Notes on Software Design
Part I: Compiler Project

PJA de Villiers

Chapter 1

Introduction

In this course the goal is to develop a simple compiler to illustrate various aspects of software design. The first part of these notes will guide you to design and implement a working compiler. The second part explains how the general approach taken to develop the compiler can be used to develop other kinds of software.

Too often people without formal training labour unnecessarily to devise their own suboptimal solutions to problems without knowing that good solutions exist. It is a much better idea to reuse good solutions over and over again. A good software designer is a person who can almost instinctively divide a large project into smaller parts and immediately identify subproblems for which good solutions are known. Such skills almost seem to guarantee success and therefore good designers are always sought after. If you want to make your mark in the software industry, it should be your goal to become a good designer.

Although few students of Computer Science will write production compilers in industry, the experience of developing a compiler is definitely valuable. The famous Computer Scientist and brilliant programmer Per Brinch Hansen once motivated this as follows: "if you can manage the details of a compiler, if you know how to test it systematically so that it never fails, and if you can write an understandable description of it, you know how to program!" Furthermore, a deeper understanding of the details of compilation and how code is executed by a computer provides the kind of know-how to write better programs in general.

There are many books that cover compiler construction techniques in depth, but since it is not the primary focus of this course, an in-depth study of different compilation techniques is inappropriate. Instead, we shall study a single, practical technique without getting bogged down by theory. A similar approach was taken by Brinch Hansen and Wirth in two excellent books [3, 2] but unfortunately both are out of print. In what follows, some of the material in these books is summarised. The language Oberon is used for all code examples which are adopted without change from [2]. Since Oberon is a highly readable language, these examples should be easy to follow. If needed, a formal definition of the language can be found in [1].

1.1 The programming language Oberon-0

The compiler to be developed will accept programs written in a specific source language and generate machine code for a simple computer. The source language, which is a subset of the Oberon programming language, is called Oberon-0. The language is expressive enough to illustrate several interesting concepts, but simple enough to make the project suitable for a one semester course.

The assignment is the most basic statement in Oberon-0. Composite statements include the concepts of a sequence of statements, conditional and repetitive execution. Oberon-0 also includes procedures with parameters and the concept of a module. The types BOOLEAN and INTEGER are the only elementary data types supported. It is possible to declare integer-valued constants and to construct expressions with the usual arithmetic operators. Comparison of expressions yield values of type BOOLEAN which can be subjected to logical operations. Data structures supported are the array and the record, which may be nested arbitrarily. There are no pointers in Oberon-0. A sequence of statements may be grouped together as a procedure. Oberon-0 offers the possibility to declare identifiers local to a procedure, that is, in such a way that identifiers are visible only within the procedure itself.

The following is a definition of the syntax of Oberon-0:

```
ident = letter {letter | digit}.
integer = digit {digit}.
selector = {"." ident | "[" expression "]"}.
number = integer.
factor = ident selector | number |
   "(" expression ")" | "~" factor.
term = factor {("*"|"DIV"|"MOD"|"&") factor}.
SimpleExpression = ["+" | "-"] term
   {("+" | "-" | "OR") term}.
expression = SimpleExpression
   [("=" | "#" | "<" | "<=" | ">" | ">=")
   SimpleExpression].
assignment = ident selector ":=" expression.
ActualParameters = "(" [expression
    {"," expression}] ")".
ProcedureCall =
   ident selector [ActualParameters].
IfStatement =
   "IF" expression "THEN" StatementSequence
   {"ELSIF" expression "THEN" StatementSequence}
   ["ELSE" StatementSequence] "END".
WhileStatement = "WHILE" expression "DO"
   StatementSequence "END".
statement = [assignment | ProcedureCall |
   IfStatement | WhileStatement].
```

```
StatementSequence = statement {";" statement}.
IdentList = ident {"," ident}.
ArrayType = "ARRAY" expression "OF" type.
FieldList = [IdentList ":" type].
RecordType = "RECORD" FieldList
   {";" FieldList} "END".
type = ident | ArrayType | RecordType.
FPSection = ["VAR"] IdentList ":" type.
FormalParameters = "(" [FPSection
   {";" FPSection}] ")".
ProcedureHeading = "PROCEDURE" ident
    [FormalParameters].
ProcedureBody = declarations
   ["BEGIN" StatementSequence] "END" ident.
ProcedureDeclaration =
   ProcedureHeading ";" ProcedureBody.
declarations =
   ["CONST" {ident "=" expression ";"}]
   ["TYPE" {ident "=" type ";"}]
   ["VAR" {IdentList ":" type ";"}]
   {ProcedureDeclaration ";"}.
module = "MODULE" ident ";" declarations
   ["BEGIN" StatementSequence] "END" ident ".".
```

The following example of Oberon-0 code should give a general idea of the language:

```
MODULE Sample;

PROCEDURE Multiply;
VAR x, y, z: INTEGER;
BEGIN Read(x); Read(y); z := 0;
  WHILE x > 0 DO
    IF x MOD 2 = 1 THEN z := z + y END;
    y := 2*y; x := x DIV 2
  END;
  Write(x); Write(y); Write(z); WriteLn
END Multiply;

PROCEDURE Divide;
VAR x, y, r, q, w: INTEGER;
BEGIN Read(x); Read(y); r := x; q := 0; w := y;
  WHILE w <= r DO w := 2*w END;
  WHILE w > y DO
    q := 2*q; w := w DIV 2;
    IF w <= r THEN r := r-w; q := q+1 END
```

```
  END;
  Write(x); Write(y); Write(q); Write(r); WriteLn

PROCEDURE BinSearch;
VAR i, j, k, n, x: INTEGER;
  a: ARRAY 32 OF INTEGER;
BEGIN Read(n); k := 0;
  WHILE k < n DO Read(a[k]); k := k+1 END;
  Read(x); i := 0; j := n;
  WHILE i < j DO
    k := (i+j) DIV 2;
    IF x < a[k] THEN j := k ELSE i := k+1 END
  END;
  Write(i); Write(j); Write(a[j]); WriteLn
END BinSearch;

END Sample.
```

Chapter 2

A parser for Oberon-0

The compiler has several well-defined tasks. First, terminal symbols have to be recognised by reading the source text one character at a time. This is called lexical analysis and the part of the compiler responsible for this task is called the *scanner*. Next, sentences composed of terminal symbols have to be recognised by the *parser*. If the input does not comply with the grammar rules for Oberon-0, the compiler should produce error messages. Information obtained from declarations is stored in a data structure known as the symbol table. In the past an array or table was used to represent symbols and associated information, but more effective ways are often used nowadays. The symbol table is essential for analysing the input text for type conflicts and during code generation as will be described later.

2.1 The scanner

This part of the compiler must be designed to recognise the terminal symbols of the language as efficiently as possible. This task usually represents more than 50% of the compilation time. Oberon-0 has the following so-called *reserved words* which must be recognised by the scanner and therefore cannot be used as identifiers:

```
DIV MOD OR OF THEN DO
END ELSE ELSIF IF WHILE
ARRAY RECORD CONST TYPE VAR
PROCEDURE BEGIN MODULE
```

The following special symbols must be recognised too:

```
* & + - = # < <= > >= . , : ) ] ( [ ~ := ;
```

Identifiers and numbers are also treated as terminal symbols. For example, although different character strings are obviously used to identify different variable names, the scanner should recognise any suitably formed character string

that is not a reserved word as an identifier. In a similar way, suitably formed strings of numerical characters should be recognised as constants.

It is useful to implement the scanner as a module. In this way it is possible to hide details irrelevant to the rest of the compiler. A well-known approach is to map terminal symbols onto integers. To make the code more readable, constants (e.g., times = 1; mod = 2; colon = 3) should be declared for the different terminal symbols. The main function of the scanner is to locate the next terminal symbol in the input text. A procedure with a name such as `Get`, which is called by the parser, provides this functionality. It is useful to return the terminal symbol via a reference parameter.

In the process of locating terminal symbols, the scanner should do the following:

- Skip blanks and line ends.
- Recognise reserved words such as BEGIN and END.
- Recognise sequences of alphanumeric characters starting with a letter, which are not reserved words, as identifiers. Procedure `Get` should return the symbol `ident` via a parameter and the character string should be assigned to a global variable. It is of course possible to use a second parameter to return the character string, but since this information is only relevant for identifiers it seems reasonable to make a special arrangement for the exceptional case.
- Recognise sequences of digits as numbers. The symbol `number` is returned and the value of the number assigned to a global variable.
- Recognise combinations of special characters such as := and ¡= as symbols.
- Skip comments which are sequences of arbitrary characters beginning with (* and ending with *) in Oberon-0.
- Return the symbol `null` if the scanner reads an illegal character such as $ or %. Return the symbol `eof` if the end of the text is reached. None of these symbols should appear in a well-formed Oberon-0 program.

2.2 The parser

The parser will be developed by strictly following the EBNF definition for Oberon-0. In this way we can ensure that the parser will accept precisely the language as defined. It is not always possible to derive programs directly from formal specifications, but if feasible, this simplifies the task significantly.

To ensure that the parser will always be in a position to select the appropriate production rule without backtracking, the EBNF definition for Oberon-0 must satisfy certain restrictions. To check whether the EBNF is in a suitable form, we have to compute two sets of symbols known as `First(S)` and `Follow(S)`.

The set `First(S)` contains all terminal symbols that can be derived from the non-terminal `S`. The rules for computing `First(S)` are simple:

1. If `S` is empty, `First(S)` = $\emptyset$. In Oberon-0 the empty sentence is expressed by writing noting.
2. If `s` is a single terminal symbol, `First(s)` is the symbol itself. For example, `RecordType` = {`"RECORD"`}.
3. If all sentences of the form `E` are nonempty, all sentences of the form EF must begin with one of the symbols of `E`, so `First(EF)` = `First(E)`. For example, `First(factor{("*" | "DIV" | "MOD" | "&") factor}` = `First(factor)` = {`"("`, `"~"`, `integer`, `ident`}.
4. If some sentences of the form `E` are empty, all sentences of the form `EF` must begin with one of the first symbols of either `E` or `F`, that is, `First(EF)` = `First(E)` ∪ `First(F)`. Thus `First(["TYPE" {ident "=" type ";"}])` = `First(["TYPE"])` ∪ `First({ident})` = {"TYPE", ident}.
5. All sentences of the form `E|F` must begin with one of the first symbols of `E` or `F`, that is, `First(E|F)` = `Fist(E)` ∪ `First(F)`.

To find the set of terminal symbols that may follow after a nonterminal symbol `T`, one must look at all productions of the forms `N = STU`, `N = S[T]U` and `N = S{T}U`. If all sentences of the form `U` are nonempty, `Follow(T)` includes `First(U)`. If some sentences of the form `U` are empty, `Follow(T)` also includes `Follow(N)`. Productions of the forms `N = ST`, `N = S[T]` and `N = S{T}` show that `Follow(T)` includes `Follow(N)`. If `T` occurs in the form `{T}`, `Follow(T)` also includes `First(T)`.

When the parser has reached a point in the source program text where several different kinds of sentences may occur, it must decide which of these possibilities to pursue. To avoid backtracking, one must write the EBNF in such a way that the correct choice among alternative sentences can always be made simply by looking at the next symbol. The decision is trivial if a sentence form begins with a unique symbol. However, it is often the case that the same symbol can occur at the beginning of many different sentences. To deal with this problem, we must be somewhat more clever. We know that each kind of sentence can occur at certain points in the program only. So at each point in the source program text, the parser must choose between a small number of alternatives only. If these alternatives all begin with different symbols, the parser can immediately make the right choice. This leads to the following restriction: for sentences of the form `N = E|F`, `First(E)` ∩ `First(F)` must be empty.

Furthermore, if a sentence of the form `N` may be empty, the parser must be able to decide whether the next symbol is the beginning of a nonempty or an empty sentence of that form. Consequently, `First(N)` ∩ `Follow(N)` must be empty in such a case. To compute the `Follow` sets for all non-terminal symbols it is necessary to look at the right side of each production and apply the rules defined above.

A final restriction on the EBNF is that no left recursion is allowed. Without this restriction, the parser cannot handle non-terminal symbols that are defined in terms of themselves.

Once we know that our EBNF definition of the source language conforms to the rules, it is straightforward to construct a parser. The first rule is to construct a procedure for each non-terminal symbol. It helps to give the procedure the same name as the non-terminal symbol it represents and to include the production rule defining the symbol as a comment. This makes it easy to modify the parser if necessary.

We construct a procedure to parse a rule of the form `A B C` as the algorithm for `A` followed by the algorithm for `B` followed by the algorithm for `C`. When the parser expects a single terminal symbol `s` we use the following algorithm:

```
IF symbol = s THEN Get(symbol) ELSE Error(n) END
```

The parser uses the following algorithm to recognise a possibly empty sentence of the form `[E]`:

```
IF symbol IN First(E) THEN E END
```

A sentence of the form `{E}` is recognised by the following algorithm:

```
WHILE symbol IN First(E) DO E END
```

If all sentences of the forms `A`, `B` and `C` are nonempty, the following algorithm will recognise a sentence of the form `A|B|C`:

```
IF symbol IN First(A) THEN A
ESLIF symbol IN First(B) THEN B
ELSEIF symbol IN First(C) THEN C
ELSE Error(n)
END
```

It is a good idea to draw a diagram to indicate how the different procedures of the parser depend on one another. This helps to define them in the right order if the implementation language does not allow forward references.

2.3 Coping with syntactic errors

Production compilers attempt to find as many syntax errors as possible in each run for maximum productivity. In this project, however, it is acceptable to stop at the first error encountered. To offer some support for debugging, an appropriate error message and the number of the line containing the error must be displayed. This is simple; all that needs to be done is to increment a counter for each end-of-line character encountered.

Chapter 3

Context and declarations

Although context-free parsing methods can be used to check the syntax of most programming languages, the context in which different symbols are used is important. For example, in most programming languages every variable must be declared before it can be used. More specifically, the compilation of an assignment statement $x := x + 1$ is influenced by information contained in the declaration of x that describes its type. This will determine the machine instructions generated to evaluate the expression $x + 1$. Furthermore, in most programming languages it is possible to declare variables that are local to procedures. This means that declarations only influence variables in their *scope*.

3.1 Declarations

The standard approach is to store the information contained in declarations in a special data structure known as a *symbol table*. In the early days of computing it was really a table of symbols, but more advanced data structures are often used today.

We therefore extend the parser for Oberon-0 to store information contained in declarations in a symbol table which is often packaged as an abstract data structure to hide irrelevant information. The requirements are the following:

- Insert a new entry for each declared identifier in each declaration
- Each new item requires a search of the symbol table to determine whether this is a new symbol. If not, this may be an error, depending on the definition of the programming language.

A symbol table does not only contain information about variables and the term *entity* will be used for any item with associated information that is stored. A typical attribute is the *class* of the entity. In the case of Oberon-0 this indicates whether the entity is a constant, a variable, a type or a procedure. Another attribute is the *type* of the entity, if appropriate.

A symbol table can be structured as a linked list. The most important disadvantage of this rather simple approach is that the process of looking up the information of a given entity is slow. On the other hand, if only small programs will be compiled, this is an acceptable approach. Other popular (more advanced) strategies are to use a tree or a hash table.

Here we describe a symbol table that is structured as a linked list. Each entity is a record containing the fields `name` (a character string denoting the name of the entity), class (a numeric code indicating what kind of entity it is, `type` (a pointer to an entry in the symbol table that describes the type of the entity, and `val` (the value of the entity where appropriate).

To generate new entries, we use a procedure NewEntity with explicit parameter `class`, the implied parameter `id` and the result `ent`. The procedure checks whether the new entity is already present. If so, this signifies an error—a symbol that is redefined. A new entry is appended at the end of the list so that the list reflects the order of the declarations in the source text. For programming languages (such as Oberon-0) that support nested declarations (for example nested procedures with local variables), a structured symbol table is needed. The order of declarations must be preserved where it is significant such as for procedure parameters.

3.2 Entries for data types

An important task of a compiler is to check the consistency of types. These checks are based on the type information stored in the symbol table. To support user-defined types, a type is represented as an entry in the symbol table. Even anonymous types can be represented in this way. An entry for a type is represented as follows:

```
Type = POINTER TO TypDesc;
TypDesc = RECORD
  form, ken: INTEGER;
  fields: Entity;
  base: Type
END
```

The attribute `form` differentiates between elementary types (INTEGER, BOOLEAN) and structured types (arrays and records). Further attributes are added as needed for the different forms. Characteristics for arrays are their length (number of elements) and the element type (base). For records, a list that represents the fields is needed. Its elements are of the class `field`.

Exercise

Draw a diagram to show the entries in the symbol table to represent the following declarations:

```
TYPE R = RECORD f, g: INTEGER END;
```

```
VAR x: INTEGER;
  a: ARRAY 10 OF INTEGER;
  r, s: R;
```

3.3 Data representation at run time

So far, the details of the target computer could be ignored. However, as soon as the parser is extended into a full compiler, it becomes necessary to include some particulars about the target computer.

First, we must determine the format in which data are to be represented at run time in memory. Most modern computers represent memory as a sequence of bytes (8-bit cells) that can be addressed individually. Variables are therefore represented with monotonically increasing (or decreasing) addresses. Every computer supports certain elementary data types (integer, floating point) with associated instructions such as integer and floating point addition. These are scaler types which occupy a small number of sequential memory locations. Every type has a *size* and every variable has an *address*. These attributes (`type.size` and `type.adr`) are determined when the compiler processes declarations. The sizes of the elementary types are determined by the target machine architecture. Corresponding entries are generated when the compiler is loaded and initialised. The size of structured types can be computed. For example, the size of an array is the number of elements multiplied by the element size. The address of an element is the sum of the array's address (address of element 0) and the element's index multiplied by the element size.

Note that some values such as the base address of an array and the element size are known at compile time. However, the value of the index is usually unknown at compile time and therefore appropriate machine instructions must be generated to compute index values at run time. In contrast, all information is known for record structures at compiler time.

Absolute addresses are usually unknown at compile time, so all generated addresses must be considered as relative to a common base address which becomes available at run time. The effective of a variable is therefore the sum of the relative address (as determined by the compiler) and the base address.

There is another complication that is caused by the fact that most memories are byte addressable: small numbers of bytes are usually transferred between the processor and memory as *words* (typically 2, 4 or 8 bytes at a time). If allocation occurs strictly in sequential order, it is possible that certain variables may occupy parts of several words. This should be avoided to minimise the number of memory accesses. A simple method of overcoming this problem is to round the address of each variable up (or down) to the next multiple of its size. This is called *alignment*. This rule holds for elementary types. For arrays, the size of the element type must be considered and for records we simply round up to the computer's word size. The advantage of alignment is a considerable improvement in speed. The disadvantage is the loss of some memory bytes which is usually negligible.

3.4 Implementation

To generate the required symbol table entries, the parsing of declarations must be extended as follows:

```
(* "TYPE" ident "=" type *)
IF sym = type THEN
  Get(sym);
  WHILE sym = ident DO
    NewEntity(ent, Typ); Get(sym);
    IF sym = eql THEN Get(sym) ELSE Error(1) END;
    Type1(ent.Type);
    IF sym = semicolon THEN Get(sym) ELSE Error(2) END
  END
END;

(* "VAR" ident {"'" ident} ":" type *)
IF sym = var THEN
  Get(sym);
  WHILE sym = ident DO
    IdentList(Var, first); Type1(tp; ent := first;
    WHILE ent # guard DO
      ent.type := tp; INC(adr.ent.type.size);
      ent.val := -adr; ent := ent.next
    END;
    IF sym = semicolon THEN Get(sym) ELSE Error(2) END
  END
END
```

Note that negative offsets are assigned to variables relative to the base address determined at run time. Procedure `IdentList` processes an identifier list. A new entity record is allocated for the first variable in the list. While the next symbol is a comma, additional entries are allocated for each variable in the list until a colon is encountered.

The recursive procedure Type1, with reference parameter `type` compiles a type declaration. For mboxOberon-0 there are three possibilities: (1) a type identifier can be encountered (one of the standard types INTEGER or BOOLEAN or a user-defined type), (2) an array can be specified or (3) a record can be specified. An if-command is used to distinguish between these three alternatives.

If a type identifier is encountered, the symbol table must be searched to find its entity record. Then the scanner is called to get the next symbol. The type (as specified in the entity record in the symbol table is returned via a reference parameter `type`. The appropriate error handling should be included.

If an array is specified, the scanner is called to scan past the keyword `ARRAY`. Then a number is expected, which is assigned to a local variable. The next

symbol should be the keyword `OF`. Then procedure `Type1` is invoked recursively to return the element type of the array. A new entity record is allocated to store the information associated with the array. A pointer to this record is returned via reference parameter `type` after filling in its fields.

If a record is specified, the scanner is called to scan past the keyword `RECORD` and a new entity record is allocated. Some of its fields like the `form` field which identifies it as a record, can be filled in immediately. We must be careful to open a new scope level to avoid intermixing the fields of the record with the variables in the list. Now procedure `IdentList` is used to handle the fields of the record and procedure `Type1` to handle the types of the fields. The keyword `END` signifies the end of the record and the scope should be closed.

Bibliography

[1] M. Reiser and N. Wirth. *Programming in Oberon: Steps beyond Pascal and Modula.* Addison-Wesley, 1992.

[2] N. Wirth. *Compiler Construction.* Addison-Wesley, 1996.

[3] P Brinch Hansen. *Brinch Hansen On Pascal Compilers.* Prentic-Hall, Inc., 1985.