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.

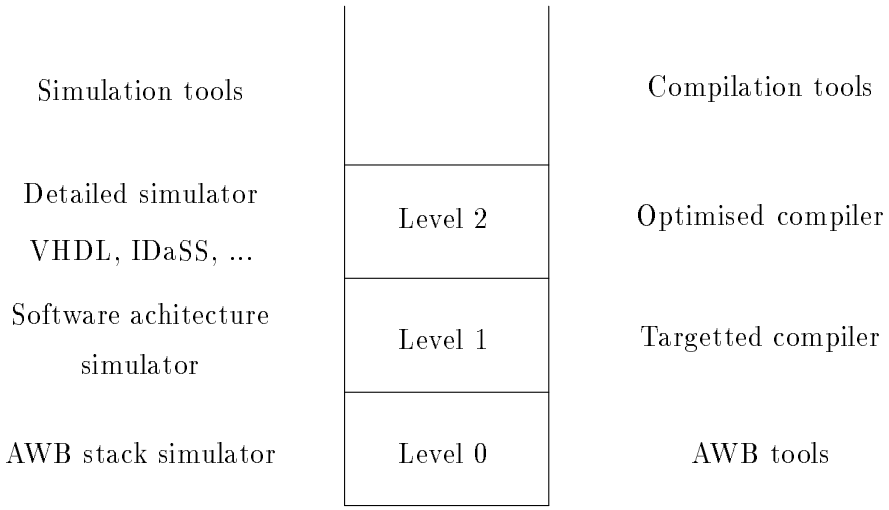

Figure 1.1: Visualisation of the design hierarchy

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).

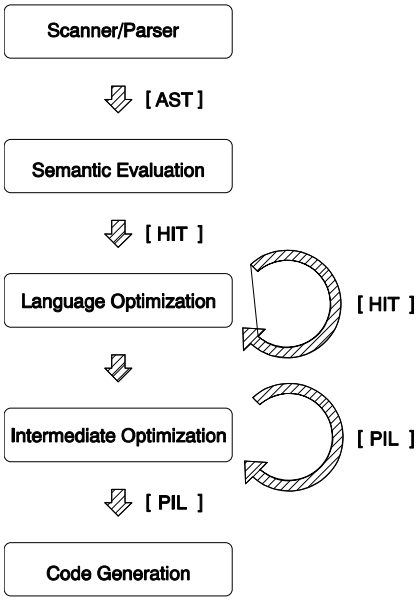

Figure 1.2: General architecture of a compiler

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.

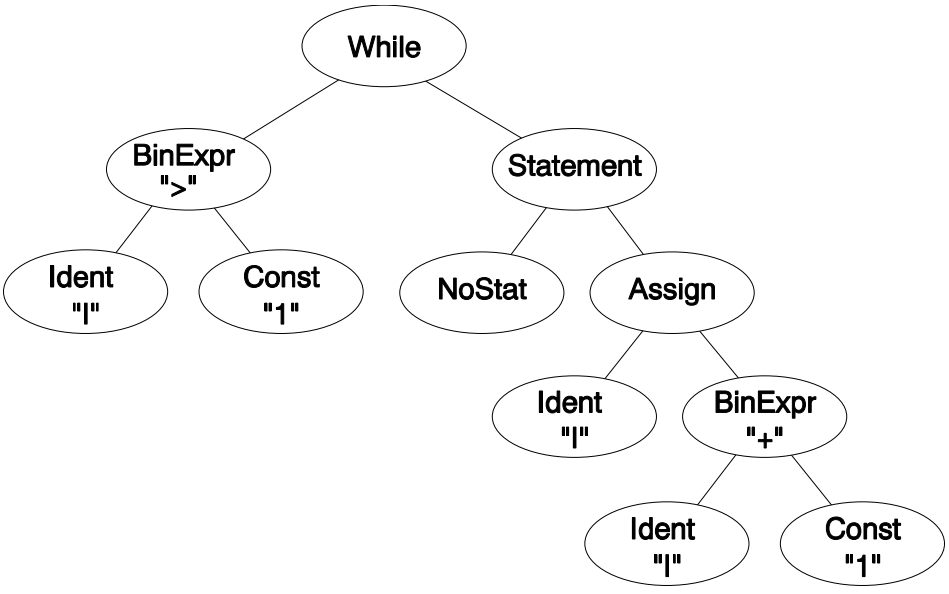

Figure 3.1: Tree construction of a code piece

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:

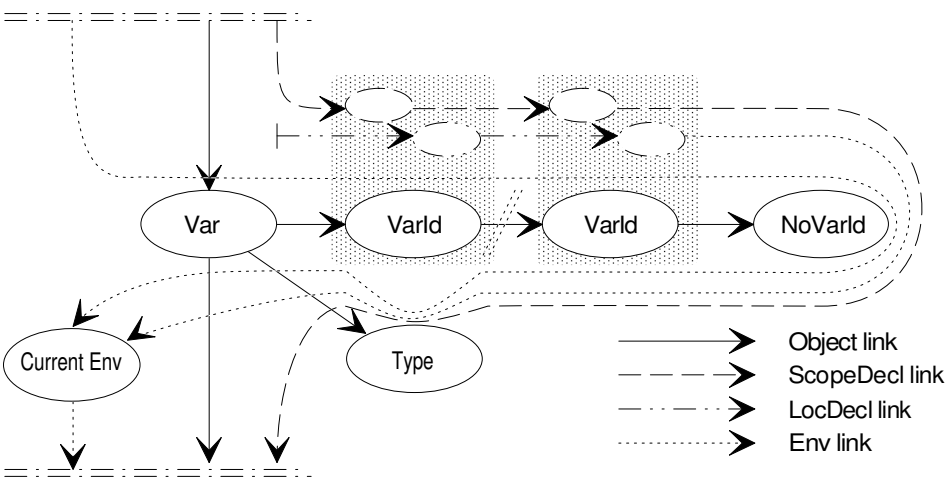

Figure 3.2: The AST for a VarList

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.