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

February 17, 2014

v0.6.1

Developer Manual for RACR v0.6.1
RACR download and homepage: `https://code.google.com/p/racr/`

Abstract

This report presents *RACR*, a reference attribute grammar library for the programming language *Scheme*.

RACR supports incremental attribute evaluation in the presence of arbitrary abstract syntax tree rewrites. It provides a set of functions that can be used to specify abstract syntax tree schemes and their attribution and construct respective trees, query their attributes and node information and annotate and rewrite them. Thereby, both, reference attribute grammars and rewriting, are seamlessly integrated, such that rewrites can reuse attributes and attribute values change depending on performed rewrites – a technique we call Reference Attribute Grammar Controlled Rewriting. To reevaluate attributes influenced by abstract syntax tree rewrites, a demand-driven, incremental evaluation strategy, which incorporates the actual execution paths selected at runtime for control-flows within attribute equations, is used. To realise this strategy, a dynamic attribute dependency graph is constructed throughout attribute evaluation – a technique we call Dynamic Attribute Dependency Analyses.

Besides synthesised and inherited attributes, *RACR* supports reference, parameterised and circular attributes, attribute broadcasting and abstract syntax tree and attribute inheritance. *RACR* also supports graph pattern matching to ease the specification of complex rewires, whereas patterns can reuse attributes for rewrite conditions such that complex analyses that control rewriting can be specified. Similarly to attribute values, tests for pattern matches are incrementally evaluated and automatically cached. Further, linear pattern matching complexity is guaranteed if involved attributes are already evaluated. Thus, the main drawback of graph rewriting, the matching problem of polynomial complexity for bounded pattern sizes, is attenuated.

The report illustrates *RACR's* motivation, features, instantiation and usage. In particular its application programming interface is documented and exemplified. The report is a reference manual for *RACR* developers. Further, it presents *RACR's* complete implementation and therefore provides a good foundation for readers interested into the details of reference attribute grammar controlled rewriting and dynamic attribute dependency analyses.

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

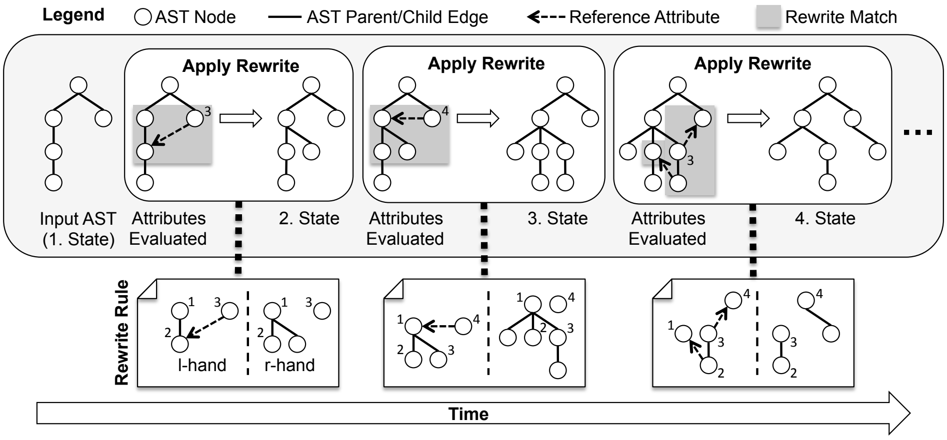

Figure 1.1.: Analyse-Synthesize Cycle of RAG Controlled Rewriting

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)*

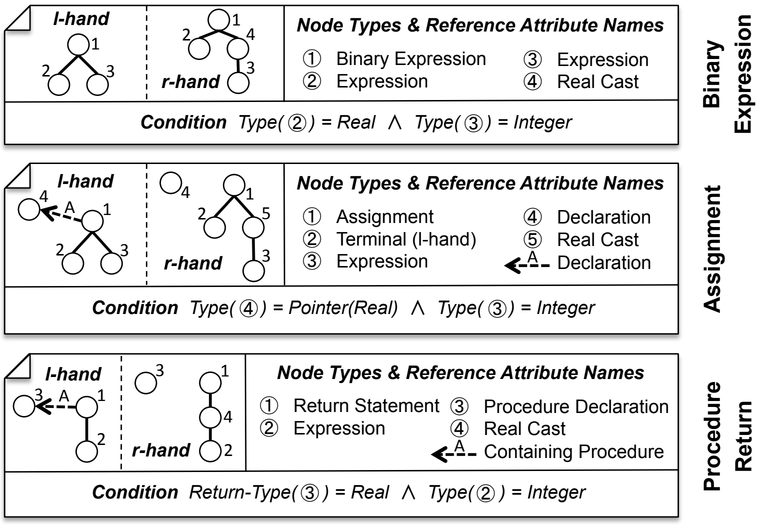

Figure 1.2.: Rewrite Rules for Integer to Real Type Coercion of a Programming Language

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 under the assumption that all involved reference attributes are already evaluated, matching is linear (to be precise, the test if the rest of a pattern matches after mapping its distinguished node to some node of a host graph is constant and to find all matches of a pattern in a host graph is linear). 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*. Different language processors developed using *RACR* can interact with each other without limitations and any need for explicit modeling of such interactions. 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

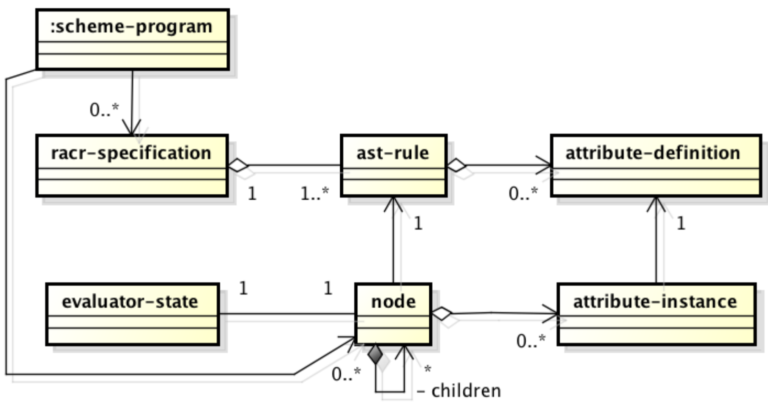

Figure 2.1.: Architecture of RACR Applications

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.

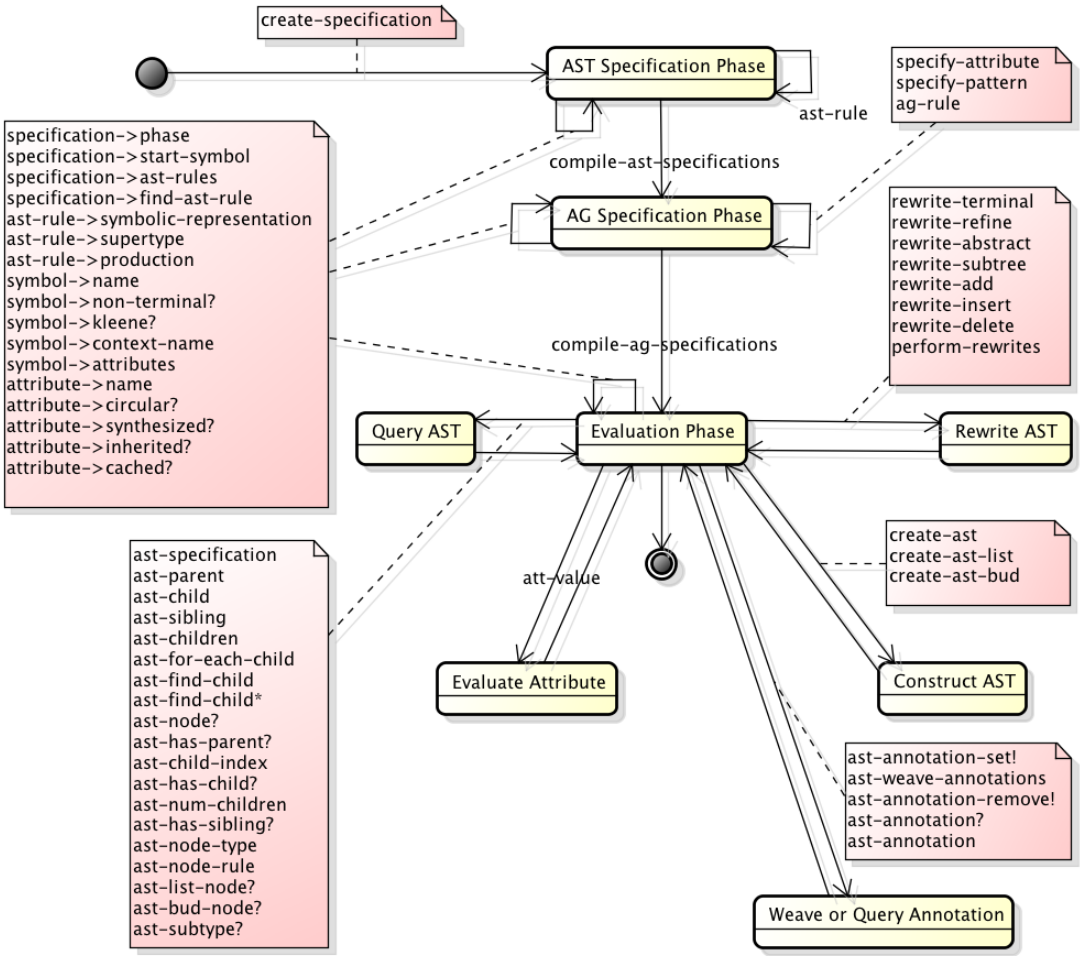

Figure 2.2.: RACR API

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 represents either, another AST node spanning a subtree or an arbitrary *Scheme* value. If it spans a subtree, it has a node type and is called a non-terminal child of the respective type, otherwise a terminal child. Non-terminal children can be unbounded repetitions, which means instead of a single node of the non-terminal's type the child is a so called list node that has arbitrary many children of the respective 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`. Given a node, the number, order, types, names and information, whether they are unbounded repetitions, of its children are induced by its type.

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 induced by 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. The set of all AST rules of a *RACR* specification are called an AST scheme. Every AST rule specifies a node type of a certain name. AST rules are first class entities in *RACR*. They can be queried about the children they induce for nodes of the type they define. It is important to distinguish the names of defined node types from their defining AST rules. Each AST rule precisely defines a node type in the context of a certain *RACR* specification. Its name however, must be unique only regarding the AST rules of the specification it is part of. It is just a *Scheme* symbol without any meaning outside the scope of some *RACR* specification.

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 and (4) all non-terminals are reachable and (5) productive. Further, it is ensured, that (5) 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

```
(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 or has attributes in evaluation.

Note: *Returned list-nodes 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-list` *does not change the children of the constructed list-node.*

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.

```
(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.

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.

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))))
```

```
(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)
```

Similar to `ast-find-child`, except instead of the first child satisfying `f` the result of `f` for the respective child is returned. If no child satisfies `f`, `#f` is returned.

```
(let ((ast
       (with-specification
        (create-specification)
        (ast-rule 'A->B)
        (ast-rule 'B->t)
        (compile-ast-specifications 'A)
        (compile-ag-specifications)
        (create-ast 'A (list (create-ast 'B (list 1)))))))
  (assert
   (ast-node?
    (ast-find-child ; Return the first child satisfying the search condition
     (lambda (i c)
       (ast-child 't c))
     ast)))
  (assert
   (=
    (ast-find-child* ; Return test result of the first child satisfying the search condition
     (lambda (i c)
       (ast-child 't c))
     ast)
    1)))
```

3.4. Node Information

`(ast-node? scheme-entity)`

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

`(ast-has-parent? n)`

Given a node, return its parent if it has any and `#f` otherwise.

`(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-has-child? context-name n)`

Given a node and context name, return whether the node has a child with the given name or not.

```
(ast-num-children n)
```

Given a node, return its number of children.

```
(ast-has-sibling? context-name n)
```

Given a node and context name, return whether the node has a parent node that has a child with the given name or not.

```
(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-node-rule n)
```

Given a node, return the AST rule it represents a derivation 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.

```
(ast-bud-node? n)
```

Given a node, return whether it 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: Name of the specified attribute (Scheme symbol).
; non-terminal: AST rule R in whose context the attribute is defined (Scheme symbol).
; 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))
```

```
(specify-pattern
 spec ; RACR specification
 att-name ; Name of the specified pattern attribute (a Scheme symbol).
 ; Pattern specification consisting of:
 distinguished-node ; Name of the distinguished node (a Scheme symbol).
 fragments ; List of connected AST fragments reachable from the distinguished node.
 references ; List of references connecting pattern nodes.
 condition?) ; #f or function restricting the applicability of the pattern.
```

Calling this function adds to the given *RACR* specification an attribute of the given name that can be used to decide if the given pattern matches at the location a queried instance of the attribute is associated with. The attribute's definition context is the type of the distinguished node of the given pattern (cf. below for distinguished node). Attributes defined using `specify-pattern` are called pattern attribute. They are ordinary *RACR* attributes.

The specified pattern of a pattern attribute consists of arbitrary many AST fragments (`fragments` argument), references between nodes of fragments (`references` argument), a distinguished node that is part of some fragment (`distinguished-node` argument) and a condition (`condition?` argument). Each reference represents a directed edge induced by a parameterless reference attribute. To describe fragments and references, *Scheme* lists of the following structure are used (`(` and `)` are the ordinary list delimiters and `Scheme-Symbol` and `Scheme-Integer` are arbitrary *Scheme* symbol and integer values respectively):

```
fragments ::= Fragment*;
Fragment ::= (TypeRestriction NodeName (Node*));
Node ::= (ContextName TypeRestriction NodeName (Node*));
ContextName ::= Scheme-Symbol | Scheme-Integer;
TypeRestriction ::= "#f" | Scheme-Symbol;
NodeName ::= "#f" | Scheme-Symbol;

references ::= (Reference*);
Reference ::= (AttributeName Source Target);
AttributeName ::= Scheme-Symbol;
Source ::= Scheme-Symbol;
Target ::= Scheme-Symbol;
```

The non-terminals of the above grammars have the following semantics regarding the homomorph mapping of pattern nodes and edges to a host graph:

- **TypeRestriction** The node must be of the given type or a subtype. In case of `#f` it can be of arbitrary type (including list and bud nodes).

- **NodeName** If the node name is not `#f`, the node `h` of the host graph the respective pattern node is mapped to is bound to the given name. To do so a pair of the form `(node-name host-graph-node)` is constructed. The list of these pairs is called binding list. It is a proper association list and returned if the pattern matches. Node names are also required to specify the distinguished node and references between nodes. The distinguished node of a pattern is the node named liked the given `distinguished-node` argument when `specify-pattern` was called.
- **ContextName** If the context name is a *Scheme* symbol `s`, the node must be an `s` child. If the context is an integer `i`, the node must be the `i`'th element of a list.
- **Node*** The children of a node.
- **AttributeName** Name of the parameterless reference attribute inducing the respective reference edge.
- **Source** Name of the node the reference edge starts from.
- **Target** Name of the node the reference edge points to.

Given an attribute instance for a pattern, the pattern does not match at the respective instance, if and only if, it does not match structurally or conditionally. A pattern does not match structurally, if and only if, mapping the distinguished node of the pattern to the node its respective pattern attribute instance is associated with, the rest of the pattern cannot be mapped. A pattern does not match conditionally, if and only if, it does not match structurally or it has a condition which is not satisfied. A given condition is not satisfied if, and only if, it evaluates to `#f` when applied to the binding list constructed throughout structural matching and the arguments given when calling the pattern attribute. If `#f` is given as `condition?` argument, the pattern has no condition, otherwise the given `condition?` argument is used as condition during matching.

If a pattern attribute instance is queried and its pattern does not match at the instance, it returns `#f`, otherwise it returns the binding list constructed throughout structural matching.

Patterns must satisfy certain conditions to be well-formed:

- A node of the pattern is named like the given `distinguished-node` argument.
- Node names are unique.
- The distinguished node has a type restriction which is not a list.
- If a node of the pattern has a type restriction, the type is defined according to the given *RACR* specification.
- For every source and target of a reference there exists an equally named node in the pattern.
- All nodes of the pattern are reachable from the distinguished node considering bidirectional child/parent edges and directed references.
- There exists an AST w.r.t. the given *RACR* specification where the pattern matches only considering its AST fragments. In case of any violation, `specify-pattern` throws an exception.

Note: *The binding list given to a condition for its evaluation is **not** a copy. Conditions can manipulate it and therefore the bindings returned in case the pattern matches. This is considered to be bad style however.*

Note: *Conditions often depend on nodes bound throughout structural matching. To ease their development, the `with-bindings` form can be used. It provides convenient means to establish bindings for matched nodes without manual searches in binding lists.*

Note: *Patterns can be used for the convenient specification of complex AST rewrites. The general idea is to specify patterns that only match in situations when to rewrite and bind all the nodes relevant for rewriting. The patterns, in combination with `perform-rewrites` and `create-transformer-for-pattern`, can then be reused to define a set of transformers that rewrite the AST until no further rewrites are possible. This scenario corresponds to the application of a set of rewrite rules on a host graph. The patterns are the l-hands of the rewrite rules, the transformers are their respective r-hands and `perform-rewrites` is their application.*

Note: *Attributes specified using `specify-pattern` are ordinary non-circular synthesised attributes. They are inherited, shadowed, broadcasted, automatically cached, demand-driven and incrementally evaluated and can reuse other attributes in pattern conditions. They can also be parameterised, in which case their condition is applied to the binding list and the additional arguments, if they have any condition at all.*

Note: *Assuming the reference edges induced by reference attributes of a pattern are already evaluated, RACR guarantees constant structural matching time for the evaluation of pattern attributes. In consequence, matching is linear in RACR and not polynomial as in general graph rewriting with bounded pattern sizes. To find all matches of a pattern, each node of the host graph has to be visited once and, in case the node is of the type of the pattern's distinguished node, its respective pattern attribute evaluated.*

```
(specify-pattern
 some-racr-specification
 some-pattern-attribute-name
 some-distinguished-node-name
 some-ast-fragments
 some-references
 ; Assume the pattern establishes bindings A and B, we can specify
 ; a pattern condition depending on both:
 (with-bindings (A B)
  ...)) ; Arbitrary code that somehow uses A and B
```

```
(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			×				
	Does not fit in 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. If rewriting succeeds, the old/rewritten value of the terminal is returned.

Note: *`rewrite-terminal` does not compare the old and new value for equivalence. If they are equal, the rewrite is still performed such that the caches of depending attributes are flushed. Developers are responsible to avoid such unnecessary rewrites.*

```
(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 all influenced attributes are flushed and dependencies are maintained. An exception is thrown, if `t` is not a supertype of `n`'s current type, `t` does not fit w.r.t. the context in which `n` is 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 node `n`, part of some AST fragment G called work graph, a strategy for traversing G and a set of transformers, apply the transformers on the nodes of G 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 the transformers one by one 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 of G to visit. If there are no further nodes to visit (i.e., all nodes to visit have been visited and no transformer performed any rewrite) `perform-rewrites` terminates. If any transformer performed a rewrite, the visit strategy starts from scratch, i.e., the AST fragment resulting after the just performed rewrite becomes the new work graph G' and the visit strategy is initialised to the state it would have if `perform-rewrites` has just been applied on G' with the given transformers.

`Perform-rewrites` supports two general visit strategies, both deduced from 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 at any time a transformer performs a rewrite that inserts the work graph G into another 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 executed transformers).

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 developer to ensure, that transformers are properly implemented, i.e., they return `#f` if, and only if, they perform no 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.*

Note: *To ease the development of transformers for patterns specified using `specify-pattern`, the `create-transformer-for-pattern` function can be used. Similarly to attribute specifications it provides convenient means to describe the context for which a transformer is defined and the pattern that must match for the transformer to apply. The actual rewrites that reuse the bindings established by respective pattern attributes can be conveniently specified using `with-bindings`.*

```
(create-transformer-for-pattern spec node-type att-name rewrite-fun . args)
 ; spec: RACR specification
 ; node-type: Type of the pattern's distinguished node (Scheme symbol)
 ; att-name: Name of the pattern attribute (Scheme symbol)
 ; rewrite-fun: Function performing the actual rewrite
 ; args: Additional arguments for the pattern's condition and the rewrite function
```

Given a *RACR* specification, a context describing a certain pattern attribute, an arbitrary function called rewrite function and additional attribute arguments, construct a proper transformer that can be argument for `perform-rewrites`. The constructed transformer returns `#f` if, and only if, either, its node argument does not have a non-broadcasted instance of the given pattern attribute or its respective instance applied to the given pattern arguments returns `#f`. Otherwise, the transformer applies the given rewrite function to the binding list returned by the pattern attribute and the given pattern arguments. If the result of this application is not `#f`, it is the result of the transformer; otherwise the transformer returns `#t`.

An exception is thrown, if the given pattern attribute is not defined considering the given *RACR* specification, the given rewrite function's arity is not `|pattern-arguments| + 1` or it throws an exception throughout execution.

Note: *For the convenient specification of rewrite functions,* `with-bindings` *can be used to provide bindings for the pattern's named nodes. The actual rewrites have to be performed by primitive rewrite function applications within the rewrite function's body. The constraints of such primitive rewrites must be satisfied however, or exceptions are thrown as used to. It is the developers responsibility to ensure, that rewrite functions indeed perform any rewrites at all if applied. In particular, applied rewrite functions should manipulate the AST in such a way that their respective pattern attribute evaluates to* `#f` *at some point, otherwise* `perform-rewrites` *will not terminate.*

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 bound). 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))
```

```
(with-bindings ((association-list-keys ...)
                (parameter-names ...))
  ; Arbitrary code with references to keys and parameters:
  ...)
(with-bindings (association-list-keys ...)
  ; Arbitrary code with references to keys:
  ...)
```

Syntax form, that given a list of key variables `k`, an optional list of parameter variables `p` and arbitrary many s-expressions `s` constructs an `1 + |p|` arity function `f` whose body is `s` and which provides for each key in `k` and parameter in `p` a respective binding. The bindings are established as follows: The first argument of the constructed function `f` must be an association list `l`. Each key in `k` is bound to the `cdr` of its respective entry in `l`. The further arguments of `f` are bound to the respective parameter variables in `p` regarding the order of `p` (i.e., the second argument is bound to the first variable in `p`, the third to the second, etc.).

An exception is thrown, if the first argument to the constructed function `f` is not an association list, does not contain a key for a variable in `k` or the number of further arguments does not equal the number of parameter variables in `p`.

Note: *`With-bindings` eases the specification of pattern conditions for `specify-pattern` and of transformers for these patterns using `create-transformer-for-pattern`. Using `with-bindings`, developers can denote the nodes bound throughout matching without writing boilerplate code to search through and bind the values of returned binding lists.*

```
(assert
 (=
  ((with-bindings (A C)
    (+ A C))
   (list (cons 'A 1) (cons 'B 2) (cons 'C 3) (cons 'D 4)))
  4))
(assert
 (=
  ((with-bindings ((A B)
                   (X Y Z))
    (- (+ A B Z) X Y))
   (list (cons 'A 1) (cons 'B 2))
   100
   200
   1000)
```

```
703))
```

```
(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. Bibliography

RACR is based on previous research in the fields of attribute grammars and rewriting. For convenient programming, *RACR* developers should be familiar with the basic concepts of these fields. This includes attribute grammar extensions and techniques like reference, parameterised and circular attributes and demand-driven and incremental attribute evaluation and rewriting basics like matching and rules consisting of left- and righ-hand sides.

To understand the advantages, in particular regarding expressiveness and complexity, of combining attribute grammars and rewriting, it is also helpful to know basic rewrite approaches, their limitations and relationships (term rewriting, context-free and sensitive graph rewriting). Knowledge in programmed or strategic rewriting may be additionally helpful to get started in the development of more complex rewrites whose applications are steered by attributes.

The following bibliography summarises the literature most important for *RACR*. It is grouped w.r.t. attribute grammars and rewriting and respective research problems. References are not exclusively classified; Instead references are listed in all problem categories they are related to. To support *Scheme* and compiler construction novices, also some basic literature is given. It is highly recommended to become used to *Scheme* programming and compiler construction concepts before looking into *RACR*, attribute grammar or rewriting details. An overview of recent and historically important attribute grammar and rewrite systems and applications complements the bibliography.

Scheme Programming

[1] Harold Abelson, Gerald J. Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. 2nd ed. MIT Press, 1996. ISBN: 0-262-51087-1.

[2] R. Kent Dybvig. *The Scheme Programming Language*. 4th ed. MIT Press, 2009. ISBN: 978-0-262-51298-5.

Compiler Construction: Introduction and Basics

[1] Harold Abelson, Gerald J. Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. 2nd ed. MIT Press, 1996. ISBN: 0-262-51087-1.

[3] Alfred V. Aho et al. *Compilers: Principles, Techniques, & Tools*. 2nd ed. Prentice Hall, 2006. ISBN: 978-0-321-48681-3.

[4] Uwe Kastens. *Übersetzerbau*. Ed. by Albert Endres, Hermann Krallmann, and Peter Schnupp. Vol. 3.3. Handbuch der Informatik. Oldenbourg, 1990. ISBN: 3-486-20780-6.

[5] Lothar Schmitz. *Syntaxbasierte Programmierwerkzeuge*. Leitfäden der Informatik. Teubner, 1995. ISBN: 3-519-02140-4.

[6] Reinhard Wilhelm and Dieter Maurer. *Compiler Design*. Addison-Wesley, 1995. ISBN: 0-201-42290-5.

[7] Niklaus Wirth. *Grundlagen und Techniken des Compilerbaus*. 2nd ed. Oldenbourg, 2008. ISBN: 978-3-486-58581-0.

Attribute Grammar Foundations

[8] J. Craig Cleaveland and Robert C. Uzgalis. *Grammars for Programming Languages*. Ed. by Thomas E. Cheatham. Vol. 4. Programming Languages Series. Elsevier, 1977. ISBN: 0-444-00187-5.

[9] Pierre Deransart, Martin Jourdan, and Bernard Lorho. *Attribute Grammars: Definitions, Systems and Bibliography*. Lecture Notes in Computer Science. Springer, 1988. ISBN: 978-3-540-50056-8.

[10] Martin Jourdan. "An Optimal-time Recursive Evaluator for Attribute Grammars". In: *International Symposium on Programming: 6th Colloquium*. Ed. by Manfred Paul and Bernard Robinet. Vol. 167. Lecture Notes in Computer Science. Toulouse, Haute-Garonne, France: Springer, Apr. 1984, pp. 167–178. ISBN: 978-3-540-12925-7.

[11] Donald E. Knuth. "Semantics of Context-Free Languages". In: *Theory of Computing Systems* 2.2 (June 1968), pp. 127–145. ISSN: 1432-4350.

[12] Donald E. Knuth. "Semantics of Context-Free Languages: Correction". In: *Theory of Computing Systems* 5.2 (June 1971), pp. 95–96. ISSN: 1432-4350.

[13] Armin Kühnemann and Heiko Vogler. *Attributgrammatiken: Eine grundlegende Einführung*. Vieweg, 1997. ISBN: 3-528-05582-0.

[14] Jukka Paakki. "Attribute Grammar Paradigms: A High-Level Methodology in Language Implementation". In: *ACM Computing Surveys* 27.2 (June 1995), pp. 196–255. ISSN: 0360-0300.

Attribute Grammar Extensions

[15] John T. Boyland. "Remote attribute grammars". In: *Journal of the ACM* 52.4 (July 2005), pp. 627–687. ISSN: 0004-5411.

[16] Peter Dencker. *Generative attribuierte Grammatiken*. Vol. 158. Berichte der Gesellschaft für Mathematik und Datenverarbeitung. PhD thesis. Oldenbourg, 1986. ISBN: 3-486-20199-9.

[17] Rodney Farrow. “Automatic Generation of Fixed-Point-Finding Evaluators for Circular, but Well-defined, Attribute Grammars”. In: *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*. Ed. by Richard L. Wexelblat. Palo Alto, California, United States: ACM, June 1986, pp. 85–98. ISBN: 0-89791-197-0.

[18] Görel Hedin. “An Object-Oriented Notation for Attribute Grammars”. In: *ECOOP’89: Proceedings of the 1989 European Conference on Object-Oriented Programming*. Ed. by Stephen A. Cook. Nottingham, England, United Kingdom: Cambridge University Press, July 1989, pp. 329–345. ISBN: 0-521-38232-7.

[19] Görel Hedin. “Reference Attributed Grammars”. In: *Informatica (Slovenia)* 24.3 (July 2000), pp. 301–317. ISSN: 0350-5596.

[20] Eva Magnusson. “Object-Oriented Declarative Program Analysis”. PhD thesis. University of Lund, Dec. 2007. ISBN: 978-91-628-7306-6.

[21] Eva Magnusson and Görel Hedin. “Circular Reference Attributed Grammars: Their Evaluation and Applications”. In: *Science of Computer Programming* 68.1 (Aug. 2007), pp. 21–37. ISSN: 0167-6423.

[22] Harald H. Vogt, Doaitse Swierstra, and Matthijs F. Kuiper. “Higher Order Attribute Grammars”. In: *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*. Ed. by Richard L. Wexelblat. Portland, Oregon, USA: ACM, June 1989, pp. 131–145. ISBN: 0-89791-306-X.

Incremental Attribute Evaluation

[23] John T. Boyland. “Incremental Evaluators for Remote Attribute Grammars”. In: *Electronic Notes in Theoretical Computer Science* 65.3 (Apr. 2002), pp. 9–29. ISSN: 1571-0661.

[24] Alan J. Demers, Thomas W. Reps, and Tim Teitelbaum. “Incremental Evaluation for Attribute Grammars with Application to Syntax-Directed Editors”. In: *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Ed. by John White, Richard J. Lipton, and Patricia C. Goldberg. Williamsburg, Virginia, USA: ACM, Jan. 1981, pp. 105–116. ISBN: 0-89791-029-X.

[25] Roger Hoover and Tim Teitelbaum. “Efficient Incremental Evaluation of Aggregate Values in Attribute Grammars”. In: *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*. Ed. by Richard L. Wexelblat. Palo Alto, California, USA: ACM, June 1986, pp. 39–50. ISBN: 0-89791-197-0.

[26] William H. Maddox III. “Incremental Static Semantic Analysis”. PhD thesis. University of California at Berkeley, May 1997.

[27] Thomas Reps, Tim Teitelbaum, and Alan Demers. “Incremental Context-Dependent Analysis for Language-Based Editors”. In: *ACM Transactions on Programming Languages and Systems* 5.3 (July 1983), pp. 449–477. ISSN: 0164-0925.

[28] Thomas W. Reps. “Generating Language-Based Environments”. PhD thesis. Cornell University, Aug. 1982.

Attribute Grammar Systems and Applications

[29] Torbjörn Ekman. "Extensible Compiler Construction". PhD thesis. University of Lund, June 2006. ISBN: 91-628-6839-X.

[30] Torbjörn Ekman and Görel Hedin. "The JastAdd System: Modular Extensible Compiler Construction". In: *Science of Computer Programming* 69.1-3 (Dec. 2007), pp. 14–26. ISSN: 0167-6423.

[31] Robert W. Gray et al. "Eli: A Complete, Flexible Compiler Construction System". In: *Communications of the ACM* 35.2 (Feb. 1992), pp. 121–130. ISSN: 0001-0782.

[32] Uwe Kastens. "Attribute Grammars in a Compiler Construction Environment". In: *Attribute Grammars, Applications and Systems: International Summer School SAGA*. Ed. by Henk Alblas and Bořivoj Melichar. Vol. 545. Lecture Notes in Computer Science. Prague, Czechoslovakia: Springer, June 1991, pp. 380–400. ISBN: 978-3-540-54572-9.

[33] Uwe Kastens, Brigitte Hutt, and Erich Zimmermann. *GAG: A Practical Compiler Generator*. Ed. by Gerhard Goos and Juris Hartmanis. Vol. 141. Lecture Notes in Computer Science. Springer, 1982. ISBN: 3-540-11591-9.

[20] Eva Magnusson. "Object-Oriented Declarative Program Analysis". PhD thesis. University of Lund, Dec. 2007. ISBN: 978-91-628-7306-6.

[34] Thomas Reps and Tim Teitelbaum. "The Synthesizer Generator". In: *Proceedings of the first ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. Ed. by William Riddle and Peter B. Henderson. Pittsburgh, Pennsylvania, USA: ACM, Apr. 1984, pp. 42–48. ISBN: 0-89791-131-8.

[35] Thomas Reps and Tim Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Ed. by David Gries. Texts and Monographs in Computer Science. Springer, 1989. ISBN: 978-1-461-39625-3.

[36] Anthony M. Sloane. "Lightweight Language Processing in Kiama". In: *Generative and Transformational Techniques in Software Engineering III: International Summer School*. Ed. by João M. Fernandes et al. Vol. 6491. Lecture Notes in Computer Science. Braga, Norte, Portugal: Springer, July 2011, pp. 408–425. ISBN: 978-3-642-18022-4.

[37] Eric Van Wyk et al. "Silver: An Extensible Attribute Grammar System". In: *Science of Computer Programming* 75.1–2 (Jan. 2010), pp. 39–54. ISSN: 0167-6423.

Graph Rewriting Foundations

[38] Hartmut Ehrig et al. *Fundamentals of Algebraic Graph Transformation*. Ed. by Wilfried Brauer, Grzegorz Rozenberg, and Arto Salomaa. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2006. ISBN: 978-3-540-31187-4.

[39] Sven O. Krumke and Hartmut Noltemeier. *Graphentheoretische Konzepte und Algorithmen*. 3rd ed. Leitfäden der Informatik. Springer Vieweg, 2012. ISBN: 978-3-8348-1849-2.

[40] Manfred Nagl. "Formal Languages of Labelled Graphs". In: *Computing* 16.1–2 (Mar. 1976), pp. 113–137. ISSN: 0010-485X.

[41] Manfred Nagl. *Graph-Grammatiken: Theorie, Anwendungen, Implementierung*. Vieweg, 1979. ISBN: 3-528-03338-X.

[42] Manfred Nagl. "Set Theoretic Approaches to Graph Grammars". In: *Graph-Grammars and Their Application to Computer Science: 3rd International Workshop*. Ed. by Hartmut Ehrig et al. Vol. 291. Lecture Notes in Computer Science. Warrenton, Virginia, USA: Springer, Dec. 1987, pp. 41–54. ISBN: 978-3-540-18771-4.

[43] Tobias Nipkow and Franz Baader. *Term Rewriting and All That*. 2nd ed. Cambridge University Press, 1999. ISBN: 978-0-521-77920-3.

[44] Grzegorz Rozenberg, ed. *Handbook of Graph Grammars and Computing by Graph Transformation: Foundations*. Vol. 1. World Scientific Publishing, 1997. ISBN: 978-9-8102-2884-2.

Programmed Graph Rewriting

[45] Horst Bunke. "Attributed Programmed Graph Grammars and Their Application to Schematic Diagram Interpretation". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 4.6 (June 1982), pp. 574–582. ISSN: 0162-8828.

[46] Horst Bunke. "On the Generative Power of Sequential and Parallel Programmed Graph Grammars". In: *Computing* 29.2 (June 1982), pp. 89–112. ISSN: 0010-485X.

[47] Horst Bunke. "Programmed Graph Grammars". In: *Graph-Grammars and Their Application to Computer Science and Biology: International Workshop*. Ed. by Volker Claus, Hartmut Ehrig, and Grzegorz Rozenberg. Vol. 73. Lecture Notes in Computer Science. Bad Honnef, North Rhine-Westphalia, Germany: Springer, Nov. 1979, pp. 155–166. ISBN: 978-3-540-09525-5.

[48] Markus von Detten et al. *Story Diagrams: Syntax and Semantics*. Tech. rep. tr-ri-12-324. Version 0.2. Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, July 2012.

[49] Thorsten Fischer et al. "Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language and Java". In: *Theory and Application of Graph Transformations: 6th International Workshop*. Ed. by Hartmut Ehrig et al. Vol. 1764. Lecture Notes in Computer Science. Paderborn, North Rhine-Westphalia, Germany: Springer, Nov. 1998, pp. 296–309. ISBN: 3-540-67203-6.

[50] Andreas Schürr. *Operationales Spezifizieren mit programmierten Graphersetzungssystemen: Formale Definitionen, Anwendungsbeispiele und Werkzeugunterstützung*. PhD thesis. Deutscher Universitäts-Verlag, 1991. ISBN: 3-8244-2021-X.

[51] Andreas Schürr. "Programmed Graph Replacement Systems". In: *Handbook of Graph Grammars and Computing by Graph Transformation: Foundations*. Ed. by Grzegorz Rozenberg. Vol. 1. World Scientific Publishing, 1997, pp. 479–545. ISBN: 978-9-8102-2884-2.

[52] Eelco Visser. "A Survey of Strategies in Rule-based Program Transformation Systems". In: *Journal of Symbolic Computation* 40.1 (July 2005), pp. 831–873. ISSN: 0747-7171.

Graph Rewrite Systems and Applications

[53] Martin Bravenboera et al. "Stratego/XT 0.17: A Language and Toolset for Program Transformation". In: *Science of Computer Programming* 72.1–2 (June 2008), pp. 52–70. ISSN: 0167-6423.

[54] James R. Cordy. "Excerpts from the TXL Cookbook". In: *Generative and Transformational Techniques in Software Engineering III: International Summer School*. Ed. by João M. Fernandes et al. Vol. 6491. Lecture Notes in Computer Science. Braga, Norte, Portugal: Springer, Jan. 2011, pp. 27–91. ISBN: 978-3-642-18022-4.

[55] James R. Cordy. "The TXL Source Transformation Language". In: *Science of Computer Programming* 61.3 (Aug. 2006), pp. 190–210. ISSN: 0167-6423.

[56] James R. Cory, Ian H. Carmichael, and Russell Halliday. *The TXL Programming Language*. Tech. rep. Version 10.6. Software Technology Laboratory, Queen's University, July 2012.

[48] Markus von Detten et al. *Story Diagrams: Syntax and Semantics*. Tech. rep. tr-ri-12-324. Version 0.2. Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, July 2012.

[57] Hartmut Ehrig, Gregor Engels, and Hans-Jörg Kreowski, eds. *Handbook of Graph Grammars and Computing by Graph Transformation: Applications, Languages and Tools*. Vol. 2. World Scientific Publishing, 1999. ISBN: 978-9-8102-4020-2.

[58] Claudia. Ermel, Michael Rudolf, and Gabriele Taentzer. "The AGG Approach: Language and Environment". In: *Handbook of Graph Grammars and Computing by Graph Transformation: Applications, Languages and Tools*. Ed. by Hartmut Ehrig, Gregor Engels, and Hans-Jörg Kreowski. Vol. 2. World Scientific Publishing, 1999, pp. 551–603. ISBN: 978-9-8102-4020-2.

[59] Rubino R. Geiß. "Graphersetzung mit Anwendungen im Übersetzerbau". PhD thesis. Universität Fridericiana zu Karlsruhe, Oct. 2007.

[60] Rubino R. Geiß et al. "GrGen: A Fast SPO-Based Graph Rewriting Tool". In: *Graph Transformations: Third International Conference*. Ed. by Andrea Corradini et al. Vol. 4178. Lecture Notes in Computer Science. Natal, Rio Grande do Norte, Brazil: Springer, Sept. 2006, pp. 383–397. ISBN: 978-3-540-38870-8.

[61] Manfred Nagl, ed. *Building Tightly Integrated Software Development Environments: The IPSEN Approach*. Vol. 1170. Lecture Notes in Computer Science. Springer, 1996. ISBN: 978-3-540-61985-7.

[62] Ulrich Nickel, Jörg Niere, and Albert Zündorf. "The FUJABA Environment". In: *Proceedings of the 22nd International Conference on Software Engineering*. Ed. by Anthony Finkelstein. Limerick, Munster, Ireland: ACM, June 2000, pp. 742–745. ISBN: 1-581-13206-9.

[50] Andreas Schürr. *Operationales Spezifizieren mit programmierten Graphersetzungssystemen: Formale Definitionen, Anwendungsbeispiele und Werkzeugunterstützung*. PhD thesis. Deutscher Universitäts-Verlag, 1991. ISBN: 3-8244-2021-X.

[63] Gabriele Taentzer. "AGG: A Tool Environment for Algebraic Graph Transformation". In: *Applications of Graph Transformations with Industrial Relevance: International Workshop*. Ed. by Manfred Nagl, Andreas Schürr, and Manfred Münch. Vol. 1779. Lecture Notes in Computer Science. Kerkrade, Limburg, The Netherlands: Springer, Sept. 2000, pp. 481–488. ISBN: 978-3-540-67658-4.

[64] Eelco Visser. "Program Transformation with Stratego/XT". In: *Domain-Specific Program Generation: International Seminar*. Ed. by Christian Lengauer et al. Vol. 3016. Lecture Notes in Computer Science. Castle Dagstuhl by Wadern, Saarland, Germany: Springer, Mar. 2004, pp. 216–238. ISBN: 978-3-540-22119-7.

[65] Albert Zündorf. *PROgrammierte GRaphErsetzungs Systeme: Spezifikation, Implementierung und Anwendung einer integrierten Entwicklungsumgebung*. PhD thesis. Deutscher Universitäts-Verlag, 1996. ISBN: 3-8244-2075-9.

B. *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))
  ; Specification query interface:
  specification->phase
  specification->start-symbol
  specification->ast-rules
  specification->find-ast-rule
  ast-rule->symbolic-representation
  ast-rule->supertype
  ast-rule->production
  symbol->name
  symbol->non-terminal?
  symbol->kleene?
  symbol->context-name
  symbol->attributes
  attribute->name
  attribute->circular?
  attribute->synthesized?
  attribute->inherited?
  attribute->cached?
  ; ASTs: Specification
  (rename (specify-ast-rule ast-rule))
  compile-ast-specifications
  ; ASTs: Construction
  create-ast
  create-ast-list
  create-ast-bud
  create-ast-mockup
  ; ASTs: Traversal
  ast-parent
  ast-child
  ast-sibling
  ast-children
  ast-for-each-child
  ast-find-child
  ast-find-child*
  ; ASTs: Node Information
  ast-node?
  ast-specification
  ast-has-parent?
  ast-child-index
  ast-has-child?
  ast-num-children
  ast-has-sibling?
  ast-node-type
  ast-node-rule
  ast-list-node?
  ast-bud-node?
  ast-subtype?
  ; Attribution: Specification
  specify-attribute
  specify-pattern
  (rename (specify-ag-rule ag-rule))
  compile-ag-specifications
  ; Attribution: Querying
  att-value
  ; Rewriting: Primitive Rewrite Functions
  rewrite-terminal
  rewrite-refine
  rewrite-abstract
  rewrite-subtree
  rewrite-add
```

```
 rewrite-insert
 rewrite-delete
 ; Rewriting: Rewrite Strategies
 perform-rewrites
 create-transformer-for-pattern
 ; Annotations: Attachment
 ast-annotation-set!
 ast-weave-annotations
 ast-annotation-remove!
 ; Annotations: Querying
 ast-annotation?
 ast-annotation
 ; Support
 with-specification
 with-bindings
 ; Utility interface:
 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))
  (opaque #t)(sealed #t)
  (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?))
  (opaque #t)(sealed #t))

; INTERNAL FUNCTION: Given an AST rule find a certain child context by name. If the rule defines no such
; context, return #f, otherwise the the production symbol defining the respective context.
(define ast-rule-find-child-context
  (lambda (r context-name)
    (find
     (lambda (symbol)
       (eq? (symbol-context-name symbol) context-name))
     (cdr (ast-rule-production r)))))

; 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.
; BEWARE: Only works correct if supertypes are resolved, otherwise an exception can be thrown!
(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.
; BEWARE: Only works correct if supertypes are resolved, otherwise an exception can be thrown!
(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 is part of a certain ast rule and has 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 ast-rule (mutable non-terminal?) kleene? context-name (mutable attributes))
  (opaque #t)(sealed #t))

; Record type for attribute definitions. An attribute definition has a certain name, a definition context
; (i.e., a symbol of an AST rule), an equation and an optional circularity−definition used for 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?)
  (opaque #t)(sealed #t))

; 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)
    (if (attribute-definition-circularity-definition att) #t #f)))

; INTERNAL FUNCTION: Given an attribute definition, return whether it specifies
; a synthesized attribute or not.
(define attribute-definition-synthesized?
  (lambda (att-def)
    (let ((symbol (attribute-definition-context att-def)))
      (eq? (car (ast-rule-production (symbol-ast-rule symbol))) symbol))))

; 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, attribute cache entries they influence and annotations.
(define-record-type node
  (fields
   (mutable evaluator-state)
   (mutable ast-rule)
   (mutable parent)
   (mutable children)
   (mutable attributes)
   (mutable cache-influences)
   (mutable annotations))
  (opaque #t)(sealed #t)
  (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 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 it 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 if it has a parent, otherwise return #f.
(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))))
        #f)))

; 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) and a cache.
(define-record-type attribute-instance
  (fields (mutable definition) (mutable context) cache)
  (opaque #t)(sealed #t)
  (protocol
   (lambda (new)
     (lambda (definition context)
       (new definition context (make-hashtable equal-hash equal? 1))))))

; Record type for attribute cache entries. Attribute cache entries represent the values of
; and dependencies between attribute instances evaluated for certain arguments. The attribute
; instance of which an entry represents a value is called its context. If an entry already
; is evaluated, it caches the result of its context evaluated for its arguments. If an entry is
; not evaluated but its context is circular it stores an intermediate result of its fixpoint
; computation, called cycle value. Entries also track whether they are already in evaluation or
; not, such that the attribute evaluator can detect unexpected cycles.
(define-record-type attribute-cache-entry
  (fields
   (mutable context)
   (mutable arguments)
   (mutable value)
   (mutable cycle-value)
   (mutable entered?)
   (mutable node-dependencies)
   (mutable cache-dependencies)
   (mutable cache-influences))
  (opaque #t)(sealed #t)
  (protocol
   (lambda (new)
     (lambda (att arguments) ; att: The attribute instance for which to construct a cache entry
       (new
        att
        arguments
```

```
        racr-nil
        (let ((circular? (attribute-definition-circularity-definition (attribute-instance-definition att))))
          (if circular?
              (car circular?)
              racr-nil))
        #f
        (list)
        (list)
        (list)))))

; 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 evaluation-stack))
  (opaque #t)(sealed #t)
  (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-evaluation-stack state))) (car (evaluator-state-evaluation-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 &violation 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-continuable
      (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        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;             ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(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 ()
                  (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)))
               (#,(datum->syntax #'k 'specify-pattern)
                (lambda (att-name distinguished-node fragments references condition)
                  (specify-pattern spec* att-name distinguished-node fragments references condition)))
               (#,(datum->syntax #'k 'create-transformer-for-pattern)
                (lambda (node-type pattern-attribute rewrite-function . pattern-arguments)
                  (apply create-transformer-for-pattern spec* node-type pattern-attribute rewrite-function pattern-arguments))))
          (let-syntax ((#,(datum->syntax #'k 'ag-rule)
                       (syntax-rules ()
                         ((_ attribute-name definition (... ...))
                          (specify-ag-rule spec* attribute-name definition (... ...))))))
            body ...))))))

(define-syntax with-bindings
  (syntax-rules ()
    ((_ ((binding ...) (parameter ...)) body body* ...)
     (lambda (l parameter ...)
       (let ((binding (cdr (assq 'binding l))) ...)
         body
         body* ...)))
    ((_ (binding ...) body body* ...)
     (with-bindings ((binding ...) ()) body 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 (ast-list-node? node)) (not (ast-bud-node? node)) (ast-subtype? node type))
        (ast-annotation-set! node name value))
       ((and (ast-list-node? node) (eq? type 'list-node))
        (ast-annotation-set! node name value))
       ((and (ast-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* ((ast-rule ; The parsed AST rule that will be added to the given specification.
               (make-ast-rule
                spec
                rule
                racr-nil
                racr-nil))
              (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.
              (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?))
             (kleene?
              (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
         ast-rule
         (not terminal?)
         kleene?
         (if context-name?
             (string->symbol (list->string context-name?))
             (if kleene?
                 (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)))))
(match-char! #\-)
(match-char! #\>)
(ast-rule-production-set!
 ast-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)))))))
(ast-rule-supertype?-set!
 ast-rule
 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 specification.
             (racr-specification-rules-table spec)
             (symbol-name l-hand)
             ast-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."))

      ;;; 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)
              rule
              (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 (ast-rule)
       (for-each
        (lambda (symbol)
          (unless (eq? (ast-rule-find-child-context ast-rule (symbol-context-name symbol)) symbol)
            (throw-exception
             "Invalid AST grammar; "
             "The context name " (symbol-context-name symbol) " is not unique for rule " (ast-rule-as-symbol ast-rule) ".")))
        (cdr (ast-rule-production ast-rule))))
     rules-list)

    ;;; Ensure, that all non-terminals can be derived from the start symbol:
    (let* ((start-rule (racr-specification-find-rule spec start-symbol))
           (to-check (cons start-rule (ast-rule-subtypes start-rule)))
           (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 (symbol-kleene? symb*)) ; Unbounded repetitions are always productive because of the empty list.
                       (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* ((context? ; ...the given context exists,...
              (if (symbol? context-name-or-position)
                  (if (eq? context-name-or-position '*)
                      (car (ast-rule-production ast-rule))
                      (ast-rule-find-child-context ast-rule context-name-or-position))
                  (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 ".")
                      (list-ref (ast-rule-production ast-rule) context-name-or-position)))))
        (unless context?
          (throw-exception
           "Invalid attribute definition; "
           "The non-terminal " non-terminal " has no " context-name-or-position " context."))
        (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
           context?
           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)
                  (car sub-prod) ; ...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. If the
; extended search finds a solution, appropriate copy propergation attributes (i.e., broadcasters) are added.
; If 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 ; ...If it has, return the found defining attribute instance.
            (let ((parent (node-parent n))) ; ...If no defining attribute instance can be found...
              (if (not parent) ; ...check if there exists a parent node that may provide a definition....
                  (throw-exception ; ...If not, throw an exception,...
                   "AG evaluator exception; "
                   "Cannot access unknown " name " attribute.")
                  (let* ((att (loop parent)) ; ...otherwise proceed the search at the parent node. If 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....
                                (list-ref ; ...If it is a list node the broadcaster's context is...
                                 (ast-rule-production (node-ast-rule (node-parent parent))) ; ...the list node's parent node and...
                                 (node-child-index? parent)) ; ...child position.
                                (list-ref ; ...If the parent node is not a list node the broadcaster's context is...
                                 (ast-rule-production (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*-values (; The evaluator state used and changed throughout evaluation:
                  ((evaluator-state) (values (node-evaluator-state n)))
                  ; The attribute instance to evaluate:
                  ((att) (values (lookup-attribute name n)))
                  ; The attribute's definition:
                  ((att-def) (values (attribute-instance-definition att)))
```

```
            ; The attribute cache entries used for evaluation and dependency tracking:
            ((evaluation-att-cache dependency-att-cache)
             (if (attribute-definition-cached? att-def)
                 ; If the attribute instance is cached, no special action is required, except...
                 (let ((att-cache
                        (or
                         ; ...finding the attribute cache entry to use...
                         (hashtable-ref (attribute-instance-cache att) args #f)
                         ; ...or construct a respective one.
                         (let ((new-entry (make-attribute-cache-entry att args)))
                           (hashtable-set! (attribute-instance-cache att) args new-entry)
                           new-entry))))
                   (values att-cache att-cache))
                 ; If the attribute is not cached, special attention must be paid to avoid the permament storing
                 ; of fixpoint results and attribute arguments on the one hand but still retaining correct
                 ; evaluation which requires these information on the other hand. To do so we introduce two
                 ; different types of attribute cache entries:
                 ; (1) A parameter approximating entry for tracking dependencies and influences of the uncached
                 ;     attribute instance.
                 ; (2) A set of temporary cycle entries for correct cycle detection and fixpoint computation.
                 ; The "cycle-value" field of the parameter approximating entry is misused to store the hashtable
                 ; containing the temporary cycle entries and must be deleted when evaluation finished.
                 (let* ((dependency-att-cache
                         (or
                          (hashtable-ref (attribute-instance-cache att) racr-nil #f)
                          (let ((new-entry (make-attribute-cache-entry att racr-nil)))
                            (hashtable-set! (attribute-instance-cache att) racr-nil new-entry)
                            (attribute-cache-entry-cycle-value-set!
                             new-entry
                             (make-hashtable equal-hash equal? 1))
                            new-entry)))
                        (evaluation-att-cache
                         (or
                          (hashtable-ref (attribute-cache-entry-cycle-value dependency-att-cache) args #f)
                          (let ((new-entry (make-attribute-cache-entry att args)))
                            (hashtable-set!
                             (attribute-cache-entry-cycle-value dependency-att-cache)
                             args
                             new-entry)
                            new-entry))))
                   (values evaluation-att-cache dependency-att-cache))))
            ; Support function that given an intermediate fixpoint result checks if it is different from the
            ; current cycle value and updates the cycle value and evaluator state accordingly:
            ((update-cycle-cache)
             (values
              (lambda (new-result)
                (unless ((cdr (attribute-definition-circularity-definition att-def))
                         new-result
                         (attribute-cache-entry-cycle-value evaluation-att-cache))
                  (attribute-cache-entry-cycle-value-set! evaluation-att-cache new-result)
                  (evaluator-state-ag-cycle-change?-set! evaluator-state #t))))))
; Decide how to evaluate the attribute dependening on whether its value already is cached or its respective
; cache entry is circular, already in evaluation or starting point of a fix-point computation:
(cond
  ; CASE (0): Attribute already evaluated for given arguments:
  ((not (eq? (attribute-cache-entry-value evaluation-att-cache) racr-nil))
   ; Maintaine attribute cache entry dependencies, i.e., if this entry is evaluated throughout the
   ; evaluation of another entry, the other entry depends on this one. Afterwards,...
   (add-dependency:cache->cache dependency-att-cache)
   (attribute-cache-entry-value evaluation-att-cache)) ; ...return the cached value.

  ; CASE (1): Circular attribute that is starting point of a fixpoint computation:
  ((and (attribute-definition-circular? att-def) (not (evaluator-state-ag-in-cycle? evaluator-state)))
   (dynamic-wind
    (lambda ()
      ; Maintaine attribute cache entry dependencies, i.e., if this entry is evaluated throughout the
      ; evaluation of another entry, the other depends on this one. Further this entry depends
      ; on any other entry that will be evaluated through its own evaluation. Further,..
      (add-dependency:cache->cache dependency-att-cache)
      (evaluator-state-evaluation-stack-set!
       evaluator-state
       (cons dependency-att-cache (evaluator-state-evaluation-stack evaluator-state)))
      ; ...mark, that the entry is in evaluation and...
      (attribute-cache-entry-entered?-set! evaluation-att-cache #t)
      ; ...update the evaluator's state that we are about to start a fix-point computation.
      (evaluator-state-ag-in-cycle?-set! evaluator-state #t))
    (lambda ()
      (let loop () ; Start fix-point computation. Thus, as long as...
        (evaluator-state-ag-cycle-change?-set! evaluator-state #f) ; ...an entry's value changes...
        (update-cycle-cache (apply (attribute-definition-equation att-def) n args)) ; ...evaluate this entry.
        (when (evaluator-state-ag-cycle-change? evaluator-state)
         (loop)))
      (let ((result (attribute-cache-entry-cycle-value evaluation-att-cache)))
        ; When fixpoint computation finished update the caches of all circular entries evaluated. To do so,...
        (let loop ((att-cache
                    (if (attribute-definition-cached? att-def)
```

```
                              evaluation-att-cache
                              dependency-att-cache)))
               (let ((att-def (attribute-instance-definition (attribute-cache-entry-context att-cache))))
                 (if (not (attribute-definition-circular? att-def))
                     ; ... ignore non−circular entries and just proceed with the entries they depend on (to
                     ; ensure all strongly connected components within a weakly connected one are updated)....
                     (for-each
                      loop
                      (attribute-cache-entry-cache-dependencies att-cache))
                     ; ... In case of circular entries...
                     (if (attribute-definition-cached? att-def) ; ...check if they have to be cached and...
                         (when (eq? (attribute-cache-entry-value att-cache) racr-nil) ; ...are not already processed....
                           ; ... If so cache them,...
                           (attribute-cache-entry-value-set!
                            att-cache
                            (attribute-cache-entry-cycle-value att-cache))
                           (attribute-cache-entry-cycle-value-set! ; ...reset their cycle values to the bottom value and...
                            att-cache
                            (car (attribute-definition-circularity-definition att-def)))
                           (for-each ; ...proceed with the entries they depend on.
                            loop
                            (attribute-cache-entry-cache-dependencies att-cache)))
                         ; ... If a circular entry is not cached, check if it already is processed....
                         (when (> (hashtable-size (attribute-cache-entry-cycle-value att-cache)) 0)
                           ; ... If not, delete its temporary cycle cache and...
                           (hashtable-clear! (attribute-cache-entry-cycle-value att-cache))
                           (for-each ; ...proceed with the entries it depends on.
                            loop
                            (attribute-cache-entry-cache-dependencies att-cache)))))))
             result))
         (lambda ()
           ; Mark that fixpoint computation finished,...
           (evaluator-state-ag-in-cycle?-set! evaluator-state #f)
           ; the evaluation of the attribute cache entry finished and...
           (attribute-cache-entry-entered?-set! evaluation-att-cache #f)
           ; ... pop the entry from the evaluation stack.
           (evaluator-state-evaluation-stack-set!
            evaluator-state
            (cdr (evaluator-state-evaluation-stack evaluator-state))))))

      ; CASE (2): Circular attribute already in evaluation for the given arguments:
      ((and (attribute-definition-circular? att-def) (attribute-cache-entry-entered? evaluation-att-cache))
       ; Maintaine attribute cache entry dependencies, i.e., if this entry is evaluated throughout the
       ; evaluation of another entry, the other entry depends on this one. Finally,...
       (add-dependency:cache->cache dependency-att-cache)
       ; ... the intermediate fixpoint result is the attribute cache entry's cycle value.
       (attribute-cache-entry-cycle-value evaluation-att-cache))

      ; CASE (3): Circular attribute not in evaluation and entered throughout a fixpoint computation:
      ((attribute-definition-circular? att-def)
       (dynamic-wind
        (lambda ()
          ; Maintaine attribute cache entry dependencies, i.e., if this entry is evaluated throughout the
          ; evaluation of another entry, the other depends on this one. Further this entry depends
          ; on any other entry that will be evaluated through its own evaluation. Further,..
          (add-dependency:cache->cache dependency-att-cache)
          (evaluator-state-evaluation-stack-set!
           evaluator-state
           (cons dependency-att-cache (evaluator-state-evaluation-stack evaluator-state)))
          ; ... mark, that the entry is in evaluation.
          (attribute-cache-entry-entered?-set! evaluation-att-cache #t))
        (lambda ()
          (let ((result (apply (attribute-definition-equation att-def) n args))) ; Evaluate the entry and...
            (update-cycle-cache result) ; ...update its cycle value.
            result))
        (lambda ()
          ; Mark that the evaluation of the attribute cache entry finished and...
          (attribute-cache-entry-entered?-set! evaluation-att-cache #f)
          ; ... pop it from the evaluation stack.
          (evaluator-state-evaluation-stack-set!
           evaluator-state
           (cdr (evaluator-state-evaluation-stack evaluator-state))))))

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

      (else ; CASE (5): Non−circular attribute not in evaluation:
       (dynamic-wind
        (lambda ()
          ; Maintaine attribute cache entry dependencies, i.e., if this entry is evaluated throughout the
```

```
              ; evaluation of another entry, the other depends on this one. Further this entry depends
              ; on any other entry that will be evaluated through its own evaluation. Further,...
              (add-dependency:cache->cache dependency-att-cache)
              (evaluator-state-evaluation-stack-set!
               evaluator-state
               (cons dependency-att-cache (evaluator-state-evaluation-stack evaluator-state)))
              ; ... mark, that the entry is in evaluation.
              (attribute-cache-entry-entered?-set! evaluation-att-cache #t))
            (lambda ()
              (let ((result (apply (attribute-definition-equation att-def) n args))) ; Evaluate the entry and,...
                (when (attribute-definition-cached? att-def) ; ...if caching is enabled,...
                  (attribute-cache-entry-value-set! evaluation-att-cache result)) ; ...cache its value.
                result))
            (lambda ()
              ; Mark that the evaluation of the attribute cache entry finished and...
              (if (attribute-definition-cached? att-def)
                  (attribute-cache-entry-entered?-set! evaluation-att-cache #f)
                  (hashtable-delete! (attribute-cache-entry-cycle-value dependency-att-cache) args))
              ; ... pop it from the evaluation stack.
              (evaluator-state-evaluation-stack-set!
               evaluator-state
               (cdr (evaluator-state-evaluation-stack evaluator-state))))))))))

;              ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;       Specification Query Interface     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;              ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; General Note: Because RACR specifications never change after compilation, there is no need to add and
;   maintain dependencies when attributes query specifications. The specification query API therefore just
;   forwards to the respective internal functions. Lists must be copied before they are returned however.

; Specification Queries:

(define specification->phase
  (lambda (spec)
    (racr-specification-specification-phase spec)))

(define specification->start-symbol
  (lambda (spec)
    (racr-specification-start-symbol spec)))

(define specification->ast-rules
  (lambda (spec)
    (racr-specification-rules-list spec))) ; Already creates copy!

(define specification->find-ast-rule
  (lambda (spec node-type)
    (racr-specification-find-rule spec node-type)))

; AST Rule Queries:

(define ast-rule->symbolic-representation
  (lambda (ast-rule)
    (ast-rule-as-symbol ast-rule)))

(define ast-rule->supertype
  (lambda (ast-rule)
    (ast-rule-supertype? ast-rule)))

(define ast-rule->production
  (lambda (rule)
    (append (ast-rule-production rule) (list)))) ; Create copy!

; Production Symbol Queries:

(define symbol->name
  (lambda (symb)
    (symbol-name symb)))

(define symbol->non-terminal?
  (lambda (symb)
    (symbol-non-terminal? symb)))

(define symbol->kleene?
  (lambda (symb)
    (symbol-kleene? symb)))

(define symbol->context-name
  (lambda (symb)
    (symbol-context-name symb)))

(define symbol->attributes
  (lambda (symbol)
    (append (symbol-attributes symbol) (list)))) ; Create copy!

; Attribute Definition Queries:
```

```
(define attribute->name
  (lambda (att-def)
    (attribute-definition-name att-def)))

(define attribute->circular?
  (lambda (att-def)
    (attribute-definition-circular? att-def)))

(define attribute->synthesized?
  (lambda (att-def)
    (attribute-definition-synthesized? att-def)))

(define attribute->inherited?
  (lambda (att-def)
    (attribute-definition-inherited? att-def)))

(define attribute->cached?
  (lambda (att-def)
    (attribute-definition-cached? att-def)))

;              ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;      Abstract Syntax Tree Query Interface  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;              ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define ast-node? ; Scheme entities are either allocated as AST nodes or never will be => No need to add dependencies!
  (lambda (n)
    (node? n)))

(define ast-specification
  (lambda (n)
    (when (or (ast-list-node? n) (ast-bud-node? n)) ; Remember: Terminal nodes as such are never exposed to users.
      (throw-exception
       "Cannot query specification; "
       "List and bud nodes are not part of any specification."))
    ; The specification of a node can never change => No need to add dependencies!
    (ast-rule-specification (node-ast-rule n))))

(define ast-list-node? ; No dependency tracking needed!
  (lambda (n)
    (node-list-node? n)))

(define ast-bud-node? ; No dependency tracking needed!
  (lambda (n)
    (node-bud-node? n)))

(define ast-node-rule
  (lambda (n)
    (when (or (ast-list-node? n) (ast-bud-node? n)) ; Remember: Terminal nodes as such are never exposed to users.
      (throw-exception
       "Cannot query type; "
       "List and bud nodes have no type."))
    (add-dependency:cache->node-type n)
    (node-ast-rule n)))

(define ast-node-type
  (lambda (n)
    (symbol-name (car (ast-rule-production (ast-node-rule n))))))

(define ast-subtype?
  (lambda (a1 a2)
    (when (or
           (and (ast-node? a1) (or (ast-list-node? a1) (ast-bud-node? a1)))
           (and (ast-node? a2) (or (ast-list-node? a2) (ast-bud-node? a2))))
      (throw-exception
       "Cannot perform subtype check; "
       "List and bud nodes cannot be tested for subtyping."))
    (when (and (not (ast-node? a1)) (not (ast-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:cache->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:cache->node-sub-type a1 t2)
              (cons t1 t2))
            (begin
              (add-dependency:cache->node-sub-type a1 (node-ast-rule a2))
              (add-dependency:cache->node-super-type a2 (node-ast-rule a1))
              (cons (node-ast-rule a1) (node-ast-rule a2))))))))

(define ast-has-parent?
  (lambda (n)
    (let ((parent (node-parent n)))
      (if parent
          (begin
            (add-dependency:cache->node parent)
            parent)
          (begin
            (add-dependency:cache->node-is-root n)
            #f)))))

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

(define ast-has-child?
  (lambda (context-name n)
    (add-dependency:cache->node-defines-context n context-name)
    (if (node-find-child n context-name) #t #f))) ; BEWARE: Never return the child if it exists, but instead just #t!

(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 query non-existent " i (if (symbol? i) "" "'th") " child."))
      (add-dependency:cache->node child)
      (if (node-terminal? child)
          (node-children child)
          child))))

(define ast-has-sibling?
  (lambda (context-name n)
    (let ((parent? (ast-has-parent? n)))
      (and parent? (ast-has-child? context-name parent?)))))

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

(define ast-child-index
  (lambda (n)
    (ast-find-child*
     (lambda (i child)
       (if (eq? child n) i #f))
     (ast-parent n))))

(define ast-num-children
  (lambda (n)
    (add-dependency:cache->node-num-children n)
    (length (node-children n))))

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

(define ast-for-each-child
  (lambda (f n . b)
    (let ((b (if (null? b) (list (cons 1 '*)) b)))
```

```
      (for-each
       (lambda (b)
         (if (eq? (cdr b) '*)
             (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 (cdr b))
                 (f pos (ast-child pos n))
                 (loop (+ pos 1))))))
       b))))

(define ast-find-child
  (lambda (f n . b)
    (call/cc
     (lambda (c)
       (apply
        ast-for-each-child
        (lambda (i child)
          (when (f i child)
            (c child)))
        n
        b)
       #f))))

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

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

(define create-ast
  (lambda (spec rule children)
    ;;; Before constructing the node ensure, that...
    (when (< (racr-specification-specification-phase spec) 3) ; ...the RACR system is completely specified,...
      (throw-exception
       "Cannot construct " rule " fragment; "
       "The RACR specification still must be compiled."))
    (let* ((ast-rule (racr-specification-find-rule spec rule))
           (new-fragment
            (make-node
             ast-rule
             #f
             (list))))
      (unless ast-rule ; ...the given AST rule is defined,...
        (throw-exception
         "Cannot construct " rule " fragment; "
         "Unknown non-terminal/rule."))
      (unless (satisfies-contexts? children (cdr (ast-rule-production ast-rule))) ; ...and the children fit.
        (throw-exception
         "Cannot construct " rule " fragment; "
         "The given children do not fit."))
      ;;; When all constraints are satisfied, construct the new fragment,...
      (node-children-set! ; ...add its children,...
       new-fragment
       (map ; ...set it as parent of each child,...
        (lambda (symbol child)
          (if (symbol-non-terminal? symbol)
              (begin
                (for-each ; ...flush all attribute cache entries depending on any added child being a root,...
                 (lambda (influence)
                   (flush-attribute-cache-entry (car influence)))
                 (filter
                  (lambda (influence)
```

```
                   (vector-ref (cdr influence) 1))
                 (node-cache-influences child)))
               (node-parent-set! child new-fragment)
               child)
             (make-node 'terminal new-fragment child)))
        (cdr (ast-rule-production ast-rule))
        children))
      (distribute-evaluator-state (make-evaluator-state) new-fragment) ; ...distribute the new fragment's evaluator state and...
      (update-synthesized-attribution new-fragment) ; ...initialize its synthesized and...
      (for-each ; ...each child's inherited attributes.
       update-inherited-attribution
       (node-children new-fragment))
      new-fragment))) ; Finally, return the newly constructed fragment.

(define create-ast-list
  (lambda (children)
    ;;; Before constructing the list node ensure, that...
    (let ((new-list
           (make-node
            'list-node
            #f
            (append children (list))))) ; BEWARE: create copy of children!
      (unless
          (for-all ; ...all children fit.
              (lambda (child)
                (valid-list-element-candidate? new-list child))
            children)
        (throw-exception
         "Cannot construct list node; "
         "The given children do not fit."))
      ;;; When all constraints are satisfied,...
      (for-each ; ...flush all attribute cache entries depending on the children being roots,...
       (lambda (child)
         (for-each
          (lambda (influence)
            (flush-attribute-cache-entry (car influence)))
          (filter
           (lambda (influence)
             (vector-ref (cdr influence) 1))
           (node-cache-influences child))))
       children)
      (for-each ; ...set the new list node as parent of every child,...
       (lambda (child)
         (node-parent-set! child new-list))
       children)
      (distribute-evaluator-state (make-evaluator-state) new-list) ; ...construct and distribute its evaluator state and...
      new-list))) ; ...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)))

(define create-ast-mockup
  (lambda (rule)
    (create-ast
     (ast-rule-specification rule)
     (symbol-name (car (ast-rule-production rule)))
     (map
      (lambda (symbol)
        (cond
          ((not (symbol-non-terminal? symbol))
           racr-nil)
          ((symbol-kleene? symbol)
           (create-ast-list (list)))
          (else (create-ast-bud))))
      (cdr (ast-rule-production rule))))))

; INTERNAL FUNCTION: Given two non−terminal nodes, return if the second can replace the first regarding its context.
(define valid-replacement-candidate?
  (lambda (node candidate)
    (if (node-list-node? (node-parent node))
        (valid-list-element-candidate? (node-parent node) candidate)
        (and
         (satisfies-context?
          candidate
          (list-ref (ast-rule-production (node-ast-rule (node-parent node))) (node-child-index? node)))
         (not (node-inside-of? node candidate))))))

; INTERNAL FUNCTION: Given a list node and another node, return if the other node can become element of
; the list node regarding its context.
(define valid-list-element-candidate?
  (lambda (list-node candidate)
    (let ((expected-type? ; If the list node has a parent, its parent induces a type for the list's elements.
           (if (node-parent list-node)
```

```
               (symbol-non-terminal?
                (list-ref
                 (ast-rule-production (node-ast-rule (node-parent list-node)))
                 (node-child-index? list-node)))
               #f)))
      (and ; The given candidate can be element of the list, if (1)...
       (if expected-type? ; ...either,...
           (satisfies-context? candidate expected-type? #f) ; ...the candidate fits regarding the context in which the list is, or,...
           (and ; ... in case no type is induced for the list's elements,...
            (ast-node? candidate) ; ...the candiate is a non-terminal node,...
            (not (node-list-node? candidate)) ; ...not a list node,...
            (not (node-parent candidate)) ; ...not already part of another AST and...
            (not (evaluator-state-in-evaluation? (node-evaluator-state candidate))))) ; ...non of its attributes are in evaluation,...
       (not (node-inside-of? list-node candidate)))))) ; ...and (2) its spaned AST does not contain the list node.

; INTERNAL FUNCTION: Given a node or terminal value and a context, return if the
; node/terminal value can become a child of the given context.
(define satisfies-context?
  (case-lambda
    ((child context)
     (satisfies-context? child (symbol-non-terminal? context) (symbol-kleene? context)))
    ((child non-terminal? kleene?)
     (or ; The given child is valid if either,...
      (not non-terminal?) ; ...a terminal is expected or,...
      (and ; ... in case a non-terminal is expected,...
       (ast-node? child) ; ...the given child is an AST node,...
       (not (node-parent child)) ; ...does not already belong to another AST,...
       (not (evaluator-state-in-evaluation? (node-evaluator-state child))) ; ...non of its attributes are in evaluation and...
       (or
        (node-bud-node? child) ; ...the child either is a bud node or,...
        (if kleene?
            (and ; ... in case a list node is expected,...
             (node-list-node? child) ; ...is a list...
             (for-all ; ...whose children are...
                 (lambda (child)
                    (or ; ... either bud nodes or nodes of the expected type, or,...
                     (node-bud-node? child)
                     (ast-rule-subtype? (node-ast-rule child) non-terminal?)))
               (node-children child)))
            (and ; ... in case a non-list node is expected,...
             (not (node-list-node? child)) ; ...is a non-list node of...
             (ast-rule-subtype? (node-ast-rule child) non-terminal?))))))))) ; ...the expected type.

; INTERNAL FUNCTION: Given list of nodes or terminal values and a list of contexts, return if the
; nodes/terminal values can become children of the given contexts.
(define satisfies-contexts?
  (lambda (children contexts)
    (and
     (= (length children) (length contexts))
     (for-all satisfies-context? children contexts))))

; 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-instance att)
              (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?
                    (symbol-ast-rule (attribute-definition-context (attribute-instance-definition att)))
                    (node-ast-rule n))))))
            (when remove?
              (flush-attribute-instance att))
            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-instance att)
                      (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-instance att))
              remove?))
          (node-attributes n)))))
    ;;; Perform the update:
    (let* ((parent (node-parent n))
           (att-defs
            (cond
              ((not parent)
               (list))
              ((not (node-list-node? parent))
               (symbol-attributes
                (list-ref
                 (ast-rule-production (node-ast-rule parent))
                 (node-child-index? n))))
              ((node-parent parent)
               (symbol-attributes
                (list-ref
                 (ast-rule-production (node-ast-rule (node-parent parent)))
                 (node-child-index? parent))))
              (else (list)))))
      (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-instance att))
             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)))))

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

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

; INTERNAL FUNCTION: See "add-dependency:cache->node-characteristic".
(define add-dependency:cache->node-is-root
  (lambda (influencing-node)
    (add-dependency:cache->node-characteristic influencing-node (cons 1 racr-nil))))

; INTERNAL FUNCTION: See "add-dependency:cache->node-characteristic".
(define add-dependency:cache->node-num-children
  (lambda (influencing-node)
    (add-dependency:cache->node-characteristic influencing-node (cons 2 racr-nil))))

; INTERNAL FUNCTION: See "add-dependency:cache->node-characteristic".
(define add-dependency:cache->node-type
  (lambda (influencing-node)
    (add-dependency:cache->node-characteristic influencing-node (cons 3 racr-nil))))

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

; INTERNAL FUNCTION: See "add-dependency:cache->node-characteristic".
(define add-dependency:cache->node-sub-type
  (lambda (influencing-node comparision-type)
    (add-dependency:cache->node-characteristic influencing-node (cons 5 comparision-type))))

; INTERNAL FUNCTION: See "add-dependency:cache->node-characteristic".
(define add-dependency:cache->node-defines-context
  (lambda (influencing-node context-name)
    (add-dependency:cache->node-characteristic influencing-node (cons 6 context-name))))

; INTERNAL FUNCTION: Given a node N and a correlation C add an dependency-edge marked with C from
; the attribute cache entry 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 cache entry is in evaluation
; no edges are added. The following seven correlations exist:
;  0) Dependency on the existence of the node (i.e., existence of a node at the same location)
;  1) Dependency on the node being a root (i.e., the node has no parent)
;  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
;  6) Dependency on whether the node defines a certain context (i.e., has child with a certain name) or not
(define add-dependency:cache->node-characteristic
  (lambda (influencing-node correlation)
    (let ((dependent-cache (evaluator-state-in-evaluation? (node-evaluator-state influencing-node))))
      (when dependent-cache
        (let ((dependency-vector
               (let ((dc-hit (assq influencing-node (attribute-cache-entry-node-dependencies dependent-cache))))
                 (and dc-hit (cdr dc-hit)))))
          (unless dependency-vector
            (set! dependency-vector (vector #f #f #f #f (list) (list) (list)))
            (attribute-cache-entry-node-dependencies-set!
             dependent-cache
             (cons
              (cons influencing-node dependency-vector)
              (attribute-cache-entry-node-dependencies dependent-cache)))
            (node-cache-influences-set!
             influencing-node
             (cons
              (cons dependent-cache dependency-vector)
              (node-cache-influences influencing-node))))
          (let ((correlation-type (car correlation))
                (correlation-arg (cdr correlation)))
            (vector-set!
             dependency-vector
             correlation-type
             (case correlation-type
               ((0 1 2 3)
```

```
                #t)
               ((4 5 6)
                (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 cache entry C, add an dependency−edge from C to the entry currently
; in evaluation (considering the evaluator state of the AST C is part of) and an influence−edge vice−versa.
; If no attribute cache entry is in evaluation no edges are added.
(define add-dependency:cache->cache
  (lambda (influencing-cache)
    (let ((dependent-cache
           (evaluator-state-in-evaluation?
            (node-evaluator-state
             (attribute-instance-context
              (attribute-cache-entry-context influencing-cache))))))
      (when (and dependent-cache (not (memq influencing-cache (attribute-cache-entry-cache-dependencies dependent-cache))))
        (attribute-cache-entry-cache-dependencies-set!
         dependent-cache
         (cons
          influencing-cache
          (attribute-cache-entry-cache-dependencies dependent-cache)))
        (attribute-cache-entry-cache-influences-set!
         influencing-cache
         (cons
          dependent-cache
          (attribute-cache-entry-cache-influences influencing-cache)))))))

;              ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;       Primitive Rewrite Interface     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;              ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; INTERNAL FUNCTION: Given an attribute instance, flush all its cache entries.
(define flush-attribute-instance
  (lambda (att)
    (call-with-values
     (lambda ()
       (hashtable-entries (attribute-instance-cache att)))
     (lambda (keys values)
       (vector-for-each
        flush-attribute-cache-entry
        values)))))

; INTERNAL FUNCTION: Given an attribute cache entry, delete it and all depending entries.
(define flush-attribute-cache-entry
  (lambda (att-cache)
    (let ((influenced-caches (attribute-cache-entry-cache-influences att-cache))) ; Save all influenced attribute cache entries.
      ; Delete foreign influences:
      (for-each ; For every cache entry I the entry depends on,...
       (lambda (influencing-cache)
         (attribute-cache-entry-cache-influences-set! ; ...remove the influence edge from I to the entry.
          influencing-cache
          (remq att-cache (attribute-cache-entry-cache-influences influencing-cache))))
       (attribute-cache-entry-cache-dependencies att-cache))
      (for-each ; For every node N the attribute cache entry depends on...
       (lambda (node-dependency)
         (node-cache-influences-set!
          (car node-dependency)
          (remp ; ...remove the influence edge from N to the entry.
           (lambda (cache-influence)
             (eq? (car cache-influence) att-cache))
           (node-cache-influences (car node-dependency)))))
       (attribute-cache-entry-node-dependencies att-cache))
      ; Delete the attribute cache entry:
      (hashtable-delete!
       (attribute-instance-cache (attribute-cache-entry-context att-cache))
       (attribute-cache-entry-arguments att-cache))
      (attribute-cache-entry-cache-dependencies-set! att-cache (list))
      (attribute-cache-entry-node-dependencies-set! att-cache (list))
      (attribute-cache-entry-cache-influences-set! att-cache (list))
      ; Proceed flushing, i.e., for every attribute cache entry D the entry originally influenced,...
      (for-each
       (lambda (dependent-cache)
         (flush-attribute-cache-entry dependent-cache)) ; ...flush D.
       influenced-caches))))

; INTERNAL FUNCTION: Given an AST node n, flush all attribute cache entries that depend on
; information of the subtree spaned by n but are outside of it and, if requested, all attribute
; cache entries within the subtree spaned by n that depend on information outside of it.
(define flush-inter-fragment-dependent-attribute-cache-entries
  (lambda (n flush-outgoing?)
    (let loop ((n* n))
      (for-each
       (lambda (influence)
         (unless (node-inside-of? (attribute-instance-context (attribute-cache-entry-context (car influence))) n)
```

```
           (flush-attribute-cache-entry (car influence))))
       (node-cache-influences n*))
      (for-each
       (lambda (att)
         (vector-for-each
          (lambda (att-cache)
            (let ((flush-att-cache?
                   (and
                    flush-outgoing?
                    (or
                     (find
                      (lambda (dependency)
                        (not (node-inside-of? (car dependency) n)))
                      (attribute-cache-entry-node-dependencies att-cache))
                     (find
                      (lambda (influencing-cache)
                        (not (node-inside-of? (attribute-instance-context (attribute-cache-entry-context influencing-cache)) n)))
                      (attribute-cache-entry-cache-dependencies att-cache))))))
              (if flush-att-cache?
                  (flush-attribute-cache-entry att-cache)
                  (for-each
                   (lambda (dependent-cache)
                     (unless (node-inside-of? (attribute-instance-context (attribute-cache-entry-context dependent-cache)) n)
                       (flush-attribute-cache-entry dependent-cache)))
                   (attribute-cache-entry-cache-influences att-cache)))))
          (call-with-values
           (lambda ()
             (hashtable-entries (attribute-instance-cache att)))
           (lambda (key-vector value-vector)
             value-vector))))
       (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.
        (throw-exception
         "Cannot change terminal value; "
         "The given context does not exist or is no terminal."))
      ;;; Everything is fine. Thus,...
      (let ((old-value (node-children n)))
        (for-each ; ...flush all attribute cache entries influenced by the terminal,...
         (lambda (influence)
           (flush-attribute-cache-entry (car influence)))
         (node-cache-influences n))
        (node-children-set! n new-value) ; ...rewrite its value and...
        old-value)))) ; ...return its old value.

(define rewrite-refine
  (lambda (n t . c)
    ;;; Before refining the non-terminal node 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 and...
        (throw-exception
         "Cannot refine node; "
         t " is not a subtype of " (symbol-name (car (ast-rule-production old-rule))) "."))
      (let ((additional-children (list-tail (ast-rule-production new-rule) (length (ast-rule-production old-rule)))))
        (unless (satisfies-contexts? c additional-children) ; ...all additional children fit.
          (throw-exception
           "Cannot refine node; "
           "The given additional children do not fit."))
        ;;; Everything is fine. Thus,...
        (for-each ; ...flush the influenced attribute cache entries, i.e., all entries influenced by the node's...
         (lambda (influence)
           (flush-attribute-cache-entry (car influence)))
```

```
         (filter
          (lambda (influence)
            (or
             (and (vector-ref (cdr influence) 2) (not (null? c))) ; ...number of children,...
             (and (vector-ref (cdr influence) 3) (not (eq? old-rule new-rule))) ; ...type,...
             (find ; ...supertype,...
              (lambda (t2)
                (not (eq? (ast-rule-subtype? t2 old-rule) (ast-rule-subtype? t2 new-rule))))
              (vector-ref (cdr influence) 4))
             (find ; ...subtype or...
              (lambda (t2)
                (not (eq? (ast-rule-subtype? old-rule t2) (ast-rule-subtype? new-rule t2))))
              (vector-ref (cdr influence) 5))
             (find ; ...defined contexts and...
              (lambda (context-name)
                (let ((old-defines-context? (ast-rule-find-child-context old-rule context-name))
                      (new-defines-context? (ast-rule-find-child-context new-rule context-name)))
                  (if old-defines-context? (not new-defines-context?) new-defines-context?)))
              (vector-ref (cdr influence) 6))))
          (node-cache-influences n)))
        (for-each ; ...all entries depending on the new children being roots. Afterwards,...
         (lambda (child)
           (for-each
            (lambda (influence)
              (flush-attribute-cache-entry (car influence)))
            (filter
             (lambda (influence)
               (vector-ref (cdr influence) 1))
             (node-cache-influences child))))
         c)
        (node-ast-rule-set! n new-rule) ; ...update the node's type,...
        (update-synthesized-attribution n) ; ...synthesized attribution,...
        (node-children-set! ; ...insert the new children and...
         n
         (append
          (node-children n)
          (map
           (lambda (child context)
             (let ((child
                    (if (symbol-non-terminal? context)
                        child
                        (make-node 'terminal n child))))
               (node-parent-set! child n)
               (distribute-evaluator-state (node-evaluator-state n) child) ; ...update their evaluator state and...
               child))
           c
           additional-children)))
        (for-each
         update-inherited-attribution ; ...inherited attribution.
         (node-children n)))))

(define rewrite-abstract
  (lambda (n t)
    ;;; Before abstracting the node 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 node is not a list or bud node,...
      (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)))
      (unless (and new-rule (ast-rule-subtype? old-rule new-rule)) ; ...the new type is a supertype and...
        (throw-exception
         "Cannot abstract node; "
         t " is not a supertype of " (symbol-name (car (ast-rule-production old-rule))) "."))
      ; ... permitted in the context in which the node is:
      (unless (or (not (node-parent n)) (valid-replacement-candidate? n (create-ast-mockup new-rule)))
        (throw-exception
         "Cannot abstract node; "
         "Abstraction to type " t " not permitted by context."))
      ;;; Everything is fine. Thus,...
      (let* ((num-new-children (length (cdr (ast-rule-production new-rule))))
             (children-to-remove (list-tail (node-children n) num-new-children)))
        (for-each ; ...flush all influenced attribute cache entries, i.e., all entries influenced by the node's...
         (lambda (influence)
           (flush-attribute-cache-entry (car influence)))
         (filter
          (lambda (influence)
            (or
             (and (vector-ref (cdr influence) 2) (not (null? children-to-remove))) ; ...number of children,...
             (and (vector-ref (cdr influence) 3) (not (eq? old-rule new-rule))) ; ...type...
             (find ; ...supertype,...
              (lambda (t2)
```

```
               (not (eq? (ast-rule-subtype? t2 old-rule) (ast-rule-subtype? t2 new-rule))))
             (vector-ref (cdr influence) 4))
            (find ; …subtype or…
             (lambda (t2)
               (not (eq? (ast-rule-subtype? old-rule t2) (ast-rule-subtype? new-rule t2))))
             (vector-ref (cdr influence) 5))
            (find ; …defined contexts and…
             (lambda (context-name)
               (let ((old-defines-context? (ast-rule-find-child-context old-rule context-name))
                     (new-defines-context? (ast-rule-find-child-context new-rule context-name)))
                 (if old-defines-context? (not new-defines-context?) new-defines-context?)))
             (vector-ref (cdr influence) 6))))
         (node-cache-influences n)))
       (for-each ; …all entries cross−depending the removed ASTs. Afterwards,…
        (lambda (child-to-remove)
          (flush-inter-fragment-dependent-attribute-cache-entries child-to-remove #t))
        children-to-remove)
       (node-ast-rule-set! n new-rule) ; …update the node's type and its…
       (update-synthesized-attribution n) ; …synthesized (because of possibly less) and…
       (update-inherited-attribution n) ; …inherited (because of unshadowed) attributes. Further,…
       (for-each ; …for every child to remove,…
        (lambda (child)
          (detach-inherited-attributes child) ; …delete its inherited attributes,…
          (node-parent-set! child #f) ; …detach it from the AST and…
          (distribute-evaluator-state (make-evaluator-state) child)) ; …update its evaluator state. Then,…
        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 (evaluator-state-in-evaluation? (node-evaluator-state l)) ; …no attributes of the list are in evaluation,…
       (throw-exception
        "Cannot add list element; "
        "There are attributes in evaluation."))
     (unless (node-list-node? l) ; …indeed a list is given as context and…
       (throw-exception
        "Cannot add list element; "
        "The given context is no list-node."))
     (unless (valid-list-element-candidate? l e) ; …the new element fits.
       (throw-exception
        "Cannot add list element; "
        "The new element does not fit."))
     ;;; When all rewrite constraints are satisfied,…
     (for-each ; …flush all attribute cache entries influenced by the list−node's number of children and…
      (lambda (influence)
        (flush-attribute-cache-entry (car influence)))
      (filter
       (lambda (influence)
         (vector-ref (cdr influence) 2))
       (node-cache-influences l)))
     (for-each ; …all entries depending on the new element being a root. Afterwards,…
      (lambda (influence)
        (flush-attribute-cache-entry (car influence)))
      (filter
       (lambda (influence)
         (vector-ref (cdr influence) 1))
       (node-cache-influences e)))
     (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 no attributes of the old fragment are in evaluation and…
     (when (evaluator-state-in-evaluation? (node-evaluator-state old-fragment))
       (throw-exception
        "Cannot replace subtree; "
        "There are attributes in evaluation."))
     (unless (valid-replacement-candidate? old-fragment new-fragment) ; …the new fragment fits in its context.
       (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. Then,…
     ; … flush all attribute cache entries cross−depending the old fragment and…
     (flush-inter-fragment-dependent-attribute-cache-entries old-fragment #t)
```

```
    (for-each ; …all entries depending on the new fragment being a root. Afterwards,…
     (lambda (influence)
       (flush-attribute-cache-entry (car influence)))
     (filter
      (lambda (influence)
        (vector-ref (cdr influence) 1))
      (node-cache-influences new-fragment)))
    (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 new element ensure, that…
    (when (evaluator-state-in-evaluation? (node-evaluator-state l)) ; …no attributes of the list are in evaluation,…
      (throw-exception
       "Cannot insert list element; "
       "There are attributes in evaluation."))
    (unless (node-list-node? l) ; …indeed a list 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 and…
      (throw-exception
       "Cannot insert list element; "
       "The given index is out of range."))
    (unless (valid-list-element-candidate? l e) ; …the new element fits.
      (throw-exception
       "Cannot add list element; "
       "The new element does not fit."))
    ;;; When all rewrite constraints are satisfied…
    (for-each ; …flush all attribute cache entries influenced by the list's number of children. Further,…
     (lambda (influence)
       (flush-attribute-cache-entry (car influence)))
     (filter
      (lambda (influence)
        (vector-ref (cdr influence) 2))
      (node-cache-influences l)))
    (for-each ; …for each tree spaned by the successor element's of the insertion position,…
     ; … flush all attribute cache entries depending on, but still outside of, the respective tree. Then,…
     (lambda (successor)
       (flush-inter-fragment-dependent-attribute-cache-entries successor #f))
     (list-tail (node-children l) (- i 1)))
    (for-each ; …flush all attribute cache entries depending on the new element being a root. Afterwards,…
     (lambda (influence)
       (flush-attribute-cache-entry (car influence)))
     (filter
      (lambda (influence)
        (vector-ref (cdr influence) 1))
      (node-cache-influences e)))
    (let ((insert-head (list-tail (node-children l) (- i 1)))) ; …insert the new element,…
      (set-cdr! insert-head (cons (car insert-head) (cdr insert-head)))
      (set-car! insert-head e))
    (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 element of a list.
      (throw-exception
       "Cannot delete list element; "
       "The given node is not element of a list."))
    ;;; When all rewrite constraints are satisfied, flush all attribute cache entries influenced by…
    (for-each ; …the number of children of the list node the element is part of. Further,…
     (lambda (influence)
       (flush-attribute-cache-entry (car influence)))
     (filter
      (lambda (influence)
        (vector-ref (cdr influence) 2))
      (node-cache-influences (node-parent n))))
    (detach-inherited-attributes n) ; …delete the element's inherited attributes and…
    (flush-inter-fragment-dependent-attribute-cache-entries n #t) ; …the attribute cache entries cross—depending its subtree…
    (for-each ; …and for each tree spaned by its successor elements,…
```

```
    ; ... flush all attribute cache entries depending on, but still outside of, the respective tree. Then,...
    (lambda (successor)
      (flush-inter-fragment-dependent-attribute-cache-entries successor #f))
   (list-tail (node-children (node-parent n)) (node-child-index? n)))
  (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.

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

(define perform-rewrites
  (lambda (n strategy . transformers)
    (define root
      (let loop ((n n))
        (if (ast-has-parent? n)
            (loop (ast-parent n))
            n)))
    (define root-deleted/inserted?
      (let ((evaluator-state (node-evaluator-state root)))
        (lambda ()
          (not (eq? evaluator-state (node-evaluator-state root))))))
    (define find-and-apply
      (case strategy
        ((top-down)
         (lambda (n)
           (and
            (not (node-terminal? n))
            (or
             (find (lambda (transformer) (transformer 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 (transformer) (transformer n)) transformers)))))
        (else (throw-exception
               "Cannot perform rewrites; "
               "Unknown " strategy " strategy."))))
    (let loop ()
      (when (root-deleted/inserted?)
        (throw-exception
         "Cannot perform rewrites; "
         "A given transformer manipulated the root of the AST."))
      (let ((match (find-and-apply root)))
        (if match
            (cons match (loop))
            (list))))))

(define create-transformer-for-pattern
  (lambda (spec node-type pattern-attribute rewrite-function . pattern-arguments)
    (let ((ast-rule (specification->find-ast-rule spec node-type)))
      (unless ast-rule
        (throw-exception
         "Cannot construct transformer; "
         "Undefined " node-type " node type."))
      (unless (find
               (lambda (attribute-definition)
                 (eq? (attribute->name attribute-definition) pattern-attribute))
               (symbol->attributes (car (ast-rule->production ast-rule))))
        (throw-exception
         "Cannot construct transformer; "
         "No " pattern-attribute " attribute defined in the context of " node-type " nodes."))
    (lambda (n)
      (when (and (not (or (ast-bud-node? n) (ast-list-node? n))) (ast-subtype? n node-type))
        (let ((match? (apply att-value pattern-attribute n pattern-arguments)))
          (if match?
              (or
               (apply rewrite-function match? pattern-arguments)
               #t)
              #f)))))))

;          ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;     Pattern Matching     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;          ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define pattern-language (make-racr-specification))

(define specify-pattern
  (lambda (spec att-name distinguished-node fragments references condition?)
    (define process-fragment
      (lambda (context type binding children)
```

```
        (unless (and
                 (or (symbol? context) (integer? context))
                 (or (not type) (symbol? type))
                 (or (not binding) (symbol? binding)))
          (throw-exception
           "Invalid pattern definition; "
           "Wrong argument type (context, type or binding of fragment)."))
        (create-ast
         pattern-language
         'Node
         (list
          context
          type
          binding
          (create-ast-list
           (map
            (lambda (child)
              (apply process-fragment child))
            children))))))
    (define process-reference
      (lambda (name source target)
        (unless (and (symbol? name) (symbol? source) (symbol? target))
          (throw-exception
           "Invalid pattern definition; "
           "Wrong argument type (name, source and target of references must be symbols)."))
        (create-ast pattern-language 'Ref (list name source target))))
    (let ((ast
           (create-ast
            pattern-language
            'Pattern
            (list
             (create-ast-list (map (lambda (frag) (apply process-fragment (cons 'racr-nil frag))) fragments))
             (create-ast-list (map (lambda (ref) (apply process-reference ref)) references))
             #f
             spec))))
      ; Resolve symbolic node references (i.e., perform name analysis):
      (rewrite-terminal 'dnode ast (att-value 'lookup-node ast distinguished-node))
      (for-each
       (lambda (ref)
         (let ((source? (att-value 'lookup-node ast (ast-child 'source ref)))
               (target? (att-value 'lookup-node ast (ast-child 'target ref))))
           (if source?
               (rewrite-terminal 'source ref source?)
               (throw-exception
                "Invalid pattern definition; "
                "Undefined reference source " (ast-child 'source ref) "."))
           (if target?
               (rewrite-terminal 'target ref target?)
               (throw-exception
                "Invalid pattern definition; "
                "Undefined reference target " (ast-child 'target ref) "."))))
       (ast-children (ast-child 'Ref* ast)))
      ; Ensure well-formedness of the pattern (valid distinguished node, reachability, typing, unique node naming):
      (unless (att-value 'well-formed? ast)
        (throw-exception
         "Invalid pattern definition; "
         "The pattern is not well-formed."))
      ; Every thing is fine. Thus, add a respective matching attribute to the given specification:
      (specify-attribute
       spec
       att-name
       (ast-child 'type (ast-child 'dnode ast))
       '*
       #t
       (let ((pmm (att-value 'pmm-code ast))) ; Precompute the PMM => The pattern AST is not in the equation's closure
         (if condition?
             (lambda (n . args)
               (let ((bindings (pmm n)))
                 (if (and bindings (apply condition? bindings args))
                     bindings
                     #f)))
             pmm))
       #f))))

;;; Pattern Matching Machine Instructions:

(define pmmi-load-node ; Make already stored node the new current one.
  (lambda (next-instruction index)
    (lambda (current-node node-memory)
      (next-instruction (vector-ref node-memory index) node-memory))))

(define pmmi-store-node ; Store current node for later reference.
  (lambda (next-instruction index)
    (lambda (current-node node-memory)
      (vector-set! node-memory index current-node)
      (next-instruction current-node node-memory))))
```

```
(define pmmi-ensure-context-by-name ; Ensure, the current node is certain child & make its parent the new current node.
  (lambda (next-instruction context-name)
    (lambda (current-node node-memory)
      (let ((parent? (ast-has-parent? current-node)))
        (if (and parent? (ast-has-child? context-name parent?) (eq? (ast-child context-name parent?) current-node))
            (next-instruction parent? node-memory)
            #f)))))

(define pmmi-ensure-context-by-index ; Ensure, the current node is certain child & make its parent the new current node.
  (lambda (next-instruction index)
    (lambda (current-node node-memory)
      (let ((parent? (ast-has-parent? current-node)))
        (if (and parent? (>= (ast-num-children parent?) index) (eq? (ast-child index parent?) current-node))
            (next-instruction parent? node-memory)
            #f)))))

(define pmmi-ensure-subtype ; Ensure, the current node is of a certain type or a subtype.
  (lambda (next-instruction super-type)
    (lambda (current-node node-memory)
      (if (and
           (not (ast-list-node? current-node))
           (not (ast-bud-node? current-node))
           (ast-subtype? current-node super-type))
          (next-instruction current-node node-memory)
          #f))))

(define pmmi-ensure-list ; Ensure, the current node is a list node.
  (lambda (next-instruction)
    (lambda (current-node node-memory)
      (if (ast-list-node? current-node)
          (next-instruction current-node node-memory)
          #f))))

(define pmmi-ensure-child-by-name ; Ensure, the current node has a certain child & make the child the new current node.
  (lambda (next-instruction context-name)
    (lambda (current-node node-memory)
      (if (ast-has-child? context-name current-node)
          (next-instruction (ast-child context-name current-node) node-memory)
          #f))))

(define pmmi-ensure-child-by-index ; Ensure, the current node has a certain child & make the child the new current node.
  (lambda (next-instruction index)
    (lambda (current-node node-memory)
      (if (>= (ast-num-children current-node) index)
          (next-instruction (ast-child index current-node) node-memory)
          #f))))

(define pmmi-ensure-node ; Ensure, the current node is a certain, already stored node.
  (lambda (next-instruction index)
    (lambda (current-node node-memory)
      (if (eq? current-node (vector-ref node-memory index))
          (next-instruction current-node node-memory)
          #f))))

(define pmmi-traverse-reference ; Evaluate attribute of current node, ensure value is a node & make it the new current one.
  (lambda (next-instruction reference-name)
    (lambda (current-node node-memory)
      (if (and (not (ast-bud-node? current-node)) (ast-node? (att-value reference-name current-node)))
          (next-instruction (att-value reference-name current-node) node-memory)
          #f))))

(define pmmi-terminate ; Construct association list of all binded nodes.
  (lambda (bindings)
    (let ((bindings ; Precompute list of (key, index) pairs => The pattern AST is not in the instruction's closure
           (map
            (lambda (n)
              (cons (ast-child 'binding n) (att-value 'node-memory-index n)))
            bindings)))
      (lambda (current-node node-memory)
        (map
         (lambda (binding)
           (cons (car binding) (vector-ref node-memory (cdr binding))))
         bindings)))))

(define pmmi-initialize ; First instruction of any PMM program. Allocates memory used to store nodes throughout matching.
  (lambda (next-instruction node-memory-size)
    (lambda (current-node)
      (next-instruction current-node (make-vector node-memory-size)))))

;;; Pattern Language:

(when (= (specification->phase pattern-language) 1)
  (with-specification
   pattern-language
```

```scheme
(ast-rule ’Pattern->Node*-Ref*-dnode-spec)
(ast-rule ’Node->context-type-binding-Node*)
(ast-rule ’Ref->name-source-target)
(compile-ast-specifications ’Pattern)

;;; Name Analysis:

(ag-rule ; Given a binding name, find its respective binded node.
 lookup-node
 (Pattern
  (lambda (n name)
    (ast-find-child*
     (lambda (i n)
       (att-value ’local-lookup-node n name))
     (ast-child ’Node* n)))))

(ag-rule
 local-lookup-node
 (Node
  (lambda (n name)
    (if (eq? (ast-child ’binding n) name)
        n
        (ast-find-child*
         (lambda (i n)
           (att-value ’local-lookup-node n name))
         (ast-child ’Node* n))))))

(ag-rule ; Given a non−terminal, find its respective RACR AST rule.
 lookup-type
 (Pattern
  (lambda (n type)
    (specification->find-ast-rule (ast-child ’spec n) type))))

;;; Support API:

(ag-rule ; Root of the AST fragment a node is part of.
 fragment-root
 ((Pattern Node*)
  (lambda (n)
    n)))

(ag-rule ; Is the node a fragment root?
 fragment-root?
 ((Pattern Node*)
  (lambda (n) #t))
 ((Node Node*)
  (lambda (n) #f)))

(ag-rule ; List of all references of the pattern.
 references
 (Pattern
  (lambda (n)
    (ast-children (ast-child ’Ref* n)))))

(ag-rule ; List of all named nodes of the pattern.
 bindings
 (Pattern
  (lambda (n)
    (fold-left
     (lambda (result n)
       (append result (att-value ’bindings n)))
     (list)
     (ast-children (ast-child ’Node* n)))))
 (Node
  (lambda (n)
    (fold-left
     (lambda (result n)
       (append result (att-value ’bindings n)))
     (if (ast-child ’binding n) (list n) (list))
     (ast-children (ast-child ’Node* n))))))

(ag-rule ; Number of pattern nodes of the pattern/the subtree spaned by a node (including the node itself).
 nodes-count
 (Pattern
  (lambda (n)
    (fold-left
     (lambda (result n)
       (+ result (att-value ’nodes-count n)))
     0
     (ast-children (ast-child ’Node* n)))))
 (Node
  (lambda (n)
    (fold-left
     (lambda (result n)
       (+ result (att-value ’nodes-count n)))
     1
```

```
        (ast-children (ast-child 'Node* n))))))

    ;;; Type Analysis:

    (ag-rule ; Must the node be a list?
     must-be-list?
     (Node ; A node must be a list if:
      (lambda (n)
        (or
         (eq? (ast-child 'type n) '*) ; (1) the pattern developer defines so,
         (ast-find-child ; (2) any of its children is referenced by index.
          (lambda (i n)
            (integer? (ast-child 'context n)))
          (ast-child 'Node* n))))))

    (ag-rule ; Must the node not be a list?
     must-not-be-list?
     (Node ; A node must not be a list if:
      (lambda (n)
        (or
         (and ; (1) the pattern developer defines so,
          (ast-child 'type n)
          (not (eq? (ast-child 'type n) '*)))
         (and ; (2) it is child of a list,
          (not (att-value 'fragment-root? n))
          (att-value 'must-be-list? (ast-parent n)))
         (ast-find-child ; (3) any of its children is referenced by name or must be a list.
          (lambda (i n)
            (or
             (symbol? (ast-child 'context n))
             (att-value 'must-be-list? n)))
          (ast-child 'Node* n))))))

    (ag-rule ; List of all types being subject of a Kleene closure, i.e., all list types.
     most-general-list-types
     (Pattern
      (lambda (n)
        (let ((list-types
               (fold-left
                (lambda (result ast-rule)
                  (fold-left
                   (lambda (result symbol)
                     (if (and (symbol->kleene? symbol) (not (memq (symbol->non-terminal? symbol) result)))
                         (cons (symbol->non-terminal? symbol) result)
                         result))
                   result
                   (cdr (ast-rule->production ast-rule))))
                (list)
                (att-value 'most-concrete-types n))))
          (filter
           (lambda (type1)
             (not
              (find
               (lambda (type2)
                 (and
                  (not (eq? type1 type2))
                  (ast-rule-subtype? type1 type2)))
               list-types)))
           list-types)))))

    (ag-rule ; List of all types (of a certain type) no other type inherits from.
     most-concrete-types
     (Pattern
      (case-lambda
        ((n)
         (filter
          (lambda (type)
            (null? (ast-rule-subtypes type)))
          (specification->ast-rules (ast-child 'spec n))))
        ((n type)
         (filter
          (lambda (type)
            (null? (ast-rule-subtypes type)))
          (cons type (ast-rule-subtypes type)))))))

    (ag-rule ; Satisfies a certain type a node's user defined type constraints?
     valid-user-induced-type?
     (Node
      (lambda (n type kleene?)
        (or
         (not (ast-child 'type n))
         (if (eq? (ast-child 'type n) '*)
             kleene?
             (let ((user-induced-type (att-value 'lookup-type n (ast-child 'type n))))
               (and
                user-induced-type
```

```
            (ast-rule-subtype? type user-induced-type))))))))

(ag-rule ; Satisfies a certain type all type constraint of a node and its subtree?
 valid-type?
 (Node
  (lambda (n type kleene?)
    (and
     (not (and (att-value 'must-be-list? n) (not kleene?)))
     (not (and (att-value 'must-not-be-list? n) kleene?))
     (att-value 'valid-user-induced-type? n type kleene?)
     (if kleene?
         (not
          (ast-find-child
           (lambda (i child)
             (not
              (find
               (lambda (child-type)
                 (att-value 'valid-type? child child-type #f))
               (att-value 'most-concrete-types n type))))
           (ast-child 'Node* n)))
         (not
          (ast-find-child
           (lambda (i child)
             (let* ((context? (ast-rule-find-child-context type (ast-child 'context child)))
                    (context-types?
                     (cond
                       ((not (and context? (symbol->non-terminal? context?))) (list))
                       ((symbol->kleene? context?) (list (symbol->non-terminal? context?)))
                       (else (att-value 'most-concrete-types n (symbol->non-terminal? context?))))))
               (not
                (find
                 (lambda (type)
                   (att-value 'valid-type? child type (symbol->kleene? context?)))
                 context-types?))))
           (ast-child 'Node* n))))))))

(ag-rule ; Is the pattern satisfiable (a matching AST exists regarding fragment syntax & type constraints)?
 well-typed?
 ((Pattern Node*)
  (lambda (n)
    (or
     (find
      (lambda (type)
        (att-value 'valid-type? n type #f))
      (att-value 'most-concrete-types n))
     (find
      (lambda (type)
        (att-value 'valid-type? n type #t))
      (att-value 'most-general-list-types n))))))

;;; Reachability:

(ag-rule ; Is the reference connecting two different fragments?
 inter-fragment-reference?
 (Ref
  (lambda (n)
    (not
     (eq?
      (att-value 'fragment-root (ast-child 'source n))
      (att-value 'fragment-root (ast-child 'target n)))))))

(ag-rule ; List of the child contexts to follow to reach the root.
 fragment-root-path

 ((Pattern Node*)
  (lambda (n)
    (list)))

 ((Node Node*)
  (lambda (n)
    (cons (ast-child 'context n) (att-value 'fragment-root-path (ast-parent n))))))

(ag-rule ; List of the cheapest inter fragment references of a fragment and their respective costs.
 inter-fragment-references
 ((Pattern Node*)
  (lambda (n)
    (define walk-costs ; Sum of distances of a reference's source & target to their roots.
      (lambda (ref)
        (+
         (length (att-value 'fragment-root-path (ast-child 'source ref)))
         (length (att-value 'fragment-root-path (ast-child 'target ref))))))
    (reverse
     (fold-left ; Filter for each target the cheapest inter fragment reference:
      (lambda (result ref)
        (if
         (memp
```

```
            (lambda (weighted-ref)
              (eq?
               (att-value 'fragment-root (ast-child 'target ref))
               (att-value 'fragment-root (ast-child 'target (car weighted-ref)))))
            result)
           result
           (cons (cons ref (walk-costs ref)) result)))
        (list)
        (list-sort ; Sort the inter fragment references according to their costs:
         (lambda (ref1 ref2)
           (< (walk-costs ref1) (walk-costs ref2)))
         (filter ; Find all inter fragment references of the fragment:
          (lambda (ref)
            (and
             (eq? (att-value 'fragment-root (ast-child 'source ref)) n)
             (att-value 'inter-fragment-reference? ref)))
          (att-value 'references n))))))))

  (ag-rule ; List of references best suited to reach other fragments from the distinguished node.
   fragment-walk
   (Pattern
    (lambda (n)
      (let ((dummy-walk
             (cons
              (create-ast 'Ref (list #f (ast-child 'dnode n) (ast-child 'dnode n)))
              0)))
        (let loop ((walked ; List of pairs of already followed references and their total costs.
                    (list dummy-walk))
                   (to-visit ; Fragment roots still to visit.
                    (remq
                     (att-value 'fragment-root (ast-child 'dnode n))
                     (ast-children (ast-child 'Node* n)))))
          (let ((next-walk? ; Find the next inter fragment reference to follow if there is any,...
                 (fold-left ; ...i.e., for every already walked inter fragment reference R,...
                  (lambda (best-next-walk performed-walk)
                    (let ((possible-next-walk ; ...find the best walk reaching a new fragment from its target....
                           (find
                            (lambda (weighted-ref)
                              (memq
                               (att-value 'fragment-root (ast-child 'target (car weighted-ref)))
                               to-visit))
                            (att-value 'inter-fragment-references (ast-child 'target (car performed-walk))))))
                      (cond
                        ((not possible-next-walk) ; ...If no new fragment is reachable from the target of R,...
                         best-next-walk) ; ...keep the currently best walk. Otherwise,...
                        ((not best-next-walk) ; ...if no next best walk has been selected yet,...
                         possible-next-walk) ; ...make the found one the best....
                        (else ; Otherwise,...
                         (let ((costs-possible-next-walk (+ (cdr possible-next-walk) (cdr performed-walk))))
                           (if (< costs-possible-next-walk (cdr best-next-walk)) ; ...select the better one.
                               (cons (car possible-next-walk) costs-possible-next-walk)
                               best-next-walk))))))
                  #f
                  walked)))
            (if next-walk? ; If a new fragment can be reached,...
                (loop ; ...try to find another reachable one. Otherwise,...
                 (append walked (list next-walk?))
                 (remq
                  (att-value 'fragment-root (ast-child 'target (car next-walk?)))
                  to-visit))
                (map car (cdr walked))))))))) ; ...return the references defining all reachable fragments.

  ;;; Well-formedness:

  (ag-rule ; Is the pattern specification valid, such that PMM code can be generated?
   well-formed?

   (Pattern
    (lambda (n)
      (and
       (att-value 'local-correct? n)
       (not
        (ast-find-child
         (lambda (i n)
           (not (att-value 'well-formed? n)))
         (ast-child 'Node* n))))))

   (Node
    (lambda (n)
      (and
       (att-value 'local-correct? n)
       (not
        (ast-find-child
         (lambda (i n)
           (not (att-value 'well-formed? n)))
         (ast-child 'Node* n)))))))
```

```
(ag-rule ; Is a certain part of the pattern AST valid?
 local-correct?

 (Pattern
  (lambda (n)
    (and
     (ast-node? (ast-child 'dnode n)) ; A distinguished node must be defined, whose...
     (ast-child 'type (ast-child 'dnode n)) ; ...type is user specified and...
     (not (att-value 'must-be-list? (ast-child 'dnode n))) ; ...not a list.
     (= ; All fragments must be reachable from the distinguished node:
      (+ (length (att-value 'fragment-walk n)) 1)
      (ast-num-children (ast-child 'Node* n)))
     (not ; All fragments must be well typed, i.e., there exists an AST where they match:
      (ast-find-child
       (lambda (i n)
         (not (att-value 'well-typed? n)))
       (ast-child 'Node* n))))))

 (Node
  (lambda (n)
    (and
     (or ; Binded names must be unique:
      (not (ast-child 'binding n))
      (eq? (att-value 'lookup-node n (ast-child 'binding n)) n))
     (let loop ((children (ast-children (ast-child 'Node* n)))) ; Contexts must be unique:
       (cond
         ((null? children) #t)
         ((find
           (lambda (child)
             (eqv? (ast-child 'context (car children)) (ast-child 'context child)))
           (cdr children))
          #f)
         (else (loop (cdr children)))))))))

;;; Code generation:

(ag-rule ; Index within node memory. Used during pattern matching to store and later load matched nodes.
 node-memory-index

 ((Pattern Node*)
  (lambda (n)
    (if (> (ast-child-index n) 1)
        (+
         (att-value 'node-memory-index (ast-sibling (- (ast-child-index n) 1) n))
         (att-value 'nodes-count (ast-sibling (- (ast-child-index n) 1) n)))
        0)))

 ((Node Node*)
  (lambda (n)
    (if (> (ast-child-index n) 1)
        (+
         (att-value 'node-memory-index (ast-sibling (- (ast-child-index n) 1) n))
         (att-value 'nodes-count (ast-sibling (- (ast-child-index n) 1) n)))
        (+ (att-value 'node-memory-index (ast-parent n)) 1)))))

(ag-rule ; Function encoding pattern matching machine (PMM) specialised to match the pattern.
 pmm-code
 (Pattern
  (lambda (n)
    (pmmi-initialize
     (att-value
      'pmm-code:match-fragment
      (ast-child 'dnode n)
      (fold-right
       (lambda (reference result)
         (pmmi-load-node
          (pmmi-traverse-reference
           (att-value 'pmm-code:match-fragment (ast-child 'target reference) result)
           (ast-child 'name reference))
          (att-value 'node-memory-index (ast-child 'source reference))))
       (att-value
        'pmm-code:check-references
        n
        (pmmi-terminate (att-value 'bindings n)))
       (att-value 'fragment-walk n)))
     (+ (att-value 'nodes-count n) 1)))))

(ag-rule ; Function encoding PMM specialised to match the fragment the pattern node is part of.
 pmm-code:match-fragment
 (Node
  (lambda (n continuation-code)
    (fold-right
     (lambda (context result)
       (if (integer? context)
           (pmmi-ensure-context-by-index result context)
```

```
            (pmmi-ensure-context-by-name result context)))
       (att-value 'pmm-code:match-subtree (att-value 'fragment-root n) continuation-code)
       (att-value 'fragment-root-path n)))))

  (ag-rule ; Function encoding PMM specialised to match the subtree the pattern node spans.
   pmm-code:match-subtree
   (Node
    (lambda (n continuation-code)
      (let ((store-instruction
            (pmmi-store-node
             (fold-right
              (lambda (child result)
                (pmmi-load-node
                 (if (integer? (ast-child 'context child))
                     (pmmi-ensure-child-by-index
                      (att-value 'pmm-code:match-subtree child result)
                      (ast-child 'context child))
                     (pmmi-ensure-child-by-name
                      (att-value 'pmm-code:match-subtree child result)
                      (ast-child 'context child)))
                 (att-value 'node-memory-index n)))
              continuation-code
              (ast-children (ast-child 'Node* n)))
             (att-value 'node-memory-index n))))
        (cond
          ((att-value 'must-be-list? n)
           (pmmi-ensure-list store-instruction))
          ((ast-child 'type n)
           (pmmi-ensure-subtype store-instruction (ast-child 'type n)))
          (else store-instruction))))))

  (ag-rule ; Function encoding PMM specialised to match the reference integrity of the pattern.
   pmm-code:check-references
   (Pattern
    (lambda (n continuation-code)
      (fold-left
       (lambda (result reference)
         (pmmi-load-node
          (pmmi-traverse-reference
           (pmmi-ensure-node
            result
            (att-value 'node-memory-index (ast-child 'target reference)))
           (ast-child 'name reference))
          (att-value 'node-memory-index (ast-child 'source reference))))
       continuation-code
       (filter
        (lambda (reference)
          (not (memq reference (att-value 'fragment-walk n))))
        (ast-children (ast-child 'Ref* n)))))))

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

C. MIT License

API Index