



Compiler Project (Group 2)

BCSE UG-3 SEMESTER II



Prof. (Dr.) Nandini Mukhopadhyay

Prof. (Dr.) Kamal Sarkar

Contributors

- Rwitick Ghosh (001910501016)
- Saurabh Mukherjee (001910501006)
- Atanu Ghosh (001910501005)
- Sumon Chakrabarty (302010501002)

Overview

April 18, 2022

Consider a simple C-like language with

Data Types: integer, real and character

Declaration statements: identifiers are declared in declaration statements as basic data types and may also be assigned constant values (integer or floating) Condition constructs: if, then, else. Relational operators used in the if statement are < (less than), > (greater than), == (equal) and != (not equal).

Example:

```
if (a<10) then{  
    a=a*a;}  
else {a=a/2;}
```

Overview (Contd.)

Nested statements are supported. There cannot be if statement without else statement.

Assignments to the variables are performed using the input/output constructs:

`cin >> x`- Read into variable x

`cout << x`-Write variable x to output

Arithmetic operators (+, -, *, /, %) and assignment operator '=' are supported

Only function is `main()`, there is no other function. The `main()` function does not contain arguments and no return statements.

Goals

1. **CFG** : Create CFG for this language.
2. **Lexer** : Write a Lexical Analyser to scan the stream of characters from a program written in the above language and generate a stream of tokens.
3. **Tables** : Maintain tables to implement scope rules.
4. **Parser** : Implementing a bottom-up parser for this language (modules include Item-set construction, computation of FOLLOW, parsing table construction and parsing).
5. **Additional** : Visual representation of Transition Graph (DFA) for Item-Sets.

Milestones

1. Constructing the DFA for the item set transitions.
2. Constructing the Context Free Grammar.
3. Constructing the First and Follow Sets.
4. Implementing the Lexical Analyser.
5. Implementing the Parser using First and Follow Sets.
6. Visualizing the transition diagram (used `graphviz` library of python3)

Folder Structure

grammar - Grammar rules and descriptions

lexer - Lex file. Reads the input and writes the tokens to out/lex/<input filename>.tkl

parser - Parses program files

results - Parsing table, Transition graph, Graph visualization

symbol table - Symbol table generating program and writes the tokens to out/symbol table/<input filename>.csv

util - Utility Cpp program to prepare for read and write to files

out - Output files from the lexer, parser, and symbol table

bin - Stores all intermediate files that are generated.

cmd - Consists shell scripts to simulate the whole project

Directory and Files Tree

```
bin
├── lexer.bin
├── lexer.yy.c
├── parser.bin
├── symboltable.bin
├── symboltable.yy.c
├── tests
│   └── online.txt
└── visualize.bin

cmd
├── lex.sh
├── parse.sh
├── symboltable.sh
└── visualize.sh

grammar
├── finalGrammar.txt
└── terminals description.txt

lexer
└── lexer.l

LICENSE

out
├── lex
│   ├── sample.tkl
│   └── test.tkl
├── parser
│   ├── parser.txt
│   └── visualize.txt
└── symbol table
    ├── sample.csv
    └── test.csv
```


Directory and Files Tree (contd.)

```
├── parser
│   └── parser.cpp
├── question.txt
├── README.md
├── results
│   ├── graph.dat
│   ├── graph.dot
│   ├── graph.svg
│   └── table.dat
├── run.sh
├── sample.prg
├── symbol table
│   └── symboltable.l
├── test.prg
├── util
│   ├── filereadwrite.cpp
│   └── filereadwrite.h
├── visualizer/
├── transition.py
└── visualize.cpp
```

A Sample Program

```
sample.prg
sample.prg

1  main()
2  {
3      int x, n2, y;
4      cin >> x;
5      y = x + 2;
6      cout << y;
7
8      if (x ≤ y)
9          then
10             {
11                 real x2;
12                 cout << x;
13                 x2 = 34.256 - -85;
14                 if (x = y)
15                     then
16                         {
17                             char y1, n2;
18                             y1 = 'c';
19                             n2 = 34;
20                             cout << y;
21                             x = x * x;
22                             n2 = 40 % -70;
23                         }
24                     else
25                         {
26                         }
27                     x = y = 2;
28             }
29         else
30             {
31                 while (x < y)
32                 {
33                     int n25, y75;
34                     x = x + 1;
35                 }
36             }
37     }
38
```

Flowchart



Phases of Project

The four main phases of this project are -

- Constructing a **CFG**
- Lexical Analysis and **Tokenlist generation**
- Maintaining a **Symbol Table**
- Implementing a **Bottom-Up** Parser



PART 1 : CFG

**Generating a Context-Free-Grammar for the
given language**



What is a Context-Free-Grammar ?

⇒ Context free grammar is a formal grammar which is used to generate all possible strings in a given formal language.

Context free grammar G can be defined by four tuples as : **$G = (V, T, P, S)$**

Where,

G describes the grammar

T describes a finite set of terminal symbols.

V describes a finite set of non-terminal symbols

P describes a set of production rules

S is the start symbol.

CFG For C-Like Language

$G \rightarrow P$

$P \rightarrow mfgbS$

$S \rightarrow c | AS | DS | CS | LS | IS$

$D \rightarrow dV$

$V \rightarrow vzV | v;$

$T \rightarrow v | n$

$A \rightarrow v = X$

$X \rightarrow TaT; | T; | A$

$C \rightarrow \text{ifRgtbSebS}$

$L \rightarrow wfRgbS$

$I \rightarrow pT; | sT;$

$R \rightarrow TrT$

Where,

m - main

b - opening brace curly ({)

c - closing brace curly (})

v - variable name (max size is 32 by lexer)

n - constant

a - arithmetic operator

r - relational operator

d - data type (declaration must be in the beginning of the block)

i - if

t - then

e - else

w - while

f - Opening brace / (

g - Closing brace /)

p - cout<<

s - cin>>

z - comma(,)

other terminals occurring with the above

; - semicolon

= - assignment operator


Important Features of Our CFG

- The Grammar has no **left recursion** (A Grammar $G (V, T, P, S)$ is left recursive if it has a production in the form. $A \rightarrow A \alpha \mid \beta$.)
- **Grammar is Unambiguous** (A grammar can be unambiguous, if the grammar does not contain ambiguity. This means if it does not contain more than one left most derivation (LMD) or more than one right most derivation (RMD) or more than one parse tree for the given input string).
- The Grammar is parsable by SLR(1) .
- The if-then block must be succeeded by else block.
- Blocks can be empty.



PART 2 : Lexical Analysis

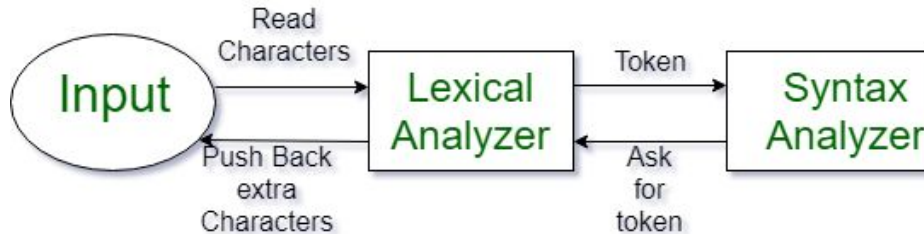
Lexical Analysis to scan the stream of characters from a program



What is Lexical Analysis ?

➡ Lexical Analysis is the first phase of the compiler also known as a scanner. It converts the High level input program into a sequence of **Tokens**.

- Lexical Analysis can be implemented with the [Deterministic finite Automata](#).
- The output is a sequence of tokens that is sent to the parser for syntax analysis.



How Lexical Analyzer Functions ?

1. **Tokenization** i.e. Dividing the program into valid tokens.
2. Removes white space characters.
3. Removes comments.
4. It also provides help in generating error messages by providing row numbers and column numbers.
5. The lexical analyzer identifies the error with the help of the automation machine and the grammar of the given language on which it is based like C, C++, and gives row number and column number of the error.

[Note : **Regular expressions** have the capability to express finite languages by defining a pattern for finite strings of symbols. The grammar defined by regular expressions is known as **Regular Grammar**. The language defined by regular grammar is known as **Regular Language**.]

Directory `./lexer`

- The goal of this module is to read a text file as input and convert them to **meaningful tokens**. We've considered a bunch of different types of tokens for that.
- Here in this lexer, we basically simulated a **DFA** which reacts specifically to an input character and transits to a same/different state based on the same. And this requires us to implement the following packages and structures.

The DFA can be found as **graph.svg** in the `./results` directory.

Rules for lexer

m	"main"
b	"\{"
c	"\}"
a	"+" "-" "*" "/" "%"
r	"<" ">" "<=" ">=" "==" "!="
d	"int" "real" "char"
i	"if"
t	"then"
e	"else"
w	"while"
f	"("
g	")"
sp	["\t" "\n"]
p	"cout"{sp}*"<<"
s	"cin"{sp}*">>"
z	","
v	[a-zA-Z][a-zA-Z0-9]*
int	[0-9]+
real	[0-9]+ "." [0-9]+
char	\".\\"
n	(\-{int} {real}) {char}



PART 3 : Symbol Table

Maintaining a Symbol Table



What is a Symbol Table?

⇒ It is a data structure being used and maintained by the compiler, consisting of all the identifier's names along with their types. It helps the compiler to function smoothly by finding the identifiers quickly.

- It is built-in lexical and syntax analysis phases.
- The information is collected by the analysis phases of the compiler and is used by the synthesis phases of the compiler to generate code.
- It is used by the compiler to achieve compile-time efficiency.

Directory ./symbol table

- The goal of this module is to store the information about the occurrence of various entities such as objects, classes, variable name, interface, function name etc. it is used by both the analysis and synthesis phases

Rules for Symbol Table

```
b "{"  
sp [\t " \n]  
d "int"|"real"|"char"  
ignore "main"|"if"|"then"|"else"|"while"|"cout"|"cin"  
v [a-zA-Z][a-zA-Z0-9]*  
int [0-9]+  
real [0-9]+ "."[0-9]+  
char \'\  
n (\-{int}|{real})|{char}
```


Sample output of symboltable.l in .csv format

```
Line,Column,Value,Type,Scope
3,9,x,int,1
3,12,n2,int,1
3,16,y,int,1
5,13,2,constant,
11,18,x2,real,2
13,14,2,constant,
13,18,34.256,constant,
13,27,-85,constant,
17,26,y1,char,3
17,30,n2,char,3
18,22,1,constant,
18,26,'c',constant,
19,22,2,constant,
19,26,34,constant,
22,22,2,constant,
22,26,40,constant,
22,31,-70,constant,
24,21,2,constant,
30,17,n25,int,5
30,22,y75,int,5
31,21,1,constant,
```


Sample Output of `symboltable.1`

1	Line	Column	Value	Type	Scope
2	3	9	x	int	1
3	3	12	n2	int	1
4	3	16	y	int	1
5	5	13	2	constant	
6	11	18	x2	real	2
7	13	14	2	constant	
8	13	18	34.256	constant	
9	13	27	-85	constant	
10	17	26	y1	char	3
11	17	30	n2	char	3
12	18	22	1	constant	
13	18	26	'c'	constant	
14	19	22	2	constant	
15	19	26	34	constant	
16	22	22	2	constant	
17	22	26	40	constant	
18	22	31	-70	constant	
19	27	21	2	constant	
20	33	17	n25	int	6
21	33	22	y75	int	6
22	34	21	1	constant	



PART 4 : Parsing

Bottom-Up Parser for this language



What is a parser ?

⇒ The **parser** is that phase of the compiler which takes a token string as input and with the help of existing grammar, converts it into the corresponding Intermediate Representation.

- INPUT OF A PARSER

A parser takes input in the form of a **sequence of tokens**, interactive commands, or program instructions.

- OUTPUT OF A PARSER

A parser breaks the input tokens into parts that can be used by other components in programming. Generally, it produces output in the form of **parse tree**.

Bottom Up Parser

- It will start from string and proceed to start.
- In Bottom-up parser, Identifying the correct handle (substring) is always difficult.
- It will follow rightmost derivation in reverse order.
- Build the parse tree from leaves to root. Bottom-up parsing can be defined as an attempt to reduce the input string w to the start symbol of grammar by tracing out the rightmost derivations of w in reverse.

PARSER USED : SLR PARSER

Steps for constructing the SLR parsing table :

1. Writing augmented grammar
2. LR(0) collection of items to be found
3. Find FOLLOW of LHS of production
4. Defining 2 functions:
 - a. `goto[list of terminals]` and
 - b. `action[list of non-terminals]` in the parsing table

RULE –

- If any non-terminal has ' . ' preceding it, we have to write all its production and add ' . ' preceding each of its production.
- From each state to the next state, the ' . ' shifts to one place to the right.

Directory ./parser

- Global Data Structures Used in parser.cpp and their uses:

1. **map<char, vector<string>> productionMap:**
 - > production map stores the non terminal and its all productions
2. **map<string, int> productionRuleNumberingMap:**
 - > assigns a rule number to each individual production
3. **map<int, string> RuleProductionNumberingMapForReduction:**
 - > stores the rule for the assigned rule number in previous for easier access while reduction
4. **map<set<string>, int> RuleToItemNumberMap:**
 - > stores item set number for the closure of each item set
5. **map<int, set<string>> ItemNumberToRuleMap:**
 - > this is for accessing the item set easily using the given item set number
6. **queue<int> itemProcessQueue:**
 - > this is for storing the items in FCFS order for breadth first search of the graph where vertices are item sets and edges are transitions

Directory `./parser` (contd.)

- Global Data Structures Used in `parser.cpp` and their use-cases (contd.) :

7. `map<pair<int, char>, string> ActionTable:`

-> this is used to assign action to each item number corresponding to a symbol in the grammar.

8. `map<pair<int, char>, string> GotoTable:`

-> this is used to store the state changes corresponding to any non terminal appearing.

9. `map<char, set<char>> followSetMap:`

-> stores the follow set for each symbol in the grammar.

10. `set<int> ss:`

-> stores the first set at any point of time.

11. `char start:`

-> 'start' is used to store the augment symbol of the grammar.

12. `map<char, set<char>> firstSetMap:`

-> stores the first set for each production rule.

Functions used in `parser.cpp`

1. `FirstSet()`

- a. **Return type** - bool
- b. **Arguments** - char, char, char
- c. **Use**: This function generates the first set for the character supplied to it as an argument

2. `prepareFirstSet()`

- a. **Return type** - void
- b. **Arguments** - void
- c. **Use** : generates the firstSet for all the terminals/non-terminals appearing in our grammar

3. `prepareFollowSet()`

- a. **Return type** - void
- b. **Arguments** - void
- c. **Use** : generates the followSet for all the terminals/non-terminals appearing in the grammar

Functions used in `parser.cpp` (contd.)

4. `processGrammar()`

- a. **Return type** - void
- b. **Arguments** - void
- c. **Use** : This function reads the grammar from the text file and stores it for further actions

5. `insertIntoReduceRule()`

- d. **Return type** - void
- e. **Arguments** - string, int
- f. **Use** : Supplied a string which is a reduced rule in the Item Set , it inserts that into the Reduce Rules data structure.

Functions used in `parser.cpp` (contd.)

6. `insertIntoActionTable()`

- a. **Return type** - void
- b. **Arguments** - int, char, int
- c. **Use** : prepares the actionTable for the given char and item set number

7. `printClosure()`

- a. **Return type** - void
- b. **Arguments** - set<string>
- c. **Use** : prints the Closure of an item set

8. `prepareClosure()`

- a. **Return type** - void
- b. **Arguments** - set<string>
- c. **Use** : it prepares the closure of an item set from a set of base production rules with dots

Sample C like language

```
main() {  
    int x;  
    cin >> x;  
    cout << x;  
}
```

Generated Token List

mfgbdv;sv;pv;c

Generated Symbol Table

Line,Column,Value,Type,Scope

2,9,x,int,1

Generated Transition Diagram

Can be found in this link : [here](#)

FIRST SET FOR OUR GRAMMAR

FIRST:

$A = \{v\}$

$C = \{i\}$

$D = \{d\}$

$I = \{p, s\}$

$L = \{w\}$

$P = \{m\}$

$R = \{n, v\}$

$S = \{c, d, i, p, s, v, w\}$

$T = \{n, v\}$

$V = \{v\}$

$X = \{n, v\}$

FOLLOW SET OF OUR GRAMMAR

FOLLOW:

$A = \{c, d, i, p, s, v, w\}$

$C = \{c, d, i, p, s, v, w\}$

$D = \{c, d, i, p, s, v, w\}$

$I = \{c, d, i, p, s, v, w\}$

$L = \{c, d, i, p, s, v, w\}$

$P = \{\$ \}$

$R = \{g\}$

$S = \{\$, c, d, e, i, p, s, v, w\}$

$T = \{;, a, g, r\}$

$V = \{c, d, i, p, s, v, w\}$

$X = \{c, d, i, p, s, v, w\}$

PARSING OF THE SAMPLE C LIKE LANGUAGE

Shift m and move to :2
Shift f and move to :3
Shift g and move to :4
Shift b and move to :5
Shift d and move to :13
Shift v and move to :25
Shift ; and move to :33
Reduce: V->v;
Reduce: D->dV
Shift s and move to :16
Shift v and move to :29
Reduce: T->v
Shift ; and move to :38



Reduce: l->sT;
Shift p and move to :15
Shift v and move to :29
Reduce: T->v
Shift ; and move to :37
Reduce: l->pT;
Shift c and move to :12
Reduce: S->c
Reduce: S->IS
Reduce: S->IS
Reduce: S->DS
Reduce: P->mfgbS

---ACCEPTED---

An ERRONEOUS SCENARIO

```
main()  
    int x;  
    cin >> x;  
    cout << x;  
}
```

Shift m and move to :2

Shift f and move to :3

Shift g and move to :4

The input string can't be parsed, No entry in Action Table

Goals Achieved

- Successfully incorporated support for **int**, **real** (float in traditional C lang) and **char** data types. Variables can be declared and initialized in the same line. Also declaring multiple variables of any particular data type in a single line is possible.
- Scope rules has been implemented in the parser. **Identifiers with same name can be declared and initialized inside different scopes** and inside functions but it will throw in an error if the same has been done inside same block of code. Different scopes can be achieved with help of curly braces (i.e {} => C like syntax).
- Conditional constructs like **If...else** blocks have been implemented successfully. **Nesting of If..else** blocks are also possible. And they work fine with deep nesting also.
- Additionally, looping construct is provided through **while** block which can be **nested** too.
- So, to conclude , we can say that all the milestones which were set has been successfully achieved .

ACKNOWLEDGEMENT

At last, we would like to give our profound thanks to **Prof. Nandini Mukhopadhyay** Ma'am and **Prof. Kamal Sarkar** Sir for providing us the opportunity to work on something which is very practical and essential for any computer science student .This was a great opportunity for all of us and we are highly grateful to and would welcome any further suggestions for the scope of improvement.



Thank You !!