Technical Documentation of Compiler Construction (Final)

1 Lexical Analysis

In computer science, lexical analysis is an essential part of designing a compiler. Lexical analyzer which is also known as scanner is the first phase of compiler design. It is the process of converting a sequence of characters into a sequence of tokens. In this process, the stream of characters making up the source program is read from left-to-right and grouped into tokens. In this project, we have developed a scanner from scratch for the programming language PL. The scanner continues to read input stream until it reaches to the end of file and group the input characters into lexical units called terminal symbols. The scanner also ignores the input characters such as spaces and blank as well as comments in the source program. All the token are stored in a symbol-table which will be used later by the parser.

For the programming language PL, we have the following terminal symbols-

- **Keywords:** Word symbols with special meaning.
- **Names:** User defined identifiers.
- **Numerals:** Any sequence of decimal digits.
- **Comments:** Starts with the character $ and goes to the end of the line.
- **Special Symbols:** . ; [ ] ( ) + - * / := etc

For building a scanner we have divided our project into 4 parts. They are as follows:

1. **Driver**
2. **Administrator**
3. **Scanner**
4. **Symbol table for storing word-tokens**

1.1 Scanner Construction Details:

Description about the parts mentioned above is given below:

1.1.1 Driver

Associated file: plc.cc

The main function of the program is written in the file named **plc.cc**. The users are allowed to call the scanner from command line. When the user runs the program, it calls the main function written in **plc.cc** file.

1.1.2 Administrator

Associated files: administration.h, administration.cc

This part performs the tasks that are not directly related to compiler phases. **administration.h** contains the the declaration of the private and public variables and methods of **Administration** class. The implementation of the class is written in the **administration.cc** file. The member functions of this class are:

1. **Administration(ifstream &, ofstream &, Scanner &):** This function is used to set up the input and output files for scanning process.
2. **bool validTok(Symbol):** This function returns true, if the token is valid; otherwise, the function returns false for invalid tokens.
3. **int scan():** From **scan()** function, the method **getToken()** of **Scanner** class is called until it reaches to the end of file or it reaches to the maximum number of errors. In our scanner design, we set the maximum number of errors before the compiler bails out is 20. In each iteration, the following three scenarios may occur:
   - **A valid token:** In this case, the attributes of the token are stored in object named **outputfile** using the **validTok(Symbol)** method.
   - **New line :** If we get a token for newline we don't store it, we increase the line counter which is needed for the purpose of scanner. We call the **NewLine()** to perform the task.
   - **An invalid token :** When we get any invalid token, we increase the number of error (**errorCount**) found in the program.

1.1.3 Scanner

Associated files: symbol.h, token.h, token.cc, scanner.h, scanner.cc

A. Defining the symbols for keywords and Identifiers

Associated files: symbol.h

symbol.h file contains the list of symbol-name for tokens. We define the symbols using ***enum*** Symbol type. An integer number is assigned for every symbol and is stored this number rather than it's name in the string for every symbol type. We started the symbol value from 256 to avoid conflict with ASCII characters. The operators and special characters are defined here. The reserve words or keywords for PL are also defined here and the symbol values of the keywords start from 286.

The symbols are defines as the following way:

ID=256, NUMERAL, BADNUMERAL, BADNAME, //256-259
BADSYMBOL, BADCHAR, NEWLINE, NONAME, //260-263
ENDOFFILE, DOT, COMMA, SEMICOLON, //264-267

LEFTBRACKET,RIGHTBRACKET, AND, OR, //268-271
NOT,LESST, EQUAL, GREATERT, //272-275
PLUS, MINUS, TIMES, DIV, //276-279
MOD, LEFTP, RIGHTP, ASSIGN, //280-283
GC1,GC2, //284-285

The symbols for the keywords are defines as the following way:

BEGIN, END, CONST, ARRAY, //286-289
INT, BOOL, PROC, SKIP, //290-293
READ, WRITE, CALL, IF, //294-297
DO, FI, OD, FALSE, TRUE //298-302

B. Token class

Associated files: token.h and token.cc

token.h is the interface of **Token** class. The data members and member functions of the **token.cc** class are defined in this file. The implementation of these functions are written in **token.cc** file.

In order to store the attribute's value and lexeme, we have designed a structure named **attVal** as a private member of **Token** class. Class **Token** has data members of type **Symbol (sname)** and type **attVal (sval)**.

The constructor functions and three get functions are defined as the public member function of the **Token** class.

1. **Constructor Functions**
   - **Token() :** At the time of creating any object of **Token** class, it creates a token of **NONAME** symbol type using it's default constructor function **Token()**.
   - **Token(Symbol, int, string):** If we want to create a token of any specific symbol then the constructor overloading is performed and Symbol, value and lexeme for that token are stored using this function.
2. **Get Functions**
   - **Symbol getSymbol():** This function is used o get the token symbol.
   - **int getValue():** In order to get the numeral value of Token, this function is used.
   - **string getLexeme():** This function is used to get the lexeme value.

C. Scanner class

Associated files:scanner.h, scanner.cc

This is one of the most important parts of the lexical analyzer. **scanner.h** is interface of **Scanner** class and we implemented the methods inside the **scanner.cc** file.

The member functions of this class are:

1. **Scanner(ifstream &,Symboltable):** This is the constructor function to initialize a scanner object with input stream and symbol table.

2. **Token getToken():** This is the most important function of this class. It scans up to the next sequence of characters and tries to recognize whether it matches with some defined pattern or not. If a pattern is recognized, then the token is constructed. Let us explain this with two case scenarios.

   - Suppose, our program has read a digit from the input file. So it enters inside "**else if(isdigit(ch))**"code block. Inside this code block, we call the function **recognizeNumeral()**. It continues to read digits until it reaches to a valid or invalid ending. For that reason, our compiler looks one character ahead while reading the input file with the help of **peek()** method. When we reach to an end of a digit we check either the numeral is valid or not. If it is valid, then we construct a token object of **NUMERAL** type with it's associated value. The lexeme is set to be an empty string as this is not needed here and we return it.
   - The second example is about how we deal with the guarded command "$- >$ ". If the scanner reads the first character as *minus* $(-)$ operator, then we use look-ahead character to find the next character. If the next character is $>$, then we consider the token as a guarded command; otherwise, returning token is considered as *minus* operator.

3. **Token recognizeName():** This function is used to return the token if it is recognized as an Identifier (ID) or reserve word.

4. **Token recognizeSpecial()**: This function is used to return the token if it is recognized as a special symbol.

5. **Token recognizeNumeral():** This function recognizes a decimal whole number and returns the token.

6. **void recognizeComment():** This function is used to recognize the comments. It continues reading until the next character is newline.

1.1.4 Hash table for storing word-tokens

Associated files: symboltable.h, symboltable.cc

We have implemented a Hash table named symbol-table to store the keywords and the identifiers. The main purpose of the designing the symbol table in scanning phase is to classify the word tokens into identifiers and keywords. We have implemented symboltable.h and symboltable.cc in this regard. The data members and members function of the symboltable.cc class is defined in symboltable.h file and the implementation of done inside the file symboltable.cc.

The symbol table or hash table is defined as "htable" in our program. We defined the size of symbol table to 401. We have used a prime number as the size of the hash table to avoid collision while distributing the keys to the table. We are using 17 keywords which were pre-defined by the PL Minus language. So, at the very beginning of the scanning process, we preload the "htable" (hash table) with the keywords.

In **symboltable.h**, we constructed a class named "Symboltable". The private members of this class are

1. A data structure of vector type is used to define the hash table in order to store tokens.
2. **int hashfn(string) :** This function returns the index for a token by using a simple linear probing function to map between token and key.
3. **Variables**
   - **int occupied :** It is used to check the occupied cells in hash table.
   - **int position :** It is used to hold the position returned by the function **"hashfn"**.

The public members of this class are

- **Symboltable() :** This is the default constructor function to initialize the symbol table and preload with reserve keywords.
- **void insert(Token):** This function is used to insert the keywords.
- **int insert(string) :** This function is used to insert the identifiers in the symbol table. Before storing the identifier in the symbol table, we checked few conditions such as if the token is already exists in the symbol table or not. If the answer is yes, then we do not need to insert the same token again in the table. As the pattern of the keywords and identifiers are the same, so we compared the incoming lexemes wih the stored keywords. As we already mentioned earlier, the **"hashfn(string)"** function returns an index for the lexeme. If it returns a position which is empty, then the lexeme is considered as an identifier. However, if it returns a position which is already occupied, then three cases can happen -
  - The lexeme can be an already declared identifier.
  - It can be a keyword.
  - **"hashfn(string)"** returns the same index for a new lexeme.

  For identifying if the new lexeme is a keyword or not, we checked the symbol value of the occupied position returned by the function as we assigned the symbol value of keywords starting from 286 and upwards. If the occupied position consists of a token whose symbol value is greater or equals to 286, then it is obvious that the object in mentioned position is a keyword. Suppose, the symbol value of the occupied cell is 292. We subtracted 286 from 292 which is 6. Afterwards, we compared our lexeme with the string at forth position of keywords array. If they match then we came to a conclusion that, its a keyword. This searching is also quite efficient as the cost of this searching is done with $O(1)$. If the the returned symbol value is less than 286, then we compared it with the lexeme that is already stored in that position. If it matches then then we consider that this lexeme is already stored in the symbol table. If none of the above two cases happens, then we go to the next position and repeat the above process.
- **int find(string) :** This is used to search for a lexeme. It returns its position if the lexeme is found otherwise returns -1.
- **bool full() :** This function returns true if the table is full otherwise it returns false.

- **int getOC() :** This function returns the number of occupied cells in symbol table.
- **void print() :** For printing the symbol table, this function is used.

The implementation of all these functions are done in **symboltable.cc** file.

2 Syntax Analysis

Syntax analysis or parsing is the second phase of a compiler construction. Lexical analyzer reads the sequence of character from the input stream and converts it to the sequence of tokens. Syntax analyzer takes these tokens as input from source file and by using the Context Free Grammar (CFG) parse those tokens.

A **Context Free Grammar** has four components:

- A set of non-terminals (V) : Non-terminals basically generate grammar.
- A set of terminal symbols ($\sum$) : Terminals are the basic symbols or set of tokens.
- A set of productions (P) : Productions specify the manner in which the terminals and non-terminals can be combined to form strings. left side of a production is always non-terminal and right side is the sequence of terminals and/or non-terminal that can be deduced form the left side.
- A start symbol (S): One of the non-terminals from where the production starts.

In the second phase of our project, we have built a parser for PL language that takes the tokens as input produced by our scanner in the first phase and parse these tokens with the help of Context free grammar.

2.1 Parser Construction Details:

We have divided the parser construction into three parts.

1. Recursive Descent Parsing
2. First and Follow Set Computation
3. Error Recovery

Before constructing the parser, we have slightly changed our previous scanner work. Previously, we had only one type of symbol ID for identifiers but now there are three types of identifiers in the Context Free Grammar for our programming language PL.

In parser, variable name or constant name or procedure name can referred as ID. We need to resolve this problem to avoid ambiguous production. Let us define one example,

factor = constant | variableAccess

First of **constant** contains **constantName** and first of **variableAccess** contains **variableName**. So when we are at **factor** we must decide which production to be used next. To avoid these kind of situations, we added an attribute for ID symbols which is **idType**.

- For **variableName**, we set, **idType = 1**
- For **constantName**, we set, **idType = 2**; It occurs after **CONST** symbol
- For **procedureName**, we set, **idType = 3**; It occurs after **PROC** symbol.

When these cases occurs, we set the **idType** of those id in symbol table.

2.1.1 Steps of Syntax Analysis:

A) Recursive Descent Parsing:

Associated Files: parser.cc, parser.h

parser.h is the interface of the parse class. The data members and member functions of the **parser** class are defined in this file. The implementation of these functions are written in **parser.cc** file.

We have designed our parser as recursive descent parser. For every non-terminal we declared a method. As for any look ahead token, there is at most one production, so the grammar is ready for $LL(1)$ parsing. We did not find any case in which the grammar is ambiguous (as the identifier issue is resolved). Here is an example how we implemented the recursive descent parser. We have a non-terminal block and the method for this non-terminal is **block()**. The production rule for this non-terminal is :

block = 'begin' definitionPart statementPart 'end'.
definitionPart = {definition';'}
statementPart = {statement';'}

Here **'begin'** and **'end'** are terminal symbols and **definitionPart**, **statementPart** are non-terminals. When we expect a terminal we match the symbol. If it does not match then we show corresponding error message. For non-terminals, the corresponding method is called. The above mentioned production rule is implemented as follows.

```
void Parser::program(Symbol sym)
{
        outFile<<"program()"<<endl;

        //Building stop set using vector
        stopSet.push_back(sym);
        stopSet.push_back(DOT);

        //non terminal bloc
        block(stopSet);

        //Building stop set using vector that may appear after dot symbol
        vector<Symbol>().swap(stopSet);
        stopSet.push_back(sym); //adding symbols in stop set
```

```
        //this portion matches dot and if not matched the shows corresponding
         error message and finds the next stop symbol
        match(DOT, stopSet);

        //parsing done
        admin.done();
}
```

In this code block, first we match **'begin'** symbol, if it does not match, then we show the corresponding error message. When any terminal symbol is matched, we send an additional argument which is **stop set**. It helps parser to recover from error. We will discuss this in details in the Error recovery part. **definitionPart** can be empty so we need to grab a token in advance to see whether it is in **first** of **definitionPart** or in **follow** of **definitionPart**.If it is in **first** of **definitionPart**, we call the **definitionPart(vector<Symbol>)** and **statementPart(vector<Symbol>)** and finally match the **'end'** symbol. The whole recursive descent parser is implemented in the same way.

B) First and Follow set computation:

Associated Files: firstfollow.cc, firstfollow.h

To implement an **LL(1)** parser, we first calculate the first and follow set. Avoiding backtracking is one of the main concerns in designing the parser. Among the alternative choices (some production produces several grammar rule, some production produces empty strings), our parser should make its choice by looking at the next symbol only. In these case we need first and follow sets. For example:

definition = constantDefinition | variableDefinition | procedureDefinition

In this scenario, we use a look ahead token and decide which method should be chosen. We compute the **firstOfStatement()**, **firstOfDefinition()**, **firstOfExpList()** and **firstOfGCList()**. We have **firstOfConstDef()**, **firstOfVariDef()** and **firstOfProcDef()** methods in which returns set of first symbol in a vector for the corresponding non-terminal. Union operation for sets of symbol is needed in construction of first and follow sets. We have used + operator overloading of c++ for union operation of two vectors that are actually sets of symbols.

int(vector<Symbol>) method:
This functions is defined inside **parser.cc** file. After getting the vector for any first or follow set with the help of this function we check whether our look ahead token is in the set or not. If the look ahead symbol is in the set then it returns true otherwise false. In this scheme we choose which method to call using look ahead token.

In our PL grammar two productions produce empty in right hand side. We need to compute follow only for these two non-terminals. They are **definitionPart** and **statementPart**. We have methods for computing follow sets: **followOfDefPart()**, **followOfStatePart()**, **followOf-**

VaList(), **followOfExpression()** and **followOfGuardedCommand()**. As guardedCommand = expression '− >' statementPart, so follow of guardedCommand is also included in follow set of statement part.

C) Error Recovery:

Associated Files: parser.cc, parser.h

Error recovery is one of the important parts of syntax analysis. As our parser is a recursive descent parser, the parser will exit from the program after encountering any error, if there is no process for error recovery. We will have some fixed number of stop symbols. When any error will occur, we will start to read tokens until we have any of the stop symbols and then will start parsing from that sate. It's good but crude as well because we may miss many errors in between. Another way of error recovery is for every terminal there are different sets of stop symbols that may occur after the terminal symbol. So when we try to match any terminal but fail, then we call **syntaxError(vector<Symbol>)** method with corresponding stop symbols.

syntaxError(vector<Symbol>) method:

This method is used for error recovery. When we try to match any terminal symbol and fail, then we call this method and pass set of stop symbols in vector form. We take a look ahead token and check if the token is in stop symbol set using **in(vector<Symbol>)** method. If the look ahead token is in stop set, we stop there but if not then we continue to grab tokens until we reach to any of the stop symbols for the symbol which occurred error. **EOF** is always in stop symbol set as we must stop at **EOF** otherwise the parser will fall in loop. Recursive descent parser itself helps us to construct stop sets. At first when we call **program(Symbol)** method, pass **EOF** to it because **EOF** will occur at he end of the whole program. Then from this base we add or keep the stop symbol set and pass it to another non-terminal function calling. From a new method we see what may occur after this. If no non-terminal or terminal will occur afterwards then we keep the stop symbol as it was otherwise we add the probable symbols with previous stop symbols.

syntaxCheck(vector<Symbol>) method:

Our goal for error recovery was to ensure that the parser will always be in error free state. This **syntaxCheck(vector<Symbol>)** method with **syntaxError(vector<Symbol>)** method make that possible. This method checks that if our current look ahead token is in stop set or not. If it is not then after reporting syntax error it calls **syntaxError(vector<Symbol>)** which actually recovers parser from error.

match(Symbol,vector<Symbol>) method:

Though the main task of this method is to match terminals but if any terminal does not match it call **syntaxError(vector<Symbol>)** method and pass the stop set as parameter which recovers the parsers from error.

Error reporting is not done inside **parser.cc** or **scanner.cc**. These files detected the errors and reported to **administrations.cc**. Error reporting is done in this file. New line counter was also implemented here.

3 Semantic Analysis (Scope and Type check)

Type and scope checking is the third phase of compiler construction. In syntax analysis phase, the parser only verifies that whether the tokens are arranged in a syntactically valid form. In semantic analysis, we check if the tokens form a sensible set of instruction or not in the programming language. If we want our PL to be semantically correct, all of our variables must be properly defined and types of the variables must be valid. Semantic analysis is the last phase to wipe out errors from the source code. We perform two types of checking in semantic analysis.

1. Scope Check
2. Type Check

3.1 Detail Description of Semantic Analysis

Before describing scope and type check we would like to discuss about block table which is used in both scope and type checking.

3.1.1 Block Table

Associated Files: blocktable.h blocktable.cc

Our Block table is actually a vector of vectors. Rows of the vector represent individual blocks of source code and row consists of several vectors which are actually **TableEntry** of the variables. **blocktable.h** is the interface of the class in which Block table is implemented and the implementation is done inside **blocktable.cc**. Brief description of the important methods of this class are given below.

bool search(int) : This method returns true if the id we are looking for is found in the current block and returns false otherwise.

bool define(int, PL_Kind, PL_Type, int, int): This method creates new entry in block table based on **blockTable** variable. It returns true if the current block does not contain an object with our current id after creating an entry. Otherwise it returns false indicating ambiguous name i.e there is already a definition for the specific id in the current block. When we create an entry we insert an structure in **Block table**. The structure is given below.

```
typedef struct
{
    int id;//position of name in symbol table
    PL_Kind kind;// kind of name
    PL_Type type;// data type of name
    int size; // size of name
    int value; // numerical value
} TableEntry;
```

For setting type and kind we declared a header file **type.h** inside which we declared 2 **Enum** types **PL_Type** and **PL_Kind**. They are given bellow:

```
//Name Kind
enum PL_Kind
{
    CONSTANT=305, VAR, ARR, PROCEDURE, UNDEFINED
};

//Name type
enum PL_Type
{
    INTEGRAL=310, BOOLEAN, UNIVERSAL
};
```

bool newBlock() : This method returns an empty block and pushes it onto the stack of blocks.

TableEntry find(int, bool&): Using this method we search the entire table for an id in inside out fashion. Variable error is false if the id we are looking for is in the block table. Error is true if not found in the Block table.

3.1.2 Scope check

Associated Files: parser.h parser.cc

We just need to modify some methods inside **parser** class to do the scope checking. We know that there are two parts inside a block of our PL. They are **definitionPart** and **statementPart**. The **definitionPart** has some sub divisions like **constantDefinition**, **variableDefinition**, **procedureDefinition** and each of them has corresponding methods. Before returning from these methods we create entries for **constantName**, **variableName** and **procedureName** in block table. While creating an entry we set appropriate **kind** and **type** of these names. While creating an entry we also check that if the name is already inserted in the current block. If it is already there then we show ambiguous name error.

In **statementPart** we simply check if the name is defined or not and we also check whether the name is of correct kind. For example inside **procedureStament** when we match **procedureName**, we check its existence in the block table, if it exists we check if it's kind is **PROCEDURE** or not. If one the the two condition is false we show error message.

3.1.3 Type check

Associated Files: parser.h parser.cc

There are only two types in our PL : **Integer** and **Boolean**. In **definitionPart** we set the types of the individual names. For example inside method **variableDefinition(vector<Symbol>)** when **typeSymbol(vector<Symbol>)** method is called we get the type of our **variableName**. Previously the return type of **typeSymbol(vector<Symbol>)** was void and now we changed it to **PL_Type**. So after calling this method we get the type for **variableName**. We did this

and similar kind of modifications in some other methods of the parser that makes sure that the source code semantically correct or shows error if not. Other than **definitionPart**, we modified numbers of methods of **statementPart** for type checking. For example

$guarderCommand = expression \rightarrow statementPart$

Here expression must be of type **BOOLEAN**. Method that corresponds to expression has now return type **PL_Type** and when expression calls **primaryExpression** and inside **primaryExpression** if **relationaOperatior** is called after **simpleExpression** we know that the type of the **primaryExpression** is **Boolean** so it returns **BOOLEAN** (otherwise INTEGER and return type of primary expression is also changed to PL_Type). Finally expression method returns BOOLEAN.
For fulfilling this purpose we modified 17 methods of our parser so that it can perform type checking.

matchName(Symbol , vector<Symbol>): We have declared this new method only for matching names. The purpose is quite similar to our **match(Symbol sym , vector<Symbol> stops)** function. The difference after matching the name it returns the position of the name in symbol table which is stored as id in **TableEntry** in block table.

4 Code Generation

The final phase of compiler construction is code generation. Now output of a given source code can been seen by implementing lexical analysis, syntax analysis, scope & type checking and finally this phase. We used an abstract machine which was already developed and given to us by our professor. So our compiler generates PL abstract machine assembler code for given abstract machine, if the input source code is free from error. Our contribution in this final phase is to generate intermediate code for the abstract machine.

PL abstract machine has two parts:

- **Assembler** and
- **Interpreter**

From our PL source code, our compiler generates appropriate code and passes it to the abstract machine.

For simplicity two passes are implemented in Assembler. Now we can visualize what is happening in Assembler by describing Pass I and Pass II.

- **Pass I:** This pass ends with generating the PL source code in symbolic form. The forward references are unresolved in this form.
- **Pass II:** The output we get from pass I is the input of Pass II. Forward references are resolved. Then it produces machine code for the PL abstract machine.

The PL abstract machine interpreter then executes the output we get from pass II.

Associated files:

Assembler and Interpreter files are added but those files are provided by our professor. Besides, no new files were added to our existing project. Inside parser file, the code generation part was implemented. The code generation part was straight forward as we were given the code generation rules for PL. The challenge was to determine the relative block level and displacement of the variables. To resolve this problem we introduced two new entries for table entry structure in block table.

- One entry stores the block level where the variable was defined and
- the other entry stores the displacement of the variable.

We store these information in compile time. We get the relative block level from the difference of current level In runtime and the level we stored in block table. Using these information, we can correctly identify the variables and use them in abstract machine.

An example of our implementation:

In the final step of our project, we generated the intermediate code for phase I of the Assembler. To generate this code, we wrote three functions emit1(string), emit2(string,int) and emit3(string,int,int) in administrator class. This functions print the parameters provided to the corresponding method one at a line. For example, if we call admin.emit3("VARIABLE",0,5) from parser it will be written in the file as:

.
.
.
VARIABLE
0
5

Our generated code is in the postfix form. For example, if our input is: $a := b + c * 3$
Then the corresponding code will be in the following form:
a.code
b.code
c.code
3.code
MULTIPLY
PLUS
ASSIGN

The following code block is an example of the implementation of ifStament method.

```
// ifStatement = 'if' guardedCommandList 'fi'
void Parser::ifStatement(vector<Symbol> stops)
{
        for(int i=0; i<2; i++)
        {
            outFile1 << "        |\n";
        }
```

```
            outFile1<<"                ifStmt()"<<endl;
        int startLabel = NewLabel();// label of the instructions if
                                    // the boolean expression
                                    // evaluates to false
        int doneLabel = NewLabel(); // label of the instruction
                                    // immediately after code for if
                                    // instruction
        vector<Symbol>().swap(stopSet);
        stopSet.push_back(FI);

        match(IF, ff.firstOfGCList() + stops);
        guardedCommandList(startLabel,doneLabel,stopSet + stops);

        admin.emit2("DEFADDR", startLabel);  // If we have a runtime
                                             // error; address of the
                                             // FI instruction is
                                             // known
        // Add the FI command.
        admin.emit2("FI", admin.lineNo);
         // Define the address to jump to on
         // successful completion of the command.
        admin.emit2("DEFADDR", doneLabel); // The if-instruction is a
                                           // success, doneLabel is
                                           //the address of the next
                                           // instruction.

        match(FI,stops);
}
```

In this code we can see a new function NewLabel(). This function generates and returns a new label every time it is called and the label is used as address and sometimes it assigns variable lengthif required.

In summary, we modified the methods of parser. We added admin.emit1(string), admin.emit2(string,int), admin.emit3(string,int,int) where applicable. Finally, if the input code is free from any error, then we get the intermediate machine code written in a file.

Then the file pointer is passed to the Assembler. Using *the intermediate code*, we perform two passes in Assembler and get assembly code for interpreter. Then using *the machine code*, Interpreter executes the machine code, which is equivalent to the input PL source code.

Limitation

In our program, we have set the size of the symbol-table to 307 which is a static number. So, this compiler is limited to store only 307 symbols during lexical analysis. We could increase it's size. But, if the input program is much less than predefined size of the symbol-table, then a huge space will remain unused. This limitation might be solved by using a text buffer.