i.e., the other attribute throughout whose evaluation
              ; this attribute is evaluated must depend on this one. Finally,...
              (add-dependency:att->att att)
              ; ... the result is the attribute's cycle cache entry.
              (set! result (car cc-hit)))

             ; EVALUATION−CASE (3): Circular attribute not in evaluation and entered throughout a fix−point evaluation:
             (circular?
              (dynamic-wind
               (lambda ()
                ; Maintaine attribute dependencies, i.e., iff this attribute is evaluated throughout the evaluation
                ; of another attribute, the other attribute depends on this one and this attribute must depend on
                ; any other attributes that will be evaluated through its own evaluation. Further,..
                (add-dependency:att->att att)
                (evaluator-state-att-eval-stack-set! evaluator-state (cons att (evaluator-state-att-eval-stack evaluator-state)))
                ; ... mark, that the attribute is in evaluation and construct an appropriate cycle−cache entry if required.
                (if cc-hit
                    (set-cdr! cc-hit #t)
                    (begin
                      (set! cc-hit (cons (car (attribute-definition-circularity-definition att-def)) #t))
                      (hashtable-set! (attribute-instance-cycle-cache att) args cc-hit))))
               (lambda ()
                (set! result (apply (attribute-definition-equation att-def) n args)) ; Evaluate the attribute and...
                (update-cycle-cache)) ; ...update its cycle−cache.
               (lambda ()
                ; Mark that the evaluation of the attribute is finished and...
                (set-cdr! cc-hit #f)
                ; ... pop the attribute from the attribute evaluation stack.
                (evaluator-state-att-eval-stack-set! evaluator-state (cdr (evaluator-state-att-eval-stack evaluator-state))))))

             ; EVALUATION−CASE (4): Non−circular attribute already in evaluation:
             (entered?
              ; Maintaine attribute dependencies, i.e., the other attribute throughout whose evaluation
              ; this attribute is evaluated must depend on this one. Then,...
              (add-dependency:att->att att)
              (throw-exception ; ...thrown an exception because we encountered an unexpected dependency cycle.
               "AG evaluator exception; "
               "Unexpected " name " cycle."))

             (else ; EVALUATION−CASE (5): Non−circular attribute not in evaluation.
              (dynamic-wind
               (lambda ()
                ; Maintaine attribute dependencies, i.e., iff this attribute is evaluated throughout the evaluation
                ; of another attribute, the other attribute depends on this one and this attribute must depend on
                ; any other attributes that will be evaluated through its own evaluation. Further,..
                (add-dependency:att->att att)
                (evaluator-state-att-eval-stack-set! evaluator-state (cons att (evaluator-state-att-eval-stack evaluator-state)))
                ; ... mark, that the attribute is in evaluation, i.e.,...
                (set! cc-hit (cons racr-nil #t)) ; ...construct an appropriate cycle−cache entry and...
                (hashtable-set! (attribute-instance-cycle-cache att) args cc-hit)) ; ...add it to the attribute's cycle−cache.
               (lambda ()
                (set! result (apply (attribute-definition-equation att-def) n args)) ; Evaluate the attribute and...
                (when (attribute-definition-cached? att-def) ; ...if caching is enabled...
                  (hashtable-set! (attribute-instance-value-cache att) args result))) ; ...cache its value.
               (lambda ()
                ; Mark that the attribute's evaluation finished, i.e., clear its cycle−cache. Finally,...
                (hashtable-clear! (attribute-instance-cycle-cache att))
                ; ... pop the attribute from the attribute evaluation stack.
                (evaluator-state-att-eval-stack-set! evaluator-state (cdr (evaluator-state-att-eval-stack evaluator-state)))))))
           result))))) ; Return the computed value.

;              ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;       Abstract Syntax Tree Access Interface   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;              ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define ast-node-type
  (lambda (n)
    (when (or (node-list-node? n) (node-bud-node? n)) ; Remember: (node−terminal? n) is not possible
      (throw-exception
       "Cannot access type; "
       "List and bud nodes have no type."))
    (add-dependency:att->node-type n)
    (symbol-name (car (ast-rule-production (node-ast-rule n))))))
```

```
(define ast-list-node?
  (lambda (n)
    (if (node-bud-node? n)
      (throw-exception
       "Cannot perform list node check; "
       "Bud nodes have no type.")
      (node-list-node? n))))

(define ast-subtype?
  (lambda (a1 a2)
    (when (or
          (and (node? a1) (or (node-list-node? a1) (node-bud-node? a1)))
          (and (node? a2) (or (node-list-node? a2) (node-bud-node? a2))))
      (throw-exception
       "Cannot perform subtype check; "
       "List and bud nodes cannot be tested for subtyping."))
    (when (and (not (node? a1)) (not (node? a2)))
      (throw-exception
       "Cannot perform subtype check; "
       "At least one argument must be an AST node."))
    ((lambda (t1/t2)
       (and
        (car t1/t2)
        (cdr t1/t2)
        (ast-rule-subtype? (car t1/t2) (cdr t1/t2))))
     (if (symbol? a1)
        (let* ((t2 (node-ast-rule a2))
               (t1 (racr-specification-find-rule (ast-rule-specification t2) a1)))
          (unless t1
            (throw-exception
             "Cannot perform subtype check; "
             a1 " is no valid non-terminal (first argument undefined non-terminal)."))
          (add-dependency:att->node-super-type a2 t1)
          (cons t1 t2))
        (if (symbol? a2)
            (let* ((t1 (node-ast-rule a1))
                   (t2 (racr-specification-find-rule (ast-rule-specification t1) a2)))
              (unless t1
                (throw-exception
                 "Cannot perform subtype check; "
                 a2 " is no valid non-terminal (second argument undefined non-terminal)."))
              (add-dependency:att->node-sub-type a1 t2)
              (cons t1 t2))
            (begin
              (add-dependency:att->node-sub-type a1 (node-ast-rule a2))
              (add-dependency:att->node-super-type a2 (node-ast-rule a1))
              (cons (node-ast-rule a1) (node-ast-rule a2))))))))

(define ast-parent
  (lambda (n)
    (let ((parent (node-parent n)))
      (unless parent
        (throw-exception "Cannot access parent of roots."))
      (add-dependency:att->node parent)
      parent)))

(define ast-child
  (lambda (i n)
    (let ((child
          (if (symbol? i)
              (node-find-child n i)
              (and (>= i 1) (<= i (length (node-children n))) (list-ref (node-children n) (- i 1))))))
      (unless child
        (throw-exception "Cannot access non-existent " i (if (symbol? i) "'th" "") " child."))
      (add-dependency:att->node child)
      (if (node-terminal? child)
          (node-children child)
          child))))

(define ast-sibling
  (lambda (i n)
    (ast-child i (ast-parent n))))

(define ast-child-index
  (lambda (n)
    (add-dependency:att->node n)
    (node-child-index n)))

(define ast-num-children
  (lambda (n)
    (when (node-bud-node? n)
      (throw-exception
       "Cannot access number of children; "
       "Bud nodes have no children."))
    (add-dependency:att->node-num-children n)
    (length (node-children n))))
```

```
(define-syntax ast-children
  (syntax-rules ()
    ((_ n b ...)
     (reverse
      (let ((result (list)))
       (ast-for-each-child
        (lambda (i child)
          (set! result (cons child result)))
        n
        b ...)
       result)))))

(define-syntax ast-for-each-child
  (syntax-rules ()
    ((_ f n b)
     (let* ((f* f)
            (n* n)
            (b* b)
            (ub (cdr b*)))
       (when (node-bud-node? n*)
         (throw-exception
          "Cannot visit children; "
          "No valid operation on bud nodes."))
       (if (eq? ub '*)
           (let ((pos (car b*))
                 (ub (length (node-children n*))))
             (dynamic-wind
              (lambda () #f)
              (lambda ()
                (let loop ()
                  (when (<= pos ub)
                    (f* pos (ast-child pos n*))
                    (set! pos (+ pos 1))
                    (loop))))
              (lambda ()
                (when (> pos ub)
                  (ast-num-children n*))))) ; BEWARE: Access to number of children ensures proper dependency tracking!
           (let loop ((pos (car b*)))
             (when (<= pos ub)
              (f* pos (ast-child pos n*))
              (loop (+ pos 1)))))))
    ((_ f n)
     (ast-for-each-child f n (cons 1 '*)))
    ((_ f n b ...)
     (let ((f* f)
           (n* n))
       (ast-for-each-child f* n* b) ...))))

(define-syntax ast-find-child
  (syntax-rules ()
    ((_ f n b ...)
     (let ((f* f))
       (call/cc
        (lambda (c)
          (ast-for-each-child
           (lambda (i child)
             (when (f* i child)
               (c child)))
           n
           b ...)
          #f))))))

;             ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;      Abstract Syntax Tree Construction Interface ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;             ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define create-ast
  (lambda (spec rule children)
    ;;; Ensure, that the RACR system is completely specified:
    (when (< (racr-specification-specification-phase spec) 3)
      (throw-exception
       "Cannot construct " rule " fragment; "
       "The RACR specification still must be compiled."))

    (let ((ast-rule* (racr-specification-find-rule spec rule)))
      ;;; Ensure, that the given AST rule is defined:
      (unless ast-rule*
        (throw-exception
         "Cannot construct " rule " fragment; "
         "Unknown non-terminal/rule."))

      ;;; Ensure, that the expected number of children are given:
      (unless (= (length children) (- (length (ast-rule-production ast-rule*)) 1))
        (throw-exception
         "Cannot construct " rule " fragment; "
```

```
         (length children) " children given, but " (- (length (ast-rule-production ast-rule*)) 1) " children expected."))

      ;;; Construct the fragment, i.e., (1) the AST part consisting of the root and the given children and (2) the root's
      ;;; synthesized attribute instances and the childrens' inherited ones.
      (let (;;; For (1) - the construction of the fragment's AST part - first construct the fragment's root. Then...
            (root
             (make-node
              ast-rule*
              #f
              (list))))
        (node-children-set! ; ...ensure, that the given children fit and add them to the fragment to construct. Therefore,...
         root
         (let loop ((pos 1) ; ...investigate every...
                    (symbols (cdr (ast-rule-production ast-rule*))) ; ...expected and...
                    (children children)) ; ...given child....
           (if (null? symbols) ; ...If no further child is expected,...
               (list) ; ...we are done, otherwise...
               (let ((symb* (car symbols))
                     (child (car children)))
                 (if (symbol-non-terminal? symb*) ; ...check if the next expected child is a non-terminal....
                     (let ((ensure-child-fits ; ...If we expect a non-terminal we need a function which ensures, that...
                            (lambda (child)
                              ; ...the child either is a bud-node or its type is the one of the
                              ; expected non-terminal or a sub-type....
                              (unless (or
                                       (node-bud-node? child)
                                       (ast-rule-subtype? (node-ast-rule child) (symbol-non-terminal? symb*)))
                                (throw-exception
                                 "Cannot construct " rule " fragment; "
                                 "Expected a " (symbol-name symb*) " node as " pos "'th child, not a " (ast-node-type child) ".")))))
                       (unless (node? child) ; ...Then, check that the given child is an AST node,...
                         (throw-exception
                          "Cannot construct " rule " fragment; "
                          "Expected a " (symbol-name symb*) " node as " pos "'th child, not a terminal."))
                       (when (node-parent child) ; ...does not already belong to another AST and...
                         (throw-exception
                          "Cannot construct " rule " fragment; "
                          "The given " pos "'th child already is part of another AST fragment."))
                       ; ...non of its attributes are in evaluation....
                       (when (evaluator-state-in-evaluation? (node-evaluator-state child))
                         (throw-exception
                          "Cannot construct " rule " fragment; "
                          "There are attributes in evaluation."))
                       (if (symbol-kleene? symb*) ; ...Now, check if we expect a list of non-terminals...
                           (if (node-list-node? child) ; ...If we expect a list, ensure the given child is a list-node and...
                               (for-each ensure-child-fits (node-children child)) ; ...all its elements fit....
                               (throw-exception
                                "Cannot construct " rule " fragment; "
                                "Expected a list-node as " pos "'th child, not a "
                                (if (node? child)
                                    (string-append "single [" (symbol->string (ast-node-type child)) "] node")
                                    "terminal")
                                "."))
                           (ensure-child-fits child)) ; ...If we expect a single non-terminal child, just ensure that the child fits....
                       (node-parent-set! child root) ; ...Finally, set the root as the child's parent,...
                       (cons
                        child ; ...add the child to the root's children and...
                        (loop (+ pos 1) (cdr symbols) (cdr children)))) ; ...process the next expected child.
                     (cons ; If we expect a terminal,...
                      (make-node ; ...add a terminal node encapsulating the given value to the root's children and...
                       'terminal
                       root
                       child)
                      (loop (+ pos 1) (cdr symbols) (cdr children)))))))) ; ...process the next expected child.
        ; ...When all children are processed, distribute the new fragment's evaluator state:
        (distribute-evaluator-state (make-evaluator-state) root)

        ;;; The AST part of the fragment is properly constructed so we can proceed with (2) - the construction
        ;;; of the fragment's attribute instances. Therefore,...
        (update-synthesized-attribution root) ; ...initialize the root's synthesized and...
        (for-each ; ...each child's inherited attributes.
         update-inherited-attribution
         (node-children root))

        root)))) ; Finally, return the newly constructed fragment.

(define create-ast-list
  (lambda (children)
    (let* ((child-with-spec
            (find
             (lambda (child)
               (and (node? child) (not (node-list-node? child)) (not (node-bud-node? child))))
             children))
           (spec (and child-with-spec (ast-rule-specification (node-ast-rule child-with-spec)))))
      (let loop ((children children) ; For every child, ensure, that the child is a...
                 (pos 1))
```

```
        (unless (null? children)
          (when (or (not (node? (car children))) (node-list-node? (car children))) ; ...proper non-terminal node,...
            (throw-exception
             "Cannot construct list-node; "
             "The given " pos "'th child is not a non-terminal, non-list node."))
          (when (node-parent (car children)) ; ...is not already part of another AST,...
            (throw-exception
             "Cannot construct list-node; "
             "The given " pos "'th child already is part of another AST."))
          ; ... non of its attributes are in evaluation and...
          (when (evaluator-state-in-evaluation? (node-evaluator-state (car children)))
            (throw-exception
             "Cannot construct list-node; "
             "The given " pos "'th child has attributes in evaluation."))
          (unless (or ; ...all children are instances of the same RACR specification.
                   (node-bud-node? (car children))
                   (eq? (ast-rule-specification (node-ast-rule (car children)))
                        spec))
            (throw-exception
             "Cannot construct list-node; "
             "The given children are instances of different RACR specifications."))
          (loop (cdr children) (+ pos 1)))))
    (let ((list-node ; ...Finally, construct the list-node,...
           (make-node
            'list-node
            #f
            children)))
      (for-each ; ...set it as parent for every of its elements,...
       (lambda (child)
         (node-parent-set! child list-node))
       children)
      (distribute-evaluator-state (make-evaluator-state) list-node) ; ...construct and distribute its evaluator state and...
      list-node))) ; ...return it.

(define create-ast-bud
  (lambda ()
    (let ((bud-node (make-node 'bud-node #f (list))))
      (distribute-evaluator-state (make-evaluator-state) bud-node)
      bud-node)))

; INTERNAL FUNCTION: Given an AST node update its synthesized attribution (i.e., add missing synthesized
; attributes, delete superfluous ones, shadow equally named inherited attributes and update the
; definitions of existing synthesized attributes.
(define update-synthesized-attribution
  (lambda (n)
    (when (and (not (node-terminal? n)) (not (node-list-node? n)) (not (node-bud-node? n)))
      (for-each
       (lambda (att-def)
         (let ((att (node-find-attribute n (attribute-definition-name att-def))))
           (cond
             ((not att)
              (node-attributes-set! n (cons (make-attribute-instance att-def n) (node-attributes n))))
             ((eq? (attribute-definition-equation (attribute-instance-definition att)) (attribute-definition-equation att-def))
              (attribute-instance-definition-set! att att-def))
             (else
              (flush-attribute-cache att)
              (attribute-instance-context-set! att racr-nil)
              (node-attributes-set!
               n
               (cons (make-attribute-instance att-def n) (remq att (node-attributes n))))))))
       (symbol-attributes (car (ast-rule-production (node-ast-rule n)))))
      (node-attributes-set! ; Delete all synthesized attribute instances not defined anymore:
       n
       (remp
        (lambda (att)
          (let ((remove?
                 (and
                  (attribute-definition-synthesized? (attribute-instance-definition att))
                  (not (eq? (car (attribute-definition-context (attribute-instance-definition att))) (node-ast-rule n))))))
            (when remove?
              (flush-attribute-cache att)
              (attribute-instance-context-set! att racr-nil))
            remove?))
        (node-attributes n))))))

; INTERNAL FUNCTION: Given an AST node update its inherited attribution (i.e., add missing inherited
; attributes, delete superfluous ones and update the definitions of existing inherited attributes.
; If the given node is a list-node the inherited attributes of its elements are updated.
(define update-inherited-attribution
  (lambda (n)
    ;;; Support function updating n's inherited attribution w.r.t. a list of inherited attribute definitions:
    (define update-by-defs
      (lambda (n att-defs)
        (for-each ; Add new and update existing inherited attribute instances:
         (lambda (att-def)
           (let ((att (node-find-attribute n (attribute-definition-name att-def))))
```

```
             (cond
               ((not att)
                (node-attributes-set! n (cons (make-attribute-instance att-def n) (node-attributes n))))
               ((not (attribute-definition-synthesized? (attribute-instance-definition att)))
                (if (eq?
                     (attribute-definition-equation (attribute-instance-definition att))
                     (attribute-definition-equation att-def))
                    (attribute-instance-definition-set! att att-def)
                    (begin
                      (flush-attribute-cache att)
                      (attribute-instance-context-set! att racr-nil)
                      (node-attributes-set! n (cons (make-attribute-instance att-def n) (remq att (node-attributes n))))))))))
         att-defs)
        (node-attributes-set! ; Delete all inherited attribute instances not defined anymore:
         n
         (remp
          (lambda (att)
            (let ((remove?
                   (and
                    (attribute-definition-inherited? (attribute-instance-definition att))
                    (not (memq (attribute-instance-definition att) att-defs)))))
              (when remove?
                (flush-attribute-cache att)
                (attribute-instance-context-set! att racr-nil))
              remove?))
          (node-attributes n)))))
    ;;; Perform the update:
    (let* ((n* (if (node-list-node? (node-parent n)) (node-parent n) n))
           (att-defs (symbol-attributes (list-ref (ast-rule-production (node-ast-rule (node-parent n*))) (node-child-index n*)))))
      (if (node-list-node? n)
          (for-each
           (lambda (n)
             (unless (node-bud-node? n)
               (update-by-defs n att-defs)))
           (node-children n))
          (unless (node-bud-node? n)
            (update-by-defs n att-defs))))))

; INTERNAL FUNCTION: Given an AST node delete its inherited attribute instances. Iff the given node
; is a list node, the inherited attributes of its elements are deleted.
(define detach-inherited-attributes
  (lambda (n)
    (cond
      ((node-list-node? n)
       (for-each
        detach-inherited-attributes
        (node-children n)))
      ((node-non-terminal? n)
       (node-attributes-set!
        n
        (remp
         (lambda (att)
           (let ((remove? (attribute-definition-inherited? (attribute-instance-definition att))))
             (when remove?
               (flush-attribute-cache att)
               (attribute-instance-context-set! att racr-nil))
             remove?))
         (node-attributes n)))))))

; INTERNAL FUNCTION: Given an evaluator state and an AST fragment, change the
; fragment's evaluator state to the given one.
(define distribute-evaluator-state
  (lambda (evaluator-state n)
    (node-evaluator-state-set! n evaluator-state)
    (unless (node-terminal? n)
      (for-each
       (lambda (n)
         (distribute-evaluator-state evaluator-state n))
       (node-children n)))))

;               ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;       ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;        Rewrite Interface       ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;               ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define perform-rewrites
  (lambda (n strategy . transformers)
    (define find-and-apply
      (case strategy
        ((top-down)
         (lambda (n)
           (and
            (not (node-terminal? n))
            (or
             (find (lambda (r) (r n)) transformers)
             (find find-and-apply (node-children n))))))
        ((bottom-up)
```

```
         (lambda (n)
           (and
            (not (node-terminal? n))
            (or
             (find find-and-apply (node-children n))
             (find (lambda (r) (r n)) transformers)))))
        (else (throw-exception
               "Cannot perform rewrites; "
               "Unknown " strategy " strategy."))))
    (let loop ()
      (when (node-parent n)
        (throw-exception
         "Cannot perform rewrites; "
         "The given starting point is not (anymore) an AST root."))
      (let ((match (find-and-apply n)))
        (if match
            (cons match (loop))
            (list))))))

; INTERNAL FUNCTION: Given an AST node n, flush all attributes that depend on information of
; the subtree spaned by n but are outside of it.
(define flush-depending-attributes-outside-of
  (lambda (n)
    (let loop ((n* n))
      (for-each
       (lambda (influence)
         (unless (node-inside-of? (attribute-instance-context (car influence)) n)
           (flush-attribute-cache (car influence))))
       (node-attribute-influences n*))
      (for-each
       (lambda (att)
         (for-each
          (lambda (influenced)
            (unless (node-inside-of? (attribute-instance-context influenced) n)
              (flush-attribute-cache influenced)))
          (attribute-instance-attribute-influences att)))
       (node-attributes n*))
      (unless (node-terminal? n*)
        (for-each
         loop
         (node-children n*))))))

(define rewrite-terminal
  (lambda (i n new-value)
    ; Before changing the value of the terminal ensure, that...
    (when (evaluator-state-in-evaluation? (node-evaluator-state n)) ; ...no attributes are in evaluation and...
      (throw-exception
       "Cannot change terminal value; "
       "There are attributes in evaluation."))
    (let ((n
           (if (symbol? i)
               (node-find-child n i)
               (and (>= i 1) (<= i (length (node-children n))) (list-ref (node-children n) (- i 1))))))
      (unless (and n (node-terminal? n)) ; ...the given context is a terminal. If so,...
        (throw-exception
         "Cannot change terminal value; "
         "The given context does not exist or is no terminal."))
      (unless (equal? (node-children n) new-value)
        (for-each ; ...flush the caches of all attributes influenced by the terminal and...
         (lambda (influence)
           (flush-attribute-cache (car influence)))
         (node-attribute-influences n))
        (node-children-set! n new-value))))) ; ...rewrite its value.

(define rewrite-refine
  (lambda (n t . c)
    ;;; Before refining the non−terminal ensure, that...
    (when (evaluator-state-in-evaluation? (node-evaluator-state n)) ; ...non of its attributes are in evaluation,...
      (throw-exception
       "Cannot refine node; "
       "There are attributes in evaluation."))
    (when (or (node-list-node? n) (node-bud-node? n)) ; ...it is not a list or bud node,...
      (throw-exception
       "Cannot refine node; "
       "The node is a " (if (node-list-node? n) "list" "bud") " node."))
    (let* ((old-rule (node-ast-rule n))
           (new-rule (racr-specification-find-rule (ast-rule-specification old-rule) t)))
      (unless (and new-rule (ast-rule-subtype? new-rule old-rule)) ; ...the given type is a subtype,...
        (throw-exception
         "Cannot refine node; "
         t " is not a subtype of " (ast-node-type n)))
      (let ((additional-children (list-tail (ast-rule-production new-rule) (length (ast-rule-production old-rule)))))
        (unless (= (length additional-children) (length c)) ; ...the expected number of new children are given,...
          (throw-exception
           "Cannot refine node; "
           "Unexpected number of additional children."))
```

```
        (let ((c
               (map ; ...each child...
                (lambda (symbol child)
                  (cond
                    ((symbol-non-terminal? symbol)
                     (unless (node? child) ; ...fits,...
                       (throw-exception
                        "Cannot refine node; "
                        "The given children do not fit."))
                     (when (node-parent child) ; ...is not part of another AST,...
                       (throw-exception
                        "Cannot refine node; "
                        "A given child already is part of another AST."))
                     (when (node-inside-of? n c) ; ...does not contain the refined node and...
                       (throw-exception
                        "Cannot refine node; "
                        "The node to refine is part of the AST spaned by a given child."))
                     (when (evaluator-state-in-evaluation? (node-evaluator-state child)) ; ...non of its attributes are in evaluation.
                       (throw-exception
                        "Cannot refine node; "
                        "There are attributes in evaluation."))
                     (if (symbol-kleene? symbol)
                         (if (node-list-node? child)
                             (for-each
                              (lambda (child)
                                (unless
                                    (or
                                     (node-bud-node? child)
                                     (ast-rule-subtype? (node-ast-rule child) (symbol-non-terminal? symbol)))
                                  (throw-exception
                                   "Cannot refine node; "
                                   "The given children do not fit.")))
                              (node-children child))
                             (throw-exception
                              "Cannot refine node; "
                              "The given children do not fit."))
                         (unless
                             (and
                              (node-non-terminal? child)
                              (not (node-list-node? child))
                              (or (node-bud-node? child) (ast-rule-subtype? (node-ast-rule child) (symbol-non-terminal? symbol))))
                           (throw-exception
                            "Cannot refine node; "
                            "The given children do not fit.")))
                     child)
                    (else
                     (when (node? child)
                       (throw-exception
                        "Cannot refine node; "
                        "The given children do not fit."))
                     (make-node 'terminal n child))))
                additional-children
                c)))
          ;; Everything is fine. Thus,...
          (for-each ; ...flush the influenced attributes, i.e., all attributes influenced by the node’s...
           (lambda (influence)
             (when (or
                    (and (vector-ref (cdr influence) 1) (not (null? c))) ; ...number of children,...
                    (and (vector-ref (cdr influence) 2) (not (eq? old-rule new-rule))) ; ...type,...
                    (find ; ...supertype or...
                     (lambda (t2)
                       (not (eq? (ast-rule-subtype? t2 old-rule) (ast-rule-subtype? t2 new-rule))))
                     (vector-ref (cdr influence) 3))
                    (find ; ...subtype. Afterwards,...
                     (lambda (t2)
                       (not (eq? (ast-rule-subtype? old-rule t2) (ast-rule-subtype? new-rule t2))))
                     (vector-ref (cdr influence) 4)))
               (flush-attribute-cache (car influence))))
           (node-attribute-influences n))
          (node-ast-rule-set! n new-rule) ; ...update the node’s type,...
          (update-synthesized-attribution n) ; ...synthesized attribution,...
          (node-children-set! n (append (node-children n) c (list))) ; ...insert the new children,...
          (for-each
           (lambda (child)
             (node-parent-set! child n)
             (distribute-evaluator-state (node-evaluator-state n) child)) ; ...update their evaluator state and...
           c)
          (for-each ; ...update the inherited attribution of all children.
           update-inherited-attribution
           (node-children n)))))))

(define rewrite-abstract
  (lambda (n t)
    ;; Before abstracting the non-terminal ensure, that...
    (when (evaluator-state-in-evaluation? (node-evaluator-state n)) ; ...no attributes are in evaluation,...
      (throw-exception
```

```
       "Cannot abstract node; "
       "There are attributes in evaluation."))
    (when (or (node-list-node? n) (node-bud-node? n)) ; ...the given node is not a list or bud node and...
      (throw-exception
       "Cannot abstract node; "
       "The node is a " (if (node-list-node? n) "list" "bud") " node."))
    (let* ((old-rule (node-ast-rule n))
           (new-rule (racr-specification-find-rule (ast-rule-specification old-rule) t))
           (num-new-children (- (length (ast-rule-production new-rule)) 1)))
      (unless (and new-rule (ast-rule-subtype? old-rule new-rule)) ; ...the given type is a supertype.
        (throw-exception
         "Cannot abstract node; "
         t " is not a supertype of " (ast-node-type n) "."))
      ;;; Everything is fine. Thus,...
      (let ((children-to-remove (list-tail (node-children n) num-new-children)))
        (for-each ; ...flush the caches of all influenced attributes, i.e., (1) all attributes influenced by the node's...
         (lambda (influence)
           (when (or
                  (and (vector-ref (cdr influence) 1) (not (null? children-to-remove))) ; ...number of children,...
                  (and (vector-ref (cdr influence) 2) (not (eq? old-rule new-rule))) ; ...type...
                  (find ; ...supertype or...
                   (lambda (t2)
                     (not (eq? (ast-rule-subtype? t2 old-rule) (ast-rule-subtype? t2 new-rule))))
                   (vector-ref (cdr influence) 3))
                  (find ; ...subtype and...
                   (lambda (t2)
                     (not (eq? (ast-rule-subtype? old-rule t2) (ast-rule-subtype? new-rule t2))))
                   (vector-ref (cdr influence) 4)))
             (flush-attribute-cache (car influence))))
         (node-attribute-influences n))
        (for-each ; ...(2) all attributes depending on, but still outside of, an removed AST. Afterwards,...
         flush-depending-attributes-outside-of
         children-to-remove)
        (node-ast-rule-set! n new-rule) ; ...update the node's type and...
        (update-synthesized-attribution n) ; ...synthesized attribution and...
        (for-each ; ...for every child to remove,...
         (lambda (child)
           (detach-inherited-attributes child) ; ...delete its inherited attribution,...
           (node-parent-set! child #f) ; ...detach it from the AST and...
           (distribute-evaluator-state (make-evaluator-state) child)) ; ...update its evaluator state. Further,...
         children-to-remove)
        (unless (null? children-to-remove)
          (if (> num-new-children 0)
              (set-cdr! (list-tail (node-children n) (- num-new-children 1)) (list))
              (node-children-set! n (list))))
        (for-each ; ...update the inherited attribution of all remaining children. Finally,...
         update-inherited-attribution
         (node-children n))
        children-to-remove)))) ; ...return the removed children.

(define rewrite-add
  (lambda (l e)
    ;;; Before adding the element ensure, that...
    (when (or ; ...no attributes are in evaluation,...
           (evaluator-state-in-evaluation? (node-evaluator-state l))
           (evaluator-state-in-evaluation? (node-evaluator-state e)))
      (throw-exception
       "Cannot add list element; "
       "There are attributes in evaluation."))
    (unless (node-list-node? l) ; ...indeed a list-node is given as context,...
      (throw-exception
       "Cannot add list element; "
       "The given context is no list-node."))
    (when (node-parent e) ; ...the new element is not part of another AST,...
      (throw-exception
       "Cannot add list element; "
       "The element to add already is part of another AST."))
    (when (node-inside-of? l e) ; ...its spaned AST does not contain the list-node and...
      (throw-exception
       "Cannot add list element; "
       "The given list is part of the AST spaned by the element to add."))
    (when (node-parent l)
      (let ((expected-type
             (symbol-non-terminal?
              (list-ref
               (ast-rule-production (node-ast-rule (node-parent l)))
               (node-child-index l)))))
        (unless (or (node-bud-node? e) (ast-rule-subtype? (node-ast-rule e) expected-type)) ; ...it can be a child of the list-node.
          (throw-exception
           "Cannot add list element; "
           "The new element does not fit."))))
    ;;; When all rewrite constraints are satisfied,...
    (for-each ; ...flush the caches of all attributes influenced by the list-node's number of children,...
     (lambda (influence)
       (when (vector-ref (cdr influence) 1)
         (flush-attribute-cache (car influence))))
```

```
     (node-attribute-influences l))
    (node-children-set! l (append (node-children l) (list e))) ; ...add the new element,...
    (node-parent-set! e l)
    (distribute-evaluator-state (node-evaluator-state l) e) ; ...initialize its evaluator state and...
    (when (node-parent l)
      (update-inherited-attribution e)))) ; ...any inherited attributes defined for its new context.

(define rewrite-subtree
  (lambda (old-fragment new-fragment)
    ;;; Before replacing the subtree ensure, that...
    (when (or ; ... no attributes are in evaluation,...
           (evaluator-state-in-evaluation? (node-evaluator-state old-fragment))
           (evaluator-state-in-evaluation? (node-evaluator-state new-fragment)))
      (throw-exception
       "Cannot replace subtree; "
       "There are attributes in evaluation."))
    (unless (and (node? new-fragment) (node-non-terminal? new-fragment)) ; ...the new fragment is a non-terminal node,...
      (throw-exception
       "Cannot replace subtree; "
       "The replacement is not a non-terminal node."))
    (when (node-parent new-fragment) ; ...it is not part of another AST...
      (throw-exception
       "Cannot replace subtree; "
       "The replacement already is part of another AST."))
    (when (node-inside-of? old-fragment new-fragment) ; ...its spaned AST did not contain the old-fragment and...
      (throw-exception
       "Cannot replace subtree; "
       "The given old fragment is part of the AST spaned by the replacement."))
    (let* ((n* (if (node-list-node? (node-parent old-fragment)) (node-parent old-fragment) old-fragment))
           (expected-type
            (symbol-non-terminal?
             (list-ref
              (ast-rule-production (node-ast-rule (node-parent n*)))
              (node-child-index n*)))))
      (if (node-list-node? old-fragment) ; ...it fits into its new context.
          (if (node-list-node? new-fragment)
              (for-each
               (lambda (element)
                 (unless (or (node-bud-node? element) (ast-rule-subtype? element expected-type))
                   (throw-exception
                    "Cannot replace subtree; "
                    "The replacement does not fit.")))
               (node-children new-fragment))
              (throw-exception
               "Cannot replace subtree; "
               "The replacement does not fit."))
          (unless (and
                   (not (node-list-node? new-fragment))
                   (or (node-bud-node? new-fragment) (ast-rule-subtype? (node-ast-rule new-fragment) expected-type)))
            (throw-exception
             "Cannot replace subtree; "
             "The replacement does not fit."))))
    ;;; When all rewrite constraints are satisfied,...
    (detach-inherited-attributes old-fragment) ; ...delete the old fragment's inherited attribution,...
    (flush-depending-attributes-outside-of old-fragment) ; ...flush all attributes depending on it and outside its spaned tree,...
    (distribute-evaluator-state (node-evaluator-state old-fragment) new-fragment) ; ...update both fragments' evaluator state,...
    (distribute-evaluator-state (make-evaluator-state) old-fragment)
    (set-car! ; ...replace the old fragment by the new one and...
     (list-tail (node-children (node-parent old-fragment)) (- (node-child-index old-fragment) 1))
     new-fragment)
    (node-parent-set! new-fragment (node-parent old-fragment))
    (node-parent-set! old-fragment #f)
    (update-inherited-attribution new-fragment) ; ...update the new fragment's inherited attribution. Finally,...
    old-fragment )) ; ...return the removed old fragment.

(define rewrite-insert
  (lambda (l i e)
    ;;; Before inserting the element ensure, that...
    (when (or ; ...no attributes are in evaluation,...
           (evaluator-state-in-evaluation? (node-evaluator-state l))
           (evaluator-state-in-evaluation? (node-evaluator-state e)))
      (throw-exception
       "Cannot insert list element; "
       "There are attributes in evaluation."))
    (unless (node-list-node? l) ; ...indeed a list-node is given as context,...
      (throw-exception
       "Cannot insert list element; "
       "The given context is no list-node."))
    (when (or (< i 1) (> i (+ (length (node-children l)) 1))) ; ...the list has enough elements,...
      (throw-exception
       "Cannot insert list element; "
       "The given index is out of range."))
    (when (node-parent e) ; ...the new element is not part of another AST,...
      (throw-exception
       "Cannot insert list element; "
       "The element to insert already is part of another AST."))
```

```
    (when (node-inside-of? l e) ; ...its spaned AST does not contain the list−node and...
     (throw-exception
      "Cannot insert list element; "
      "The given list is part of the AST spaned by the element to insert."))
    (when (node-parent l)
      (let ((expected-type
             (symbol-non-terminal?
              (list-ref
               (ast-rule-production (node-ast-rule (node-parent l)))
               (node-child-index l)))))
        (unless (or (node-bud-node? e) (ast-rule-subtype? (node-ast-rule e) expected-type)) ; ...it can be a child of the list−node.
          (throw-exception
           "Cannot insert list element; "
           "The new element does not fit."))))
    ;;; When all rewrite constraints are satisfied...
    (for-each ; ...flush the caches of all attributes influenced by the list−node's number of children. Further,...
     (lambda (influence)
       (when (vector-ref (cdr influence) 1)
         (flush-attribute-cache (car influence))))
     (node-attribute-influences l))
    (for-each ; ...for each tree spaned by the successor element's of the insertion position,...
     ; ... flush the caches of all attributes depending on, but still outside of, the respective tree. Then,...
     flush-depending-attributes-outside-of
     (list-tail (node-children l) (- i 1)))
    (node-children-set! ; ...insert the new element,...
     l
     (let loop ((l (node-children l)) (i i))
       (cond
         ((= i 1) (cons e (loop l 0)))
         ((null? l) (list))
         (else (cons (car l) (loop (cdr l) (- i 1)))))))
    (node-parent-set! e l)
    (distribute-evaluator-state (node-evaluator-state l) e) ; ...initialize its evaluator state and...
    (when (node-parent l)
      (update-inherited-attribution e)))) ; ...any inherited attributes defined for its new context.

(define rewrite-delete
  (lambda (n)
    ;;; Before deleting the element ensure, that...
    (when (evaluator-state-in-evaluation? (node-evaluator-state n)) ; ...no attributes are in evaluation and...
      (throw-exception
       "Cannot delete list element; "
       "There are attributes in evaluation."))
    (unless (and (node-parent n) (node-list-node? (node-parent n))) ; ...the given node is a list−node element.
      (throw-exception
       "Cannot delete list element; "
       "The given node is not element of a list."))
    ;;; When all rewrite constraints are satisfied, flush the caches of all attributes influenced by
    ; the number of children of the list−node the element is part of. Further,...
    (for-each
     (lambda (influence)
       (when (vector-ref (cdr influence) 1)
         (flush-attribute-cache (car influence))))
     (node-attribute-influences (node-parent n)))
    (detach-inherited-attributes n) ; ...delete the element's inherited attributes and,...
    (for-each ; ...for each tree spaned by the element and its successor elements,...
     ; ... flush the caches of all attributes depending on, but still outside of, the respective tree. Then,...
     flush-depending-attributes-outside-of
     (list-tail (node-children (node-parent n)) (- (node-child-index n) 1)))
    (node-children-set! (node-parent n) (remq n (node-children (node-parent n)))) ; ...remove the element from the list,...
    (node-parent-set! n #f)
    (distribute-evaluator-state (make-evaluator-state) n) ; ...reset its evaluator state and...
    n)) ; ... return it.

;              ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;       Dependency Tracking Support   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;              ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; INTERNAL FUNCTION: Given an attribute, flush its and its depending attributes' caches and dependencies.
(define flush-attribute-cache
  (lambda (att)
    (let ((influenced-atts (attribute-instance-attribute-influences att))) ; Save all attributes influenced by the attribute,...
      (attribute-instance-attribute-influences-set! att (list)) ; ...remove the respective influence edges and...
      (hashtable-clear! (attribute-instance-value-cache att)) ; ...clear the attribute's value cache. Then,...
      (for-each ; ...for every attribute I the attribute depends on,...
       (lambda (influencing-att)
         (attribute-instance-attribute-influences-set! ; ...remove the influence edge from I to the attribute and...
          influencing-att
          (remq att (attribute-instance-attribute-influences influencing-att))))
       (attribute-instance-attribute-dependencies att))
      (attribute-instance-attribute-dependencies-set! att (list)) ;...the attribute's dependency edges to such I. Then,...
      (for-each ; ...for every node N the attribute depends on...
       (lambda (node-influence)
         (node-attribute-influences-set!
          (car node-influence)
          (remp ; ...remove the influence edge from N to the attribute and...
```

```
           (lambda (attribute-influence)
             (eq? (car attribute-influence) att))
           (node-attribute-influences (car node-influence)))))
       (attribute-instance-node-dependencies att))
      (attribute-instance-node-dependencies-set! att (list)) ; ...the attribute's dependency edges to such N. Finally,...
      (for-each ; ...for every attribute D the attribute originally influenced,...
       (lambda (dependent-att)
         (flush-attribute-cache dependent-att)) ; ...flush D.
       influenced-atts))))

; INTERNAL FUNCTION: See "add−dependency:att−>node−characteristic".
(define add-dependency:att->node
  (lambda (influencing-node)
    (add-dependency:att->node-characteristic influencing-node (cons 0 racr-nil))))

; INTERNAL FUNCTION: See "add−dependency:att−>node−characteristic".
(define add-dependency:att->node-num-children
  (lambda (influencing-node)
    (add-dependency:att->node-characteristic influencing-node (cons 1 racr-nil))))

; INTERNAL FUNCTION: See "add−dependency:att−>node−characteristic".
(define add-dependency:att->node-type
  (lambda (influencing-node)
    (add-dependency:att->node-characteristic influencing-node (cons 2 racr-nil))))

; INTERNAL FUNCTION: See "add−dependency:att−>node−characteristic".
(define add-dependency:att->node-super-type
  (lambda (influencing-node comparision-type)
    (add-dependency:att->node-characteristic influencing-node (cons 3 comparision-type))))

; INTERNAL FUNCTION: See "add−dependency:att−>node−characteristic".
(define add-dependency:att->node-sub-type
  (lambda (influencing-node comparision-type)
    (add-dependency:att->node-characteristic influencing-node (cons 4 comparision-type))))

; INTERNAL FUNCTION: Given a node N and a correlation C add an dependency−edge marked with C from
; the attribute currently in evaluation (considering the evaluator state of the AST N is part of) to N and
; an influence−edge vice versa. If no attribute is in evaluation no edges are added. The following six
; correlations exist:
;  1) Dependency on the existence of the node (i.e., existence of a node at the same location)
;  2) Dependency on the node's number of children (i.e., existence of a node at the same location and with
;     the same number of children)
;  3) Dependency on the node's type (i.e., existence of a node at the same location and with the same type)
;  4) Dependency on whether the node's type is a supertype w.r.t. a certain type encoded in C or not
;  5) Dependency on whether the node's type is a subtype w.r.t. a certain type encoded in C or not
(define add-dependency:att->node-characteristic
  (lambda (influencing-node correlation)
    (let ((dependent-att (evaluator-state-in-evaluation? (node-evaluator-state influencing-node))))
      (when dependent-att
        (let ((dependency-vector
               (let ((dc-hit (assq influencing-node (attribute-instance-node-dependencies dependent-att))))
                 (and dc-hit (cdr dc-hit)))))
          (unless dependency-vector
            (begin
              (set! dependency-vector (vector #f #f #f (list) (list)))
              (attribute-instance-node-dependencies-set!
               dependent-att
               (cons
                (cons influencing-node dependency-vector)
                (attribute-instance-node-dependencies dependent-att)))
              (node-attribute-influences-set!
               influencing-node
               (cons
                (cons dependent-att dependency-vector)
                (node-attribute-influences influencing-node)))))
          (let ((correlation-type (car correlation))
                (correlation-arg (cdr correlation)))
            (vector-set!
             dependency-vector
             correlation-type
             (case correlation-type
               ((0 1 2)
                #t)
               ((3 4)
                (let ((known-args (vector-ref dependency-vector correlation-type)))
                  (if (memq correlation-arg known-args)
                      known-args
                      (cons correlation-arg known-args)))))))))))

; INTERNAL FUNCTION: Given an attribute instance A, add an dependency−edge from A to the attribute currently
; in evaluation (considering the evaluator state of the AST A is part of) and an influence−edge vice−versa.
; If no attribute is in evaluation no edges are added.
(define add-dependency:att->att
  (lambda (influencing-att)
    (let ((dependent-att (evaluator-state-in-evaluation? (node-evaluator-state (attribute-instance-context influencing-att)))))
      (when (and dependent-att (not (memq influencing-att (attribute-instance-attribute-dependencies dependent-att))))
```

```
         (attribute-instance-attribute-dependencies-set!
          dependent-att
          (cons
           influencing-att
           (attribute-instance-attribute-dependencies dependent-att)))
         (attribute-instance-attribute-influences-set!
          influencing-att
          (cons
           dependent-att
           (attribute-instance-attribute-influences influencing-att))))))))
```

B. MIT License

Copyright (c) 2012 by Christoff Bürger

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

API Index