Lab Report

for

Compiler Laboratory

Submitted in requirement for the course

CSN - 362

of Bachelor of Technology in Computer Science and Engineering

by

Garvit Dewan

(Enrollment Number : 16114025)

Submitted to

Dr. Pradumn K. Pandey

Department of Computer Science & Engineering

Indian Institute of Technology Roorkee

Contents

Assignment 1

1.1 Problem Statement

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

1.2 Resources

- C++

1.3 Files

- `lexical.cpp` - program file of program to read a program file and output identified tokens.

1.4 Program Logic

`lexical.cpp` takes in the file name of the program file as an argument. It contains predefined keywords (C language), operators (C style), delimiters (`(){}[];,.<space>\t\n`) and comment tokens. The program file is read character by character. A buffer of 32 bytes is used to store the intermediate token value. The procedure for detecting tokens is as follows:

1. If a token begins with `/` and if the next character is also a `/`, then is a single line comment. Continue taking input until a `\n` (newline) is encountered, after which a new token begins. All the input taken during this process is ignored.

2. If a token begins with `/` and if the next character is `*`, then this it is a block comment. Continue taking input until a `*/` is encountered, after which a new token begins. All the input taken during this process is ignored.

3. If an alphanumeric character is encountered, continue taking input and storing in buffer. If any character encountered during this process is an alphabet, then the token is an `identifier` and the identifier flag is set to true.

4. If a delimiter or operator is encountered, it marks the end of the token.

   If identifier flag is set, then the token is matched with the list of keywords. If successful, the token is identified as that `keyword`. If all keywords fail to match, then the token is identified as `identifier`.

   If the identifier flag is not set, then the token is identified as `number`. A new token begins from the next character.

1.5 How to Run and Example

```
$ g++ lexical.cpp -o lexical
$ ./lexical sample.c
int is a keyword.
a is an identifier.
1112 is a number.
```

Assignment 2

2.1 Problem Statement

Implement a desk calculator using operator precedence parsing.

2.2 Resources

- C++

2.3 Files

- `op_pred.cpp` - program file of program that takes an input and parses it using operator precedence parsing and calculates the result.

2.4 Program Logic

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

For example, if we want to know the precedence of `<+, *>` we can do this easily by `table['+']['*']`. This eliminates the use of indices.

	+	-	*	/	^	id	(	)	$
+	>	>	<	<	<	<	<	>	>
-	>	>	<	<	<	<	<	>	>
*	>	>	>	>	<	<	<	>	>
/	>	>	>	>	<	<	<	>	>
^	>	>	>	>	<	<	<	>	>
id	>	>	>	>	>			>	>
(	<	<	<	<	<	<	<	≐	
)	>	>	>	>	>			>	>
$	<	<	<	<	<	<	<		

Figure 1: Operator Precedence Table

To optimize memory and make the functioning clearer, the program, instead of including an overhead of parsing a parse table first, contains `switch-case` statements for every operator, which helps it decide which action to perform next.

There are two stacks S1 and S2. The stack S1 is used to perform operator precedence parsing whereas the stack S2 is used to perform the calculations. The program reads the input line by line, and based upon the stack top and `switch-case` statements, performs the next action.

An action can either be either pushing the input to the stack when the input is of higher priority than the top of the stack, or by popping from stack when the input is of lower precedence than the top of the stack. When this happens, the corresponding calculation is performed on stack S2 by popping the top two numbers, applying the operator to the numbers, and saving the result by pushing it back on to the stack.

If the parsing is successful, the program prints `Accepted`, and the top item on S2 is printed as the result of the calculation.

2.5 How to Run and Example

```
$ g++ op_pred.cpp -o op_pred -std=c++11
$ ./op_pred
Make sure the input ends with $, else the program will return
an error.
3+4*5-6/3$
Accepted.
21
```

Assignment 3

3.1 Problem Statement

Imagine 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 comparison expression (such as `x == 20`, etc.); and statement is the assignment to a location the result of a single arithmetic 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/C++ program that performs the required task using that lexical analyzer.

3.2 Resources

1. C
2. flex
3. yacc
4. Make

3.3 Files

1. `main.c` - the driver program
2. `file1.l` - file containing token actions in flex format
3. `file2.y` - file containing the grammar for the while loop construct in yacc format
4. `Makefile` - build automation commands

3.4 Program Logic

First of all the tokens and their corresponding actions are defined in the file `file1.l`. This file is then called with flex, which creates an intermediate file `lex.yy.c` which is a C program of a lexical analyzer for the tokens we provided. It returns the next token by calling `yylex()`.

Next, the grammar of the language (`while` loop) is defined in `file2.y`. This file is then called with yacc, which produces the intermediate files `y.tab.c` and `y.tab.h`. These files form a C program for a parser for the provided grammar. By default, the generated parser uses flex's `yylex()` to take input the next token. The parser generated is a LR(1) parser. A `yyerror()` function is also added to the yacc file for error reporting.

Finally, the `main.c` program opens the target program file and calls yacc's `yyparse()` to parse the program. On successful parsing, it prints nothing. If there is an error in the program file, the parser reports the line number and the token of error.

3.5 How to run

```
$ make
$ ./parse <file>
```

Assignment 4

4.1 Problem Statement

Write a C/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.

4.2 Resources

1. C++
2. flex
3. make

4.3 Files

1. `main.cpp` - the main program file containing the parser program
2. `calc.lex` - file containing token actions in flex format
3. `parse_table.txt` - file containing the LR(1) parse table
4. `Makefile` - build automation commands

4.4 Program Logic

We first define the tokens in the program, i.e. the lexical tokens and the grammar tokens. A map and a reverse map is created for convenient programming. Next, the parse table is created, with the tokens in the same order as defined earlier. This parse table file is read and parsed in the main program. The data structure used to store the parsing table is `vector<map<int, pair<int, char>>>`. It is a vector of mappings of next input character to pair of next state and action (**s**hift, **r**educe or **a**ccept) for each state.

Next, the reduce rules are defined. The data structure used for representing reduce rules is `vector<pair<int, int>>`, which is a pair of token to reduce to, and the number of tokens to be reduced.

A loop is run, which calls `yylex()` for token. The top token from the stack is taken, and using the input value and the stack top value, the parse table is read. If an entry exists, it returns the next action:

1. If the next action is `shift`, it also returns the next state to shift to, which is pushed onto the stack.

2. If the next action is `reduce`, it also returns the reduce rule by which the expression is to be reduced. The stack is accordingly popped, and the nodes are created and joined together for the parse tree. The operator table is read again for the `GOTO` state and is pushed onto the stack.

3. If the next action is `accept`, the parsing is successful and the loop terminates. The parser prints `Accepted` and the parse tree is printed out by traversing it using `DFS Traversal`.

If the entry does not exist, the parser throws an `error` with the token name and line number of the error. The parsing is halted.

4.5 How to Run

```
$ make
$ ./parse <file>
```

Assignment 5

5.1 Problem Statement

Write a LR parser program in C/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.

5.2 Resources

1. C++
2. flex
3. make

5.3 Files

1. `main.cpp` - the main program file containing the parser program
2. `calc.lex` - file containing token actions in flex format
3. `parse_table.txt` - file containing the LR(1) parse table
4. `Makefile` - build automation commands

5.4 Program Logic

We first define the tokens in the program, i.e. the lexical tokens and the grammar tokens. A map and a reverse map is created for convenient programming. Next, the parse table is created, with the tokens in the same order as defined earlier. This parse table file is read and parsed in the main program. The data structure used to store the parsing table is `vector<map<int, pair<int, char>>>`. It is a vector of mappings of next input character to pair of next state and action (**s**hift, **r**educe or **a**ccept) for each state.

Next, the reduce rules are defined. The data structure used for representing reduce rules is `vector<pair<int, int>>`, which is a pair of token to reduce to, and the number of tokens to be reduced.

A loop is run, which calls `yylex()` for token. The top token from the stack is taken, and using the input value and the stack top value, the parse table is read. If an entry exists, it returns the next action:

1. If the next action is `shift`, it also returns the next state to shift to, which is pushed onto the stack.

2. If the next action is `reduce`, it also returns the reduce rule by which the expression is to be reduced. The stack is accordingly popped. The operator table is read again for the `GOTO` state and is pushed onto the stack.

3. If the next action is `accept`, the parsing is successful and the loop terminates. The parser prints `SUCCESS`.

If the entry does not exist, the parser prints `FAILURE` with the token name and line number of the error. The parsing is halted.

5.5 How to Run

```
$ make
$ ./parse <file>
```

Assignment 6

6.1 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 $E \rightarrow E + T$ 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.

6.2 Resources

1. C++
2. flex
3. make

6.3 Files

1. `main.cpp` - the main program file containing the parser program
2. `calc.lex` - file containing token actions in flex format
3. `parse_table.txt` - file containing the LR(1) parse table
4. `Makefile` - build automation commands

6.4 Program Logic

We first define the tokens in the program, i.e. the lexical tokens and the grammar tokens. A map and a reverse map is created for convenient programming. Next, the parse table is created, with the tokens in the same order as defined earlier. This parse table file is read and parsed in the main program. The data structure used to store the parsing table is `vector<map<int, pair<int, char>>>`. It is a vector of mappings of next input character to pair of next state and action (**s**hift, **r**educe or **a**ccept) for each state.

Next, the reduce rules are defined. The data structure used for representing reduce rules is `vector<pair<int, int>>`, which is a pair of token to reduce to, and the number of tokens to be reduced.

The parser program uses two stacks, one for parsing and one for the evaluation of expressions. A loop is run, which calls `yylex()` for token. The top token from the parse stack is taken, and using the input value and the parse stack top value, the parse table is read. If an entry exists, it returns the next action:

1. If the next action is `shift`, it also returns the next state to shift to, which is pushed onto the parse stack. If the input token was a number, then it is pushed onto the evaluation stack.

2. If the next action is `reduce`, it also returns the reduce rule by which the expression is to be reduced. The parse stack is accordingly popped. The operator table is read again for the `GOTO` state and is pushed onto the stack. The corresponding evaluation is performed in the evaluation stack by popping two values, performing the operator on the two operands and pushing back the result on the evaluation stack.

3. If the next action is `accept`, the parsing is successful and the loop terminates. The parser prints `SUCCESS`, along with the evaluated result.

If the entry does not exist, the parser prints `FAILURE` with the token name and line number of the error. The parsing is halted.

6.5 How to Run

```
$ make
$ ./parse <file>
```

Assignment 7

7.1 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.

7.2 Resources

1. The C Programming Language (by Kernighan and Ritchie)
2. C
3. flex
4. yacc
5. make

7.3 Files

1. `main.cpp` - the main program file containing the parser program
2. `parse.lex` - file containing token actions in flex format
3. `parse.y` - file containing the grammar for the C language in yacc format
4. `Makefile` - build automation commands

7.4 Program Logic

First of all the tokens and their corresponding actions are defined in the file `file1.l`. This file is then called with flex, which creates an intermediate file `lex.yy.c` which is a C program of a lexical analyzer for the tokens we provided. It returns the next token by calling `yylex()`.

Next, the grammar of the C language is defined in `file2.y`. This file is then called with yacc, which produces the intermediate files `y.tab.c` and `y.tab.h`. These files form a C program for a parser for the provided grammar. By default, the generated parser uses flex's `yylex()` to take

input the next token. The parser generated is a LR(1) parser. A `yyerror()` function is also added to the yacc file for error reporting.

Finally, the `main.c` program opens the target program file and calls yacc's `yyparse()` to parse the program. On successful parsing, it prints nothing. If there is an error in the program file, the parser reports the line number and the token of error.

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 s1 else s2`

This gives rise to an ambiguity in interpretation when there are nested statements, specifically whenever an `if-then` form appears as `s1` in an `if-then-else` form:

- `if a then if b then s else s2`

In this example, `s` is unambiguously executed when `a` is true and `b` is true, but one may interpret `s2` 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 s2`
- `if a then (if b then s else s2)`

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
```

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 can be handled by setting flag in yacc to `expect` shift reduce errors. And the action to be taken at that stage would be shift operation, which is the default action taken in such a situation. This effectively solves the dangling `if-then-else` problem and any other shift reduce conflicts that might arise.

7.5 How to Run

```
$ make
$ ./parse <file>
```

Assignment 8

8.1 Problem Statement

Take a common programming language construct of an HLL, such as the for-loop construct of the C language. Use `FLEX` 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, and make necessary assumptions for the intermediate code format.

8.2 Resources

1. Compilers: Principles, Techniques, and Tools (by Aho. A. V.)
2. C
3. flex
4. yacc

8.3 Files

1. `gen.l` - file containing token actions in flex format
2. `gen.y` - file containing the grammar for the C language in yacc format, and also the main program file

8.4 Program Logic

First of all the tokens and their corresponding actions are defined in the file `gen.l`. This file is then called with flex, which creates an intermediate file `lex.yy.c` which is a C program of a lexical analyzer for the tokens we provided. It returns the next token by calling `yylex()`.

Next, the grammar of the language (`for`, `while` loops) is defined in `gen.y`. This file is then called with yacc, which produces the intermediate files `y.tab.c` and `y.tab.h`. These files form a C program for a parser for the provided grammar. By default, the generated parser uses flex's `yylex()` to take input the next token. The parser generated is a LR(1) parser. A `yyerror()` function is also added to the yacc file for error reporting.

Finally, the `main()` program defined in `gen.y` opens the target program file and calls yacc's `yyparse()` to parse the program. On successful parsing and evaluation, the output is stored in a

file named `output.txt`. If there is an error in the program file, the parser reports the line number and the token of error.

The `gen.y` file along with grammar also contains the code generation instructions for each production rule. The code generation rules have been taken from the book *Compilers: Principles, Techniques, and Tools (by Aho. A. V.)*. Functions for generating labels, creating new temporary variables and data structures for storing attributes and line numbers were defined. All the data is managed through a symbol table.

8.5 How to Run

```
$ lex gen.l
$ yacc -d gen.y
$ gcc y.tab.c lex.yy.c -lfl -o gen
$ ./gen <file>
```