Friedrich Wilhelm Schröer

The GENTLE Compiler Construction System

Friedrich Wilhelm Schröer
The GENTLE Compiler Construction System
Metarga, Berlin, 2005

Metarga GmbH
Joachim-Friedrich-Str. 54
D-10711 Berlin
info@metarga.com
www.metarga.com

Friedrich Wilhelm Schröer
Fraunhofer Institute for
Computer Architecture and Software Technology
Kekuléstr. 7
D-12489 Berlin
f.w.schroeer@first.fraunhofer.de

Preface (2005)

Fifteen Years in Industry

The *GENTLE Compiler Construction System*, originally designed in 1989 at the German National Research Center for Information Technolgy, is now in industrial use for fifteen years: The very first system was sold to Nixdorf Computers, who used it to implement their *Combined Object-Oriented Language*, a *C++* alternative for large mission-critical projects, e.g. in the chemical industry. The first *Gentle* system in 2005 was sold to Ritlabs, creators of *The Bat*, a leading email client.

Here are some more examples: Robot control software generated with Gentle is in use at the automotive industry (e.g. at Volkswagen); in a project conducted by Secunet, *Gentle* was used to create a compiler for the *Security Policy Specification Language* of the IETF; Siemens licensed *Gentle* for telecommunication applications, and Fraunhofer writes their compiler for the ETSI *Testing and Test Control Notation* in *Gentle*; the International Institute for Software Technology of the United Nations University is using *Gentle* in education and various projects to support *RAISE*, a *Rigorous Approach to Industrial Software Engineering*.

Distributions

The *GENTLE Compiler Construction System* is available in two editions:

- *GENTLE 97* was published in 1997 together with the first edition of this manual (Oldenbourg Verlag, Munich and Vienna, 1997)
- *GENTLE 21* is distributed since 2001 by Metarga and is continuously maintained according to the requirements of its users

This manual covers the common functionality of both editions.

However, *GENTLE 21* provides additional features such as a powerful parsing strategy that overcomes the limitations of *Yacc*, iteration statements that simplify the traversal of data structures, a compact notation to embed target text into code generation rules, and more. These extensions are described in a companion manual.

Friedrich Wilhelm Schröer, January 2005

Preface (1997)

This book presents *Gentle*, an integrated system for compiler writers.

Gentle supports the description of compilers at a very high level and relieves users from the need to deal with implementation details. It has been used in large industrial projects and for constructing various commercial products. Users report that *Gentle* significantly increases productivity.

Uniform Framework and Strong Methodology

Gentle provides a uniform framework for specifying the components of a compiler.

The *Gentle* language was designed around a specific paradigm: *recursive definition and structural induction*. Input and internal data structures are defined by listing alternative ways to construct items from given constituents. Then the properties of these items are described by giving rules for the possible alternatives. These rules recursively follow the structure of items by processing their constituents. Experience has shown that this is the underlying paradigm of virtually all translation tasks.

The same concepts apply to analysis, transformation, and synthesis. They can be used to describe the backend of a compiler as a simple unparsing scheme such as sufficies for most source-to-source translations. They can also be used to specify a cost-augmented mapping to a low-level language which is used for optimal rule selection.

The rule-based approach follows the *principle of locality*: complex interactions are avoided and a system can be understood by understanding small pieces in isolation. Individual constructs are described independently.

The paradigm leads to a *data-oriented methodology*: the structure of data is mirrored by the structure of algorithms. This methodology proves to be a useful guideline in compiler projects.

Safety, Portability, Efficiency, and Openness

Gentle has been designed to enabale errors to be detected as early as possible. Although variables need not be declared, *Gentle* is a strongly typed language that allows the compiler to check their consistent use. Moreover, it can be ensured statically that each variable has a value when it is accessed. In cases where static rule analysis cannot prove that a program is well-behaved, this is done at run time. The violation of a language constraint is reported when it occurs, not when it has disastrous consequences. This drastically reduces debugging effort.

A specification written in *Gentle* can be translated automatically into highly portable *C* code. In many cases, the generated code is faster than hand written code.

Moreover, *Gentle* is an open language: compilers can be written partly in *Gentle* and partly in *C*.

Organisation of this Book

This book provides a tutorial, reference manuals, and a case study:

The *Gentle Primer* gives an introduction to the *Gentle* compiler description language. After presenting the basic concepts using simple examples (*1.1*), , these concepts are discussed in more detail in the following sections (*1.2-1.5*). Then more advanced features of *Gentle* are introduced (*1.6-1.14*).

Getting Started explains how to install and use *Gentle*. It describes the steps required to generate a compiler from a given specification. The *Language Reference Manual* gives a concise specification of the *Gentle* compiler description language. The *Reflex Reference Manual* describes a simple tool that facilitates the construction of lexer specifications. The *Library Reference Manual* describes some utility modules that come with the *Gentle* system.

The *Case Study* discusses a compiler for a small programming language.

To get an initial impression, read Section *1.1* and take a look at the *Case Study*. To start with your own first compiler, read Sections *1.2-1.5* and refer to *Getting Started*.

Acknowledgments

Gentle is a descendant of Kees Koster's *CDL* compiler description language [7]. *CDL* was designed in 1969 and used in a portable *COBOL-74* compiler of MPB, in the *mprolog* system of SzKI, and to program the *Mephisto* chess computer.

Gentle [10] was designed in 1989 at the GMD Karlsruhe Lab, the group that also created the Karlsruhe *Ada* and *Modula* compilers [12, 11], the *BEG* backend generator [3], and the *Cocktail* toolbox [4].

I wish to thank the following people for fruitful discussions and valuable comments: *Phil Bacon*, *Helmut Emmelmann*, *Thomas Fries*, *Gerhard Goos*, *Joseph Grosch*, *Ulrich Grude*, *Stefan Jähnichen*, *Kees Koster*, *Rudolf Landwehr*, *Birgit Schwarz*, *William M. Waite*, and *Dwight VandenBerghe*.

Friedrich Wilhelm Schröer, November 1997

Contents

1 GENTLE PRIMER

The *Gentle Compiler Construction System* generates efficient compilers from high-level specifications. The *Gentle Compiler Description Language* provides a simple and uniform notation for such specifications.

1.1 At a Glance

This section gives some simple but complete examples of *Gentle* specifications. The intend is not to discuss details, but to provide the reader with a feeling for the general framework. Subsequent sections will describe the specific concepts and conventions.

We start with a simple parser for arithmetic expressions, which we will then extend to a calculator. While the calculator is written along the concrete syntax of expressions, a compiler is often based on abstract syntax. We show how abstract syntax can be defined in *Gentle*, and how a concrete source text can be mapped onto a corresponding abstract representation. Two examples then show how to process abstract syntax: a refined version of the calculator, and a program that computes the derivative of an expression. We conclude with a simple code generator.

1.1.1 A Parser for Expressions

This section presents a parser for expressions such as `10+20*30`.

```
'root' expression

'nonterm' expression

   'rule' expression: expr2
   'rule' expression: expression "+" expr2
   'rule' expression: expression "-" expr2

'nonterm' expr2

   'rule' expr2: expr3
   'rule' expr2: expr2 "*" expr3
   'rule' expr2: expr2 "/" expr3

'nonterm' expr3

   'rule' expr3: Number(-> N)
   'rule' expr3: "-" expr3
   'rule' expr3: "+" expr3
   'rule' expr3: "(" expression ")"
```

```
'token' Number(-> INT)
```

In *Gentle* a parser is written as a grammar for the language to be processed. Such a grammar consists of a set of nonterminal symbols that represent phrases, and terminal symbols (tokens) that stand for themselves.

The nonterminal symbols that represent phrases are defined by rules listing the constituents of the phrase.

```
'nonterm' expression
```

introduces the nonterminal `expression`. The rule

```
'rule' expression: expression "+" expr2
```

specifies that a phrase of the class `expression` may be given by a (simpler) phrase of the class `expression`, followed by a plus symbol, followed by a phrase of the class `expr2` (expressions involving operators with a higher priority than plus and minus). For example, if `10` is an `expression` and if `20*30` is an `expr2`, then `10+20*30` is an `expression`.

Terminal symbols appear in quotes (e.g `"+"`) or are introduced by token declarations.

```
'token' Number(-> INT)
```

introduces a token `Number`, the representation of which is not given in this specification. It is defined elsewhere as a nonempty sequence of decimal digits. The actual value of the symbol is of type `INT` (we ignore parameters in this section).

The line

```
'root' expression
```

states that this specification is elaborated by invoking the expression parser.

As it stands, this program is not of much use: it reads and analyzes its input, and if the input is a valid expression, it does nothing. If the input is not an expression according to the given grammar, the program will flag the first token that does not allow us to complete the text read so far as a valid expression.

Note that the phrase structure imposed on the input reflects the binding strengths of the operators. E.g. `10+20*30` is parsed as the sum of `10` and the product of `20` and `30`. This is not necessary if the task is just to reject invalid expressions, but it is helpful in the next example.

1.1.2 A Calculator

In this section, we turn the parser into a simple calculator. It will read its input (which must be an expression as defined by the above grammar) and print the value of the expression.

Our starting point is the parser given above. Each nonterminal gets an output parameter that specifies its value.

```
'root' expression(-> X) print(X)

'nonterm' expression(-> INT)

   'rule' expression(-> X): expr2(-> X)
   'rule' expression(-> X+Y): expression(-> X) "+" expr2(-> Y)
   'rule' expression(-> X-Y): expression(-> X) "-" expr2(-> Y)

'nonterm' expr2(-> INT)

   'rule' expr2(-> X): expr3(-> X)
   'rule' expr2(-> X*Y): expr2(-> X) "*" expr3(-> Y)
   'rule' expr2(-> X/Y): expr2(-> X) "/" expr3(-> Y)

'nonterm' expr3(-> INT)

   'rule' expr3(-> X): Number(-> X)
   'rule' expr3(-> - X): "-" expr3(-> X)
   'rule' expr3(-> + X): "+" expr3(-> X)
   'rule' expr3(-> X): "(" expression(-> X) ")"

'token' Number(-> INT)
```

Nonterminals and tokens can have output parameters.

```
'nonterm' expression(-> INT)
```

introduces the nonterminal `expression` with one output parameter of type `INT`.

The rule

```
'rule' expression(-> X+Y): expression(-> X) "+" expr2(-> Y)
```

specifies how the value of an `expression` is computed from the values of its constituents.

`expression(-> X)` and `expr2(-> Y)` on the right hand side define the variables `X` and `Y` as the values of the `expression` and the `expr2` constituents of the parsed phrase. For example, if the phrase is `10+20*30`, then `X` is defined as 10 and `Y` is defined as 600.

`expression(-> X + Y)` on the left hand side defines the value of the phrase parsed according to this rule as the sum of `X` and `Y`, i.e. as 610 in the example given.

```
'root' expression(-> X) print(X)
```

defines the following elaboration: Parse the input as an `expression`. This yields a value `X`. Print `X`.

1.1.3 Abstract Syntax

We presented above a concrete syntax for expressions. In this section, we discuss a so-called *abstract syntax*.

Abstract syntax is often much better suited as the basis for the specification of program transformations, and is used as an intermediate program representation in compilers.

Similar to concrete syntax, a class of phrases (here called *terms*) are defined by listing the alternatives (corresponding to the rules) and specifying the constituents of each alternative. Each alternative has a name (called a *functor*).

```
'type' Expr

   plus(Expr, Expr)
   minus(Expr, Expr)
   mult(Expr, Expr)
   div(Expr, Expr)
   neg(Expr)
   num(INT)
```

This introduces the type `Expr`. Values of this type are constructed as specified by the given alternatives.

For example, the alternative

```
num(INT)
```

specifies that the functor `num` used with an argument of type `INT` represents a value of type `Expr`. For example, `num(20)` and `num(30)` are values of type `Expr`.

The alternative

```
mult(Expr, Expr)
```

specifies that the functor `mult` may be used with two arguments of type `Expr`. E.g. since `num(20)` and `num(30)` are values of type `Expr` the term `mult(num(20), num(30))` is also a value of type `Expr`.

Similarly the alternative

```
plus(Expr, Expr)
```

can be used to construct the value `plus(num(10), mult(num(20), num(30)))`. This abstract syntax term can be used as a representation of the expression `10+20*30`.

Note that in abstract syntax the structure of terms is uniquely determined. Thus, we do not need to introduce several types to reflect the binding strength of operators. `plus(num(10), mult(num(20), num(30)))` is used as a representation of `10+(20*30)` and also as a representation of the expression `10+20*30`.

1.1.4 Translating Concrete Syntax into Abstract Syntax

We now show how concrete syntax can be translated into abstract syntax. This can be done in exactly the same way as with the calculator. We use the expression grammar where each nonterminal gets an output parameter that specifies the corresponding abstract syntax.

```
'root' expression(-> X) print(X)

'nonterm' expression(-> Expr)

   'rule' expression(-> X): expr2(-> X)
   'rule' expression(-> plus(X,Y)): expression(-> X) "+" expr2(-> Y)
   'rule' expression(-> minus(X,Y)): expression(-> X) "-" expr2(-> Y)

'nonterm' expr2(-> Expr)

   'rule' expr2(-> X): expr3(-> X)
   'rule' expr2(-> mult(X,Y)): expr2(-> X) "*" expr3(-> Y)
   'rule' expr2(-> div(X,Y)): expr2(-> X) "/" expr3(-> Y)

'nonterm' expr3(-> Expr)

   'rule' expr3(-> num(X)): Number(-> X)
   'rule' expr3(-> neg(X)): "-" expr3(-> X)
   'rule' expr3(-> X): "+" expr3(-> X)
   'rule' expr3(-> X): "(" expression(-> X) ")"

'token' Number(-> INT)
```

Consider the rule

```
'rule' expression(-> plus(X, Y)): expression(-> X) "+" expr(-> Y)
```

that corresponds to the rule

```
'rule' expression(-> X + Y): expression(-> X) "+" expr(-> Y)
```

of the calculator.

If `X` is the abstract syntax of the `expression` appearing on the right hand side and `Y` is the abstract syntax of the `expr2` phrase then `plus(X,Y)` is the abstract syntax of the whole phrase.

For example, if `X` has the value `num(10)` and `Y` has the value `mult(num(20),num(20))`, then `plus(num(10)), mult(num(20), num(30))` is defined as the abstract syntax of the whole phrase parsed by the rule.

```
'root' expression(-> X) print(X)
```

causes the abstract syntax to be printed.

1.1.5 Traversal

In this section we show how to process abstract syntax. For this purpose we discuss a calculator which is based on abstract syntax.

The root clause of this specification has the form

```
'root' expression(-> X) eval(X -> N) print(N)
```

where `expression` is defined as in the preceding example. It yields a term `X` of type `Expr`, as defined above.

`eval` is a procedure with one input parameter of type `Expr` and one output parameter of type `INT`. It maps its input value (the abstract syntax of a particular expression) onto the numeric value of this expression.

This mapping is defined by the following rules.

```
'action' eval (Expr -> INT)

'rule' eval(plus(X1, X2) -> N1+N2): eval(X1 -> N1) eval(X2 -> N2)
'rule' eval(minus(X1, X2) -> N1-N2): eval(X1 -> N1) eval(X2 -> N2)
'rule' eval(mult(X1, X2) -> N1*N2): eval(X1 -> N1) eval(X2 -> N2)
'rule' eval(div(X1, X2) -> N1 / N2): eval(X1 -> N1) eval(X2 -> N2)
'rule' eval(neg(X) -> -N): eval(X -> N)
'rule' eval(num(N) -> N)
```

Whereas in concrete syntax rules are selected according to the structure of the concrete input, here rules are selected according to the structure of the terms of the abstract syntax. In this example, there is a rule for each alternative of the type `Expr` that specifies how to compute the numerical value from the numerical value of the constituents of the term.

Consider the rule

```
'rule' eval(plus(X1, X2) -> N1+N2): eval(X1 -> N1) eval(X2 -> N2)
```

which is selected if the input term of `eval` has the structure `plus(`S_1`,`S_2`)` with subterms S_1 and S_2. The variables `X1` and `X2` are defined as these subterms. The invocations `eval(X1 -> N1)` and `eval(X2 -> N2)` define `N1` and `N2` as the numeric values of the corresponding subterms. These values are then used to compute the output parameter `N1 + N2`.

For example, if the input term is `plus(num(10), mult(num(20), num(30))`, then `X1` is defined as `num(10)` and `X2` is defined as `mult(num(20), num(30))`. The recursive invocations of `eval` then define `N1` as 10 (the numeric value of `X1`) and `N2` as 600 (the numeric value of `X2`). With these values the output parameter of the left-hand side is computed, `N1+N2` yields 610.

1.1.6 Transformation

We now look at a more interesting algorithm that works on the abstract syntax. We introduce the placeholder `x` and interpret an expression as a function in `x`. Given such an expression E, we will compute an expression E' that represents the derivative of the function.

We extend the abstract syntax by the alternative

```
x
```

and add to the concrete syntax the rule

```
'rule' expr3(-> x): "x"
```

The root clause now takes the form

```
'root' expression(-> X) deriv(X -> D) print(D)
```

The procedure `deriv` is then defined as follows

```
'action' deriv (Expr -> Expr)

'rule' deriv(mult (U,V) -> plus(mult(Ud,V), mult(U,Vd))):
   deriv(U -> Ud)
   deriv(V -> Vd)
'rule' deriv(div(U,V) -> div(minus(mult(Ud,V),mult(U,Vd)),mult(V,V))):
   deriv(U -> Ud)
   deriv(V -> Vd)
```

```
'rule' deriv(plus(U,V) -> plus(Ud, Vd)):
   deriv(U -> Ud)
   deriv(V -> Vd)
'rule' deriv(minus(U,V) -> minus(Ud, Vd)):
   deriv(U -> Ud)
   deriv(V -> Vd)
'rule' deriv(num(N) -> num(0))
'rule' deriv(x -> num(1))
```

1.1.7 Code Generation

While the parser of a compiler translates a concrete program (of the source language) into a (high-level) abstract syntax (which may then be transformed into a lower-level intermediate representation), the code generator of a compiler translates the abstract syntax into a concrete program (of the target language). This mapping can be specified in exactly the same way as within the parser with the exception that abstract syntax now becomes an input parameter that controls rule selection.

As an example, we translate expressions into code for a stack computer: here, the code for operands is followed by an operator that works on these operands.

The root clause of this compiler is

```
'root' expression(-> X) code(X)
```

The procedure `code` is defined as follows

```
'action' code (Expr)

'rule' code(plus(X1, X2)): code(X1) code(X2) print("plus")
'rule' code(minus(X1, X2)): code(X1) code(X2) print("minus")
'rule' code(mult(X1, X2)): code(X1) code(X2) print("mult")
'rule' code(div(X1, X2)): code(X1) code(X2) print("div")
'rule' code(neg(X)): code(X) print("neg")
'rule' code(num(N)): print(N)
```

1.2 Elements of Specifications

A *Gentle* specification is a list of declarations.

A declaration may introduce a *data type* as in

```
   'type' Expr

      plus(Expr, Expr)
```

```
minus(Expr, Expr)
mult(Expr, Expr)
div(Expr, Expr)
neg(Expr)
num(INT)
```

a *predicate* as in

```
'action' eval (Expr -> INT)

    'rule' eval(plus(X1, X2) -> N1+N2): eval(X1 -> N1) eval(X2 -> N2)
    'rule' eval(minus(X1, X2) -> N1-N2): eval(X1 -> N1) eval(X2 -> N2)
    'rule' eval(mult(X1, X2) -> N1*N2): eval(X1 -> N1) eval(X2 -> N2)
    'rule' eval(div(X1, X2) -> N1 / N2): eval(X1 -> N1) eval(X2 -> N2)
    'rule' eval(neg(X) -> -N): eval(X -> N)
    'rule' eval(num(N) -> N)
```

a *global variable* as in

```
'var' CurLoopContext: LoopContext
```

or a *global table* as in

```
'table' Label (Address: INT)
```

Declarations may be given in any order.

The Root Definition

One of the declarations must be a *root definition*. It consists of the the keyword `'root'` followed by one or more statements. The program generated from the specification elaborates these statements.

Consider

```
'root' expression(-> X) code(X)
```

Here, `expression` is elaborated first yielding a value `X`, then `code` is elaborated with `X` as an argument. `expression` and `code` must be declared as predicates.

Here is a complete *Gentle* specification

```
'root' print("Hello World!")
```

using the predefined predicate `print`.

Comments

Comments may be interspersed into the specification. They allow documentation to be included for a human reader and are ignored by the compiler. There are two forms of comments in *Gentle*.

Comments of the first form start with "`--`" and finish at the end of the line. For example,

```
-- this comment extends to the end of the line
```

The second form is text enclosed in "`/*`" and "`*/`", these comments can range over several lines. E.g.

```
/* this comment
   ranges over two lines */
```

Comments of the second form may be nested. This is of particular value because it allows us to "comment out" a text without bothering whether the text contains comments.

1.3 Describing Data

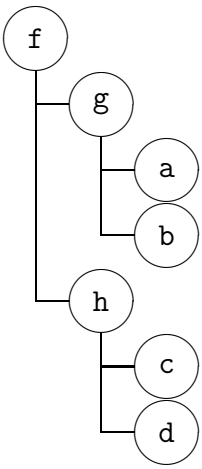
1.3.1 Terms

The most important kind of data a *Gentle* program operates on are so-called *terms*. A term may be a simple constant, as e.g. `red` or `yellow`, or it may be a compound structure, as e.g. `signal(red, yellow)`. Here, the term is obtained by prefixing a so-called *functor*, `signal`, to two arguments: `red` and `yellow`. The simple constants are merely functors with no arguments. Hence, terms are constructed by applying functors to simpler terms. This can be done to an arbitrary depth of nesting.

A term may be viewed as a linear notation of a tree. For example, the term

```
f( g(a,b), h(c,d) )
```

may be represented graphically as a tree



1.3.2 Type Definitions

A set of term values is introduced by a type definition. For example, the set of terms `red`, `yellow`, and `blue` can be defined as a type `Color` by the type definition

```
'type' Color
   red
   yellow
   blue
```

A type definition gives the name of the type and lists the possible alternatives.

If an alternative is given by a functor with arguments, then the types of the arguments are listed. For example, a type `Signal` is defined by

```
'type' Signal
   signal(Color, Color)
```

A possible value of this type is `signal(red, yellow)`.

Type definitions may be recursive. E. g. a type `ColorList` can be defined by

```
'type' ColorList
   list(Color, ColorList)
   nil
```

Type definitions specify how to construct valid terms of a type: a value of a type may be obtained by replacing the type names in an alternative for that type by valid terms of the corresponding types. For example, since `red` is a `Color` and `nil` is a `ColorList`, one can choose the alternative `list(Color, ColorList)` and replace `Color` by `red` and `ColorList` by `nil` to obtain `list(red, nil)` as valid term of type `ColorList`. Using `yellow` as `Color` and the `ColorList` just constructed, one can build the more complex term `list(yellow, list(red, nil))` of type `ColorList`.

In a type definition the fields may be given names to document their meaning, e. g.

```
   list(Head: Color, Tail: ColorList)
```

The terms considered so far are so-called *ground terms*: they do not contain variables as placeholders. A ground term is built from a simple functor without arguments or by taking already available ground terms and applying a functor to them, thus yielding a more complex term.

1.3.3 Using Terms in Expressions and Patterns

When a *Gentle* program is executed, the actual values of term types are always ground terms. But in the text of a program we also use terms that contain variables as subterms (i.e. as arguments of a functor). In the extreme case the whole term may be just one variable name. A variable serves as a placeholder.

To avoid confusion, variables start with an upper-case letter, and functors start with a lower case letter.

Variables need not be declared. Nevertheless, they are strongly typed. The type is derived by the *Gentle* compiler, and the use of the variables is checked for consistency.

Depending on the context in which they are used, terms appear in two roles: as *expressions* and as *patterns*.

An expression is used to construct a term from given components.

If the variable `Hd` is of type `Color` and holds the value `red`, and if the variable `Tl` is of type `ColorList` and holds the value `nil`, the expression `list(Hd, Tl)` yields the value `list(red, nil)`.

A value may be decomposed into constituents by matching the value against a pattern.

If the value `list(red, nil)` is matched against the pattern `list(Hd, Tl)` the variable `Hd` is defined as `red`, and the variable `Tl` as `nil`. The matching succeeds.

If the value `nil` is matched against the pattern `list(Hd, Tl)`, the matching is said to fail.

Matching a pattern against a term means trying to find values for the variables inside the pattern: replacing the variables by these values makes the pattern equal to the term. For example, replacing `Hd` by `red` and `Tl` by `nil` turns the pattern `list(Hd, Tl)` into the term `list(red, nil)`.

1.3.4 Numbers and Strings

In compilers written in *Gentle* most computations are carried out on terms. In addition there are built-in types for numbers and strings.

The type `INT` comprises integer numbers.

They are denoted by sequences of decimal digits. Expressions of type `INT` may be constructed from integer constants, variables of type `INT`, or by applying an operator to subexpressions. These operators are monadic plus (`+`), monadic minus (`-`), addition (`+`), subtraction (`-`), multiplication (`-`), and division (`/`). The operators have the usual binding strength (multiplication and division have a higher priority than addition and subtraction, the monadic operators having highest priority). Parenthesis may be used to explicitly indicate grouping.

Here are some examples of `INT` expressions:

```
1024
```

```
N+1
A-(B-C)
-K
```

The type `STRING` comprises sequences of characters.

Expressions of type `STRING` are string constants or variables of type `STRING`. A string constant is a character sequence enclosed in apostrophes "`"`". Inside string constants certain characters must be represented by an escape sequence: "`\"`" represents "`"`", "`\\`" represents "`\`", "`\n`" represents a newline character, and "`\t`" represents a tabulator.

Here are some examples of `STRING` expressions:

```
"Hello World!"
"The magic word is \"needed\"."
```

1.4 Describing Computations

1.4.1 Invocations and Rules

A *predicate* is used to specify a computation. Such a computation is triggered by a *predicate invocation*. For example, if we have defined a predicate `favorite` that determines for a given `Person` the favorite `Color`, then we can write

```
favorite(P -> C)
```

to ask for the favorite color `C` of `P`.

A predicate is described by *rules*. For example, the predicate `favorite` could have been defined by the following rules:

```
'rule' favorite (jim -> red)
'rule' favorite (julia -> blue)
'rule' favorite (jane -> red)
'rule' favorite (john -> yellow)
```

An item like `favorite (jim -> red)` here is called a *rule heading* (rule bodies, here empty, are discussed later).

An invocation

```
favorite (julia -> C)
```

would define the variable `C` as `blue` because the second rule defines the output `blue` for the input `julia`.

A predicate may have input and output parameters. The input parameters come first, the output parameters being separated by an arrow. Here is a predicate with no output parameters:

```
IsJimsColor(Val)
```

with rule

```
'rule' IsJimsColor(red)
```

and one with no input parameters:

```
JimsColor(-> Val)
```

with rule

```
'rule' JimsColor(-> red)
```

If a list of input or output parameters is supplied, a comma is used to separate the members.

1.4.2 Failure

The invocation of a predicate can *fail* for two reasons: the predicate is not applicable for the given input values, or the returned output values do not match the corresponding values in the invocation.

Consider

```
IsJimsColor(yellow)
```

This invocation will fail because `IsJimsColor` is only applicable for `red`. An invocation fails if there is no applicable rule.

Consider also

```
JimsColor(-> yellow)
```

This invocation will fail since `JimsColor` returns `red` which does not match `yellow`.

Failure does not mean error. As we shall see, it can be used to control predicate evaluation.

1.4.3 Expressions and Patterns

We have made a distinction between expressions and patterns, and have said that the role of a term depends on its context.

The input parameters of a predicate invocation and the output parameters of rule headings are expressions. Here, terms are constructed from given components. Variables inside the expressions must hold these components.

The output parameters of a predicate invocation and the input parameters of rule headings are patterns. Here, a given value is matched against the pattern. Variables inside the pattern are defined.

To illustrate this, we discuss a predicate `swap` that, given a term `signal(X, Y)`, delivers the term `signal(Y, X)`. It is defined by the rule

```
'rule' swap (signal(X, Y) -> signal(Y, X))
```

Assume that the variable `A` holds the value `red` and consider the invocation

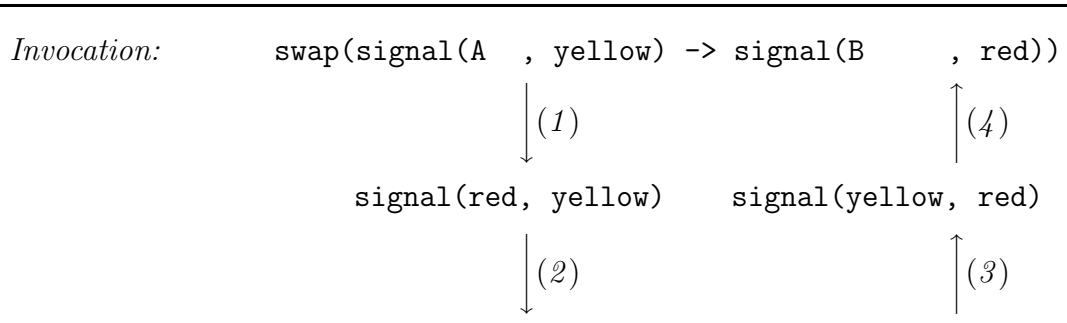
```
swap (signal(A, yellow) -> signal(B, red))
```

First, the expression `signal(A, yellow)` (the input parameter of the invocation) is evaluated. Since `A` holds `red`, substitution yields the value `signal(red, yellow)`.

Then, rules defining the predicate are inspected; here there is only one. The input value `signal(red, yellow)` is matched against the pattern `signal(X, Y)` (the input parameter of the rule head). The matching succeeds, thereby defining the variable `X` as `red` and `Y` as `yellow`.

Now the expression `signal(Y, X)` (the output parameter of the rule head) is evaluated. Since `Y` is `yellow` and `X` is `red`, this yields `signal(yellow, red)`, which is returned by the predicate.

Finally, the returned value `signal(yellow, red)` is matched against the pattern `signal(B, red)` (the output parameter of the invocation). This succeeds and defines the variable `B` as `yellow` (see *Fig. 1.1*).



(1) Construction of term from input expression
(2) Matching against input pattern
(3) Construction of term from output expression
(4) Matching against output pattern

Fig. 1.1 Construction of Terms and Pattern Matching

1.4.4 Rule Bodies

So far we have discussed rules without bodies. A *rule body* is a sequence of statements, in most cases predicate invocations, that can be used to break down the computation of the particular rule into subtasks or to reduce a problem to another problem for which there is a solution.

For example, we could add a rule to the definition of predicate `favorite` stating that `jack` prefers the same color as `jane`:

```
'rule' favorite(jack -> X): favorite(jane -> X)
```

Consider the invocation

```
favorite(jack -> C)
```

The new rule is selected (because the expression `jack` matches the pattern `jack`), and its body is evaluated. This means that the invocation `favorite(jane -> X)` is elaborated. This succeeds and defines `X` as `red`. After processing the body the expressions constituting the output parameters of the rule head are evaluated. Here the expression is simply a variable, `X`, which has just been defined as `red`. Hence, `red` is the output value of the original invocation. So `C` is defined as `red`.

It is very common for the rules of a predicate to directly follow the type definition of its argument. There is a rule for each alternative of the type, and inside this rule the members of the body process the constituents of that alternative. The result is given by combining the results of the members.

Assume that, besides lists of colors, we have also defined lists of persons:

```
'type' PersonList
   list(Person, PersonList)
   nil
```

(We can use the same functors in both list type definitions because it is clear from the context which type is meant.)

We now want to define a predicate `favorites` that takes a list of persons and returns a list of colors of the same length. An element of the color list is the favorite color of the person at the corresponding position in the person list. For example,

```
list(julia, list(jane, nil))
```

is mapped to

```
list(blue, list(red, nil))
```

We introduce two rules, each one handling an alternative of the above type definition:

```
'rule' favorites(list(Head, Tail) -> list(ColorHead, ColorTail)):
   favorite(Head -> ColorHead)
   favorites(Tail -> ColorTail)

'rule' favorites(nil -> nil)
```

The first rule decomposes its argument into `Head` and `Tail` and uses the suitable predicates to compute the favorite color of the `Head` and the favorite color list of the `Tail`. Given these values the resulting color list can be constructed.

The second rule simply has an empty body. For an empty person list we return an empty color list.

A rule body is evaluated by evaluating its members in the given order. If a member fails then the rule fails. If all members succeed, then the rule succeeds.

A rule

'rule' A : B_1 B_2 ... B_n

can be understood as a logical statement

A **if** B_1 **and** B_2 **and** ... **and** B_n

For example, the rule

```
'rule' grandfather(X -> Z): father(X -> Y) father(Y -> Z)
```

can be read as

the grandfather of `X` is `Z` **if**
(there is a `Y` such that)
the father of `X` is `Y` **and**
the father of `Y` is `Z`.

1.4.5 Variables

Variables defined in a rule are local to that rule. Here are the constraints that must hold for the data flow inside a rule:

Each variable appearing inside a rule must be defined exactly once (i.e. occur in a pattern). A variable that appears in an expression of a member of the rule body (i.e. that is used as part of an input parameter of that member) must be defined in the rule heading (i.e. inside an input parameter) or in a preceding member of the rule body (i.e. inside an output parameter of a preceding member).

These laws make it possible for the disastrous error of uninitialized variables to be totally excluded by compile-time analysis.

Assume that we want to express that two persons like each other if they prefer the same color. Then we can write

```
'rule' likes(A, B):
   favorite(A -> AsFavorite)
   favorite(B -> BsFavorite)
   eq(AsFavorite, BsFavorite)
```

using a predefined predicate `eq` that succeeds if invoked with equal arguments and fails if invoked with unequal arguments.

The first and second members compute the favorite color for `A` and `B`. Now as the values of `AsFavorite` and `BsFavorite` are known, they can be used as input to `eq`.

The first two members always succeed. The third member succeeds if `AsFavorite` and `BsFavorite` are equal, and fails otherwise. If the third member succeeds, then the rule succeeds, otherwise the rule fails. Since this is the only rule, an invocation `likes(X, Y)` succeeds if `X` and `Y` prefer the same color.

The example also shows that a variable cannot be defined twice. Using a common variable (say `X`) instead of `AsFavorite` and `BsFavorite` to express that these values must be equal, would violate the constraint because `X` would occur in two patterns. Instead, the equality must be expressed explicitly using `eq`. (This is to catch typing errors and to rule out the misunderstanding that the second definition overwrites the old value.)

1.4.6 Evaluation Strategy

In the examples we have discussed in this chapter one rule at most was applicable for a given input. This is typical for many predicates because they are often written in a way that follows the type definition of their arguments.

There may also be definitions where more than one rule is applicable for a given input. For example, the definition of `favorite` could contain

```
'rule' favorite (julia -> blue)
'rule' favorite (julia -> red)
```

Several interpretations would be possible:

Shallow Backtracking. The first applicable rule is selected and it determines the result. For example, in

```
favorite(julia -> X)
```

`X` is defined as `blue`. This approach treats predicates as functions.

Deep Backtracking. All results are delivered, one after the other. For example, in

```
favorite(julia -> X) IsJimsColor(X)
```

`X` is first defined as `blue`, but `IsJimsColor` fails for this value, so `favorite` is resumed and then defines `X` as `red`. This approach treats predicates as relations.

Most of the concepts a compiler has to deal with are functions of other items: e.g. "the type of an expression", "the offset of a variable" (an expression has unique type, a variable has a unique offset). Hence, the usual strategy in *Gentle* is shallow backtracking.

However, there is one important concept that is not functional by nature: a given source construct may have various possible representations in the target language. Here, one wishes to list possible translations of pieces which are automatically selected and combined into a globally optimal translation. This is also supported by *Gentle*. Since the number of possibilities in general is exponential, *Gentle* code does not enumerate them, but computes the best rule selection using a linear time algorithm. This will be discussed later.

1.4.7 Shallow Backtracking

Using shallow backtracking, a predicate invocation is elaborated as follows. The rules are tested in the given order. If a rule fails, the next rule is considered. If a rule succeeds, the predicate succeeds and the result is determined. If all rules fail, the predicate fails.

This implies that, when one considers a certain rule, one can be sure that preceding rules were not applicable.

For example, a predicate `max` computing the maximum of two integers (using the built-in predicates `ge` and `lt` for the relations greater-or-equal and less-than) that is defined by

```
'rule' max(X, Y -> X): ge(X, Y)
'rule' max(X, Y -> Y): lt(X, Y)
```

could also have been written as

```
'rule' max(X, Y -> X): ge(X, Y)
'rule' max(X, Y -> Y)
```

because the second rule is only used when `X` is less than `Y`. The disadvantage is that now the rule cannot be understood in isolation.

This style is often used for abbreviation. For example, a predicate `kind` that maps the set of letters `a`, `b`, `c`, ... , `z` onto the constants `vowl` or `consonant` can be written is this way:

```
'rule'  kind(a -> vowl)
'rule'  kind(e -> vowl)
'rule'  kind(i -> vowl)
'rule'  kind(o -> vowl)
'rule'  kind(u -> vowl)

'rule'  kind(Other -> consonant)
```

where the last rule replaces 21 rules dealing with consonants. Since none of the first five rules was applicable, we know that `Other` is a consonant.

1.4.8 Predicate Declarations

A predicate is declared by giving its signature and its rules. For example,

```
'action' length (ColorList -> INT)

   'rule' length (list(Head, Tail) -> N+1) : length(Tail -> N)
   'rule' length (nil -> 0)
```

The signature of a predicate specifies its interface to the caller. It gives the name of the predicate and lists its parameter types. For example,

```
'action' length (ColorList -> INT)
```

introduces a predicate `length` that has one input parameter of type `ColorList` and one output parameter of type `INT`.

Expressions and patterns in rule headings and invocations must be valid for the corresponding type.

In addition, a signature specifies the *category* of the predicate, which is `action` in the above example. The category of a predicate indicates how it is used and what its evaluation strategy is.

The general form is

Category Name `(` *InputTypes* `->` *OutputTypes* `)`

Type names are separated by commas. If there are no output parameters, the arrow is omitted.

Parameter specifications may include identifiers for documentation:

```
'action' length (L: ColorList -> N: INT)
-- on exit N is the number of elements of L
```

1.4.9 Conditions and Actions

Predicates of the category `'condition'` and `'action'` are evaluated by shallow backtracking, i.e. the rules are evaluated until one succeeds, in which case the predicate succeeds.

Should all rules fail, the behavior is different for conditions and actions.

In the case of conditions, the predicate fails. Predicates that are supposed to fail for certain arguments are used as guards in rules: if they fail, then at the place of invocation the next rule will be considered. By way of an example, consider

```
'condition' IsJimsColor(Color)
   'rule' IsJimsColor(red)
```

This predicate is intended to succeed for `red` and to fail for all other colors.

In contrast, consider the predicate `length` declared above. The predicate is intended to work for all arguments. It is not used to signal failure. Such predicates are declared with the keyword `'action'`. If for an action no rule is applicable, this does not indicate failure but an error in the specification. Hence, in this case an error message is issued that indicates the incomplete predicate, and the program is terminated.

For example, if we had omitted the rule dealing with the case `nil`, the specification of `length` would have be incomplete. If `length` had been declared as a condition, this error would not have been detected at the earliest possible point, but would simply result in failure of the rule that invoked `length` with an empty list. This again could propagate the failure to even wider contexts, making it hard to locate the error.

If a program is correct, all actions could be declared as conditions without changing the behavior. But if the program contains incompletely specified predicates, these can be detected much easier when using actions.

In addition, declaring a predicate as an action indicates to the human reader that this predicate is not intended to test its arguments for a certain condition, but to compute ouput values for all input values.

It turns out that the majority of predicates are actions. The reason is that, in *Gentle*, the program flow is controlled by pattern matching, i.e. the structure of the data being processed.

Other predicate categories are discussed in following sections.

1.5 Describing Syntax

We have discussed rules of the form

'rule' A : B_1 B_2 ... B_n

that could be interpreted as a decomposition scheme describing how a task A can be solved: Select a suitable rule for A, and solve the subtasks B_1, B_2, ... , B_n. Given the interpretation

A **if** B_1 **and** B_2 **and** ... **and** B_n

this could also be read as: A proof of A can be obtained from proofs of B_1 , B_2 , ... , B_n.

The same kind of rules can also be used to describe formal languages by recursive definitions. In the definition of *Algol 60* the following notation (known as *Backus-Naur-Formalism*, BNF) was used:

$$\langle A \rangle ::= \langle B_1 \rangle \ \langle B_2 \rangle \ ... \ \langle B_n \rangle$$

which states that a member of a syntactic class $\langle A \rangle$ can be composed from members of syntactic classes $\langle B_1 \rangle$, $\langle B_2 \rangle$, ..., $\langle B_n \rangle$.

1.5.1 Syntax Definitions

Using this grammatical interpretation of rules we can write

```
'rule' Program : Declarationpart Statementpart
```

which states that a `Program` is given by a `Declarationpart` and a `Statementpart`.

We also need basic symbols of the language that are not defined by rules. These symbols are called *terminal symbols* or *tokens* and are written as strings denoting themselves. For example,

```
'rule' Statement : "IF" Expression "THEN" Statement "ELSE" Statement
```

Here, `"IF"`, `"THEN"`, and `"ELSE"` are terminal symbols.

Grammatical symbols such as `Statement` that are defined by rules are called *nonterminal symbols*.

A grammar not only describes what is a valid text of a language, but also imposes a structure on the text. A grammar is given by a set of terminal symbols, a set of nonterminal symbols, and a set of rules. Rules have the form:

$$\text{'rule'} \ A : \ B_1 \ B_2 \ ... \ B_n$$

where A is a nonterminal and the B_i are terminal or nonterminal symbols. A valid phrase of class A can be constructed by replacing each B_i by a valid phrase of B_i (if B_i is a terminal, the valid phrase is the terminal itself).

Given the rule for `Statement` and assuming that `x>y` is an expression and that `x:=x-y` and `y:=y-x` are statements, we can construct the more complex statement

```
IF x>y THEN x:=x-y ELSE y:=y-x
```

The distinction between tokens and nonterminals reflects the fact that the analysis of the input text is generally decomposed into two (interleaved) steps: First the input is partitioned into tokens (lexical analysis), then the token stream is structured into hierarchical phrases (syntax analysis). Different techniques are used for each task.

1.5.2 Nonterm Predicates

Nonterminals must be declared as predicates of the class `'nonterm'`. For example, the full declaration of `Statement` might be

```
'nonterm' Statement

   'rule' Statement: Variable ":=" Expression
   'rule' Statement: "IF" Expression "THEN" Statement "ELSE" Statement
   'rule' Statement: "WHILE" Expression "DO" Statement
   'rule' Statement: "BEGIN" StmtSeq "END"
```

1.5.3 Token Predicates

Besides recognizing tokens that appear literally in the grammar, lexical analysis often also deals with items that appear as atoms for syntactic analysis but are structured at a lower level. An example of this are numbers that are treated as tokens, but specified as a sequence of digits at the level of lexical analysis.

Things like blanks, comments, and other separators that do not contribute to the syntactic structure are usually filtered out by lexical analysis as white space. Thus one criterion for deciding whether an item should be handled as a token is whether its components can be separated by white space: in a `Statement` a newline can (or must) appear between `IF` and the following `Expression`. In a `Number` this is not allowed as it would split the number into two tokens.

A complex token is introduced as a predicate of the category `token`. A predicate of this category is not defined by rules.

For example, a token `Variable` can be declared as

```
'token' Variable
```

and can then be used as member of a rule body:

```
'rule' Statement: Variable ":=" Expression
```

The declaration of a token does not provide rules. The actual description is given outside the *Gentle* specification. In many cases existing descriptions can be reused.

(See the manual on the *Reflex* tool for details of how to describe tokens and white space conventions.)

1.5.4 Attributes

Nonterm and token predicates may have parameters that are declared and used in the same way as with other predicates, except that for grammar symbols only output parameters are supported.

Given a structuring of the input into phrases (a parse), each nonterm application corresponds to a particular part of the input. The parameter values for that nonterm are often called *attributes* of the construct that corresponds to the nonterm application. Attributes can be computed from the attributes of constituents.

Here is a classical grammar that computes the value of a binary number:

```
'root' bits(-> N) print(N)

'nonterm' bits(-> INT)
'rule' bits(-> B): bit(-> B)
'rule' bits(-> N*2+B): bits(-> N) bit(-> B)

'nonterm' bit(-> INT)
'rule' bit(-> 0): "0"
'rule' bit(-> 1): "1"
```

Consider the input

```
1 0 1
```

which is decomposed by the second rule into `bits` (`1 0`) with value `N` = 2 and `bit` (`1`) with value `B` = 1. The attribute of the whole sequence is computed as `N*2+B` = 5.

1.5.5 Actions in Grammar Rules

Besides computations on parameter position (as in the above example), grammar rules can also contain normal statements such as predicate invocations to compute attribute values.

The second rule could have been written

```
'rule' bits(-> R): bits(-> N) bit(-> B) ShiftAndAdd(N, B -> R)
```

where `ShiftAndAdd` is an action.

1.5.6 Rule Selection

The rule-selection strategy used for nonterm predicates differs from that of actions and conditions. Instead of backtracking, an LALR(1) parsing algorithm is used. The *Gentle* compiler passes the underlying grammar to the parser generator *Yacc*. When *Yacc* processes the grammar it may flag certain rules as violating restrictions of this method. (See the documentation on *Yacc* for details of how to deal with this.)

Rule selection depends exclusively on the structure of the input and is not controlled by failure of predicates.

1.5.7 Abstract Syntax

It is advisable not to mix up the syntactic description of a language with its semantic analysis and transformation but to separate concerns. The syntax description gives the concrete grammar of the language and introduces a more abstract term representation. This term representation is then processed by predicates that follow the structure of these terms. This pays off because abstract syntax is usually much simpler than concrete syntax and because predicate evaluation is much more flexible than interleaving computations with parsing.

Given a concrete grammar, a straight forward way of defining an abstract representation is simply to mirror the grammar by term definitions.

For each nonterm there is corresponding type; for each grammar rule there is a corresponding functor. Terms with this functor have an argument for each nonterm and for each named token. Each nonterm has one attribute, its term representation. This is constructed from term representation of the members of the rule body.

For example, the nonterm `Statement` from above gets an associated type `STATEMENT` which is defined as follows:

```
'type' STATEMENT
   assignment(VARIABLE, EXPRESSION)
   if(EXPRESSION, STATEMENT, STATEMENT)
   while(EXPRESSION, STATEMENT)
   compound(STMTSEQ)
```

We can then decorate the above grammar:

```
'nonterm' Statement(-> STATEMENT)

   'rule' Statement(-> assignment(V, E)):
      Variable(-> V) ":=" Expression(-> E)
   'rule' Statement(-> if(E, S1, S2)):
      "IF" Expression(-> E) "THEN" Statement(-> S1)
      "ELSE" Statement(-> S2)
   'rule' Statement(-> while(E, S)):
      "WHILE" Expression(-> E) "DO" Statement(-> S)
   'rule' Statement(-> compound(S)):
      "BEGIN" StmtSeq(-> S) "END"
```

Abstract syntax can be simpler than concrete syntax.

Since abstract syntax has a unique tree structure, it is not necessary to reflect binding strength using different types, as we did with different nonterminals to avoid ambiguity. For example, in a preceding example, the nonterminals `expression` (introducing additive operators) and `expr2` (introducing multiplicative operators) both have an argument of type `Expr` that covers all operators.

Abstract syntax can allow cases that are forbidden in concrete syntax. For example, whereas in concrete syntax a list may require at least one member, in abstract syntax one could use empty lists. (Lists that are terminated by an empty tail, `nil`, are simpler to process because the rule for the end of the list does not have to deal with a list member.)

Abstract syntax can normalize cases in which the concrete syntax allows several representations for the same meaning, e.g. `A[i,j]` and `A[i][j]` in *Pascal*.

Abstract syntax can simplify complex constructs of the concrete syntax enforced by restrictions of the parsing strategy.

1.5.8 Source Coordinates

Syntactic errors are detected while the source file is being read, hence the actual position is available and can be used as part of an error message (this does not need to be programmed by the user). Semantic errors are detected when processing abstract syntax terms. In order to be able to issue a user-friendly error message, the source coordinates of the program text that is represented by the term should be available.

Gentle code automatically computes the coordinate of the "most significant" terminal of a construct. This is defined as follows: If a rule body contains one or more terminals, the position of the leftmost terminal is taken. If the rule body does not contain any terminals but one or more nonterminals then the coordinate of the leftmost nonterminal is taken. If the rule body is empty then the actual source position is taken.

This means, for example, that for expressions the coordinate of its top operator is taken. The expression `a*b+c*d` gets the coordinate of `+`. The expression `a-b*c-d` gets the coordinate of the second `-`.

This coordinate is available via the built-in predicate `@`, which can appear at the beginning of a syntax rule or after a nonterminal or token. If it appears at the beginning of a rule, it delivers the most significant coordinate of the phrase recognized by that rule; if it appears after a symbol, it delivers the coordinate of that symbol. An invocation has the form

```
@(-> P)
```

and the output parameter has the type `POSITION`.

The type `POSITION` need not be declared. Its representation is not defined by the *Gentle* language. A library provides functions that implement and process this type. Values normally contain an indication of the line and column of a coordinate. See the *Library Reference Manual* for details.

The position should be part of those abstract syntax terms that may be the subject of error messages. For example,

```
'type' DECLARATION
```

```
      declare(VARIABLE, TYPE, POSITION)

   'nonterm' Declaration(-> DECLARATION)
      'rule' Declaration(-> declare(V,T,P)):
         "DECLARE" Variable(-> V) @(-> P) ":" Type(-> T) ";"
```

selects the coordinate of the `Variable` and stores it in the abstract syntax term for a declaration. This can then be used for semantic processing.

1.6 Using Types and Predicates Written in C

Gentle tries to cover most tasks of compiler writing within a high-level uniform framework. However, there are also tasks for which languages like *C* are better suited.

Gentle allows the user to implement types and predicates in *C*, but use them as if they were implemented in *Gentle*.

The *Gentle* system comes with some ready-to-use *C* types and functions that are accessible in this way. The mechanism may also be applied by the user to build his or her own external types and functions.

1.6.1 External Types

An external type, i.e. a type whose values are not defined in *Gentle*, may be introduced by a type declaration without functor definitions.

By way of example, consider the declaration

```
   'type' IDENT
```

which introduces an external type `IDENT`.

Since in *Gentle* the values of this type are not specified they cannot be denoted by terms. On the other hand, values of an external type such as `IDENT` can be used in the same way as values of other types: they can appear as parameters of predicates and may be used as constituents of terms.

We could, for example, define a new type that introduces lists of `IDENT`s:

```
   'type' IDENTLIST
      list (IDENT, IDENTLIST)
      nil
```

and pass values of type `IDENT` as parameters as in

```
   'action' DeclareIdentList(IDENTLIST, TYPE)
```

```
'rule' DeclareIdentList(list(Ident, IdentList), Type):
   DeclareIdent(Ident, Type)
   DeclareIdentList(IdentList, Type)

'rule' DeclareIdentList(nil, Type)
```

1.6.2 External Predicates

Actions and conditions may also be implemented externally. To make them accessible, the *Gentle* specification must provide a declaration without rules (just as a type declaration without functors introduces an external type).

The declaration

```
'action' string_to_id (STRING -> IDENT)
```

that comes without rules introduces a predicate that takes a `STRING` and yields an `IDENT`. The implementation is not given in *Gentle*. The actual *C* implementation may insert the string into a hash table and return a unique reference to the entry. `IDENT` is the type of these references.

Other predicates may be declared that allow us to associate a "meaning" with an `IDENT` and access this later:

```
'action' DefMeaning (IDENT, MEANING)
'condition' HasMeaning (IDENT -> MEANING)
```

(See the *Library Reference Manual* for information on how to implement external types and predicates.)

1.7 Handling Global Information

Gentle encourages a declarative style of programming. However, there are cases in which non declarative elements are appropriate.

One problem is parameter lists that tend to get too long if every bit of context information is passed explicitly.

Consider a programming language that features a loop-statement and exit-statement and specifies that an exit-statement can only occur inside a loop statement. To check this constraint, the predicate that processes statements would have to be equipped with a parameter indicating whether one is inside or outside a loop. This parameter must be passed for each statement, e.g. an if-statement, because a constituent of that statement - e.g. the then-part - could contain an exit-statement. On the other hand, the constraint checking involves only the loop-statement and the exit-statement.

Gentle provides global variables. They are often used to represent a concept that is global to the concept processed by a particular rule.

A global variable is introduced by a declaration of the form

'var' *Name* : *Type*

For example,

```
'var' CurLoopContext: LoopContext
```

declares a global variable with the name `CurLoopContext` of type `LoopContext` (for which we assume values `inside` and `outside`).

This variable cannot be used inside expressions and patterns. The only way to manipulate and inspect its value is the *update statement* and the *query statement* that can be used as a member in a rule body.

The update statement has the form

Name <- *Expression*

which assigns the value of the *Expression* to the variable given by *Name*.

For example,

```
CurLoopContext <- outside
```

sets the variable `CurLoopContext` to `outside`. The old value of the variable gets lost.

The query statement has the form

Name -> *Pattern*

It takes the value of the variable and matches it against the given pattern. This may succeed, defining the variable inside the pattern. It may also fail, in which case the query statements fails.

For example,

```
CurLoopContext -> outside
```

succeeds only if the variable holds the value `outside`.

```
CurLoopContext -> K
```

defines a new local variable of type `LoopContext` whose value is the current value of `CurLoopContext`.

It is an error to access a global variable in a query statement if the variable has not been defined by a preceding update statement. This error is checked when the query statement is executed. Hence, term data structures cannot be corrupted by uninitialized global variables.

We now show how the above loop exit constraint can be checked. Assume that a predicate `ProcessStatement` traverses the abstract syntax of statements (we leave out all details). This could be extended at two points:

```
'rule' ProcessStatement(loop(Body)):
   CurLoopContext -> OldLoopContext
   CurLoopContext <- inside
   ProcessStatement(Body)
   CurLoopContext <- OldLoopContext

'rule' ProcessStatement(exit):
   CurLoopContext -> K
   CheckContextForExit(K)
```

1.8 Handling Mutable Information

Besides global variables, *tables* are another data structure that can be updated. They are motivated by the observation that in certain cases not all information required to construct a term is already available, and we need a mechanism to supply this later. As we will see, tables may also be used to represent cyclic structures.

In *Gentle*, one cannot modify the fields of a term, but one can supply an immutable key, that refers to a mutable record.

A *table* is an unbounded collection of records. Each record is identified by a unique key. A record is a list of one or more mutable fields.

A *table declaration* has the form

'table' *Name* (*Fields*)

where *Fields* is a comma-separated list of one or more field specifications. The identifier *Name* serves as a type name for the keys.

A field is specified in the form

FieldName : *Type*

To illustrate the use of tables, let us discuss the so-called basic-block-graph representation of programs. In this representation, the instructions of a program are grouped into blocks (only the last instruction of a block can be a jump; only the first instruction of a block can be the target of a jump).

Each block is characterized by a list of its instructions and a list of its successor blocks.

The basic-block graph can be described by a table:

```
'table' BLOCK (Instructions: INSTRLIST, Successors: BLOCKLIST)
```

This introduces a table of records, where each record has a field `Instructions` (of type `INSTRLIST`) and a field `Successors` (of type `BLOCKLIST`). It also introduces a new type, `BLOCK`, the values of which are references to block records.

The type `BLOCK` can be used in the same way as any other type, e.g. as the type of the fields of a term. Hence we can define the type `BLOCKLIST` as usual:

```
'type' BLOCKLIST
   list(BLOCK, BLOCKLIST)
   nil
```

A new record and its unique key are created by an *allocation statement*. It has the form

Var :: $Name$

This defines Var as a local variable of the key type $Name$, holding a unique key of a fresh record. The fields of this record are still undefined.

For example,

```
B :: BLOCK
```

defines the variable `B` as a new key of a `BLOCK` record. This key can already be used as a value in terms, and it cannot be altered (hence terms still have the property of being constants). But the associated record can be modified by subsequent statements.

The fields of an entry can be modified and inspected in exactly the same way as a global variable, i.e. by update and query statements.

Instead of a simple variable name, a *designator* is used that specifies the key and the field:

Key ' $FieldName$

For example,

```
B'Successors
```

designates the `Successors` field of the record associated with the key given by the local variable `B`.

An update statement

```
B'Successors <- nil
```

sets this field to `nil`.

A query statement

```
B'Successors -> L
```

accesses the field and copies its value into `L`.

With the statements

```
B'Successors -> OldList
B'Successors <- list(NewBlock, OldList)
```

a new block can be inserted into the list of successors of `B`.

It is a checked run-time error to access a field that has not been defined.

In general, the basic blocks of a program form a graph that contains cycles. This can be expressed with tables, for example:

```
'action' BuildLoop(-> BLOCK)

   'rule' BuildLoop(-> B1)

      B1 :: BLOCK
      B2 :: BLOCK

      B1'Instructions <- nil
      B2'Instructions <- nil

      B1'Successors <- list(B2, nil)
      B2'Successors <- list(B1, nil)
```

After an invocation

```
BuildLoop(-> B)
```

`B` will refer to a block with one successor, the successor of which is `B` itself.

1.9 Control Structures

So far, rules were the only mechanism for describing decisions. When two cases had to be described, two different rules were required. When a local decision was necessary inside a rule body, a new predicate was introduced to handle this.

In this section, we introduce further constructs to deal with decisions.

1.9.1 The Alternative Statement

Assume that we wish to process the maximum of two values by applying a predicate `process` to one of the values. Defining a predicate `processmax`, we can write

```
'action' processmax(INT, INT -> INT)
   'rule' processmax(X, Y -> Z): ge(X, Y) process(X -> Z)
   'rule' processmax(X, Y -> Z): lt(X, Y) process(Y -> Z)
```

and invoke it as in

```
get(-> A, B)
processmax(A, B -> C)
print(C)
```

The *alternative statement* discussed in this section allows us to insert the two rule bodies inline and get rid of the predicate `processmax`:

```
get(-> A, B)
(|
   ge(A, B) process(A -> C)
||
   lt(A, B) process(B -> C)
|)
print(C)
```

An alternative statement has the form

(| A_1 || A_2 || ... || A_n |)

where the A_i are called alternatives and have the same form as a rule body (i.e. an alternative is a sequence of statements).

The alternative statement is elaborated as follows. The alternatives are elaborated in the given order. If an alternative succeeds, then the alternative statement succeeds. If an alternative fails then the next alternative is elaborated. If all alternatives fail, then the whole alternative statement fails.

An alternative is elaborated by elaborating its statements. If one statement fails, then the alternative fails. If all statements succeed, then the alternative succeeds.

Roughly speaking, members of an alternative are connected by **and**, and alternatives are connected by **or**, but since we use shallow backtracking, inside an alternative one can assume that the preceding alternatives failed.

Thus the above could have been written as

```
get(-> A, B)
(| ge(A, B) process(A -> C) || process(B -> C) |)
print(C)
```

1.9.2 Dataflow

In our example, the variables `A` and `B` are defined outside the alternative statement. They can be used inside.

The general rule is: A variable that is used in a pattern must be in a defined state at the beginning of the whole alternative statement, or must defined by a preceding member of the same alternative.

In the example, the variable `C` is used after the alternative statement, and has no definition outside the alternative statement. This is allowed because it is defined in both alternatives. Regardless of which alternative is taken, `C` would be defined in all cases (if no alternative is applicable then statement using `C` will not be reached).

The general rule is: A variable that is defined in all alternatives is said to be defined by the whole alternative statement, and may be used subsequently. If a variable is defined in all alternatives, it must be defined everywhere with the same type.

If a variable is defined in one or more - but not all - alternatives, it is said to be *local* to the alternative containing the definition and cannot be used outside the alternative statement.

If, in the above example, we had written

```
(| ge(A, B) process(A -> ResultA) || process(B -> ResultB) |)
```

neither `ResultA` nor `ResultB` could have been used in a subsequent statement because it could not be guaranteed that it is always defined.

1.9.3 The Predicate `where`

In the previous example, each alternative invoked a procedure `process` that defined the common result variable `C`. If no such action is adequate, we have to use something similar to specify the result.

Gentle has a predefined predicate `where` that merely copies its input to its output. It is defined for each type `T`

```
'action' where(T -> T)
'rule' where(X -> X)
```

This can be used to define variables. For example,

```
where(red -> Col)
```

defines `Col` as `red`

Consider the predicate `max` with rules

```
'rule' max(X, Y -> X): ge(X, Y)
'rule' max(X, Y -> Y): lt(X, Y)
```

If we want to replace an invocation

```
max(A, B -> C)
```

we can write

```
(| ge(A, B) where(A -> C) || lt(A, B) where(B -> C) |)
```

Here, `C` is defined as the maximum of `A` and `B`.

In the example, an alternative was selected by giving a condition such as `ge` as a guard. If we wish to replace a predicate inline, the rules of which are selected by pattern matching, we have to use a different device.

Again, we can use the predefined predicate `where`. If we supply a pattern as the output value the input value is matched against this pattern. The pattern may contain variables, which are defined if matching succeeds. For examples,

```
where(Col -> red)
```

succeeds if `Col` is `red`.

Consider the predicate `MapType(Type -> Rep)` with rules

```
'rule' MapType(array(N,Type) -> Rep): ArrayType(N,Type -> Rep)
'rule' MapType(int -> Rep): IntType(-> Rep)
```

An invocation

```
MapType(Type -> Rep)
```

can be replaced by

```
(|
   where(Type -> array(N, Type))
   ArrayType(N, Type -> Rep)
||
   where(Type -> int)
   IntType(-> Rep)
|)
```

Here, `Rep` is defined in both alternatives, and therefore by the whole alternative statement. `N` and `Type` are variables that are local to the first alternative.

1.9.4 Disjunctions

Since an alternative statement succeeds if one of its alternatives succeeds, it can be used to express disjunction (**or** connection), whereas the members of a statement sequence express conjunction (**and** connection). Alternative statements can be nested. For example,

```
'rule' HandleType (Type):
   (|
      (| where(Type -> array(N, T)) || where(Type -> record(List) |)
      HandleStructuredType(Type)
   ||
      HandleSimpleType(Type)
   |)
```

`HandleStructuredType` is invoked if `Type` matches `array(N,T)` or `record(List)`. Otherwise `HandleSimpleType` is invoked.

1.9.5 The Conditional Statement

A frequent case is that, if a certain condition holds, a specific action should be initiated, otherwise nothing should happen. This could be expressed by an alternative statement with an empty second alternative.

For example,

```
(|  IsBad(Item) Complain(Item) || /*nothing*/ |)
```

This can be abbreviated by a *conditional statement*.

```
[|  IsBad(Item) Complain(Item) |]
```

An alternative statement has the form

`[|` M_1 M_2 ... M_n `|]`

The members M_i are elaborated in the given order. If one member fails elaboration is completed and and the conditional statement succeeds. If all members succeed the conditional statement succeeds.

A conditional statement cannot define variables because if one member fails its conceptual empty second case is taken which defines no variables.

1.9.6 Using Alternative and Conditional Statements

Alternative and conditional statements allow decisions within a rule body without the necessity of introducing an auxiliary predicate. When should predicates be used?

Of course predicates are necessary if the computation is recursive. Predicates should also be used when the same piece of code occurs more than once.

In addition, a predicate should be used to name a subcomputation if it is otherwise hard to understand or if the rule body gets longer than one can grasp. With conditional statements, there is a the danger of writing long rule bodies that are hard to understand.

On the other hand, a short conditional, given directly, is often more expressive than a newly invented name for an auxiliary predicate.

Conditional statements should also be used when correctness relies on the fact that earlier alternatives did not succeed (shallow backtracking). While rules are best understood if one can read them in isolation, conditional statements are read in an if-then-else style.

1.10 Smart Traversal

Some computations involve only a few functors, but to access them a traversal of a complex data structure must be specified.

Assume that in an abstract syntax a type `Expr` is defined with a functor

```
id(IDENT, POS)
```

which is used to represent an application of an identifier at a given position. We wish to describe the task of printing all identifier applications together with their coordinates. For this purpose we use a predicate

```
ListApplication(Id, Pos)
```

each time we encounter a term `id(Id, Pos)`.

For example,

```
'rule' VisitExpr(id(Id, Pos)): ListApplication(Id, Pos)
```

In order to list all identifier applications, we have to inspect the whole abstract syntax by a recursive traversal. For example, when we process a term `if(Cond, Then, Else))`, all three constituents can contain identifier applications. We have to write

```
'rule' VisitStmt(if(Cond,Then,Else)):
   VisitExpr(Cond) VisitStmt(Then) VisitStmt(Else)
```

and similar rules for all other functors.

This can be abbreviated using *sweep predicates*.

A sweep predicate traverses the data structure given as its argument. When an explicitly written rule is applicable this rule is taken. Otherwise a default rule is used that visits recursively the constituents of the argument. Hence rules such as

```
'rule' VisitExpr(id(Id, Pos)): ListApplication(Id, Pos)
```

must be specified, but rules such as

```
'rule' VisitStmt(if(Cond,Then,Else)):
   VisitExpr(Cond) VisitStmt(Then) VisitStmt(Else)
```

need not be written.

A sweep predicate is declared as being of of category `'sweep'`. It is a *generic predicate* in the sense that it works on all term types, the parameter is specified as being of type `ANY` (which is used as a generic type name).

The cross-reference application discussed above is completely specified by the following declaration

```
'sweep' Visit(ANY)
   'rule' Visit(id(Id, Pos)): ListApplication(Id, Pos)
```

Assuming that we specified the grammar by a predicate `Program`, we can list all applications of identifiers by writing

```
'root' Program(-> Pgm) Visit(Pgm)
```

1.10.1 Default Rule

Consider a predicate

'sweep' P (ANY)

i.e. with one parameter. For a given functor

f (T_1 , T_2 , ... , T_n)

the default rule has the form

'rule' $P($ $f(X_1$, X_2 , ... , $X_n)$) :
$P($ X_1) $P($ X_2) ... $P($ X_n)

where a member $P(X_i)$ is omitted if the corresponding type T_i is not defined by terms (e.g if it is `INT`).

1.10.2 Broadcasted Parameter

A sweep predicate may also have an additional input parameter of an arbitrary type T

'sweep' P (ANY, T)

Then, the additional parameter is passed to all constituents. The default rule has the form:

'rule' $P($ $f(X_1$, X_2 , ... , $X_n)$, Z) :
$P($ X_1, Z) $P($ X_2, Z) ... $P($ X_n, Z)

1.10.3 Threaded Parameter

A sweep predicate may also have an additional input and an output parameter, which must then both be of the same type:

'sweep' P (ANY, T -> T)

In the default rule, the parameter is threaded through the arguments (the input of a member is the output of its predecessor):

'rule' $P(\ f(X_1\ ,\ X_2\ ,\ ...\ ,\ X_n),\ Z_0$ -> $Z_n)$:
$P(\ X_1,\ Z_0$ -> $Z_1\ )\ P(\ X_2,\ Z_1$ -> $Z_2\ )\ ...\ P(\ X_n,\ Z_{n-1}$ -> $Z_n\ )$

For example, to count all identifier applications, we can write

```
'root' Program(-> Pgm) Count(Pgm, 0 -> N) print(N)

'sweep' Count(ANY, INT -> INT)
   'rule' Count(id(Id, Pos), N -> N+1)
```

1.11 Optimal Rule Selection

This section introduces cost-driven rule selection. This is of particular use for unparsing systems that do not define a unique translation. The compiler writer does not have to specify how to combine pieces to form the best translation but simply lists possible alternatives for the parts.

1.11.1 Code Generation for Expressions

Here, we discuss how to generate code for expressions.

For the sake of simplicity we let the abstract syntax of expressions be defined as

```
'type' Stmnt
   assign(Variable, Expr)

'type' Expr
   plus(Expr, Expr)
   minus(Expr, Expr)
   var(Variable)
```

1.11.2 A Stack Computer

Our target computer is a stack computer with the following instructions (the instructions are given as predicates, that, when invoked, emit the instruction):

```
'action' PLUS
'action' MINUS
'action' PUSH(Variable)
'action' POP(Variable)
```

The instructions modify the stack of the computer. If the stack has the form

... X Y

then `PLUS` replaces the top two elements by their sum $Z = X+Y$, i.e. the stack becomes

... Z

Similarly, `MINUS` replaces X and Y by $Z = X - Y$.

If K is the value of a variable `V`, then `PUSH(V)` puts K onto the stack.

If K is the value on top of the stack, then `POP(V)` removes it from the stack and stores it in `V`.

Let us write predicates `Encode` and `StackCode` that emit instructions for statements and expressions.

If we have to generate code for `assign(V, X)`, we have to emit code for the expression `X` that computes its value on top of the stack. Then, we can use the `POP` instruction to store it in `V`.

```
'rule' Encode(assign(V,X)):
   StackCode(X)
   POP(V)
```

The predicate `StackCode` processes the alternatives for `Expr`:

```
'rule' StackCode(plus(X,Y)):
   StackCode(X)
   StackCode(Y)
   PLUS
'rule' StackCode(minus(X,Y)):
   StackCode(X)
   StackCode(Y)
   MINUS
'rule' StackCode(var(V)):
   PUSH(V)
```

For the `Statement`

```
assign("Res", plus("A", minus("B", "C")))
```

the generated code is

```
PUSH A
PUSH B
PUSH C
MINUS
PLUS
POP Res
```

which, when executed, results in the following stack configurations

... (A)
... (A) (B)
... (A) (B) (C)
... (A) $(B - C)$
... $(A + (B - C))$
...

1.11.3 Unparsing

The above scheme is called an *unparsing scheme*. While a parser reads a source text (driven by the structure of the source text) and constructs abstract syntax, an unparser processes abstract syntax terms (driven by their structure) and emits a concrete target program.

The rules of an unparser are similar to grammar rules. Predicates like `StackCode` correspond to nonterminals, and predicates that directly emit target code correspond to tokens. In a source grammar, symbols are decorated with output parameters that specify their abstract representation. In a target grammar, symbols are decorated with abstract input parameters for which a concrete representation has to be generated.

In many cases an unparsing scheme simply follows the structure of terms. In general there is one rule for each functor, although sometimes larger patterns are used. This results in a functional description such as that for the stack-computer code generator.

This will especially be the case if the target language is a higher-level language, as in source-to-source translations (e.g. if a parallel extension of C is translated into plain C).

1.11.4 A Computer with an Accumulator

Let us consider a different kind of computer that uses an accumulator to compute values. The accumulator holds one of the operands of a computation, the other being given as a parameter of an instruction. The result is placed into the accumulator.

These are the instructions:

```
'action' ACCUPLUS(Variable)
'action' ACCUMINUS(Variable)
'action' LOADACCU(Variable)
'action' STOREACCU(Variable)
```

`ACCUPLUS(X)` means `Accu := Accu + X`, `ACCUMINUS(X)` means `Accu := Accu - X`, `LOADACCU(X)` means `Accu := X`, `STOREACCU(X)` means `X := Accu`.

Here is an incomplete specification of code generation:

```
'rule' Encode(assign(V,X)):
   AccuCode(X)
   STOREACCU(V)

'rule' AccuCode(plus(X, var(V))):
   AccuCode(X)
   ACCUPLUS(V)
'rule' AccuCode(minus(X, var(V))):
   AccuCode(X)
   ACCUMINUS(V)
'rule' AccuCode(var(V)):
   LOADACCU(V)
```

(The specification is still incomplete because we only consider the case where the second operand of `plus` and `minus` is a variable. If it were be a more complex expression, we would have to introduce temporary variables.)

1.11.5 A Combined Stack and Accumulator Computer

We show cost driven-rule selection by a computer combining stack and accumulator processing (following Professor Ulrich Grude, who coined this example for his lectures on *Gentle*).

The combined computer provides both sets of instructions, i.e. one can switch between stack- and accumulator-based computation. In addition, it offers two further instructions that move data from stack to accumulator and vice versa:

```
'action' STACKTOACCU
'action' ACCUTOSTACK
```

`STACKTOACCU` pops the top element from the stack and stores it in the accumulator. `ACCUTOSTACK` pushes the current value of the accumulator onto the stack.

Since the specification for the stack computer covers all cases, we can simply copy these rules. Hence, we already have a correct code generator for the combined computer.

Because the rules for the accumulator-based computer are also correct, we copy them as well.

We add a rule that computes stack values in the accumulator:

```
'rule' StackCode(X):
   AccuCode(X)
   ACCUTOSTACK
```

and a rule that computes accumulator values using the stack:

```
'rule' AccuCode(X):
   StackCode(X)
   STACKTOACCU
```

1.11.6 Nondeterministic Unparsing

While the specification for stack code was functional, i.e. there was one unique translation for each abstract syntax term, the combined specification allows more than one translation.

The combined specification is *correct* in the sense that each possible rule selection results in correct target code. It is also *complete* in the sense that for each argument term there is a possible selection of rules that emits code for the term. Hence, it should suffice as a specification of code generation.

This is the case if we use *cost-driven* rule selection.

Cost-driven rule selection is performed for predicates that are declared as being of the category `choice`. Rules of such predicates may be augmented by cost values. A rule is given a cost by providing a specification of the form

`$` *N*

at the end of the rule. *N* is a positive integer indicating the price for applying the rule.

When evaluating invocations of `choice` predicates, conceptually all possible sequences of rule applications are considered. The total cost of a sequence of rule applications is the sum of the costs of all rules. The sequence of rule applications that leads to the minimal total cost is actually used for evaluation (if there are several best sequences one of them is taken).

1.11.7 A Code-Generator Specification

Here is a specification of a code generator for the combined stack and accumulator computer:

```
'choice' Encode(Stmnt)

   'rule' Encode(assign(V,X)):
      StackCode(X)
      POP(V)
```

```
      $ 10
   'rule' Encode(assign(V,X)):
      AccuCode(X)
      STOREACCU(V)
      $ 10

'choice' StackCode(Expr)

   'rule' StackCode(plus(X,Y)):
      StackCode(X)
      StackCode(Y)
      PLUS
      $ 20
   'rule' StackCode(minus(X,Y)):
      StackCode(X)
      StackCode(Y)
      MINUS
      $ 20
   'rule' StackCode(var(V)):
      PUSH(V)
      $ 10
   'rule' StackCode(X):
      AccuCode(X)
      ACCUTOSTACK
      $ 10

'choice' AccuCode(Expr)

   'rule' AccuCode(plus(X, var(V))):
      AccuCode(X)
      ACCUPLUS(V)
      $ 10
   'rule' AccuCode(minus(X, var(V))):
      AccuCode(X)
      ACCUMINUS(V)
      $ 10
   'rule' AccuCode(var(V))
      LOADACCU(V)
      $ 10
   'rule' AccuCode(X):
      StackCode(X)
      STACKTOACCU
      $ 20
```

1.11.8 Cost-Driven Rule Selection

A rule gives the cost of the respective operation of the rule. For example, in the `PLUS` operation counts `20` in

```
'rule' StackCode(minus(X,Y)):
   StackCode(X)
   StackCode(Y)
   MINUS
   $ 20
```

The total cost of translating `minus(X,Y)` with this rule would be `20` plus the total cost of translating `X` plus the total cost of translating `Y`.

The corresponding accumulator operation is cheaper:

```
'rule' AccuCode(minus(X, var(V)):
   AccuCode(X)
   ACCUMINUS(V)
   $ 10
```

The computation of an expression is cheaper if performed in the accumulator; hence the translation of source source text

```
r := x-y
```

is

```
LOADACCU x
ACCUMINUS y
STOREACCU r
```

and not

```
PUSH a
PUSH b
MINUS
POP r
```

which is also a possible result of a series of rule applications.

If the expression appears in a context where stack code is required (the outer minus does not have a simple operand) as in

```
r := (x-y)-(a-b)
```

it is computed on the stack:

```
PUSH x
PUSH y
MINUS
PUSH a
PUSH b
MINUS
MINUS
POP r
```

because an operation that moves the intermediate result from the stack to the accumulator would be too costly.

If a larger subexpression is used, then the cheaper accumulator computation outweighs the cost of the `ACCUTOSTACK` instruction. The input

```
r := (x-y-z)-(a-b)
```

is translated into

```
LOADACCU x
ACCUMINUS y
ACCUMINUS z
ACCUTOSTACK
PUSH a
PUSH b
MINUS
MINUS
POP r
```

1.11.9 Using Output Parameters

In the above example, we used predicates such as `PLUS` to write the generated code to the target file. It can also be returned as an output parameter of the predicate.

In many specifications, both methods are used. For example, an invocation

```
Register(X -> R)
```

emits code for the term `X` such that the result is stored in a register. The output parameter `R` states which register is used.

Often, these output parameters are then used to construct a more complex result in the rule heading. This result can specify a computation for which no instructions have to be emitted, but which can be performed "via an addressing mode".

By way of an example, consider a rule from a code-generator specification for the MC68020:

```
'rule' EffectiveAddress
       (  addrplus(Array, intmult(Index, intconst(Size)))
       -> ax(AR, 0, DR, Size) ) :
   IsSuitableScaleFactor (Size)
   AddressRegister (Array -> AR)
   DataRegister (Index -> DR)
   $ 14
```

Here, the `choice` predicates `EffectiveAddress`, `AddressRegister`, and `DataRegister` translate their input argument such that the result is given as an "effective address", an address register, or a data register. The rule states that the term

```
addrplus(Array, intmult(Index, intconst(Size)))
```

(which is the translation of an array subscription) may be mapped to

```
ax(AR, 0, DR, Size)
```

(the "address register with index" addressing mode) when the value of `Array` is computed in `AR` and the value of `Index` is computed in `DR`. To be a suitable operand for the `ax` mode, `Size` must fulfill the condition `IsSuitableScaleFactor`.

1.11.10 The Two Processing Phases

Rules of `choice` predicates are processed in two phases: in the first phase, the optimal rule combination is determined; in the second phase, these rules are elaborated just as for ordinary predicates.

To determine which rules should be applied, the first phase performs pattern matching and evaluates conditions. Conditions invoked from rules of `choice` predicates should only test their arguments, but must have no side effects such as changing global variables.

In the second phase, only those rules are elaborated that were selected in the first phase. During rule elaboration, now also predicates of the category `action` are invoked. These invocations may have side effects, such as emitting code.

1.11.11 Well-formed Rules

The typical structure of rules describing code selection is as follows.

The predicate has a distinguished input argument (the first one) that represents the construct to be translated; we call it the *primary argument*.

If the rule invokes conditions, this is to test certain properties of the primary argument (such as `IsSuitableScaleFactor` in the example above).

If the rule invokes predicates of the category `choice`, this is to translate constituents of the primary argument, e.g.

```
'rule' StackCode(plus(X,Y)):
   StackCode(X)
   StackCode(Y)
   PLUS
   $ 20
```

In some cases, the argument is simply passed as a whole to the invoked predicate, e.g.

```
'rule' StackCode(X):
   AccuCode(X)
   ACCUTOSTACK
   $ 10
```

Since `choice` predicates were introduced to specify code selection, we can restrict ourselves to this form in order to support an efficient implementation.

Rules of `choice` predicates must obey the following restrictions:

The first input argument (the primary argument) of a `choice` must be of a type that is defined with functors.

If `condition` or `choice` predicates are invoked in the rule body, then the input arguments of `condition` predicates and the primary arguments of `choice` predicates are given by constituents of the primary argument of the rule heading (or by that argument as a whole).

Rule selection does not depend on the output values of predicates in the rule body.

These restrictions make it possible to avoid an exponential search and to use a linear-time algorithm to find optimal rules.

1.12 Special Patterns and Expressions

1.12.1 Joker

In some cases, one is not interested in all constituents of a term. For example, if we wish to process the head of a list but without consisdering the tail, we may write:

```
'rule' ProcessHead(list(Head, Tail)):
   ProcessColor(Head)
```

In this rule, two variables are defined in a pattern (`Head` and `Tail`), but only one of them is applied in an expression (`Head`), In such cases, it is not necessary to invent a name for a variable that is not used (here `Tail`); instead, we can use a *joker*, which is written as an underscore ("`_`") and matches any value. Hence, the above rule can be rewritten:

```
'rule' ProcessHead(list(Head, _)):
   ProcessColor(Head)
```

1.12.2 Named Patterns

Sometimes one wishes to process the constituents of a pattern, but also the pattern as a whole. We can use the `where` predicate to inspect the term, as in

```
'rule' HandleSignal(S):
   where (S -> signal(C1, C2))
   ProcessSignal(S)
   ProcessColor(C1)
   ProcessColor(C2)
```

Alternatively, we can prefix a pattern with a variable name (followed by a colon). This is equivalent to the unprefixed pattern but if matching succeeds, then the named variable is defined as if the pattern had simply been the variable name.

Thus, the above rule could also be written:

```
'rule' HandleSignal(S : signal(C1, C2)):
   ProcessSignal(S)
   ProcessColor(C1)
   ProcessColor(C2)
```

1.12.3 Type Prefixes

In virtually all cases, the type of a particular position is clear from the context. Hence one can use the same names for functors of different types.

There is one exception where the type is not known: the parameter of "generic" predicates such as the built-in predicate `where` or user defined predicates of the category `sweep`.

For example, in

```
where(nil -> List)
```

it is not clear what kind of list should be constructed if there are several types (say, `ColorList` and `StatementList`) that introduce `nil`.

To resolve the ambiguity, we can prefix the functor with the name of the type (followed by a single quote ("'")) as in

```
where(ColorList'nil -> List)
```

The same holds for the first parameter of `sweep` predicates. For example, in

```
'sweep' Visit(ANY)

'rule' Visit(list(Head, Tail)):
   ProcessElem(Head)
   Visit(Tail)
```

it is not clear whether the rule deals with values of type `ColorList` or `StatementList` if both types introduce the functor `list`. Again, we can prefix the ambiguous term:

```
'sweep' Visit(ANY)

'rule' Visit(ColorList'list(Head, Tail)):
   ProcessElem(Head)
   Visit(Tail)
```

1.13 A Summary of Predefined Predicates

Gentle has some built-in predicates that can be used like user-defined predicates but that need not be declared. This section summarizes them.

1.13.1 The Predicates `eq`, `ne`, `lt`, `le`, `ge`, `gt`

The conditions `eq` and `ne` are used to compare values for equality.

```
eq(X, Y)
```

succeeds if X is equal to Y and fails otherwise.

```
ne(X, Y)
```

succeeds if X is not equal to Y and fails otherwise. These predicates work for all types including user-defined types, i.e. they may be used to compare terms. For example,

```
eq(list(red, X), list(red, list(yellow,nil)))
```

succeeds if `X` is `list(yellow,nil)`.

The predicates `gt`, `ge`, `lt`, and `le` are defined only for the types `INT` and `STRING`.

```
gt(X, Y)
```

succeeds if `X` is greater than `Y` and fails otherwise.

For example,

```
gt(N, 0)
```

succeeds if `N` is a positive number.

In the case of strings, the lexicographical order is used for comparison. For example,

```
gt("axy", "abc")
gt("aaaa", "aa")
```

succeed.

Similarly, `ge`, `lt`, and `le` stand for "greater or equal", "less than", and "less or equal", respectively

1.13.2 The Predicate `print`

The predicate `print` may be used to print a value. It is meant for test output and may be used, for example, to print the generated abstract syntax of a source text:

```
Expression(-> X) print(X)
```

The value will be printed using indentation for subterms.

1.13.3 The Predicate `where`

The predicate `where` is defined for each type. It simply copies its input parameter to its output parameter. Hence, it may be used to construct a value and assign it to a variable as in

```
where(list(red, nil) -> L)
```

or to inspect a value and define variables with subterms as in

```
where(L -> list(Head, Tail))
```

This predicate is of particular use in alternative statements and is discussed in the corresponding section.

1.13.4 The Predicate `@`

The predicate `@` is used in grammar rules to obtain the source coordinate of a construct as in

```
Expr(-> X) @(-> P)
```

where `P` is defined as the source coordinate of the preceding `Expr`.

This predicate is discussed in the section on grammars.

1.14 Organizing Larger Projects

A *Gentle* specification is a list of declarations of types, predicates, and other items. In simple cases, a specification is written into a single file and submitted to the compiler.

If projects become larger and if more than one programmer is involved, it becomes desirable to split a specification into several files. Nevertheless, the compiler must still be able to check whether an application of an item (say, a predicate) matches the corresponding definition, even if that appears in a different file. *Gentle* supports this kind of separate compilation.

A specification may be decomposed into *modules*, where each module is written into its own file.

The content of a module is a list of declarations. Hence, the simple case can be understood as a one-module specification.

In addition to the list of declarations, a module may have a heading that lists the name of the module, other modules whose items are used here, as well as items declared here and made available to other modules.

Here is an example of a module heading:

```
'module' CodeGen

'use'
   IR, TargetFile

'export'
   Encode
```

This introduces a module name `CodeGen`. The module uses items that are declared in two other modules named `IR` and `TargetFile`. It declares an item (say a predicate) `Encode` that can be used by other modules.

1.14.1 The Module Name

A module has a unique name (an identifier). If the name of a module is `Mod`, then the corresponding file must have the name `Mod.g` .

The module name is introduced by a *name clause* of the form

```
'module' Mod
```

where `Mod` is the name of the module.

If the name clause is missing, the module name is determined by the file name (this style should only be used for single-module specifications).

1.14.2 The Use Clause

The *use clause* follows the name clause. It may be empty or of the form

```
'use' M1 , ... , Mn
```

where `M1 , ... , Mn` is a list of module names (optionally separated by commas). The items provided by modules that appear in this list can be used in the module containing the use clause.

For example. if module `IR` provides a definition of a type `Expr` and module `CodeGen` contains a use clause

```
'use'
   IR, TargetFile
```

then the type `Expr` can be used inside `CodeGen`.

A module with an empty use clause cannot use items declared in other modules.

1.14.3 The Export Clause

The *export clause* follows the use clause. It may be empty or of the form

```
'export' I1 , ... , In
```

where `I1 , ... , In` is a list of identifiers (optionally) separated by commas. The identifiers name items that are declared inside the module. These items are provided for other modules. They can be used there if the module containing the export clause appears in the use clauses of the other modules.

E.g. if a module `CodeGen` contains an export clause

```
'export'
   Encode
```

then it must declare an item (say a predicate) `Encode`. This item can be used in the module `Compiler` if `Compiler` lists `CodeGen` in its use clause.

Items declared in a module but not listed in the export clause are not available to other modules. (A module does not constitute a separate name space; hence the name of such an item cannot be reused in other modules.)

1.14.4 The Root Module

Exactly one module must contain a root definition that triggers the computation.

For example, a specification could be given by three modules: `Compiler`, `IR`, and `CodeGen`, where the module heading of `Compiler` is

```
'module' Compiler

'use'
   IR, CodeGen
```

The module `Compiler` can contain a definition

```
'root'
   Program(-> P)
   Encode(P)
```

This specification is elaborated by first invoking `Program` declared in `Compiler`, and then `Encode` declared in `CodeGen`.

2 GETTING STARTED

2.1 Installation

Gentle runs on various platforms, including *Unix*, *DOS*, and *Windows 95*. For *Unix* systems, *Gentle* is distributed as a compressed tar file `gentle-97.tar.gz` . To unpack the system, type

```
gunzip gentle-97.tar.gz
tar -xvf gentle-97.tar
```

This creates a directory `gentle-97` with subdirectories `gentle`, `lib`, `reflex`, and `examples`.

Go to directory `gentle` and type

```
build
```

This creates the executable program `gentle`, the *Gentle* compiler, as well as the object file `grts.o`, the *Gentle* run time system.

Go to directory `lib` and type

```
build
```

This compiles the modules of the user library.

Go to directory `reflex` and type

```
build
```

This creates the executable program `reflex`, a generator for *Lex* specifications.

To test the installation, go to directory `examples/calc` and type

```
build
```

This creates a simple desk calculator.

2.2 A First Example

This section describes how to use the *Gentle* system to construct a simple desk calculator. This calculator will read (from standard input or from a specified file) an expression according to the following syntax:

```
expression ::=

   expression "+" expr2
|  expression "-" expr2
|  expr2

expr2 ::=

   expr2 "*" expr3
|  expr2 "/" expr3
|  expr3

expr3 ::=

   Number
|  "-" expr3
|  "+" expr3
|  "(" expression ")"
```

where the token `Number` represents a sequence of decimal digits. The program will calculate the value of the expression and print it on standard output.

Here is a *Gentle* specification of the calculator (file `calc.g`). This simply mirrors the above grammar, where each nonterminal has an output parameter that represents its value.

```
'root' expression(-> X) print(X)

'nonterm' expression(-> INT)

   'rule' expression(-> X): expr2(-> X)
   'rule' expression(-> X+Y): expression(-> X) "+" expr2(-> Y)
   'rule' expression(-> X-Y): expression(-> X) "-" expr2(-> Y)

'nonterm' expr2(-> INT)

   'rule' expr2(-> X): expr3(-> X)
   'rule' expr2(-> X*Y): expr2(-> X) "*" expr3(-> Y)
   'rule' expr2(-> X/Y): expr2(-> X) "/" expr3(-> Y)

'nonterm' expr3(-> INT)

   'rule' expr3(-> X): Number(-> X)
   'rule' expr3(-> - X): "-" expr3(-> X)
   'rule' expr3(-> + X): "+" expr3(-> X)
```

```
   'rule' expr3(-> X): "(" expression(-> X) ")"

'token' Number(-> INT)
```

The representation of the token `Number` is not specified in the *Gentle* specification. Instead, we use a token description file from the `reflex` directory (file `Number.t`):

```
[0-9]+ {
   yylval.attr[1] = atoi(yytext);
   yysetpos();
   return Number;
}
```

Assume that `$GENTLE` is the path name for the program `gentle`, `$GRTS` for the *Gentle* run time system `grts.o`, `$REFLEX` for the program `reflex`, and `$LIB` is the user library.

Then the command sequence

```
$GENTLE calc.g
$REFLEX
lex gen.l
yacc gen.y
cc -o calc       \
   calc.c        \
   lex.yy.c      \
   y.tab.c       \
   $LIB/errmsg.o \
   $LIB/main.o   \
   $GRTS
```

can be used to create the calculator program `calc`.

Here the command

```
$GENTLE calc.g
```

invokes the *Gentle* compiler, which translates the specification `calc.g` into a *C* file `calc.c`. In addition, it generates some files `gen.*`, which the user need not bother about.

The command

```
$REFLEX
```

invokes the program `reflex`. This creates a specification for the scanner generator *Lex* (from files created by `gentle` and files provided by the user such as `Number.t`).

The commands

```
lex gen.l
yacc gen.y
```

invoke the scanner generator *Lex* and the parser generator *Yacc* respectively.

Finally, the *C* compiler creates the program `calc`:

```
cc -o calc       \
   calc.c        \
   lex.yy.c      \
   y.tab.c       \
   $LIB/errmsg.o \
   $LIB/main.o   \
   $GRTS
```

`lex.yy.c` and `y.tab.c` are *C* files created by *Lex* and *Yacc*. `$LIB/errmsg.o` and `$LIB/main.o` are modules from the user library. They provide an error message routine and and main function that invokes the code created by the *Gentle* compiler (a user may use these components as they are, or adapt them according to his or her needs).

If `testfile` contains the line

```
2+3*4
```

then the command

```
calc testfile
```

emits `14` .

2.3 Generating a Compiler

To generate a compiler from a *Gentle* specification the *Gentle* compiler has to be invoked for each *Gentle* module. Then the *Reflex* program is used to produce a *Lex* specification. *Lex* and *Yacc* are used to generate a scanner and a parser. The *C* compiler then translates the generated and (possibly user supplied) *C* modules. Figure 1 shows the way these tools cooperate.

Gentle

The *Gentle* compiler is invoked by the command

```
gentle module.g
```

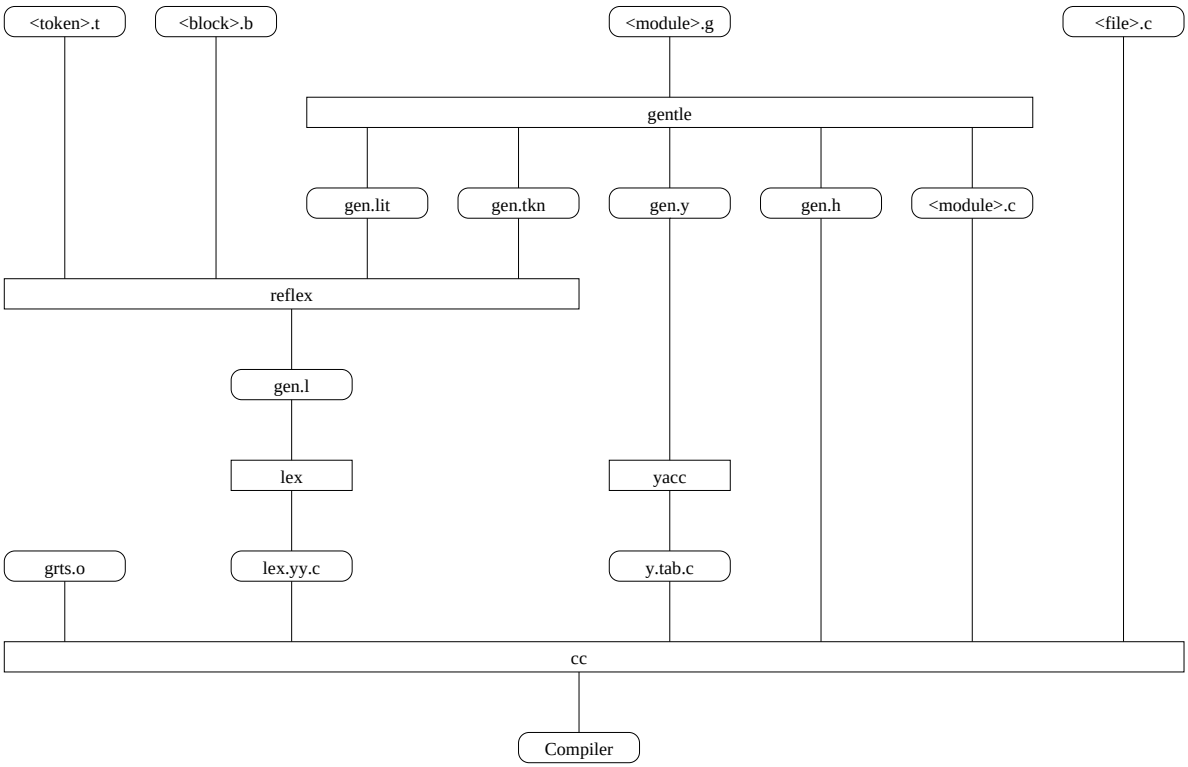


Figure 1: Generating a Compiler

This translates a *Gentle* module `module.g` into a *C* module `module.c` (file names for *Gentle* modules must have the suffix `.g`).

If the module contains a grammar specification, then the following files are generated in addition: `gen.lit` contains *Lex* specifications for terminal symbols that appear literally in the specification. `gen.tkn` contains a list of tokens introduced in the specification. `gen.h` is a *C* header file that introduces a data type for token attributes and defines the codes of the tokens. `gen.y` is a *Yacc* specification of the underlying grammar.

See the *Language Reference Manual* for a description of *Gentle*.

Reflex

Reflex assembles various files and combines them to form a *Lex* specification.

For each token `Token` introduced in the specification, there must be a token description file `Token.t` . This file must specify a regular expression for the token and an action that computes the attributes of the token.

A block `Block` of the *Lex* specification may be overridden by a file `Block.b` . This allows the specific definition of white space and comment conventions (`LAYOUT.b`, `COMMENTS.b`).

The directory `reflex` contains reusable token description and block files.

Reflex is invoked by the command

```
reflex
```

This creates a *Lex* specification `gen.l` .

See the *Reflex Reference Manual* for details.

Lex

The command

```
lex gen.l
```

invokes the scanner generator *Lex* with the specification `gen.l` . This creates a *C* module `lex.yy.c` .

Yacc

The command

```
yacc gen.y
```

invokes the parser generator *Yacc* with the specification `gen.y` . This creates a *C* module `y.tab.c` .

C Compiler

The command

```
cc file.c ... grts.o
```

creates an executable program. `file.c ...` is a list of generated or user-written *C* files. One of these modules must supply a function `main` that invokes the generated function `ROOT()`. The library provides such a function (module `main.c`).

See the *Library Reference Manual* for a description of library modules.

3 LANGUAGE REFERENCE MANUAL

3.1 Introduction

Gentle is a high-level programming language for compiler writers. It covers the whole spectrum of compiler construction, ranging from analysis over transformation to synthesis. *Gentle* provides a uniform notation for all tasks.

Language Overview

The language is based on recursive definition and structural induction, the underlying paradigm of virtually all translation tasks.

Gentle allows the user to define mutally recursive types by enumerating alternative structures

```
Expr = plus(Expr,Expr), minus(Expr,Expr), const(INT)
```

Programs in *Gentle* are expressed by rules of the form

```
G : A B C
```

These rules may be interpreted as grammar rules (`G` is constructed from `A`, `B`, and `C`), as logical statements (`G` is true if `A`, `B`, and `C` are true), or in a procedural manner (to solve task `G`, solve subtasks `A`, `B`, and `C`).

Members of a rule may have parameters (an arrow separating input from output parameters). This results in attributed grammars

```
AddingExpression(-> plus(X1, X2)):
   AddingExpression(-> X1) "+" Primary(-> X2)
```

or in transformation schemes that inductively follow the structure of terms

```
Eval(plus (X1, X2)->N1+N2): Eval(X1->N1) Eval(X2->N2).
Eval(minus(X1, X2)->N1-N2): Eval(X1->N1) Eval(X2->N2).
Eval(const(N)     ->N).
```

Unparsing may be expressed in a similar way

```
Code(plus(X1, X2) -> Reg2) :
   Code(X1 -> Reg1)
   Code(X2 -> Reg2)
   Emit("add", Reg1, Reg2).
```

Such rules can be augmented by cost values. Then, rules are selected in such a way that the sum of the costs of all selected rules is optimal. This allows a nondeterministic specification that handles idioms of the target machine by rules with corresponding patterns.

Gentle is a strongly typed language that allows a tool to statically check the consistency of the specification. Traditional errors such as uninitialized variables can be detected at the earliest point.

Gentle has a declarative flavor. It also provides mutable global variables and tables to represent deferred information and cyclic structures. *Gentle* supports procedures written in other languages.

Related Languages

Gentle descends from *CDL* [7] and *Prolog* [15]. An early version was described in [10].

3.2 Syntax and Vocabulary

The syntax of *Gentle* is defined in the *Extended Backus Naur Formalism* which uses the following conventions: `X|Y` denotes `X` or `Y`, `[X]` denotes `X` or empty, `{X}` denotes a possibly empty sequence of `X`'s, and `(X)` denotes `X`.

Terminal symbols are enclosed in quotes. In addition, Ident denotes a sequence composed of letters, digits, and underscores and starting with a letter. IdentLC is an Ident that starts with a lower-case letter, IdentUC is an Ident that starts with an upper-case letter. Number denotes a sequence of decimal digits. String denotes a possibly empty sequence of characters enclosed in quotes (‘`"`’). Inside a String, the sequence ‘`\t`’ represents a tabulator, ‘`\n`’ represents a new-line character, ‘`\\`’ stands for ‘`\`’, and ‘`\"`’ stands for ‘`"`’.

Blanks, newlines, tabulators, and comments may be inserted between symbols. A comment is a sequence of characters starting with ‘`--`’ and ending at the end of the line, or a sequence of characters enclosed in ‘`/*`’ and ‘`*/`’ (comments of this style may be nested).

3.3 Specifications

A *Gentle* specification is a list of declarations. A declaration introduces a type, a predicate, a context variable, or a context table. Exactly one declaration of a specification must be a root definition, this defining the elaboration of the specification. Declarations may be given in any order.

Example

```
'root' bits(-> N) print(N)

'nonterm' bits(-> INT)
'rule' bits(-> B): bit(-> B)
'rule' bits(-> N*2+B): bits(-> N) bit(-> B)

'nonterm' bit(-> INT)
'rule' bit(-> 0): "0"
'rule' bit(-> 1): "1"
```

This program reads a sequence of binary digits and prints the corresponding decimal value.

3.4 Types

TypeDecl = "'type'" Ident ["="] TermSpec {[","] TermSpec}] .
TermSpec = IdentLC ["("[ParamSpec {"," ParamSpec}]")"] .
ParamSpec = [Ident ":"] Ident .

Term Types

A type declaration

'type' T
f_1 ($T_{1,1}$, ... , T_{1,n_1})
...
f_m ($T_{m,1}$, ... , T_{m,n_m})

defines a type T and a set of values of that type. A value is called a term and is constructed by applying a functor f_i to arguments. If τ_1, ... , τ_{n_i} are values of types $T_{i,1}$, ..., T_{i,n_i}, then $f_i(\tau_1, \ldots, \tau_{n_i})$ is a value of type T.

Examples

```
'type' Color
   red, yellow, blue
```

Values of type `Color` are: `red`, `yellow`, and `blue`.

```
'type' List
   list(Color, List)
   nil
```

Values of type `List` include:
`nil`, `list(red,nil)`, `list(yellow(list(red,nil))`.

A parameter specification may be written as

$N : T$

where N is an identifier documenting the meaning of the parameter.

Example
`list(Head: Color, Tail: List)`

Abstract Types

A type declaration without functor specifications introduces an abstract type. The values of such a type are not specified in *Gentle*.

Example
`'type' IDENT`

Basic Types

There are three basic types that may be used without being declared. The type `INT` comprises integer numbers. The type `STRING` comprises strings of characters. The type `POS` comprises source coordinates.

3.5 Expressions

Expression = Expr2 | Expression ("+" | "−") Expr2 .
Expr2 = Expr3 | Expr2 ("∗" | "/") Expr3 .
Expr3 = IdentUC | Number | String

```
    | Functor ["("[Expression {"," Expression}]")"]
    | ( "+" | "-" ) Expr3 | "(" Expression ")"
Functor = [Ident "'"] IdentLC .
```

An expression describes the computation of a value. The value of an expression E on a position with type T is computed as follows.

Variables

If E has the form

V

where V is a variable, the value of the expression is the value of the variable. The type of the variable must be T.

Numbers

If E has the form

N

where N is a number, T must be `INT`. The value of E is the value of the number.

Strings

If E has the form

S

where S is a string, T must be `STRING`. The value of E is the value of the string.

Terms

If E has the form

f `(` E_1 `,` ... `,` E_n `)`

the declaration of T must contain a functor specification f `(` T_1 `,` ... `,` T_n `)`. E_1, ..., E_n appear on positions with types T_1, ..., T_n. E_1, ..., E_n are evaluated yielding values σ_1, ..., σ_n. The value of E is given as f `(` σ_1 `,` ... `,` σ_n `)`.

If the type T of the position is unknown (in the case of generic predicates) and if f is defined for more than one type, f must be prefixed by the type: T `'` f .

Example
`list(X1, list(yellow, X2))`

If the value of `X1` is `red` and the value of `X2` is `nil`, the value of the expression is `list(red(list(yellow,nil))`.

Arithmetic Expressions

If E has the form

E_1 *op* E_2

where *op* is one of `+`, `-`, `*`, `/`, T must be `INT` and E_1 and E_2 must be expressions of type `INT`. The value of P is computed by applying the operator to the values of E_1 and E_2.
If E has the form

op E_1

where *op* is one of `+`, `-`, T must be `INT` and E_1 must be an expression of type `INT`. The value of P is computed by applying the operator to the value of E_1.
If E has the form

`(` E_1 `)`

the value of E is the value of E_1. E_1 must be an expression of type T.

3.6 Patterns

```
Pattern = IdentUC [":" Pattern]
    | Functor ["("[Pattern {"," Pattern}]")"]
    | "_"
```

A given value τ may be matched against a pattern P. This may succeed, thereby defining the variables appearing in P, or fail.

Variables

If P has the form

V

where V is a variable, the value of V is defined as τ and the type of V is the type of the position.

Terms

If P has the form

$$f \texttt{ (} P_1 \texttt{ , } \ldots \texttt{ , } P_n \texttt{)}$$

the matching succeeds if τ has the form f `(` τ_1 `,` $\ldots$ `,` τ_n `)` and matching τ_1, ..., τ_n against P_1, ..., P_n succeeds, thereby defining the variables appearing in P_1, ..., P_n. Otherwise, the matching fails.

If P appears on a position with type T, the declaration of T must contain a functor specification f `(` T_1 `,` $\ldots$ `,` T_n `)`. P_1, ..., P_n appear on positions with types T_1, ..., T_n.

Example

```
list(X1, list(yellow, X2))
```

If a value τ is matched against this pattern (on a position of type `List`), the matching succeeds if τ has the form `list(` τ_1 `,list(yellow,` τ_2 `))`. It defines `X1` as a variable of type `Color` with value τ_1 and `X2` as a variable of type `List` with value τ_2. Otherwise, the matching fails.

Named Patterns

If P has the form

$$V \texttt{ : } P_1$$

τ is matched against P_1 as if the pattern P had been simply P_1. If this succeeds, V is defined as if the pattern P had been simply V.

Joker

If P has the form

_

(underscore) the matching always succeeds.

3.7 Predicates

PredicateDecl = Category Ident Signature Rules .
Category = "'nonterm'" | "'token'" | "'action'"
    | "'condition'" | "'choice'" | "'sweep'" .
Signature =
    ["(" [ParamSpec {"," ParamSpec}] ["->"[ParamSpec {"," ParamSpec}]]")"] .
Rules = {Head ":" Body "."} | {"'rule'" Head [":"] Body ["."]} .
Head = Ident ["(" [Pattern {"," Pattern}] ["->" [Expression {"," Expression}]] ")"] .
Body = {Member} [CostSpec]
CostSpec = "$" Number .

Predicate Declarations

A predicate declaration of the form

Category *p* (T_1 , ... , T_n -> S_1 , ... , S_m)
$Rule_1$
...
$Rule_r$

introduces a predicate *p* of category *Category* with *n* input parameters of types T_1, ..., T_n, and with *m* output parameters of types S_1, ..., S_m.

A predicate may be used in a predicate invocation.

A predicate that is invoked with input values τ_1, ..., τ_n is elaborated as follows. If no rule is applicable, the invocation fails. Otherwise, a rule is selected and elaborated, yielding output values σ_1, ..., σ_n, and the invocation succeeds.

Example

```
'condition' HasCode (Color -> INT)
'rule' HasCode(red -> 1)
'rule' HasCode(yellow -> 2)
```

An invocation of `HasCode` with input value `yellow` succeeds and yields output value 2. An invocation of `HasCode` with input value `blue` fails.

Rules

A rule has the form

'rule' p (P_1 , ... , P_n -> E_1 , ... , E_m) : M_1 ... M_k

For given input values τ_1, ..., τ_n, the rule is elaborated as follows. Let V_1, ..., $V_{n_{in}}$ be the variables appearing in P_1, ..., P_n, and W_1, ..., $W_{n_{out}}$ the variables appearing in E_1, ..., E_m. If the input values τ_1, ..., τ_n do not match the patterns P_1, ..., P_n, the rule is not applicable. Otherwise, the variables V_1, ..., $V_{n_{in}}$ are defined. Then the members M_1 ... M_k are elaborated. If one of the members fails, the rule is not applicable. Otherwise, the variables W_1, ..., $W_{n_{out}}$ are defined. The expressions E_1 ... E_m are evaluated, yielding output values σ_1, ..., σ_m.

A variable that appears in a pattern is said to be defined. A variable that appears in an expression is said to be applied. Each applied variable must be a defined variable. Variables that are applied in a member M_i must be defined by one of the patterns P_1, ..., P_n or in a member M_j that appears to the left of M_i . A defined variable cannot be redefined.

Example

```
'action' append(List, List -> List)
'rule' append(list(H,T), L -> list(H, TL): append(T, L -> TL)
'rule' append(nil, L -> L)
```

For input values τ_1 = `list(red,nil)` and τ_2 = `list(yellow,nil)`, the second rule is not applicable because $\tau_1 \neq$ `nil`.

The elaboration of the first rule matches τ_1 with `list(H,T)` defining `H` as `red` and `T` as `nil`; it matches τ_2 with `L`, defining `L` as `list(yellow,nil)`.

The member `append(T,L -> TL)` defines `TL` as `list(yellow,nil)`.

The output value σ_1 = `list(red,(list(yellow,nil))` is yielded by replacing `H` and `TL` in `list(H,TL)` by their values.

3.8 Context Variables

VariableDecl = "'var'" Ident ":" Ident .

A declaration of the form

'var' C : T

introduces a context variable C of type T.

The value of a context variable is defined by a context update and is accessed in a context query.

Example

`'var' NestingLevel: INT`
defines a context variable `NestingLevel` of type `INT`.

This variable may be defined as in
`NestingLevel <- N`
and accessed as in
`NestingLevel -> N`

3.9 Context Tables

TableDecl = "`'table'`" Ident "`(`" Ident "`:`" Ident {"`,`" Ident "`:`" Ident} "`)`" .

A declaration of the form

`'table'` *T* `(` F_1 `:` T_n `,` ... `,` F_n `:` T_n `)`

introduces a context table *T*.

A context table is an unbounded collection of entries. An entry has a unique key and a record of fields. These fields have names $F_1, \ldots, F_n$ and types $T_1, \ldots, T_n$. The name of the table, *T*, acts as the name for the type of the keys.

A unique key is created by a key definition statement. The value of a field is defined by a context update and accessed in a context query.

Example

`'table' Label (Coordinate: INT)`
This defines a table `Label`. Entries of this table have one field with name `Coordinate` and type `INT`. Keys of entries have type `Label`.

A value of type `Label` can be created without knowing the value of field `Coordinate`.
`Lab :: Label`
defines `Lab` as unique value of type `Label`.
`Lab'Coordinate <- Loc`
may be used later to define the field `Coordinate`.
`Lab'Coordinate -> Loc`
is used to access the field `Coordinate`.

3.10 Members

```
Member = Ident ["(" [Expression {"," Expression}] ["->" [Pattern {"," Pattern}]] ")"]
    | ContextDesignator "<-" Expression
    | ContextDesignator "->" Pattern
    | IdentUC "::" Ident
    | String
    | "(|" {Member} "||" {Member} {"||" {Member}} "|)"
    | "[|" {Member} "|]" .
ContextDesignator = [IdentUC "'"] Ident .
```

The body of a rule is given by a sequence of members. A member is elaborated as follows.

Predicate Invocations

A member of the form

$$p\ (\ E_1\ ,\ \ldots\ ,\ E_n\ \texttt{->}\ P_1\ ,\ \ldots\ ,\ P_m\ )$$

denotes a predicate invocation.

p must be a predicate with n input parameters and m output parameters. Let T_1, ..., T_n be the input types and S_1, ..., S_m the output types. The expressions E_1, ..., E_n appear on positions with types T_1, ..., T_n, and the patterns P_1, ..., P_m appear on positions with types S_1, ..., S_m.

The expressions E_1, ..., E_n are evaluated, yielding values τ_1, ..., τ_n. p is invoked with these input values. If the invocation fails, the member M fails. If it succeeds, it yields output values σ_1, ..., σ_m. These values are matched against the patterns P_1, ..., P_m. If the matching fails, the member M fails. Otherwise, the member succeeds and the variables appearing in P_1, ..., P_m are defined.

Context Updates

A member of the form

$$D\ \texttt{<-}\ E$$

denotes an context update.

The value of D is defined as the value of the expression E. If T is the type of D, E appears on a position with type T.

The context designator D may be of the form C, where C must be declared as a context variable. The type of the designator is the type of the context variable.

Example

```
NestingLevel <- N
```

This sets the value of the context variable `NestingLevel` to the value of the local variable `N`.

A context designator may also have the form I `'` F ; I must be a variable of the key type of a context table T. Entries of T must have a field F. The type of the designator is the type of the F.

Example

Let `Lab` be a local variable that refers to a table entry with a field `Coordinate`. Then

```
Lab'Coordinate <- Loc
```

sets this field to the value of `Loc`.

Context Queries

A member of the form

D `->` P

denotes a context query.

The value of the context designator D is matched against the pattern P. If this succeeds, the member succeeds and the variables appearing in P are defined. Otherwise, the member fails.

Examples

```
NestingLevel -> N
```

This defines the local variable `N` according to the current value of the context variable `NestingLevel`.

Let `Lab` be a local variable that refers to a table entry with a field `Coordinate`. Then

```
Lab'Coordinate -> Loc
```

defines the local variable `Loc` according to the value of this field.

It is a checked run-time error to access an undefined context designator.

Key Definitions

A member of the form

V `::` T

denotes a key definition.

The variable V is defined as a unique key of type T.

Example
`Lab :: Label`
The value of the local variable `Lab` is the unique key of a newly created entry of table `Label`.

Anonymous Tokens

A member of the form

S

where S is a string, is equivalent to a member of the form p, where p is an implicitly declared predicate of class token whose lexical representation is S.

Alternative Statements

A member of the form

`(|` A_1 `||` ... `||` A_n `|)`

denotes an alternative statement.

The alternatives A_1, ..., A_n are evaluated from left to right until one alternative A_i succeeds. Then M succeeds. If all alternatives fail, M fails.

An alternative A_i has the form $M_{i,1}$, ..., M_{i,n_i}. These members are evaluated from left to right until one member fails. Then the alternative fails. If all members succeed, the alternative succeeds.

A variable that is defined in all alternatives A_1, ..., A_n is defined by the alternative statement and may be used outside it. Such a variable must be defined with the same type in all alternatives.

A variable that is defined in one but not all alternatives is local to that alternative.

Conditional Statements

A member of the form

`[|` *A* `|]`

is equivalent to `(|` *A* `||  /* empty */  |)`.

3.11 Root Definition

RootDef = "`'root'`" {Member} .

A root definition has the form

`'root'` $M_1 \ldots M_n$

The elaboration of a specification is the elaboration of the members M_1, ..., M_n.

3.12 Modules

Module = ["`'module'`" Ident] [UseClause] [ExportClause] Declarations .
UseClause = "`'use'`" Ident {"`,`" Ident} .
ExportClause = "`'export'`" Ident {"`,`" Ident} .

A specification may be decomposed into several modules.

A Module M has the form

`'module'` M
`'use'` M_1 `,` ... `,` M_n
`'export'` E_1 `,` ... `,` E_m
D_1
...
D_d

Items introduced by the declarations D_1, ..., D_d and items exported by modules M_1, ..., M_n are visible in M. E_1, ..., E_m must be items declared in D_1, ..., D_d. They are exported by M, i.e. the can be made visible in other modules.

3.13 Predicate Categories

The *Category* specification of a predicate declaration may be one of

"`'nonterm'`", "`'token'`", "`'action'`",
"`'condition'`", "`'choice'`", "`'sweep'`".

The category of a predicate determines the rule-selection strategy applied for that predicate.

Grammar Predicates

`token` and `nonterm` predicates constitute the underlying context-free grammar and are evaluated by LR parsing.

Action and Condition Predicates

`condition` and `action` predicates are evaluated by shallow backtracking. `action` predicates may not fail.

Choice Predicates

Rules of `choice` predicates may be augmented by cost specifications. Rules are selected in such a way that the sum of the costs of all rules applied is minimal.

Rule selection for `choice` predicates must depend exclusively on the first argument of the predicate: only the first argument or its constituents may be used as arguments to `condition` predicates or as the first argument of `choice` predicates invoked in a rule of a `choice` predicate.

Invocations of `condition` predicates in a rule of `choice` predicates may not change context variables or tables or have other side effects.

Sweep Predicates

`sweep` predicates provide implicit traversals where only rules deviating from the default rule have to be specified. The default rule recursively visits the fields of its first argument.

A `sweep` predicate p is declared in one of three forms

```
'sweep' p (ANY)
'sweep' p (ANY, T )
'sweep' p (ANY, T -> T )
```

where T is an arbitrary type. The first argument of a `sweep` predicate may be of any type that is declared with functors

f (T_1 , ... , T_n)

The default rule for a functor f has the form

```
'rule' p ( f ( X_1 , ... , X_n )):
    p ( X_1 )
    ...
    p ( X_n )
```

or

```
'rule' p ( f ( X1 , ... , Xn ), Z ):
    p ( X1 , Z )
    ...
    p ( Xn , Z )
```

or

```
'rule' p ( f ( X1 , ... , Xn ), Z0 -> Zn ):
    p ( X1 , Z0 -> Z1 )
    ...
    p ( X1 , Zn-1 -> Zn )
```

3.14 Predefined Predicates

The following predicates can be used without being declared:

`eq`, `ne`, `gt`, `ge`, `lt`, `le`,
`print`, `where`, `@` .

Relations

The predicates `eq` and `ne` are declared for each type `T`

```
'condition' eq(T, T)
'condition' ne(T, T)
```

The invocations

```
eq(X, Y)
ne(X, Y)
```

succeed if `X` is equal (not equal) to `Y`.

The predicates `gt`, `ge`, `lt`, `le`,
are declared

```
'condition' gt(T, T)
'condition' ge(T, T)
'condition' lt(T, T)
'condition' le(T, T)
```

where `T` is `INT` or `STRING`.

The invocations

```
lt(X, Y)
le(X, Y)
gt(X, Y)
ge(X, Y)
```

succeed if `X` is less than (less or equal to, greater than, greater or equal to) `Y`.

Printing Values

The predicate `print` is defined for each type `T`.

```
'action' print(T)
```

An invocation

```
print(X)
```

prints the value of `X`.

Inspecting and Defining Variables

The predicate `where` is defined for each type `T`.

```
'action' where(T -> T)
'rule' where(X -> X)
```

Examples

`where (List -> list(Head, Tail))`
inspects the value of `List` and succeeds if it matches the pattern `list(Head,Tail)`, thereby defining the variables `Head` and `Tail`.

`where (list(Head, Tail) -> List)`
constructs the value `list(Head,Tail)` and defines the variable `List`.

Source Coordinates

The predicate `@` is defined

```
'action' @(-> POS)
```

It may appear after a `nonterm` or `token` predicate and it defines its output variable as the coordinate of the source text recognized by the preceding symbol. It may also appear at the beginning of a rule for a grammar rule, in which case it defines its output variable as the coordinate of the source text recognized by the body of the rule.

The coordinate of the text recognized by a rule body is the coordinate of the leftmost `token`, if there is one, or else the coordinate of the leftmost `nonterm`. If the body is empty, the current coordinate is used.

Examples

```
'rule' Statement: "IF" @(->P) Expression "THEN" Statement
```

Here, `P` is defined as the coordinate of `"IF"`.

```
'rule' Statement: "IF" Expression @(->P) "THEN" Statement
```

Here, `P` is defined as the coordinate of `Expression`.

3.15 Syntax Summary

```
Module = ["'module'" Ident] [UseClause] [ExportClause] Declarations .
UseClause = "'use'" Ident {"," Ident} .
ExportClause = "'export'" Ident {"," Ident} .
Declarations = {Declaration} .
Declaration = TypeDecl | PredicateDecl | VariableDecl | TableDecl | RootDef .
TypeDecl = "'type'" Ident [["="] TermSpec {["|"] TermSpec}] .
TermSpec = IdentLC ["("[ParamSpec {"," ParamSpec}]")"] .
ParamSpec = [Ident ":"] Ident .
VariableDecl = "'var'" Ident ":" Ident .
TableDecl = "'table'" Ident "(" Ident ":" Ident {"," Ident ":" Ident} ")" .
PredicateDeclaration = Category Ident Signature Rules .
Category = "'nonterm'" | "'token'" | "'action'"
    | "'condition'" | "'choice'" | "'sweep'" .
Signature =
    ["(" [ParamSpec {"," ParamSpec}] ["->"[ParamSpec {"," ParamSpec}]]")"] .
Rules = {Head ":" Body "."} | {"'rule'" Head [":"] Body ["."]} .
Head = Ident ["(" [Pattern {"," Pattern}] ["->" [Expression {"," Expression}]] ")"] .
Pattern = IdentUC [":" Pattern]
    | Functor ["("[Pattern {"," Pattern}]")"]
    | "_"
Expression = Expr2 | Expression ( "+" | "-" ) Expr2 .
Expr2 = Expr3 | Expr2 ( "*" | "/" ) Expr3 .
Expr3 = IdentUC | Number | String
    | Functor ["("[Expression {"," Expression}]")"]
    | ( "+" | "-" ) Expr3 | "(" Expression ")"
Functor = [Ident "'"] IdentLC .
Body = {Member} [CostSpec]
CostSpec = "$" Number .
Member = Ident ["(" [Expression {"," Expression}] ["->" [Pattern {"," Pattern}]] ")"]
    | ContextDesignator "<-" Expression
    | ContextDesignator "->" Pattern
    | IdentUC "::" Ident
    | String
    | "(|" {Member} "||" {Member} {"||" {Member}} "|)"
    | "[|" {Member} "|]" .
ContextDesignator = [IdentUC "'"] Ident .
RootDef = "'root'" {Member} .
```

4 REFLEX REFERENCE MANUAL

4.1 Introduction

The scanner for a compiler generated with *Gentle* must be implemented as a function `yylex()`. This may be done manually or by using the scanner generator *Lex*. The tool *Reflex* provides a convenient way of constructing a scanner specification for *Lex* from information extracted from the *Gentle* specification, token description files, and other sources. In many cases no manual intervention is required, because existing descriptions can be reused.

4.2 How To Describe a Token

For each token `Token` introduced in the *Gentle* specification, there must be a token description file `Token.t` . This file must specify a *Lex* rule that handles the token. Such a rule is given by a regular expression that matches the token, and an action that computes the attributes of the token.

```
pattern { action }
```

If the rule does not fit on one line, the action may be written on several lines, but the opening brace must appear on the first line.

By way of an example, a token introduced in a *Gentle* specification as

```
'token' Number (-> INT)
```

may be described in file `Number.t`:

```
[0-9]+ {
   yylval.attr[1] = atoi(yytext);
   yysetpos();
   return Number;
}
```

The pattern

```
[0-9]+
```

matches a non-empty sequence of digits.

The line

```
yylval.attr[1] = atoi(yytext);
```

converts the matched input token (`yytext`) into an integer (using the C function `atoi`) and assigns it as the first attribute. The *Lex* variable `yylval` has a field `attr` which is an array of token attributes. `attr[`*i*`]` stands for the *i*-th attribute.

The macro

```
yysetpos();
```

defines the source position of the token (the source position is stored in `yylval.attr[0]`). In a *Gentle* specification, this value may be accessed with the `@`-predicate.

The line

```
return Number;
```

returns the code of the token. The token code is a constant that has the same name as the token.

4.3 How To Describe Layout and Comments

The generated *Lex* specification contains blocks that can be overridden by the user. A block `Block` may be replaced by the content of a file `Block.b` .

There is a block `LAYOUT` which unless overridden has the following form:

```
\  { yypos += 1; }
\t { yypos += 1; }
\r { yypos += 1; }
\n { yyPosToNextLine(); }
```

These are three *Lex* rules that match blank, newline, and tabulator. They do not include `return` statements, so the input recognized by the rule is skipped.

The variable `yypos` keeps track of the source position. It is incremented by the length of the input. In the case of newlines the variable is adjusted by the library function `yyPosToNextLine`.

This handling of white space may be overridden by providing a file `LAYOUT.b` .

In addition, a file `COMMENTS.b` may define comments that are to be skipped. By default, the `COMMENTS` block is empty.

4.4 Usage

Reflex is invoked by the command

```
reflex [ name=file ... ]
```

Each token `Token` has to be defined by a file `Token.t`. Alternatively, an argument `Token=file` may be used to specify a file with a different name (the file must have the suffix `.t`).

Each block `Block` may be overridden by a file `Block.b`. An alternative file name may be specified by an argument `Block=file` (the file must have the suffix `.b`).

4.5 Output

A specification constructed by *Reflex* has the following structure:

```
%{
YYSTYPE Block
SETPOS Block
LITBLOCK Block
%}
LEXDEF Block
%%
Token Section
COMMENTS Block
LAYOUT Block
ILLEGAL Block
%%
LEXFUNC Block
YYWRAP Block
```

The block `YYSTYPE` has the form

```
#include "gen.h"
extern YYSTYPE yylval;
```

The block `SETPOS` has the form

```
extern long yypos;
#define yysetpos() { yylval.attr[0] = yypos; yypos += yyleng; }
```

The blocks `LITBLOCK` and `LEXDEF` are empty.

The Token Section contains, for each token `"alpha"` appearing literally in the *Gentle* specification a *Lex* rule of the form

```
"alpha" { yysetpos(); return token_code; }
```

These rules are followed by the rules appearing in the token description files.

The block `COMMENTS` and `LAYOUT` are described above.

The block `ILLEGAL` has the form

```
. { yysetpos(); yyerror("illegal token"); }
```

The block `LEXFUNC` is empty.

The block `YYWRAP` has the form.

```
#ifndef yywrap
yywrap() { return 1; }
#endif
```

5 LIBRARY REFERENCE MANUAL

5.1 Introduction

Besides the *C* code generated by *Lex*, *Yacc*, and *Gentle* some additional code must be provided to construct a functioning compiler. This includes a main procedure for handling command-line arguments and procedures for error reporting. *Gentle* allows the invocation of procedures implemented in *C*.

The library supplies a set of *C* modules. It covers the basic functionality as required for classroom projects, and provides a foundation for user-specific extensions.

5.2 Module `main`

For compilers generated with *Gentle* a *C* function `main` must be provided. This function must call the generated function `ROOT`, which has no parameters and elaborates the `'root'` clause of the *Gentle* specification. In the simplest case the procedure `main` could have the form

```
main() { ROOT(); }
```

In addition the function `main` may contain code for initialization and finalization. The parameters of main provide access to the *Unix* command arguments.

The library provides a simple version of `main`. It implements a command with none or one argument. If no argument is specified, standard input is used as source file. If one argument is specified, this is taken as name of the source file.

5.3 Module `errmsg`

The module `errmsg` provides procedures to emit error messages. This module supports the *Gentle* type `POS` and encodes the current file, line, and column into a value.

```
long yypos;
```

The variable `yypos` keeps track of the current source position.

To advance to the next column this variable must be incremented by one. To advance to the next line or the next file the procedures `yyPosToNextLine` or `yyPosToNextFile` must be used.

At the beginning of a line the column is set to one, after reading a token the column is the first column after the token.

```
void yyGetPos(ref_pos)
   long *ref_pos;
```

The procedure `yyGetPos` sets the variable referred by its argument to the current source position minus 1.

```
void yyPosToNextLine()
```

The procedure `yyPosToNextLine` must be called by the lexer when a new source line is read. It adjust the variable `yypos`.

```
void yyPosToNextFile()
```

The procedure `yyPosToNextFile` must be called by the lexer when a new source file is read. It adjust the variable `yypos`.

```
'action' Error (STRING, POS)
```

```
void Error(msg, pos)
   char *msg;
   long pos;
```

The procedure `error` prints the message specified by the first parameter together with the source position specified by the second parameter. Then the program is terminated.

```
yyerror(msg)
   char *msg;
```

The procedure `yyerror` is called by the *Yacc* generated parser in case of syntax errors. It may also be called by the lexer. It behaves like `Error` where the actual source position is used.

5.4 Module `idents`

The module `idents` implements an abstract type `IDENT`.

```
'type' IDENT
```

A value of type `IDENT` (an identifier) is a reference to an entry of the identifier table maintained by the module.

Each entry has a unique string representation which may be used to select the entry: the procedure `string_to_id` provides a mapping from strings to values of type `IDENT`. Such an identifier may have a meaning: procedure `HasMeaning` defines a mapping from identifiers to meanings.

```
'action' string_to_id (STRING -> IDENT)

void string_to_id (string, ref_id)
   char *string;
   IDENT *ref_id;
```

If an identifier with the string representation given by the first parameter exists in the identifier table it is a assigned to the variable referred by the second parameter. Otherwise such an identifier is created and assigned to that variable.

```
'action' id_to_string (IDENT -> STRING)

void id_to_string (id, ref_string)
   IDENT id;
   char **ref_string;
```

The procedure assigns the string representation of the identifier given as first parameter to the variable referred by the second parameter.

```
'action' DefMeaning (IDENT, MEANING)

void DefMeaning (id, m)
   IDENT id;
   long m;
```

The procedure `DefMeaning` defines the value of its second parameter as the meaning of the identifier given as the first parameter (a previous value is overridden).

`MEANING` may be any *Gentle* type.

```
'action' UndefMeaning (IDENT)

void UndefMeaning (id)
   IDENT id;
```

The procedure `UndefMeaning` indicates that the identifier given as parameter has no associated meaning (a previous value is overridden).

```
'condition' HasMeaning (IDENT -> MEANING)

int HasMeaning (id, ref_meaning)
   IDENT id;
   long *ref_meaning;
```

If the identifier given as first parameter has an associated meaning it is assigned to the variable referred by the second parameter, the procedure succeeds (returns 1). Otherwise the procedure fails (returns 0).

```
'action' ErrorI (STRING, IDENT, STRING, POS)

ErrorI (str1, id, str2, pos)
   char *str1;
   IDENT id;
   char *str2;
   long pos;
```

Procedure `ErrorI` is equivalent to procedure `Error` of module `errmsg` except that the text of the error message is given as the concatenation of the first parameter, the string representation of the identifier given as second parameter, and the third parameter.

5.5 Module `output`

The module `output` provides procedures to write the target file of a compiler.

```
'action' OpenOutput (STRING)

OpenOutput (Name)
   char *Name;
```

If open the target file is closed. The procedure `OpenOutput` opens the target file for output with the file name given as parameter.

Before writing to the file this procedure must be called.

```
'action' CloseOutput

CloseOutput ()
```

The procedure `CloseOutput` flushes and closes the target file.

It must be called after the last output to the target file.

```
'action' Put(STRING)

Put(Str)
   char *Str;
```

The procedure `Put` writes its argument to the target file.

```
'action' PutI(INT)

PutI (N)
   long N;
```

The procedure `PutI` writes its argument to the target file.

```
'action' Nl(INT)

Nl ()
```

The procedure `Nl` writes a newline character to the output file.

5.6 Module strings

The module `strings` provides a string memory that can handle strings of arbitrary length. It may be used for the lexical processing of strings. The lexer may extend the current string (which is initially empty) character by character using procedure `AppendToString`. The string is terminated by procedure `GetStringRef` which yields a reference to the string.

```
AppendToString (ch)
   char ch;
```

The procedure `AppendToString` appends the character given as parameter to the current string.

```
GetStringRef(ref_string)
   char **ref_string;
```

The procedure `GetStringRef` terminates the current string with a null character and assigns a reference to the string to the variable referred by the parameter. The current string is set to empty.

5.7 Implementing Types and Predicates in C

A type that is is defined without functors or a procedure that is defined without rules must be implemented in *C*.

Abstract types may be implemented as values of type `long` or as pointers.

For example, if a *Gentle* specification contains the declaration

```
'type' IDENT
```

this type may be implemented in a *C* module as

```
typedef struct IDENTSTRUCT *IDENT;
```

which defines values of type `IDENT` as references to structures.

Abstract procedures are implemented by *C* routines that take `long` arguments as input parameters and use `(long *)` arguments to realize output parameters.

For example, if a *Gentle* specifications defines a procedure as

```
'action' Max (INT, INT -> INT)
```

it may be implemented in C as

```
void Max (x, y, ref_result)
   long x;
   long y;
   long *ref_result;
{
   *ref_result = (x > y ? x : y);
}
```

The *Gentle* type `INT` corresponds to the *C* type `long`. *Gentle* terms and the type `POS` may also be treated as `long`. The *Gentle* type `STRING` corresponds to `(char *)`.

A `'condition'` must be implemented as a function returning `int` that yields 1 if the call succeeds and 0 if it fails.

For example, a procedure introduced by

```
'condition' HasMeaning (IDENT -> MEANING)
```

may be implemented by

```
int HasMeaning (id, ref_meaning)
   IDENT id;
   long *ref_meaning;
{
   if (id->meaning == 0)
      return 0;
   *ref_meaning = id->meaning;
   return 1;
}
```

6 CASE STUDY

We present a guided tour through the compiler for a simple programming language. We first introduce the source language *MiniLAX* and the target instruction set *ICode*. Then we discuss the compiler.

MiniLAX and *ICode* were designed for classes on compiler construction. They have also been used as a basis for comparing compiler construction kits [13] (The presentations of *MiniLAX* and *ICode* were taken from [13]).

6.1 The Source Language

The programming language *MiniLAX* (Mini LAnguage eXample) is a *Pascal* relative. To be more specific, it is a subset of the example language *LAX* [14], which is used to illustrate problems in compiler construction. *MiniLAX* contains a carefully selected set of language concepts:

- types
- type coercion
- overloaded operators
- arrays
- procedures
- reference and value parameters
- nested scopes

Concepts with a low didactical value and concepts that would make the language unnecessary complex have been left out, along with "syntactic sugar".

6.1.1 Summary of the Language

A computer program consists of two essential parts, a description of actions which are to be performed, and a description of the data, which are manipulated by these actions. Actions are described by statements, and data are described by declarations.

The data are represented by constants and values of variables. Every variable occurring in a statement must be introduced by a variable declaration which associates an identifier and a data type with that variable. The data type essentially defines the set of values which may be assumed by that variable. The data type is directly described in the variable declaration.

There exist three basic types: Boolean, integer, and real. The values of the type Boolean are denoted by reserved identifiers, the numeric values are denoted by numbers.

Array types are defined by describing the types of their components and an integer range. A component of an array value is selected by an integer index. The type of the component is the component type of the corresponding array type.

The most fundamental statement is the assignment statement. It specifies that a newly computed value be assigned to a variable (or a component of a variable). The value is obtained by evaluating an expression. Expressions consist of variables, constants and operators operating on the denoted quantities and producing new values. *MiniLAX* defines a fixed set of operators, each of which can be regarded as describing a mapping from the operand types onto the result type. The set of operators is subdivided into

- Arithmetic operators: addition and multiplication
- Boolean operators: negation
- Relational operators: comparison

The result of a comparison is of type Boolean. The procedure statement causes the execution of the designated procedure (see below). Assignment and procedure statements are the components or building blocks of structured statements, which specify sequential, selective, or repeated execution of their components. Sequential execution of statements is specified by statement sequences, selective execution by the if statement, and repeated execution by the while statement. The if statement serves to make the execution of two alternative statements dependent on the value of a Boolean expression. The while statement serves to execute a statement while a Boolean expression is true.

A statement sequence can be given a name (identifier), and be referenced through that identifier. The statement sequence is then called a procedure, and its declaration a procedure declaration. Such a declaration may additionally contain a set of variable declarations and further procedure declarations. The variables and procedures thus declared can be referenced only within the procedure itself, and are therefore called local to the procedure. Their identifiers have significance only within the program text which constitutes the procedure declaration and which is called the scope of these identifiers. Since procedures may be declared local to other procedures, scopes may be nested. Entities which are declared in the main program, i.e. not local to some procedure, are called global. A procedure has a fixed number of parameters, each of which is denoted within the procedure by an identifier called the formal parameter. Upon an activation of the procedure statement, an actual quantity has to be indicated for each parameter, which can be referenced from within the procedure through the formal parameter. This quantity is called the actual parameter. There are two kinds of parameters: value parameters and variable parameters. In the first case, the actual parameter is an expression which is evaluated once. The formal parameter represents a local variable to which the result of this evaluation is assigned before the execution of the procedure. In the case of a variable parameter, the actual parameter is a variable, and the formal parameter stands for this variable. Possible indices are evaluated before execution of the procedure.

6.1.2 Notation, Terminology, and Vocabulary

The syntax is described in Extended Backus Naur Form. Syntactic constructs are denoted by (abbreviated) English words consisting of upper- and lower-case letters, and containing at least one lower-case letter. The angular brackets < and > are omitted. Strings of letters consisting solely of upper-case letters stand for themselves, i.e. for reserved identifiers of the language. Strings of characters enclosed in single quotes ' ' are also to be taken literally. Square brackets [ ] denote optional constructs. Curly brackets { } stand for zero or more repetitions of the enclosed construct. Alternative constructs are separated by a vertical bar |. Parentheses ( ) are used for grouping.

The basic vocabulary of *MiniLAX* consists of basic symbols classified into delimiters, identifiers, and constants.

Spaces, line ends, and comments may occur anywhere in a program except within a basic symbol. At least one space, line end or comment must occur between any two adjacent identifiers or constants. Otherwise, spaces, line ends, and comments do not influence the meaning of a program.

A comment has the form

```
'(*' any sequence of characters not containing *) '*)'
```

Delimiters

Delimiters are reserved identifiers or (strings of) special characters.

```
Delim ::= ':' | ';' | ':=' | '(' | ')' | '.' | ','
      | '..' | '[' | ']' | '+' | '*' | '<'
      | 'ARRAY' | 'BEGIN' | 'BOOLEAN' | 'DECLARE' | 'DO'
      | 'ELSE' | 'END' | 'FALSE' | 'IF' | 'INTEGER'
      | 'NOT' |'OF' | 'PROCEDURE' | 'PROGRAM' | 'READ'
      | 'REAL' |'THEN' | 'TRUE' | 'VAR' | 'WHILE'
      | 'WRITE'
```

Identifiers

Identifiers serve to denote variables and procedures. Their association must be unique within their scope of validity, i.e. within the procedure or function in which they are declared.

```
Id ::= Letter { Letter | Digit }
```

All letters and digits of an identifier are significant. Upper and lower case letters are distinguished. Delimiters are reserved identifiers that cannot be used otherwise.

Numbers

The usual decimal notation is used for numbers, which are the constants of the data types integer and real. The letter 'E' preceding the scale factor is pronounced as "'times"10

```
IntConst    ::= Digit { Digit }
RealConst   ::= [ IntConst ] '.' IntConst [ ScaleFactor ]
ScaleFactor ::= 'E' [ '+' | '-' ] IntConst
```

Examples:

```
1   100   .1   87.35E-8
```

6.1.3 Data Types

A data type determines the set of values which variables of that type may assume.

```
Type ::= SimpleType | ArrayType
```

Simple Types

```
SimpleType ::= 'INTEGER' | 'REAL' | 'BOOLEAN'
```

The values of type INTEGER are a subset of the whole numbers defined by individual implementations. Its values are the integers.

The values of type REAL are a subset of the real numbers depending on a particular implementation. The values are denoted by real numbers.

The values of type BOOLEAN are the truth values denoted by the reserved identifiers TRUE and FALSE.

Array Types

An array type is a structure consisting of a fixed number of components which are all of the same type, called the component type. The elements of the array are designated by integer indices. The array type specifies the component type as well as a subrange of the integers to be used as indices.

```
ArrayType ::= 'ARRAY' '[' IntConst '..' IntConst ']'
              'OF' Type
```

Examples:

```
ARRAY [1..100] OF INTEGER
ARRAY [4..7] OF ARRAY [2..2] OF BOOLEAN
```

The index range must contain at least one element, i.e. the lower bound of an index range must not exceed the upper bound.

6.1.4 Declarations and Denotations of Variables

Variable declarations consist of an identifier denoting the new variable, followed by its type.

```
VarDecl ::= Id ':' Type
```

Examples:

```
i: INTEGER
r: REAL
b: BOOLEAN
a: ARRAY [4..7] OF ARRAY [2..2] OF INTEGER
```

Denotations of variables either designate an entire variable or a component of an array variable. Variables occurring in examples in subsequent chapters are assumed to be declared as indicated above.

```
Var ::= Id | Var '[' Expr ']'
```

Examples:

```
i   a[4][2]
```

An entire variable is denoted by its identifier. A component of an array variable is denoted by the variable followed by an index expression. The value of the index expression must lie in the range of the indices of the corresponding array type.

6.1.5 Expressions

Expressions are constructs denoting rules of computation for obtaining values of variables and generating new values by the application of operators. Expressions consist of operators and operands, i.e. variables and constants.

The rules of composition specify operator precedences according to four classes of operators. The operator NOT has the highest precedence, followed by the multiplying operator '*', the adding operator '+', and finally, with the lowest precedence, the relational operator. Sequences of operators of the same precedence are executed from left to right.

```
Expr ::= Expr ( '+' | '*' | '<' ) Expr | 'NOT' Expr
     | '(' Expr ')' | Var | IntConst | RealConst
     | 'TRUE' | 'FALSE'
```

Examples:

```
i       15        TRUE       2*(i+r)
NOT b   NOT (i<1)
```

The operators are summarized in the following table:

Table of Operators

Priority	Operator	left Operand	right Operand	Result	Operation
4	NOT	BOOLEAN		BOOLEAN	negation
3	*	INTEGER	INTEGER	INTEGER	integer multiplication
		REAL	REAL	REAL	real multiplication
2	+	INTEGER	INTEGER	INTEGER	integer addition
		REAL	REAL	REAL	real addition
1	<	INTEGER	INTEGER	BOOLEAN	integer comparison
		REAL	REAL	BOOLEAN	real comparison
		BOOLEAN	BOOLEAN	BOOLEAN	boolean comparison

Note that, for Boolean values, FALSE < TRUE.

6.1.6 Statements

Statements denote algorithmic actions, and are said to be executable.

```
Stat ::= AssignStat | CondStat | LoopStat | ProcStat
```

Statement sequences

A statement sequence specifies that its component statements are to be executed in the same sequence as they are written.

```
StatSeq ::= Stat { ';' Stat }
```

Assignment Statements

The assignment statement serves to replace the current value of a variable by a new value specified as an expression.

```
AssignStat ::= Var ':=' Expr
```

Examples:

```
i := i+1
r := r*3.141592
b := i<1
a[4][2] := r
```

The variable and the expression must be of identical type, with the following exception being permitted: The type of the variable is REAL, and the type of the expression is INTEGER. In any case, the variable must be of a simple type.

Procedure Statements

A procedure statement serves to execute the procedure denoted by the procedure identifier. The procedure statement may contain a list of actual parameters which are substituted in place of their corresponding formal parameters defined in the procedure declaration. The correspondence is established by the positions of the parameters in the lists of actual and formal parameters respectively. There exist two kinds of parameters: value parameters and variable parameters.

In the case of a value parameter, the actual parameter must be an expression (of which a variable is a simple case). The corresponding formal parameter represents a local variable of the called procedure, and the current value of the expression is initially assigned to this variable. Value parameters must have a simple type. In the case of a variable parameter, the actual parameter must be a variable of the same type, and the corresponding formal parameter represents this actual variable during the entire execution of the procedure. If this variable is a component of an array, its index is evaluated when the procedure is called. A variable parameter must be used whenever the parameter represents a result of the procedure.

```
ProcStat ::= Id [ '(' Expr { ',' Expr } ')' ]
```

Examples:

```
next   Transpose(a,m,n)
```

Conditional Statements

The if statement specifies that a statement be executed only if a certain condition (Boolean expression) is true. If it is false, the statement following the delimiter ELSE is to be executed.

```
CondStat ::= 'IF' Expr 'THEN' StatSeq
             'ELSE' StatSeq 'END'
```

Examples:

```
IF i < 0 THEN i := 1 ELSE i := 2 END
```

The expression between the delimiters IF and THEN must be of type Boolean.

Repetitive Statements

The while statement specifies that a certain statement is to be executed repeatedly.

```
LoopStat ::= 'WHILE' Expr 'DO' StatSeq 'END'
```

The expression controlling repetition must be of type Boolean. The statement is repeatedly executed as long as the expression is true. If it evaluates to false at the beginning, the statement is not executed at all. The while statement

```
WHILE b DO s END
```

is equivalent to

```
IF b
THEN s; WHILE b DO s END
ELSE (* nothing *)
END
```

Examples:

```
WHILE a [i] < r DO i := i + 1 END
```

```
WHILE i < n DO
   r := 2 * r;
   i := i + 1
END
```

Declarations

Procedure declarations serve to define parts of programs and to associate identifiers with them so that they can be activated by procedure statements.

```
ProcDecl ::= ProcHead ';' Block
Block    ::= 'DECLARE' Decl { ';' Decl }
             'BEGIN' StatSeq 'END'
Decl     ::= VarDecl | ProcDecl
```

The procedure heading specifies the identifier naming the procedure and the formal parameter identifiers (if any). The parameters are either value or variable parameters.

```
ProcHead ::= 'PROCEDURE' Id
             [ '(' Formal { ';' Formal } ')' ]
Formal   ::= [ 'VAR' ] Id ':' Type
```

If a formal starts with the delimiter VAR it specifies a variable parameter, otherwise a value parameter.

The statement sequence of the block specifies the algorithmic actions to be executed upon an activation of the procedure by a procedure statement.

All identifiers introduced in the formal parameter part of the procedure heading and in the declaration part of the associated block are local to the procedure declaration which is called the scope of these identifiers. They are not known outside their scope. In the case of local variables, their values are undefined at the beginning of the statement part.

The use of the procedure identifier in a procedure statement within its declaration implies recursive execution of the procedure.

Examples of procedure declarations:

```
PROCEDURE ReadPosInteger (VAR i: INTEGER);
DECLARE
  j: INTEGER;
BEGIN
  i := 0;
  WHILE NOT (0 < i) DO READ (i) END
END

PROCEDURE Sort
  (VAR a: ARRAY [1..10] OF REAL; n: INTEGER);
DECLARE
  i: INTEGER; j: INTEGER; k: INTEGER; h: REAL;
BEGIN
  i := 1;
```

```
  WHILE i < n DO
    (* a [1], ... , a [i] is sorted *)
    j := i; k := i;
    WHILE j < n DO
      (* a [k] = min {a [i], ... , a [j]} *)
      j := j + 1;
      IF a [j] < a [k] THEN k := j ELSE k := k END
    END;
    h := a [i]; a [i] := a [k]; a [k] := h;
    i := i + 1
  END
END
```

6.1.7 Input and Output

Input and output of values of simple types is achieved by the standard procedures READ and WRITE.

The procedure READ takes one actual parameter which must be a variable of a simple type. It reads a value of the corresponding type from the standard input and assigns it to that variable.

The procedure WRITE takes one actual parameter which must be an expression with a simple type. It writes the value of that expression onto the standard output.

Example:

```
(* read integers and write            *)
(* until a nonpositive number is read *)
READ (i);
WHILE 0 < i DO
  WRITE (i); READ (i)
END
```

6.1.8 Programs

A *MiniLAX* program has the form of a procedure declaration except for its heading.

```
Program ::= 'PROGRAM' Id ';' Block '.'
```

The identifier following the symbol PROGRAM is the program name; it has no further significance inside the program.

Example:

```
PROGRAM test;

  (* read, sort and write an array of n numbers     *)
  (* this program shows the following features:     *)
  (*   procedure calls from main level, to a local, *)
  (*     and to a global procedure                  *)
  (*   access to a global array                     *)
  (*   access to local, global and                  *)
  (*     intermediate variables                     *)
  (*   recursion                                    *)
  (*   reading and writing of all types             *)
  (*   integer to real conversion                   *)

DECLARE
  test : BOOLEAN;
  n    : INTEGER;
  a    : ARRAY [1..100] OF REAL;

  PROCEDURE skip; (* do nothing *)
  DECLARE
    n: INTEGER
  BEGIN
    n := n
  END;

  PROCEDURE read
     (VAR n: INTEGER; VAR a: ARRAY [1..100] OF REAL);
  DECLARE
    i: INTEGER
  BEGIN
    WRITE (TRUE); READ (test);
    WRITE (5); READ (n);
    i := 1;
    WHILE i < n DO
      i := i + 1; WRITE (1.0E-7); READ (a [i])
    END
  END;

  PROCEDURE write (m: INTEGER); (* write a [m..n] *)
  DECLARE
    x: INTEGER
  BEGIN
    WRITE (a [m]);
    IF m < n THEN write (m + 1) ELSE skip END
```

```
END;

PROCEDURE sort (VAR a: ARRAY [1..100] OF REAL);
  (* sort a [1..n] *)
DECLARE
  i : INTEGER;
  j : INTEGER;
  k : INTEGER;
  h : REAL;
  ok: BOOLEAN;

  PROCEDURE check (VAR ok: BOOLEAN);
    (* check order of a [1..n] *)
  DECLARE
    continue: BOOLEAN
  BEGIN
    IF test THEN write (1) ELSE skip END;
    i := 1; continue := TRUE;
    WHILE continue DO
      IF i < n THEN
        continue := NOT (a [i + 1] < a [i]);
        IF continue THEN i := i + 1 ELSE skip END
      ELSE
        continue := FALSE
      END
    END;
    ok := NOT (i < n)
  END

BEGIN (* sort *)
  i := 1;
  WHILE i < n DO
    write (1);
    j := i; k := i;
    WHILE j < n DO   (* a [k] = MIN a [i..j] *)
      j := j + 1;
      IF a [j] < a [k] THEN k := j ELSE skip END
    END;
    h := a [i]; a [i] := a [k]; a [k] := h;
    i := i + 1
  END;
  check (ok); WRITE (ok)
END
```

```
BEGIN (* main program *)
  a [1] := 2.1415926536;
  a [1] := a [1] + 1.0;
  read (n, a);
  sort (a);
  IF NOT test THEN write (0) ELSE skip END
END.
```

6.2 The Target Machine

The intermediate code (*ICode*) for *MiniLAX* is a subset of the intermediate code for *Pascal* (*P-Code*) [9]. *ICode* programs consist of simple instructions for a hypothetical computer - a stack machine.

6.2.1 The ICode Machine

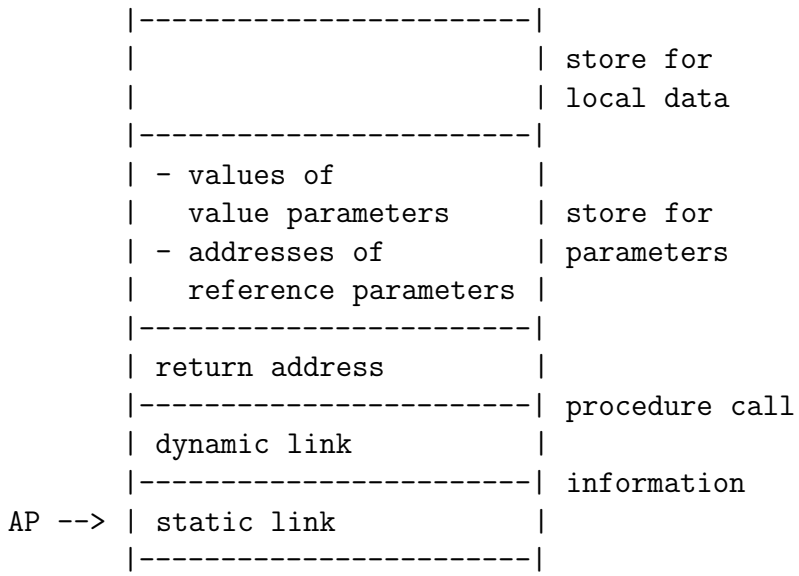
The *ICode* Machine consists of three registers and memory. The registers are

- PC the program counter
- SP the stack pointer
- AP the activation record pointer

The program counter points to the current instruction in the memory. The stack pointer points to the highest occupied stack cell. The activation record pointer points to the 'static link' field of the current activation record.

The memory is divided in two parts, one containing the program (Code) and the other containing data (Store). Code is an array of *ICode* instructions. Store is organized as stack (growing upwards) which contains the data of the program executed. Each activation of a procedure results in pushing an activation record on the stack, which contains storage for parameters and local data.

An activation record has the following layout:



At initialization time, the static and dynamic links and the return address of the main program are all set to 0. The registers are initialized as follows: PC := 0, SP := 3, and AP := 1. The start address is 0, i.e. Code [0] contains the first *ICode* instruction to be executed. PC is incremented before the according instruction is executed. The interpreter stops at return from the main program. The stop condition is: (PC = 0).

A procedure call enforces

- the creation of static and dynamic links of the new activation record (*ICode* instruction: MST)
- parameter passing: The values of value parameters and the addresses of reference parameters are evaluated and pushed on the stack.
- storing the return address and a jump to the procedure (*ICode* instruction: JSR)
- reservation of store for local data of the new activation record (*ICode* instruction: ENT)

A return from a procedure enforces

- discarding the current activation record by updating the registers

6.2.2 ICode Instructions

For each *ICode* instruction its operation code, its parameters and its meaning are given in the following. The meaning is given as text and as formula which describe operations

on the runtime stack. To simplify the description, within formulas it is not taken care about the types of the stack elements.

If not further mentioned, the operations apply to the top of the stack, which contains the actual element. The following shorthand notations are used:

S	runtime stack
base(P)	returns a pointer to the P'th static predecessor of the current activation record

An instruction may have up to two parameters with the following meaning:

o	offset
c, c1, c2	constants
a	address (index of code section)
t	indicates type integer (1), real (2) or boolean (3)
l	block level difference between current and referenced activation record

Note: The types integer, real and boolean are encoded with 1, 2, and 3. The boolean values FALSE and TRUE are encoded by 0 and 1.

Load Instructions

LDA l o

load address with base and offset

```
SP:=SP+1
S[SP]:=base(l)+o;
```

LDC t c

load constant c of type t

```
SP:=SP+1;
S[SP]:=c;
```

LDI

load indirect

```
S[SP]:=S[S[SP]]
```

Store Instructions

STI

store into address contained in the element below the top

```
S[S[SP-1]]:=S[SP];
SP:=SP-1;
```

Jump Instructions

JMP a
unconditional jump

```
PC:=a;
```

FJP a
conditional jump

```
if not S[SP] then PC:=a;
SP:=SP-1;
```

Arithmetic Instructions

ADD t
addition of type t

```
SP:=SP-1;
S[SP]:=S[SP]+S[SP+1];
```

SUB
integer subtraction

```
SP:=SP-1;
S[SP]:=S[SP]-S[SP+1];
```

MUL t
multiplication of type t

```
SP:=SP-1;
S[SP]:=S[SP]*S[SP+1];
```

Logic Instructions

INV

```
S[SP]:=not S[SP];
```

LES t
less operation of type t

```
SP:=SP-1;
S[SP]:=S[SP]<S[SP+1];
```

Address Calculation Instructions

IXA c"

compute indexed address

```
SP:=SP-1;
S[SP]:=c*S[SP+1]+S[SP];
```

Convert Instructions

`FLT`

converts from integer to real

```
S[SP]:=real(S[SP]);
```

Input-Output Instructions

`WRI t`

```
write(S[SP]);
SP:=SP-1;
```

`REA t`

```
SP:=SP+1;
read(S[SP]);
```

Subroutine Handling Instructions

`MST l`

activation record initialization:

```
S[SP+1]:=base(l);- store static predecessor
S[SP+2]:=AP; - store dynamic predecessor
SP:=SP+3; - return address (=S[SP+3]) is stored by JSR
```

`JSR o a`

set AP to point to new activation record
o = number of locations for parameters

```
AP:=SP-(o+2);
S[AP+2]:=PC; - store return address
PC:=a ; - set PC to first instruction of subroutine
```

`ENT o`

storage reservation for new block
o = length of local data segment

```
SP:=SP+o
```

`RET`

return from subroutine:

```
SP:=AP-1;
PC:=S[SP+3]; - fetch return address to restore PC
AP:=S[SP+2]; - restore activation record pointer AP
```

Check Instructions

`CHK c1 c2`

check against upper and lower bounds

```
if (S[SP]<c1) or (S[SP]>c2) then error
```

6.3 The Compiler

We now present a compiler for *MiniLAX*.

This section contains the complete *Gentle* specification for *MiniLAX*.

6.3.1 Overall Structure

The task of a compiler is to translate a program written in a source language into a semantically equivalent program in a target language.

This task can be decomposed into subtasks

- Discover the structure of the source program.
- Process this structure to generate the target program.

In *Gentle* this can be written as

```
Program(-> P) Translate(P)
```

Here, `Program` reads the source program and returns a structured representation (the "abstract syntax") in its output parameter `P`. This representation is then processed by `Translate`, which is invoked with `P` as an input parameter.

Consider the example *MiniLAX* program in *Fig. 6.1*.

```
PROGRAM test;

DECLARE
  i: INTEGER
BEGIN
  i := 0
END.
```

Fig. 6.1 Example Program in MiniLAX

For this program the predicate `Program` will deliver a value of `P` as shown in *Fig. 6.2*.

```
dcl(
   <identifier "test">,
   proc(
      nil,
      decllist(
         dcl( <identifier "i">, variable( integer), <line 4, col 3> ),
         nil
      ),
      assign(
         id( <identifier "i">, <line 6, col 3> ),
         int( 0 ),
         <line 6, col 5>
      )
   ),
   <line 1, col 9>
)
```

Fig. 6.2 Example Program in Abstract Syntax

This will be translated by `Translate` into the *ICode* program shown in *Fig. 6.3*.

```
0:   ENT    1
1:   LDA    0    3
2:   LDC    1    0
3:   STI
4:   RET
```

Fig. 6.3 Example Program in ICode

The *abstract syntax* representation of a prgram is a structured value (a "term") that expresses the hierarchical composition of the program. Abstract syntax ignores syntactic sugar and concrete tokens, that only serve to make the input unambiguous. It merely specifies which alternative was used for a construct and what its constituents were.

A term type is introduced by a type declaration. For example, the following declaration introduces the abstract syntax of statements:

```
'type' STMT
   assign(DESIG, EXPR, POS)
   read(DESIG, POS)
   write(EXPR, POS)
   call(IDENT, EXPRLIST, POS)
   if(EXPR, STMT, STMT, POS)
   while(EXPR, STMT, POS)
   seq(STMT, STMT)
```

These alternatives serve as templates for values: If `E` is an `EXPR`, `S` an `STMT`, and `Pos` the source coordinate of the phrase, then

```
while (E, S, Pos)
```

is a value of type `STMT`.

Concrete syntax is described by grammar rules, e.g.

```
'rule' Stat(-> while(E, S, Pos)) :
   "WHILE" Expr(-> E) @(-> Pos) "DO" StatSeq(-> S) "END"
```

This rule parses a phrase

```
WHILE Expr DO StatSeq END
```

`E` is the abstract syntax of `Expr`, `S` the abstract syntax of `StatSeq`, and `Pos` the source coordinate of `Expr`. From these values, the abstract syntax of the phrase is constructed:

```
while (E, S, Pos)
```

Translation is described by action rules, e.g.

```
'rule' Statement(while(E, S, Pos)) :
    NewLabel(-> L1)
    NewLabel(-> L2)
    JMP(L2)
    LAB(L1)
    Statement(S)
    LAB(L2)
    Expression(E -> T)
    CheckBool(T, Pos)
    INV
    FJP(L1)
```

This rule unparses the term

```
while (E, S, Pos)
```

thereby emitting the target code

```
    JMP L2
L1: Statement
L2: Expression
    INV
    FJP L1
```

where `Statement` is the code for `S`, and `Expression` is the code for `E`.

When the source program is translated into an internal representation, rules are selected by the parser according to the given source. Consider the rules

```
'rule' Stat(-> read(V, Pos)) :
   "READ" "(" Desig(-> V) @(-> Pos) ")"
'rule' Stat(-> write(E, Pos)) :
   "WRITE" "(" Expr(-> E) @(-> Pos) ")"
```

If a statement has the form `READ(Desig)`, the first rule is selected and the internal form is `read(V,Pos)`. If a statement has the form `WRITE(Expr)`, the second rule is selected and the internal form is `write(E,Pos)`.

When the internal representation is processed, rules are selected according to the given term. Consider the rules

```
'rule' Statement(read(V, Pos)) :
   Designator(V -> T)
   CheckSimple(T, Pos)
   TypeCode(T -> N)
   REA(N)
   STI
'rule' Statement(write(E, Pos)) :
   Expression(E -> T)
   CheckSimple(T, Pos)
   TypeCode(T -> N)
   WRI(N)
```

If the predicate `Statement` is called with the term `read(V, Pos)`, the first rule is selected and corresponding target code for reading an item is emitted. If the predicate is called with `write(E, Pos)`, the the second rule is selected and emits code to write an item.

The overall organization of the compiler is expressed by the *root clause*, which is elaborated by executing its statements. Our specification starts with the root clause:

```
001  'root'
002     Program(-> P) Translate(P)
```

Now we have to define the abstract syntax and have to specify how a concrete source program is mapped to the abstract representation (`Program`) and how this representation is processed to produce the target program (`Translate`).

6.3.2 Abstract Syntax

Here is the definition of the abstract representation of *MiniLAX* programs:

```
003  'type' DECL
004     dcl(IDENT, DEF, POS)

005  'type' DECLLIST
006     decllist(DECL, DECLLIST)
007     nil

008  'type' DEF
009     variable(TYPE)
010     valueparam(TYPE)
011     varparam(TYPE)
012     proc(Formals: DECLLIST, Locals: DECLLIST, STMT)

013  'type' TYPE
014     integer
015     real
016     boolean
017     array(INT, INT, TYPE)
018     none

019  'type' STMT
020     assign(DESIG, EXPR, POS)
021     read(DESIG, POS)
022     write(EXPR, POS)
023     call(IDENT, EXPRLIST, POS)
024     if(EXPR, STMT, STMT, POS)
025     while(EXPR, STMT, POS)
026     seq(STMT, STMT)

027  'type' DESIG
028     id(IDENT, POS)
029     subscr(DESIG, EXPR, POS)
030     none

031  'type' EXPR
032     binary(OP, EXPR, EXPR, POS)
033     opnot(EXPR, POS)
034     int(INT)
035     float(FLOAT)
036     true
037     false
038     desig(DESIG)
```

```
039  'type' EXPRLIST
040     exprlist(EXPR, EXPRLIST, POS)
041     nil(POS)

042  'type' OP
043     less
044     plus
045     mult
```

6.3.3 Concrete Syntax

Here is the concrete grammar of *MiniLAX* and the mapping to abstract syntax:

```
046  'nonterm' Program(-> DECL)
047     'rule' Program(-> dcl(I, proc(nil, Ds, S), Pos)) :
048        "PROGRAM" Ident(-> I) @(-> Pos) ";"
049        "DECLARE" DeclList(-> Ds) "BEGIN" StatSeq(-> S) "END" "."

050  'nonterm' DeclList(-> DECLLIST)
051     'rule' DeclList(-> decllist(D, Ds)) :
052        Decl(-> D) ";" DeclList(-> Ds)
053     'rule' DeclList(-> decllist(D, nil)) :
054        Decl(-> D)

055  'nonterm' Decl(-> DECL)
056     'rule' Decl(-> dcl(I, variable(T), Pos)) :
057        Ident(-> I) @(-> Pos) ":" Type(-> T)
058     'rule' Decl(-> dcl(I, proc(Fs, Ds, S), Pos)) :
059        "PROCEDURE" Ident(-> I) @(-> Pos) FormalPart(-> Fs) ";"
060        "DECLARE" DeclList(-> Ds)
061        "BEGIN" StatSeq(-> S) "END"

062  'nonterm' Type(-> TYPE)
063     'rule' Type(-> integer) :
064        "INTEGER"
065     'rule' Type(-> real) :
066        "REAL"
067     'rule' Type(-> boolean) :
068        "BOOLEAN"
```

```
069     'rule' Type(-> array(Lwb, Upb, T)) :
070       "ARRAY" "[" Number(-> Lwb) ".." Number(-> Upb) "]"
071       "OF" Type(-> T)

072   'nonterm' FormalPart(-> DECLLIST)
073     'rule' FormalPart(-> nil) :
074     'rule' FormalPart(-> Fs) :
075       "(" FormalList(-> Fs) ")"

076   'nonterm' FormalList(-> DECLLIST)
077     'rule' FormalList(-> decllist(F, nil)) :
078       Formal(-> F)
079     'rule' FormalList(-> decllist(F, Fs)) :
080       Formal(-> F) ";" FormalList(-> Fs)

081   'nonterm' Formal(-> DECL)
082     'rule' Formal(-> dcl(I, varparam(T), Pos)) :
083       "VAR" Ident(-> I) ":" @(-> Pos) Type(-> T)
084     'rule' Formal(-> dcl(I, valueparam(T), Pos)) :
085       Ident(-> I) ":" @(-> Pos) Type(-> T)

086   'nonterm' StatSeq(-> STMT)
087     'rule' StatSeq(-> S) :
088       Stat(-> S)
089     'rule' StatSeq(-> seq(S1, S2)) :
090       StatSeq(-> S1) ";" Stat(-> S2)

091   'nonterm' Stat(-> STMT)
092     'rule' Stat(-> assign(V, E, Pos)) :
093       Desig(-> V) ":=" @(-> Pos) Expr(-> E)
094     'rule' Stat(-> read(V, Pos)) :
095       "READ" "(" Desig(-> V) @(-> Pos) ")"
096     'rule' Stat(-> write(E, Pos)) :
097       "WRITE" "(" Expr(-> E) @(-> Pos) ")"
098     'rule' Stat(-> call(I, Es, Pos)) :
099       Ident(-> I) "(" ExprList(-> Es) ")" @(-> Pos)
100     'rule' Stat(-> call(I, nil(Pos), Pos)) :
101       Ident(-> I) @(-> Pos)
102     'rule' Stat(-> if(E, S1, S2, Pos)) :
103       "IF" Expr(-> E) @(-> Pos) "THEN" StatSeq(-> S1)
104       "ELSE" StatSeq(-> S2) "END"
105     'rule' Stat(-> while(E, S, Pos)) :
106       "WHILE" Expr(-> E) @(-> Pos) "DO" StatSeq(-> S) "END"
```

```
107  'nonterm' Desig(-> DESIG)
108     'rule' Desig(-> id(I, Pos)) :
109        Ident(-> I) @(-> Pos)
110     'rule' Desig(-> subscr(V, E, Pos)) :
111        Desig(-> V) "[" Expr(-> E) "]" @(-> Pos)

112  'nonterm' ExprList(-> EXPRLIST)
113     'rule' ExprList(-> exprlist(E, nil(Pos), Pos)) :
114        Expr(-> E) @(-> Pos)
115     'rule' ExprList(-> exprlist(E, Es, Pos)) :
116        Expr(-> E) @(-> Pos) "," ExprList(-> Es)

117  'nonterm' Expr(-> EXPR)
118     'rule' Expr(-> binary(less, X, Y, Pos)) :
119        Expr(-> X) "<" @(-> Pos) Expr2(-> Y)
120     'rule' Expr(-> X) :
121        Expr2(-> X)

122  'nonterm' Expr2(-> EXPR)
123     'rule' Expr2(-> binary(plus, X, Y, Pos)) :
124        Expr2(-> X) "+" @(-> Pos) Expr3(-> Y)
125     'rule' Expr2(-> X) :
126        Expr3(-> X)

127  'nonterm' Expr3(-> EXPR)
128     'rule' Expr3(-> binary(mult, X, Y, Pos)) :
129        Expr3(-> X) "*" @(-> Pos) Expr4(-> Y)
130     'rule' Expr3(-> X) :
131        Expr4(-> X)

132  'nonterm' Expr4(-> EXPR)
133     'rule' Expr4(-> opnot(X, Pos)) :
134        "NOT" @(-> Pos) Expr4(-> X)
135     'rule' Expr4(-> X) :
136        "(" Expr(-> X) ")"
137     'rule' Expr4(-> desig(D)) :
138        Desig(-> D)
139     'rule' Expr4(-> int(N)) :
140        Number(-> N)
141     'rule' Expr4(-> float(F)) :
142        Float(-> F)
143     'rule' Expr4(-> true) :
144        "TRUE"
145     'rule' Expr4(-> false) :
```

```
146        "FALSE"

147  'token' Ident(-> IDENT)
148  'token' Number(-> INT)
149  'token' Float(-> FLOAT)
```

The tokens `Ident`, `Number`, and `REALCONST` are specified by *token description files* outside the *Gentle* specification. (These files are processed by the tool *Reflex* and integrated into a specification for *Lex*.)

For example, here is the description for `Ident`:

```
[A-Za-z][A-Za-z0-9]* {
   long id;
   string_to_id (yytext, &id);
   yylval.attr[1] = id;
   yysetpos();
   return Ident;
}
```

This rule gives a regular expression that matches identifiers and an action that computes its attributes. The type `IDENT`, which is used to maintain identifiers (see the discussion of scope handling below), is defined by a module of the *Gentle* library. The library predicate `string_to_id` converts the recognized input (`yytext`) into a value of type `IDENT`. This can later be used to attach a "meaning" to the identifier.

6.3.4 Translating Procedures

We have just described how a source program is translated into an internal representation. We now discuss how to translate the internal representation into a target program.

```
150  'action' Translate(Program: DECL)
151     'rule' Translate(P):
152        SetCurrentNesting(0)
153        Procedure(P)
154        Emit

155  'action' Procedure(Proc: DECL)
156     'rule' Procedure(dcl(Ident, proc(Formals,Locals,Body), Pos)) :
157        GetCurrentNesting(-> OuterLevel)
158        SetCurrentNesting(OuterLevel+1)
```

```
159        DeclareList(Formals, 3 -> SizeFormals)
160        DeclareList(Locals, SizeFormals+3 -> SizeLocals)
161        ENT(SizeLocals)
162        Statement(Body)
163        RET
164        LocalProcedures(Locals)
165        UndeclareList(Locals)
166        UndeclareList(Formals)
167        SetCurrentNesting(OuterLevel)

168  'action' LocalProcedures(Decls: DECLLIST)
169     'rule' LocalProcedures(decllist(Head, Tail)) :
170        LocalProcedure(Head)
171        LocalProcedures(Tail)
172     'rule' LocalProcedures(nil)

173  'action' LocalProcedure(DECL)
174     'rule' LocalProcedure(dcl(Ident, proc(Fs, Ds, S), Pos)) :
175        HasMeaning (Ident -> object(procobj(Start,_),_,_))
176        LAB(Start)
177        Procedure(dcl(Ident, proc(Fs, Ds, S), Pos))
178     'rule' LocalProcedure(Decl) :
```

`Translate`

The predicate `Translate(Program)` is invoked in the root clause of the specification. The whole program is merely represented as a simple procedure without parameters. Hence, we can simply use `Procedure` to process the given abstract syntax tree. Before calling `Procedure`, we initialize the current nesting level of procedures (which is implemented as a global variable). After generating the code, we invoke `emit` to write it to the target file.

`Procedure`

The predicate `Procedure(Proc)` handles a single procedure declaration the abstract syntax of which is

```
dcl(Ident, proc(Formals, Locals, Body), Pos)
```

The procedure has a name given by `Ident`; formal parameters and local declarations are given by `Formals` and `Locals`; the procedure body is given by `Body`.

First, the current procedure nesting is updated accordingly. It is reset to its old value at the end of the rule.

During the processing of `Body` and local procedures, the formal parameters and the local declarations must be accessible. Hence, they are made visible by processing `Formals` and `Locals` with the predicate `DeclareList`. Outside the given procedure, the formal parameters and local declarations cannot be used. Hence, they are made invisible again (`UndeclareList`) after processing the body and the local procedures.

`DeclareList` (to be discussed later) has additional parameters: `DeclareList(X, Loc -> Size)` indicates that the items in `X` are stored from location `Loc` and that they occupy `Size` units.

The stack frame of the procedure is organized as follows: 3 units for administrative data (static link, dynamic link, return address), `SizeFormal` units as to store parameters, and `SizeLocals` units to store local variables.

The target code of the procedure is given as follows:

```
ENT(SizeLocals)
Code of body
RET
Code of local procedure
```

The code for the body is generated by the predicate `Statement`, the code for the local procedures is generated by `LocalProcedures`.

`LocalProcedures`

`LocalProcedures(Decls)` invokes `LocalProcedure` for each element of the declaration list(`Decls`).

`LocalProcedure`

If `LocalProcedure(Decl)` is invoked with a procedure declaration, it processes this declaration. Otherwise the passed element is skipped. A procedure is processed by `Procedure` (in the case of a local procedure, we first emit a start label).

We now refine the processing of procedure bodies. We discuss statements, expressions, and designators.

6.3.5 Translating Statements

Here is the specification for translating statements:

```
179  'action' Statement(Stmt: STMT)
180     'rule' Statement(assign(V, E, Pos)) :
181        Designator(V -> TV)
182        Expression(E -> TE)
```

```
183       Assign(TV, TE, Pos)
184    'rule' Statement(read(V, Pos)) :
185       Designator(V -> T)
186       CheckSimple(T, Pos)
187       TypeCode(T -> N)
188       REA(N)
189       STI
190    'rule' Statement(write(E, Pos)) :
191       Expression(E -> T)
192       CheckSimple(T, Pos)
193       TypeCode(T -> N)
194       WRI(N)
195    'rule' Statement(call(Ident, Actuals, Pos)) :
196       Apply(Ident, Pos -> Obj)
197       CheckProcedure(Obj, Pos -> Formals, Level, Start)
198       GetCurrentNesting(-> CurLev)
199       MST(CurLev-Level)
200       ParamList(Formals, Actuals  -> Size)
201       JSR(Size, Start)
202    'rule' Statement(if(E, S1, S2, Pos)) :
203       Expression(E -> T)
204       CheckBool(T, Pos)
205       NewLabel(-> L1)
206       NewLabel(-> L2)
207       FJP(L1)
208       Statement(S1)
209       JMP(L2)
210       LAB(L1)
211       Statement(S2)
212       LAB(L2)
213    'rule' Statement(while(E, S, Pos)) :
214       NewLabel(-> L1)
215       NewLabel(-> L2)
216       JMP(L2)
217       LAB(L1)
218       Statement(S)
219       LAB(L2)
220       Expression(E -> T)
221       CheckBool(T, Pos)
222       INV
223       FJP(L1)
224    'rule' Statement(seq(S1, S2)) :
225       Statement(S1)
226       Statement(S2)
```

```
227  'action' Assign(LhsType: TYPE, RhsType: TYPE, Pos: POS)
228     'rule' Assign(integer, integer, Pos) : STI
229     'rule' Assign(real, integer, Pos) : FLT STI
230     'rule' Assign(real, real, Pos) : STI
231     'rule' Assign(boolean, boolean, Pos) : STI
232     'rule' Assign(T1, T2, Pos) :
233        Error("Invalid types in assignment", Pos)

234  'action' ParamList(Formals: DECLLIST, Actuals: EXPRLIST
235           -> Size: INT)
236     'rule' ParamList(decllist(dcl(Id,D,_), Fs), exprlist(E,Es,Pos)
237            -> S+1) :
238        Param(D, E, Pos)
239        ParamList(Fs, Es -> S)
240     'rule' ParamList(nil, nil -> 0)
241     'rule' ParamList(decllist(D, Fs), nil(Pos) -> 0) :
242        Error("Too few actual parameters", Pos)
243     'rule' ParamList(nil, exprlist(E, Es, Pos) -> 0) :
244        Error("Too many actual parameters", Pos)

245  'action' Param(Formal: DEF, Actual: EXPR, Pos: POS)
246     'rule' Param(valueparam(FType), Actual, Pos) :
247        Expression(Actual -> AType)
248        CheckEquiv(FType, AType, Pos)
249     'rule' Param(varparam(FType), Actual, Pos) :
250        CheckDesignator(Actual, Pos -> D)
251        Designator(D -> AType)
252        CheckEquiv(FType, AType, Pos)
```

`Statement`

The predicate `Statement(Stmt)` analyzes and translates statements. For each alternative of the abstract syntax, there is a separate rule.

As an example, consider the treatment of *if-statements* that are represented by

```
if(E, S1, S2, Pos)
```

The code for this construct is

```
code for expression E
FJP(L1)
code for statement S1
```

```
JMP(L2)
LAB(L1)
code for statement S2
LAB(L2)
```

where L1 and L2 are two unique labels. First, the expression is evaluated. If this yields false, the FJP instruction jumps to L2. Otherwise, the statement S1 is executed. After that, we jump over the code for S2. The code for S2 is preceded by a label S2, which is the target of FJP.

New labels are created by NewLabel. The predicate Expression not only generates codes for its argument E, but also computes the type T of E. In an if-statement this type must be bool. This is checked by CheckBool.

Hence, the rule for if is

```
'rule' Statement(if(E, S1, S2, Pos)) :
   Expression(E -> T)
   CheckBool(T, Pos)
   NewLabel(-> L1)
   NewLabel(-> L2)
   FJP(L1)
   Statement(S1)
   JMP(L2)
   LAB(L1)
   Statement(S2)
   LAB(L2)
```

An *assignment*

```
assign(V, E, Pos)
```

is compiled into code for the designator V and code for the expression E.

After evaluation of these code sequences, the stack top comprises the address given by V and the value of E. An instruction STI is used to store the value at the given address and to remove the two items from the stack. The types TV and TE of V and E must be equal and scalar. In addition, TV may be real and TE may be int. Then the STI must be preceeded by an FLT instruction. This check and the generation of FLT and STI are expressed by the predicate Assign.

The code for a *procedure call* is

```
MST(CurLev-Level)
code for parameters
JSR(Size, Start)
```

where `CurLev` is the current nesting level and the procedure has been declared at level `Level` and with start label `Start`. The code for parameters is generated by `ParamList`.

`Assign`

`Assign(LhsType, RhsType, Pos)` checks whether a value of type `RhsType` can be assigned to a designator of type `LhsType`. If so, it emits the corresponding instruction(s).

`ParamList`

The code for parameter passing is constructed by the predicate `ParamList(Formals, Actuals -> Size)` which processes the lists of formal and actual parameters in parallel.

`Param`

A pair of formal and actual parameters is handled by `Param(Formal, Actual, Pos)`.

6.3.6 Translating Expressions

Here is the specification for translating expressions:

```
253  'action' Expression(Expr: EXPR -> Type: TYPE)
254     'rule' Expression(binary(Op, X, Y, Pos) -> T) :
255        Expression(X -> TX)
256        Expression(Y -> TY)
257        BinaryOp(Op, TX, TY, Pos -> T)
258     'rule' Expression(opnot(X, Pos) -> TX) :
259        Expression(X -> TX)
260        CheckBool(TX, Pos)
261     'rule' Expression(int(N) -> integer) :
262        LDC(1, N)
263     'rule' Expression(float(Float) -> real) :
264        LDF(Float)
265     'rule' Expression(true -> boolean) :
266        LDC(3, 1)
267     'rule' Expression(false -> boolean) :
268        LDC(3, 0)
269     'rule' Expression(desig(D) -> T) :
270        Designator(D -> T)
271        LDI

272  'action' BinaryOp(Op: OP, T1: TYPE, T2: TYPE, Pos: POS -> T: TYPE)
273     'rule' BinaryOp(less, integer, integer, Pos -> boolean) : LES(1)
274     'rule' BinaryOp(less, real, real, Pos -> boolean) : LES(2)
```

```
275     'rule' BinaryOp(less, boolean, boolean, Pos -> boolean) : LES(3)
276     'rule' BinaryOp(plus, integer, integer, Pos -> integer) : ADD(1)
277     'rule' BinaryOp(mult, integer, integer, Pos -> integer) : MUL(1)
278     'rule' BinaryOp(plus, real, real, Pos -> real) : ADD(2)
279     'rule' BinaryOp(mult, real, real, Pos -> real) : MUL(2)
280     'rule' BinaryOp(Op, T1, T2, Pos -> none) :
281        Error("Invalid types for operator", Pos)
```

Expression

The predicate `Expression(Expr -> Type)` translates an expression `E` and computes its type `T`.

An expression that is simply a literal is translated into an instruction that pushes the value onto the stack.

An expression involving an operator is translated into code for the operands, followed by an instruction depending on the operator. In the case of binary operators, this instruction is selected by the predicate `BinaryOp`.

If the operand is a designator (a construct that designates a variable or an array element), then code to compute the address is generated, followed by a load instruction (`LDI`).

BinaryOp

In the case of binary operators the corresponding instruction is selected by `BinaryOp(Op, T1, T2, Pos -> T)`, which also checks the types of the arguments and yields the type of the result.

6.3.7 Translating Designators

Here is the specification for translating designators:

```
282  'action' Designator(Desig: DESIG -> Type: TYPE)
283     'rule' Designator(id(Ident, Pos) -> T) :
284        Apply(Ident, Pos -> Obj)
285        Access(Obj, Pos -> T)
286     'rule' Designator(subscr(Array, Index, Pos) -> T) :
287        Designator(Array -> TArray)
288        Expression(Index -> TIndex)
289        CheckArrayType(TArray, Pos -> Lwb, Upb, T)
290        TypeSize(T -> Size)
291        CheckInt(TIndex, Pos)
```

```
292         CHK(Lwb, Upb)
293         LDC(1, Lwb)
294         SUB
295         IXA(Size)

296   'action' Access(Obj: OBJ, Pos: POS -> Type: TYPE)
297     'rule' Access(object(varobj(Offset, Type), Level, Hidden), Pos
298             -> Type) :
299         GetCurrentNesting(-> CurLev)
300         LDA(CurLev-Level, Offset)
301     'rule' Access(object(varparamobj(Offset,Type), Level, Hidden),
302             Pos -> Type) :
303         GetCurrentNesting(-> CurLev)
304         LDA(CurLev-Level, Offset)
305         LDI
306     'rule' Access(object(valueparamobj(Offset,Type),Level,Hidden),
307             Pos -> Type) :
308         GetCurrentNesting(-> CurLev)
309         LDA(CurLev-Level, Offset)
310     'rule' Access(object(procobj(_,_),_,_), Pos -> none) :
311         Error("procedure not allowed here", Pos)
```

`Designator`

`Designator(Desig -> Type)` generates code for the designator `Desig` (a construct yielding an address) and computes its type `Type`.

If the designator is a simple identifier, we look up its definition and use `Access` to determine the code and type.

`Access`

`Access(Obj, Pos -> Type)` generates the code to access the object `Obj` and return its type. By way of an example, consider the rule that handles variables:

```
'rule' Access(object(varobj(Offset, Type), Level, Hidden), Pos
             -> Type) :
   GetCurrentNesting(-> CurLev)
   LDA(CurLev-Level, Offset)
```

The object has the offset `Offset` and the type `Type` ; it was declared on nesting level `Level`. This results in an instruction

```
   LDA CurLev-Level, Offset
```

where `CurLevel` is the nesting level of the location where the object is used.

The representation of objects is discussed next.

6.3.8 Scope Handling

We now discuss *scope handling*, a topic that has to be treated in most compilers.

While in most cases the translation of a phrase can be constructed from the translation of its constituents, the meaning of identifiers depends on the context.

The *Gentle* library provides the type `IDENT` and predicates for defining and querying the meaning of `IDENT` values.

In order to allow fast access, `IDENT` values are not merely strings that stand for identifiers, but references into a "symbol table". The library module `idents` provides a procedure `string_to_id` for converting a string into an `IDENT` value. This procedure is used in the token description for identifiers.

Given a value `Id` of type `IDENT`

```
DefMeaning(Id, M)
```

associates a meaning `M` with `Id`.

```
GetMeaning(Id -> M)
```

later returns the meaning `M` associated with `Id`. `GetMeaning` is defined as a *condition*: it signals *failure* if there was no previous `DefMeaning` for `Id`. Thus it can be used to check whether an identifier is undeclared.

Here is the signature of these predicates (`IDENT` is an abstract type defined by the library, and `OBJ` is a user-defined type, discussed below):

```
312  'type' IDENT

313  'action' DefMeaning(Id: IDENT, Obj: OBJ)
314  'condition' HasMeaning(Id: IDENT -> Obj: OBJ)
```

These predicates can be used to implement a flat name space, i.e. they can be used directly to deal with languages without nested scopes.

Most languages (including *MiniLAX*) support nested scopes. These can be implemented on top of the predicates discussed so far.

Consider the program

```
PROGRAM test;

DECLARE
  x : BOOLEAN;

  PROCEDURE P;
  DECLARE
    x : INTEGER
  BEGIN
    x := 1
  END

BEGIN
  x := TRUE
END.
```

Here the `x` declared as `INTEGER` local to procedure `P` hides the globally declared `x` of type `BOOLEAN`.

We want to associate a descriptor with each declared item. For this we introduce the type `DESCR`:

```
315  'type' DESCR
316     varobj(INT, TYPE)
317     varparamobj(INT, TYPE)
318     valueparamobj(INT, TYPE)
319     procobj(LABEL, DECLLIST)
```

For example, if an item is declared as a variable with offset `3` and `BOOLEAN` (such as the global `x`) it gets the descriptor

```
varobj( 3, boolean )
```

The local `x` gets the descripror

```
varobj( 3, integer )
```

In the case of a flat name space we could merely associate the descriptor with an identifier, using the action `DefMeaning`. In case of nested scopes the situation is sligthly more complex.

In the discussion of translating procedures we already sketched the general strategy:

When we enter a procedure (recall that the program itself is treated as a procedure), we make the items declared there visible. We do this by invoking `DeclareList` for the formal parameters and the local declarations, thereby hiding possible declarations for items with the same name. When leaving the procedure, the hidden declarations are made visible again. This is done by invoking `UndeclareList`.

We therefore associate an entry with an identifier, that not only specifies its descriptor but also what is currently hidden by the actual declaration. When we use the identifier we get the actual descriptor. When leaving the procedure, we can return to the old meaning. For this prpose we introduce the type `OBJ`:

```
320  'type' OBJ
321     object(DESCR, INT, OBJ)
322     none
```

If a term

```
object( Descr, Level, Hidden )
```

is associated with an identifier, this means that the actual descriptor is `Descr`, that the corresponding declaration was at nesting level `Level`, and that `Hidden` was the previous association of the identifier (the `Hidden` entry forms a list of zero or more associations).

For example, if we access the identifier `x` at the assignment `x := TRUE` in the global scope, the association is

```
object(
   varobj( 3, boolean ),
   1,
   none
)
```

The current descriptor is

```
varobj( 3, boolean )
```

the nesting level is `1`, and there is no hidden item.

If we access `x` at the assignment `x := 1` in the local scope, we get the association

```
object(
   varobj( 3, integer ),
   2,
```

```
      object(
         varobj( 3, boolean ),
         1,
         none
      )
   )
```

The current descriptor is

```
varobj( 3, integer ),
```

the nesting level is 2, and the previous association is defined as hidden.

We use the predicates `GetMeaning` and `HasMeaning` define and query these associations.

Note that with this strategy accessing the meaning of identifiers has the same cost as in the case of a flat name space. It is done in constatnt time because the library predicates use a hash table algorithm.

Here are the predicates for scope handling:

```
323  'action' DeclareList(Decls: DECLLIST, Location: INT -> Size: INT)
324     'rule' DeclareList(decllist(D, Ds), Loc -> Size1+Size2) :
325        Declare(D, Loc -> Size1) DeclareList(Ds,Loc+Size1 -> Size2)
326     'rule' DeclareList(nil, Loc -> 0)

327  'action' Declare(Decl: DECL, Location: INT -> Size: INT)
328     'rule' Declare(dcl(Ident, Def, Pos), Loc -> Size) :
329        GetCurrentNesting(-> Level)
330        CurObject(Ident -> Hidden)
331        CheckNotYetDeclared(Hidden, Level, Pos)
332        Descriptor(Def, Loc, Pos -> Descr, Size)
333        DefMeaning(Ident, object(Descr, Level, Hidden))

334  'action' UndeclareList(Decls: DECLLIST)
335     'rule' UndeclareList(decllist(D, Ds)) :
336        Undeclare(D) UndeclareList(Ds)
337     'rule' UndeclareList(nil)

338  'action' Undeclare(Decl: DECL)
339     'rule' Undeclare(dcl(Ident, Def, Pos)) :
340        HasMeaning(Ident -> object(Descr, Level, Hidden))
341        DefMeaning(Ident, Hidden)

342  'action' CurObject(Ident: IDENT -> Obj: OBJ)
343     'rule' CurObject(Ident -> Obj) : HasMeaning(Ident -> Obj)
```

```
344     'rule' CurObject(Ident -> none)

345  'action' Apply(Ident: IDENT, Pos: POS -> Obj: OBJ)
346     'rule' Apply(Ident, Pos -> Obj) : HasMeaning(Ident -> Obj)
347     'rule' Apply(Ident, Pos -> none) : Error("id not declared", Pos)

348  'action' Descriptor(Def: DEF, Loc: INT, Pos: POS
349           -> Descr: DESCR, Size: INT)
350     'rule' Descriptor(variable(T), Loc, Pos
351            -> varobj(Loc, T), Size) :
352        CheckType(T, Pos) TypeSize(T -> Size)
353     'rule' Descriptor(valueparam(T), Loc, Pos
354            -> valueparamobj(Loc, T), 1) :
355        CheckType(T, Pos) CheckSimple(T, Pos)
356     'rule' Descriptor(varparam(T), Loc, Pos
357            -> varparamobj(Loc, T), 1) :
358        CheckType(T, Pos)
359     'rule' Descriptor(proc(Fs, Ds, S), _, _
360            -> procobj(Start, Fs), 0)
361        NewLabel(-> Start)
```

`DeclareList`

`DeclareList(List, Loc -> Size)` declares the items appearing in the declaration list `List`, i.e. it makes them visible. The items start at `Loc` in the stackfame and occupy `Size` locations.

`Declare`

`Declare(dcl(Ident, Def, Pos), Loc -> Size)` declares the identfier `Ident` with definition `Def`. It determines the current nesting level (`Level`), gets the current association of `Ident` (`Hidden`), builds a descriptor (`Descr`), and the defines

```
object(Descr, Level, Hidden)
```

as the the new meaning of `Ident`.

`UndeclareList`

`UndeclareList(Decls)` invokes the predicate `Undeclare` for each member of `Decls`.

`Undeclare`

`Undeclare(dcl(Ident, Def, Pos))` resets the meaning of `Ident` to its old value.

`CurObject`

`CurObj(Ident -> Obj)` returns the current object associated with `Ident` (which is `none` if there was no previous declaration for `Ident`.

The specification

```
'rule' CurObject(Ident -> Obj) : HasMeaning(Ident -> Obj)
'rule' CurObject(Ident -> none)
```

relies on the fact the rules are evaluated in the given order. Hence we know in the second rule that `HasMeaning` was not applicable (failed) for `Ident` and that we have to return `none`.

This could also have been written in one rule using a conditional statement.

`Apply`

`Apply(Ident, Pos -> Obj)` is similar to `CurObj(Ident -> Obj)` except that it emits an error message if there is no current association for `Ident`.

`Descriptor`

`Descriptor(Def, Loc, Pos -> Descr, Size)`, where `Def` is a definition of an item that should be placed at location `Loc` and was declared at position `Pos`, construct a corresponding descriptor `Descr`. It also computes the size `Size` of the item. The invocation also checks the definition for semantic constraints.

6.3.9 Auxiliary Predicates

We now define some auxiliary predicates that were used in the above specifications.

(The checks could also be written "inline" using conditional statements. We refrained from doing so in order to keep the number of concepts small in the example compiler).

```
362  'action' CheckInt(Type: TYPE, Pos: POS)
363     'rule' CheckInt(integer, Pos)
364     'rule' CheckInt(T, Pos) : Error("Integer expected", Pos)

365  'action' CheckBool(Type: TYPE, Pos: POS)
366     'rule' CheckBool(boolean, Pos) :
367     'rule' CheckBool(T, Pos) : Error("Boolean expected", Pos)

368  'action' CheckSimple(Type: TYPE, Pos: POS)
369     'rule' CheckSimple(integer, _)
```

```
370     'rule' CheckSimple(real, _)
371     'rule' CheckSimple(boolean, _)
372     'rule' CheckSimple(T, Pos) : Error("simple type expected", Pos)

373  'action' CheckArrayType(Type: TYPE, Pos: POS
374           -> Lwb: INT, Upb: INT, ElemType: TYPE)
375     'rule' CheckArrayType(array(Lwb, Upb, ElemType), Pos
376            -> Lwb, Upb, ElemType)
377     'rule' CheckArrayType(T, Pos -> 0, 0, none) :
378        Error("Subscripted var is not an array", Pos)

379  'action' CheckDesignator(E: EXPR, Pos: POS -> D: DESIG)
380     'rule' CheckDesignator(desig(D), _ -> D)
381     'rule' CheckDesignator(E, Pos -> none) :
382        Error("VAR parameter expected", Pos)

383  'action' CheckEquiv(T1: TYPE, T2: TYPE, Pos: POS)
384     'rule' CheckEquiv(T1, T2, Pos) : eq(T1, T2)
385     'rule' CheckEquiv(T1, T2, Pos) :
386        Error("Types are not equivalent", Pos)

387  'action' CheckType(Type: TYPE, Pos: POS)
388     'rule' CheckType(array(Lwb, Upb, ElemType), Pos) :
389        CheckBounds(Lwb, Upb, Pos) CheckType(ElemType, Pos)
390     'rule' CheckType(_, _) :

391  'action' CheckBounds(Lwb: INT, Upb: INT, Pos: POS)
392     'rule' CheckBounds(Lwb, Upb, _): le(Lwb, Upb)
393     'rule' CheckBounds(_, _, Pos):
394        Error("lower bound exceeds upper bound", Pos)

395  'action' CheckNotYetDeclared(Obj: OBJ, Lev: INT, Pos: POS)
396     'rule' CheckNotYetDeclared(none, Level, Pos) :
397     'rule' CheckNotYetDeclared(object(_, ObjLev, _), Level, Pos) :
398        lt(ObjLev, Level)
399     'rule' CheckNotYetDeclared(_, _, Pos) :
400        Error("id already declared", Pos)

401  'action' CheckProcedure(OBJ, POS -> DECLLIST, INT, LABEL)
402     'rule' CheckProcedure(object(procobj(Start,Formals),Level,_),_
403        -> Formals, Level, Start)
404     'rule' CheckProcedure(_, Pos -> nil, 0, Null):
405        Error("procedure expected", Pos)
406        NewLabel(-> Null)
```

```
407  'action' Error(Msg: STRING, Pos: POS)
```

The above predicates check their first argument for semantic constraints. If it is erroneous, then an error message is issued using the library predicate `Error(Msg, Pos)`).

Some predicates return certain characteristics of their argument.

For example, `CheckArrayType(Type, Pos -> Lwb, Upb, ElemType)` checks whether the argument is an array type, and returns the lower and upper bound and the element type if this is true.

```
408  'var' CurNestingLevel: INT

409  'action' SetCurrentNesting(N: INT)
410     'rule' SetCurrentNesting(N): CurNestingLevel <- N

411  'action' GetCurrentNesting(-> INT)
412     'rule' GetCurrentNesting(-> N): CurNestingLevel -> N

413  'action' TypeCode(Type: TYPE -> Code: INT)
414     'rule' TypeCode(integer -> 1)
415     'rule' TypeCode(real -> 2)
416     'rule' TypeCode(boolean -> 3)

417  'action' TypeSize(Type: TYPE -> Size: INT)
418     'rule' TypeSize(integer -> 1)
419     'rule' TypeSize(real -> 1)
420     'rule' TypeSize(boolean -> 1)
421     'rule' TypeSize(array(Lwb, Upb, ElemType)
422            ->((Upb-Lwb)+1)*ElemSize) :
423        TypeSize(ElemType -> ElemSize)
```

The current nesting level of procedures is implemented as a global variable, the predicates `SetCurrentNesting` and `GetCurrentNesting` are used to set and get its value.

The predicates `TypeCode` and `TypeSize` compute the encoding and the size of a type.

6.3.10 Target Interface

The virtual *MiniLAX* machine is implemented as a *C* module. Here is the *Gentle* interface for this module:

```
424  'type' LABEL
425  'type' FLOAT

426  'action' LDA(INT, INT)
427  'action' LDC(INT, INT)
428  'action' LDF(FLOAT)
429  'action' LDI
430  'action' STI
431  'action' JMP(LABEL)
432  'action' FJP(LABEL)
433  'action' ADD(INT)
434  'action' SUB
435  'action' MUL(INT)
436  'action' INV
437  'action' LES(INT)
438  'action' IXA(INT)
439  'action' FLT
440  'action' WRI(INT)
441  'action' REA(INT)
442  'action' MST(INT)
443  'action' JSR(INT, LABEL)
444  'action' ENT(INT)
445  'action' RET
446  'action' CHK(INT, INT)

447  'action' NewLabel(-> L: LABEL)
448  'action' LAB(L: LABEL)
449  'action' Emit
```

For each instruction (`LDA`, `LDC`, ... , `CHK`), there is a corresponding predicate. For example,

```
'action' CHK (INT, INT)
```

corresponds to the *ICode* `CHK` instruction.

```
CHK (1, 20)
```

appends an instruction

```
CHK 1, 20
```

to the target program.

The type `Label` comprises "symbolic values" that are used as labels; final machine addresses are not used as instruction operands. The translation of symbolic values into addresses is done by the virtual machine if the program is complete.

A new label is created by

```
NewLabel(-> Lab)
```

Thereafter, `Lab` can be used in instructions, e.g.

```
JMP(Lab)
```

The location of the label must be defined by the pseudo-instruction

```
LAB(Lab)
```

which defines the following instruction as the target of a jump to `Lab`.

The action `Emit` writes the target program into a file (alternatively, it could be interpreted directly).

References

[1] A.W. Appel: Modern Compiler Implementation in C: Basic Techniques. Cambridge University Press, 1997.

[2] F.L. Bauer, F.L. de Remer, A.P. Ershow, D. Gries, M. Griffith, U. Hill, J.J. Horning, C.H.A. Koster, W.M. Mc Keeman, P.C. Poole, W.M. Waite: Compiler Construction. An Advanced Course. Springer-Verlag, New York, Heidelberg, Berlin, Second Edition, 1976

[3] H. Emmelmann, F. W. Schröer, R. Landwehr: BEG - a Generator for Efficient Back Ends. Proceedings of the Sigplan'89 Conference on Programming Language Design and Implementation, Portland, Oregon, 1989, Sigplan Notices, Vol. 24, Number 7, July 1989

[4] H. Emmelmann, J. Grosch: A Tool Box for Compiler Construction. GMD Karlsruhe, Technical Report 1990.

[5] S.C. Johnson: Yacc - Yet Another Compiler-Compiler. Comp Sci. Tech rep. No 32. Bell Laboratories, July 1975

[6] B.W. Kernighan, D.M. Ritchie: The C Programming Language. Second Edition. Prentice Hall, 1978

[7] C.H.A. Koster: Using the CDL compiler-compiler. In [2].

[8] M.E. Lesk: Lex - A Lexical Analyzer Generator. Comp. Sci. Tech. rep. No. 39. Bell Laboratories, October 1975.

[9] K. V. Nori, U. Ammann, K Jensen, H.H. Nägeli, C. Jacobi: The Pascal-P Compiler: Implementation Notes. Bericht 10, Eidgenössische Technische Hochschule, Zürich, July 1976

[10] F. W. Schröer: Gentle. In [13].

[11] F.W. Schröer, F. Engelmann, D. Schwarz-Hertzner: The GMD Modula-2 System MOCKA. GMD Karlsruhe, Technical Report 1986.

[12] B. Schwarz, W. Kirchgässner, R. Landwehr: An Optimizer for Ada - Design, Experiences and Results. PLDI 1988: 175-184

[13] W.M. Waite, J. Grosch, F.W. Schröer: Three Compiler Specifications. GMD-Studien Nr. 166. GMD German National Research Center for Information Technology, 1989.

[14] W.M. Waite, G. Goos: Compiler Construction. Texts and Monographs in Computer Sciences. Springer-Verlag, New York, Berlin, Heidelberg, Tokyo, 1984

[15] D.H.D Warren: Logic Programming and Compiler Writing. Software, Practice and Experience, Vol 10, pp 97-125, 1980.