A flexible front-end for a pascal compiler

Willem Jan Withagen
Eindhoven University of Technology
February 24, 1993, Version: 1.1
Email: `wjw@eb.ele.tue.nl`

February 24, 1993

Contents

Chapter 1

Introduction

As part of a larger research program to investigate the performance analyses and performance prediction of processor architectures, there is a need for a (set of) compiler(s) which have a very open design.

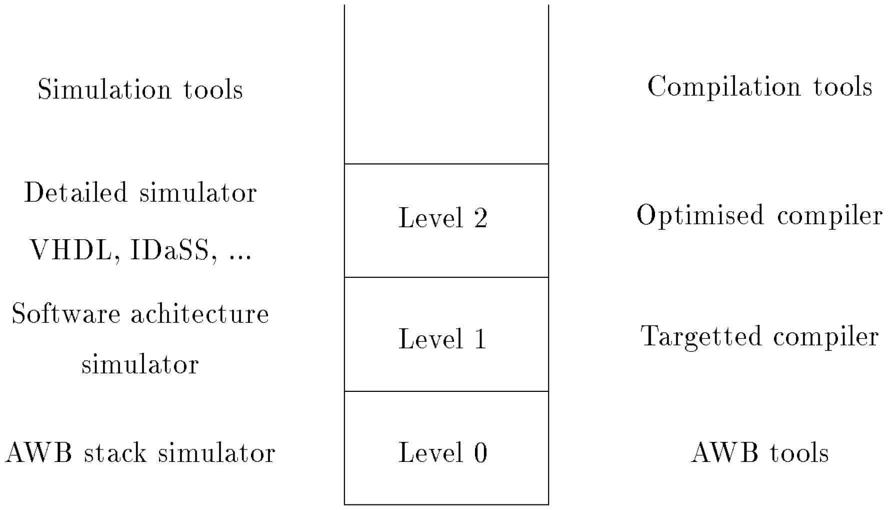

Figure 1.1: Visualisation of the design hierarchy

In this research program, processors are modeled at three levels of architecture. (See figure 1) The compiler requirements become more complex with increasing detail of description. This calls for a layered approach where modules at different layer of the compiler architecture can easily be replaced by different modules, without affecting the functioning of the other components. The design as displayed in figure 1.2 shows the traditional elements of a compiler:

- Front-end (Scanner/Parser and Semantic Evaluation).

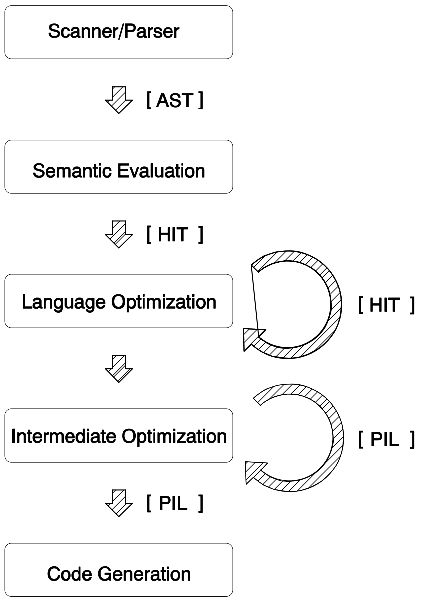

Figure 1.2: General architecture of a compiler

The source text is read, and transformed into an internal representation. This representation is checked for semantic correctness.

- Optimizer (general and Language dependant)

  Optimization is separated into two parts. The first part transforms the Abstract Syntax Tree (AST) into another, called HIT[1], removing all superfluous information which was only required for semantical analysis. Since this is mostly language dependant, it can be part of the front-end.

  Also on this level are those optimisations which are language dependant. The most obvious example here would be pointer optimsation for Pascal. This is totally different from pointer optimisationin `"C"`, since Pascaldoes not allow pointer aliassing other than by other pointers.

  Then this tree is transformed into a universal format which is shared by several compiler front-ends. The generic optimization are performed on the Intermediate description (PIL), possibly influenced by the type of the source language and the architecture to compile for.

  The reason for this two level seperation is the mere fact that certain optimisation are easier performed on trees, where others are beter suited for implementation on a lower level. THe HIT still keeps the structures which are part of the source language, where as in the PIL description most of this type of information is lost.

- Code generation or back-end.

---

[1] Higher Intermediate Tree, as opposed to PIL. (Primitive Intermediate Language)

> PIL is the format used by the code generator to derive the actual code from. The back-end can also include optimizing phases, most common is a peephole optimizer. But other architecture dependant optimizations should occur here. They are in the back-end, since they are different for each and every new architecture.

This design allows the construction of a compiler which consists of a set of modules transform trees within their own tree language (optimizers), connected by modules which transform from one tree type to another tree type.

The first working version of the compilers will not include the higher intermediate tree (HIT) description, but the (modified) abstract syntax tree (AST) is directly used for code generation. And thus is a first transform from AST to PIL made, which is then translated into actual code.

1.1 Pascal

In this technical report the design and internals of the Pascal front-end are described. It is not intended as another manual on the usage of the language.

As reference for the design of the compiler two reference are used: the standard[1] and a reference work by the primary designers of Pascal[11].

The standard is used as primary source of definitions and descriptions. References to this work are not in the usual citations format, but they also include a section number where certain information can be found. (eg. Standard[1.1]) The Pascal User Manual[11] is used to get a clearer interpretation of the text in the standard.

Over the years Pascal has developed a whole series of cousins (dialects) which are extensions to the original language. The differences are mainly due to the poor I/O-capabilities the standard language has. It is the intention to support several of the more common extensions to the language. In this report, the extensions are described in the appendices. The detailed description of the implementation of the extensions is usually in the module description in the source.

1.2 GMD COCKTAIL compiler tools

The GMD COCKTAIL compiler tools[3] are used as the backbone for the generation of the compiler parts. For a full description of the tools, readers are referred to a large number of references. [2, 5, 7, 6, 10, 4, 8, 9, **?**]

Also was the generation of the Pascal front-end used as a vehicle to gain experience in the usage of the versatile but complex set of tools.

One of the strong points of COCKTAIL is that there are tools available for all layers of a compiler. This gives the programmer the possibility to express his actions in a more natural way. And that the interfacing between the layers can be specified with the aid of tree grammars. Thus helping in reducing the amount and complexity of the code written in the target language (`"C"`).

1.3 Notational conventions

All the grammars are written, using the tree grammars used in the COCKTAIL-tools. Check [7] where the grammar for the tree grammars is specified.

But here is the short version (only the rule part):

```
Rules           = <
    NoRule      = .
    Nonterminal = Rules Name '='  AttrDecls Extensions '.' .
    Terminal    = Rules Name ':'  AttrDecls Extensions '.' .
>.
Extensions      = <
                = .
                = '<' Rules '>'
>.
AttrDecls       = <
                = .
    ChildSelct    = AttrDecls     Name ':' Name .
    ChildNoSelct  = AttrDecls          ':' Name .
    AttrTyped     = AttrDecls '[' Name ':' Name ']'.
    AttrInteger   = AttrDecls '[' Name          ']'.
    Arrow         = AttrDecls '->'
>.
```

An example of a simple rule is:

```
    If = 'if' Expr: Expression 'then' Then:Stat 'else' Else:Stat .
```

Here `Expr:`, `Then:` and `Else:` function as selectors for the nonterminals `Expression` and `Stat`. The elements in `''` are considered terminals if they do not appear on the left size of any rule.

A more complex example is:

```
Expr            = Next: Expr [IsConst :boolean] <
    Bin         = Lop: Expr Rop: Expr [Operator :int] .
    Unary       = Expr [Operator] .
    RealConst   = [Value :float] -> [InRange :boolean].
>.
```

Which shows the base rule `Expr` and several extensions. (`Bin`, `Unary`, `RealConst`) Note that each of the extensions is a nonterminal which can be used again in any other right hand side. With the rules are several attributes: All rules have an attribute `IsConst` and `Next` because the attributes and children of the base node are distributed to the extensions. The `Unary` rule shows the use of the default type `int`. The `-> [InRange]` for `RealConst` indicates that the value for this attribute is only calculated after the node for this rule has been created.

1.4 Source

The sources of the Pascal frontend are publicly available to people connected to the internet. Using anonymous `FTP`, the sources can be found at:

`ftp.eb.ele.tue.nl` in `/pub/src/pascal/frontend.zoo`

Chapter 2

Scanner

The larger part of the scanner definition, the reserved words, are extracted automaticaly from the definition of the concrete grammar. Most of the remainder of the scanner follows a standard approach in the implementation. The additional code in the scanner is to process:

- numbers (integer and floating point)
- strings
- comments
- options in comments
- identifiers

The scanner also recognizes the lexical alternatives as indicated in Standard[6.1.9].

2.1 Identifiers

Identifiers (Standard[6.1.3]) consist of a string of characters of any length containing letters and digits, where the first character has to be a letter.

The extension to the identifiers allows _ and $ to be part of the identifiers. But when they are used as first character, a second one has to follow, which has to be a letter. (Eg. `_a` and `$a` are valid identifiers, where as `_`, `$`, `_1` or `$1` are not.)

Note: In near future it could be that hexadecimal constants are introduced as integer constants. When using the TurboPascal definition for this, it could generate a "conflict" between identifiers and constants. (Eg. `$abs` is an identifier, where `$abc` would be an integer constant.) The scanner will be able to resolve this, but the text will not become any clearer when the programmer uses this.

Al list of reserved words is included in the apendices.(see D.1

2.2 Comments

The Standard[6.1.9] matches a comment opening { with a close `*)`, and vice versa. Since it is a very common case to use the {} to out-comment parts of the program which use `(**)`

as regular comments, this policy is adopted in the scanner. However, comments cannot be nested, other than `{(**)}` or `(*{}*)`.

2.2.1 Options

Although the contents of comments are not supposed to influence the program, they are used to pass certain options to the internals of the compiler. Very often they (re)set switches to modify some compilation properties.

2.3 Errors

Few errors can be detected in the scanner. Most common are the invalid characters. Each occurrence is reported separately. One possible cause for invalid characters lies in the transportation of sources for systems which use different line terminators. For one explicit case (MS-DOS), the extra invalid characters are accepted and transformed into spaces. Line-feeds are also accepted.

One other type of error which can be caught is the open ended string. When a string contains a `NL`-character then the string is considered unterminated.

2.4 Ignored characters

To be able to use this frontend in several environments, as few constraints as possible are enforced. Thus the following characters are ignored: `<LF>,<FF>,<VT>, <SUB>`[1]

---

[1] The values of these codes follow the ASCII character set.

Chapter 3

Parser and Abstract Tree generation

The parser has two objectives:

1. verification that the input conforms with the context-free grammar of Pascal.
2. transform the input into an abstract grammar tree (AST).

3.1 The concrete grammar

The context-free or concrete grammar (in short CG) is extracted from the grammar as specified in the Standard[Appendix A], and as such does it accept the "Level 1" definition of the ISO-standard. However, correct processing of the *Conformant array* is only limited to the parser, it is not processed any further.

The grammar is specified in a right recursive manner to fit the needs of LALR-tools. The concrete grammar can be subdivided in several major parts: Declarations, statements and expressions. And although the tools actually allow one to specify the grammar in a very compact and concise way, chain-rules are rarely used.

It contains 4 terminal symbols which have attributes in which the values from the scanner are stored: `IDENTIFIER`, `UNSIGNED_INT`, `UNSIGNED_REAL` and `STRING`. Another attribute accepted from the scanner is `[Position]`, indicating the location of textual counterpart in the source file.

3.2 Abstract grammar

The abstract grammar (AG for short) listed in appendix D.3.2) has a clear relation to the concrete grammar. Its purpose is to give an accurate representation of the source, without the additional syntactic sugar of the concrete grammar. The purpose of the parser is to accept or refuse the input source, with regards to the context-free grammar. After this, the context-free grammar is no longer required.

The abstract grammar can again be subdivided into three major parts:

- Scoping and declarations.

- Statements.
- Expressions.

Where possible the terminal symbols are either enclosed in `''`, in capitals or both. And although the tools allow the designer to make heavy use of chaining rules, these are rarely used.

During the design of the abstract grammar a careful balance has to be struck between a short, compact and (but too) generic grammar and a grammar which has (too) many language elements for similar cases. In the compact grammar case will it require extra code to differentiate between almost equal cases, where in the second case the abstract grammar must have identical code for the elements which are similar. Two examples will try to show the problem.

The CG contains several rules to match procedure and functions declarations in all their manifestations(actual, forward or external), but they are all mapped onto only one rule in the AG which is annotated with several attributes to indicate what instance is represented.

The CG is:

```
proc_dcl_part     =  <
  procedure_dcl   =  proc_head body .
  function_dcl    =  func_head body .
>.

proc_head         =  proc_heading ';'.

proc_heading      =  'PROCEDURE' IDENTIFIER formal_params .

func_head         =  func_heading ';'.

func_heading      =  'FUNCTION' IDENTIFIER function_form .

function_form     =  <
  function_form1  =  .
  function_form2  =  formal_params ':' IDENTIFIER .
>.

body              =  <
  body1           =  block .
  body2           =  'FORWARD'.
  body3           =  'EXTERNAL'.
>.
```

And in the AG everything collapses into one rule:

```
Decls            = <
      Proc       = [ ProcType :tProcType ]
                   Formals
                   PrimType
                   Scopes .
```

```
>.
```

Here `ProcType` indicates whether the routine is a procedure or a function, and if it is a forward, external or actual definition. This gives that advantage that there is little duplicated code, for the semantic processing of routines.

Another example shows a simple construction in the CG, but for which the AG requires very different actions.

```
ident_list          =  <
  moreidents        =  ident_list ',' IDENTIFIER .
  oneident          =  IDENTIFIER .
>.
```

These rules are used in several places in the CG Since lists with names are required in many place in the CG, these rules are used in several places. It is only during semantic analyses that different identifier-lists get different meanings.

It is very cumbersome to translate `ident_list` into the equivalent in AG and then differentiate with an attribute (one like the `ProcType` in the routine equivalent) for various different parts of code, because it would create to much complex code.

Instead the `ident_list` trees are transformed into the appropriate AG trees once they are parsed. And although the AG rules look very similar, the semantic code which goes with each of the rules is very different. And thus can the semantic evaluator apply the appropriate code the the various instances of the `ident_list`.

Elements in AG which use lists are:

```
Decls               = <
      Var           = VarList Type .
      Proc          = [ ProcType :tProcType ]
                      Formals
                      PrimType
                      Scopes .
>.
Type                = <
    Enumeration     = EnumIds .
>.

Formals             = <
  NoFormal          = .
  Formal            = Formals
                      [ IsVar :Boolean ]
                      ParIds Type .
>.
ParIds              = <
  NoParId           = .
  ParId             = [ Ident :tIdent ] [ Position :tPosition ]
                      ParIds .
>.
VarList             = <
```

```
  NoVar            = .
  VarId            = [ Ident :tIdent ] [ Position :tPosition ]
                       VarList .
>.
EnumIds            = <
  NoEnumId         = .
  EnumId           = [ Ident :tIdent ] [ Position :tPosition ]
                       EnumIds .
>.
```

3.2.1 Expressions

The parser is capable of processing a grammar with a general set of expressions and a list of operators with fixed priorities . Using this would reduce the grammar with a reasonable amount of rules. This possibility is not used since it would remove the strong congruence with the grammar as specified in the standard.

3.3 Building the Abstract Tree

As already mentioned in a previous section, is the input text verified against the concrete syntax, and while parsing the input a corresponding Abstract Syntax Tree (in short AST) is build.

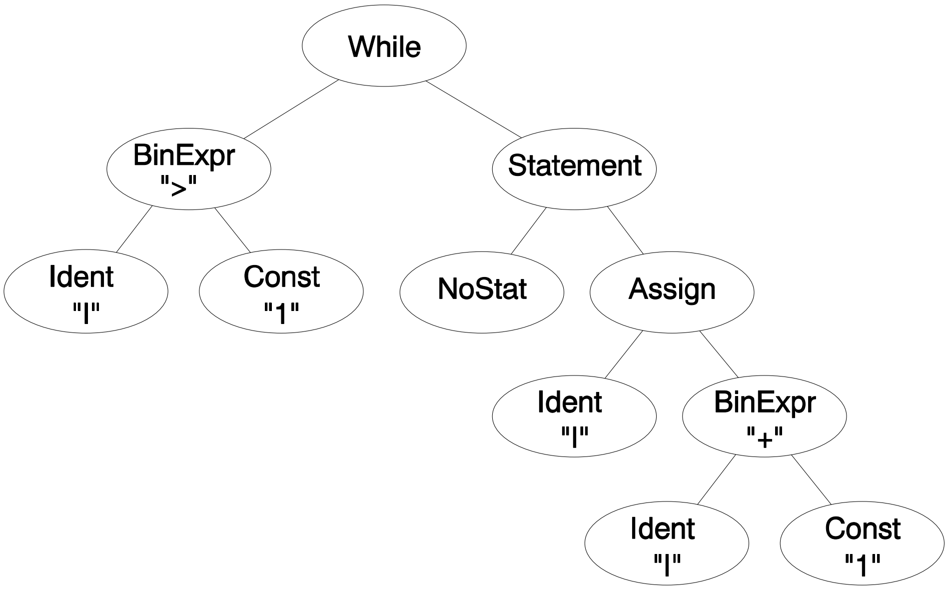

Figure 3.1: Tree construction of a code piece

The actions to be executed upon completion of a concrete syntax rule is usually the creation of a AST-node, which is used to represent part of the language construct in the AST. Except for a few rules are the actions the creation of a node which is comparable to the concrete syntax construct. The resulting node is then passed onto the parenting grammar rule which uses it to incorporate it in the node it is going to construct in it turn. For this purpose all concrete grammar rules have an attribute `[Tree :tTree]`, used to propagate the links to subtrees.

Example:

```
while I > 0
do I := I - 1
```

`NIL`-pointers Note that by specification in the abstract grammar all edges in a AST are pointing to existent items. Eg. list terminations are created by rules with empty right hand sides, and thus actual (leaf) nodes in the AST. For example, `NoVar` terminates the `VarList`:

```
VarList              = <
  NoVar              = .
  VarId              = [ Ident :tIdent ] [ Position :tPosition ]
                         VarList .
>.
```

As a consequence of this, no `NIL`-pointers can occur in the AST. This motto will be used during the remainder of the design of the program. It will allow very rigorous testing of pointers: Any `NIL`-pointer in the AST is an invalid one. A VarList would look like:

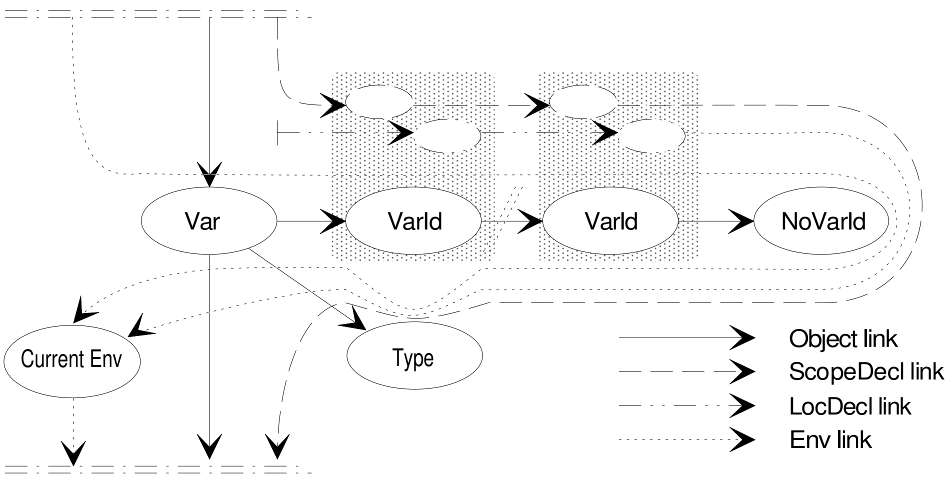

Figure 3.2: The AST for a VarList

3.3.1 Modification of `VarList`

As was described in a previous section(3.2, is the `ident_list` construction used at several places. And not at every of those places should the list be constructed using `VarList` nodes. Therefor, three routines 'translate' `VarList` lists to other types of lists. (`EnumIds`, `FielsIds`, `ParIds`)[1]

---

[1] Code for small jobs like this are easily programmed in Puma.

The transformations are only invoked once the whole `VarList` is complete. This is done at the instance where the list is passed on to parent rules.

3.3.2 Routines

Due to the fact that the rules for the routines are not written as chained rules, but are split in several parts, is it necessary to propagate the information generated in the 'sub'-rules to the rules where the actual routine node is generated.

As is shown in the following example where the attributes for the name, position, formal parameters, result type and declaration type are constructed and passed on to the parent rule (`function_dcl`). Here the information is combined and procedure node is created *one* node: (`mProc()`).

```
function_dcl  = /*   func_head body ';' . */
{
  Tree := {
     switch(body:DeclType) {
       case DeclActual\: Tree = mProc(
                   NoTree, func_head:FuncName, func_head:FuncPos, FuncActual
                 , func_head:Formals, func_head:TypeName, body:Tree);
                         break;
       case DeclForward\: Tree = mProc(
                   NoTree, func_head:FuncName, func_head:FuncPos, FuncForward
                 , func_head:Formals, func_head:TypeName, mNoScope());
                         break;
       case DeclExternal\: Tree = mProc(
                   NoTree, func_head:FuncName, func_head:FuncPos, FuncExtern
                 , func_head:Formals, func_head:TypeName, mNoScope());
                         break;
     }
  };
}.

func_head     =  /*    func_heading ';'. */
{    FuncName := func_heading:FuncName;
     FuncPos  := func_heading:FuncPos;
     Formals  := func_heading:Formals;
     TypeName := func_heading:TypeName;
}.

func_heading  = /*   'FUNCTION' IDENTIFIER function_form . */
{    FuncName := IDENTIFIER:id;
     FuncPos  := IDENTIFIER:Position;
     Formals  := function_form:Formals;
     TypeName := function_form:TypeName;
}.
```

```
function_form1 = /*   EMPTY  . */
{    Formals  := mNoFormal();
     TypeName := mNoPrimType();
}.

function_form2 = /*   formal_params ':' IDENTIFIER . */
{    Formals  := formal_params:Tree;
     TypeName := mTypeId(IDENTIFIER:id,IDENTIFIER:Position);
}.

body1          = /*   block . */
{    DeclType := DeclActual;
     Tree     := block:Tree;
}.

body2          = /*   'FORWARD'. */
{    DeclType := DeclForward;
     Tree     := mNoStat();
}.

body3          = /*   'EXTERNAL'. */
{    DeclType := DeclExternal;
     Tree     := mNoStat();
}.
```

3.3.3 Character or string constant

The scanner/parser combination determines every item in '' '' to be a string. Pascal however differentiates between character and string constants. Although this distinction should/could be made in the scanner, it is currently done at the instance where a AST-node is generated. Modification requires revision of all elements, which is too much effort at the moment.

Note that it will have little or no effect on the resulting AST. This is due to the fact that attempts are made to keep the compiler phases as independent as possible.

3.3.4 Appending to already existing lists

In most rules the creation of the tree in nothing more than the creation of a node to which a possible already created subtree is added. Then when the whole list/tree is ready, the nodes are reversed in the list to cancel the bottom-up, right-most strategy. (Using `ReverseTree`) This bottom-up ordering places the textual last element as first element in the list. And although this is not necessarily incorrect, it does not repesent the actual order of specification. It also complicates matters during the processing of declarations, where this order is significant.

On several occasions two lists have to be joined, meaning that one list is appended to the end of the other. This requires the traversal of one list until the end is found, where then the second list is appended. Since only synthesized attributes are available during the generation of the AST, there are two possibilities.

1. Create a synthesized attribute which contains the last node of a list to which the second list needs to be appended.

   This requires careful examination of the elements of the list, since the this last element is not the list terminator, but the last node required is the parent of this terminal node.

   Construction of the joined list is very simple, since the extra attribute gives the node to which the second list should be appended.

2. Additional code is added to the rule, which traverses list one to find the last used node, to which then the second list is appended. This solution leaves all rules untouched, except the rule which performs the join.

The second solution is used, since it leaves most code parts of the rules unchanged. And it is the intention to use a easily coded `Puma` routine, instead of in-lined `"C"`-code.

3.3.5 Other attributes used.

Other attributes used are:

- `[DirUp :Boolean]` to indicate the next step direction in `FOR`-statements.

- `[Operator]` to pass on the type of the operator.

  The values of the operators are defined in a global module. (`globals.h`). The actual value is of little importance at the moment, but it holds the numerical value of the operator string. This is allowed in `"C"`, as long a a standard integer can hold these characters. This limits it to a maximum of 4 characters, but allows easy printing of the operator.

  Note that if cases need to be build with operator, it is more sensible to select a simple enumerate.

- `[Position :tPosition]` used during semantic evaluation. This gives position information while reporting the error.

  The scanner only puts position information in the items it returns to the parser. Now if these need to be used in other rules, these positions need to be copied to extra attributes in these rules.

Chapter 4

Semantic evaluation

Semantic rules specify "the meaning" of any syntactically valid program written in Pascal. Although syntax has a large influence on the appearance of a language, the semantics gives meaning to the sentences of the language.

In this chapter will describe the elements which will allow the frontend to check that the semantics of a program are correct. This is done by means of attribute evaluation.

First the functioning is of attribute evaluation is discussed in more detail. Then another global structure is introduced: The symbol table. After which elements of the semantic evaluation are discussed in more detail. For the simplest cases is code include, since it gives the reader a better understanding of the attribute evaluation mechanism. More complex case are not included since it would then required a detailed explanation of the code. Instead will the text try to explain the functionality of the actions in these cases.

4.1 Attribute evaluation

For every node type an arbitrary number of attributes of arbitrary types can be declared using the ASTs notation. These attributes have properties *input*, *output*, *synthesized*, *inherited*, *threaded* and *virtual*.

Input attributes receive a value at node-creation time, whereas others receive their values at later times. *Output* attributes are supposed to hold a value at the end of the node's existence, where others may become undefined. *Synthesized* and *inherited* describe the kinds of attributes occurring in attribute grammars, and indicate the direction of data-flow through the AST.

The values of the attributes are now computed by visiting the elements of the tree in a certain order and at some points during the visits a value is calculated using the contents of already calculated attributes. The calculation of the visit sequence is again a chore done by a tool.

For every node type, attribute computations (ACs) or actions are specified. ACs are written in the desired target language, using expressions, statements, and/or calls to external functions of separately compiled abstract data types. However, ACs have to be functional in order to allow AG to calculate the derivation of dependencies among the attributes and to determine the appropriate evaluation order. And a large part of the remainder of this chapter shall be devoted to describing the used attributes.

The full description of the attribute evaluation mechanism of COCKTAILis available in [6]. In [12], a large portion of the text is dedicated to working with attributes.

4.1.1 A simple example of an attribute

The attribute `[Level :int INHERIT ]` is used to count the number of scope nestings. Every time a new scope is entered `[Level :int]` is incremented. The code for this is amazingly short:

```
MODULE Levels
/*  Keep the number of nested levels.
/*  The Levels are thus that after every procedure identifier
/*  the Lex-Level is incremented. Thus all arguments are placed
/*  into the new environment.
 */
DECLARE
        Scopes
        Decls Formals
        Type
        Statmts Cases
        FormalTypes
        Fields TagField
        Variants
                = [ Level :int INHERITED ].

PROG    = { /* PROG = Decls Scope. */
            Scope:Level   := 1;
}.
Scope   = { /* Scope = Decls Statmts.
            /* Calculate level as first for everything.
             */
            Decls:Level   :- Level;
            Statmts:Level :- Level;
}.
Proc    = { Formals:Level    := Level+1;
            ResultType:Level :- Level;
            Scopes:Level     :- Formals:Level;
}.
Record  = { Fields:Level  := Level+1;
}.
```

First are all nodes enumerated which require this attribute. And if the node has child nodes, then the child nodes also inherit the attribute.

The `PROG` scope is defined to be level 1. This level gets incremented at every instance of a routine and record. Note that in `routines` the return type is determined to be in the outer scope. And in routines the scope is the smae for the declarations and the statements.

The code, generated by the attribute evaluator, only calculates a new `[Level]` for a scope after the value of the outer scope is calculated.

Due to the `INHERIT` type of the attribute, the value is copied unchanged on to the other nodes in the AST. (At least when they are declared to have this attribute.) This does not require any additional user programming.

4.1.2 What is the order of evaluation?

The nodes in the AST can be associated with attributes and attribute computations. It is up to the generator tools to decide what the dependencies between all the attributes and computations is. It is very simple to create loops in the computations, where attribute `[A]` depends on attribute `[B]`, and vice versa. These instances are reported by the tools. The user has very little influence in this other than to modify attribute calculations, or introduce extra attributes[1].

If relations become more complex, then dependencies are not always obvious. Users of AG should invest time to read [6] carefully and experiment with the interactive dialog system. It also has the possibility to look at the inserted default attribute computations.

It is also possible to trace the order of evaluation, but this produces a large volume of text. And it might require a little "hacking" in the print routines use in the AGgenerated code.

4.2 Symbol table and type management

A compiler uses a symbol table keeps track of scope and binding information about names. Changes to the table occur if a new name of new information about a name is discovered. Information is entered into the symbol table at various times. Standard available items are inserted initially.

Although the term "symbol table" is used, is it not implemented as a table but as a tree. And where it speaks of symbols, it should actually speak about declarations. For the management of names is done by a module, supplied by the COCKTAIL-tools. The tree consists of nodes describing the scoping of levels and each of the scoping levels has a list of declared objects. These objects have subtrees associated with them to describe elements of the object. Eg. a variable has a type-tree connected, a function has a parameter-list and a return type, a record has fixed fields, and possible variant fields.

The whole definition of this structure is captured in the small tree grammar which is included below.

```
Objects         = <
  ONoObject     = .
  Object        = ONext: Objects <
  OLabel        = [ LabelNum :int ].
  OVField = Objects -> [ Vindex :tIntSet ] .
  ONamedObj     = [ Name :tIdent ] [ Pos :tPosition ] <
    OProg = StdObjs:Objects MyObjs:Objects
    OConst      = -> TType [ Val :tValue ] .
    OEnum       = [ EnumVal :int ] -> TType.
    OField      = TType .
```

---

[1] It is possible to specify extra relations with `BEFORE` or `AFTER`, but these only work in limited cases.

```
    OTypeDecl   = -> TType .
    OVar        = TType .
    OStdProc    = [ StdProcKey :tStdProcKey ] TType.
    OProc       = [ ProcType :tProcType ] ->
                  ParTypes TType
  ScopeObjs:Objects EnvObjs:Objects
  MyObjs:Objects.
  >.
 >.
>.

TType          = <
  TNoType      = .
  TInteger     = .
  TReal        = .
  TBoolean     = .
  TChar        = .
  TString      = [ Size :int ].
  TText        = .
  TNIL         = .
  TEmptySet    = .
  TProcStdType = [ StdProcKey :tStdProcKey ]
                 [ Parno :int ] .
  TProcTransfer = [ StdProcKey :tStdProcKey ].
  Constructor  = TypeObj: Objects <
    TEnum      = -> [ MaxEnum :int ] Objects.
    TSubrange  = -> TType [ Lwb :int] [ Uwb :int ] .
    TArray     = -> IndexType: TType ElementType:TType.
    TRecord    = -> Objects.
    TSet       = -> TType.
    TFile      = -> TType.
    TPointer   = [ ForwardDecl :Boolean ] -> TType.
  >.
>.

ParTypes       = <
  NoParType    = .
  ParType      = Next: ParTypes [ IsVAR :Boolean]
                 TType [ Ident :tIdent ] .
>.

Env            = Objects Hidden: Env .
```

Using this (small) grammar, it is possible to capture the whole state of the "symbol table" during semantic analysis. Possibly, other attributes are required during the allocation and code generation, but these are not included here.

It is important to realize that there are now two AST's in the compiler. The first one is the actual program tree. The second one is a tree build by the semantic pass, and contains the declaration structure of the program. And although Grosch[4] gives an example in which both items are place in *one* tree grammar, this is considered to complicated for our problem.

Note that is is possible to access the symbol table from two diferent entries.

The first way is the regular way: As a tree containing symbols. And a lists of objects are placed in nodes where they are declarated. Access to symbols goes through to top-level node.

Figure 4.1: Structure of the symbol table

Another entrance to the symbol table is through the list of environments. Each of the scoping levels creates an enviroment. And all enviroments are linked in the hierarchical order specified the program.

Following the lists with objects and enviroments, one can only find those objects that should be visible according to the Pascal scoping rules.

4.3 Scoping

The Standard[6.2.2] defines the scoping rules in term of *regions* and *defining-points*, this document shall refer to this principle in terms of scoping and use a more informal terminology.

Figure 4.2: Environments in the symbol table

During the semantic phase scoping determines the visibility of identifiers and labels. Previously defined items can be hidden by new ones with the same name, if the old one was defined at an outer scope. The creation of new scopes can occur at three places:

- At the begining of a program structure.

  The objects required by the standard are considered to be at a higher scoping level that the global objects in the program[2].

---

[2] This makes that they can be easily overruled by local declared objects (Standard[6.2.2.10])

- Upon entry of a `routine`.

  The name of the routine, and its return type, are still part of the outer scope, the parameters however start in a new scope.

- Within a `record`.

  In a record all declared items are not directly accessible from the program, but have to be prefixed by the record variable descriptor.

4.3.1 Attributes used for scoping

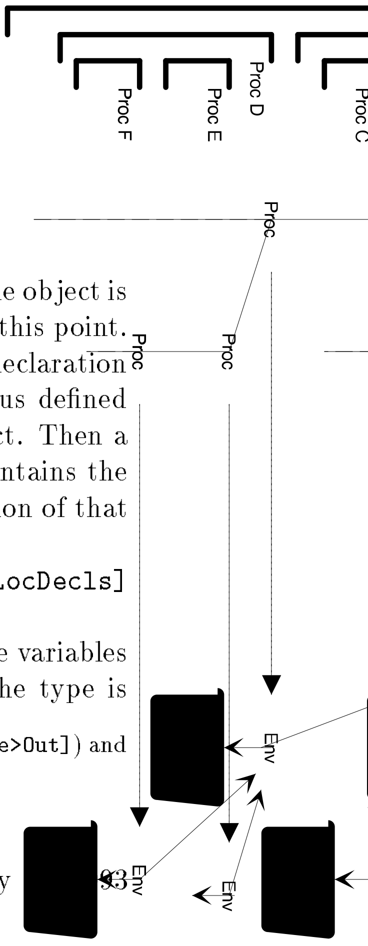

Figure 4.3: Chain of nested scopes

Several rules which define the occurrence of an object require that the name of the object is unique within the current scope. A `threaded`[3] list keeps the names available up to this point. It only holds the names of the objects defined. This list is passed on to every declaration node that can create new names. During verification of the name are only previous defined names considered, so if the name was already available, then there will be a conflict. Then a list is kept per definition-statement: `[LocDecls :tObjects THREAD]`. This list contains the nodes of the objects defined in the definition-statement, and holds all the information of that node.

Eg. If the definition statement declares a variable-list of type `integer`, then `[LocDecls]` will contain a list of nodes, all of type `OVar` with the type appended.

The attribute evaluation first creates a list of nodes which holds the names of the variables declared, and then processes the type of the list (in this case integer). After the type is

---

[3] A threaded list `[<Name>]` is a list which consists of two attributes. One is synthesized (`[<Name>Out]`) and the other inherited (`[<Name>In]`). This is used for simple notational convenience.

obtained, an attribute is returned which is then placed in the already created variable nodes. The exact order of visit sequences is determined by the attribute evaluator tool.

The `[LocDecls]` is inserted in the environment immediately after the processing of the declaration-statement has completed. The environment is kept in a list `[Env :tEnv INHERIT]` which is passed onto the next declaration-statement. Thus `[Env]` always contains all names and information on the names, up till the current declaration. After the environment is completely calculated in the declarations-part, it is passed onto the statements-part. Here it is used to lookup the references made in the statements.

In figure 4.3 shows the relation between `[Env]` and the nesting of routines in scoped modules. It clearly shows that the scoped environment is not visible from the program-statement block, but the procedure statement block can reach all declarations by walking down the `[Env]`-chain. Notice that the procedure is also connected to the `[LocDecls]` of the program declarations block, this is to indicate that the declaration of the routine is done in the scope in which the routine is declared.

Usually are the attributes propagate by default copy operations as is described in [6]. But on several occasions is there a need to overrule these defaults. Strong examples of these are found in the code for Var and Array where the `[{Loc,Scope}Decls]` have to be propagated to the subtrees in a specific order.

Maintaining the name-space for a scope

Several rules which defined the occurrence of an object require that the name of the object in unique within that scope. For this a threaded list is kept: (`[ScopeDecls :tObjects THREAD]`) It only contains nodes with the names of the defined objects. This list is passed onto/into every node that can create a new name. Verification for duplicated names is done using this list.

Loading a `with`-statement scope

The purpose of the `with`-statement is to bind names in the record's scope closer than the same named items in outer scopes.

This is easily done by creating a extra environment which contains the names in the record scope, and prepend this environment to the already available environments. Name searches will then check the record scope first, and in case of a match with a record element, the record object is returned.

Forward references made by pointer declarations

Although the above description is rather simple and efficient, there is one problem with it. It does not account for forward references made in pointer declarations:

```
Type
        pointer = ^forwardreference;
        forwardreference = ...... ;
```

Here the name `forwardreference` has not been declared, and it can only be considered an error after the whole declarations-part is processed. Furthermore has the type of the forward reference to be substituted in the pointer declaration.

This can be achieved in several ways:

1. All unresolved references are stored in a separate list, and in the list is the location of the pointer declaration. Then once the declaration-part is finished, all nodes on the unresolved list are preprocessed and it is considered an error when any of the items cannot be resolved is it considered an error.
2. Once the all declarations are processed, the thus created environment is passed onto all the declarations again. And all instances with forward pointer use this `[FullEnv :tEnv INHERIT]` to find the `forwardreference`.

   The complete environment is passed up int the tree from the end of the declaration part by `[EnvUp :tEnv]`.
3. .... Others

The second version is implemented. The only references to the `[FullEnv]` are made in the pointer-declaration rule. Finally when this has taken place in the whole declaration part, `[FullEnvUp]` is worked to the top of the declarations part , which is the signal that everything in decl-part has been calculated.

Forward routine declarations

Another forward reference problem is the forward declaration of routines. In comparison of with the pointers are the routine forward declarations explicit, and should contain all the correct information of the routine header.

To be able to administer forward declarations, and the correct name usage, is the routine inserted in the symbol table, with all its additional information (parameters, return type), but with an attribute indicating its forward declaration. (`[ProcType]` = `ProcForward` or `FuncForward`) Once the actual routine is defined, the attribute is changed, and the parameters are inserted in the environment created for the scope of this routine.

Extension The semantics of Pascal are widened a little in this topic. It is valid to repeat the parameters of the forward declaration in the actual definition of the routine. The two declarations have to be an *exact* match for all parameters and the return type.

Obtaining the *real scope* environment

Certain statements need access to the environment introduced by the scope the statements are in, even when nested `with`-statement environment are created. Examples are the `For`-statement and the `Goto`-statement. In the `For`-statement the standard requires that the loop variable is declared in the innermost scope. The `Goto`-statement requires that the jump does not exit the current scope, hence that the label is declared in the innermost scope.

For this `[ScopeEnv :tEnv INHERIT]` is passed onto the statements of a scope. It is just a plain copy of the regular `[Env]`, but it does not get modified upon entry of local scopes introduced by `With`-statement. It is used in the semantic checks for the nodes: For, Goto and ActLabel.

4.4 Declarations

All declarations are stored in elements of the tree grammar which is described in the section on the symbol table. The resulting nodes are then combined to linked lists which are found in the `[Env]`, `[FullEnv]`, `[LocDecls]` and `[ScopeDecls]` chains. In the following sections are parts of the attribute evaluation explained which have to do with the creating of the "simple" parts of the symbol table.

4.4.1 Labels

The declaration of a label contains all the required information, and the newly created node can be inserted in the `[Decls]` chains.

```
Label   = {         LocDeclsOut        := mOLabel(LocDeclsIn, labelnum);
               Next:ScopeDeclsIn       := mOLabel(ScopeDeclsIn, labelnum);

}.
```

4.4.2 Constants

First the constant is inserted into the `[ScopeDecls]` chain to make the name available, then the attribute `[Object]` is used the keep a link to the the node which goes into the symbol table. since the actual value and type are not yet evaluated. This incomplete **OConst** is inserted in the symbol table, and the evaluator will insert the `Typ` and `Val` information in the node during a next visit.

`[TmpObj]` is only used a place holder for the evaluation, the stored value is not actually used. It is used at various other locations, but since it is an attribute is it local to each and every node.

```
Const   = {     /* There is no need to pass the {Loc,Scope} decls into
                /* the constant expression since nothing will be declared
                /* there.
                 */
                Next:ScopeDeclsIn:= mOConst(ScopeDeclsIn, Ident, Position);
                Object           := mOConst(LocDeclsIn, Ident, Position);
                LocDeclsOut      :- Object;

                TmpObj           := FillOConst(Object, Expr:Typ, Expr:Val );
}.
```

4.4.3 Types

This is a large section of code which is spread over several modules, and has a number of very complicated calculations. Its main concern is to obtain the type of a subtree. Usually in a declaration, but it could also be in parameters and expressions. The `[Typ :tType]` is a link to elements in the symbol table which are in the **TType** rule, so each `[Typ]` points to a **TType** structure.

Simple types

Lets do a simple one first. Consider the code: `'type NewType = OldType;'`. Here `NewType` also gets the type of `OldType`. Here `OldType` is a TypeId, an identifier which represent a type. The accompanying attribute evaluation is:

```
/*      A Type identifier has to be defined some where in this or outer
/*      environments.
 */
TypeId  = {     /* In pascal are types with different names but the same
                /* structure considered different.
                /* So it's good practice to return the Type Object pointer
                /* of the Type. Comparing addresses can tell us then if
                /* TYPE's are equal.
                 */
                Typ := GetObjectType(IdentifyNameEnv( Ident, Env ));
}.
```

Where `IdentifyNameEnv` delivers a pointer to an object with the name, and searches the whole environment in `Env`. `GetObjectType` returns the type of that object.

Likewise simple code is used for SetType and FileType.

Type declarations The whole statement `type NewType = integer;` is a type declaration, and is processed by the following code:

```
TypeDec = {
        /* create the Object and pass the scope list on into Type */
        Type:ScopeDeclsIn  := mOTypeDecl(ScopeDeclsIn, Ident, Position);
        Next:ScopeDeclsIn  :- Type:ScopeDeclsOut;
             ScopeDeclsOut :- Next:ScopeDeclsOut;

        Object             := mOTypeDecl(LocDeclsIn, Ident, Position);
        /* and Fill the object with the Type found */
        Type:LocDeclsIn    :- Object;
             LocDeclsOut   :- Type:LocDeclsOut;
        TmpObj             := FillOTypeDecl(Object, Type:Typ);

        /* make shure that the Type is substituted before the
        /* attribute gets returned.
         */
        Typ                :- Type:Typ;
}.
```

This code shows all the characteristics of most declaration processing. In the first three statements is the `[ScopeDecls]` chain updated, making `NewType` an element of this scope. In the second set of actions, `[Object: tObject]` creates the object in the symbol table, inserted in the `[LocDecls]` chain, and then `[TmpObj]` is used to update the entry in the symbol table with the `[Typ]`[4]. The last part puts the type in the attribute in the TypeDec.

---

[4] An interesting exercise would be the removal of the temporary `[TmpObj]` and calculate `[Object]` in one go.

Subranges Slightly more complex is a subrange. Here several subtrees are "joined" in the SubRange. In the AST are there two expression subtrees which return the bounds of the subrange.[5]

```
/* Type = SubRange =   LowRange:Expr HighRange:Expr.
/* TType = TSubrange = TypeObj -> TType [Lwb :int] [Uwb :int]
 */
SubRange = {
        /* Subrange doesn't define any extra objects */
        ScopeDeclsOut   :- ScopeDeclsIn;
        LocDeclsOut     :- LocDeclsIn;

        TmpType         := mTSubrange(LocDeclsIn);
        Typ             := FillTSubrange(TmpType,
                        LowRange:Typ, LowRange:Val, HighRange:Val );

}.
```

Using the dialog system of AG the following visit sequence is calculated:

```
SubRange

visit   parent 1. time to compute
        Level
        ScopeDeclsIn
compute ScopeDeclsOut
visit   parent 2. time to compute
        LocDeclsIn
        Env
compute HighRange:Env
visit   HighRange 1. time to compute
        HighRange:IsConst
        HighRange:Val
        HighRange:Typ
compute LowRange:Env
visit   LowRange 1. time to compute
        LowRange:IsConst
        LowRange:Val
        LowRange:Typ
check   condition 12
check   condition 10
check   condition 11
compute LocDeclsOut
compute Typ
visit   parent 3. time to compute
```

---

[5] Here a clear dependancy is shown, which need not be there: `[Typ]` depends on `[TmpType]`, but `[TmpType]` is only used as local attribute. And `[Typ]` is only evaluated once the `[Typ]` and the `[Val]` of the subexpression trees is know. So this code is actually rewritten to directly evaluate `[Typ]`.

```
        FullEnv
visit   parent
```

Which shows that 3 visits to this node are made. The first one calculates `[Level]` and `[ScopeDecls]`. On the second visit the two sub-nodes are visited, and several conditions are checked, and finally `[LocDecls]` and `[Typ]` are done. On the third visit `[FullEnv]` is calculated.

Enumerations Enumerations consist of a Enumeration-node, which has an EnumId-list appended. The following items have to be created:

- an enumerate type node (TEnum) which serves a main anchor for this enumeration.
- for each of the elements of the enumeration an object node (OEnum) which holds the name, type, and the "index" value of the identifier.

The code for Enumeration creates the type, and initializes in index counter(`[IndexIn]`. The enumeration type is passed onto the elements of the enumeration. The index value is incremented every time is passed onto a new EnumId. At each of the EnumId is a new name defined, which is inserted into the environment. (through `[{Loc,Scope}Decls]`)

At the end of the EnumId-list, is the total number of elements known. The total is compensated because the first element has value `0`. And this value is copied back to the root of the list by automatic generated rules.

A second attribute to OEnum is the basic type of the enumeration, it points *back* to the TEnum to which it belongs. Thus creating a cycle in the constructed tree. During attribute evaluations in the symbol table this has to be taken into account, since otherwise the evluator never terminates. This is done by hidding the back-link in the tree from the evaluator: the back-links are specified in a seperate CG module `Cycles` which is only used during the creation of tree nodes.

More complex types

Chins can get more complex when nodes need to merge more information. Or first have to process the information before attributes are evaluated.

Arrays Shown here is the the code which creates an node for an array type. Most of the code involves the creation of the TArray and getting the types calculated. But the standard calls for the creation of an other node (TString) if certain constraints are met. (These constraints are tested with the Puma function `CanBeStringType()`) The standard uses this special type on several occasions, thus is decided to make it a node, instead of adding an attribute to TArray.

Now the problem with this code is, that the constrains are only know after the node has been created. (Due to the order of attribute evaluation) So the the node is "retyped" into TString by modifying its type, instead of disposing the node and recreating a new one. (This would require another indirection). This can only be done if the allocated size for a TArray-node is bigger than for a TString-node.

```
/*  ISO: 6.4.3.2
/*  Any type designated packed and denoted by an array-type having as
```

```
/*  its index-type a denotation of a subrange-type specifying a smallest
/*  value of 1 and a largest value of greater than 1, and having as its
/*  component-type a denotation of the char-type, shall be designated a
/*  string-type.
 */
Array   = {
          Typ             := mTArray(LocDeclsIn);
          TmpType         := {
              if (!CanbeStringType(IndexType:Typ, ElementType:Typ)){
                  /* It's just another array to fill */
                  TmpType = FillTArray(Typ, IndexType:Typ, ElementType:Typ);
              } else {
                  /* HACK-ALLERT
                  /* Transform it into a TString-type.
                  /* Note that this requires that the actual size of
                  /* the node TString is smaller than the TArray size.
                   */
                  TyKind(Typ)       = kTString;
                  TyStringSize(Typ) = GetSubrangeHighLimit(IndexType:Typ);
              }
          };
}.
```

Pointers Pointers are already discussed in previous sections, due to their possible forward references. Pointer-nodes have an attribute indicating the nature of the pointer: `[IsForward :Boolean OUT]`. It's value is determined by a scan through the environment for the identifier. The result of this determines whether the type is ready available, or is going to be obtained in a future pass.

Note the following code:

```
TmpType := { /* is if was a forward reference now is the
             /* time to really substitute the type
              */
            TmpType = ((IsForward)
                ?TmpType = FillTPointer(Typ,
                   GetObjectType(IdentifyNameEnv(Ident,FullEnv)))
                :Typ);
};
```

Which shows the use of `[FullEnv]`, to obtain possible forward references. Due to the usage of this attribute is this code executed in an extra pass which is only required to resolve forward pointer references.

Records

Of all parts of the semantic evaluation are the processing of record declarations and routines the most difficult ones. A good knowledge of both the CG and the AG is required to get a grasp of what is going on.

The only simple part is the processing of the Record-node. It just generates a type TRecord, which is updated with the record structure once it is known.

Now how is the record structure created?

The first and most essential observation is that in a `record` structure no forward references can occur to to elements defined within that same record. Thus can the environment for all elements of the record be equal to that of the outer environment. And the record scope can be constructed after the whole record is processed.

NExt, are all object declared within the record in the same scope. Even when they are defined within a variant part.

These observations greatly simplify the implementation. Upon entry of a record new and empty `[{Loc,Scope}Decls]`-lists are created, and the outer environment is used as current enviroment. The `[LocDecls]`-list is only merged into the environment after the whole record is completely processed.

Fixed record parts The fixed record part are those declarations which are not in the case-variant. They are very, very similar to regular declarations: a list with names followed by a type. This is also cause for the simple code. The names are inserted into the standard lists: `[{Loc,Scope}Decls]`, and its type is inserted once the attributes are known in the root of the list(FixedField).

Variant record parts The variant part is a little harder since it concerns a case-expression, whose type is going to determine the validity of the used case-selectors. The case expression can either be typed or untyped. But in both cases (TagId and NoTagId) an object is created to hold the expression type (CaseIndexType), which in the case of NoTagId does not have an tag-identifier connected to it.

The tag-type has to be of enumerate type, and thus a integer list `[CaseLabels :IntSet THREAD]` is kept to be able to verify possible double usage of tags. For this is an new type created which holds integer sets of arbitrary size. Both references, to this list and to the tag-type, are passed on to all VariantFields

The actual declaration lists in the VariantFields are again constructed from Fields, which allow the variant to contain more sub-variants and/or fixed parts. And the `[{Loc,Scope}Decls]`-lists are propagated through all these elements.

4.4.4 Variables

The declaration of variables is again a very simple one. Variable declarations consist of a list of names, which are inserted into the declaration lists. And a Type to construct the the type for these names.

The is only once item which makes life a little more complex here: `[[VarTypeDone VIRTUAL]]`. This attribute is used to force the attribute evaluator in processing the types of the VarList before constructing the `[Scope:Env]` of the program scope. This is a `VIRTUAL` attribute no storage is used, and for the regular variable lists it does not cause any problems. Thus is this a simple way of forcing the order of the evaluation.

Forcing the order of evaluation Why could this not be done using the `AFTER` and `BEFORE` constructs? Since `[TmpObj]` is a local attribute, and none of the other attributes ever depends on its evaluation, is the evaluator generator allowed to postpone the calculation of `[TmpObj]` to the very last visit sequence. The `AFTER` and `BEFORE` can not be used since there is no other attribute target to specify the relation to. By introducing the `VIRTUAL` an artificial target is created, and is the evaluator generator forced to put the attribute evaluation in the desired visit of the node.

Another solution would be to use a third list with declarations and environments for this purpose, but then this would contain the exact same nodes as the current lists. Thus waisting at least the space for the attributes, and possibly extra copies of parts of the symbol table, next to extra time required to compute the attributes.

The code which forces all this:

```
VarId = { /* to guide the evaluator to insert the type before starting
          /* to process the statements part.
           */
          VarTypeDone     :- Next:VarTypeDone;
          VarTypeDone     AFTER TmpObj;
}.
PROG = {  /* Process the program header parameters before doing anything
  /* in the program scope.
   */
          VarList:VarTypeDone     BEFORE Scope:Env;
}.
```

4.5 Routines

During the semantic evaluation of the routines the formal parameter list is stored with the object of the routine definition. When the routine is used in a call, the actual parameters have to be verified against the formal ones.

The case for regular routines is simple, where as the handling of most of the standard routines is very cumbersome.

4.5.1 Formal and actual parameters

Regular routines

Regular routines are those routines where the parameter list and/or the return type completely adhere to the general Pascal syntax. (Eg. they could be written by a programmer in his Pascal program.)

When the header of a routine is processed a list in the definition space which consists of the following elements:

```
ParTypes       = <
  NoParType    = .
  ParType      = Next: ParTypes [IsVAR :Boolean] TType [Ident :tIdent] .
>.
```

Where `[IsVar]` indicates if the parameter was declared as a `VAR` parameter. `[Ident]` contains the name of the formal identifier.

The actual parameters are stored in a list of the following construction:

```
Params          = <
  NoParam       = .
  Param         = Next: Params REVERSE Par: Expr <
    RegParam    = .
  >.
>.
```

For regular routines the type of the expressions is derived and compared with the type of the parameter at that position. Comparison of the type is done in the `PREDICATE ParamCompare (formal, actual)` routine which checks for general compatibility rules of the parameters. An error is reported when they are not considered compatible.

The formal parameter list is terminated by a NoParType-node, the actual parameter-list is ended by a NoParam-node. If either of them does not find the matching counterpart, an error is returned indicating that there are too many, or few, parameters.

standard routines

Although the standard routines are not actually declared by any programming, they need to be available in the symbol table, manually inserted by the compiler code.

The simplicity of the parameter testing of regular routines is well compensated for by the problems caused by the standard routines. They fall in several categories: fixed number of parameters versus variable number of parameters, (`abs`, `sin`, ...versus `eof`, `write`, ...) and in the type of the accepted parameters: fixed (e.g. `odd`), ordinal (e.g. `succ`), variable (e.g. `write`), allocation (e.g. `new`). Where the last two cause even more difficulties by having versions where all the types are influenced by the type of the first parameter.

For this a special type is created:

```
TType           = <
  TProcStdType  =     /* The type for parameters of STD-routines
                      /* They require some extra work before they turn
                      /* into types which fit into the real type system.
                      /* The Parno indicates the position of the parameter
                      /* on the call list.
                       */
                   [StdProcKey :tStdProcKey ]
                   [Parno :int ]
                   .
>.
```

And the Params-node of the actual parameters is expanded to:

```
Params          = <
  NoParam       = .
  Param         = Next: Params REVERSE Par: Expr <
    RegParam    = .
    ColParam    = Size: Expr .
    ColColParam = Size: Expr Fract: Expr .
  >.
>.
```

(The (Col)ColParam-node elements are only used when the parameters are used in a `write(ln)`-statement.)

In `PREDICATE ParamCompare(formal, actual)`, which is also used for the standard routines, most of the code is dedicated to special cases derived from the requirements for these special parameter lists.

Additional to the `PREDICATE ParamCompare` routine two extra routines are available which also coöperate in this problem:

- `FUNCTION GetProcFormalType`

  `GetProcFormalType` derives, given the actual parameter type, the type of the formal parameter. (it still requires `ParamCompare` to verify compatibility)

- `FUNCTION GetNextParamType`

  `GetNextParamType` generates the type for the next parameter. E.g. if the `[Key]`-value indicates that it is a routine with only one parameter, than the next parameter will be NoParType. (with the actions in normal routines) Or when the routine is a `write` to a non text file, then all next parameters will be the type of the file.

Not implemented: Processing of the parameters for the routines `pack` and `unpack` has not (yet) been done.

4.5.2 Function return types

For regular functions there is only one thing to return: A value with the type as specified upon declaration of the function. Which is easily retrieved from the stored routine object.

standard routines

Again the standard-routines make life interesting.

The easy ones Several standard functions return a fixed type: Arithmetic functions, `odd`, ... This type is just hard coded into the initial definition of these routines.

Transfer functions Other routines have the result type depend on the type of their parameter(s). (They usually only have one parameter). Functions with this property[6]have a hard coded type of:

```
TType          = <
  TProcTransfer =    /* Used for 'required' functions where the return
                     /* type is determined by the input type. (eg. abs())
                      */
                   [StdProcKey :tStdProcKey ].
>.
```

The function type is then determined by calling `PREDICATE IsTransferType`. And if this is the case, then the code calculating the function result type will take the type of the first parameter.

---

[6] Examples are `Succ` and `Abs`

4.6 Expressions

Once an expression is translated to part of an AST, the priorities are found in the tree by the order of the elements in the tree. Thus are there only two expression nodes with operators: BinExpr and UniExpr. The remainder of the sub nodes of Expr have to do with references to the symbol table, are constants or manipulate a reference.

All expression nodes have three specific attributes[7]:

```
[ Typ: tType ]
[ IsConst: Boolean ]
[ Val :tValue ]
```

Note, that although `[Val :tValue]` is available, there are no attempts for *Constant Expression Elimination* made during the semantic evaluation. This value is only used to propagate the value during from parser into the semantic evaluation. Note that this is sufficient for Pascal, as long as constant declarations are not allowed to be full expression of constants. If an expression like `const ZeroOffset = ord('0');` is to be allowed, then constant expressions have to be calculated during the semantic evaluation. Otherwise, the checks for ranges can not be done during this phase.

Currently that part of the AST that holds a constant expression is marked by `[IsConst]`, which is enough information for the current implementation. Like at all other places, `[Typ]` is used to convey the type of the subtree. Since this is not always a trivial computation, results are obtained through the Puma functions `{Bin,Uni}ResultType`, which take as operands the operator and the subexpression(s)-type and return a resulting type.

Variables When an identifier is used, its object is searched in the environment. From this object are the tree attributes extracted and placed in the AST.

Array indices An Array has two types involved: The type of the index, and the type of its elements. The first one is compared against the indexing expression, the second one is returned as result of the array reference.

Things are a little more complex in `matrix[1,10]` from `matrix = array[1..10,1..10] of Type`. `[ArrayTyp: tType THREAD]` keeps the type administration. With each index operation the remainder of the type is passed on to the next indexing expression, using `GetBaseType()`. Once all indexing is done, the remainder is the type of the complete array reference. This type is then propagated back to the origin of the indexing expression tree, and returned as result type of the array reference.

Pointers If an expression uses a a pointer, then the result type is the type of the item pointed to. `GetBaseType()` solves this.

The `NIL` pointer has a special entry which returns the special type for `NIL` pointers. This pointer will be compatible with all other pointers.

Record selectors A record selector is like a regular variable, only the environment it starts is limited to the current record scope. The type of the expression is the type of the referenced field.

---

[7] Next to general available `[Env]`, `[Level]` and `[Position]`.

Sets A set constructing creates a specific object for a set. Unless the set was the empty set, for which there is a special type (TTypeEmptySet), again compatible with all other sets.

4.7 Statements

Statements require very little processing during the semantics phase. For most of the nodes, verifications of attributes are the only required actions. Except for a few parts:

`Case-statement` The `Case`-statement is required to have only one instance of each label. Of this purpose a list is maintained`[CaseLabels :tIntSet THREAD]`. Which is passed along with the type of the case-expression to the labels of the case elements.

`For-statement` The `For`-statement needs the controlling identifier to be in the local scope. And the type has to match with the types of the limiting expressions.

`With-statement` The statement block in a `with`-statement has extra elements added to the environment. for every element in the `With` header is an extra level of record scope prepended.

4.8 Semantic checks

A large part of the semantic evaluation is done to verify semantic correctness of the source code. In the `Checks` module are the test to assure the compliance with the standard. The checks all are dependant on the attribute `[DoCheck]` which only gets evaluated after all attributes are evaluated.

All tests consist of code which asserts a certain requirement and of code to be executed when the assertion fails. Eg., when an identifier is not declared:

```
Qualid = {      CHECK IGNOREVIRTUAL(DoCheck)
                    ! MYISEQUAL(defs, Object, nNoObject)
                ==> ErrorName ("identifier not declared" , Position, Ident);
}.
```

(Here `MYISEQUAL(c, t1, t2)`, compares the trees `t1` and `t2` for class/node `c`. If all members are equal, the function returns `true`)

All test should be of very simple nature[8]. Otherwise extra attributes should be added which calculate the complex parts of the test.

4.9 Changing the order of evaluation

The extra order requirements are specified to get the evaluator to generate certain visit sequences. The problem is that attribute grammars usually allow a lot of freedom in the order of evaluation, and the extra order requirements are then used to fix some of the bad choices make by the tools.

Maybe not all of these rules are really required, and could even be causing unneeded problems. Also, some of the rules are there to give to order a more intuitive feeling to the

---

[8] This is note required, but using this objective makes programming much easier.

order of evaluation. That is, although the tool generated a correct sequence, it did not follow the order which was considered obvious. Some of these were 'fixed' by using order relations.

For example: The is no strict requirement to check all assertions in the declarations before processing the statements of a scope. It is however the more natural thing to do. First finish one thing, only then continue with the next.

The ones that are required for correct evaluators are marked: 'important'. The aesthetic rules are not marked. (but maybe should be marked 'cosmetic') So be aware of this tar pit, since it is a cause of many problems.

To evaluate the effects of the extra requirements use: `cg -cW7J pascal.cg` and then check the visit sequences in the interactive mode, looking for visit orders which you think are wrong.

Chapter 5

Symbol table evaluations

As means of using the symbol table in the front-end, two operations are performed on the attributed source tree and the symbol table. The fist one is a typical front-en application: Crossreferencing. The second has already more to do with code generation: Storage allocation.

5.1 Crossreferencing

The objects in the symbol table are augmented with two extra attributes: `[ReferencedTo :tIntSet]` and `[AssignedTo:tIntSet]` which are used to keep lists of on which lines certain objects are refered to, and on which lines values are assigned to these objects.

These values are inserted into the symbol table by an evaluator which processes the attributed source tree. These actions are interleaved with other attribute evaluations in this tree, and as such is there no seperate pass over the tree.

The rules for evaluation are specified in a seperate module, which maintains an attribute `[LeftHand :Boolean INHERITED]` to indicate that this expression can be modified by an assignment.

For the generation of the actual crossreference output, a Puma module is created which visits elements of the symbol table in a predetermined order, and at certain nodes prints the contents of the attributes.

5.2 storage allocation

A second application of the symbol table is the storage allocation of the variables.

In this case is the evaluator actually evaluates attributes in the symbol table. Siz attributes are responsible for the allocation:

- `size :int SYNTHESIZED`, `varsum :int SYNTHESIZED`, and `varoffset :int INHERITED`

  For regular variables they keep the size of the object and the total of allocated space in this environment. The inherited attribute `varoffset` is used to specify the offset to the current frame.

- `size :int SYNTHESIZED`, `parsum :int SYNTHESIZED` and `paroffset :int INHERITED`

  these attributes function like the attributes for variables, but the objects in this case are parameter allocations.

- `size :int SYNTHESIZED`, `varsum :int SYNTHESIZED`, `varoffset :int INHERITED` and `variantsize :int INHERITED`

  These are used in the evaluation of the size of a record. `varsum` and `varoffset` are used to caluclate the total size of the object and the offset of each of the elements within the record.

  `variantsize` is used to calculate the variant part,size. During evaluation of the variant parts is the largest part allocated so far stored in `variantsize`. It is added to the total allocation size of the fixed part to obtain the total record size.

Bibliography

[1] British Standards Institution. *Specification fo computer programming language PASCAL*, 1984. ISO 7185 / BS 6192.

[2] H. Emmelmann, F. W. Schroer, and R. Landwehr. Beg – a generator for efficient back ends. *SIGPLAN Notices 24*, july 1989.

[3] J. Grosch and H. Emmelmann. *A Tool box for Compiler construction.*, pages 106–116. Springer Verlag, October 1991.

[4] Josef Grosch. Specification of a Minilax-Interpreter. Project Compiler Generation. 16, Geselschaft für Mathematik und Datenverarbeitung mbH, Forschungsstelle and der Universität Karlsruhe, Vincenz-Priessnitz-Str. 1, D-7500 Karlsruhe, september 1991.

[5] Josef Grosch. Generators for high-speed front-ends. In D. Hammer, editor, *Compiler Compilers, Second International Workshop, CC '88, Procedings*, page 10. Springer Verlag, may 1989.

[6] Josef Grosch. Ag - an attribute evaluator generator. Project Compiler Generation. 16, Geselschaft für Mathematik und Datenverarbeitung mbH, Forschungsstelle and der Universität Karlsruhe, Vincenz-Priessnitz-Str. 1, D-7500 Karlsruhe, september 1991.

[7] Josef Grosch. Ast - a generator for abstract syntax trees. Project Compiler Generation. 15, Geselschaft für Mathematik und Datenverarbeitung mbH, Forschungsstelle and der Universität Karlsruhe, Vincenz-Priessnitz-Str. 1, D-7500 Karlsruhe, september 1991.

[8] Josef Grosch. Preprocessors. Project Compiler Generation. 24, Geselschaft für Mathematik und Datenverarbeitung mbH, Forschungsstelle and der Universität Karlsruhe, Vincenz-Priessnitz-Str. 1, D-7500 Karlsruhe, february 1991.

[9] Josef Grosch. Puma. Project Compiler Generation. 15, Geselschaft für Mathematik und Datenverarbeitung mbH, Forschungsstelle and der Universität Karlsruhe, Vincenz-Priessnitz-Str. 1, D-7500 Karlsruhe, september 1991.

[10] Josef Grosch. Toolarintroduction. Project Compiler Generation. 25, Geselschaft für Mathematik und Datenverarbeitung mbH, Forschungsstelle and der Universität Karlsruhe, Vincenz-Priessnitz-Str. 1, D-7500 Karlsruhe, september 1991.

[11] Kathleen Jensen and Niklaus Wirth. *Pascal User and Reference Manual.* Springer Verlag, second edition, 1974.

[12] Thomas Pittman and James Peters. *The art of compiler design. (theory and practice).* Prentice-Hall, 1992.

Appendix A

Omissions and/or bugs

The following elements are not (yet) implemented. Or are partially implemented and can give cause to aborted runs.

- packed arrays

  Currently this argument is ignored. In records is nothing being packed but things are going aligned to natural boundaries. The same holds for arrays except for `packed array of [] char`, where general byte-strings are being used.

- conformant arrays

  These types of arrays are accepted by the scanner/parser, but no ast-nodes are generated. And thus there is no semantic check, or anything else done with these parts of the code.

A.1 Bugs

None of the bugs are yet known. But be shure, they are there.

Appendix B

Extensions

It is the intention to allow this front-end to accept several dialects of Pascal. Extensions can occur in several modules, where it varies on how the extensions are detected and/or reported as being errors.

The regular expressions which turns the input into tokens cannot be modified on the fly. They are 'compiled' by the scanner generator and transformed into large tables, which are loaded at compile-time. Detection of extensions in the scanner are therefor handled in an ad-hoc way.

The parser is currently build to accept the superset of all extensions. (For as long as no conflicts in the concrete grammar arise.) Diversions of the selected dialect are reported as parser errors.

A much larger distinction in dialects lies in the number of available standard identifiers and standard routines. These are created by auxiliary code, and thus is it very easy to include the required objects dependant on the chosen dialect.

B.1 Comments

Since it is a very common case to use the {} to out-comment parts of the program which use `(**)` as regular comments, this policy is adopted in the scanner. However, comments cannot be nested, other than {`(**)`} or `(*`{}`*)`.

This however means that a { is not terminated by `*)`, as required by the standard.

B.2 Symbols

The following word(s) are also reserved, they cannot be used as identifiers:

1. `OTHERWISE` or `CASE ELSE END`

   It allows to specify a default action to be executed when none of the case specified case selectors matches the case expression.

B.3 Grammar

B.3.1 Declarations

Standard Pascal requires that the declarations are in a restricted order: `labels`, `types`, `vars`, `procs`. This requirement is not enforced in the grammar. If this restriction is required it should be verified in the semantic evaluation, which it currently is not.

Another approach would be a restriction in the grammar, but then it would not be possible to have one front-end accept both versions.

B.3.2 `EXTERNAL` routines

The Standard[6.1.4] already hints at the use of the reserved word `EXTERNAL` as extension to indicate that procedure of function code is not available in the program being compiled. But that the code for the `EXTERNAL` routines is available in separately compiled entities.

B.3.3 `CASE` dfi2aults

It allows the use of an `OTHERWISE` or `ELSE` clause in a `CASE`-statement. When the selector of the `CASE` can not find a matching case-label to proceed with, the 'default' code is executed.

This in contrast to what the Standard[6.8.3.5] indicates, where it is an error when a matching case-label is not available. No such error will be generated when the default action is specified.

B.4 Semantic tolerance

There can be two reasons for semantic tolerance:

- The first case involves non intentional, and thus not documented freedom. This is cause by insufficient restrictions during the semantic checking, and should be considered a programming error in the front-end code.
- The second type are the intended extensions. These are documented in this section

Of the second case there should be only one. The first case will have more, but then those should be fixed. (Or at least documented as being bugs.)

B.4.1 The order of declarations

In standard PASCAL is the order of declarations fixed:

```
label .....
const   .....
type    .....
var     .....
proc    .....
```

This restriction is not enforced by the concrete grammar. The grammar allows any mix of these items in any order. It is up to the remainder of the fronted to determine the validity of the input.

Reasons for allowing this up front, is that it makes it possible to include other files has header files for external modules. If this was not allowed, a separate include file would be required for each of the declaration parts.

The following restrictions are enforced by the semantic evaluator:

- Scoping and visibility

  The regular order of declaration and usage is still enforced. This means that a `Type`-definition is required before any reference to this type can be made in other declarations. The one exception to this rule are pointer-type declarations, which is also exempt in Pascal.

  Declarations and definitions made with an inner scope do not become visible in the parent scope. Which makes the following invalid input:

  ```
  type
      pointer = ^reftype;
  procedure nested;
      type
          reftype = othertype;
      begin ...... end
  ```

- `Write` and `Read` from no text files. The **Standard[]** gives these as short notations for `f^= value; put(f)` or `get(f); value := f^`. This implementation accepts the `Write(f,value,...)` and `Read(f,value,....)` as well.

B.5 identifiers and routines

Listed here are identifiers and routines which are recognized by the front-end, but are not required by the standard.

- `MinInt`, a constant representing the largest negative integer number.

B.6 Making more extensions

Here several guidelines/remarks so additional extensions to the current front-end can be made. Note that it is possible to create (or want) conflicting extensions. If this is the case, then the programmer should try to solve these by introducing switches selecting either one of the dialects.

B.6.1 to the scanner

One should note that the scanner is created at compiler creation time. Thus the rules for the regular expressions are fixed. Thus any extensions to the scanner should be carefully studied.

As an example: Currently the list of reserved words is extended by `OTHERWISE`. This has the consequence that this word cannot be used as identifier.

Programs using this word as regular identifier are no longer accepted by this front-end, because it is always considered a keyword. The keywords are automatically extracted from the grammar specification.

The solution[1] to this would be extra code in the scanner source which matches the keyword, and returns either the token of the keyword, or it just returns the token for identifier, and the value of the identifier. Command line switches can then be used to steer the selection.

B.6.2 to the parser

Like the scanner, is the parser constructed during compiler creation time. So extensions are to be implemented in the grammar specification. It is probably wise to have the parser accept an as large as possible grammar which encompassed all extensions.

Then other means are required to restrict extensions. Eg. by not returning the keyword `OTHERWISE` is it not possible to make use of the `CASE-OTHERWISE-END` construct. But the `CASE-ELSE-END` construct is still allowed in this situation. Here the parser should recognize that this extension is not valid during this compile, and hence report an error.

This is rather simple because it allows or disallows a certain construct. It becomes more difficult when different dialects want almost the same syntax, but the implementation of both creates a conflict in the parser specification. In this case is a careful rewrite of the grammar in place. But this is a lot or work and rather dangerous.

B.6.3 to the semantic evaluation

It might be required to add additional elements to the AST-grammar. The additions are usually harmless, as long as no conflicts the grammar specification itself are created. These new additions are only used when extra code in the AST-constructor starts using these constructs.

Once the new elements are being used, they also required extra code in the semantic description to evaluate the semantic properties of the new elements.

Tools which use the AST will require modifications as well, since they now have to adhere to the expanded input grammar.

B.6.4 to the set of predefined names

Adding predefined names *can* be rather simple. But in one case it is going to be a real pain.

In the simple case the identifier and it's associated information is inserted into the symbol-table before it is returned to the semantic evaluator. Currently this is done in the `predef` routine in `defs.imp`. The actions are simple:

1. create a object for the name.
2. substitute the additional data. Which might involve the creation of a parameter-list first.

This can of course be subjected to conditional inclusion guided by command-line switches.

Another good possibility would be to process extra include files, before any actual information is read from the original input, thus extending the standard environment.

---

[1] This solution is currently not implemented.

Using non standard parameter types The difficult case is the creation of routines which do not have simple prototypes, or any of the more complex ones which are already in the standard implementation.

This could require the extension of the symbol table grammar with extra `TType`s, and additional code in `deftype.pum` to handle the processing of these new objects.

Only then can the new routine be inserted into the symbol table.

Appendix C

Sources

A large part of the sources are specification to the compiler generator tools. These tools will, in turn, generate `"C"`-code which is then compiled. Next to that are several extra modules used which are programmed directly into `"C"`.

C.1 Cocktail sources

- pascal.sca Description of the 'terminals' which are not derived from the parser specification and/or require extra treatment.

  This includes: `string`, `identifier`, `integer`, `real` and `comments` with possible options.

  Is translated into pascal.rex by `rpp` to insert the keywords.

  Generates: Errors.*, Scanner.* Source.*
- pascal.gra Holds the definition of the concrete grammar.
- pascal.2cg Contains the code to construct the AST while parsing the input.

  pascal.gra and pascal.2cg are combined to Parser.par which is run through CG to deliver: Parser.*, Tree.*
- pascal.pum Auxiliary code used during the construction of the AST.

  It generates the file ast_puma.*
- defs.abs Definition of the symbol-table structure.
- defs.imp Code for operations on the symbol-table.

  defs.abs and defs.imp are combined to defs.cg which generates defs.{c,h,TS} when run through CG.
- deftype.pum Puma code used for operations on the symbol table.

  Generates deftype.*.
- pascal.abs Holds the definition of the abstract grammar.
- pascal.sem Code for the semantic evaluation of the AST These are combined to pascal.cg and deliver the files Semantic.*.

C.2 additional sources

C.2.1 extra modules

- globals.h Global definitions.
- olean.h Construct a boolean type.
- myassert.h Redefines assert.h to operate on pointers as well.
- warning.h `#define` to print errors and warnings.
- options.h Prototypes for `getopt(3)`
- debug.c Process debugging options
- intset.c An integer set class by binary trees, and thus the size need not be known in advance. Definitely slower than the `Set`-code in the `reuse`-lib.
- values.c Operations on constants.
- treetest.c `main` body to generate a AST-constructor.
- semtest.c `main` dy to generate a semantic verifier.

C.2.2 small programs

Most of these programs require `Perl` to be available on the system.

- debug-lalr Add additional debugging code to an `LALR`-file. I works for files generated by CG, manually generated files could cause errors. (This is not tested)
- fix-yyw Fix the definitions in the `yy*.w` files, so that the preprocessor do not choke.
- pascal Execute the current front-end. It is a lazy mans user interface, but for now it works.
- testall Run the front-end on all tests in this directory.
- run-tests Run all tests.

Appendix D

Language definitions

D.1 Reserved words

The following words are reserved, they cannot be used as identifiers:

1. `AND`
2. `ARRAY`
3. `BEGIN`
4. `CASE`
5. `CONST`
6. `DIV`
7. `DO`
8. `DOWNTO`
9. `ELSE`
10. `END`
11. `FILE`
12. `FOR`
13. `FUNCTION`
14. `GOTO`
15. `IF`
16. `IN`
17. `PACKED`
18. `PROCEDURE`
19. `PROGRAM`
20. `OTHERWISE`
21. `RECORD`
22. `REPEAT`
23. `SET`
24. `THEN`
25. `TO`
26. `TYPE`
27. `UNTIL`
28. `VAR`
29. `WHILE`
30. `WITH`

D.2 Required identifiers

These identifiers are required by the standard, and are declared at the outermost scoping level.

1. `ABS`
2. `ARCTAN`
3. `BOOLEAN`
4. `CHAR`
5. `CHR`
6. `COS`
7. `DISPOSE`
8. `EOF`

9. EOLN
10. EXP
11. FALSE
12. GET
13. INPUT
14. INTEGER
15. LN
16. MAXINT
17. NEW
18. ODD
19. ORD
20. OUTPUT
21. PACK
22. PAGE
23. PRED
24. PUT
25. READ
26. READLN
27. REAL
28. RESET
29. REWRITE
30. ROUND
31. SIN
32. SQR
33. SUCC
34. TEXT
35. TRUE
36. TRUNC
37. UNPACK
38. WRITE
39. WRITELN

D.3 Grammars

D.3.1 Concrete grammar

```
program           =  'PROGRAM' IDENTIFIER
                     external_files ';'
                     block '.' .

external_files    = <
  noexternfiles   =  .
  filelist        =  '(' ident_list ')'.
>.

block             =  declarations
                     'BEGIN'
                       statements
                     'END'  .

declarations      =  <
  nodeclaration   =  .
  moredeclaration =  declarations
                     declaration .
>.

declaration       =  <
  label_dcl_part  =  'LABEL'
                       labels ';' .
  const_dcl_part  =  'CONST'
                       const_defs ';' .
  type_dcl_part   =  'TYPE'
                       typedefs ';' .
  var_dcl_part    =  'VAR'
                       variable_dcls ';'.
  procdeclaration =  proc_dcl_part ';' .
>.

labels          =  <
  morelabels    =  labels
                   ',' onelabel .
  onelabel      =  UNSIGNED_INT .
>.

const_defs      =  <
  const_defs1   =  const_defs
                   ';' const_def .
  const_defs2   =  const_def .
>.

const_def       =  IDENTIFIER
                   '=' constant .

constant        =  <
  constnum      =  unsigned_num .
```

```
  constplus     =  '+' unsigned_num .
  constminus    =  '-' unsigned_num .
  unsignedident =  IDENTIFIER .
  plusident     =  '+' IDENTIFIER .
  minusident    =  '-' IDENTIFIER .
  stringconstant=  STRING .
>.

unsigned_num    =  <
  intconstant   =  UNSIGNED_INT .
  realconstant  =  UNSIGNED_REAL .
>.

typedefs        =  <
  moretypedefs  =  typedefs
                   ';' typedefn .
  onetypedef    =  typedefn .
>.

typedefn        =  IDENTIFIER '=' type .

type            =  <
  symtype       =  simple_type .
  packstructype =  'PACKED' struct_type .
  structype     =  struct_type .
  pointertype   =  '^' IDENTIFIER .
>.

simple_type     =  <
  simple_list   =  '(' ident_list ')' .
  simple_range  =  constant
                   '..' constant .
  simple_ident  =  IDENTIFIER .
>.

struct_type     =  <
  arraystruct   =  'ARRAY'
                   '['index_t_list ']'
                   'OF' type .
  recordstruct  =  'RECORD'
                       field_list
                   'END'.
  setstruct     =  'SET' 'OF'
                       simple_type .
  filestruct    =  'FILE' 'OF' type .
>.

index_t_list     =  <
  index_t_list1  =  index_t_list
                    ',' simple_type .
  index_t_list2  =  simple_type .
>.
```

```
field_list     =  <
  field_list0   =  .
  field_list0a  =  ';' .
  field_list1   =  fixed_part .
  field_list1a  =  fixed_part ';' .
  field_list2   =  fixed_part
                   ';' variant_part .
  field_list3   =  variant_part .
>.

fixed_part     =  <
  onerecord    =  record_section .
  morerecords  =  fixed_part
                  ';' record_section .
>.

record_section    =  <
  record          =  ident_list
                       ':' type .
>.

variant_part   = <
  variant_part1 =  'CASE' tag_field
                   'OF' variants .
  variant_part2 =  'CASE' tag_field
                   'OF' variants
                   ';' .
>.

tag_field      =  <
  named_tag    =  IDENTIFIER
                  ':' IDENTIFIER.
  tag          =  IDENTIFIER .
>.

variants       =  <
  onevariant   =  variant .
  morevariants =  variants
                  ';' variant .
>.

variant        =  <
  variant1     =  case_label_list
                  ':' '(' field_list ')'.
>.

variable_dcls  =  <
  morevar_dcls =  variable_dcls
                  ';' variable_dcl .
  onevar_dcl   =  variable_dcl .
>.

variable_dcl    =  ident_list ':' type .
```

```
ident_list      = <
  moreidents    = ident_list
                  ',' IDENTIFIER .
  oneident      = IDENTIFIER .
>.

proc_dcl_part     = <
  procedure_dcl   = proc_head body .
  function_dcl    = func_head body .
>.
proc_head         = proc_heading ';'.

proc_heading      = 'PROCEDURE'
                    IDENTIFIER
                    formal_params .

func_head         = func_heading ';'.

func_heading      = 'FUNCTION'
                    IDENTIFIER
                    function_form .

function_form     = <
  function_form1  = .
  function_form2  = formal_params
                    ':' IDENTIFIER .
>.

body              = <
  body1           = block .
  body2           = 'FORWARD'.
  body3           = 'EXTERNAL'.
>.

formal_params     = <
  formal_params1  = .
  formal_params2  = '(' formal_p_sects ')'.
>.

formal_p_sects    = <
  formal_p_sects1 = formal_p_sects
                    ';' formal_p_sect .
  formal_p_sects2 = formal_p_sect .
>.

formal_p_sect     = <
  formal_p_sect1  = param_group .
  formal_p_sect2  = 'VAR' param_group .
  formal_p_sect3  = proc_heading .
  formal_p_sect4  = func_heading .
>.
```

```
param_group       = ident_list
                    ':' paramtype .

paramtype         = <
  paramtype1      = IDENTIFIER .
  paramtype2      = 'ARRAY'
                    '[' index_specs ']'
                    'OF' paramtype .
  paramtype3      = 'PACKED' 'ARRAY'
                    '[' index_spec ']'
                    'OF' IDENTIFIER .
>.

index_specs       = <
  index_specs1    = index_specs
                    ';' index_spec .
  index_specs2    = index_spec .
>.

index_spec        = IDENTIFIER
                    '..' IDENTIFIER
                    ':' IDENTIFIER .

statements        = <
  statements1     = statements
                    ';' statement .
  statements2     = statement .
>.

statement         = <
  statement1      = .
  statement2      = UNSIGNED_INT
                    ':' statement .
  statement3      = 'BEGIN'
                       statements
                    'END' .
  statement4      = variable
                      ':=' expression .
  statement5      = procedure_call .
  statement6      = 'GOTO'
                      UNSIGNED_INT .
  statement7      = 'IF' expression
                    'THEN' statement .
  statement8      = 'IF' expression
                    'THEN' statement
                    'ELSE' statement .
  statement9      = 'CASE' expression
                    'OF' case_list
                         case_end .
  statement10     = 'WHILE' expression
                    'DO' statement .
  statement11     = 'REPEAT' statements
                    'UNTIL' expression .
```

```
  statement12     =  'FOR' IDENTIFIER
                           ':=' expression
                     direction expression
                     'DO' statement .
  statement13     =  'WITH' rec_var_list
                     'DO' statement .
>.

direction         =  <
  direction1      =  'TO'.
  direction2      =  'DOWNTO'.
>.

procedure_call   =  <
  noarg_proc_call= IDENTIFIER .
  arg_proc_call  = IDENTIFIER
                   actual_params .
>.

actual_params    =  '(' actuals_list ')'.

actuals_list     =  <
  actuals_list1  =  actuals_list
                    ',' actual_param .
  actuals_list2  =  actual_param .
>.

actual_param     =  <
  actual_param1  =  expression .
  actual_param2  =  expression
                    ':' expression .
  actual_param3  =  expression
                    ':' expression
                    ':' expression .
>.

case_list        =  <
  case_list1     =  case_list
                    ';' case_list_elem .
  case_list2     =  case_list_elem .
>.

case_list_elem    =  <
  case_list_elem1 =  case_label_list
                     ':' statement .
  case_list_elem2 =  .
>.

case_label_list    =  <
  case_label_list1 =  case_label_list
                      ',' constant .
  case_label_list2 =  constant .
>.
```

```
case_end         =  <
  case_end1      =  'END'.
  case_end2      =  'OTHERWISE'
                       statements
                    'END' .
  case_end3      =  'ELSE'
                       statements
                    'END'.
>.

rec_var_list     =  <
  rec_var_list1  =  rec_var_list
                    ',' variable .
  rec_var_list2  =  variable .
>.

expression      =  <
  expression1   =  simple_expr .
  expression2   =  simple_expr
                   relational_op
                   simple_expr .
>.

simple_expr     =  <
  simple_expr1  =  term .
  simple_expr2  =  '+' term .
  simple_expr3  =  '-' term .
  simple_expr4  =  simple_expr
                   add_op
                   term .
>.

add_op          =  <
  add_op1       =  '+'.
  add_op2       =  '-' .
  add_op3       =  'OR'.
>.

term            =  <
  term1         =  factor .
  term2         =  term mult_op factor .
>.

factor          =  <
  factor1       =  variable .
  factor2       =  unsigned_lit .
  factor3       =  '(' expression ')' .
  factor4       =  set .
  factor5       =  'NOT' factor .
>.

unsigned_lit    =  <
```

```
  unsigned_lit1 =  unsigned_num .
  unsigned_lit2 =  STRING .
  unsigned_lit3 =  ’NIL’ .
>.

set             =  ’[’ member_list ’]’ .

member_list     =  <
  member_list1  =  .
  member_list2  =  members .
>.

members         =  <
  members1      =  members
                   ’,’ member .
  members2      =  member .
>.

member          =  <
  member1       =  expression .
  member2       =  expression
                   ’..’ expression .
>.

variable        =  <
  identvar      =  IDENTIFIER .
  procfuncvar   =  IDENTIFIER
                   actual_params .
  arrayvar      =  variable
                   ’[’ expressions ’]’ .
  recvar        =  variable
                   ’.’ IDENTIFIER .
  pointvar      =  variable ’^’ .
>.

expressions       =  <
  moreexpressions =  expressions
                     ’,’ expression .
  oneexpression   =  expression .
>.

relational_op     =  <
  relational_op1 :  ’=’ .
  relational_op2 :  ’<’.
  relational_op3 :  ’>’.
  relational_op4 :  ’<=’.
  relational_op5 :  ’>=’.
  relational_op6 :  ’<>’.
  relational_op7 :  ’IN’.
>.

mult_op         =  <
  mult_op1      :  ’*’ .
  mult_op2      :  ’/’ .
  mult_op3      :  ’DIV’ .
  mult_op4      :  ’MOD’ .
  mult_op5      :  ’AND’ .
>.
```

D.3.2 Abstract grammar

```
PROG          = [ Ident :tIdent ]
                [ Position :tPosition ]
                VarList
                Scope.
Scopes        = <
  Scope       = Decls Statmts.
  NoScope     = .
>.

Decls         = <
  NoDecl      = .
  Decl        = Next: Decls <
    Label     = [ labelnum ]
                [ Position :tPosition ].
    Var       = VarList Type .
    DeclObj   = [ Ident :tIdent ]
                [ Position :tPosition ]<
      Const   = Expr .
      TypeDec = Type .
      Proc    = [ ProcType :tProcType ]
                Formals
                PrimType
                Scopes .
    >.
  >.
>.

Formals       = <
  NoFormal    = .
  Formal      = Formals
                [ IsVar :Boolean ]
                ParIds Type .
>.

ParIds        = <
  NoParId     = .
  ParId       = [ Ident :tIdent ]
                [ Position :tPosition ]
                ParIds .
>.

VarList       = <
  NoVar       = .
  VarId       = [ Ident :tIdent ]
                [ Position :tPosition ]
```

```
                VarList .
>.

Type          = <
  NoType      = .
  SimpleType  = <
    Enumeration = EnumIds .
    SubRange  = Expr Expr .
    PrimType  = <
      NoPrimType = .
      TypeId  = [ Ident :tIdent ]
                [ Position :tPosition ].
    >.
  >.
  Pointer     = [ Ident :tIdent ]
                [ Position :tPosition ].
  SetType     = SimpleType .
  FileType    = Type .
  Array       = SimpleType
                Type .
  Record      = Fields.
  ProcType    = Formals PrimType.
  ConfArray   = SimpleType
                SimpleType
                Type
                Type .
>.

Fields        = <
  NoField     = .
  Field       = Fields  <
    EmptyField  = .
    FixedField  = FieldIds Type .
    VariantField= TagField Variants .
  >.
>.

FieldIds      = <
  NoFieldId   = .
  FieldId     = [ Ident :tIdent ]
                [Position :tPosition]
                FieldIds .
>.

EnumIds       = <
  NoEnumId    = .
  EnumId      = [ Ident :tIdent ]
                [Position :tPosition]
                EnumIds .
>.

TagField      = [Position :tPosition]
                Type <
```

```
  NoTagId     = .
  TagId       = [ Ident :tIdent ] .
>.

Variants      = <
  NoVariant   = .
  Variant     = CaseExprs Fields
                Variants  .
>.

Statmts       = <
  NoStat      = .
  Stat        = [Position :tPosition]
                Statmts  <
    EmptyStat = .
    ActLabel  = [ HereLabel ] Stat .
    Assign    = Designator Expr .
    Call      = Qualid Params .
    Goto      = [ ToLabel ] .
    If        = Expr
                Statmts
                Statmts .
    While     = Expr
                Statmts .
    Repeat    = Statmts
                Expr .
    For       = Qualid Expr Expr
                [ ForUp :Boolean ]
                Statmts .
    Case      = Expr Cases .
    With      = WithExprs Statmts .
  >.
>.

Cases         = <
  NoCase      = .
  CasePart    = Cases  <
    CaseBlock = CaseExprs Statmts.
    CaseDefault = Statmts .
  >.
>.

Params        = <
  NoParam     = .
  Param       = Params  Expr <
    RegParam  = .
    ColParam  = Expr .
    ColColParam = Expr Expr .
  >.
>.

CaseExprs     = <
  NoCaseExpr  = .
```

```
  CaseExpres  = Expr CaseExprs  .
>.

WithExprs     = <
  NoWithExpr  = .
  WithExpr    = Expr WithExprs  .
>.

IndexExprs    = <
  NoIndexExpr = .
  IndexExpr   = Expr IndexExprs  .
>.

Expr          = [Position :tPosition] <
  NoExpr      = .
  BinExpr     = [ Operator :int]
                Expr
                Expr .
  UniExpr     = [ Operator    :int]
                Expr .
  Consts      = [ CVal :tValue ] <
    IntConst  = [ sign :int ].
    RealConst = [ sign :int ].
    StringConst = .
    CharConst = .
  >.
  FuncCall    = Qualid Params .
  Set         = SetElems .
  Designator  = <
    Qualid    = [ Ident :tIdent ].
    Subscript = Designator IndexExprs.
    Deref     = Designator .
    NIL       = .
    Select    = Designator
                [ Field :tIdent  ] .
  >.
>.

SetElems      = <
  NoSetElem   = .
  SetElem     = SetElems  <
    SetElement = Expr .
    SetRange  = Expr
                Expr .
  >.
>.
```