|  |
| --- |
| System Manual |
| T[] Programming Language |
| Version 0.1 |

|  |
| --- |
| Travis O’Donnell  5/15/2014 |

Table of Contents

[List of Tables 2](#_Toc387826386)

[Grammar 3](#_Toc387826387)

[Operators 4](#_Toc387826388)

[Parser 4](#_Toc387826389)

[Symbol Table 5](#_Toc387826390)

[Lexical Analyzer and Intermediate Code 5](#_Toc387826391)

[Assignment Statement 6](#_Toc387826392)

[Evaluating Expressions 7](#_Toc387826393)

[Control Statements 7](#_Toc387826394)

[Loops 8](#_Toc387826395)

[Appendix 9](#_Toc387826396)

[Summary of Errors 9](#_Toc387826397)

[Program Examples 10](#_Toc387826398)

[Summary of Syntax 11](#_Toc387826399)

# List of Tables

[1: Reserved Words 3](#_Toc387826400)

[2 Intermediate Code - OPCODE 6](#_Toc387826401)

[3 Examples: Declaring Variables 6](#_Toc387826402)

[4 Examples: Assigning Variables 6](#_Toc387826403)

[5 Examples: Expression Evaluations 7](#_Toc387826404)

[6 Example: Control Statement 7](#_Toc387826405)

# Grammar

The T[] programming language is a LALR grammar based off of a PL/SQL scripting programming language. The following rules are to be used when using the programming language:

* All reserved words are to be uppercase

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **DECLARE** | **BOOLEAN** | **CHAR** | **NUMBER** | **INT** |
| **SMALLINT** | **POSITIVE** | **BEGIN** | **END** | **NULL** |
| **DBMS\_OUTPUT.PUT\_LINE** | **DBMS\_OUTPUT.PUT** | **DBMS\_OUTPUT.NEW\_LINE** | **IF** | **THEN** |
| **WHILE** | **LOOP** | **TRUE** | **FALSE** | **NOT** |

1: Reserved Words

* Identifiers
  + Case sensitive
  + Maximum character length of an identifier is 20
  + String of letters, digits, and underscores starting with a letter
  + All identifiers must be declared in the DECLARATION section
* Data types
  + Boolean
    - **TRUE**
    - **FALSE**
    - **NULL**
  + **CHAR**
    - All characters are to be wrapped in single quotation marks
  + **INT**
    - A number within in the range of (-99999999,99999999)
  + **SMALLINT**
    - A number within the range of (-9999,9999)
  + **POSITIVE**
    - A number within the range of (0,99999999)
* Operators

**<** | **>** | **<=** | **<=** | **<>** | **+** | **-** | **\*** | **/** | **MOD**

* A size may be declared in parenthesis while declaring a number to specify the maximum number of digits
* If no size is to be set, the grammar still needs the parenthesis with a space between them
* Blocks are started with BEGIN and ends with END
* Outputs
  + **DBMS\_OUTPUT.PUT\_LINE** ( id )
    - Print id and new line
  + **DBMS\_OUTPUT.PUT** ( id )
    - Print id
  + **DBMS\_OUTPUT.NEW\_LINE**
    - Print new line
* Input
  + User input is specified by ‘& id’ where id will where input is stored
  + The compiler will automatically check to make sure data types match up
* Program ends with the $ symbol
* Every statement ends with a semicolon, except for the last sentence. An exception is, the NULL statement always uses a semicolon to terminate the line
* Comments
  + Single line comment
    - --This is a single line comment
  + Multiple line comment
    - /\* This

is a multiple line comment \*/

# Operators

## Parser

Like most high level programming languages, the T[] programming language uses a Look-A-Head LR (1) parsing technique. The parser is also commonly referred to as the driver of the compiler. There are a total of four classes that make up the structure of the parser.

The first class, GrammarTable, is a simple two-dimensional array of the given grammar in the form of token identifiers. Rather than using full length words to denote the representation of the grammar, integer identifiers start at 1 and end at 67. For more information on identifiers, see Appendix.

Another class that helps the parser run is the parse table itself. The parse table was helped generated by using an external program known as Bison. Bison takes the input grammar and generates a parser for the language. The output of the Bison file is then used to create a two-dimensional array in the ParseTable class. The basic rules of the parser table are as follows: 0 is an accepting value, a positive number tells the parser which state to go to next, and a negative number tells the parser which rule to reduce by. Each reduction rule is a column in the GrammarTable’s two-dimensional array.

The QuadArray class is a basic data structure of a two-dimensional array. The general purpose of the class is to keep track of the outputs for the intermediate code. The two-dimensional array starts at a size of 4 columns by 10 rows. Each time the array becomes full, the rows double in length. The structure of the array is “| line number | OPCODE | operand1 | operand2 | operand3 |”. When the parser is finally finished parsing, a method called getOutput() iterates through each row of the array generating lines in the form of “line#, OPCODE, operand1, operand2, operand3”.

Finally, the Parser class uses all the prior classes mentioned in this section to run. The parser relies on three things: an integer named action, tokens from the scanner and stacks to keep track of states and tokens. By default state 0 is pushed on top of the stacks and action is set to 0. Once the parser is ready, it will request the next token from the Scanner, based on the token identification value and what state we are currently in, certain actions get performed. To decide which action is to be performed, the parse table is checked. As mentioned before, the states are the rows of the two-dimensional array and the token identification numbers are the columns. There are four possible outcomes from the parse table. First, if a 0 is returned, then we know the parser is in an accepting state and can generate output and quit. Another possible outcome is the number 999, which represents an error within the syntax error. When the number 999 gets returned, error #6 gets returned and the program exits. A positive number, besides 999, tells the parser to “push and shift.” When this happens, the parser pushes the current token followed by the current state. To shift, the parser requests the next token from the scanner. Finally, a negative number sends the parser into a reduction stage. The negative number represents which reduction state is to be used. The parser pops tokens off the stacks and matches the id’s with the id’s found in the grammar table. If there are no conflicts, the final value left on the grammar table is our new token identification value which gets pushed onto the stack.

## Symbol Table

A symbol table is a way to store tokens used for parsing, namely variables. There are many ways to accomplish setting up a symbol table. The most efficient way we found was to use Java’s data structure HashMap. The key of the HashMap is the variable name, and the retrieval value is a token. By using the built in data structure, rather than writing our own, collisions are already set up and dealt with in a way that saves time and cost.

## Lexical Analyzer and Intermediate Code

Each time the parser enters the reduction stage, it is generating some type of intermediate code. The intermediate code is best described as an Assembly Language, which is the lowest level programming language, besides machine code. Some of the reduction rules, such as rules 48-58, are helping the parser understand for future parsing reductions. Whereas rule number 4 is taking fully reduced tokens and generating intermediate code for storing a value into a certain variable.

The following rules of the intermediate code were given via a handout called ‘Assembly Code Instruction’ by Professor Chitsaz:

1. Instructions are in the form of ‘line# OPCODE operand1, operand2, operand3’
2. OPCODE are case insensitive

|  |  |  |  |
| --- | --- | --- | --- |
| NOP | ADD | SUB | MUL |
| DIV | MOD | INC | DEC |
| STO | JMP | JEQ | JNE |
| JLT | JLE | JGT | JGE |
| SYS | AND | OR | XOR |
| NOT | HLT |  |  |

2 Intermediate Code - OPCODE

1. A space is mandatory between the OPCODE and the first operand
2. Any other white space is ignored
3. Operands must be separated with commas
   1. The instructions must contain two commas, even if less operands are needed
      1. I.G. ’20 JMP , , #10’
4. Operands are in the form of [#]digitString
5. The # is used to indicate the operand is immediate data
6. If the data is directly addressed, no # is needed (default)
7. The digitString is a string of decimal digits that represent an integer quantity
   1. Can either begin with + or –
8. Data memory addresses are to be an address of an integer, not a byte address
9. Destinations of jump instructions are the actual instruction number of the instruction to jump to

### Assignment Statement

As stated above, all variables must first be declared in the DECLARE section of the code. Any variable attempted to be declared after that point will result in the program crashing. Here are a few examples of assignment statements as well with its intermediate code:

|  |  |
| --- | --- |
| **T[] programming Language Code** | **Generated Intermediate Code** |
| myInt INT ( ) := 5; //create and assign variable | 0 STO #5, ,0 //Store digit 5 into address 0 |
| myTrue BOOLEAN := TRUE;  myFalse BOOLEAN := FALSE;  myNull BOOLEAN := NULL; | 0 STO #1, 0 //Store ‘TRUE’ (1), in address 0  1 STO #0, , 1 //Store ‘FALSE’ (0) in address 1  2 STO #-1, , 2 //Store ‘NULL’ (-1) in address 2 |
| myChar CHAR; | 0 STO #32, ,0 //Store space by default in address 0 |

3 Examples: Declaring Variables

|  |  |
| --- | --- |
| **T[] programming Language Code** | **Generated Intermediate Code** |
| myInt := 5; | # STO #5, ,myInt //myInt will be an address location |
| myBool := TRUE; | # STO #1, , myBool //myBool will be an address location |
| myChar := ‘a’; | # STO #97, , myChar |

4 Examples: Assigning Variables

### Evaluating Expressions

Like other programming languages, operator precedence plays a big part in getting the correct solution from a given algorithm. The operator precedence used in this language is as follows:

1. \* / % Multiplicative
2. + - Additive
3. < > <= Relative
4. == Equality

|  |  |
| --- | --- |
| **T[] programming Language Code** | **Generated Intermediate Code** |
| myInt INT( 5 ) := 5;  myNum INT ( ) := myInt + 1; | 0 STO #5, , 0 //Store digit 5 to address 0  1 ADD #1, 0, 1 //Add digit 1 with myInt, store at 1  2 STO 1, , 2 //Store added terms to address 2 |
| myInt INT ( 2 ) := 2 + 3 | 0 ADD #2, #3, 0 //Add digit 2 and 3, store address 0  1 STO 0, , 1 //Store address 0 into 1  \*\*Note that when myInt is called in program, address 1 will be used |
| myInt INT( ) := 1 + 5 \* 4 + 2; | 0 MUL #4, #5, 0  1 ADD #1, 0, 1  2 ADD #2, 1, 2  3 STO 2, , 3 |

5 Examples: Expression Evaluations

### Control Statements

The simple format of a control statement is

IF expression THEN

Statement

END IF

|  |  |
| --- | --- |
| **T[] programming Language Code** | **Generated Intermediate Code** |
| IF 5 > 3 THEN  DBMS\_OUTPUT.NEW\_LINE  END IF | 0 JGT #5, #3, #3  1 STO #0, , 1  2 JMP , , #4  3 STO #1, , 1  4 JNE 1, #1, #6  5 SYS #0, ,  6 … |

6 Example: Control Statement

### Loops

The T[] programming language can run very basic loops. The grammar given to the class may have not been set up correctly to run more advanced loops. One huge problem is that the user cannot update a variable inside the while loop. So since a while loop runs based off of an expression, the expression is always going to be true or false. If the expression is true, the while loop runs indefinite. The format of a while loop is structured as

WHILE expression LOOP

Statement

END LOOP

|  |  |
| --- | --- |
| **T[] programming Language Code** | **Generated Intermediate Code** |
| WHILE myNum > 3 LOOP  DBMS\_OUTPUT.PUT ( myNum );  END LOOP | 0 JGT #5, #4, #4  1 STO #0, , 1  2 JMP , , #4  3 STO #1, , 1  4 JNE 1, #1, #7  5 sys #0, ,  6 jmp , , #0  7 … |

# Appendix

## Summary of Errors

|  |  |
| --- | --- |
| Could not read file | Compiler could not open specified input file |
| Unexpected end of file | Compiler was not finished parsing before it hit the end of the file |
| Unexpected token | While parser is reducing, an incorrect token was found. Error will print out the token id of the token found and the expected token id. |
| Type Error | User is trying to perform arithmetic operations to a non-number data type |
| Token name already found in symbol table | Variable name was already used |
| Mismatch data types | User is trying to assign a wrong data type |
| Unknown token found | Variable was never initialized in DECLARE |
| Internal Compiler Error | Bad syntax |

## Program Examples

|  |  |
| --- | --- |
| BEGIN  NULL;  END;  $ | 0 hlt , , |
| DECLARE  myInt INT ( ) := 8;  BEGIN  DBMS\_OUTPUT.PUT\_LINE ( myInt )  END;  $ | 0 sto #8, , 0  1 sys #-1, 0,  2 sys #0, ,  3 hlt , , |
| DECLARE  myBool BOOLEAN := TRUE;  myOut SMALLINT ( ) := 20;  BEGIN  WHILE myOut > 3 LOOP  IF myOut > 3 THEN  myOut := myOut - 1  END IF  END LOOP  END;  $ | 0 sto #20, , 0  1 sto #1, , 1  2 JGT 0, #3, #5  3 sto #0, , 2  4 jmp , , #6  5 sto #1, , 2  6 jne 2, #1, #15  7 JGT 0, #3, #10  8 sto #0, , 3  9 jmp , , #11  10 sto #1, , 3  11 jne 3, #1, #14  12 sub 0, #1, 4  13 sto 4, , 0  14 jmp , , #2  15 hlt , , |
| DECLARE  myInt INT ( ) := 8;  BEGIN  IF 5 <= 3 THEN  myInt := 3  END IF  END;  $ | 0 sto #8, , 0  1 JLE #5, #3, #4  2 sto #0, , 1  3 jmp , , #5  4 sto #1, , 1  5 jne 1, #1, #7  6 sto #3, , 0  7 hlt , , |
| BEGIN  IF 5 <= 3 THEN  DBMS\_OUTPUT.NEW\_LINE  END IF  END;  $ | 0 JLE #5, #3, #3  1 sto #0, , 0  2 jmp , , #4  3 sto #1, , 0  4 jne 0, #1, #6  5 sys #0, ,  6 hlt , , |

## Summary of Syntax

|  |  |
| --- | --- |
| block | declarations compound\_statement $ |
| declarations | DECLARE declare\_rest  λ |
| declare\_rest | id data\_type default ; declare\_rest |
| default | := righthandside  λ |
| data\_type | characters  numbers  BOOLEAN |
| characters | CHAR |
| size | num size\_option  λ |
| size\_option | , num  λ |
| 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 |