IMPLEMENTING AN OBERON COMPILER WITH PCCTS

Steven A. Robenalt
Scobenalt Engineering
robenalt@orange.digex.net

Abstract. This paper describes the implementation of a compiler for the Oberon-2 programming language using the tools included in the Purdue Compiler Construction Tool Set (PCCTS). A project overview describes the structure of the compiler and highlights some of the problems which were noted. Next, the use of semantic predicates to resolve some of the ambiguities in the Oberon-2 language specification is discussed. The included examples highlight the use of some features of PCCTS as they are used in an actual project.

1. Introduction

The Oberon language was originally defined as part of a research project performed by Niklaus Wirth at the Institut für Computersysteme, ETH Zentrum, Zurich, Switzerland from 1986 through 1989 [WG92]. The goal of the project was to design and implement a complete workstation-class computer from scratch, including hardware, operating system, compiler, and software. The name Oberon refers to both the language and operating system developed as part of that project. The Oberon language was later revised and extended to incorporate support for Object Oriented Programming constructs, resulting in the language known as Oberon-2 [Möss93].

This paper describes the implementation of a new compiler based on the Oberon-2 language specification and implemented using the tools in the Purdue Compiler Construction Tool Set: DLG (DFA-based Lexical analyzer Generator) for implementing the lexical rules, ANTLR (ANother Tool for Language Recognition) for implementing the syntactic rules and generating the AST (Abstract Syntax Tree) intermediate form, and SORCERER for implementing the tree parser which generates code from the intermediate AST form. Currently, the Oberon compiler is being built with the PCCTS 1.20 release [PQC94] and the SORCERER 1.20B8 release [PSF94]. The syntax description of the language from which the ANTLR rules were built is based on the EBNF (Extended Backus-Naur Formalism) language description given in [Möss93]. ANTLR and DLG are used in the C-language code generation mode, rather than the C++ mode.

The Oberon-2 compiler described herein has been designed initially for the IBM OS/2 2.X Operating System. It generates native 32-bit code using the 80X86 "Flat" memory

model. However, the code generator is the only processor-dependent phase of the compiler as it is implemented. Due to the portability of PCCTS to a wide variety of platforms, all but the code generation phase can be re-used without change for a wide variety of 32-bit Architectures and Operating Systems.

2. Project Overview

Language Standard:

The Oberon language shares many syntactic similarities to its ancestors, Pascal and Modula-2. Indeed, the first implementation of Oberon was compiled using a subset of Modula-2. At this time, no standards for the language have been formalized by any standardization committee such as ANSI or ISO. In absence of such a standard, the implementation published by Wirth and Gutknecht in [WG92] remains as the reference standard, although minor revisions have been made since the original publication. Some of the more notable additions to Oberon-2 compared to its ancestors include case sensitivity, type extension constructs and typeguards, and the reclassification of some reserved words as "predeclared identifiers".

The addition of case sensitivity to the lexical analysis phase is a useful feature for improving the interoperability of Oberon with other languages, particularly C, which frequently uses only capitalization rules to distinguish between two dissimilar elements, such as a type specifier and a variable. Type extension provides one of the fundamental mechanisms of Object-Oriented Programming (OOP), commonly referred to as *inheritance*, although inheritance alone is not sufficient for a complete OOP implementation. A related construct called a typeguard provides a mechanism for determining the runtime type of a variable prior to performing an operation on the variable. Note that the syntax of the typeguard is the major source of difficulty in parsing an Oberon program, as will be discussed in section 3 of this paper. Finally, the reclassification of some of the formerly reserved words as predeclared identifiers changes the parsing semantics significantly and, as a by-product, opens up the possibility for a few rather unusual user errors in programming, such as the redeclaration of intrinsic typenames.

Lexical and Syntactic Rules:

Since the published Oberon-2 language description is given in EBNF form and ANTLR/DLG also use a form of EBNF as their source language, the initial source file is nearly an exact translation of the language description. For example, the language specification defines a lexical rule for an identifier as:

```
ident = letter {letter|digit}.
```

where a letter is defined as the set of characters a-z and A-Z, and a digit is the set of characters 0-9 and the curly braces denote zero or more occurrences of their contents. It is worth noting here that the language specification and PCCTS differ in the meaning of many of the EBNF delimiters.

The equivalent DLG lexical rule is:

```
#token IDENT "[_a-zA-Z] [_a-zA-Z0-9]*"
```

where the first pair of brackets indicate a single occurrence of the enclosed characters and the second pair of brackets, trailed by an "*" character indicate zero or more occurrences of the enclosed characters. Note that the inclusion of underscores in identifiers is not "Standard" Oberon-2 syntax. Also, in the context of PCCTS lexical rules, the "-" character is a range operator unless it is preceded with a "\" character. Also, the declaration of a reserved word such as:

```
#token Procedure "PROCEDURE"
```

supersedes the definition of an **IDENT**, even though the word **PROCEDURE** matches the lexical specification given for an **IDENT**.

The same generalization regarding interpretation of the lexical rules of the Oberon-2 language applies to the syntactic rules. The EBNF language description of syntactic rules is also a direct translation, except for minor differences in the characters used for punctuation. For example, the syntax of the statement rule in the Oberon-2 specification is given by:

```
statement = [assignment |ProcedureCall | IfStatement |
   CaseStatement | WhileStatement | RepeatStatement |
   LoopStatement | WithStatement  | ForStatement | EXIT |
   RETURN [expression]].
```

where in this case the square brackets indicate an optional occurrence of the contents of the brackets, the "=" begins the rule, and the "." terminates the rule. Note that this allows for the "empty statement" case since the entire rule is enclosed in square brackets. Also, **EXIT** and **RETURN** are capitalized to indicate that they are reserved words (terminals), where all other options are given by additional rules. The equivalent ANTLR rule is given by:

```
statement       :
                { assignment
                | procedureCall
                | ifStatement
                | caseStatement
                | whileStatement
                | repeatStatement
                | forStatement
                | loopStatement
                | withStatement
                | Exit
                | Return { expr }
                }
                ;
```

where the curly braces indicate an optional occurrence of the contents, the colon defines

the start of the rule definition, and the semicolon terminates the rule. Also, as per the ANTLR convention, the **Exit** and **Return** options are capitalized to indicate that they are tokens (terminals), where all other options begin with lower case to indicate that they refer to other rules.

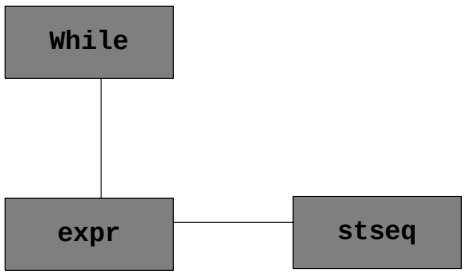
Abstract Syntax Tree Construction:

The intermediate form of the Oberon-2 compiler is a form of Abstract Syntax Tree which can be built semiautomatically by ANTLR as the parsing process proceeds. The structure of the tree is a child-sibling list as introduced in [PDC91] §4.2, wherein the first token in a rule becomes the root node and all subsequent tokens become siblings of the root node. If the child is a subrule, that node becomes a root for a subtree. Under automatic construction, each token becomes a node, each rule becomes a subtree. Trailing a token with a "^" causes it to become a root of a subtree of all tokens that follow, while trailing a token with a "!" causes it to be ignored. An illustrative set of examples follows:

The ANTLR rule for an Oberon-2 WHILE statement is given by:

```
whileStatement  : While^ expr Do! statementSequence End! ;
```

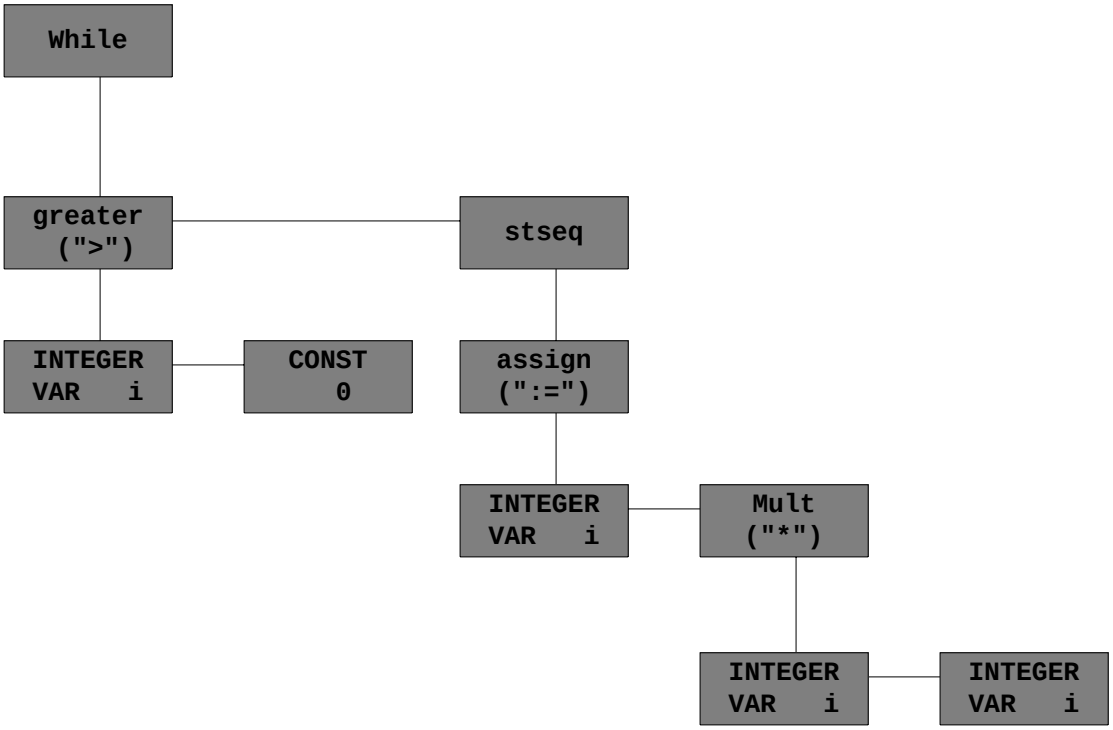
The **DO** and **END** tokens add nothing to the semantics of the statement, serving only as syntactic delimiters for the parsing process, thus they are omitted from the tree. The **WHILE** token serves as the identifying node for the code generation process, thus it becomes the subtree root, with two children as shown:



However, since each of the child nodes corresponds to an additional rule, a complete tree must be expanded to reflect these nodes as well. A detailed description of each of the rules will not be included here, rather an example will be given which indicates the relevant portions of the Oberon-2 source code, followed by the resulting parse tree, which will be based on the While statement shown above:

```
...
VAR
    i : INTEGER;
...
i := 3;
WHILE i > 0 DO
    i := i*i;
END;
...
```

The resulting parse tree is given by:



Where the **expr** node has been replaced with a subtree corresponding to the parsed expression shown above and the **stseq** node has been expanded to include the assignment and the expression which is calculated in the assignment. This subtree can be passed to the code generator and provides all of the information necessary to generate the code corresponding to the source fragment shown above.

3. The Use Of Semantic Predicates

Some constructs used in the Oberon-2 language are inherently ambiguous (i.e. context-sensitive). Consider the following rule for a qualident, which is a constant, variable, or procedure, optionally imported from another module. However, both the module identifier and the name of the element used are indicated by an IDENT token. Note also that the **IDENT Dot IDENT** pattern may occur in a context that includes the qualident, namely the designator rule, which will be covered next.

```
qualident       : IDENT Dot IDENT
                | IDENT
                ;
```

In order to resolve whether the **Dot IDENT** portion of the first option is part of the qualident rule or not, we need to determine whether or not the first **IDENT** represents a module name or any other type of **IDENT**. For this purpose, ANTLR provides a construct known as a semantic predicate, which allows the parser to determine sufficient information from the context of the first **IDENT** to determine how to handle the remainder of the rule. To this end, we define a simple function which "looks ahead" to the first **IDENT**, finds that **IDENT** in the symbol table, and determines whether the **IDENT** represents a module, or anything else. If it is a module, the first alternative will be chosen, otherwise, the second alternative is chosen. The function, which is shown in the second variant of the rule below, is called **IsModID**, and returns a nonzero value if the **IDENT** represents a module name, zero otherwise.

```
qualident       : << isModID(LATEXT(1))>>? IDENT Dot IDENT
                | IDENT
                ;
```

This form of the rule will use symbol table information at runtime to determine which alternative is correct.

A second construct which causes significant problems in parsing involves an Oberon-2 construct known as a *Type Guard*. The purpose of the type guard is to allow for runtime type checking on a variable or procedure to determine if an operation is allowable. This construct occurs in the form of a rule known as a *designator*, given as follows:

```
designator      : qualident
                ( Dot IDENT            /* field of record */
                | Lbrack exprList Rbrack /* array indices */
                | Lparen qualident Rparen   /* type guard */
                | Fwd                    /* pointer deref */
                )*
                ;
```

Where the parentheses trailed by a "*" indicate that the contents may occur zero or more times. Note that the first alternative given here shows the source of the ambiguity in the qualident rule discussed above, since it takes the same form as the optional portion of that rule. However, the qualident rule has already resolved that ambiguity, so the construct which causes problems here is actually the third alternative, shown as a type guard in the comment. The problem occurs because of a rule which may enclose the designator as follows:

```
procedureCall   : designator {actualParams} ;
```

Where the **actualParams** rule may take the form **Lparen qualident Rparen**. In order to decide which path to take, it must be determined whether or not the qualident enclosed in the parentheses represents a *type*. If so, the type guard is processed, otherwise, the designator rule must back up to before the Lparen token and exit. One possible way to perform this task is to rewrite the rule as follows:

```
designator      : qualident
                ( Dot IDENT           /* field of record */
                | Lbrack exprList Rbrack /* array indices */
                | << typeGuard >>? typeGuard
                | Fwd                 /* pointer deref */
                )*
                ;

typeGuard > [int r] :  << AST *node; >>

            Lparen qualident > $node Rparen

            << $r = IsType(node); >>

                  ;
```

There are two major changes to the structure of the new rules. First, the **typeGuard** rule was created and defined as returning an integer value, allowing it to be enclosed in a semantic predicate. The return value of the rule is returned by a function called **IsType** which functions similarly to the **isModID** function discussed previously. Second, the new **typeGuard** rule was enclosed in a semantic predicate, allowing the parser to back up as required if the predicate fails. Note that the **typeGuard** rule will leave additional AST nodes unused when the predicate is exercised, but this should not cause harm for finite-sized programs. Alternatively, on failure of the predicate, additional actions may be taken to free the created node, or the function could change an existing node, rather than creating a new node at each call.

4. Conclusion

In this paper, I have presented portions of a compiler project which uses the Purdue Compiler Construction Tool Set to build a complete compiler for the Oberon-2 language. Examples of lexical rules for DLG and syntactic rules for ANTLR were illustrated as compared to the original language specification. A sample of the parse tree which is constructed as an intermediate form was given to illustrate the tree construction functions of ANTLR. Additionally, fragments of the grammar were included to illustrate the use of semantic predicates in resolving ambiguous parts of the language. The code generation portion of the compiler is still undergoing significant changes and is thus regrettably omitted from this paper.

5. Acknowledgments

I would like to express my sincere thanks to Terence Parr who answered my questions without laughing...

References

[WG92] Niklaus Wirth, Jürg Gutknecht, *Project Oberon*, Addison-Wesley Publishing Company (ACM Press). Reading, Massachusetts, 1992.

[Möss93] Hanspeter Mössenböck, *Object-Oriented Programming in Oberon-2 (English Translation)*, Springer-Verlag, Berlin Heidelberg, 1993.

[PQC94] Terence J. Parr, Russell W. Quong, William E. Cohen, Purdue Compiler Construction Tool Set Version 1.20 Release Notes, Army High Performance Computing Research Center, March 31, 1994.

[PSF94] Terence John Parr, Aaron Sawdey, Gary Funck, The SORCERER Reference Manual Version 1.00B8, Army High Performance Computing Research Center, June, 1994.

[PDC92] T. J. Parr, H. G. Dietz, W. E. Cohen, PCCTS 1.00: The Purdue Compiler Construction Tool Set, *SIGPLAN Notices*, 1992