Report

Compiler Lab (CSN-362)

Ankur Parihar

16114015

All the source code and documents available at my repository:

https://github.com/ankurparihar/Compiler-Lab-Codes

Assignment 1

Problem Statement

Write a program to scan a program and identify tokens (identifiers, keywords, and numbers) in it.

Files

scan.cpp – main program to read program from file and output identified tokens.

program.txt – file that contains our program to be scanned.

Working

scan.cpp contains predefined keywords and operators. The program takes program file's name (*program.txt*) as an argument. The program is scanned character by character. We have defined word separators to determine when a word begins or ends. These include $< \boldsymbol{space} >$ $\boldsymbol{\backslash n}$ $\boldsymbol{\backslash t}$ $\boldsymbol{\backslash 0}$ ( ) { } [ ] ; , . " ' + − ∗ / =

If there is an alphanumeric character it is stored in a character array called **word**. If we see a single-quote or double-quote, we know that it denotes string or char literal. So the program continues to scan until it finds it again, and prints out the string literal. Otherwise if it is an operator it is directly printed out.

When we find a word separator, we append a \0 to the **word,** now we have a word stored in **word** array. There are some utility function namely – isKeyword, isOperator, isNumber. They function as the name suggest (described below) and takes the **word** with it's length as argument. So after a word separator the checking is done for all three cases and corresponding output is printed.

Detailed utility functions working

- isKeyword – takes **word** and length as argument. Compares the **word** with every member of predefined **keywords** array. Return true if match found else false.
- isOperator – takes **word** and length as argument. Compares the **word** with every member of predefined **operators** array. Return true if match found else false.
- isNumber – takes **word** and length as argument. Scans the word from start to end and if all the characters are between 0-9 return true else false.

After this is done the **word**'s index position is set to 0. And the cycle continues until the file pointer reaches end of file.

How to run

- compile `g++ scan.cpp -o scan`
- Input file : program.txt - Enter a simple program
- run `./scan program.txt`

Assignment 2

Problem Statement

Implement a desk calculator using operator precedence parsing.

Files

main.cpp – main program to read input and output result of calculation with steps.

table.cpp – stores and creates the operator precedence table

Working

The operator precedence table is a 2-dimensional character array. And it is modified in a way that instead of using indices you can directly input operators or characters to get precedence.

For example if you want to know the precedence of $< +,* >$ you can do this easily by $operator\_table[' + '][' - ']$. No need to remember indices and makes the code way easier.

Now the main program starts by taking input from command line. Input is an expression to be evaluate. It can contain $+$, $-$, $*$, $/$, numbers (int and float), and brackets ().

The program has **states** to check for invalid input. For example the input can only start from either a number or an opening bracket, or after a closing bracket there can only be an operator, another closing bracket or end of input. So the states are: `S_START, S_OPRTR, S_NUMRL, S_BRCKT_OPN, S_BRCKT_CLS.` States are updated and checked in every step of input processing. The program always starts with `S_START` state.

Then some utility functions are programmed that are listed below –

- isNumeric – takes a character as input and returns true if it's between 0-9 or a decimal-dot otherwise false.
- isOperator – takes a character as input and returns true if it's permitted operator else false.
- isAllowed – takes `prevState` (previous state) and `nextState` (next state) as input and returns if the state rules are not violated and the current state can be updated to next state. False means the input is not good, so the program exits immediately with an error message on console.
- char_to_double – takes input a character array and returns it's base-10 number equivalent with double precision. The number is initialized to 0. The array is parsed one by one and until it sees a dot the number is multiplied by 10 and added with character just parsed. It the dot has been seen, number is added with 10^{th} inverse power of just parsed character.
- getClosingBracket – takes input a character array, a start index and an end index. Returns the position of pairing closing bracket if found else -1. The method works based on stack counter. The counter is initialized to 1. The array is parsed and if an opening bracket is encountered the counter is incremented, else if closing bracket is encountered the counter is decremented. We know that for the pair closing bracket to

be found the counter must be 0. So the moment counter reaches 0, current index is returned or if the loop reaches to end -1 is returned.

- append, appendUnit – these are made to dynamically allocate memory. append increase the array by 5 units when needed whereas the appendUnit increase it with 1 every time.
- printStack – print the stack build during operations.
- reduceStack – takes a **values** array of double numbers, index **p** (where to store operation result), character **op** (what operation to be performed) and integer reference **p_end** (stack pointer to be reduced after operation). Based on the operation type desired operation is performed on target numbers and the result is stored in target position. The target number are at p^{th} and $p + 1^{th}$ position. After operation we have $p + 1^{th}$ index empty, so all the values after p are shifted back one position to fill the emptied position. In the end stack pointer is reduced by one.
- parseExp – takes in an expression that contains no brackets. It takes expression in character array form. The values associated with id's are provided in form of double array values. Two poiners p_val and p_val2 are used for operator precedence method to evaluate expression. An empty stack is initialized with $ symbol in 0^{th} position. The expression pointer's character and stack pointer's character are compared using operator precedence table and the precedence is determined. Based on that either new input character is append or stack is reduced using reduceStack.When both arrays come exhausts i.e. they both have pointing character $ the evaluated value is returned. It is always true that when both exhaust there is only one value left in **value** array. So the first value is returned.
- eval – It's the main outer loop. eval takes the input expression and using a recursive approach evaluates the whole expressions values. It is described in detail below.

The input is passed to eval. This function checks brackets and have the ability to detect divide by zero errors and state errors. It parses the expression character by character. If an opening bracket is encountered is uses getClosingBracket to isolate the bracketed sub expression. The subexpression is recursively parsed until all the brackets are resolved. The expression is converted to character array for operator precedence parsing and the token values are stored in double array. These are used by parseExp to get a final value. So a bracket-pair is always resolved in a double value. If there is only one values that is returned by eval

How to run and example

- compile `g++ main.cpp -std=c++11`
- run `./a.out` or `./a.exe`

```
$ 6+(5*3-(10/5))
> i / i $
> 10 5
> i * i - i $
> 5 3 2
> i + i $
> 6 13
> Evaluation: 19
```

Assignment 3

Problem Statement

magine the syntax of a programming language construct such as while-loop --

while (condition)

begin

statement ;

:

end

where *while*, *begin*, *end* are keywords; condition can be a single comparision expression (such as $x == 20$, etc.); and statement is the assignment to a location the result of a single arithmatic operation (eg., $a = 10 * b$).

Write a program that verifies whether the input follows the above syntax. Use *flex* to create the lexical analyzer module and write a C program that performs the required task using that lexical analyzer.

Files

main.l – stores regex definitions for tokens.

test.c – while loop program to test

inter – Intermediate file generated using *flex*.

phase2.c – parse *inter* file and determine result.

script.sh – automation script.

Working

First of all write a simple while loop program in *test.c*. *main.l* contains regular expressions defined for keywords like *while*, *begin*, *end* , brackets, numbers, operators, tokens, statement and comparison.

The lex program is provided to *flex* and it generates an intermediate file *lex.yy.c*. It is a C program which have properties defined in lex file and will be used to parse our while loop.

It is compiled using gcc and executed with input stream as file. The file is scanned and all the regex conversion are performed as defined in *main.l*. The output is streamed to *inter* file.

Now we know that the while loop in inspection follows a pattern. The pattern is as follows:

- It must only begin with `while-bracket_open-token-comparison_op-number-bracket_close`
- The next line must contain `key_begin` and the last line must contain `key_end`.
- Between `key_begin` and `key_end` there can only be `Statement`s.

The *phase2.c* file has a program to detect this pattern. It starts with character by character comparison of first two lines, as they must be exactly as pattern says.

Then there is checking for `Statement` strings. In our *main.l* `Statement` is printed with a tab in beginning. So while we encounter a tab there must be a `Statement`.

In the last step we check for loop ending by checking for string `key_end`.

If there is any case where the strings do not match we print corresponding error to indicate that while loop's syntax is not correct.

How to run

- Make script executable by `chmod +x script.sh only in linux`
- Input while loop in file: test.c
- run `./script.sh`

Assignment 4

Problem Statement

Write a C program for implementing shift-reduce parsing using a given grammar.

Firstly, define the data structures for representing the given CFG in BNF, the stack for the parsing, and the parse tree to be created.

Files

main.c – main program to read input from file and output result of parsing.

grammer.txt – grammar rule specification file

program.c.txt – C style syntax while loop program

output.txt – redirected output stream, results or execution in specified input

Data structure for CGF in BNF

Grammar rules are stored in a structure which have a size and array of grammar rules.

- Size – integer
- Rules – grammar rule structure

The grammar rule structure is made of three components –

- starting symbol – character
- right hand side rule – character array pointer
- rule length - integer

Working

We specify the while loop grammar in *grammer.txt* and a while loop in C styled syntax in *program.c.txt*.

We store the intermediate (in process) expression in a stack, which have a size and a character array pointer.

There are some utility functions created –

- pushStack – push character in stack
- popStack – replace a part of stack with character and reduce stack pointer by 1
- replaceStack – pop character and reduce stack pointer accordingly
- isalphanumeric – takes a character and determine if the character is alpha-numeric
- isarithop – takes a character and determine if it is arithmetic operator
- isimmutable – takes a character and returns true if it is one of the following – ( ) { } = ; These characters are not modified in any way and used in grammer.

- Iswhile – takes a character array with length and determine if the array represents the string *while* by comparing in character by character.
- SubStr – takes two strings as input and return index of first occurrence of second string in first string, otherwise -1;

In the beginning our program starts to read while loop syntax character by character. The whole program is stored in a character array. While reading the program all the unnecessary characters like $\backslash n,\ \backslash t,\ \backslash 0,\ <space>$ are removed. And now we call it a *linear code* because it is just one line.

In the next step the *linear code* is parsed from beginning to end character by character. Lexems from code are identified and converted accordingly. The result is stored in another character array called *transformed code*. In the *transformed code* all the lexems are converted to predefined character. The rules for conversion are:

- *while* – *W*
- Identifier – *I*
- Comparison operator – *C*
- Arithmetic operator – *A*

Now that we have transformed our code in a very simple form, we need grammar to check it.

The grammar is scanned from file *grammer.txt*. We count the lines in file to determine number of rules present in file. The size of resultant grammar structure array is determined accordingly. Now the grammar is read line by line and stored in data structure described above. For verification of grammar rules input we print grammar rules.

Now the shift-reduce is used. The *transformed code* is parsed character by character. We declare a push variable which is initialized to true because in the beginning we need to push. The loop starts traversing the code. If the push is enabled it will push the character to stack otherwise reduce the stack using some grammar rule. In every loop we check the stack for reduction possibility for every rule using SubStr method. If for some rule's r.h.s. substring is found then push if set to false. After reducing we set the push to true for next loop iteration. This is the case until we have the *transformed code*.

After we have parsed the entire code there is only reduction possibility. So in next step we check for reductions until either the stack is completely reduced to starting symbol or the stack cannot be reduced further.

During all the iterations we print the stack and remaining string of code along with shift/reduce action with additional information.

If there is only starting symbol in the end of everything the syntax is true, otherwise it is not .

How to run

- Compile `gcc main.c`
- Write a while loop in file: `program.c.txt` and grammar rules in file: `grammer.txt`
- Run `./a.out`

Assignment 5

Problem Statement

Write a LR parser program in C. Define the data structure for the parsing table in such a way that it can be initialized easily (manually) for a given grammar. Take a simple grammar, e.g., expression grammar, compute the parsing table entries by hand using the steps discussed in the class, and initialize the table in your program with these values. Try to parse input expressions scanned by a lexical analyzer (which can be easily created using flex). The output of the parser should be SUCCESS or FAILURE depending on the input. In case of FAILURE the parser should indicate the incorrect token in the input.

Files

main.c – main program to read input from file and output result of parsing.

table.c – table data structure

Data structure for Table

The *table.c* has a table structure called *lr_table*. It contains the following –

- Number of states in table (*num_states*)
- Number of non-terminals (*num_nonterm*)
- Number of terminals (*num_term*)
- Action table structure (*at*)
- GoTo table structure (*gt*)

The action table structure is called *action_table*. It has the following components –

- Symbols array – terminals (*symbols*)
- Action matrix (*table*)

The action structure is called *action*. It is composed of –

- Type - shift/reduce/accept/error (*type*)
- Value (*value*)

The goto table structure is called *goto_table*. It has the following components –

- Symbols array – non terminals (*symbols*)
- Integer matrix *table*

That's all we need for LR parser table. Now the next step is to create table and store input in it. For this we have CreateTable function.

We first allocate the memory for our table. Then we ask users about no. of states, no. of terminals and no. of non-terminals and accordingly allocate arrays and matrices. Now that we have a solid structure of table we only need to fill it.

We ask user to provide terminal and nonterminal symbols and then table entries first action table and then goto table row by row.

After creating the table a pointer to table is returned.

It has some utility function like char_to_col (takes a character and returns it's location in symbol array of any table), printTableNice (Print table).

Main program

The input format is as follows –

- The first line of input contains number of rules *n*.
- Then next *n* lines contains rules property *lhs* (left hand side symbol) and *rhs* (no of symbols in right hand side). e.g. $A -> Aa$ becomes $A\ 2$
- Next line no of terminal symbols
- Next line no of non-terminal symbols
- Next line no of states generated for LR table
- Next line non-terminal symbols space separated
- Next line terminal symbols space separated
- Next line Table rules (action) in matrix form: `00=blank, si=shift i, ri=reduce i, a0=accept`
- Next line Table rules (goto) in matrix form: `-1=blank`
- Final line expression to evaluate

In *main.c* there is structure for rules. It has a character *c* (left hand side) and integer *n* (no. of characters in right hand side). And using this rule structure a rules array containing whole grammar data necessary for further process.

The program has some utility function like –

- append_int – append integer to dynamically allocated integer array
- getReduction – count how many characters to reduce given a rule
- getRedChar – return which symbol to be replaced after reduction (non-terminal)
- printStack – print the stack
- appendRule – append rule structure to dynamically allocated rules array

In main function take used input like no. of rules, and rules properties. Then retrieve a *lr_table* named *lrt* using table.c's CreateTable function and expression to be evaluated in string form. For verification we print all the retrieved information in a nice format.

The expression is parsed character by character. Every character is checked against LR parser table using a loop. The stack is initialized with state 0. After this stack always contains character and state in pair.

Inside the loop the character is first checked with action table. The next action is determined.

- If the next action it is error or accept then we simply print a message and exit program.
- If it is shift action, we shift the character in stack along with the action value. Now the value is our new state.

- If it is reduce state. The rule to which reduction is to be done is provided by action's value. Using getReduction we determine how many pairs (character and state) to remove. The stack pointer is reduced accordingly. Now the new symbol is retrieved using getRedChar. Using this symbol and latest state in the stack and the goto table we determine the next state and insert it into stack.

In each iteration the stack along with next action is printed out.

That's how the whole expression is parsed until we encounter accept or error and corresponding message is printed out.

How to run

It is better to specify input in a file e.g. *input.txt*.

- Compile `gcc main.c`
- Run `./a.out < input.txt > output.txt`

Output is stored in output.txt

Assignment 6

Problem Statement

Modify your LR parser program of the preceding assignment such that along with the reduce actions, the parser invokes routines associated with the particular grammar rule. For example, for a reduction by the rule E -> E + T of the exression grammar, the parser computes the sum of the numbers corresponding to the symbols E and T on the RHS, and associates the sum to the symbol E on the LHS.

Hint: Observe that it will be necessary to associate values with different symbols in the stack. Whenever a reduce action is taken some symbols of the stack are to be replaced by a non-terminal symbol. In this step the value to be associated with the non-terminal is to be computed using the values associated with the symbols that are being replaced.

Files

main.c – main program to read input from file and output result of parsing.

table.c – table data structure

Data structure for Table

The *table.c* has a table structure called *lr_table*. It contains the following –

- Number of states in table (*num_states*)
- Number of non-terminals (*num_nonterm*)
- Number of terminals (*num_term*)
- Action table structure (*at*)
- GoTo table structure (*gt*)

The action table structure is called *action_table*. It has the following components –

- Symbols array – terminals (*symbols*)
- Action matrix (*table*)

The action structure is called *action*. It is composed of –

- Type - shift/reduce/accept/error (*type*)
- Value (*value*)

The goto table structure is called *goto_table*. It has the following components –

- Symbols array – non terminals (*symbols*)
- Integer matrix *table*

That's all we need for LR parser table. Now the next step is to create table and store input in it. For this we have CreateTable function.

We first allocate the memory for our table. Then we ask users about no. of states, no. of terminals and no. of non-terminals and accordingly allocate arrays and matrices. Now that we have a solid structure of table we only need to fill it.

We ask user to provide terminal and nonterminal symbols and then table entries first action table and then goto table row by row.

After creating the table a pointer to table is returned.

It has some utility function like char_to_col (takes a character and returns its location in symbol array of any table), printTableNice (Print table).

Main program

The input format is as follows –

- The first line of input contains number of rules *n*.
- Then next *n* lines contains rules property *lhs* (left hand side symbol) and *rhs* (no of symbols in right hand side). e.g. $A -> Aa$ becomes $A\ 2$
- Next line no of terminal symbols
- Next line no of non-terminal symbols
- Next line no of states generated for LR table
- Next line non-terminal symbols space separated
- Next line terminal symbols space separated
- Next line Table rules (action) in matrix form: `00=blank, si=shift i, ri=reduce i, a0=accept`
- Next line Table rules (goto) in matrix form: `-1=blank`
- Final line expression to evaluate

In *main.c* there is structure for rules. It has a character *c* (left hand side) and integer *n* (no. of characters in right hand side). And using this rule structure a rules array containing whole grammar data necessary for further process.

The program has some utility function like –

- append_int – append integer to dynamically allocated integer array
- getReduction – count how many characters to reduce given a rule
- getRedChar – return which symbol to be replaced after reduction (non-terminal)
- printStack – print the stack
- appendRule – append rule structure to dynamically allocated rules array
- printBuffer – print values associated with id's in any step
- reduceBuffer – customized rules to reduce stack associated values in buffer according to rule specified.

In main function take used input like no. of rules, and rules properties. Then retrieve a *lr_table* named *lrt* using *table.c*'s CreateTable function and expression to be evaluated in string form. For verification we print all the retrieved information in a nice format.

We have a buffer array to store the values corresponding to stack id's. The values in buffer are changed according to situations.

The expression is parsed character by character. Every character is checked against LR parser table using a loop. The stack is initialized with state 0. After this stack always contains character and state in pair. And there is always a value in buffer associated with each identifier in stack.

Inside the loop the character is first checked with action table. The next action is determined.

- If the next action it is error or accept then we simply print a message and exit program.
- If it is shift action, we shift the character in stack along with the action value. Now the value is our new state. If the character is identifier we append its numeric value to buffer.
- If it is reduce state. The rule to which reduction is to be done is provided by action's value. Using getReduction we determine how many pairs (character and state) to remove. The stack pointer is reduced accordingly. Now the new symbol is retrieved using getRedChar. Using this symbol and latest state in the stack and the goto table we determine the next state and insert it into stack. In order to update buffer values we use the customizable reduceBuffer function. Provided buffer array, rule number and index it updates the buffer accordingly

In each iteration the stack along with values and next action is printed out.

That's how the whole expression is parsed until we encounter accept or error and corresponding message is printed out.

How to run

It is better to specify input in a file e.g. *input.txt*.

- Compile `gcc main.c`
- Run `./a.out < input.txt > output.txt`

Output is stored in output.txt

Assignment 7

Problem Statement

Take the C grammar from the book - The C Programming Language (by Kernighan and Ritchie), and try to generate a parser for the language using YACC. The notation for the grammar in the book is not strictly BNF (e.g. use of subscript "opt" with some symbols, use of "one of", etc.). Some rewriting is required due to that. Apart from that there are some LALR conflicts which are to be resolved by appropriate means.

Files

lab7.l: The lex file which returns the token corresponding to each lexeme scanned by the C program.
lab7.y: The yacc file which calls the flex file for getting token values. It is the file which contains the C code for reading input, getting the token and matching it with the context-free grammar.
input: The file which contains the input to be parsed by the lexical analyzer defined in the previous two files.
build.sh: script to run flex, bison and gcc commands

Grammar for C

- https://www.lysator.liu.se/c/ANSI-C-grammar-l.html : The link contains the Lex specification of the C grammar. The version is very close to the current C standard grammar. It has all the required definitions of the tokens which can be encountered in a C program. This lex code was used in the assignment.
- https://www.lysator.liu.se/c/ANSI-C-grammar-y.html : The link contains the Yacc specification of the C grammar. The version is very close to the current C standard grammar. It has the grammar to match C code along with the required functions to check the success or failure. This code was used for yacc file in this assignment.

Conflicts in Grammar

There is a problem of **dangling else** in C grammar. It is a problem in programming in which an optional else clause in an *if_then_else* statement results in nested conditionals being ambiguous. Formally, the reference context-free grammar of the language is ambiguous, meaning there is more than one correct parse tree.

In many languages, one may write conditionally executed code in two forms: the *if-then* form, and the *if-then-else* form – the else clause is optional:

- *if a then s*
- *if b then* s_1 *else* s_2

This gives rise to an ambiguity in interpretation when there are nested statements, specifically whenever an if-then form appears as s_1 in an if-then-else form:

- *if a then if b then s else* s_2

In this example, s is unambiguously executed when a is true and b is true, but one may interpret s_2 as being executed when a is false (thus attaching the else to the first if) or when a is true and b is false (thus attaching the else to the second if). In other words, one may see the previous statement as either of the following expressions:

- *if a then (if b then s) else* s_2
- *if a then (if b then s else* s_2*)*

In LR parsers, the dangling else is the archetypal example of a "shift-reduce conflict".

In C, the grammar reads, in part:

```
statement = …

   | selection-statement

selection-statement = …

   | IF ( expression ) statement

   | IF ( expression ) statement ELSE statement
```

Thus, without further rules, the statement could ambiguously be parsed as if it were either:

```
if (a)              |       if (a)
{                   |       {
  if (b)            |          if (b)
    s;              |            s;
  else              |       }
    s2;             |       else
}                   |          s2;
```

AVOIDING CONFLICT: The grammar is changed such that there are no ambiguous if statements.

```
statement: open_statement
        | closed_statement
        ;

open_statement: IF '(' expression ')' simple_statement
        | IF '(' expression ')' open_statement
        | IF '(' expression ')' closed_statement ELSE open_statement
        | WHILE '(' expression ')' open_statement
        ;

closed_statement: simple_statement
        | IF '(' expression ')' closed_statement ELSE closed_statement
        | WHILE '(' expression ')' closed_statement
        ;

simple_statement: ...
        ;
```

Instead of changing the grammar, the shift-reduce conflict was handled by telling the bison and flex file in advance that they will face such a conflict. And the action to be taken at that stage would be shift operation. Reduction should be delayed. This results in solving almost all the dangling if else problems.

Assignment 8

Problem Statement

Take a common programming language construct of an HLL, such as the for-loop construct of the C language. Use LEX and YACC to create a translator that would translate input into three-address intermediate code. The output of the translator should finally be in a file. Assume a simple structure for "statements" that may appear inside the construct, a make necessary assumptions for the intermediate code format.

Files

lab8.l: The lex file which returns the token corresponding to each lexeme scanned by the C program.

lab8.y: The yacc file which calls the flex file for getting token values. It is the file which contains the C code for reading input, getting the token and matching it with the context-free grammar.

input: The file which contains the input to be parsed by the lexical analyser defined in the previous two files.

build.sh: script to run *flex*, *bison* and *gcc* commands

Working

lex and *yacc* often work well together for developing compilers.

The program uses the lex-generated scanner by repeatedly calling the function yylex(). This name is convenient because a yacc-generated parser calls its lexical analyser with this name.

To use lex to create the lexical analyser for a compiler, end each lex action with the statement return token, where token is a defined term with an integer value.

The integer value of the token returned indicates to the parser what the lexical analyser has found. The parser, called yyparse() by yacc, then resumes control and makes another call to the lexical analyser to get another token.

In a compiler, the different values of the token indicate what, if any, reserved word of the language has been found or whether an identifier, constant, arithmetic operator, or relational operator has been found. In the latter cases, the analyser must also specify the exact value of the token: what the identifier is, whether the constant is, say, 9 or 888, whether the operator is + or *, and whether the relational operator is = or >.

The lab8.y file contains the following sections:

- Declarations section. This section contains entries that:
- Include standard I/O header file
- Define global variables
- Define the list rule as the place to start processing

- Define the tokens used by the parser
- Define the operators and their precedence
- Rules section. The rules section defines the rules that parse the input stream.
- *%start* - Specifies that the whole input should match stat.
- *%union* - By default, the values returned by actions and the lexical analyzer are integers. yacc can also support values of other types, including structures. In addition, yacc keeps track of the types, and inserts appropriate union member names so that the resulting parser will be strictly type checked. The yacc value stack is declared to be a union of the various types of values desired. The user declares the union, and associates union member names to each token and nonterminal symbol having a value. When the value is referenced through a $$ or $n construction, yacc will automatically insert the appropriate union name, so that no unwanted conversions will take place.
- *%type* - Makes use of the members of the %union declaration and gives an individual type for the values associated with each part of the grammar.
- *%tokens* - Lists the tokens which come from lex tool with their type.
- Programs section. The programs section contains the following subroutines. Because these subroutines are included in this file, you do not need to use the yacc library when processing this file.

Subroutines

- main – The required main program that calls the yyparse subroutine to start the program.
- yyerror – This error-handling subroutine only prints a syntax error message.
- yywrap – The wrap-up subroutine that returns a value of 1 when the end of input occurs.

Lexical analyser

The *y.tab.h* file contains definitions for the tokens that the parser program uses. In addition, the *lab8.l* file contains the rules to generate these tokens from the input stream.

A stack is maintained in the *lab8.y* file to store the tokens that are analysed by the lexical file using the yyparse and yytext commands. The push() function is designed to undergo this task of pushing the tokens onto the stack. For loop normally comprises of the initialisation, then condition and then operations on loop variable.

The following functions are implemented for the generation of these different expressions:

- codegen - generates the code for initialisation
- codegen_rel - generates the code for relational operators by picking the top element and top-2 element and assigning the parsed relational operator $<, >, =$
- codegen_umin - generates the unary minus of the topmost element of stack
- codegen_assign - generates the code by assigning the top-2 value to the top token or identifier on the stack.

The For Loop statement is evaluated in the following fashion –

```
S: FOR '(' E ';'{lab1();} E {lab2();}';' E {lab3();}')' E';'{lab4(); exit(0);}
```

These four labels in the for statement are evaluated as follows:

- lab1 - Increment the label number l_no value by 1.
- lab2 - Assign temporary variables such that t0 is the limit - index variable and t1 as not 0. For example, in `for(i=0;i<3;i=i+1) a=3;`

  ```
  T0 = 3 - i

  T1 = not 0
  ```

  Generate the following statements next:

  If t1 goto L1

  Goto L2

  Which implies the control will go to L1 i.e. out of the loop once the value of t1 becomes 0.

  Increment the label number l_no by 1.

- lab3 - It generates the code for "`goto Label L0`" which jumps to the function lab2. A temporary variable *t2* is also assigned which helps to increment the value of counter variable *i* with each iteration.
- lab4 - It generates the code for "`goto Label L3`" to increment the value of *i* once the statement is executed.

How to run

- Run batch script ./build.sh to run the following commands:
- lex c.l
- yacc c.y
- gcc y.tab.c -ll -ly -w4
- Run ./a.out and enter the for expression