**System Manual**

**William Briggs**

**Spring 2015**

**Grammar**

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

| **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** **(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**

**| FULSE**

**| NULL**

| **NOT** factor

relop **>**

| **>=**

| **==**

|**<=**

| **<**

| **<>**

addop **+**

| **-**

mulop \*

| **/**

| %

Parser

Using Bison, a parser generator that converts annotated context-free grammar into a parse table, I was able to use this as my foundation of the compiler. I ran bison, and passed it the grammar we were given and it produced the states, shifts, reduction rules, grammar rules, terminals, and nonterminals. Using this information I was able to directly implement the rules that our compiler needed to accept the given grammar.

Symbol Table

The symbol table was built as a 2D array that held a Symbol Object. This symbol object holds the name, type, numerical value corresponding, value, id, and size of the given symbol. Each symbol that we come across is stored in the 2D array at the first available spot.

ParseTable

The Parse table was implemented as a 2D array, given the rules we obtained using Bison. The parse table is initialized with 9999(error) in every available spot, then re-written using the correct rules (shift/reduction) rules. I also implemented a fill Row Method, which will fill a row with a negative number(reduction rule) that I hard coded in, given to us by Bison.

Grammar Table

The Grammar Table is a 2D array filled with our states and transitions. The first element in our array spots, will give us the size of elements in that row. Due to how stacks are dealt with, we had to insert these grammar rules in backwards. This will allow us to pop the last element off(first) allowing the parser to utilize the stack.

Lexical Analyzer

The Scanner will go through each line, producing a stream of tokens. These tokens will then be used by the Semantic analyzer. Each line that is scanned in will be stored as a character array, then added to a larger list. The comments will be ignored when scanning through. A string list will be scanned through and tokenized, with information determining if it is a reserved word, identifier, or constant. The scanner then takes the stream and builds the symbol table, then uses this to start parsing

Code Generator

The code generator holds a 2D array of intermediate code. Once the parser finishes, the code generator will use the 2D array and produce a text file that can be used by mini and mice.