Project Report
Complier Design Project

Sumit Bhagwani Karanveer Singh Sailesh R Ish Dhand
Padi Ramesh

Group-15, April 16, 2011

Abstract

We have implemented a compiler for the Source Language Oberon-II. The implementation has been done in Python using the Python Lex and Yacc utility, PLY. The target language is MIPS assembly code. This document details the features that have been implemented.

Contents

1 Language Specifications **3**

2 Basic features implemented **3**
- 2.1 Basic Types . . . 3
- 2.2 Complex Types . . . 3
- 2.3 Pointers . . . 3
- 2.4 Expressions . . . 4
  - 2.4.1 Form of Expression . . . 4
  - 2.4.2 Defining the expressions in the Language . . . 4
- 2.5 Constant . . . 4
  - 2.5.1 Integer Constant . . . 4
  - 2.5.2 Real Constant . . . 5
  - 2.5.3 Character Constant . . . 5
  - 2.5.4 String Constant . . . 5
- 2.6 Comments . . . 5
- 2.7 Operators . . . 5
  - 2.7.1 Logical Operators . . . 5
  - 2.7.2 Arithmetic operators . . . 5
- 2.8 Keywords . . . 6

3 Language features **7**
- 3.1 Typechecking . . . 7
- 3.2 Built-in's . . . 7
- 3.3 Control Flow . . . 7
  - 3.3.1 The IF statement . . . 7
  - 3.3.2 The FOR statement . . . 8
  - 3.3.3 The WHILE statement . . . 8
  - 3.3.4 The REPEAT statement . . . 8
- 3.4 Procedures . . . 8
- 3.5 Modules . . . 8
  - 3.5.1 Module Out . . . 8
  - 3.5.2 Module In . . . 9
- 3.6 Code Optimization . . . 9
  - 3.6.1 Data-Flow optimization: Constant folding . . . 9
  - 3.6.2 Data-Flow optimization: Strength Reduction . . . 9

4 Features not implemented **10**

1 Language Specifications

The following are the SIT Languages for our compiler.

Source Language	Oberon-2
Target Language	MIPS Assembly level Instructions
Implementation Language	Python

2 Basic features implemented

2.1 Basic Types

Oberon implements 8 Basic types. Of them we implemented 4 as a part of the Basic set-up. Given in the Second column are the number of bytes be used along with each basic type.

Basic Type	Bytes Allocated	Remarks
Char	4	Is used to implement characters in ASCII
Int	4	Numeric. Integer type. Is exact.
Real	4	Numeric. Stored in Floating Point Representation. Is Inexact.
Boolean	4	Used to Store True/False Value

Note that all four data types occupy 4 bytes of memory. This was done to avoid handling the data mis-alignment.

2.2 Complex Types

We have implemented the following Complex types present in Oberon.

Complex Type	Remarks
Fixed length Single Dimensional Array	Homogeneous and structured data types
Strings	Treated in the .data section of the MIPS code

Note that we did not implement strings as an array of Chars. Instead, it was treated in the *.data* section of the MIPS code. For the purpose of implementation, we saved all the strings that we encountered in the parse stage in a dictionary in Python. During code generation, these strings were saved in the *.data* section of the MIPS code and used from there.

2.3 Pointers

Pointers in Oberon are seen as Data-types used to connect Nodes (Static Structures) to other Nodes.

The Syntax of PointerType is defined as:
PointerType = POINTER TO BaseType
BaseType = qualident| ArrayType| RecordType

Oberon allows only pointers to arrays. We have implemented the same.

2.4 Expressions

For the above data types, we have implemented expressions in the following form:

2.4.1 Form of Expression

The general form of the expression is:

$$T_0 \oplus T_1 \oplus T_2 \oplus \ldots T_n$$

where T_i's are Terms and $\oplus$'s are Add-type operations like plus or minus. A Term consists of consecutive factors like:

$$f_0 \otimes f_1 \otimes f_2 \otimes \ldots f_n$$

where f_i's are Factors and $\otimes$'s is the multiplcation operators

2.4.2 Defining the expressions in the Language

Expression	$\longrightarrow$	One more more [Simple Expressions] separated by [Relations].
Relations	$\longrightarrow$	= | # | <| >| <= | >=| IN | IS.
Simple Expression	$\longrightarrow$	One or More [Term] separated by [Add Operator]
Add Operators	$\longrightarrow$	+| − |OR
Term	$\longrightarrow$	One or more [Factor] separated by [Multiplication Operator]
Multiplication Operator	$\longrightarrow$	* | /| & | DIV | MOD
Factor	$\longrightarrow$	Data Types | Designator | Function Call | (Expression)| Expression.
Function Call	$\longrightarrow$	Designator ([Actual Parameters])
Actual Parameters	$\longrightarrow$	One or More [Expressions] separated by a comma.
Designator	$\longrightarrow$	Identifier

2.5 Constant

2.5.1 Integer Constant

An integer constant consists of a sequence of digits optionally preceded by an optional minus sign. We have implemented only signed integer.

2.5.2 Real Constant

Floating Point Constant A real constant consists of an integer part, a decimal point, a fractional part followed by optional e or E and a signed integer exponent. it has single precisions

2.5.3 Character Constant

A character constant is a sequence of one or more characters enclosed in single quotes such as c. Value of character constants will be the numeric value of the character in the machines character set. Multi-character character constants are not permitted.

2.5.4 String Constant

A string constant is a sequence of characters within double/ single quotes.

2.6 Comments

Comments in Oberon-II are of the following form:
(* Comments come here. *)

2.7 Operators

2.7.1 Logical Operators

These operators apply to BOOLEAN operands and yield a BOOLEAN result. We have been able to implement boolean operators in the case of simple Boolean Expressions

symbol	result
OR	logical disjunction
&	logical conjunction
˜	negation

2.7.2 Arithmetic operators

These operators apply to NUMERIC operands and yield a NEUMERIC result. The following numerical operators have been implemented:

symbol	result
$+$	sum
$-$	difference
$*$	product
/	quotient
DIV	integer quotient
MOD	modulus

2.8 Keywords

The following Keywords have be implemented:

DIV	MOD	OR	OF
THEN	DO	END	ELSE
ELSIF	IF	WHILE	ARRAY
RECORD	CONST	VAR	PROCEDURE
BEGIN	MODULE	INTEGER	DEFINITION
IMPORT	CHAR	BOOLEAN	IN
RETURN	LOOP	TO	INC
UNTIL	ABS	MAX	MIN
SIZE	DEC		

3 Language features

3.1 Typechecking

Strong Typing shall be implemented for the compiler. The Type checking shall be done Statically.
Oberon has the following rules for type-checking and coercion:

- INT op INT gives INT
- REAL op REAL gives REAL
- INT op REAL gives REAL
- REAL op INT gives REAL

Note that in the last two cases, we have employed type-casting of int to real and the value has finally be calculated using the two reals.

3.2 Built-in's

The following Built-in function procedures have been implemented

Procedure	Description
ABS(x)	Absolute value (accepts and returns any numeric type)
MAX(T)	If T is a type, returns the maximum value for that type(INTEGER)
MIN(T)	As MAX but minimum
SIZE(T)	Size of type in bytes (integral type)

Here, x can be any valid numeric expression and T can be either of Integer, Real, Char or Bool.
The following Built in proper procedures are present:

Procedure	Description
DEC(i)	Decrement integer
DEC(i, n)	Subtract n from i
INC(i)	Increment
INC(i, n)	Add n to i

Here, i and n can belong to any numeric type subject to the conditions that i is necessarily a variable and that if i is integer, then n must be an Integer.

3.3 Control Flow

3.3.1 The IF statement

The general form of the IF statement in oberon is:

```
IF $B_1$ THEN $S_1$
    ELSEIF $B_2$ THEN $S_2$
    ...
    IF $B_n$ THEN $S_n$
ELSE S
```

Note that we have not implemented the case statement.

The following loop statements have been implemented:

3.3.2 The FOR statement

Like other languages, Oberon defines the statements for the for loop as:

```
ForStatement =          FOR ID ASSIGN expression TO expression (optional)BY expression DO
                                StatementSequence
                        END
```

3.3.3 The WHILE statement

Like other languages, Oberon defines the statements for the while loop as:

```
WhileStatement =        WHILE expression DO
                                StatementSequence
                        END
```

3.3.4 The REPEAT statement

Oberon defines the statements forthe REPEAT loop as:

```
RepeatStatement =       REPEAT
                            StatementSequence
                        UNTIL expression
```

3.4 Procedures

The Procedure is a statement with a name. The syntax is given by:

```
PROCEDURE <name >
BEGIN
. . .
END <name >
```

3.5 Modules

We have implemented only I/O modules in Oberon. Sequential Input and Output is done by Modules In and Out
These Modules are the equivalent of Streams in Java,

3.5.1 Module Out

In module out, we have implemented the following:
DEFINTION Out;

PROCEDURE Int(VAR i:INTEGER);
PROCEDURE Char(VAR i:INTEGER);
PROCEDURE String(i,n: LONGINT);
PROCEDURE Real(x: REAL; n:Integer)
END Out

3.5.2 Module In

In module In , we have implemented the following:
DEFINTION In;
PROCEDURE Int(VAR i:INTEGER);
PROCEDURE String(i,n: LONGINT);
PROCEDURE Real(x: REAL; n:Integer)
END Out

3.6 Code Optimization

3.6.1 Data-Flow optimization: Constant folding

replacing expressions consisting of constants ($e.g., 3 + 5$) with their final value (8) at compile time, rather than doing the calculation in run-time.

3.6.2 Data-Flow optimization: Strength Reduction

Expensive operations are replaced with equivalent but less expensive operations. The classic example of strength reduction converts “strong” multiplications inside a loop into “weaker” additions something that frequently occurs in array addressing.

4 Features not implemented

- Following are the built-ins present in language, which did not implement: ENTIER(x) , LEN(v, n) , LONG(x) , ODD(x) , SHORT(x)
- Also, the following Built in proper procedures are also not implemented(Procedures involving pointer manipulations): COPY(source, dest) , EXCL(v, x) , NEW(p) , NEW(p, i0, i1, ... in)
- String expressions (String literals are OK, of course).
- More than one dimensional arrays.
- Pointers to RECORDs.
- Arrays of RECORDs.
- Inheritance and polymorphism.
- Support for graphics IO.
- Support for library func