COSC 470 Final Project

Mini-PL/SQL Compiler

System Manual

Developed by:

Ryan Kasprzyk

Release Candidate 3

Date of Release:

5/14/2012

Contents

[Introduction 3](#_Toc356294718)

[Grammar 3](#_Toc356294719)

[Parser 5](#_Toc356294720)

[Parse Table 5](#_Toc356294721)

[Limitations 6](#_Toc356294722)

[Symbol Table 6](#_Toc356294723)

[Structure and Information 6](#_Toc356294724)

[Hashing and Collision 7](#_Toc356294725)

[Lexical Analyzer and Intermediate Code 7](#_Toc356294726)

[Assignment 7](#_Toc356294727)

[Evaluation of Expressions 7](#_Toc356294728)

[Arithmetic Expressions 7](#_Toc356294729)

[Control Statements 8](#_Toc356294730)

[Loops 9](#_Toc356294731)

[Basic Algorithm 10](#_Toc356294732)

[Error Messages 10](#_Toc356294733)

[Crashes 11](#_Toc356294734)

[Appendix A – Sample Programs i](#_Toc356294735)

[Sample 1 i](#_Toc356294736)

[Sample 2 i](#_Toc356294737)

[Appendix B – Error Codes ii](#_Toc356294738)

[Appendix C – Syntax Diagrams ii](#_Toc356294739)

[Block ii](#_Toc356294740)

[Declarations ii](#_Toc356294741)

[Declare\_rest iii](#_Toc356294742)

[Type iii](#_Toc356294743)

[Default iii](#_Toc356294744)

[Data\_type iii](#_Toc356294745)

[Characters iii](#_Toc356294746)

[Size iii](#_Toc356294747)

[Numbers iv](#_Toc356294748)

[Compund\_statement iv](#_Toc356294749)

[Optional\_statements iv](#_Toc356294750)

[Statement\_list iv](#_Toc356294751)

[Statement v](#_Toc356294752)

[Lefthandside v](#_Toc356294753)

[Righthandside vi](#_Toc356294754)

[Casting vi](#_Toc356294755)

[Simple\_expression vi](#_Toc356294756)

[Term vii](#_Toc356294757)

[Factor vii](#_Toc356294758)

[Relop viii](#_Toc356294759)

[Addop viii](#_Toc356294760)

[Mulop viii](#_Toc356294761)

# Introduction

As the semester project for COSC 470: Compiler Design and Implementation, we have implemented a basic compiler of a small scripting language, based on our understanding of the course material. Using a number of design tools and the class material, we have created a front-end compiler, namely one that translates our programming language into a set of intermediate code that will be executed using a back-end compiler called micE (Mini-Intermediate Code Engine). Parse tables were generated based on the language grammar supplied using GNU-Bison, and open-source version of YACC. This compiler is distributed under the GNU General Public License and may be freely distributed and modified, except for profit. Any section of this compiler may be used with credit.

# Grammar

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

block: declarations compound\_statement \

declarations: DECLARE declare\_rest

|

declare\_rest: identifier type ; declare\_rest

|

type: data\_type default

default: ':=' righthandside

|

data\_type: characters

| numbers

| BOOLEAN

characters: CHAR

| VARCHAR2 size

size: ( num )

numbers: NUMBER 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 ( identifier ) ;

| DBMS\_OUTPUT.PUT ( identifier ) ;

| DBMS\_OUTPUT.NEW\_LINE ;

| & identifier ;

| IF BEGIN expression THEN statement END IF ;

| WHILE expression LOOP statement END LOOP ';'

lefthandside: identifier := righthandside

righthandside: expression

| ’ stringliteral ’

| ; c ’

| casting ( expression )

casting: data\_type

expression: simple\_expression

| simple\_expression relop simple\_expression

simple\_expression: term

| simple\_expression addop term

term: factor

| term mulop factor

factor: identifier

| num

| TRUE

| FALSE

| NULL

| NOT factor

relop: >

| >=

| ==

| <=

| <

| <>

addop: +

| -

mulop: \*

| /

| %

Reserved words are all uppercase letters and numbers, where applicable. Identifiers are must start with a lowercase letter and may include lowercase letters, pound (#), underscore (\_) and dollar sign ($). Identifiers may be no more than 20 characters long.

# Parser

This language uses an LR(0) parser, which is to say a parser that reads from left to right one character at a time with no look-ahead and evaluates the expression from right to left. This parse uses one symbol look-ahead only to verify whether it is seeing a subtraction sign (-) or a single-line comment (--), as well as to verify whether it is seeing division (/) or a multi-line comment (/\*).

## Parse Table

The structure of the parse table is a 112 by 65 entry table with columns 0 through 40 being occupied by terminal values (reserved words, identifiers, etc.) and entries 41 and onward being occupied by stack symbols (expression, lefthandside, etc.). As defined by the input language and the output rules generated by Bison, each entry in the table corresponds to the corresponding action to be taken when the parser is in a given state and sees a given input token or top stack entry. The number 0 represents and accepting state, the positive numbers indicate that the compiler should shift the input token onto the parse stack and shift to the corresponding state, negative numbers indicate the parser should reduce the stack entries based on the corresponding rule, and the value 9999 indicates that an invalid state has been reached and the source is syntactically invalid.

Whenever a reduction is performed, the parser will hold the most recently seen token while it pops and checks stack entries based on the corresponding rule. Having done this, the parser will revert back to the state prior to the last entry popped from the stack, and check the root symbol of the reduction rule against the parse table entry for that state and go to the corresponding state. Upon doing this, the parser will check the original token against the parse table and proceed as directed. This process is recursive for any occurrence where multiple reductions need be performed for a single token.

## Limitations

There are no solid limitations on the parser save those that are explicitly defined by the grammar of the language. The parse stack can theoretically grow without limit as long as the statements being passed in are syntactically correct; although in most cases the number of items on the stack is not likely to exceed 10 parse items with their corresponding states except in cases of multiple-level nesting of statements.

# Symbol Table

For compilation purposes, the compiler maintains a symbol table during the compilation process to ensure the following conditions are met:

1. Ensure that no symbols are declared in the execution body of the program, outside of the declarations section
2. Ensure that no illegal type casting is being done
3. Ensure that all operations are being stored in the appropriate data type

## Structure and Information

The symbol-table is an array of objects of type symbol. Symbols must be declared at the start of the program and be declared with a name, type, and size (where applicable). The symbols may also be declared with a size, though this is not required. Within the body of the program, symbols may have their values modified as needed by the programmer, though no new symbols may be declared outside the body of the declaration.

## Hashing and Collision

The compiler inserts symbols into the symbol table by using the sum of the ASCII values of the identifier modulo the size of the table to determine which index to store the symbol in. Currently the system does not handle collision, instead attempting to avoid collision by using a large array. In order to retrieve and modify symbols, the compiler simply uses the same process on the applicable identifier to find the corresponding symbol.

# Lexical Analyzer and Intermediate Code

## Assignment

When an assignment operation is performed, the compiler will check the symbol table to see if the symbol being assigned value exists and has a corresponding data type. In cases where the value of another variable is being assigned directly, the system will check the type of that variable as well to ensure compatibility. When the statement is successfully process, the new value of the assigned variable will be updated in the symbol table and the following intermediate code will be generated:

**STO value,,dest**

## Evaluation of Expressions

### Arithmetic Expressions

For arithmetic expressions, since we can never be sure whether we are dealing with a simple a+b operation, the system will always generate a series of temp values to be used in later computations of compound statements. When an assignment is finally made, the most recently created temp value will be marked as the value to be stored, and once the storage operation is complete, all temp values will be deleted. In cases where variables are used in arithmetic operations, type checking is used to ensure that variables are compatible with arithmetic operations.

For example, the following operation:

**A:=3\*4+2;**

Will generate the following code:

**MUL 3,4,temp1**

**ADD temp1, 2,temp2**

**STO temp2, ,A**

## Control Statements

Conditional operations are one of the most important controls used in programming. When a relational operation is encountered, whether due to an IF-THEN or a LOOP, the code generated is essentially the same. The generator creates a line of intermediate code corresponding with the relational operator used, with a jump to two lines ahead. This jump will store a value corresponding with a true evaluation of the expression. If the expression is false, the code will fall through to a statement which stores a corresponding value and an unconditional jump to a point where the truth of the expression is checked. If the expression is true, then the code will follow through with the statements enclosed in the condition, but if not, the code will jump to the end of the IF-THEN or LOOP. This issue is that at first the compiler doesn’t know how many line of code to skip, so this jump field must be left blank at first until it can be back-patched.

For example, the line:

**IF BEGIN a>b THEN**

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

**END;**

Generates the following intermediate code:

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

**7 STO #0,,t1**

**8 JMP ,,#10**

**9 STO #1,,t1**

**10 JNE #1,t1,\_ 🡨 This address is left blank until we know how many lines to skip**

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

**12**

Once the expression has been reduced by the parser, we know the next line is our jump point if the condition is false, and we may now back-patch line 10 in our example to read:

**10 JNE #1,t1,#12**

## Loops

Loops function almost identically to IF-THEN statements, with the added portion of having to jump back to the first conditional statement to evaluate it again. Let’s modify our previous example:

**WHILE a>b LOOP**

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

**END LOOP;**

This statement will generate the following:

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

**7 STO #0,,t1**

**8 JMP ,,#10**

**9 STO #1,,t1**

**10 JNE #1,t1,\_ 🡨 This address is left blank until we know how many lines to skip**

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

**12 JMP ,, #6**

**13**

Once again, a back-patch is needed, but this case will be one line further than it’s IF-THEN equivalent to jump past the loopback line:

**10 JNE #1,t1,#13**

## Basic Algorithm

Let’s say we wanted to compute the value of n!, where n is entered by the user. The algorithm would look something like this:

**DECLARE**

**sum NUMBER(4);**

**n NUMBER(3);**

**BEGIN**

**&n;**

**sum:=n;**

**WHILE n>0 LOOP**

**BEGIN**

**n:=n-1;**

**sum:=sum\*n;**

**END;**

**END LOOP;**

**DBMS\_OUTPUT.PUT (sum);**

**END;\**

The corresponding intermediate code would be:

1. **SYS #1,,n**
2. **STO n,,sum**
3. **JGT n,0,#5**
4. **STO #0,,t1**
5. **JMP ,,#6**
6. **STO #1,,t1**
7. **JNE #1,t1,#12**
8. **SUB n,1,temp1**
9. **STO temp1,,n**
10. **MUL sum,n,temp1;**
11. **STO temp1,,sum**
12. **JMP ,,#2**
13. **SYS #-1,,sum**
14. **HLT ,,**

## Error Messages

The following error messages are used by the compiler

**GrammarException** – This error is thrown by the parser when it encounters input that does not match the rules of the grammar.

**TypeMismatchException** – This error is thrown when an attempt is made to assign invalid data to a variable (e.g. assign TRUE to a NUMBER).

**InvalidTokenException** – This error is thrown when the token scanner encounters an undefined reserved word.

**SizeMismatchException** – This error is thrown when attempting to assign too large a value into a variable (e.g. attempting to store the number 999 into a number of size 2).

**UndeclaredIdentifierException** – This error is thrown when an attempt is made to declare a variable outside the declarations section of the code.

**InvalidIdentifierException** – This error is thrown when the token scanner sees an identifier longer than 20 characters or containing illegal characters.

# Crashes

In the event of a crash, the program will present the user with an error message corresponding to the termination event as well as a stack trace to help the user find where the issue is occurring in their code.

# Appendix A – Sample Programs

## Sample 1

DECLARE

a POSITIVE(3):=444;

b NUMBER(3):=222;

c CHAR:='c';

d BOOLEAN:=NOT TRUE;

BEGIN

WHILE d<>TRUE LOOP

BEGIN

a:=2+2\*8;

b:=8;

&a;

DBMS\_OUTPUT.PUT(a);

DBMS\_OUTPUT.NEW\_LINE;

d:=TRUE;

END;

END LOOP;

a:=NUMBER(2)(a);

IF BEGIN a<b THEN

DBMS\_OUTPUT.PUT(a);

END IF;

END;\

## Sample 2

DECLARE

sum NUMBER(4);

n NUMBER(3);

BEGIN

&n;

sum:=n;

WHILE n>0 LOOP

BEGIN

n:=n-1;

sum:=sum\*n;

END;

END LOOP;

DBMS\_OUTPUT.PUT (sum);

END;\

# Appendix B – Error Codes

**GrammarException** – This error is thrown by the parser when it encounters input that does not match the rules of the grammar.

**TypeMismatchException** – This error is thrown when an attempt is made to assign invalid data to a variable (e.g. assign TRUE to a NUMBER).

**InvalidTokenException** – This error is thrown when the token scanner encounters an undefined reserved word.

**SizeMismatchException** – This error is thrown when attempting to assign too large a value into a variable (e.g. attempting to store the number 999 into a number of size 2).

**UndeclaredIdentifierException** – This error is thrown when an attempt is made to declare a variable outside the declarations section of the code.

**InvalidIdentifierException** – This error is thrown when the token scanner sees an identifier longer than 20 characters or containing illegal characters.

# Appendix C – Syntax Diagrams

### Block

block

[block](http://railroad.my28msec.com/rr/ui#block) ::= [declarations](http://railroad.my28msec.com/rr/ui#declarations) [compound\_statement](http://railroad.my28msec.com/rr/ui#compound_statement) '!'

### Declarations

declarations

[declarations](http://railroad.my28msec.com/rr/ui#declarations) ::= [DECLARE](http://railroad.my28msec.com/rr/ui#DECLARE) [declare\_rest](http://railroad.my28msec.com/rr/ui#declare_rest) | e

### Declare\_rest

declare_rest

[declare\_rest](http://railroad.my28msec.com/rr/ui#declare_rest) ::= [identifier](http://railroad.my28msec.com/rr/ui#identifier) [type](http://railroad.my28msec.com/rr/ui#type) ';' [declare\_rest](http://railroad.my28msec.com/rr/ui#declare_rest) | e

### Type

type

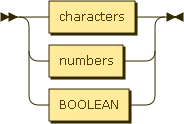
[type](http://railroad.my28msec.com/rr/ui#type) ::= [data\_type](http://railroad.my28msec.com/rr/ui#data_type) [default](http://railroad.my28msec.com/rr/ui#default)

### Default

default

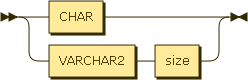
[default](http://railroad.my28msec.com/rr/ui#default) ::= ':=' [righthandside](http://railroad.my28msec.com/rr/ui#righthandside) | e

### Data\_type



[data\_type](http://railroad.my28msec.com/rr/ui#data_type) ::= [characters](http://railroad.my28msec.com/rr/ui#characters) | [numbers](http://railroad.my28msec.com/rr/ui#numbers) | [BOOLEAN](http://railroad.my28msec.com/rr/ui#BOOLEAN)

### Characters



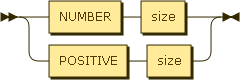
[characters](http://railroad.my28msec.com/rr/ui#characters) ::= [CHAR](http://railroad.my28msec.com/rr/ui#CHAR) | [VARCHAR2](http://railroad.my28msec.com/rr/ui#VARCHAR2) [size](http://railroad.my28msec.com/rr/ui#size)

### Size

size

[size](http://railroad.my28msec.com/rr/ui#size) ::= '(' [num](http://railroad.my28msec.com/rr/ui#num) ')'

### Numbers



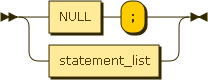
[numbers](http://railroad.my28msec.com/rr/ui#numbers) ::= [NUMBER](http://railroad.my28msec.com/rr/ui#NUMBER) [size](http://railroad.my28msec.com/rr/ui#size) | [POSITIVE](http://railroad.my28msec.com/rr/ui#POSITIVE) [size](http://railroad.my28msec.com/rr/ui#size)

### Compund\_statement

compound_statement

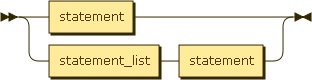
[compound\_statement](http://railroad.my28msec.com/rr/ui#compound_statement) ::= [BEGIN](http://railroad.my28msec.com/rr/ui#BEGIN) [optional\_statements](http://railroad.my28msec.com/rr/ui#optional_statements) [END](http://railroad.my28msec.com/rr/ui#END) ';'

### Optional\_statements



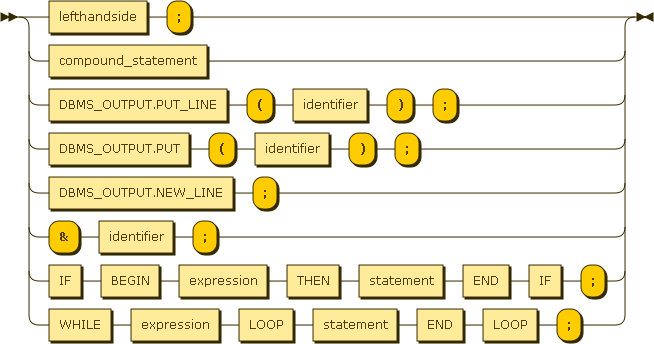
[optional\_statements](http://railroad.my28msec.com/rr/ui#optional_statements) ::= [NULL](http://railroad.my28msec.com/rr/ui#NULL) ';' | [statement\_list](http://railroad.my28msec.com/rr/ui#statement_list)

### Statement\_list



[statement\_list](http://railroad.my28msec.com/rr/ui#statement_list) ::= [statement](http://railroad.my28msec.com/rr/ui#statement) | [statement\_list](http://railroad.my28msec.com/rr/ui#statement_list) [statement](http://railroad.my28msec.com/rr/ui#statement)

### Statement



[statement](http://railroad.my28msec.com/rr/ui#statement) ::= [lefthandside](http://railroad.my28msec.com/rr/ui#lefthandside) ';'

| [compound\_statement](http://railroad.my28msec.com/rr/ui#compound_statement)

| [DBMS\_OUTPUT.PUT\_LINE](http://railroad.my28msec.com/rr/ui#DBMS_OUTPUT.PUT_LINE) '(' [identifier](http://railroad.my28msec.com/rr/ui#identifier) ')' ';'

| [DBMS\_OUTPUT.PUT](http://railroad.my28msec.com/rr/ui#DBMS_OUTPUT.PUT) '(' [identifier](http://railroad.my28msec.com/rr/ui#identifier) ')' ';'

| [DBMS\_OUTPUT.NEW\_LINE](http://railroad.my28msec.com/rr/ui#DBMS_OUTPUT.NEW_LINE) ';'

| '&' [identifier](http://railroad.my28msec.com/rr/ui#identifier) ';'

| [IF](http://railroad.my28msec.com/rr/ui#IF) [BEGIN](http://railroad.my28msec.com/rr/ui#BEGIN) [expression](http://railroad.my28msec.com/rr/ui#expression) [THEN](http://railroad.my28msec.com/rr/ui#THEN) [statement](http://railroad.my28msec.com/rr/ui#statement) [END](http://railroad.my28msec.com/rr/ui#END) [IF](http://railroad.my28msec.com/rr/ui#IF) ';'

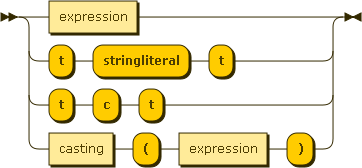
| [WHILE](http://railroad.my28msec.com/rr/ui#WHILE) [expression](http://railroad.my28msec.com/rr/ui#expression) [LOOP](http://railroad.my28msec.com/rr/ui#LOOP) [statement](http://railroad.my28msec.com/rr/ui#statement) [END](http://railroad.my28msec.com/rr/ui#END) [LOOP](http://railroad.my28msec.com/rr/ui#LOOP) ';'

### Lefthandside

lefthandside

[lefthandside](http://railroad.my28msec.com/rr/ui#lefthandside) ::= [identifier](http://railroad.my28msec.com/rr/ui#identifier) ':=' [righthandside](http://railroad.my28msec.com/rr/ui#righthandside)

### Righthandside



[righthandside](http://railroad.my28msec.com/rr/ui#righthandside) ::= [expression](http://railroad.my28msec.com/rr/ui#expression)

| 't' 'stringliteral' 't'

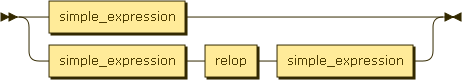
| 't' 'c' 't'

| [casting](http://railroad.my28msec.com/rr/ui#casting) '(' [expression](http://railroad.my28msec.com/rr/ui#expression) ')'

### Casting

casting

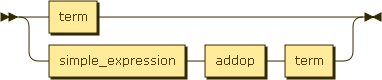
[casting](http://railroad.my28msec.com/rr/ui#casting) ::= [data\_type](http://railroad.my28msec.com/rr/ui#data_type)



[expression](http://railroad.my28msec.com/rr/ui#expression) ::= [simple\_expression](http://railroad.my28msec.com/rr/ui#simple_expression)

| [simple\_expression](http://railroad.my28msec.com/rr/ui#simple_expression) [relop](http://railroad.my28msec.com/rr/ui#relop) [simple\_expression](http://railroad.my28msec.com/rr/ui#simple_expression)

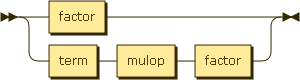
### Simple\_expression



[simple\_expression](http://railroad.my28msec.com/rr/ui#simple_expression) ::= [term](http://railroad.my28msec.com/rr/ui#term)

| [simple\_expression](http://railroad.my28msec.com/rr/ui#simple_expression) [addop](http://railroad.my28msec.com/rr/ui#addop) [term](http://railroad.my28msec.com/rr/ui#term)

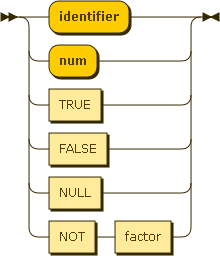
### Term



[term](http://railroad.my28msec.com/rr/ui#term) ::= [factor](http://railroad.my28msec.com/rr/ui#factor)

| [term](http://railroad.my28msec.com/rr/ui#term) [mulop](http://railroad.my28msec.com/rr/ui#mulop) [factor](http://railroad.my28msec.com/rr/ui#factor)

### Factor



[factor](http://railroad.my28msec.com/rr/ui#factor) ::= 'identifier'

| 'num'

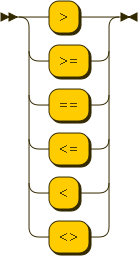
| [TRUE](http://railroad.my28msec.com/rr/ui#TRUE)

| [FALSE](http://railroad.my28msec.com/rr/ui#FALSE)

| [NULL](http://railroad.my28msec.com/rr/ui#NULL)

| [NOT](http://railroad.my28msec.com/rr/ui#NOT) [factor](http://railroad.my28msec.com/rr/ui#factor)

### Relop



[relop](http://railroad.my28msec.com/rr/ui#relop) ::= '>'

| '>='

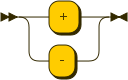
| '=='

| '<='

| '<'

| '<>'

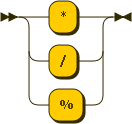
### Addop



[addop](http://railroad.my28msec.com/rr/ui#addop) ::= '+'

| '-'

### Mulop



[mulop](http://railroad.my28msec.com/rr/ui#mulop) ::= '\*'

| '/'

| '%'