The Compiler Construction System GENTLE
— Manual and Tutorial —

Jürgen Vollmer
GMD Research Group at the University of Karlsruhe
Vincenz–Prießnitz–Straße 1, D–7500 Karlsruhe 1
email: vollmer@karlsruhe.gmd.de
Phone: +/49/721/6622-14

August 25, 1992
Revision 3.9

Updated edition of *Arbeitsberichte der GMD Nr. 508*, Februar 1991

Abstract

Gentle defined by F.W. Schröer [Schröer 89] is a compiler description language in the tradition of logic programming [Clocksin *et al* 84] and two level grammars [Fisker *et al* 75, Koster 71, Watt 74]. It provides a common notation for high level description of analysis, transformation, and synthesis. A tool has been implemented to check the wellformedness of *Gentle* descriptions, and to generate efficient compilers. *Gentle* replaces a variety of special purpose languages by a general calculus: Horn logic.
The language, a programming environment, and a tutorial are presented in this paper.

Contents

Preface

Gentle defined by F.W. Schröer [Schröer 89] is a compiler description language in the tradition of logic programming and two level grammars. It provides a common notation for high level description of analysis, transformation, and synthesis. Compilation is often viewed as a process translating the source text into a sequence of intermediate languages, until the desired output is synthesized. These intermediate languages may be viewed as terms, and *Gentle* offers a simple and efficient way to transform these terms (intermediate languages). These transformations are described in a declarative way using predicates. Due to the special nature of the task (describing compilers) Horn logic as *Gentle*'s foundation is modified in several ways: *Gentle* is a typed language and the data flow inside the predicates is fixed. Several kinds of predicates are offered for different jobs during compilation. It is restricted compared to Prolog in its backtracking behaviour and its pattern matching rules. Besides the specification of terms and rules transforming them, the concrete syntax of the context free source languages is declared using the same declarative notation. This grammar specification is used to generate a parser for the language. Output of the system is generated by side effects caused by predicates.
A tool[1] has been implemented that checks the wellformedness of a *Gentle* specification and generates very fast compilers. The language *Gentle* and a supporting tool were designed and implemented by F.W. Schröer in 1989 and published in [Schröer 89], which is a comparison study of three compiler generation tools. Starting from Schröer's first implementation of a *Gentle* tool, the *Gentle* programming environment using the scanner generator *rex* [Grosch 87] the parser generator *lalr* [Vielsack 88] (instead of the initially used *yacc*) has been implemented. It provides a library containing often used predicates; better error handling etc. has been implemented. Measurements of the speed of *Gentle* and the generated programs are given in [Vollmer 91b].
The paper is structured in the following way: First in chapter 1 the *Gentle* language is presented. The next chapter gives the user manual and a description of the library. Chapter 3 is a tutorial of *Gentle* implementing a program interpreter. The appendix contains the UNIX manual page entry for the usage of the *Gentle* tool, and a *Gentle* syntax summary.
Finally, I want to express my special thanks to FriWi Schröer.

[1] The current version of *Gentle* is: 3.9 August 25, 1992

Chapter 1

Gentle Language Reference Manual

1.1 Introduction

The chapter is structured in the following way: First in section 1.2 a simple *Gentle* specification is presented, which solves constant folding. Section 1.3 informally introduces the other languages constructs. Section 1.4 gives some general information. The next section defines *Gentle*. In section 1.7 it is shown how a typical target specification looks like. In section 1.8 *Gentle* is related to *Prolog*. The *Gentle* syntax is summarized in the appendix C.

1.2 A simple example

1.2.1 Types, terms, and action predicates

This simple example shows the methods of *Gentle* to describe a term transformation, which is the main point of *Gentle*. The exercise is to fold constants, i.e. to evaluate integer expressions during compilation as far as possible. The example source language consists of binary integer expressions, identifiers for integer variables, and integer constants.
These integer expressions are represented as *Gentle* terms. In *Gentle*, all terms are explicitly typed. Here the type of those terms is called `EXPR`, which may be either a constant, an integer variable, or a binary operation on `EXPR` terms. The integer constants are represented by terms of type `INT` and identifiers by terms of type `IDENT`, which are declared in the *Gentle* library. Terms of type `OP` denote the kind of the operation.

```
EXPR = const  (INT),            -- integer constants
       var    (IDENT),          -- access of an integer variable
       binary (OP, EXPR, EXPR).
OP =   plus, minus, mult, div.
```

Terms are formed using functors, terms and term variables in the usual way. The type of the term variable is derived from its context.
The source expression `a + 3` is represented by `binary (plus, var (a), const (3))`. In the terms `const(N)` and `binary (plus, X, Y)`, the term variables are `N,X,Y`; `plus` is just a term, having no sub–terms. Such terms are called constant terms. `a`, `3` are constant terms of type `IDENT` and `INT`, respectively.
The term transformation is specified by *predicates*. As usual a predicate is defined as a disjunction of *Horn clauses*: $\texttt{clause}_1$ ...$\texttt{clause}_n$. A clause has a *head* and a *tail*. The tail is a possibly empty list of literals. The head and each literal may have *input* and *output parameters*, partioned by the arrow symbol, which specify the data flow of the parameters:

$\texttt{head(input}_h\texttt{ -> output}_h\texttt{) :}$ $\texttt{literal}_1\texttt{ (input}_1\texttt{ -> output}_1\texttt{)}$
...
$\texttt{literal}_m\texttt{ (input}_m\texttt{ -> output}_m\texttt{).}$

Such a clause is an implication, where the hypothesis is formed as a conjunction of tail literals, and the conclusion is given by the head. For each literal there must be a head of a clause with the same name. This implication

may be read as "if `literal`$_1$ is true and `literal`$_2$ is true ...then that clause is true". The entire predicate is true, if at least one of its clauses is true.
The process of term transformation may be viewed as the proof of a predicate. If a proof exists, the input–output relation holds, or with other words: the input term is transformed to the output term.
The task of our example is to evaluate statically as much of an expression as possible. Input is an expression tree (i.e. term), which should be transformed again into an expression tree (i.e. term). For example: `a + (3 + 4)` is represented as: `binary (plus, var (a), binary (plus, const(3), const (4)))` should be transformed to `a+7` or `binary (plus, var (a), const (7))`. The idea for solving this problem is first to fold the children of the `binary` term and then to apply the operator to the folded child terms. The predicates below specify this:

```
'ACTION' fold (EXPR -> EXPR).
fold (const (N) -> const (N)):.
fold (var (X)   -> var (X)):.
fold (binary (Op, E1, E2) -> Result):
     fold (E1 -> R1)
     fold (E2 -> R2)
     eval (binary (Op, R1, R2) -> Result).

'ACTION' eval (EXPR -> EXPR).
eval (binary (plus,  const (N1), const (N2)) -> const (N1 + N2)):.
eval (binary (minus, const (N1), const (N2)) -> const (N1 - N2)):.
eval (binary (mult,  const (N1), const (N2)) -> const (N1 * N2)):.
eval (binary (div,   const (N1), const (N2)) -> const (N1 / N2)):.
eval (E -> E):.
```

Gentle keywords are "enclosed" by apostrophes to distinguish them from *Gentle* identifiers. For each predicate a *signature* is declared, specifying the predicate name, the type of input and output terms, and the kind of the predicate, here *action predicates*. Both predicates have `EXPR` terms as input and output parameters. The `fold` predicate specifies some kind of recursion over terms. The first two `fold` clauses are the base of that recursion, and mean that constants and variables are not changed. The type of the term variables `N` and `X` derived from the context are `INT` and `IDENT` respectively. Note that the tails of these clauses are empty. The third `fold` clause folds the child terms `E1, E2` of `binary` and passes the result terms as parameter to the `eval` predicate. Which of the clauses is selected depends on the form of the input term. This selection is performed by *matching* the actual and formal parameters of the predicate. For example if the input term of the `eval` predicate has the form `binary (plus, const (N1), const (N2))` then the first `eval` clause is selected, and the output term has the form `const (N1 + N2)`. If the term has a form, not so specific as given in the first four `eval` clauses, the last clause matches always, because the term variable `E` matches with all kinds of `EXPR` terms.
For some term types, the so called *opaque types*, *Gentle* interprets the terms as integers, and provides integer arithmetic on them. In the term `N1 + N2` above, the fact that `N1` and `N2` are term variables of the opaque type `INT`, is derived from their context. Hence, `N1 + N2` is interpreted by *Gentle* as an integer expression with integer variables. When this term is processed, the integer value of `N1 + N2` is computed and used as a constant term of type `INT`.

1.2.2 Grammar specification, token, and nonterminal predicates

Now the question arises, how the terms are constructed initially, or the other way round, what is the input of that constant folding program. As said before, a *Gentle* specification uses a context free grammar to describe the input language of the target program. When a sentence of the input language is read (parsed), an internal representation of the sentence is constructed. The context free grammar and the construction process are specified also using terms and predicates.
Two new kinds of predicates for specifying the grammar are introduced: *token predicates* and *nonterminal predicates*. Both predicate kinds may have at most one output parameter and no input parameters. The tokens are "produced" by a scanner implemented outside of *Gentle*, hence there are no clauses for token predicates. The clauses of the nonterminal predicate may be read as production rules of the context free grammar. When such a production rule is reduced by the parser, the action predicates following the tokens or nonterminals of the production's right hand side, and the term construction of the output parameter are performed. For the example above the grammar specification looks like:

```
'TOKEN' PLUS.
'TOKEN' MINUS.
'TOKEN' MULT.
'TOKEN' DIV.
'TOKEN' LEFTPAR.
'TOKEN' RIGHTPAR.
'TOKEN' NUMBER (-> INT).
'TOKEN' IDENTIFIER (-> IDENT).

'NONTERM' Root.
Root : Expr (-> E)
       print_EXPR (E)
       fold (E -> FoldedExpr)
       print_EXPR (FoldedExpr).

'NONTERM' Expr (-> EXPR).
Expr (-> E)                     : Term (-> E) .
Expr (-> binary (plus, E1, E2)) : Expr (-> E1) PLUS Term (-> E2) .
Expr (-> binary (minus, E1, E2)): Expr (-> E1) MINUS Term (-> E2) .

'NONTERM' Term (-> EXPR).
Term (-> E)                     : Factor (-> E) .
Term (-> binary (mult, E1, E2)) : Term (-> E1) MULT Factor (-> E2) .
Term (-> binary (div, E1, E2))  : Term (-> E1) DIV  Factor (-> E2) .

'NONTERM' Factor (-> EXPR).
Factor (-> const(N)): NUMBER (-> N) .
Factor (-> var(X))  : IDENTIFIER (-> X) .
Factor (-> E)       : LEFTPAR Expr (-> E) RIGHTPAR .
```

`Root` is the root symbol of the context free grammar. The `Root` clause may be read as "parse an expression, print it, fold it, and print the folded expression", where the `print_EXPR` predicate is defined at another place.

1.3 Introduction of other Gentle constructs

1.3.1 Global variables and condition predicates

First global variables and another kind of predicate are presented. Global variables make it easy to maintain global information for example a list of all identifiers, used in a program. The following program fragment defines such a list of identifiers having type `IDENTS`.
Using `'VAR'` declares a global variable, having the type `IDENTS` and name `AllIdents`. A value is assigned to a global variable using the special predicate `Variable <- Term` while the value is used by `Variable -> Term` or writing the variable on a using position.
The action predicate `Insert` inserts an identifier into that list, only if it is not already contained in it. A *condition predicate* is used to test a condition over terms it may fail, or succeed. Action predicates are not allowed to fail. If `IsContained` fails, when called from the first `Insert` rule, the second `Insert` rule is tried, which does the actual inserting. Or more general, if in clause C of predicate P the call of a tail predicate P' fails, the entire clause C fails, and the next clause of P is tried. If there is no next clause of P, then P itself fails.

```
IDENTS = idl (IDENT, IDENTS), nil .

'VAR' IDENTS AllIdents.

'ACTION' Insert (IDENTS, IDENT).
Insert (Ids, Id): IsContained (Id, AllIdents) .
Insert (Ids, Id): AllIdents <- idl (Id, Ids) .

'CONDITION' IsContained (IDENT, IDENTS).
IsContained (Id1, idl (Id2, Ids)): Equal (Id1, Id2) .
```

```
-- When condition Equal succeeds, IsContained succeeds.

IsContained (Id1, idl (Id2, Ids)): IsContained (Id1, Ids) .
-- When condition IsContained (of the tail) succeeds, IsContained succeeds.

-- If the empty list (nil) is reached, the predicate fails, because
-- there is no rule for that case.
```

1.3.2 Dynamic global tables

A generalization of global variables is the global table concept. A global table is something like an array in common imperative languages, except, that space for entries is provided dynamically. Entries of the table are accessed using a *key*.
This concept may be used to represent graphs with *Gentle*, which is not possible using terms only. The following example, shows a graph marking algorithm to compute a minimal spanning tree for a graph and a given root node. The graph is represented as a table of nodes, each node is a tuple, consisting of a *mark* field, a field for the information stored in the graph, and a field for each successor. The number of successors of a node has a fixed upper bound (in this example four). The following program fragment shows the usage of the language constructs:

```
NODEATTR =
        node (Marked : BOOLEAN,  -- true: marks a node as visited
              Info   : INFO,     -- user defined information
              Succ1  : NODE, Succ2 : NODE, Succ3 : NODE, Succ4 : NODE),
        nil -- represent the ‘‘empty’’ successor
        .

'TABLE' NODEATTR Graph [NODE].

MST =   -- terms constructing the minimal spanning tree
        mst (Info : INFO, Succ1 : MST, Succ2 : MST, Succ3 : MST, Succ4 : MST),
        nil
        .

'ACTION' new (-> NODE).
-----------------------
new (-> Node) : 'KEY' NODE Node.

'ACTION' define (NODE, INFO, NODE, NODE, NODE, NODE).
-----------------------------------------------------
define (Node, Info, Succ1, Succ2, Succ3, Succ4) :
        Graph [Node] <- node (false, Info, Succ1, Succ2, Succ3, Succ4)
        -- the node is marked as "unvisited", i.e. with the term "false"
        .

'ACTION' ComputeMst (NODE -> MST).
----------------------------------
ComputeMst (nil -> nil): .  -- this node is the empty node
ComputeMst (Root -> nil): -- this node is marked
        Graph [Root] -> node (true, Info, Succ1, Succ2, Succ3, Succ4)
        .
ComputeMst (Root -> mst (Info, Mst1, Mst2, Mst3, Mst4)): -- this node is unmarked
        Graph [Root] -> node (false, Info, Succ1, Succ2, Succ3, Succ4)
        Graph [Root] <- node (true,  Info, Succ1, Succ2, Succ3, Succ4)
              -- mark this node and call "ComputeMst" for all children
        ComputeMst (Succ1 -> Mst1)
        ComputeMst (Succ2 -> Mst2)
        ComputeMst (Succ3 -> Mst3)
        ComputeMst (Succ4 -> Mst4)
```

```
ComputeMst (Succ1 -> Mst1)
.
```

The global table `Graph` is declared using `'TABLE'`. The terms contained as table entries have the type `NODEATTR`. Space for a new entry is created using the special `'KEY'` literal, where the local variable `Node` holds the resulting *key*, which is used for accessing entries of the table. A table entry access is done in the same way as it is done for global variables, except that after the table name the key is given.

1.3.3 Escapes from Gentle: opaque types, and external predicates

Each programming language needs access to the underlying operating system, for example performing input and output. Some languages provide special language constructs, others not. *Gentle* uses this second way.
Gentle offers so called *external (action and condition) predicates*, which are implemented in another programming language, usually C. A *Gentle* specification needs only to know the predicates signature.
Another escape is needed to use data entities, like floating point numbers, unique identifications for program identifiers, etc., which are not provided by *Gentle*. To solve this, *opaque types* may be declared. The user has to define the meaning of opaque values, by predicates dealing with them.
The `Equal` condition above is such an external predicate, comparing two identifiers, which are opaque values. The opaque values representing identifiers are usually computed by the scanner.

1.4 How things work

This section gives some general information how the *Gentle* tool is used. As *Gentle* is used for the generation of programs, which analyse and transform texts (for example compilers, interpreters, text analysis) the input to the generated program is a stream of tokens, which is constructed by a scanner. Output or effect of the generated program, which is sometimes called the target program, may be a file containing assembler instructions, or interactively interpretation of the input, or a transformed text (for examples see section 2.13).
A *Gentle* specification consists of mainly two parts. The accepted input language is specified by a context free grammar and rules are specifyed how the internal representation of that input should be transformed to produce the desired output. The grammar rules are annotated in some way to construct that internal representation. When the generated program is started, first it parses the entire input and constructs the internal representation; second the transformations are performed to produce the output, starting with the actions specified at the root symbol of the grammar.
The parser is generated out of the grammar given in a *Gentle* specification. For that purpose parser generators like *yacc* or *lalr* [Vielsack 88] are used. The user of *Gentle* is not bothered with that. But the scanner must be written by the user. Scanner generation tools like *lex* or *rex* [Grosch 87] may be used. *Gentle* produces several kinds of output: input for scanner and parser generator tools and *C* programs. Compiling and linking them together results in a program solving the specified problem.
A *Gentle* specifications may be separated into several modules, each contained in a separate file. A library of common used predicates exists (see section 2.12).

1.5 Syntax and semantics of Gentle

This section defines the syntax and the static semantics of *Gentle*. The syntax is described in extended BNF[1].

[1]

construct	meaning
"α"	α is a non–keyword terminal symbol
[α]	the part between the brackets is optional
(α)	the part between the parentheses is a unit
$\alpha//\beta$	a nonempty sequence of $\alpha's$ separated by $\beta's$
α^*	any number of α's
α^+	one or more α's
$\alpha\beta$	α then β
$\alpha\|\beta$	either α or β

1.5.1 Keywords, identifiers, and name scopes

The following keywords are used by *Gentle*:

```
'MODULE' 'TYPE' 'VAR' 'TABLE' 'TOKEN' 'NONTERM' 'ACTION'
'CONDITION' 'KEY'
```

Notice, the apostrophes surrounding the letters, belong to the keyword.
Identifiers are sequences of letters, digits, and the underscore character. The first character must be a letter. Capital and lower case letters are considered distinct. Like *Prolog Gentle* distinguishes between *identifiers* starting with a capital letter and a lower case letter. As a summary: names, whose first letter is a capital are: type names and variable names. Names beginning with a lower case letter are: functor names. Names which may start with both lower and upper case letters are: predicate names, module names.

```
LargeIdent   ::=  ( "A" | ... | "Z" ) (letter | digit)* .
SmallIdent   ::=  ( "a" | ... | "z" ) (letter | digit)* .
Identifier   ::=  LargeIdent | SmallIdent .
letter       ::=  "A" | ... | "Z" | "a" ... "z" | "_" .
digit        ::=  "0" | ... | "9" .
```

Examples:

```
isEqual match_patterns_5
```

Identifiers and keywords are separated by blanks, line breaks, and the following special symbols:

```
<-  ->  :  ,  .  (  )  [  ]  /*  */  --  +  -  *  /  "  =
```

Gentle provides three different global and several local name spaces. The names in one global space must be unique in the entire *Gentle* specification (i.e. in all used modules). Names in a local name space must be unique in that local name space. The actual meaning of an identifier is derived from its context.
The following rules for names must be observed:

- All predicate names form a global name space.
- All global variables names form a global name space.
- All type names form a global name space.
- All functor names of one type declaration form a local name space for that type.
- All local variables of a clause form a local name space for that clause. Each local variable name space must be disjoint to the global variable name space, i.e. local variable names must be different from all other global variables names.

Identifiers may be used, before they are declared.

1.5.2 Comments

There are two kinds of *comments* in a *Gentle* specification:

```
-- starts a single line comment and
/* starts a /* possibly nested */ comment, which may range
over several lines */
```

1.5.3 Modules

A *Gentle* specification may be separated into several *modules*. Each module is contained in a separate file. There are two restrictions: All tokens must be declared in the same module; the context free grammar must appear in one module. All modules processed by the *Gentle* tool (see chapter 2) form together the problem specification. An identifier declared in one module is visible in in all other processed modules (in the corresponding name space).
A *Gentle* module consists of *declarations, predicate signatures*, and *clauses*.

```
Gentle_Spec   ::=  'MODULE' Identifier ModuleBody .
ModuleBody    ::=  (Declaration | Signature | Clause )* .
```

1.5.4 Type declarations

There are two groups of declarations. First the declaration of types of terms, second the declaration of global variables (see section 1.5.12).

```
Declaration   ::=  TermTypeDecl | OpaqueTypeDecl | GlobalVarDecl | GlobalTableDecl .
```

A *type declaration* has the form:

```
TermTypeDecl   ::=  Type "=" FunctorList "." .
Type           ::=  LargeIdent.
FunctorList    ::=  ( Functor | Functor "(" Arguments ")" ) // "," .
Functor        ::=  SmallIdent.
Arguments      ::=  Argument // "," .
Argument       ::=  [LargeIdent ":" ] Type .
```

Examples:

```
Expr = binary (OP, Left : EXPR, Right : EXPR), const (INT), var (IDENT).
OP = plus, minus, mult, div.
```

Type of *Argument* may be any other declared type. The functor identifiers are local in a type declaration and must be unique there. The *LargeIdent* in *Argument* is used only for documentation purposes. These rules may be viewed as a context free grammar describing typed *values*, which are called *terms*. Functors having no arguments are called *constant terms*.
Another kind of values are *opaque* values, whose types are declared using:

```
OpaqueTypeDecl   ::=  'TYPE' Type "." .
```

Examples:

```
'TYPE' IDENT . -- represent identifiers
'TYPE' INT .   -- represent integers
'TYPE' STRING. -- represent strings.
```

Values of an opaque type are constant terms. For example the scanner returns a token which has an attribute specifying the value of an integer or an identifier.
The meaning and operations on opaque values are usually declared outside of *Gentle*. *Gentle* accepts integer and string constants as values of opaque types. Strings are used in a C like style, the escaping conventions of C are recognized.

```
IntConst      ::=  digit + .
StringConst   ::=  """ Char * """ .
Char          ::=  <any (escaped) character, except " and line break> .
```

Simple arithmetic (+, -, *, /) may be done on opaque values. The user has to ensure (outside of *Gentle*) in this case that the values represent entities, for which these operations are defined. *Gentle* does not provide any operations on string constants.

Examples:

```
"Hello \t world \n" -- \t: tab character, \n: newline
3 + (4 * X)         -- X is an variable of type INT.
```

1.5.5 Terms and pattern matching

A *simple term* is constructed in the following way:

> If $f(T_1, \ldots, T_n)$ is a functor of type T_0 and $X_1, \ldots, X_n$ are terms of types $T_1, \ldots, T_n$, then $f(X_1, \ldots, X_n)$ is a *term of type* T_0. A variable of type T_0 is also a term of type T_0 (the types of variables are derived from the context, i.e. the position in the term it occurs).

A term not containing variables is called a *ground term.*
Examples:

```
binary (plus (const (1), Y)) -- Y is a variable of type EXPR.
binary (plus (const (1), const (3))) -- is a ground term
```

Pattern matching of two terms plays a central role in parameter passing. One of the terms is always a ground term. The pattern matching process tries to make both terms equal, by assigning corresponding subterms of the ground term to variables of the other term. The pattern matching procedure may fail, if it is impossible to find such subterms. The corresponding terms are assigned to variables, if the procedure succeeds.
The pattern matching algorithm is:

```
PROCEDURE match (s, t : terms) : BOOLEAN
      --  Only  t may contain variables, which have to be disjoint.     s is a ground term.
      IF  t is a variable  X
         THEN   X := s; RETURN TRUE --  report success
      END IF
      IF   s = f (s_1, ... , s_n), t = g (t_1, ... , t_n) AND  f and  g denote the same functor symbol
         THEN  apply procedure  match to the pairs           (s_i, t_i); 1 ≤ i ≤ n
               IF the pattern matching succeeds for all  i
                   THEN RETURN TRUE --  report success.
                   ELSE RETURN FALSE --  report failure.
               END IF
         ELSE RETURN FALSE --  report failure.
      END IF
```

An example is: `match(binary(plus,const(3),const(4)),binary(Op,E1,E2))` succeeds and `plus` is assigned to `Op1`; `const(3)` to `E1` and `const(4)` to `E2`.
`match(const(3),binary(Op,E1,E2))` fails.

1.5.6 Predicate signatures

While processing an input text, the generated program has to do several jobs: tokens must be accepted, the input must be parsed using tokens, an internal representation of the text must be constructed, and transformed to produce an output. For these different jobs, *Gentle* offers four kinds of predicates: *token, nonterminal, action* and *condition* predicates. Logically they are all equivalent, but they have different side effects, for example controlling the parser.
The declaration of signatures for predicates is used to specify the kind of the predicate, and allows type checking on terms and variables.
The syntax of signatures is:

```
Signature       ::=  'TOKEN'      Identifier [ OutArguments   ] "." |
                     'NONTERM'    Identifier [ OutArguments   ] "." |
                     'ACTION'     Identifier [ InOutArguments ] "." |
                     'CONDITION'  Identifier [ InOutArguments ] "." .
OutArguments    ::=  "(" "->" Arguments ")" .
InOutArguments  ::=  "(" [ Arguments ] "->" Arguments ")" .
```

Examples:

```
'TOKEN' IDENTIFIER (-> IDENT).
'NONTERM' Expr (-> EXPR).
'ACTION' fold (EXPR -> EXPR).
'CONDITION' Eq_Int (INT, INT).
```

Notice, token and nonterminal predicates may have only output arguments. For token predicates the type of these output arguments is usually an opaque type. The nonterminal predicate given first serves as a root symbol of the context free grammar.

1.5.7 Clauses

The body of a predicate is formed by clauses. The syntax of a clause is:

```
Clause       ::=  Head ":" Tail "." .
Head         ::=  HeadLiteral .
Tail         ::=  TailLiteral * .
HeadLiteral  ::=   Identifier [ "(" FormalParameters ")" ] .
```

Examples:

```
Expr (-> var (X)) : IDENTIFIER (-> X).
fold (var (X) -> var (X)):.
```

For each clause with head name p there must be a predicate signature with name p, with the same input and output parameter types as the head. A clause with head name h *belongs* to a predicate p, if h and p denote the same identifier.

1.5.8 Literals

The right hand side of a clause consists of literals. A literal may be a predicate name and arguments for that predicate or an operation on global variables and tables.

```
TailLiterals  ::=  TailLiteral * .
TailLiteral   ::=  Identifier [ "(" ActualParameters ")" ] |
                   GlobalVarRead | GlobalVarWrite |
                   GlobalTableNewEntry |
                   GlobalTableRead | GlobalTableWrite .
```

Examples:

```
fold (binary (plus, const(3), const(4)) -> X)
fold (binary (plus, const(3), const(4)) -> const (N))
```

1.5.9 Local variables

Variables contained in terms of a clause are called *local variables* or just *variables*. A local variable is called to be in an *input position* (*output position*) if it is contained in a term, which is used as input (output) argument. The following rules must be observed, using variables in a clause:

- Local variables are *single assignment* variables. When processing the clause the variable is assigned a value only once (it is said to be *defined*).

- The value of a variable may be used at several places in a clause.
- A tail variable must be defined textually before it is used
- A local variable is defined, if it is at an
  – input position of the head, or
  – output position of a tail literal.
- A local variable is used, if it is at an
  – output position of the head, or
  – input position of a tail literal.

These two rules may be abbreviated by $head(Var_{def} \rightarrow Var_{use}) : tail(Var_{use} \rightarrow Var_{def})$.
The syntax of local variables is:

```
LocalVariable   ::=  LargeIdent.
```

1.5.10 Formal and actual parameters

The syntactic form of parameters is:

```
FormalParameters   ::=  Parameters .
ActualParameters   ::=  Parameters .
Parameters         ::=  [ InParameters ] [ "->" OutParameters ] .
InParameters       ::=  Parameter  // "," .
OutParameters      ::=  Parameter // "," .
Parameter          ::=  Term |
                        LocalVariable | GlobalVariable |
                        StringConst | IntConst |
                        Expression Operator Expression .
Term               ::=  Functor [ "(" ArgumentList ")" ] .
ArgumentList       ::=  Parameter // "," .
Expression         ::=  LocalVariable | GlobalVariable |
                        IntConst |
                        (Expression Operator Expression) .
Operator           ::=  "+" | "-" | "*" | "/" .
```

In *Parameter* global variables may be used but not defined (compare section 1.5.9).

1.5.11 Predicates

As *Gentle* is used to describe and generate text processing programs, there must be a way to specify the input of the generated program, i.e. the accepted language. This language is usually specified by a context free grammar. A scanner reads the text source file and returns a stream of tokens, which is used by a parser to analyze the syntactic text structure. The *token* and *nonterminal* predicates are used to specify the grammar. While parsing a text, usually an internal representation of that text is build. The next step is to transform this internal representation to analyze the text. The result of this step is the output of the target program. To specify this analysis *action predicates* and *condition predicates* are used.
The evaluation of predicates may succeed or fail. The meaning and reaction on success or failure depends on the predicate kind.

Token and nonterminal predicates

For token predicates only the signature is specified, i.e. there is no clause, which belongs to a token predicate. Nonterminal predicates must have at least one clause.
The token and nonterminal predicates describe a context free grammar. The nonterminal predicate names are read as nonterminal symbols, the token predicate names are read as terminal symbols of the grammar. The clauses of the nonterminals are production rules of the grammar. The tail of a nonterminal clause may contain

token, nonterminal, and action predicates, the action predicates must follow the token and nonterminals. The grammar specified with the token and nonterminal predicates must fulfill the *LALR(1)* condition [Waite *et al* 84]. If this condition may not be met, some exceptions are possible, see section 2.9.
From *Gentle*'s point of view, token and nonterminal predicates may never fail, because failure means there is a syntactic error, which is handled by the generated parser. In the error case an error message is emitted and and the syntactic error is repaired. The selection of a nonterminal clause is done by the parser using a special parsing algorithm.
Token and nonterminals may have no input parameters and may have several output parameters, which may be used to construct a term representation of the parsed input. If a nonterminal clause contains action predicates, these are evaluated, if this clause was selected by the parser, i.e. the parser reduced this grammar rule.

Action and condition predicates

If there is no clause for an action or condition signature, this predicate is said to be *external*. The body of an external predicate must be implemented outside of *Gentle*, see section 2.11.
Action and condition predicates are evaluated using the following strategy: The predicate signature and the clauses belonging to the signature form a procedure. These procedures have an additional boolean result, signaling success or failure of the predicate. The evaluation of a predicate is then transformed to a procedure call. A predicate procedure call is done in two steps, involving pattern matching. Let predicate p have formal parameters in_f, out_f be called with actual parameters in_a, out_a. The mapping of actual parameters to formal parameters is done using pattern matching. The formal input parameters get their values assigned, when a clause of the predicate is evaluated (see below). After the call of the procedure's body the output parameters are matched.

```
call_predicate ( p) : BOOLEAN;
   call_body  p;
   match (out_f, out_a);
   IF call_body AND match has been successful
   THEN RETURN TRUE --  report success
   ELSE RETURN FALSE --  report failure
   END IF
```

The body of a predicate procedure p is formed by the clauses $c_1 c_2 \ldots c_n$, numbered in their textual order. The clauses are evaluated in that order until the first succeeds. If this happens, success is reported to the caller of the body of p. If all clauses fail, failure is reported. In a short term: clauses are connected by disjunction. Let the clauses of p look like:

```
p (in_1 ->  out_1) : tail_1 --  clause c_1
...
p (in_n ->  out_n) : tail_n --  clause c_n
```

then the evaluation of the body is done by:

```
call_body ( p) : BOOLEAN;
   i := 1;
   LOOP
     IF i > n --  no more clauses
     THEN RETURN FALSE --  report failure
     END IF;
     evaluate_clause (c_i)
     IF this was successful
     THEN RETURN TRUE --  report success
     ELSE i := i + 1 --  try next clause
     END IF
```

```
   END LOOP
```

The next step is to show, how the clause c with head p and tail $p_1 \ldots p_m$. is evaluated: First the formal and actual input parameters are matched. If the matching fails, the entire clause fails, otherwise the predicate procedure p_1 is called. If the call was successful, the predicate procedure p_2 is called, otherwise failure is reported to the caller of the clause, and so on. If the calls of all literal procedures have been successful, success is reported to the caller of the clause. In a short term: tail literals are connected by conjunction. More formally the clause evaluation looks like:

```
evaluate_clause ( c) : BOOLEAN;
   match (in_a , in_f)
   IF   this matching fails
   THEN RETURN FALSE --  report failure
   END IF;
   i := 1;
   LOOP
     IF i > n --  no more tail literals
     THEN RETURN TRUE --  report success
     END IF;
     call_predicate (p_i);
     IF   this call was successful
     THEN i := i + 1 --  next tail literal
     ELSE RETURN FALSE --  report failure
     END IF
   END LOOP
```

Two assumptions for the pattern matching are made here: first, that the actual input term is a ground term before calling the procedure and second that the formal output parameter on return from the call is a ground term. These assumptions are fulfilled, because the way variables may be used in a clause (see section 1.5.9), and the fact that terms constructed during parsing are ground terms.
Action and condition predicates differ only in their behaviour in the case of failure. An action predicate is used in situations, where it is "obvious" that the term transformation may not fail. Hence an action predicate is not allowed to fail, because this failure points to a design error in the specification. If it fails in spite of that, the target program aborts printing an error message.
Condition predicates are used for testing conditions a term may fulfill or not, and hence a condition predicate may succeed or fail.

1.5.12 Global variables

To make it easier to write complex specifications or to deal with global information (for example a compiler's definition table), *Gentle* provides *global variables* and *global tables.*
A *global variable* is declared as:

```
GlobalVarDecls    ::=  'VAR' Type GlobalVariable "."
GlobalVariables   ::=  LargeIdent.
```

Type may be any type. Global variables are accessed with:

```
GlobalVarRead    ::=  GlobalVariable "->" Parameter .
GlobalVarWrite   ::=  GlobalVariable "<-" Parameter .
```

Examples:

```
'VAR' INT Counter.
Counter -> X
Counter <- X+1
'VAR' EXPR ExprVar.
ExprVar <- binary (X,Y,Z)
ExprVar -> binary (XX, YY, ZZ)
```

A global variable must get a value before [2] that value is read.
A generalization of global variables is the *global table* concept. A global table is something like an array in conventional imperative programming languages, except that the space for entries is reserved dynamically. The entries of a global table are accessed using a *key* and the conventional bracket [ ] notation. A global table is declared as:

```
GlobalTableDecl   ::=  'TABLE' ((Type GlobalTable) // "," ) "[" KeyType "]".
GlobalTable       ::=  LargeIdent.
KeyType           ::=  LargeIdent.
```

The *Type* of the *GlobalTable* entries may be any declared type. *KeyType* is introduced as a new opaque type and is used as type of the *key* of an entry in the table. The usual operations on opaque types are not allowed for keys. Access of a global table entry is done with:

```
GlobalTableRead    ::=  GlobalTable "[" KeyVariable "]" "->" Parameter .
GlobalTableWrite   ::=  GlobalTable "[" KeyVariable "]" "<-" Parameter .
KeyVariable        ::=  LocalVariable .
```

where *KeyVariable* is a local variable holding the key. Providing space for a new entry in all tables of the same declaration is done with:

```
GlobalTableNewEntry   ::=  'KEY' KeyType KeyVariable .
```

```
'TABLE' NODEATTR Graph, INT Count [NODE].
'KEY' NODE Node
Graph [Node] <- node (false, Info, Succ1, Succ2, Succ3, Succ4)
Graph [Node] -> node (Mark, Info1, Succ11, Succ21, Succ31, Succ41)
Count [Node] <- X
Count [Node] -> XX
```

KeyType must be an key type of a table declaration. *KeyVariable* is a local variable holding the key for a table access. A global table entry must get a value before [3] that value is read.
The *Parameter* of a global read access and the *KeyVariable* of a *'KEY'* literal is said to be at output position, the *Parameter* of the global write access is at input position (see section 1.5.9).
GlobalTableNewEntry, the write access of global variables and tables is a special kind of action predicate, which never fails. The read access of global variables or tables is a special kind of condition predicate. The *Parameter* is matched with the value stored in the global variable or table, hence a read access may fail.

1.5.13 Side effects in Gentle

Logic predicates usually don't have any side effects. But as *Gentle* allows the use of global variables, tables, and the call of external predicates side effects are possible and sometimes desired. An example is file input and output or handling global data. Notice, side effects of a clause are not "undone", when this clause fails!

1.6 Printing terms

For testing a system it is often useful to visualize the structure of the terms involved. *Gentle* supports this by generating action predicates, which print a complete term onto the standard output device. To use this predicates, the signatures must be declared as follows:

[2] means a time relation
[3] means a time relation

```
'ACTION' print_Type (Type ).
```

Type is the name of any declared type.
For example (see section 1.2) `print_EXPR (binary (plus, var (a), const(3)))` produces:

```
binary(
 plus
 var(
  a
 )
 const(
   3
 )
)
```

While the printing procedures for term types may be generated, those for opaque and types must be implemented by the user, because only the user knows their actual structure (see section 2.11.1).

1.7 The structure of a compiler specification

This section introduces the structure of a typical target specification. The basic idea is that compilation is done in several passes. Terms act as interface between passes. They form the *intermediate languages*. The first step is to read the input and parse it. While doing this, the first intermediate representation of the input is constructed. After parsing has finished the term transformation is started. Each transformation step has input and output parameters, taking the input (i.e. one intermediate language) and transform it to the desired output (i.e. another intermediate language). The last step is the generation of some output, for example writing a file. The following program segment gives an impression of such a specification:

```
'MODULE' example
-- Define the intermediate languages
IR_1 = ..... .  ....  IR_n = ..... .

-- The root of all:
'NONTERM' ROOT.
ROOT : Parse (-> Ir_1)
       Transform_1 (Ir_1 -> IR_2)
       .....
       Transform_n_1 (Ir_n_1 -> IR_n)
       GenerateCode (IR_n)

-- The context free grammar
'NONTERM' Parse (-> IR1).  ....

-- the transformations:
'ACTION' Transform_1 (IR_1 -> IR_2).
   ....
'ACTION' Transform_n_1 (IR_n_1 -> IR_n).

-- Producing some output
'ACTION' GenerateCode (IR_n). ....
```

1.8 Gentle versus Prolog

This section relates *Gentle* to *Prolog*, which is based on Horn-logic theorem proving. A short introduction of Horn-logic is given in [Clocksin *et al* 84, chapter 10], a complete introduction into logic programming is found in [Lloyd 87]. *Gentle* is compared to *Prolog*[Clocksin *et al* 84], which is one implementation of Horn-logic. *Gentle* differs from *Prolog* in four points:

1. *Gentle* restricts the usage of logical variables (see section 1.5.9).

2. The data flow inside of predicates is fixed. For each argument of a predicate its mode must be given, i.e. it is fixed, whether an argument is an input or an output parameter (see section 1.5.6).

3. The *Gentle* proof algorithm differs from that of *Prolog* (restricted backtracking).

4. Due to these restrictions, the unification algorithm is optimized.

1.8.1 Prolog

A *Prolog* program is formed of *variables*, *terms*, *literals*, and *clauses*.
A *term* is either a variable or a structure formed out of terms (i.e. $f(t_1, \ldots, t_n)$), where f is a functor symbol and t_i are terms. *Constants* are structures without any argument. A *ground term* is a term which contains no variables.
A *literal* or *predicate* is a structure $p(t_1, \ldots, t_n)$, where p is a predicate symbol (which are disjoint from variables and function symbols), and t_i are terms. A literal may be negated.
A *Prolog* clause is a Horn clause, i.e. is a disjunction of literals, where at most one literal is positive. The following notation is used:

Prolog clause notation	may be read as:
1. $a_0 : -a_1, \ldots, a_n.$	implication "a_0 is true if a_1 and ...and a_n are true". $(a_0 \vee \neg a_1 \vee \neg a_2 \vee \ldots \vee \neg a_n)$
2. $a_0 : -.$	a_0 is a *fact*, i.e. is always true.
3. $: -a_1, \ldots, a_n.$	question (*goal*): "are a_1 and ...and a_n true" $(\neg a_1 \vee \neg a_2 \vee \ldots \vee \neg a_n)$.

The left hand side of a clause (i.e. to the left from the :-) is called the *head*, the right hand side the *tail* of the clause.
A *Prolog* program consists of several implications and several facts. Executing a *Prolog* program means to state a goal G and to prove that there is a substitution σ such that $\sigma(G)$ is derivable from the program. σ is called an answer substitution.
A simple *Prolog* example is:

```
bachelor (X) :- male (X), unmarried (X).        an implication.
male (charly) :- .                              a fact.
unmarried (charly) :- .
:- bachelor (X).                                the question, to be derived.
```

The resulting answer substitution is: *X/charly*.
A *substitution* σ is a finite set of the form $\{v_1/t_1, \ldots, v_n/t_n\}$, where each v_i is a variable, each t_i is a term distinct from v_i and the variables $v_1, \ldots, v_n$ are distinct. Each element v_i/t_i is called a *binding* for v_i. σ is called a *ground substitution* if all t_i are ground terms. A substitution σ is applied to a formula F, if all occurrences of the variable v_i in F are simultaneously replaced by t_i.
For a set S of literals, a substitution σ is called a *unifier* for S, if $\sigma(S)$ (i.e. σ applied to S) has only one element. A unifier θ for S is called the *most general unifier (mgu)* for S, if for each unifier σ, there exists a substitution γ such that $\sigma(S) = \gamma(\theta(S))$.
A method to prove a goal w.r.t. a set of clauses is *SLD resolution* [Lloyd 87]. A *Prolog* interpreter basically consists of a SLD resolution proof procedure.

prolog_prove $(: -A_1, \ldots, A_m) \rightarrow \sigma$
    -- proves the goal $(: -A_1, \ldots, A_m)$, output is the answer substitution σ.
    $G := : -(A_1, \ldots, A_m)$
    LOOP
        search a clause $H : -L_1, \ldots, L_n$ such that $\exists \theta \in mgu(H, A_1)$
        IF there exists no such clause
        THEN $\sigma := \{\}$; RETURN FAILURE -- G can not be prooven.
        ELSE $G' := \theta(: -L_1, \ldots, L_n, A_2, \ldots, A_m)$
            -- notice if n = 0, then H is a fact.
            prolog_prove (G') $\rightarrow \theta'$
            IF this succeeds
            THEN $\sigma := \theta' \circ \theta$
                RETURN SUCCESS -- G is prooven.
            ELSE -- search a new clause, i.e. backtrack

```
            END IF
        END IF
    END LOOP
```

The clauses are tried in their textual order.

1.8.2 The Gentle proof procedure

Due to the restricted task of *Gentle* the full power of SLD resolution is not needed, and as a result a more efficient implementation is possible. The price one must pay is that the *Gentle* proof algorithm is not complete, i.e. not all possible solutions are found.
The *Gentle* proof procedure differs in two points from the *Prolog* procedure. First not all predicates are allowed to fail (see section 1.5.11). Second backtracking is restricted in the following way:
If in the *prolog_prove* procedure the proof of a literal $A_i, i > 1$ fails, backtracking is performed and a new clause is tried. If this happens in *gentle_prove*, no backtracking is performed, the proof of G fails. Backtracking is done only if the proof of : $-L_1, \ldots, L_n$ fails. That is, once the tail of a clause has been prooven completely, all alternatives for that clause are discarded.

```
gentle_prove (: −A1, ..., Am) → σ
    -- proves the goal (: −A1, ..., Am), output is the answer substitution σ.
    G :=  (: −A1, ..., Am)
    LOOP
        search a clause H : −L1, ..., Ln such that ∃θ ∈ mgu(H, A1)
        IF there exists no such clause
        THEN  σ:= {}; RETURN FAILURE -- G can not be prooven.
        ELSE  G1 := θ(: −L1, ..., Ln)
              gentle_prove (G1) → θ1
              IF this succeeds
              THEN  G2 := θ1 ∘ θ(: −A2, ..., Am)
                    gentle_prove (G2) → θ2
                    σ := θ2 ∘ θ1 ∘ θ
                    RETURN success or failure of this.
              ELSE  -- search a new clause, i.e. backtrack
              END IF
        END IF
    END LOOP
```

The clauses are tried in their textual order.

1.8.3 Optimized unification

Since the data flow is restricted in *Gentle* a simpler unifier may be computed. (It is no limitation to examine the following with only one input and one output parameter)
Such an unifier θ for two *Gentle* terms $P(I_h, O_h)$ and $P(I_g, O_g)$, can be defined as the composition of several substitutions:

$$\theta\{P(I_h, O_h), P(I_g, O_g)\} = \sigma_{out}(\sigma_P(\sigma_{in}(\sigma_C\{P(I_h, O_h), P(I_g, O_g\})))) \tag{1.1}$$

where $P(I_h, O_h)$ is the head of the selected clause, and $P(I_g, O_g)$ the goal to be proven. I denotes the input term and O the output term.
Since the tail literals are evaluated from left to right, and all variables of the input term must be defined, σ_C denotes the substitution, which has bindings for all variables of I_g. σ_{in} binds variables of the formal input parameter to terms of the actual input term. σ_P is the substitution computed by the predicate P, i.e. binds the variables of the formal output parameter. σ_{out} defines the variables of the actual output parameter. These substitutions are defined using the following equations:

$$\sigma_{in}(I_h) = \sigma_C(I_g) \tag{1.2}$$

$$\sigma_{out}(O_g) = \sigma_P(\sigma_{in}(O_h)) \tag{1.3}$$

Notice, the following statements are consequences of the data flow and evaluation strategy rules of *Gentle*:

- I_g, O_g, I_h, O_h are terms.
- I_g and O_g have no common variables[4].
- O_h may use variables defined in I_h.
- $\{I_g, O_g\}$ and $\{I_h, O_h\}$ have no common variables[5].
- $\sigma_C(I_g)$ is a ground term.
- $\sigma_{in}(I_h)$ is a ground term.
- $\sigma_{in}(O_h)$ is a term.
- $\sigma_P(\sigma_{in}(O_h))$ is a ground term.

Now it must be shown that θ is a unifier for $\{P(I_h, O_h), P(I_g, O_g)\}$. By a simple computation using the above notices, one sees that:

$$\theta\{P(I_h, O_h), P(I_g, O_g)\} = \{P(\sigma_C(I_h), \sigma_{out}(O_g))\}$$

It is obvious, that θ is not a most general unifier.
The substitution σ_{in} and σ_{out} as defined by equation 1.2 and equation 1.3 are computed by the pattern matching procedure defined in section 1.5.5 (notice, that one term must be ground for that procedure). If the selected clause C for P looks like:

$$P(I_h, O_h) : P_1(I_1, O_1) \cdots P_n(I_n, O_n)$$

σ_P is computed as:

$$\sigma_P = \sigma_{in_p} \circ \sigma_{out_1} \circ \cdots \circ \sigma_{out_n}$$

I.e. σ_P summarizes all computed variable bindings.
If $P(I_g, O_g)$ is the *j'th* tail literal, then

$$\sigma_C = \sigma_{in_C} \circ \sigma_{out_1} \circ \cdots \circ \sigma_{out_{j-1}}$$

[4] because the use / definition rules for local variables
[5] due to the scoping rules for local variables

Chapter 2

The Gentle Programming Environment User Manual

2.1 Introduction

The *Gentle* tool is used to produce an executable program, which implements the specification. This chapter is the user manual for the *Gentle* programming environment. It is assumed, that the reader knows the *Gentle* language and its technical terms (see chapter 1).
The usage and options of the commands used to start the *Gentle* system are described in the manual page, see appendix A, or try the UNIX command *man gentle*.
Section 2.3 shows what must be done to get an executable compiler. Section 2.4 presents the necessary files and explains their meaning. A *makefile* generator supports the UNIX *make* facility; it is described in section 2.5. Each program has a "main" procedure, the one of *Gentle* is explained in section 2.6. Section 2.7 shows how the executable target program must be called. Section 2.8 deals with the scanner specification, section 2.9 gives some notes on parsing and parsing conflicts. Section 2.11 gives the conventions of implementing your own external predicates. The *Gentle* library is documented in section 2.12. Section 2.13 introduces the example library.
The *Gentle* tool translates *Gentle* specifications into C programs and input for some program generators (scanner and parser generator). The *Gentle* environment uses some features of the UNIX operating system, mainly the *make* facility. For that reasons, the terminology of C and UNIX is used in some places of this manual.

2.2 Directory structure of Gentle

The directory structure of the *Gentle* tool is:

directory names	contents
gentle	*$GENTLE_DIR* must refer to this directory
bin	shell scripts
lib	*Gentle* library
examples	example library
documentation	
install	shell scripts used to install *Gentle*
reuse	the *reuse* libarary
tools-bin	scanner --, parser generator, etc.
tools-lib	library used by scanner, parser etc.
c-src	C sources of *Gentle*
spec	*Gentle* sources of *Gentle*

2.3 How to get an executable compiler

The program generated out of a *Gentle* specification is called the *target system, target program* or simply *target.* To generate a target one may proceed as follows:

Install the *Gentle* system This is usually done once by the system manager. Read the *Gentle* installation notes for that, which are delivered with the distribution tape.

Set up the *Gentle* environment for your target by creating a directory in the file system, which should contain the target system. Copy the *makemake* and *MAIN.c* files from the *Gentle* library (short library) into the target directory. This two files are frames, which must be filled with specific target information. If *MAIN.c* is not intended to be changed, it is not neened to be copied.

Be sure, that the UNIX environment variable *GENTLE_DIR* contains the name of the directory, where the *Gentle* system is located.

Install the library by executing the *makemake* command.

Write the target specification consisting of the *Gentle* "program" and the scanner specification. The example library of the *Gentle* system (see below) may be used to get an impression, how things could look like.

Generate the target. Whenever a new *Gentle* module is created (i.e. a new file containing a part of the target specification is introduced) the *makemake* command must be executed, to produce a new *makefile*, which controls the generating process. The entire generation process is invoked just by executing the UNIX *make* command (without giving parameters). If this process is successful, the target is finished.

The UNIX *make* command is also used to minimize the needed regeneration and recompilation of the target system, if one or more of the input files are changed.

If only a single *Gentle* module should be analyzed, the *g* command may be used (see *Gentle* manual page).

Execute the target program.

2.4 What files are needed for a complete target specification

A complete specification consists of a set of files. Some of these files must be completely written by the user, some must be adapted, and some are used without modifications. Table 2.1 shows these files and their meaning.

Filename	meaning	kind of manipulation
*.g	The modules of the *Gentle* specification, i.e. the grammar of the input language, the definition of terms, and rules for their transformation.	Some must be written by the user, some are taken unchanged from the standard library.
SCANNER.rex	The scanner of the target system is generated using the scanner generator *rex* [Grosch 87]. This file contains the scanner specification, i.e. the input to *rex.*	Must be adapted by the user. Some scanner specifications are contained in the library (see section 2.8).
makemake	To support an automatic generation and (re)compilation process, the UNIX *make* command is used. Input to *make* (a *makefile*) is generated by the *makemake* command.	A frame of *makemake* is contained in the library.
makefile	Generated by *makemake.*	No manipulation by the user needed.
MAIN.c	The main program, which triggers all actions of the target program.	A frame of *MAIN.c* is contained in the library.

Table 2.1: Files needed for a complete specification of a target

2.5 The Makefile generator

The UNIX *make* facility is a powerful tool for maintaining large software systems. It is used to process (i.e. compile, assemble, link, etc.) files to produce the desired system. It is also used to do this job with less redoing as possible. This minimal effort is obtained by defining dependencies of the files and their processing. This information is contained in a file named *makefile*. For more information see the UNIX manuals[1].
The *Gentle* environment offers a *makefile* generator called *makemake*. *makemake* performs three tasks:

1. It installs the library at the first call.
2. Some user defined parameters are set for the target generation process.
3. It generates the actual *makefile* with the actual dependencies for the target.

makemake is implemented as a UNIX (Bourne) shell procedure. Table 2.2 shows the parameters, which the user must set. Notice that *make* uses the Bourne shell, hence use only Bourne shell features.

Parameter name	Meaning
NAME	Name of the target system.
DEST	The directory the target system is located. All files needed for executing the target system are copied to the directory *$DEST/bin*. That place is assumed by the *MAIN* procedure (see section 2.6).
CFLAGS	The C compiler uses different options for controlling the compilation process. This options are specified here. For example *-O* for optimizing the code or *-g* for producing symbolic debugging information.
CPPFLAGS	This flags control the behaviour of the *cpp* preprocessor of the C compiler. The flags are used to compile the *Gentle* library with different options (see section 2.12). Four conditional compilation flags are defined *NR_OF_IDENT_ATTR* (see section 2.12.2) and *USER_DEFINE_OPAQUE* (see section 2.11.1), *Dialog*, and *IgnoreChar*. For more information have a look into the *makemake* file.
USER_SRCS	The files containing the implementation of external *Gentle* predicates (used in the target specification) must be specified for compilation using this variable. The implementation of external library predicates, and the system procedures are given by default.
OTHER_PROGS	If some other programs should be compiled and linked, these programs are specified using this variable. For example the *Gentle* error handling programs *Lister* and *Unlister* (which inserts and removes the error messages into a source file) are managed in this way.

Table 2.2: The *makemake* parameters.

2.6 The MAIN procedure

The "main" procedure of the target system (i.e. the procedure called first when starting the execution of the target program) is contained in the file *MAIN.c* .
Using this *MAIN* program, the scanner may be tested separately, using the conditional compilation feature of the C language. This is done by compiling *MAIN.c* with the *-DTST_SCANNER* option of the C compiler / preprocessor. This option may be specified in *makemake* as an additional *CPPFLAGS* parameter.

[1] or the GNU make manuals

2.7 Executing the target system

The executable target program reads the command line to set some parameters. Its usage is:

target [files ...]

The *files ...* parameter specify the input files for the target program. If no file is given the standard input device is used. If several file names are passed as parameter, the behaviour of the system depends on the EOF (end-of-file) specification given in the scanner description.
Other options may be defined by the user.

2.8 The scanner specification

The scanner of the target system is generated using the scanner generator *rex* [Grosch 87]. This section gives a short introduction to the parts of the scanner, which are needed by the rest of the target system. Please read the *rex* manual for detailed information!
The library contains some scanner specifications for many tokens, for example, identifiers (C, Ada and prolog style), comments (C, Ada, Modula style), numbers (decimal, hexa–decimal integers, fixed and floating point real numbers), strings (C and Modula style), operators ("+", ..., ":=", ...). Handling of more than one input file is also provided. Positional information of the tokens is computed.
The *Gentle* system generates from the *TOKEN* predicates the file *g.TOKENS.h* which *#defines* names for unique token numbers. These names consist of the prefix *g_* and the name of the *TOKEN* predicate. These *#defined* names may be used in the scanner specification.
The scanner specification has several parts:

EXPORT The things contained here are written to the C header file of the generated scanner.

```
# include "g.SCANNER_TYPES.h"
# include "SYS.h"
  extern void ErrorAttribute (); /* (int Token, tScanAttribute *Attribute) */
```

The *C* header file `g.SCANNER_TYPES.h` contains a generated type definition for the attributes (output parameters) token may return. These attributes are usually *Gentle* opaque values and hence they are implemented as pointer to some information. The definition (for tokens, which have at most two output parameters) looks like:

```
# include "Positions.h"

typedef struct {
  tPosition Position;
  long *attr1;
  long *attr2;
} tScanAttribute;
```

The scanner computes the source position automatically. `tPosition` is the type of this information. The opaque type `POS` declared in the library represents the positional information in a coded form.

The names of the attributes are predefined as `attr`*nr*, where *nr* ranges from 1 to the maximal number of attributes a token returns. `attr1` denotes the first attribute of a `'TOKEN'` signature, `attr2` the second, etc.

The procedure `ErrorAttribute` is called, if the parser repairs a syntactic error and inserts a token, which has attributes.

GLOBAL In the *GLOBAL* section procedures, variables, etc. used by the rest of the scanner are declared. The header files of used modules must be included:

```
GLOBAL {
#include <string.h>
#include "g.TOKENS.h"
#include "ERRORS.h"
#include "IDENTS.h"
#include "IO.h"
#include "SYS.h"
}
```

The parser (see section 2.9) does some error recovery and must hence know the defined "error" values of the tokens having attributes. The procedure *ErrorAttributes* may look like:

```
/* procedure returning error attributes                */
/* input is a token, output is the ''error'' attribute. */
void ErrorAttributes (Token, Attribute)
/* we have maximal two attributes  per token */
int Token;
tScanAttribute *Attribute;
{
 switch (Token) {
   case g_IDENTIFIER : Attribute->attr1 =(long*)G_No_IDENT;/*see library module 'IDENTS'*/
                       Attribute->attr2 = 0;
                       break;
   case g_NUMBER : Attribute->attr1     = (long *) G_pool_alloc(sizeof(double));
                                  /* allocate space for a floating number */
                   *(Attribute->attr2)  = 1.0;
                   Attribute->attr2     = 0;
                   break;
   case g_STRINGCONST : Attribute->attr1 = 0;
                        Attribute->attr2 = 0;
                        break;
   default : Attribute->attr1 = 0;
             Attribute->attr2 = 0;
             break;
 }
}
```

LOCAL In the *LOCAL* part, variables, etc. are defined, which are used in the statements of the *RULE*s.

```
char word [256]; char strconst [256]; register long length;
```

BEGIN Initialization of the scanner. If the "IDENTS" moudule is used, it must be initialized here:

```
BEGIN {
  G_IDENTS_init ();
}
```

CLOSE Finalization of the scanner, is empty.

EOF Actions when the end of the input file is reached. If more than one file should be read as input, the following program fragment may be used (see the IO module from section 2.12).

```
if (G_IO_MORE_FILES () == TRUE) {
    /* start reading a new file */
    G_IO_CUR_IN = G_IO_NEXT_FILE ();
    BeginFile (G_IO_NAME (G_IO_CUR_IN));
    /* see library module 'IO' */
}
```

If there is no more file, then the default action is to signal the parser the end–of–file condition.

DEFAULT This section deals with illegal characters.

```
DEFAULT {
  /* if an illegal character is read, an error message is emitted */
  { long pos;
    char word [256];
    g_GET_CUR_POS (&pos); /* returns the current source position, in the coded form*/
    GetWord (word); /* reads the illegal character */
    g_ERROR_TXT ("illegal character '$'", word, pos);
  }
}
```

DEFINE Abbreviations for sets or sequences of characters may be defined here. For example:

```
digit = {0-9} .
```

START Defines states for the scanner, see [Grosch 87, page 8]. `#STD#` is the default state.

RULES This part defines the regular expressions, forming the tokens of the language. The *TOKEN* predicates of the *Gentle* specification are used to *#define* unique numbers for the tokens. The definitions are contained in the file *g.TOKENS.h*. The *#define*d names are *g*_**token_predicate_name**.
For example:

```
#STD# ":="        : { /* 'TOKEN' ASSIGN (-> POS). */
                      g_GET_CUR_POS (&Attribute.attr1);
                      Attribute.attr2 = 0; /* only for initialization */
                      return g_ASSIGN;
                    }
#STD# "+"         : {return g_PLUS;}      /* 'TOKEN' PLUS.      */
#STD# "PROCEDURE" : {return g_PROCEDURE;} /* 'TOKEN' PROCEDURE. */

/* identifiers (Modula - style) */
/* 'TOKEN' IDENTIFIER (-> IDENT, POS). */
#STD# letter (letter | digit ) * :
         {length = GetWord (word);
          g_enter_IDENT(word, length, &(Attribute.attr1));
          /* see library module 'IDENTS' */
          g_GET_CUR_POS (&Attribute.attr2);
          return g_IDENTIFIER;
         }
```

The token `ASSIGN` has the (coded) source position as attribute, the token `IDENTIFIER` has two output parameters: the identifier and the position.

When the generated scanner is compiled using the C compiler, don't bother about the compiler warning *statement not reached*.
If the target system is used interactively, i.e. may read its input form a keyboard, the conditional compilation flag *Dialog* (see section 2.5 should be set. The effect is that when encountering a newline character another blank charcater (this default may be also changed, see *IgnoreChar*) is inserted, to give the scanner the needed look ahead symbol.

2.9 The generated parser

The *TOKEN* and *NONTERM* predicates of a *Gentle* specification are used to generate an LALR parser [Waite *et al* 84]. The parser generator tool is *lalr* [Vielsack 88]. This generated parser handles syntax errors completely by repairing them. The attribute values of inserted tokens are computed by the procedure *ErrorAttributes* as described in the scanner specification (see section 2.8).
If the specified context free grammar has no LALR–conflicts, the user is not bothered with details of that tool. But there are two kinds of grammar conflicts [Waite *et al* 84], *shift reduce* (also called *read-reduce*) and *reduce reduce* conflicts. The presence of conflicts is reported by the parser generator. The file *_Debug* contains a detailed description of the conflict and the default rules how it is solved. Parser generators usually give the user a possibility to solve these conflicts using special "directives" in the grammar specification. *Gentle* does not have such "directives". The other way solving grammar conflicts is to use the default rules of the parser generator. That is the way *Gentle* does it.
lalr provides two default rules: 1. shift–reduce conflicts are solved by shifting the token. 2. reduce–reduce conflicts are solved in the way, that the textual first grammar rule is reduced (see [Vielsack 88, page 16]).

2.10 Generated C code

This section gives an informal insight how the generated *C* [Kernighan *et al* 77] code looks like. The translation of the grammar part to input for a parser generator is not described here. Table 2.3 shows how terms, variables, action and condition predicates are implemented.

Gentle	*C*
Terms	Memory area in the heap. A term is accessed using a pointer to that area.
Pattern matching	Programmed `if` statements.
Local variables	Local variables of procedures, containing terms, i.e. pointers.
Global variables	Global variables, containing a terms, i.e. pointers.
Global tables	Memory areas on the heap. 'KEY' allocates this area, and returns a pointer to it.
Action / condition predicates	*C* functions returning a boolean value, expressing success or failure of the procedure call.
Clauses	The clauses of a predicate form the body of the predicate function.

Table 2.3: Mapping of *Gentle* onto *C* constructs

2.10.1 Terms and pattern matching

Each functor of a term type is mapped to a positive integer value. A term is implemented as a memory area on the heap, and referred by a pointer to that area. The area contains the functor's name, or more exactly, its number, and for each argument a pointer to the argument term. Figure 2.1 shows how the term `f(a,b)` looks like. The arguments of the terms are accessed using the *C* array notation, which is actually pointer arithmetic. For example, the term above is referred by the address `(long*)T`, the first component (i.e. `a`) is referred to by address `(long*)T[1]`. For all constructed terms of a clause the storage is allocated once when the evaluation of the clause starts. The variable `B` contains the base address of the returned memory area.
Remember that at least one of the two terms considered by the pattern matching must be a ground term, this term is stored already in the heap. The non-ground term is not constructed explicitly on the heap, but its functor numbers are compared with the corresponding functor numbers of the ground term. Pattern matching is implemented using direct code for the partial evaluation of the standard top down matching algorithm. For example, testing whether the term stored at location `T` is a the term `f(a,X)` is done by the following code fragment:

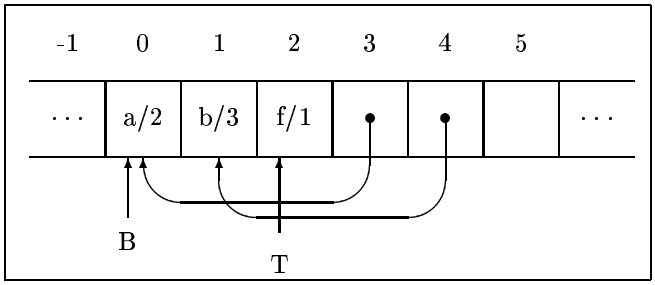

Figure 2.1: Term representation of `f(a,b)`

Pattern matching	
`if (*(long *) T != 1) goto L1;`	If term addressed by `T` is not a `f` term, pattern doesn't match. Continue at location marked with label `L1`.
`if (*(long *) T[1] != 2) goto L1;`	If the first argument of the `f` term is not an `a` term, the pattern doesn't match. The second argument of `f` is not checked, because it is a term variable.
· · ·	Now the term specified by `T` is matched to `f(a,X)`.
`L1 :` · · ·	Failure, the term does not match. Code for that case.

If this pattern matching fails, the program continues at the failure label `L1`. Using the *Gentle* variable `X`, is translated to the access path `(long *)T[2]`.

2.10.2 Action and condition predicates

All clauses of a predicate form the body of the predicate function. The predicate's signature is translated to the function head. Input parameter are passed by value, output parameter by reference. The function returns a boolean value, to indicate the success or failure of the predicate.
Each clause is translated into a piece of code, which has two "exit" points, one for successful evaluation of the clause and one for its failure. The first step of evaluating a clause is to match the actual with the formal input parameters. If this was successful, the predicates of the clause's tail are called. Again, when all calls has been successful, the output terms are constructed and the function is exited reporting success to the caller of this predicate. If one of these steps fail, a jump to the end of the clause's code is performed. If there is another clause, this is tried, otherwise failure is reported to the caller.
Calling a predicate is done in three steps. First, the actual input parameter of the predicate to be called are constructed. Second, the predicate function is called and last but not least the actual and formal output parameters has to be matched. If all these steps have been successful, the predicate call was successful, otherwise a jump to the failure label of the clause containing this call is performed.

2.11 Writing your own external predicates

External predicates of *Gentle* are used as loop holes to escape from the *Gentle* language. These predicates are implemented in another language, for example C. One important usage of external predicates is file input and output.
The user may write his/her own external predicates. For doing this the following conventions must be observed.

Naming conventions: ACTION and CONDITION predicates are called as functions. The procedure name is constructed by appending the predicate name to the prefix *g_*. Most other names (not generated from a specification) of the *Gentle* systems have the prefix *G_*

Parameter passing: Terms are represented as pointers to some entities. Input parameters are passed (as usually in C) "call by value", i.e. the address of the term is passed. Output parameters are passed as "call by reference", i.e. the address of the variable, which is used to store the output is passed. The meaning

of opaque values must be defined by the user. For example the opaque *STRING* type from the *STRINGS* library (see below) is interpreted as *char**, while *INT* is interpreted as a number of type *int*.

Return values: Both ACTION and CONDITION predicates are implemented as functions returning a boolean value, coded as 1 for *TRUE* (success) and 0 for *FALSE* (failure). ACTION predicates must always return *TRUE*. CONDITION predicates may return both values. The values for *TRUE* and *FALSE* are defined in the files *SYS.h, SYS.c* contained in the library.

2.11.1 Printing of opaque values

For testing purposes *Gentle* offers the possibility to print terms onto the standard output device. The printing routines for term types are generated by *Gentle*. For opaque types these routines may be generated or may be written by the user. In the first case a magic unique number is printed, in the latter case, the user may print the information represented by that opaque type. Which of these two ways is used, is specified in the *makemake* generator, using the *CPPFLAGS* parameter. Including *-UUSER_DEFINE_OPAQUE* into *CPPFLAGS* chooses the first, while *-DUSER_DEFINE_OPAQUE* the second way of printing. If the user wants to write print routines for opaque types the procedure interface looks like:

```
g_print_indented_Type(x,n) int x; int n;
/* 'int x' may be replaced by any other type, which has the same 'sizeof' as 'int'. */
```

This corresponds to the action predicate

```
'ACTION' print_indented_Type (Type, INT).
```

This procedure indents the printed entity n units (spaces). x specifies the opaque value. `print_Type (...)` calls this procedure with a zero indentation.
A concrete example is:

```
#include "SYS.h"
/* The routine prints STRING opaque values.   */
/* Used for 'ACTION' print_STRING (STRING).   */
g_print_indented_STRING (x,n)
char *x;
int n;
{G_print_indent(n); /* indent 'n' units   */
 printf("%s\n",x); /* print opaque value */
 return TRUE;
}
```

2.12 The library

The *Gentle* library contains *Gentle* modules, various scanner specifications which may be copied and changed, frames of *makemake* and *MAIN.c*, and some system programs.
This section introduces the *Gentle* predicates, which are usually external predicates, and their usage. The names of the modules are: *IO, IDENTS, ERRORS, STRINGS, MATH, BOOLEAN, ARRAYS,* and *STATISTICS*.
If you have suggestions for library extensions, send the sources to me, they will be made availabe for other users too.

2.12.1 The *IO* module

This module is used for handling file input and output. The target program may read (write) its input (output) from (to) several files, including standard input (output). The files are accessed one after the other, i.e. at a time only one file may be read and one file may be written.
The input works together with the *SCANNER* (see the scanner *EOF* section). Files which should be read are declared using the *IO_DECLARE* routine, which may be called from the *MAIN* program or from the *Gentle* program. A filename may be specified for reading several times using *IO_DECLARE*, but it is read only once. The files are processed in their declaration order. The "-" character is used for input (output) from (to) standard input (output) .
For writing a file the actions shown below are used.
The *IO* module is defined as follows:

```
'MODULE' IO
/*
 * For output to a file:
 * The output file may be opened with 'io_open'. "-" means stdout.
 * If 'io_open' is not called, then stdout is used.
 * The output is buffered, its size (OutBufSize) specified in file 'IO.c'
 * The buffer is written by a call of 'put_nl' or 'put_bf', if it is 90%
 * filled, or if the file is 'stdout'.
 * 'io_close' flushes the buffer unconditionally.
 *
*/

'ACTION' io_open   (STRING).
         -- All output is written to that file.
         -- If the file can't be opened for writing, an error message
         -- is written, and the program is aborted.
         -- "-" means standard output
'ACTION' io_close.
         -- If an outputfile is open,
         -- then the buffer is flushed and the file is closed.
         -- else nothing.
'ACTION' put_s     (STRING). -- writes a string.
'ACTION' put_s1    (STRING). -- writes a string, interpretes escaped char's.
'ACTION' put_d     (INT).    -- writes an integer as decimal.
'ACTION' put_x     (INT).    -- writes an integer as hexadecimal.
'ACTION' put_r     (REAL).   -- writes a real number.
                             -- using     fprintf (f, "%lg", *r);
'ACTION' put_sp    (INT).    -- writes 'n' spaces.
'ACTION' put_nl.             -- writes the new line character, and performs
                             -- 'put_bf'.
'ACTION' put_tab.            -- writes the tab character.
'ACTION' put_qt.             -- writes the (double) quote ".
'ACTION' put_str   (STRING). -- writes a string surrounded with
                             -- double quotes.
'ACTION' put_str1  (STRING). -- writes a string surrounded with
                             -- double quotes, interpretes escaped char's.
'ACTION' put_ident (IDENT).  -- writes the textual representation of IDENT
'ACTION' put_pos   (POS).    -- writes the position.
'ACTION' put_bf.             -- if the file is "stdout", then the buffer
                             -- is written to 'stdout';
                             -- if the file is not 'stdout' the buffer is
                             -- written, if it is 90% filled.

-- specifying input files:
'ACTION' IO_DECLARE (STRING). -- specifies another source file for input.

'ACTION' IO_NAME (INT -> STRING).
-- Returns for a file number its string representation.
```

```
-- The file number may be returned from ERRORS.
```

2.12.2 The *IDENTS* module

The *IDENTS* module provides general identifier handling routines. The scanner usually accepts strings (sequences of characters) representing program identifiers. The parser does not use the string representation of these identifiers but a unique integer value, referring to that string. The *enter_IDENT* action computes such a unique number out of a character sequence, using a hash function. The condition *Eq_IDENT* is used for testing identifiers for equality. *get_IDENT_name* returns the characters of the identifier. The unique "unknown" identifier is returned using *get_No_IDENT*. Somentimes one needs some kind of "new" identifiers, which are not present in the analyzed program. Each call of *generate_IDENT* returns such a new identifier.
A special feature of this module is to handle additional attributes of identifiers. The string representation of an identifier is one of them, which is handled by default. Another attribute may be the meaning of the identifier in the analyzed program. Using this feature a simple definition table of identifiers is manipulated by predicates *DEF_IDENT_ATTR* and *GET_IDENT_ATTR*, which are implemented in C but not declared by default in *Gentle*. For an example usage of that see *minilax* and *hoc* from the example library.
If this feature is used, the *CPPFLAGS* parameter of *makemake* must be set to
-DNR_OF_IDENT_ATTR=xx, where *xx* is the number of additional attributes.
The *IDENTS* module is defined as follows:

```
'MODULE' IDENTS
'TYPE' IDENT.

-- general
'ACTION'    enter_IDENT (STRING, INT -> IDENT).
            -- define a STRING with INT characters as IDENT.
'ACTION'    generate_IDENT (-> IDENT).
            -- each call generates a new unique IDENT.
'CONDITION' Eq_IDENT (IDENT, IDENT).
            -- succeeds iff the IDENT's are equal.
'CONDITION' NotEq_IDENT (IDENT, IDENT).
            -- succeeds iff the IDENT's are not equal.
'ACTION'    get_IDENT_name (IDENT -> STRING).
            -- returns the textual representation of the IDENT.
'ACTION'    get_No_IDENT (-> IDENT).
            -- returns No_IDENT

'ACTION'    print_IDENT (IDENT).
            -- write id to stdout

/* To each identifier a number of attributes may be attached.
 * To use this feature, the user has to define the following type and
 * action/condition.
 * The attributes are accessed by numbers.
 * The maximal number of attributes must be defined when compiling the system.
 * This is done by defining 'NR_OF_IDENT_ATTR' with that number.
 * The attributes are accessed with numbers in the range [1..NR_OF_IDENT_ATTR].
 * 'GET_IDENT_ATTR (Id, Nr -> Attr)' fails, iff 'DEF_IDENT_ATTR (Id, Nr, Attr)'
 * was not called before for that identifier and number, i.e. the identifer has
 * no value for that attribute.
 *
 *  IDENT_ATTR =
 *      some_user_defined_things
 *      .
 * or define IDENT_ATTR as an opaque type.
 *
 * 'ACTION'    DEF_IDENT_ATTR (IDENT, INT, IDENT_ATTR).
 * 'CONDITION' GET_IDENT_ATTR (IDENT, INT -> IDENT_ATTR).
 *
 * Another type name may be used.
```

```
 * A disadvantage of this method is, that all attributes must have the same
 * type (IDENT_ATTR) , but all external C procedures are already implemented.
 *
 * Another way is:
 * Declare types:
 *
 * IDENT_ATTR_1 = ... .
 * IDENT_ATTR_2 = ... .
 *  ....
 * IDENT_ATTR_n = ... .
 *
 * Declare external predicates:
 *
 * 'ACTION'    DEF_IDENT_ATTR_1 (IDENT, IDENT_ATTR_1).
 * ...
 * 'ACTION'    DEF_IDENT_ATTR_n (IDENT, IDENT_ATTR_n).
 * 'CONDITION' GET_IDENT_ATTR_1 (IDENT -> IDENT_ATTR_1).
 * ...
 * 'CONDITION' GET_IDENT_ATTR_n (IDENT -> IDENT_ATTR_n).
 *
 * and implement this external predicates in C as:
 * (Remember, that all opaque values are mapped to 'long'.)
 *
 * BOOL g_DEF_IDENT_ATTR_1 (ident, attr)
 * G_IDENT ident; long attr;
 * {
 *    return g_DEF_IDENT_ATTR (ident, 1, attr);
 * }
 *
 * ....
 *
 * BOOL g_GET_IDENT_ATTR_1 (ident, attr)
 * G_IDENT ident; long *attr;
 * {
 *    return g_GET_IDENT_ATTR (ident, 1, attr);
 * }
 *
 * ....
 *
*/
```

2.12.3 The *ERRORS* module

This module manages error messages emitted by the generated program while analyzing its input. The scanner computes for each token its position (filename, line, and column number). The predicate *GET_CUR_POS* returns the position of the last token coded into an integer. The problem during parsing is that action predicates must follow all nonterminal or token predicates, and hence one can get (using this predicate) only the position of the last token of that grammar rule. A better way is the introduction of a grammar rule *POS* as follows:

```
'NONTERM' POS (-> POS).
POS (-> P): GET_CUR_POS (-> P).
```

This nonterminal predicate may now be used before each token / nonterminal predicate to determine its position. The *Gentle* system uses two programs *Lister.c* and *Unlister.c* which are used to merge (remove) the error messages with (from) the source file.
The *ERROR* module is defined as follows:

```
'MODULE' ERRORS
/* This functions are used to handle error messages emitted by the generated
 * program.
 * An error message consists of
```

```
 * - a position in the source file, where the error is raised
 * - a message (string)
 * - auxiliary information, like names of identifiers etc.
 * The position of an error message is specified by a triple
 * - file : the file containing the source
 * - line number
 * - column number
 * Several source files may be handled. The error messages are written to
 * files named 'ERRORS-<name of source file>-ERRORS'
 * If the source file is unknown, the error message is written to 'stderr'.
 * The auxiliary information is merged into 'message'. The position in the
 * message string is specified as following:
 * The character   $     specifies the position of strings.
 * The character   @     specifies the position of identifiers.
*/

'TYPE' POS. -- position in source file

'CONDITION' IS_ERROR_OCCURED.
            -- Succeeds iff an error message was emitted before.
'CONDITION' NO_ERROR_OCCURED.
            -- Succeeds iff no error message was emitted before.
'ACTION'    GET_UNDEF_POS (-> POS).
            -- Returns the undefined position. Also the integer 0 is allowed.
'ACTION'    ERROR       (STRING, POS).
            -- Emits an error message.
'ACTION'    ERROR_TXT   (Msg : STRING, Txt: STRING,  POS).
            -- Emits an error message and replaces '$' in Msg by Txt.
'ACTION'    ERROR_IDENT (STRING, IDENT, POS).
            -- Emits an error message and replaces '@' by the textual
            -- representation of the IDENT.
'ACTION'    ERR_CVT_TO_POS   (File : INT, Line : INT, Col : INT -> POS).
            -- Converts filenumber/line/col information into a postion.
'ACTION'    ERR_CVT_FROM_POS (POS -> File : INT, Line : INT, Col : INT).
            -- Converts a position into filenumber/line/col information.
'ACTION'    GET_CUR_POS (-> POS).
            -- Returns position of the current parsed token in the source text
'CONDITION' Eq_POS (Pos1 : POS, Pos2 : POS).
            -- Checks, whether Pos1 is equal to Pos 2. (file, line and column)
'ACTION'    print_POS (POS).
            -- Write pos to stdout.

-- abnormal program termination
-- a message is written to 'stderr', 'io_close' is called and then
-- 'exit (n)' or 'abort()' is called.
'ACTION' EXIT  (STRING, INT).
'ACTION' ABORT (STRING).
```

2.12.4 The *STRINGS* module

Gentle defines a sequence of characters as a *STRING*, in the sense of C strings. *Gentle* itself allows string constants in a *Gentle* specification.
The *STRINGS* module is defined as follows:

```
'MODULE' STRINGS
'TYPE' STRING.

'ACTION'    STRING_concat (S1 : STRING, S2 : STRING -> S3 : STRING).
            -- S3 := S1 & S2
'ACTION'    STRING_length (STRING -> INT).
'ACTION'    STRING_norm (Str : STRING, Size: INT -> STRING).
```

```
            -- makes 'Str' at least 'Size' characters long, by adding spaces.

-- The following conditions compare the characters of the two strings.
'CONDITION' Eq_STRING (Str1 : STRING, Str2 : STRING).
'CONDITION' NotEq_STRING (Str1 : STRING, Str2 : STRING).

-- Str1 < (>)  Str2 in their lexicographic order
'CONDITION' Less_STRING (Str1 : STRING, Str2 : STRING).
'CONDITION' Greater_STRING (Str1 : STRING, Str2 : STRING).

'ACTION'    print_STRING (STRING).

'ACTION'    INT_2_STRING (INT -> STRING).
-- converts the integer value into decimal string representation
```

2.12.5 The *MATH* module

A *Gentle* specification only supports simple integer arithmetic. Compiling or interpreting a program often needs a more complex arithmetic. *Gentle*'s build in integer arithmetic is a signed 32 bit arithmetic (on 32 bit machines). The floating point arithmetic supported by this module has double precision.
The module *MATH* is defined as follows:

```
'MODULE' MATH
/* The constants used as minimum and maximum values for short, int, and long
 * integer values are defined in MATH.h
 * Current values are:
 * min_shortint =        -128      8 bit signed integer
 * max_shortint =         127
 * min_integer  =      -32768     16 bit signed integer
 * max_integer  =       32767
 * min_longint  =  0x80000000     32 bit signed integer
 * max_longint  =  0x7FFFFFFF
 *
 * The Gentle built-in arithmetic is a 32 bit signed arithmetic.
 * Real arithmetic is done with double precision.
*/

'TYPE' INT.
'TYPE' REAL.

-- return the lower and upper bound of the given integer type
'ACTION'    ShortIntMinMax (-> Min : INT, Max : INT).
'ACTION'    IntMinMax      (-> Min : INT, Max : INT).
'ACTION'    LongMinMax     (-> Min : INT, Max : INT).

-- succeeds, iff the constant fits into that range
'CONDITION' Is_ShortInt (INT).
'CONDITION' Is_Integer  (INT).
'CONDITION' Is_LongInt  (INT).

-- integer / real comparision
'CONDITION' Eq_Int         (Op1 : INT, Op2 : INT).  -- Op1 == Op2
'CONDITION' NotEq_Int      (Op1 : INT, Op2 : INT).  -- Op1 != Op2
'CONDITION' Less_Int       (Op1 : INT, Op2 : INT).  -- Op1 <  Op2
'CONDITION' LessEq_Int     (Op1 : INT, Op2 : INT).  -- Op1 <= Op2
'CONDITION' Greater_Int    (Op1 : INT, Op2 : INT).  -- Op1 >  Op2
'CONDITION' GreaterEq_Int  (Op1 : INT, Op2 : INT).  -- Op1 >= Op2

'CONDITION' Eq_Real        (Op1 : REAL, Op2 : REAL). -- Op1 == Op2
'CONDITION' NotEq_Real     (Op1 : REAL, Op2 : REAL). -- Op1 != Op2
'CONDITION' Less_Real      (Op1 : REAL, Op2 : REAL). -- Op1 <  Op2
```

```
'CONDITION' LessEq_Real    (Op1 : REAL, Op2 : REAL). -- Op1 <= Op2
'CONDITION' Greater_Real   (Op1 : REAL, Op2 : REAL). -- Op1 >  Op2
'CONDITION' GreaterEq_Real (Op1 : REAL, Op2 : REAL). -- Op1 >= Op2

-- This constant arithmetic checks operands and the result for over/underflow
-- Error messages are emitted and 1 / 1.0 is returned in that case.
-- Result := Op1 <operand> Op2

'ACTION' Add_Int    (POS, Op1 : INT,  Op2: INT  -> INT).
'ACTION' Sub_Int    (POS, Op1 : INT,  Op2: INT  -> Result : INT).
'ACTION' Mult_Int   (POS, Op1 : INT,  Op2: INT  -> Result : INT).
'ACTION' Mod_Int    (POS, Op1 : INT,  Op2: INT  -> Result : INT).
'ACTION' Div_Int    (POS, Op1 : INT,  Op2: INT  -> Result : INT).

'ACTION' Add_Real   (POS, Op1 : REAL, Op2: REAL -> Result : REAL).
'ACTION' Sub_Real   (POS, Op1 : REAL, Op2: REAL -> Result : REAL).
'ACTION' Mult_Real  (POS, Op1 : REAL, Op2: REAL -> Result : REAL).
'ACTION' Div_Real   (POS, Op1 : REAL, Op2: REAL -> Result : REAL).
'ACTION' Power_Real (POS, Op1 : REAL, Op2: REAL -> Result : REAL).
'ACTION' Zero_Real  (-> REAL).

'ACTION' Cvt_Int_To_Real (POS, INT -> REAL).
'ACTION' Cvt_Real_To_Int (POS, REAL -> INT).

-- trigonometric functions
'ACTION' math_sin   (POS, REAL -> REAL).
'ACTION' math_cos   (POS, REAL -> REAL).
'ACTION' math_atan  (POS, REAL -> REAL).
'ACTION' math_exp   (POS, REAL -> REAL).
'ACTION' math_log   (POS, REAL -> REAL).
'ACTION' math_log10 (POS, REAL -> REAL).
'ACTION' math_sqrt  (POS, REAL -> REAL).
'ACTION' math_int   (POS, REAL -> REAL).
'ACTION' math_abs   (POS, REAL -> REAL).

-- common constants
'ACTION' math_PI_value     (-> REAL).  -- Constant pi
'ACTION' math_E_value      (-> REAL).  -- Base of natural logarithms
'ACTION' math_GAMMA_value  (-> REAL).  -- Euler-Mascheroni constant
'ACTION' math_DEG_value    (-> REAL).  -- Degrees per radian
'ACTION' math_PHI_value    (-> REAL).  -- Golden Ratio

'ACTION' Align (INT, INT -> INT).
-- aligns a value to a given bound
-- Align (bound, src -> result) computes:
-- rem := src MOD bound;
-- IF rem = 0 THEN result := src ELSE result := src + (bound - rem) END

-- printing objects of opaque types on to stdout.
'ACTION' print_INT  (INT).
'ACTION' print_REAL (REAL).
```

2.12.6 The *BOOLEAN* module

The *BOOLEAN* defines the boolean values (*true* and *false*) and operation on these values.
The *BOOLEAN* module is defined as follows:

```
'MODULE' BOOLEAN
BOOLEAN =
        true,
        false
```

```
        .

'CONDITION' is_true (BOOLEAN).
is_true (true)  : .

'CONDITION' is_false (BOOLEAN).
is_false (false)  : .

'CONDITION' Eq_BOOLEAN (BOOLEAN, BOOLEAN).
Eq_BOOLEAN (true, true) : .
Eq_BOOLEAN (false, false) : .

'ACTION' and (BOOLEAN, BOOLEAN -> BOOLEAN).
and (true, true -> true) : .
and (X, Y -> false) : .

'ACTION' or (BOOLEAN, BOOLEAN -> BOOLEAN).
or (false, false -> false) : .
or (X, Y  -> true) : .

'ACTION' not (BOOLEAN -> BOOLEAN).
not (true -> false) : .
not (false -> true) : .

'ACTION' print_BOOLEAN (BOOLEAN).
```

2.12.7 The *ARRAYS* module

The *ARRAYS* offers fixed sized integer indexed arrays. The elements of the array are terms.
The *ARRAYS* module is defined as follows:

```
'MODULE' ARRAYS
/*
 * This module provides a general fixed size array data type.
 * Its index type is "INT", its element type may be any Gentle type.
 *
 * "ARRAY_new" :        Creates a fixed size array. Its index range is
 *                      ["Lower_bound" .. "Upper_bound"].
 *                      Element type may be any Gentle opaque or term type.
 * "ARRAY_dispose" :    Removes the (storage of the) array.
 *                      Further use if the array may result in a program
 *                      abortion.
 * "ARRAY_assign" :     "Array [Index] := Value"
 *                      The "Value" is assigned to the "Array" element with
 *                      the given "Index". "Index" must be in the
 *                      range [Lower_bound .. Upper_bound], specified by
 *                      "ARRAY_new" for this "Array". Otherwise the program
 *                      is aborted and an error message is emited.
 * "ARRAY_get" :        "Value := Array [Index]"
 *                      "Value" is value of the "Array" element with
 *                      the given "Index" assigned. "Index" must be in the
 *                      range [Lower_bound .. Upper_bound], specified by
 *                      "ARRAY_new" for this "Array". Otherwise the program
 *                      is aborted and an error message is emited.
 *
 * The user of this module has to declare the signatures of "ARRAY_assign" and
 * "ARRAY_get" predicates using the its own element type.
 * If several array types (i.e. with different element type) are needed, for
 * each type predicate signatures for the assign and get operation (with
 * different names) must be declared.
 * The "ARRAY_assign" and "ARRAY_get" predicate are already implemented as
```

```
 * external predicates in C.
 * If other names are used, a small C procedure must be written, to call
 * "g_ARRAY_assign" and "g_ARRAY_get".
 * For example:
 *
 * ELEM_1 = ... .
 * ELEM_2 = ... .
 * 'ACTION' assign_1  (Array : ARRAY, Index : INT,   Value : ELEM_1).
 * 'ACTION' get_1     (Array : ARRAY, Index : INT -> Value : ELEM_1).
 * 'ACTION' assign_2  (Array : ARRAY, Index : INT,   Value : ELEM_2).
 * 'ACTION' get_2     (Array : ARRAY, Index : INT -> Value : ELEM_2).
 *
 * and implement this external predicates in C as:
 *
 * #include "ARRAY.h"
 * #include "SYS.h"
 *
 * BOOL g_assign_1  (Array, Index, Value)
 * int    Index;
 * tARRAY Array;
 * long   Value;
 * {
 *    g_ARRAY_assign (Array, Index, Value);
 *    return TRUE;
 * }
 *
 * BOOL g_get_1 (Array, Index, Value)
 * int    Index;
 * tARRAY Array;
 * long   *Value;
 * {
 *    g_ARRAY_assign (Array, Index, Value);
 *    return TRUE;
 * }
 *
 * and the same for "g_assign_2" and "g_get_2"
 *
 * If the "-DUSER_DEFINE_OPAQUE" option is set in the "makemake" command,
 * printing of an ARRAY is possible by declaring "'ACTION' print_ARRAY (ARRAY)"
 * (as usual for all types).
 * The lower bound and upper bound and all array elements are printed.
 * The default case for printing array elements is to print their "address".
 * The user may specify another print routine for the elements, by assigning
 * the  C pointer to function variable
 *      BOOL (*G_print_indented_ARRAY_ELEMENT) ()
 * the printing routine (see files "ARRAYS.h" and "ARRAYS.c").
 * This assignment may be done e.g. in "MAIN.c"
 *
 */

'TYPE' ARRAY.

'ACTION' ARRAY_new     (Lower_bound : INT, Upper_bound : INT -> Array : ARRAY).
'ACTION' ARRAY_dispose (Array : ARRAY).

/*
 * ELEMENT =
 *      some_user_defined_things
 *      .
 * or define ELEMENT as an opaque type.
```

```
 * Another type name then ELEMENT may be used.
 *
 * 'ACTION' ARRAY_assign  (Array : ARRAY, Index : INT,   Value : ELEMENT).
 * 'ACTION' ARRAY_get     (Array : ARRAY, Index : INT -> Value : ELEMENT).
*/
```

2.12.8 The *STATISTICS* module

The *STATISTICS* module collects information about the UNIX process running the target program. This information (user time, memory requirements, page faults, etc.) is printed to the standard output device. The *STATISTICS* module is defined as follows:

```
'MODULE' STATISTICS
/* Collects and prints information about used times, used storage, page faults,
 * etc. of the current UNIX process to standard output.
 * To start the meassurement, call 'STATISTICS_init'.
 * To print the information, call 'STATISTICS_show (Message)', the 'Message'
 * is printed before.  'STATISTICS_show' may be called several times after the
 * last 'STATISTICS_init' call.
 * The output looks like:
 *
 * show statistics: message
 * Wed Nov 14 18:58:20 1990
 *
 *           Real time:      3.280 sec
 *           User time:      2.880 sec
 *         System time:      0.340 sec
 *          MemorySize:    312.000 Kb
 *   Major page faults: 0
 *   Minor page faults: 46
 *   Times swapped out: 0
 *         File Inputs: 0
 *        File Outputs: 1
 *             Signals: 0
 *        Wait for I/O: 22
*/

'ACTION' STATISTICS_init.
'ACTION' STATISTICS_show (Message : STRING).
```

2.13 The example library

Together with the *Gentle* system a set of several examples is distributed. Some are small toys, some are complete compilers, interpreters, and text analyzers. These examples are contained in the directory $*GENTLE_DIR/examples*.

glint Analyses *Gentle* programs and extracts information out of a *Gentle* specification. (Specified grammar in BNF notation, cross-reference listing, warnings, etc.)

hoc *Hoc* [Kernighan *et al* 78] is a simple programmable interpreter for floating point expressions. It has C–style control flow, function definition and the usual numerical built-in functions such as cosine and logarithm.

minilax The programming language MiniLAX (Mini LAnguage eXample) is a Pascal relative. To be more specific, it is a subset of the example language LAX used in the text book [Waite *et al* 84]. MiniLAX contains a carefully selected set of language concepts relevant for compiler construction: types, type coercion, overloaded operators, arrays, procedures with reference and value parameters, nested scopes. A compiler for MiniLAX and target processor MC 68020 (Motorola) is specified.

tpnet Shows how TABLE's may be used to represent graphs. The program is used to specify a network of Transputers and computes the minimal spanning connection tree, which may be used to down load code to a Transputer net.

simple A compiler for a simple language, having only assignments and expressions and a simple idealized processor (*bsp1*). Simple constant folding (*bsp2*).

Chapter 3

Writing an Interpreter Using Gentle

3.1 Introduction

This chapter presents the development of a more complex *Gentle* system, than the examples given in the language reference manual. The goal of this tutorial is the specification of an interpreter for *Hoc*. *Hoc* [Kernighan *et al* 78] is an interactive language for floating point arithmetic. It has C–style control flow, function definition, and the usual numerical built–in functions such as cosine and logarithm.
The way this tutorial proceeds is analogous to that presented in [Kernighan *et al* 78]. The interpreter is developed in eight steps, starting with a simple integer expression interpreter, going over to floating point arithmetic and control flow constructs to the complete *Hoc* interpreter. Each step adds new difficulties and solutions. On this way most of the written specification of earlier steps may be reused.
First the *Hoc* language is defined in section 3.2. A first impression of a *Gentle* specification gives section 3.3 by implementing a simple integer expression interpreter (hoc0). The first real step towards *Hoc* is presented in section 3.4, where the complete floating point arithmetic interpreter is specified (hoc1). The next two sections add variables and the standard functions to the interpreter (hoc2, hoc3). Hoc4 in section 3.7 changes the interpreter significantly, by introducing two–pass interpreting, i.e. first construct an intermediate representation of the program and then interprete it. Section 3.8 includes control flow constructs, like conditionals and loops. Section 3.9 changes the interpretation of loops from recursion into a loop over cyclic graphs. The complete *Hoc* interpreter is given in section 3.10 (hoc6).
Together with the *Gentle* system, an example library is distributed. This library contains in the directory $*GENTLE_DIR/example/hoc* all steps of this development. The steps are named *hoc0* ... *hoc6* and the interpreters for the sub–languages are contained in subdirectories with that names.
An executable *Hoc* interpreter is contained in the example library distributed together with *Gentle*. For its usage refer to the manual pages (i.e. try *man hoc*).

3.2 HOC Language Reference Manual

Hoc – An Interactive Language For Floating Point Arithmetic[1]
Brian Kernighan
Rob Pike

> *Hoc* is a simple programmable interpreter for floating point expressions. It has C–style control flow, function definition and the usual numerical built–in functions such as cosine and logarithm.

3.2.1 Expressions

Hoc[2] is an expression language, much like C: although there are several control–flow statements, most statements such as assignments are expressions whose value is disregarded. For example, the assignment operator = assigns

[1] published in [Kernighan *et al* 78]
[2] some minor changes of the *Gentle* implementation are marked with †

the value of its right operand to its left operand, and yields the value, so multiple assignments work. The expression grammer is:

```
expr:   number
      | variable
      | ( expr )
      | expr binop expr
      | unop expr
      | function ( arguments ) .
```

Numbers are floating point. The input format is that recognized by *scanf* (3): digits, decimal point, digits, *e* or *E*, signed exponent. At least one digit or decimal point must be present; the other components are optional. Variable names are formed from a letter followed by a string of letters and numbers[3]. *binop* refers to binary operators such as addition or logical comparision; *unop* refers to the two negation operators, "!" (logical negation, "not") and "-" (arithmetic negation, sign change). Table 3.1 lists the operators.

```
^        exponentation (FORTRAN **), right associative
!  -     (unary) logical and arithmetic negation
*        multiplication, division
+  -     addition, subtraction
>  >=    relational operators: greater, greater or equal
<  <=          less, less or equal
== !=          equal, not equal (all the same preceedence)
&&     logical AND (both operands always evaluated)
||       logical OR (both operands always evaluated)
=        assignment, right associative
```

Table 3.1: Operators, in decreasing order of precedence

Functions, as decribed later, may be defined by the user. Function arguments are expressions separated by commas. There are also a number of built–in functions, all of which take a single argument, described in Table 3.2.

`abs (x)`	$\|x\|$, absolute value of x
`atan (x)`	arc tangent of x
`cos (x)`	cos (x), cosine of x
`exp (x)`	e^x, exponential of x
`int (x)`	interger part of x, truncated towards zero
`log (x)`	log (x) logarithm base e of x
`log10 (x)`	log (x) logarithm base 10 of x
`sin (x)`	sin (x), sine of x
`sqrt (x)`	$\sqrt{x}, x^{1/2}$

Table 3.2: Built-in Functions

Logical expressions have values 1.0 (true) and 0.0 (false). As in C, any non–zero value is taken to be true. As always the case with floating point numbers, equality comparisons are inherently suspect.
Hoc also has a few built–in constants, shown in Table 3.3.

[3]and the _ (underscore) character †

3.2.2 Statements and Control Flow

Hoc statements have the following grammer:

```
stmt:  expr
     | variable = expr
     |  procedure ( arglist )
     |  while ( expr ) stmt
     |  if ( expr ) stmt
     |  if ( expr ) stmt else stmt
     |  { stmtlist }
     |  print expr-list
     |  return optional--expr .
stmtlist:  (nothing)
         | stmtlist stmt .
```

An assignment is parsed by default as a statement rather than an expression, so assignments typed interactively do not print their value.
Note that semicolons are not special to *Hoc* : statements are terminated by newlines[4]
This causes some peculiar behaviour. The following are legal *if* statements:

```
if (x < 0) print (y) else print (z)
if (x < 0)
        print (y)
 else
        print (z)
```

In the second example, the braces are mandatory: the newline after the *if* would terminate the statement and produce a syntax error were brace omitted.
The syntax and semantics of *Hoc* control flow facilities are bassically the same as in C. The *while* and the *if* statements are just as in C, except there are no *break* or *continue* statements.

3.2.3 Input and Output: *read* and *print*

The input function *read*, like the other built–ins, takes a single argument. Unlike the built–ins, though, the argument is not an expression: it is the name of a variable. The next number (as defined above) is read from the standard input and assigned to the named variable. The return value of *read* is 1 (true) if a variable was read, and 0 (false) if *read* encountered end of file or an error.
Output is generated with the *print* statement. The arguments to *print* are a comma separated list of expressions and strings in double quotes, as in C. Newlines must be supplied; they are never provided automatically by *print*.
Note that *read* is a special built–in function, and therefore takes a single parenthesized argument, while *print* is a statement that takes a comma–separated unparenthesized list:

[4] All chacraters after -- up to the end of the line are treates as comment. †

DEG	57.29577951308232087680	$180/\pi$, degrees per radian
E	2.71828182845904523536	e, base of natural logarithms
GAMMA	0.57721566490153286060	γ, Euler-Mascheroni constant
PHI	1.61803398874989484820	$(\sqrt{5}+1)/2$, the golden ratio
PI	3.14159265358979323846	π, circular transcendental number

Table 3.3: Built-in Constants

```
while (read (x))
        print "value is ", x, "\n"
```

3.2.4 Functions and Procedures

Functions and procedures are distinct in *Hoc* , although they are defined by the same mechanism. This distinction is simply for run–time checking: it is an error for a procedure to return a value, and for a function *not* to return one.
The definition syntax is:

```
function:  func name () stmt .
procedure: proc name () stmt .
```

name may be the name of any variable – built–in functions are excluded. The definition, up to the opening brace or statement, must be on one line, as with the *if* statement above.
Unlike C, the body of a function or procedure may be any statement, not necessarily a compound (brace-enclosed) statement. Since semicolons have no meaning in *Hoc* , a null procedure body is formed by an empty pair of braces.
Functions and procedures may take arguments, separated by commas, when invoked. Arguments are referred to as in the shell: *$3* refers to the third (1-indexed) argument. They are passed by value and within functions are semantically equivalent to variables. It is an error to refer to an argument numbered greater than the number of arguments passed to the routine. The error checking is done dynamically, however, so a routine may have variable numbers of arguments if initial arguments affect the number of arguments to be referenced (as in C's *printf*).
Functions and procedures may recurse, but the stack has limited depth (about a hundred calls)[5]. The following shows a *Hoc* definition of Ackermann's function:

```
func ackermann ()
        if ($1 == 0) return $2+1
        if ($2 == 0) return ackermann ($1-1, 1)
        return ackermann ($1-1, ackermann ($1, $2-1))
ackermann (3,2)
        29
ackermann (3,3)
        61
ackermann (3,4)
        125
```

3.2.5 Examples

Stirling's formula:

$n! \sim \sqrt{2n\ \pi}(n/e)\hat{}n(1 + \frac{1}{12n})$

```
func stirling ()
        return sqrt(2*$1*PI) * ($1/E)^$1*(1 + 1/(12*$1))
stirling (10)
        3628684.7
stirling (20)
        2.4328818e+18
```

Factorial function, n!:

[5] In the *Gentle* implementation, this is limited only by the amount of memory available †.

```
func fac1 ()
        if ($1 <= 0) return 1 else return $1 * fac ($1-1)
```

Ratio of factorial to Stirling approximation:

```
i = 9
while ((i = i+1) <= 20)
        print i, " ", fac (i) / stirling (i), "\n"
10      1.0000318
11      1.0000265
12      1.0000224
13      1.0000192
14      1.0000166
15      1.0000146
16      1.0000128
17      1.0000114
18      1.0000102
19      1.0000092
20      1.0000083
```

3.3 An integer expression interpreter, Hoc0

The first language reads integer expressions with + - * / operations from a file and computes the result. The built–in integer arithmetic of the *Gentle* system is used to compute the result, while parsing proceeds. An expression is terminated by the newline character. The operators are left associative.

3.3.1 The scanner

The used scanner generator *rex* and its input syntax is given in [Grosch 87].
The scanner specification is contained in file *SCANNER.rex* . Since only integer numbers are processed, only the *NUMBER* token has a single attribute. The error attribute computing procedure looks like:

```
/* procedure returning error attributes */
void ErrorAttribute (Token, Attribute)
 int Token;
 tScanAttribute *Attribute;
{
 switch (Token) {
  case g_NUMBER          : Attribute->attr1  = 0; break;
  default                : Attribute->attr1  = 0; break;
 }
}
```

The syntax of the description of the regular expressions is defined in [Grosch 87]. Now the regular expressions for the tokens must be defined:

```
RULES
/* new line character */
#STD#   \n                      : { yyEol (0); return g_NL;}
/* integer numbers */
#STD# {0-9} +                   : {GetWord (word);
                                   sscanf (word, "%d", &Attribute.attr1);
                                   /* sscanf converts numbers contained in a
                                      sequence of characters into a numerical
                                      value. It is contained in the C library
```

```
                                */
                                return g_NUMBER;}
/* operators */
#STD# "+"                      : {return g_PLUS;}
#STD# "-"                      : {return g_MINUS;}
#STD# "*"                      : {return g_MULT;}
#STD# "/"                      : {return g_DIV;}
#STD# "("                      : {return g_LEFTPAR;}
#STD# ")"                      : {return g_RIGHTPAR;}
```

The procedure `yyEol` must be called in the rule for the new line token, to compute the line and column information correctly, see [Grosch 87].

3.3.2 The Gentle specification

The *Gentle* specification for this language is contained in file *hoc0_inter.g* and is given by:

```
'MODULE' hoc0_inter
------------------------------------------------------------------------
-- Tokens
------------------------------------------------------------------------

'TOKEN' NL.
'TOKEN' PLUS.
'TOKEN' MINUS.
'TOKEN' MULT.
'TOKEN' DIV.
'TOKEN' LEFTPAR.
'TOKEN' RIGHTPAR.
'TOKEN' NUMBER (-> INT).
------------------------------------------------------------------------
-- Grammar
------------------------------------------------------------------------

'NONTERM' ROOT.
---------------
ROOT :  List .

'NONTERM' List.
---------------
List : .
List : List NL.
List : List Expr (-> Val)  NL
       put_s1 ("\\t") put_d  (Val) put_s1 ("\\n") .
------------------------------------------------------------------------
-- Expression Grammar
------------------------------------------------------------------------

'NONTERM' Expr (-> INT).
------------------------
Expr (-> I): SimpleExpr (-> I) .

'NONTERM' SimpleExpr (-> INT).
------------------------------
SimpleExpr (-> I)      : Term (-> I) .
SimpleExpr (-> I1 + I2): SimpleExpr (-> I1) PLUS Term (-> I2) .
SimpleExpr (-> I1 - I2): SimpleExpr (-> I1) MINUS Term (-> I2) .

'NONTERM' Term (-> INT).
------------------------
Term (-> I)      : Unary (-> I) .
```

```
Term (-> I1 * I2): Term (-> I1) MULT Unary (-> I2) .
Term (-> I1 / I2): Term (-> I1) DIV  Unary (-> I2) .

'NONTERM' Unary (-> INT).
-------------------------
Unary (-> I)  : Factor (-> I) .
Unary (-> 0-I): MINUS Factor (-> I) .

'NONTERM' Factor (-> INT).
-------------------------
Factor (-> I): NUMBER (-> I) .
Factor (-> I): LEFTPAR Expr (-> I) RIGHTPAR .
```

The `put_` action predicates are defined in the *IO* library and write strings and integers to the standard output device.

3.3.3 Other work

The *makefile* generator *makemake* must be parameterized as follows: `NAME=HOC0`, `DEST=<destination directory>`, the `-DUSER_DEFINE_OPAQUE` flag must be set in `CPPFLAGS`.

3.3.4 Generating and running the Hoc0 interpreter

After writing all these specifications and modifying the given programs and *UNIX* shell scripts, two steps are needed to generate the *Hoc0* interpreter: First, the *makefile* must be generated using the *makemake* command (see section 2.5). Second the *make* process must be started by the *make* command. After *make* has finished, the interpreter may started.

```
$ makemake  RETURN key
GENTLE_DIR = /usr/local/lib/gentle
$ make  RETURN key
cc -c  -I/usr/local/lib/gentle/reuse  -DUSER_DEFINE_OPAQUE
-DNR_OF_IDENT_ATTR=1 aux.c
/usr/local/lib/gentle/bin/gentle -noedit grammar.g BOOLEAN.g ERRORS.g IDENTS.g
IO.g MATH.g STATISTICS.g STRINGS.g aux.g calc.g exec.g grammar.g obj.g
Gentle: 3.9 92/08/25
Changed: g.cfg
Changed: g.grammar.c
Changed: g.tokens.h
/usr/local/lib/gentle/bin/make-scanner
Warning: in start state STD the default action may be triggered by:
lot more of information
$ g.HOC0  RETURN key
1 + 2   RETURN key
  3
Control-D key, terminates the interpreter
$
```

3.4 A floating point expression interpreter, Hoc1

The first step towards *Hoc* is the interpretation of floating point expressions. Relational, logical operators and the exponentiation operator are added. The "boolean" values are represented as floating point numbers, where FALSE is coded as 0.0 and each other value codes TRUE (usually 1.0). The language allows Ada style comments. All characters after -- up to the end of the line are treated as comment.
Because the *Gentle* language does not support floating point computations, this must be done by external predicates, which are defined in the library module *MATH*.

3.4.1 The scanner

The `ErrorAttribute` procedure must be changed in the `NUMBER` case, since the attribute type has been changed from `int` to `double` (see section 2.8). The new operators must be added, too. Only the regular expression for comments and floating point numbers will be given here:

```
/* single line comments (Ada - style) */
#STD# "--" ANY * \n  :- {yyEol (0); return g_NL;}

/* real numbers */
#STD# digit +   ,
#STD# digit + "." digit * (("E"|"e") {+\-} ? digit +) ? :
                 : {GetWord (word);
                    Attribute.attr1  = (double *)G_pool_alloc (sizeof (double));
                    sscanf (word, "%lf", Attribute.attr1);
                    return g_NUMBER;}
```

3.4.2 The Gentle specification

The grammar has changed to reflect the more complex precedence rules of the various operators. The actual computations of values is done during parsing, using the predicates defined in the *MATH* library. Some grammar rules (contained in file *grammar.g*) are:

```
NONTERM' Expr (-> REAL).
-------------------------
Expr (-> R): E0 (-> R) .

'NONTERM' E0 (-> REAL).   -- or
-----------------------
E0 (-> R): E1 (-> R) .
E0 (-> R): E0 (-> R1) OR E1 (-> R2)
           Calc2 (or, R1, R2 -> R) .

'NONTERM' E1 (-> REAL). -- and
-----------------------
E1 (-> R): E2 (-> R) .
E1 (-> R): E1 (-> R1) AND E2 (-> R2)
           Calc2 (and, R1, R2 -> R) .

'NONTERM' E2 (-> REAL). -- <, <=, >, >=, ==, !=
-----------------------
E2 (-> R): E3 (-> R) .
E2 (-> R): E2 (-> R1) RELOP (-> Op) E3 (-> R2)
           Calc2 (Op, R1, R2 -> R) .

'NONTERM' RELOP (-> OPERAND).
-----------------------------
RELOP (-> less)         : LESS.
 .....
```

The predicates implementing the calculations are contained in module *calc*. The ..._REAL predicates are defined in module *MATH*.

```
-- often used values
'VAR' REAL Zero.
'VAR' REAL One.
 ....

OPERAND = plus, minus, .....  .

'ACTION' Calc2 (OPERAND, REAL, REAL -> REAL).
-------------------------------------------
-- computes binary expressions
-- The value 0 is used for the "undefined" position.
Calc2 (plus,  Arg1, Arg2 -> Res):  Add_Real   (0, Arg1, Arg2 -> Res).
Calc2 (minus, Arg1, Arg2 -> Res):  Sub_Real   (0, Arg1, Arg2 -> Res).
Calc2 (mult,  Arg1, Arg2 -> Res):  Mult_Real  (0, Arg1, Arg2 -> Res).
Calc2 (div,   Arg1, Arg2 -> Res):  Div_Real   (0, Arg1, Arg2 -> Res).
Calc2 (power, Arg1, Arg2 -> Res):  Power_Real (0, Arg1, Arg2 -> Res).

-- (e1 AND e2) is true (1.0), iff (e1 != 0.0 and e2 != 0.0)
Calc2 (and,   Arg1, Arg2 -> One)  :
        NotEq_Real (Zero, Arg1)
        NotEq_Real (Zero, Arg2)
        .
Calc2 (and,   Arg1, Arg2 -> Zero)  :.
 .....
```

The global variables `Zero` and `One` must be initialized before they are used. The right place for such a kind of initialization is before parsing starts. This is done by introducing a nonterminal predicate `INIT`, which accepts nothing, but just calls the action predicate `init`,which does the actual initializations:

```
'NONTERM' ROOT.  -- the first nonterm
ROOT: INIT Parse (....) ....  .

'NONTERM' INIT.
INIT : init .

'ACTION' init.
init :  CvtIntToReal (0, 0 -> Z)
        Zero <- Z
        CvtIntToReal (0, 1 -> O)
        One <- O
        ....     .
```

3.5 Using variables, Hoc2

This section shows, how *Hoc* variables may be implemented. A variable is declared when a value is assigned. It is an error to use undeclared variables. A variable may be assigned more than once.
The problem, which must be solved here, is that a mapping from identifiers to values must be defined / implemented. Compiler writers call this mapping a *symbol or definition table*. The simplest (and fastest) solution is to use the possibility to attach several attributes to identifiers, which is offered by the *IDENTS* library.

3.5.1 The scanner

A regular expression for identifiers must be specified. The number of attributes a single token has does not change and is again one. The `ErrorAttribute` procedure must be extended (see section 2.8).

```
/* identifiers (Hoc / Modula - style) */
#STD# letter (letter | digit) * :
             {length = GetWord (word);
              g_enter_IDENT(word, length, &(Attribute.attr1));
              return g_IDENTIFIER;}
```

3.5.2 The Gentle specification

Objects

The new feature of *hoc2* is the notion of an "object". Each identifier (i.e. variable) has an additional attribute, specifying this object (i.e. floating point value). The library module *IDENTS* offers manipulation of such identifier attributes using the predicates `DEF_IDENT_ATTR` and `GET_IDENT_ATTR`. These predicates are not defined in *IDENTS*, because the kind (type) of the objects is defined by the user of the library. `GET_IDENT_ATTR` is a condition predicate, which succeeds only if the attribute was defined before:

```
'ACTION'    DEF_IDENT_ATTR (IDENT, INT, OBJECT).   -- external
'CONDITION' GET_IDENT_ATTR (IDENT, INT -> OBJECT). -- external
```

Three predicates are defined dealing with objects: `DefMeaning`, which defines a meaning of an identifier. The other steps of the development of *Hoc* will add more objects, and hence it is checked, that only undefined identifiers get a variable meaning and only for identifiers denoting variables the value may be redefined. `GetMeaning` succeeds only if the identifier has a defined meaning, and returns in that case the object. `GetValue` returns the value of a variable, if it is defined. If an error occurs in a *Hoc* program an error message is printed and usually the value *0.0* is returned.

```
OBJECT =
        variable (REAL)       -- each variable has a value
        .

'ACTION' DefMeaning (IDENT, OBJECT).
-------------------------------------
-- Associates an object with an identifier.
-- Only variable identifiers may get a new value.

DefMeaning (Id, variable (NewValue)):
        GetMeaning (Id -> variable (OldValue))
        -- checks, whether 'Id' is already declared as avariable, since
        -- only variable objects are allowed to be redefined.
        DEF_IDENT_ATTR (Id, 1, variable (NewValue)).
DefMeaning (Id, Obj): DEF_IDENT_ATTR (Id, 1, Obj).
        -- Now 'Id' is an undeclared identifier.

'CONDITION' GetMeaning (IDENT -> OBJECT).
-----------------------------------------
GetMeaning (Id -> Obj): GET_IDENT_ATTR (Id, 1  -> Obj) .

'ACTION' GetValue (IDENT -> REAL).
----------------------------------
GetValue (Id -> Value): GetMeaning (Id -> variable (Value)) .
GetValue (Id -> Zero): ERROR_IDENT (" '@' is not a variable", Id, 0) .
   -- Zero is a global variable, holding the value 0.0
```

These definitions are contained in module *obj*. The last clause shows the handling of errors using predicates contained in the *ERRORS* module. Because we don't include positional information the "undefined" position denoted by *0* is used.

The grammar part

New tokens and nonterminal rules for identifiers and assignments are added. The definition of a variable is done by assignment, a variable is used in an expression. Assignments have a result (as in C) and hence may occur in two contexts: first at "top level" and second inside of expressions.

```
List :
        List IDENTIFIER (-> Id) ASSIGN Expr (-> Val) NL
        DefMeaning (Id, variable (Val)) .
Expr (-> R):
        IDENTIFIER (-> Id) ASSIGN Expr (-> R)
        DefMeaning (Id, variable (R)) .
E7 (-> R):
        IDENTIFIER (-> Id)
        GetValue (Id -> R) .
```

The fact that assignment occurs in two contexts causes a "read – reduce" or "shift – reduce" called conflict. The *lalr* parser generator emits the following information in the file *_Debug* and offers a default resolution of that situation, namely to shift, which is in our case the desired resolution. The dotted lines show the derivation trees which cause the conflict. For more information see [Vielsack 88].

```
State 49
g_ROOT End-of-Tokens
g_INIT g_List
.......:
:
g_List g_Expr g_NL
       :
       g_IDENTIFIER g_ASSIGN g_Expr
       :.........
                 :
reduce g_Expr -> g_IDENTIFIER g_ASSIGN g_Expr. {g_NL} ?

g_ROOT End-of-Tokens
g_INIT g_List
       g_List g_IDENTIFIER g_ASSIGN g_Expr g_NL
       :.........
                 :
read   g_List -> g_List g_IDENTIFIER g_ASSIGN g_Expr.g_NL ?

ignored                g_Expr -> g_IDENTIFIER g_ASSIGN g_Expr. {g_NL}
retained               g_List -> g_List g_IDENTIFIER g_ASSIGN g_Expr.g_NL
```

3.5.3 Other work

The *makemake* files must be adapted accordingly using the new system name (*HOC2*). Besides that, it must be specified in *makemake*, that one attribute is attached to identifiers. This is done by including `-DNR_OF_IDENT_ATTR=1` in the `CPPFLAGS` parameter.

3.6 Standard procedures, Hoc3

In *Hoc3* standard procedures (like $\sin, \cos, \ldots$) and predefined constants (like $\pi, \gamma, \ldots$) are included. Two ways exist to implement this: First, introducing new tokens representing the predefined names and handle these tokens in a special way or second, declaring the names like other identifiers and treat the predefined functions as ordinary functions and constants as variables. We choose the second way.

3.6.1 The Gentle specification

The *Gentle* specification changes mainly in three points: The grammar must be changed to reflect the structure of standard function calls. New objects must be defined, representing the predefined functions and constants. Last but not least, the evaluation of function calls must be implemented.

The grammar

The change in the grammar is very small. The grammar rule:

```
E7 (-> R):
        IDENTIFIER (-> Id) LEFTPAR Expr (-> Arg) RIGHTPAR
        EvalFunction (Id, Arg -> R)
        .
```

must be added to the nonterminal E7.

Objects

Each standard function is a new object and hence the object type looks like:

```
OBJECT =
        variable (REAL),        -- each variable has a value
        -- built-in functions
        func_sin, func_cos, func_atan, func_exp, func_log, func_log10,
        func_sqrt, func_int, func_abs .
```

A new clause for the DefMeaning predicate must be specified, which checks that a standard function is not redefined.

```
DefMeaning (Id, Obj):
        GetMeaning (Id -> Func)
        IsBuiltInFunc (Func)
        ERROR_IDENT ("'@' is already declared as built-in function.",Id, 0) .

'CONDITION' IsBuiltInFunc (OBJECT).
-----------------------------------
IsBuiltInFunc (func_sin): .
IsBuiltInFunc (func_cos): .
IsBuiltInFunc (func_atan): .
 .....
```

Predefined constants are implemented as ordinary variables. (A bug (or feature??) is that predefined constants may get a new value). The meaning of the predefined identifiers must be declared in the init predicate. The enter_IDENT action predicate from the *IDENTS* module declares the identifiers, DefMeaning declares the meaning of that identifier.

```
-- enter predefined names (constants/functions)
enter_IDENT ("sin",   3 -> Sin)   DefMeaning (Sin,   func_sin)
enter_IDENT ("cos",   3 -> Cos)   DefMeaning (Cos,   func_cos)
       ....

math_PI_value     (-> Pi_val)
math_E_value      (-> E_val)
       ....

enter_IDENT ("PI",    2 -> Pi)  DefMeaning (Pi,  variable (Pi_val))
enter_IDENT ("E",     1 -> E)   DefMeaning (E,   variable (E_val))
       ....
```

Evaluation of function calls

The function application is implemented by the `EvalFunction` predicate. It checks that the identifier is defined. The `EvalFunc` predicate checks that the object is a standard procedure. The `EvalBuiltInFunc` applies the function to the passed value.

```
'ACTION' EvalFunction (IDENT, REAL -> REAL).
-------------------------------------------
-- checks, that 'IDENT' is a function name and returns the value of 'f(x)'
EvalFunction (Id, Val -> R):
        GetMeaning (Id -> Object)
        EvalFunc (Id, Object, Val-> R) .
EvalFunction (Id, Val -> Zero):
        ERROR_IDENT ("undeclared identifier '@' ", Id, 0) .

'ACTION' EvalFunc (IDENT, OBJECT, REAL -> REAL).
-----------------------------------------------
EvalFunc (Id, Func, Arg -> R):
        IsBuiltInFunc (Func)
        EvalBuiltInFunc (Func, Arg -> R) .
EvalFunc (Id, Obj, Arg -> Zero):
        ERROR_IDENT ("'@' is not declared as function", Id, 0) .

'ACTION' EvalBuiltInFunc (OBJECT, REAL -> REAL).
-----------------------------------------------
EvalBuiltInFunc (func_sin,   Arg -> R): math_sin   (0, Arg -> R).
EvalBuiltInFunc (func_cos,   Arg -> R): math_cos   (0, Arg -> R).
    ....
```

3.6.2 Other work

The *makemake* files must be adapted accordingly, using the new name of the system (*HOC3*).

3.7 Construct an intermediate language, Hoc4

The language of *hoc4* is not changed in comparision to *hoc3* but the implementation is changed significantly. The main point is that the parser constructs an intermediate representation for expressions, instead of evaluating them directly. After an expression is parsed, the expression tree is traversed and evaluated. The previous evaluation routines are reused.

3.7.1 The Gentle specification

First the representation of expressions is defined using terms:

```
TREE =
        const       (REAL),
        ident       (IDENT),
        unary       (OPERAND, TREE),
        binary      (OPERAND, TREE, TREE),
        assign      (IDENT,   TREE),
        func_call   (IDENT,   TREE) .
```

The evaluation predicate for expression trees has a `TREE` as input and returns the result as a `REAL`. The idea is first to evaluate the sub–expressions of a tree node (expression) and then apply the operation on the results. It uses the predicates, defined in the earlier versions of *Hoc* and looks like:

```
'ACTION' Evaluate (TREE -> REAL).
--------------------------------
Evaluate (const (R)  -> R) : .
Evaluate (ident (Id) -> R):
```

```
        GetValue (Id -> R) .
Evaluate (unary (Op, T) -> R):
        Evaluate (T -> Arg)
        Calc1 (Op, Arg -> R) .
Evaluate (binary (Op, T1, T2) -> R):
        Evaluate (T1 -> Left) Evaluate (T2 -> Right)
        Calc2 (Op, Left, Right -> R) .
Evaluate (func_call (Id, Args) -> R):
        EvalFunction (Id, Args -> R) .
Evaluate (assign (Id, T) -> Arg):
        Evaluate (T -> Arg)
        DefMeaning (Id, variable (Arg)) .
```

The `EvalFunction` predicate must also be modified, because now the function argument is a `TREE` instead of a `REAL` value.

```
'ACTION' EvalFunction (IDENT, TREE -> REAL).
-----------------------------------------
EvalFunction (Id, Arg -> R):
        GetMeaning (Id -> Object)
        Evaluate (Arg -> Val)
        -- execute function code
        EvalFunc (Id, Object, Val-> R) .
```

The `EvalFunc` and `EvalBuiltInFunc` predicates are not changed, the function argument is evaluated before they are called.

The grammar

The grammar specification must be modified by changing the result parameter of the nonterminal rules. The `List` nonterminal is extended by the action, which calls the evaluation of the parsed expression.

```
List :
        List Expr (-> T)  NL
        print_Expression (T)
        Evaluate (T -> Val)
        put_s1 ("\\t")
        put_r  (Val)
        put_s1 ("\\n") .
```

Only some grammar rules are shown as representative for the modifications needed:

```
'NONTERM' Expr (-> TREE). -- assign
-------------------------
Expr (-> T): E0 (-> T) .
Expr (-> assign (Id, T)): IDENTIFIER (-> Id) ASSIGN Expr (-> T) .

'NONTERM' E0 (-> TREE).   -- or
----------------------
E0 (-> T): E1 (-> T) .
E0 (-> binary (or, T1, T2)): E0 (-> T1) OR E1 (-> T2) .

'NONTERM' E1 (-> TREE). -- and
----------------------
E1 (-> T): E2 (-> T) .
E1 (-> binary (and, T1, T2)): E1 (-> T1) AND E2 (-> T2) .

'NONTERM' E7 (-> TREE).  -- numbers, var-access, func-call
----------------------
E7 (-> const (R)): NUMBER (-> R) .
E7 (-> ident (Id)): IDENTIFIER (-> Id) .
E7 (-> T): LEFTPAR Expr (-> T) RIGHTPAR .
```

```
E7 (-> func_call (Id, Arg)):
          IDENTIFIER (-> Id) LEFTPAR Expr (-> Arg) RIGHTPAR .
```

3.7.2 Other work

The *makemake* files must be adapted accordingly, using the new name of the system (*HOC4*).
For testing purposes it is sometimes useful to print terms. One way doing this is to print it always, or to print it only if a command line parameter is read, when the program is started. The second possibility is taken, which also shows, how the *MAIN* program must be changed to to this. The actual printing of term is performed by the generated predicates `print_`*Type*, where *Type* is a type name of the term.
A new external predicate is introduced to check whether the command line option *-print* was given. This predicate is implemented in file *aux.c*, and hence *aux.o* must be assigned to the *OBJS* parameter of *makemake*. Calling the *hoc4* program with the *-print* command line option prints the expression tree onto the standard output device. A variable `print_option` holding the values 1 or 0 if the option was given or not is used. In the *MAIN.c* file, the loop scanning the command line looks like:

```
while (i < argc) {
   if (strcmp (argv[i], "-print") == 0)  {  /* new */
            print_option = 1;               /* new */
   } else                                   /* new */
           {g_IO_DECLARE(argv[i]); inputs ++;}
   i++;
}
```

The files *aux.g* and *aux.c* look like:

```
MODULE' aux
'CONDITION' is_print_Option_set.
            -- succeeds, iff hoc was called with '-print' option.
```

The implementation of this external predicate is contained in file *aux.c* and looks like:

```
#include "SYS.h"
#include <stdio.h>
long print_option = 0; /* boolean flag, used to store whether the '-print' option is set */

/* implementation of 'CONDITION' is_print_Option_set' */
BOOL g_is_print_Option_set ()
{ return (print_option == 1) ? TRUE : FALSE; }
```

The predicate `print_Expression` uses the (by *Gentle*) generated term printing predicate `print_TREE` and is defined as:

```
'ACTION' print_Expression (TREE).
---------------------------------
-- prints the expression on 'stdout', if hoc is called with "-print" option
print_Expression (T):
        is_print_Option_set
        print_TREE (T)
        .
print_Expression (T) : .

'ACTION' print_TREE (TREE).
---------------------------
-- generated output routine.
```

3.8 Control flow, Hoc5a

This section adds *if, while* statements and statement sequences to *hoc4*. Again the statements are represented as terms, forming a list of statements. This list is then interpreted. The list definition looks like[6]:

[6]The specification of the interpretation of statement is contained in file *exec.g*

```
STMT =
        stmts      (STMTS),
        assign     (IDENT, TREE),
        if         (TREE, STMT, STMT),
        while      (TREE, STMT),
        print      (TREE),
        prints     (STRING),
        null.  -- for empty else part of an if-stmt

-- a list of statements is formed as STMTS
STMTS =
        s (STMT, STMTS),
        nil .
```

3.8.1 The scanner and parser

The scanner must be extended for the new keywords. The parser must get new rules for accepting the statements. Our implementation raises a read–reduce (dangling else) and a reduce–reduce conflict. They are solved automatically by the parser generator *lalr* in the default way, i.e. for the read–reduce conflict, the production reading the token is selected and for the reduce–reduce conflict the textual first grammar rule is selected (see [Vielsack 88]).
A critical point is the list of statements. If one constructs the term, the list is constructed in reverse order. This could be avoided using right recursion over the statement list, but the drawback of right recursion is that the interactive mode of parsing the input is no more possible (because for right recursion, the entire text must be present). The method used here is to build it in the "right" order by appending a new statement at the end of the list.

```
'ACTION' Append_STMTS (STMTS, STMT -> STMTS).
---------------------------------------------
-- appends a statement to a statement list
Append_STMTS (nil, S -> s(S, nil)) : .
Append_STMTS (s(S1, L1), S2 -> s(S1, L2)):
        Append_STMTS (L1, S2 -> L2)
        .

'NONTERM' StmtList (-> STMTS).
------------------------------
StmtList (-> nil) : .
StmtList (-> S):
        StmtList (-> S) NL
        .
StmtList (-> L1):
        StmtList (-> L) Stmt (-> S) NL
        Append_STMTS (L, S -> L1)
        .

Stmt (-> while (T, S)):
        WHILE Condition (-> T) Stmt (-> S)
        .
```

The "top level" rule initiating the entire process is specified now as:

```
List :
        List Stmt (-> S)  NL
        print_Statements (S)
        Execute (S)
        .
```

3.8.2 Interpretation of statements

The interpretation of the assignment statement and statement list are quite obivous. For the *if* statement the conditional expression is evaluated and then the corresponding statement is executed. This is done by two clauses for the *if* statement, the first evaluates the conditional expression and tests for *TRUE*. If the condition predicate `NotEq_Real` succeeds, the *then* part statements are executed. If this condition predicate fails, the next clause is tried, which executes the *else* statement. The conditional expression must not be evaluated again. The *while* statement is transformed to:

```
if Condition then {Body ; { while ( Condition ) Body}}
```

and then interpreted. If the conditional expression evaluates to *FALSE* (i.e. 0.0) the interpretation of the loop is finished. Notice: this kind of interpretation creates a new statement list for each iteration of the loop. The entire `Execute` predicate:

```
'ACTION' Execute (STMT).
------------------------
Execute (stmts (L)):
        ExecList (L)
        .
Execute (assign (Id, T)):
        Evaluate (T -> Arg)
        DefMeaning (Id, variable (Arg))
        .
Execute (if (Cond, ThenStmt, ElseStmt)):
        Evaluate (Cond -> CondVal)
        NotEq_Real (Zero, CondVal)
        Execute (ThenStmt)
        .
Execute (if (Cond, ThenStmt, ElseStmt)):
        Execute (ElseStmt)
        .
Execute (while (Cond, Body)):
        Evaluate (Cond -> CondVal)
        NotEq_Real (Zero, CondVal)
        Execute (stmts (s(Body, s(while (Cond, Body), nil))))
        .
Execute (while (Cond, Body)) : . -- if Cond evaluates to false
Execute (print (T)):
        Evaluate (T -> R)
        put_r (R)
        .
Execute (prints (Str)):
        put_s1 (Str)
        .
Execute (null) : .

'ACTION' ExecList (STMTS).
--------------------------
-- Executes a list of statements.
ExecList (nil) : .
ExecList (s(S,L)):
        Execute (S)
        ExecList (L)
        .
```

The `print` and `prints` statements are inserted to have a uniform handling of printing values and strings. These two features are fully implemented in *hoc6*.

3.8.3 Other work

The *makemake* files must be adapted accordingly, using the new name of the system (*HOC5a*).

For printing statements new routines are written, because the generated `print_STMT` predicate would print them as trees, which is awful to read. The implemented routines start each statement on a new line, without indentation (see file *exec.g*).

3.9 Loops using cyclic graphs, Hoc5

Another way implementing loops is to use jumps to some point of a statement list. The *while* loop may be translated to:

```
label (lab); if Condition then { Body ; goto (lab) }
```

Where `label (lab)` marks the start of the loop, `goto (lab)` performs the jump, `lab` is a unique name of the label. Using terms only, it is impossible to implement this kind of loop interpretation, because the `goto (lab)` statement refers to "another place" in the same term, i.e. needs a cyclic graph.
A method to simulate such cyclic graphs with terms is presented now. The idea is to use the global table feature of *Gentle*. The entries of the table `Continue` are statement lists, representing the body of a loop. The `goto` statement uses a table key of type `LABEL` for accessing the entries in the table. Whenever a `goto (lab)` statement is interpreted, the statement list is taken from the `Continue` table using the `lab` as key for it. This statement list is then interpreted.
The following program shows the needed changes: In the `STMT` term the `while` must be replaced, the grammar must be changed in the "while" rule. An extra predicate `WhileCode` is needed, because global variables and tables are not allowed in nonterminal clauses. When translating a loop, a `goto` statement is appended to the instructions forming the initial loop body. This instruction sequence is then stored in the `Continuate` table. When interpreting the `goto` instruction, the instructions stored in the table are retrieved and interpreted. The `label` is used only for documentation purposes, when printing the statement list.

```
STMT =
------
        stmts      (STMTS),
        assign     (IDENT, TREE),
        if         (TREE, STMT, STMT),
        label      (LABEL),
        goto       (LABEL),
        print      (TREE),
        prints     (STRING),
        proc_call  (IDENT, TREES),
        return,
        return_val (TREE),
        null -- for empty else part of an if-stmt
        .

Stmt (-> S1):
        WHILE Condition (-> T) Stmt (-> S)
        WhileCode (T, S -> S1)
        .

'TABLE' STMT Continue [LABEL].

'ACTION' WhileCode (TREE, STMT -> STMT).
----------------------------------------
WhileCode (Cond, Body -> S):
        'KEY' LABEL Lab
        Append_STMTS (s(Body, nil), goto (Lab) -> NewBody)
        Append_STMTS (s(label(Lab),nil), if(Cond, stmts(NewBody),null) -> Loop)
        Continue [Lab] <- stmts (Loop)
        Continue [Lab] -> S
        .

Execute (label (Lab)) : .
```

```
Execute (goto (Lab)):
        Continue [Lab] -> Next
        Execute (Next)
        .
```

3.9.1 Other work

The *makemake* files must be adapted accordingly, using the new name of the system (*HOC5*).

3.10 Procedures and functions, Hoc6

Now the last step is reached on the way to *Hoc*. Function and procedure declarations, function and procedure call, parameter passing, reading values from the keyboard and printing them on the terminal are added to the language.
Functions and procedures are new objects, having their body as additional information, `OBJECT` is extended by

```
-- function or procedure
function  (STMT)
procedure (STMT)
```

The `DefMeaning` and`GetMeaning` are modified reflecting this new objects. New statements are introduced:

```
proc_call  (IDENT, TREES)
return                          -- return from procedure
return_val (TREE)               -- returning a value from function call
```

Parameters are represented as an expression list:

```
TREES = t (TREE, TREES), nil .
```

The expression term type `EXPR` is extended to represent function calls and formal parameters. The scanner is modified to accept formal parameters, the attribute attached to them is their number coded as an integer.

```
func_call (IDENT, TREES)
formalparam (INT)
```

3.10.1 The parser

The grammar rules for function and procedure declarations are:

```
'NONTERM' Def.
--------------
Def :
        FUNC IDENTIFIER (-> Id) LEFTPAR RIGHTPAR Stmt (-> Body)
        DefMeaning (Id, function (stmts (s(Body, s(return, nil)))))
        .
Def :
        PROC IDENTIFIER (-> Id) LEFTPAR RIGHTPAR Stmt (-> Body)
        DefMeaning (Id, procedure (stmts (s(Body, s(return, nil)))))
        .
```

The `return` is appended, for the case that the user forgets it in functions, and because in procedures no explicit return is needed.
The grammar rules for function and procedure call are obvious.

3.10.2 Function and procedure call

For the interpretation of a procedure or function call the following things must be done:

- Get the statements implementing the procedure / function (using predicate `GetMeaning`).
- Evaluate the actual parameters (using predicate `Evaluate List`).
- Since nested function calls are possible, the parameters must be passed on a parameter stack (using global variable `ParamStack`). The parameter stack is implemented as a list of values, the value are accessed using their positional number in the list. The parameters for standard functions are passed with the same method.
- Interprete the body of the function / procedure (using the predicates `EvalFunction` and `ExecProc`). The formal parameters get their values from the parameter stack. It is checked, that there are enough actual parameters passed. A `return` or `return_val` statement stops the execution of the rest of a statement list of the procedure containing it. Dealing with that information (that a return occurred) is done by using the global variable `ReturnOccured`. A `return` or `return_val` statement sets it to `true`, after finishing a procedure call it is set to `false`. Function results are returned using the global variable `FunctionResult`.
- Remove the actual parameters from the parameter stack.
- Check that a function call has returned a value, and that a procedure call hasn't done it.

The important modifications to call a function or procedure are:

```
....
Execute (proc_call (Id, Args)):
        ExecuteProc (Id, Args)
        .
Execute (return) :
        ReturnOccured <- true
        .
Execute (return_val (T)):
        Evaluate (T -> R)
        FunctionResult <- R
        FunctionResult_IsReturned <- true
        ReturnOccured <- true
        .
....
Evaluate (func_call (Id, Args) -> R):
        EvalFunction (Id, Args -> R)
        .
Evaluate (formalparam (Nr) -> R):       -- evaluation of expressions
        GetParam (ParamStack, Nr -> R)
        .
```

The interpretation of function / procedure bodies:

```
'ACTION' ExecList (STMTS).
--------------------------
-- Executes a list of statements. If a return / return_val statement is
-- executed, the rest of the statement list is skipped.
ExecList (nil) : .
ExecList (s(S,L)):
        Execute (S)
        Is_NoReturnOccured
        ExecList (L)
        .
ExecList (s(S,L)) : .
....
'ACTION' ExecuteProc (IDENT, TREES).
-----------------------------------
```

```
-- Checks, that 'IDENT' is a procedure or function
-- and executes the corresponding statements
ExecuteProc (Id, Args):
        GetMeaning (Id -> procedure (Body))
        EvaluateList (Args -> Values)
          -- push passed arguments onto the parameter stack
        ParamStack -> Ps
        ParamStack <- ps(Values, Ps)
          -- execute procedure code
        FunctionResult_IsReturned <- false
        Execute (Body)
        CheckNoResultIsReturned (Id)
        ReturnOccured <- false
          -- pop parameter stack
        ParamStack <- Ps
        .
ExecuteProc (Id, Args):
        GetMeaning (Id -> function (Body))
        EvalFunction (Id, Args -> R)
        put_s1 ("\\t")
        put_r (R)
        put_s1 ("\\n")
        put_bf
        .
ExecuteProc (Id, Args):
        ERROR_IDENT ("'@' is not declared as procedure / function", Id, 0)
        .
....
'ACTION' EvalFunction (IDENT, TREES -> REAL).
-------------------------------------------
-- checks, that 'IDENT' is a function name and returns the value of 'f(x)'
EvalFunction (Id, Args -> R):
        GetMeaning (Id -> Object)
        EvaluateList (Args -> Values)
          -- push passed arguments onto the parameter stack
        ParamStack -> Ps
        ParamStack <- ps(Values, Ps)
          -- execute function code
        EvalFunc (Id, Object -> R)
          -- pop parameter stack
        ParamStack <- Ps
        .
EvalFunction (Id, Args -> Zero):
        ERROR_IDENT ("undeclared identifier '@' ", Id, 0)
        .

'ACTION' EvalFunc (IDENT, OBJECT -> REAL).
-----------------------------------------
EvalFunc (Id, function (Body) -> Result):
        FunctionResult_IsReturned <- false
        Execute (Body)
        FunctionResult -> Result
        CheckResultIsReturned (Id)
        ReturnOccured <- false
        .
EvalFunc (Id, Func -> R):
        IsBuiltInFunc (Func)
        GetParam (ParamStack, 1 -> Arg)
        EvalBuiltInFunc (Func, Arg -> R)
        .
```

```
EvalFunc (Id, Obj -> Zero):
        ERROR_IDENT ("'@' is not declared as function", Id, 0)
        .
```

Handling parameters:

```
PARAMSTACK =
        ps (TREES, PARAMSTACK),
        nil .
'VAR' PARAMSTACK ParamStack.

'ACTION' GetParam (PARAMSTACK, INT -> REAL).
-------------------------------------------
-- Returns value of the actual parameter 'n' of the current called func/proc.
-- If n > number of passed parameters, an error message is emitted.
GetParam (nil, Nr -> Zero):
       ERROR ("formal parameter access not inside of a procedure or
function",0)       .
GetParam (ps (T, S), Nr -> Val):
        SearchParam (T, Nr, 1 -> Val)
        .

'ACTION' SearchParam (TREES, INT, INT -> REAL).
-----------------------------------------------
-- If Nr == CurNr, then the parameter with number 'Nr' is found and returned.
-- If Nr <  CurNr, the next element of the tree list is tried (with CurNr
--                 incremented by one.
-- If there is no more element in the tree list, an error message is emitted.

SearchParam (nil, Nr, CurNr -> Zero):
        ERROR ("too less parameters passed",0)
        .
SearchParam (t(const (Val), Trees), Nr, CurNr -> Val):
        Eq_Int (Nr, CurNr)
        .
SearchParam (t(const (X), Trees), Nr, CurNr -> Val):
        SearchParam (Trees, Nr, CurNr+1 -> Val)
        .
```

3.10.3 Other work

Since *Hoc* has the *read* statement, it must be possible to read numbers from the terminal. Do not confuse this reading with the job, the scanner does. *read* reads a number during interpretation of a *Hoc* program, not while the program is parsed.
The files *aux.g* and *aux.c* are extended by :

```
BOOL = true, false .
'CONDITION' read_real (-> REAL).
            -- reads from 'stdin' a character sequence
            -- succeeds if it is a real number
            -- fails if it is not a real number or EOF

-------------------------------------------------------------------------

/* implementation of 'CONDITION' read_real (-> REAL)' */
BOOL g_read_real (r)
double **r;
{
  char line [255];
  *r=(double*) G_pool_alloc (sizeof(double));
    switch (scanf ("%lf", *r)) {
```

```
      case EOF : **r=1.0; return FALSE;
      case 0   : **r=1.0; scanf ("%s", line); return FALSE;
      default  :  return TRUE;
    }
}
```

The *makemake* files must be adapted accordingly, using the new name of the system (*HOC6*).

Appendix A

The Gentle Manual Page Entry

NAME

g, g-all, gentle, gtags, glint - Compiler description language and compiler generation tool

SYNOPSIS

g [**-noedit** | **-nolist**] *name*

g-all [**-noedit** | **-nolist**]

gentle [**-noedit** | **-nolist**] *name* ...

gtags

glint [**-nolist**] *name* ...

DESCRIPTION

Gentle is a compiler specification language, which may used for other text manipulation purposes too. The language is based on Horn logic.

Gentle is also a tool for generating an executable program (compiler) for that specification. Output of the Gentle system is a set of C programs and input to a scanner and parser generator, which must be compiled and linked together to get the desired compiler.

Commands

Before using any of the commands below, the environment variable **GENTLE_DIR** must be set to the directory containing the Gentle system, which is usually

GENTLE_DIR = /usr/local/lib/gentle

The commands are contained in *$GENTLE_DIR/bin*, they denote:

g *name*	analyses the specification contained in file *name*.**g**. All other files *.**g** in the current directory are also visited, but only for the *Gentle* specification *name* output is generated.
g-all	analyses all Gentle specifications (i.e. all files *.**g** of the current directory), and produces output for them.
gentle *name* ...	analyses all given specifications. Only for the first specification output is generated.
gtags	Supports the *tags* feature of the *vi* editor. All global visible Gentle identifiers, defined in the *Gentle* specifications contained in files of the current directory are inserted into the *tags* file. When editing a Gentle specification using the *vi* editor, pressing the *ctrl]* key searches the definition of the word, and the cursor is positioned to that point. The ' ' (two single quotes) or *ctrl* ~ (control and tilde) keys return to the original position, if the definition was contained in the same file or in another file.
glint name ...	Analyses the gentle specification contained in the specified files and prints to the standard output device an alphabeticaly sorted cross reference listing of identifers, the context free grammar (in Bachus-Naur-Form), as well as errors, and warnings.

Error handling

If an error was detected during analysis of a specification, the textual error message and the source file are shown together in the editor *vi*. The error messages are positioned below the lines containing the errors. Using the *v* key in the command modus of *vi* positions the cursor to the next line containing an error. This file may be edited, the error messages are removed after leaving the editor.

OPTIONS

-noedit If this option is present, no editor is called. If it is not present, the editor *vi* is opened with the given first *name*

-nolist Implies the option *-noedit*. If this option is given, the error handling procedure is not encountered (i.e. the error messages are not shown together with the source text). The error messages (if any) are written to files **ERRORS-***name***-ERRORS**, where *name* is the filename of the *Gentle* module containing the errors.

FILES

name.g	Gentle specification.
SCANNER.rex	Specification of the scanner using *rex*.
MAIN.c	The main procedure of the generated system.
name	The generated system. *name* must be specified in the *makemake* makefile generator.
makemake	Is the Gentle makefile generator.
makefile	The generated makefile.
tags	Produced by *gtags*.
ERRORS-*name*-ERRORS	Contains error generated by the Gentle system, where *name* is the filename of the *Gentle* module containing the errors.
LISTING-*name*-LISTING	The file contining the merged source and error messages.
g.*	Files generated by the *Gentle* tool.
g.*name*.c	C source file for *Gentle* module *name*.
g.TOKENS.h	Token specification.
g.TOKEN_STRINGS.h	Used for error handling.
g.cfg	Input (context free grammer, etc.) for *yacc*.
g.PARSER.lalr	Input (context free grammer, etc.) for *lalr*.
_Debug	Debugging information generated by the parser generator.
g.SCANNER.*	The generated scanner.
g.PARSER.*	The generated parser.
g.SCANNER_TYPES.h*	Generated type definitions for the scanner.
g.PARSER_TYPES.h*	Generated type definitions for the parser.

VERSION

The current version is 3.9, August 25, 1992

SEE ALSO

rex [Grosch 87], **lalr** [Vielsack 88, Grosch 90], **vi**
[Schröer 89] [Vollmer 91a] [Vollmer 91b]
This manual.
The file *$GENTLE_DIR/documentation/CHANGES* contains a description of the changes of the system and language.

Appendix B

The Hoc Manual Page Entry

NAME

hoc - interactive floating point language

SYNOPSIS

hoc [*file* ...]

DESCRIPTION

Hoc interpretes a simple language for floating point arithmetic, at about the level of BASIC, with C-like syntax and function and procedures with arguments and recursion.

The named *file*s are read and interpreted in order. If no *file* is given or if *file* is - (dash) *Hoc* interprets the standard input.

Hoc input consists of *expressions* and *statements*. Expressions are evaluated and their results are reported. Statements, typically assignments and function or procedure definitions, produce no output unless they explicitly call *print*.

SEE ALSO

Hoc – An Interactive Language for Floating Point Arithmetic by Brian Kernighan and Rob Pike.

bas (1), **bc (1)** and *dc (1)*.

BUGS

The treatment of newlines is not exactly user-friendly.
In interactive mode, two newline characters must be given to let the system report a result of an expression.

Appendix C

Syntax Summary

```
LargeIdent   ::=  ( "A" | ...  | "Z" ) (letter | digit)* .
SmallIdent   ::=  ( "a" | ...  | "z" ) (letter | digit)* .
Identifier   ::=  LargeIdent | SmallIdent .
letter       ::=  "A" | ...  | "Z" | "a" ...  "z" | "_" .
digit        ::=  "0" | ...  | "9" .
    -- starts a single line comment and
    /* starts a /* possibly nested */ comment, which may range
    over several lines */

Gentle_Spec   ::=  'MODULE' Identifier ModuleBody .
ModuleBody    ::=  (Declaration | Signature | Clause )* .

Declaration   ::=  TermTypeDecl | OpaqueTypeDecl | GlobalVarDecl | GlobalTableDecl .

TermTypeDecl   ::=  Type "=" FunctorList "." .
Type           ::=  LargeIdent.
FunctorList    ::=  ( Functor | Functor "(" Arguments ")" ) // "," .
Functor        ::=  SmallIdent.
Arguments      ::=  Argument // "," .
Argument       ::=  [LargeIdent ":" ] Type .

OpaqueTypeDecl   ::=  'TYPE' Type "." .

IntConst      ::=  digit + .
StringConst   ::=  """ Char * """ .
Char          ::=  <any (escaped) character, except " and line break> .

Signature        ::=  'TOKEN' Identifier [ OutArguments ] "." |
                      'NONTERM' Identifier [ OutArguments ] "." |
                      'ACTION' Identifier [ InOutArguments ] "." |
                      'CONDITION' Identifier [ InOutArguments ] "." .
OutArguments     ::=  "(" "->" Arguments ")" .
InOutArguments   ::=  "(" [ Arguments ] "->" Arguments ")" .

Clause        ::=  Head ":" Tail "." .
Head          ::=  HeadLiteral .
Tail          ::=  TailLiteral * .
HeadLiteral   ::=   Identifier [ "(" FormalParameters ")" ] .

TailLiterals   ::=  TailLiteral * .
TailLiteral    ::=  Identifier [ "(" ActualParameters ")" ] |
                    GlobalVarRead | GlobalVarWrite |
                    GlobalTableNewEntry |
                    GlobalTableRead | GlobalTableWrite .

LocalVariable   ::=  LargeIdent.
```

```
FormalParameters   ::=  Parameters .
ActualParameters   ::=  Parameters .
Parameters         ::=  [ InParameters ] [ "->" OutParameters ] .
InParameters       ::=  Parameter // "," .
OutParameters      ::=  Parameter // "," .
Parameter          ::=  Term |
                        LocalVariable | GlobalVariable |
                        StringConst | IntConst |
                        Expression Operator Expression .
Term               ::=  Functor [ "(" ArgumentList ")" ] .
ArgumentList       ::=  Parameter // "," .
Expression         ::=  LocalVariable | GlobalVariable |
                        IntConst |
                        (Expression Operator Expression) .
Operator           ::=  "+" | "-" | "*" | "/" .

GlobalVarDecls    ::=  'VAR' Type GlobalVariable "."
GlobalVariables   ::=  LargeIdent.

GlobalVarRead    ::=  GlobalVariable "->" Parameter .
GlobalVarWrite   ::=  GlobalVariable "<-" Parameter .

GlobalTableDecl   ::=  'TABLE' ((Type GlobalTable) // "," ) "[" KeyType "]".
GlobalTable       ::=  LargeIdent.
KeyType           ::=  LargeIdent.

GlobalTableRead    ::=  GlobalTable "[" KeyVariable "]" "->" Parameter .
GlobalTableWrite   ::=  GlobalTable "[" KeyVariable "]" "<-" Parameter .
KeyVariable        ::=  LocalVariable .

GlobalTableNewEntry   ::=  'KEY' KeyType KeyVariable .
```

Bibliography

[Clocksin *et al* 84] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer Verlag, Heidelberg, New York, 2 edition, 1984.

[Fisker *et al* 75] R.G. Fisker, C.H.A. Koster, C.H. Lindesy, B.J. Mailloux, L.G.L.T. Meertens, J.E.L. Peck, Sintzhoff M., and Wijngaarden A. Revised report on the algorithmic language ALGOL 68. *Acta Informatica*, pages 1–236, 1975.

[Grosch 87] Josef Grosch. Rex – a scanner generator. Technical report, GMD Forschungsstelle an der Universität Karlsruhe, 1987.

[Grosch 90] Josef Grosch. Lalr – a generator for efficient parsers. *Software–Practice and Experience*, 20(11):1115–1135, November 1990.

[Kernighan *et al* 77] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, Inc., 1977.

[Kernighan *et al* 78] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment.* Software Series. Prentice-Hall, Inc., 1978.

[Koster 71] C.H.A Koster. Affix grammars. In J.E.L Peck, editor, *ALGOL 68 Implementation*, pages 95–109. North Holland, Amsterdam, NL, 1971.

[Lloyd 87] W. Lloyd, J. *Foundations of Logic Programming.* Springer Verlag, Heidelberg, New York, second, extended edition, 1987.

[Schröer 89] F.W. Schröer. Gentle. In J. Grosch, F.W. Schröer, and W.M. Waite, editors, *Three Compiler Specifications*, GMD – Studien Nr. 166, pages 31–36. GMD Forschungsstelle an der Universität Karlsruhe, August 1989.

[Vielsack 88] Bertram Vielsack. The parser generators lalr and ell. Technical report, GMD Forschungsstelle an der Universität Karlsruhe, 1988.

[Vollmer 91a] Jürgen Vollmer. The compiler construction system GENTLE – manual and tutorial. GMD – Bericht 508, GMD Forschungsstelle an der Universität Karlsruhe, February 1991. Note: GENTLE was defined by F.W. Schröer in: *Three Compiler Specifications*, GMD – Studien Nr. 166, 1989.

[Vollmer 91b] Jürgen Vollmer. Experiences with Gentle: Efficient compiler construction based on logic programming. In J. Maluszynski and M. Wirsing, editors, *Proceedings of the Third International Symposium on Programming Language Implementation and Logic Programming – PLILP 1991*, volume 528 of *Lecture Notes in Computer Science*, pages 425–426. Springer Verlag, Heidelberg, New York, August 1991. Note: GENTLE was defined by F.W. Schröer in: *Three Compiler Specifications*, GMD – Studien Nr. 166, 1989.

[Waite *et al* 84] William M. Waite and Gerhard Goos. *Compiler Construction.* Springer Verlag, Heidelberg, New York, 1984.

[Watt 74] D.A. Watt. *Analysis Oriented Two Level Grammers.* PhD thesis, Glasgow, 1974.