|  |
| --- |
| FROSTBURG STATE UNIVERSITY |
| COSC 470 Compiler Project: System Manual |
| PL/SQL Scripting Language Compiler |
|  |
| **Daniel Herold** |
| **5/17/2014** |

|  |
| --- |
| Version 2 of my compiler project for COSC 470 in the Spring 2014 semester. |

Contents

[List of Figures 2](#_Toc388111176)

[Introduction 4](#_Toc388111177)

[Grammar 4](#_Toc388111178)

[Parser 6](#_Toc388111179)

[Parse Table 6](#_Toc388111180)

[Limitations 7](#_Toc388111181)

[Symbol Table 7](#_Toc388111182)

[Structure 7](#_Toc388111183)

[Hashing and Collisions 7](#_Toc388111184)

[Lexical Analyzer 7](#_Toc388111185)

[Intermediate Code 8](#_Toc388111186)

[Assignment 8](#_Toc388111187)

[Evaluation of Expressions 8](#_Toc388111188)

[Control Statements 8](#_Toc388111189)

[Loops 9](#_Toc388111190)

[Basic Algorithm 10](#_Toc388111191)

[Error Message 11](#_Toc388111192)

[Type Mismatch Exception 11](#_Toc388111193)

[End of File Exception 11](#_Toc388111194)

[Max ID Size Error 11](#_Toc388111195)

[Max INT Size Exception 11](#_Toc388111196)

[Max SMALLINT Size Exception 11](#_Toc388111197)

[Crashes 11](#_Toc388111198)

[Appendix A: Sample Programs i](#_Toc388111199)

[Sample 1 i](#_Toc388111200)

[Sample 2 ii](#_Toc388111201)

[Appendix B: Error Summary ii](#_Toc388111202)

[Type Mismatch Exception ii](#_Toc388111203)

[End of File Exception ii](#_Toc388111204)

[Max ID Size Error ii](#_Toc388111205)

[Max INT Size Exception ii](#_Toc388111206)

[Max SMALLINT Size Exception ii](#_Toc388111207)

[Appendix C: Syntax Diagrams iii](#_Toc388111208)

[Block iii](#_Toc388111209)

[Declarations iii](#_Toc388111210)

[Declare\_rest iii](#_Toc388111211)

[Default iii](#_Toc388111212)

[Data\_type iv](#_Toc388111213)

[Characters iv](#_Toc388111214)

[Size iv](#_Toc388111215)

[Size\_option iv](#_Toc388111216)

[Numbers v](#_Toc388111217)

[Compound\_statement v](#_Toc388111218)

[Optional\_statements v](#_Toc388111219)

[Statement\_list v](#_Toc388111220)

[Statement vi](#_Toc388111221)

[Lefthandside vi](#_Toc388111222)

[Righthandside vii](#_Toc388111223)

[Expression vii](#_Toc388111224)

[Simple\_expression vii](#_Toc388111225)

[Term vii](#_Toc388111226)

[Factor viii](#_Toc388111227)

[Relop ix](#_Toc388111228)

[Addop ix](#_Toc388111229)

[Mulop x](#_Toc388111230)

# List of Figures

[Figure 1: block iii](#_Toc388110960)

[Figure 2: declarations iii](#_Toc388110961)

[Figure 3: declare\_rest iii](#_Toc388110962)

[Figure 4: default iii](#_Toc388110963)

[Figure 5: data\_type iv](#_Toc388110964)

[Figure 6: characters iv](#_Toc388110965)

[Figure 7: size iv](#_Toc388110966)

[Figure 8: size\_option iv](#_Toc388110967)

[Figure 9: numbers v](#_Toc388110968)

[Figure 10: compound\_statement v](#_Toc388110969)

[Figure 11: optional\_statements v](#_Toc388110970)

[Figure 12: statement\_list v](#_Toc388110971)

[Figure 13: statement vi](#_Toc388110972)

[Figure 14: lefthandside vi](#_Toc388110973)

[Figure 15: righthandside vii](#_Toc388110974)

[Figure 16: expression vii](#_Toc388110975)

[Figure 17: simple\_expression vii](#_Toc388110976)

[Figure 18: term vii](#_Toc388110977)

[Figure 19: factor viii](#_Toc388110978)

[Figure 20: relop ix](#_Toc388110979)

[Figure 21: addop ix](#_Toc388110980)

[Figure 22: mulop x](#_Toc388110981)

# Introduction

In order to complete the course COSC 470, Compiler Design and Implementation, we were tasked with creating a compiler for a LALR grammar for a PL/SQL scripting language. I used the Java programming language to implement my compiler. I made use of the following: a lexical analyzer, a semantic analyzer, and an intermediate code generator. The code generator output is sent through the Intermediate Code mini-Assembler, which was written in Java, written by Truman Parks Boyer in June 2003. This program translates mini-assembler instruction files into a file of raw integer quadruples. The output from mini is sent through the mini-Intermediate Code Engine (mICE), also developed by Truman Parks Boyer. mICE runs the code and runs all the commands from the source program.

# Grammar

The Backus-Naur Form of the grammar for this language is as follows:

block ::= declarations compound\_statment **$**

declarations ::= **DECLARE** declare\_rest

|

declare\_rest ::= **id** data\_type default **;** declare\_rest

|

default ::= **:=** righthandside

|

data\_type ::= characters

| numbers

| **BOOLEAN**

characters ::= **CHAR**

size ::= **num** size\_option

|

size\_option ::= **,** **num**

|

numbers ::= **NUMBER** **(** size **)**

| **INT** **(** size **)**

| **SMALLINT** **(**  size **)**

| **POSITIVE** **(** size **)**

compound\_statement ::= **BEGIN** optional\_statements **END** **;**

optional\_statements ::= **NULL** **;**

| statement\_list

statement\_list ::= statement

| statement\_list **;** statement

statement ::= lefthandside

| compound\_statement

| **DBMS\_OUTPUT.PUT\_LINE** **(** **id )**

| **DBMS\_OUTPUT.PUT ( id )**

| **DBMS\_OUTPUT.NEW\_LINE**

| **&id**

| **IF** expression **THEN** statement **END IF**

| **WHILE** expression **LOOP** statement **END LOOP**

lefthandside ::= **id** **:=** righthandside

righthandside ::= expression

| **‘** **c** **‘**

expression ::= simple\_expression

| simple\_expression relop simple\_expression

simple\_expression ::= term

| simple\_expression addop term

term ::= factor

| term mulop factor

factor ::= **id**

| **num**

| **TRUE**

| **FALSE**

| **NULL**

| **NOT** factor

relop ::= **>**

| **>=**

| **=**

| **<=**

| **<**

| **<>**

addop ::= **+**

| **-**

mulop ::= **\***

| **/**

| **MOD**

Reserved words are all uppercase letters. Identifiers must start with a letter and may include letters, digits, and underscores. Identifiers may not be more than 20 characters long. Identifiers are case sensitive.

# Parser

The above grammar is a LALR grammar that uses an LR(0) parser. This means that the parser scans input from left to right, and uses right-most derivation. It does not use look ahead.

## Parse Table

The parse table is implemented as one large 2 dimensional array (i.e. the action and GOTO tables are in the same array). The table has a size of 106 arrays, each with space for 125 actions. The action table takes up the arrays from 0-41, and the GOTO table takes up the spaces 100-122. The other arrays not mentioned are just filled with the error state. Any value of 9999 represents an error state, in which the source file is syntactically incorrect. The number 500 represents an accepting state, and is the default value of all of the options for state 6. If the parser reaches state 6, the source file is syntactically valid.

When a reduction occurs, the parser will hold the most recently seen token while it pops values off the stack and checks them against the values in a grammar table. As long as each item popped off the stack matches the values in the grammar table (the values are matched from the end of each array until the first value in the array is reached, at which point the integer in index 0 is pushed onto the stack), the reduction is valid. This is so multiple reductions may be performed on a single token.

## Limitations

There are no limitations pertaining to the size of the stack for the parser. It will not grow infinitely large. The only limitations on this parser are the limitations imposed by the grammar.

# Symbol Table

The compiler maintains a symbol table for several purposes:

1. Ensure no variables are declared in the execution body of the program
2. Ensure all variables have their declared type
3. Ensure all operations are type-valid

## Structure

The symbol table is an array of Symbols, called **bucket**. By default, it can hold up to 50 symbols. When symbols are declared at the beginning of the program, the identifier for the variable, its type, and its size are stored, if a value is assigned it is also stored with the symbol. Symbols may have their values changed in the body of the program, as long as they are changed to something of the same type.

## Hashing and Collisions

The compiler inserts symbols into the bucket in the next available slot in the array. There is no hash function to place identifiers in the bucket. Attempting to insert a symbol into the symbol table that is full of symbols results in a new bucket being created that has more slots.

# Lexical Analyzer

The lexical analyzer first scans through the source program, and removes all comments; It writes out the program, sans comments, to a file *<name>-stripped.txt*. The lexical analyzer then scans through the stripped file and returns tokens to the parser. Tokens are reserved words, numbers, identifiers, terminals (e.g. ;, \*, /, etc.). Spaces are ignored. If an error is encountered, the line numbering corresponds to the stripped file.

# Intermediate Code

## Assignment

For assignments, the compiler first checks the symbol table and ensures the symbol being assigned has the same data type as the symbol it is being assigned to. If the types are valid, the new value of the symbol is updated in the symbol table, and the following intermediate code will be generated:

**STO value,,destination**

## Evaluation of Expressions

Arithmetic expressions can be a little tricky because the compiler may be dealing with a simple expression (e.g. a + b), or it can be dealing with a compound expression (e.g. a + b + c). This can happen in assignments as well (e.g. a := b + c \* e, or a := b + c). When these are encountered, temporary variables are used to store the intermediate operations. For example, the statement:

**a := 5 \* 3 – 8;**

generates the following intermediate code:

**MUL 5,3,temp1**

**SUB temp1,8,temp2**

**STO temp2,,a**

## Control Statements

Control statements are IF-THEN statements or LOOPs, in which a condition must be met to execute a block of code. When a relational operation is encountered, it does not matter if the relation is part of a loop or an IF statement because the output will be basically the same. The generator creates a jump, based on the relational operator in the condition, which will always jump 2 lines ahead. This jump will store a value that corresponds to an evaluation of TRUE from the expression. If the expression is false, the intermediate code produces an unconditional jump to a point where the truth of the expression is checked. If the expression is true, the code will complete the statements in the block; if the expression is false, the code will jump to the end of the conditional block or loop block. When first creating this intermediate code, the compiler does not know where to jump to, so this jump field must be blank until it can be back-patched.

An example of a conditional block follows:

**IF a > b THEN**

**DBMS\_OUTPUT.PUT\_LINE(a)**

**END IF**

This block of code generates the following intermediate code:

**6 JGT a,b,#9**

**7 STO #0,,t**

**8 JMP ,,#10**

**9 STO #1,,t**

**10 JNE #1,t,\_ 🡨 This last value is left blank to be back-patched later**

**11 SYS #-1,a,**

**12 SYS #0,,**

**13**

Once the expression has been reduced by the parser, we know which line we must jump to in line 10 of the intermediate code. This would be back-patched to become:

**10 JNE #1,t,#13**

## Loops

Loops function almost identically to IF-THEN statements, with the exception of keeping track of where the loop begins to jump back to. We can modify the previous example:

**WHILE a > b LOOP**

**DBMS\_OUTPUT.PUT(a)**

**END LOOP;**

This block generates the following intermediate code:

**6 JGT a,b,#9**

**7 STO #0,,t**

**8 JMP ,,#10**

**9 STO #1,,t**

**10 JNE #1,t,\_ 🡨This is left blank until the compiler knows how many lines to skip**

**11 SYS #-1,,a**

**12 JMP ,,#6 🡨 This is found on the loopback stack, it is where the loop begins**

**13**

A back-patch is needed to fix the unknown code at this point, since the loop is coded, which puts it out to after the loop code:

**10 JNE #1,t,#13**

## Basic Algorithm

To output the values from 0 to 100 on their own lines, the following algorithm would perform this:

**DECLARE**

**a INT(3) := 0;**

**BEGIN**

**WHILE a <= 100 LOOP**

**BEGIN**

**DBMS\_OUTPUT.PUT\_LINE(a);**

**a := a + 1**

**END;**

**END LOOP**

**END;**

**$**

The intermediate code generated by this program would be:

**0 STO #0,,a**

**1 JLE a,#100,#4**

**2 STO #0,,T**

**3 JMP ,,#5**

**4 STO #1,,T**

**5 JNE #1,T,#11**

**6 SYS #-1,a,**

**7 SYS #0,,**

**8 ADD a,#1,temp1**

**9 STO temp1,,a**

**10 JMP ,,#1**

**11 HLT ,,**

## Error Message

While parsing the source program, the compiler will throw the following error messages.

## Type Mismatch Exception

This error is thrown when a variable is not assigned the correct type, or if TRUE is trying to be added to 5.

## End of File Exception

This error is thrown when scanning in the source file and reaching the end of file before seeing the dollar sign ($).

## Max ID Size Error

This error is thrown when an identifier is used that is more than 20 characters long.

## Max INT Size Exception

This error is thrown when an INT has more than 8 digits in it.

## Max SMALLINT Size Exception

This error is thrown when a SMALLINT has more than 4 digits.

# Crashes

In the event of a crash, the compiler will display the error message corresponding to the termination event that threw it. In the case of syntax errors, the compiler will report to the user the line number of the error. This is either the actual line the error is on, or in some cases the line right before the reported line number. If the program does crash, it is because of a syntax error in the source program. Sometimes, the stack may be empty and popping attempted on the empty stack. This is because not enough things were put into it by the scanner for the generator.

# Appendix A: Sample Programs

## Sample 1

DECLARE

a POSITIVE(3) := 200;

b NUMBER(3) := 400;

c CHAR := 'c';

d BOOLEAN := TRUE;

BEGIN

WHILE a < b LOOP

BEGIN

a:= a + 50;

DBMS\_OUTPUT.PUT(a);

DBMS\_OUTPUT.NEW\_LINE;

DBMS\_OUTPUT.PUT\_LINE(d);

b := b - 50;

IF a=150 THEN

d := TRUE

END IF;

DBMS\_OUTPUT.NEW\_LINE

END;

END LOOP;

IF d <> FALSE THEN

DBMS\_OUTPUT.PUT(d)

END IF

END;

$

## Sample 2

DECLARE

a INT(3) := 0;

BEGIN

WHILE a <= 100 LOOP

BEGIN

DBMS\_OUTPUT.PUT\_LINE(a);

a := a + 1

END;

END LOOP

END;

$

# Appendix B: Error Summary

## Type Mismatch Exception

This error is thrown when a variable is not assigned the correct type, or if TRUE is trying to be added to 5.

## End of File Exception

This error is thrown when scanning in the source file and reaching the end of file before seeing the dollar sign ($).

## Max ID Size Error

This error is thrown when an identifier is used that is more than 20 characters long.

## Max INT Size Exception

This error is thrown when an INT has more than 8 digits in it.

## Max SMALLINT Size Exception

This error is thrown when a SMALLINT has more than 4 digits.

# Appendix C: Syntax Diagrams

## Block



Figure 1: block

Block ::= declarations compound\_statment '$'

## Declarations

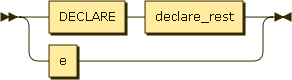


Figure 2: declarations

declarations ::= DECLARE declare\_rest

| e

## Declare\_rest

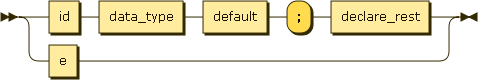


Figure 3: declare\_rest

declare\_rest ::= id data\_type default ';' declare\_rest

| e

## Default



Figure 4: default

default ::= ':=' righthandside

| e

## Data\_type

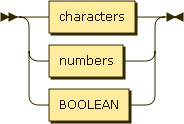


Figure 5: data\_type

data\_type ::= characters

| numbers

| BOOLEAN

## Characters



Figure 6: characters

characters ::= CHAR

## Size

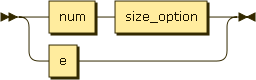


Figure 7: size

size ::= num size\_option

| e

## Size\_option

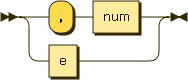


Figure 8: size\_option

size\_option ::= ',' num

| e

## Numbers

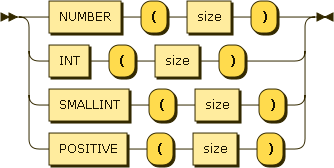


Figure 9: numbers

numbers ::= NUMBER '(' size ')'

| INT '(' size ')'

| SMALLINT '(' size ')'

| POSITIVE '(' size ')'

## Compound\_statement



Figure 10: compound\_statement

compound\_statement ::= BEGIN optional\_statements END ';'

## Optional\_statements

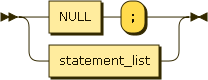


Figure 11: optional\_statements

optional\_statements ::= NULL ';'

| statement\_list

## Statement\_list

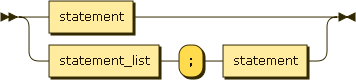


Figure 12: statement\_list

statement\_list ::= statement

| statement\_list ';' statement

## Statement

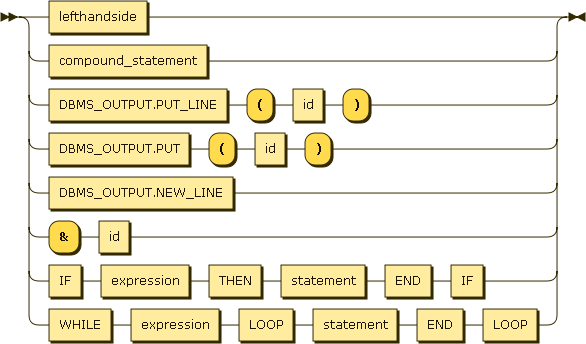


Figure 13: statement

statement ::= lefthandside

| compound\_statement

| DBMS\_OUTPUT.PUT\_LINE '(' id ')'

| DBMS\_OUTPUT.PUT '(' id ')'

| DBMS\_OUTPUT.NEW\_LINE

| '&' id

| IF expression THEN statement END IF

| WHILE expression LOOP statement END LOOP

## Lefthandside



Figure 14: lefthandside

lefthandside ::= id ':=' righthandside

## Righthandside

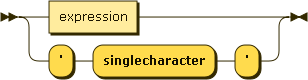


Figure 15: righthandside

righthandside ::= expression

| "'" 'singlecharacter' "'"

## Expression

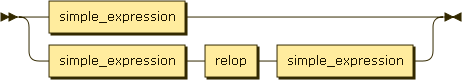


Figure 16: expression

expression ::= simple\_expression

| simple\_expression relop simple\_expression

## Simple\_expression

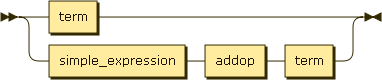


Figure 17: simple\_expression

simple\_expression ::= term

| simple\_expression addop term

## Term

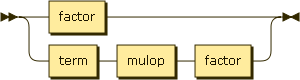


Figure 18: term

term ::= factor

| term mulop factor

## Factor

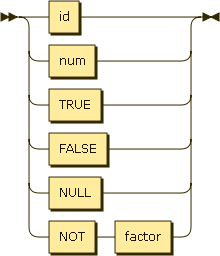


Figure 19: factor

factor ::= id

| num

| TRUE

| FALSE

| NULL

| NOT factor

## Relop

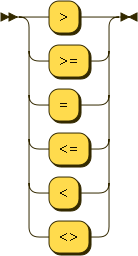


Figure 20: relop

relop ::= '>'

| '>='

| '='

| '<='

| '<'

| '<>'

## Addop

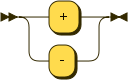


Figure 21: addop

addop ::= '+'

| '-'

## Mulop

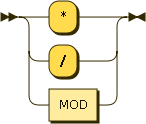


Figure 22: mulop

mulop ::= '\*'

| '/'

| MOD