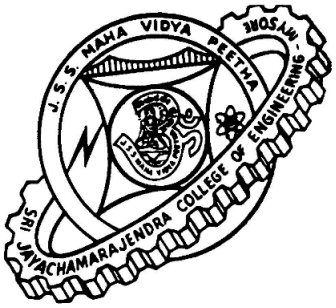
VISVESWARAYA TECHNOLOGICAL UNIVERSITY

BELGAUM

SRI JAYACHAMARAJENDRA COLLEGE OF ENGINEERING
MYSORE-570006

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING



Project on

IMPLEMENTATION OF HANDWRITTEN LEXER AND PARSER FOR βPARSE - A 'C' SUBSET

Guidance of

P.M. SHIVMURTHY

Lecturer

Department of CS&E, SJCE, Mysore.

Team:

NAME	ROLL NO	USN
VIKRAM TV	61	4JC07CS120
PRABHAKAR GOUDA	37	4JC07CS070
SHARAD D	03	4JC06CS089

Contents

1 Overview

β - parse language is the beta version of parsing of our own language. It involves the best possibilities that we have seen from the classic language 'C'. Our language is purely a subset of the higher level language - 'C'. It takes program written as per the rules and syntax of our language as input and validates the syntax of the program. In case of any error in the program it display the line number in which the error is detected and possible type of error. It also generates a parse tree for the user convenience. This implementation is done in two phases - the lexical phase and the parsing phase.

2 Glossary

βparse	Our own defined language.
source program	User program written in βparse spec.
keywords	The keywords supported by our language.
HTML	Hyper Text Markup Language.
dotty	Product to support graph display.
kgraphviewer	Product to support graph display (kde version).
*.cs	(C Subset) Our own language extension although any extension is supported.
delimiter	any character other than in the range a-z A-Z 0-9.
whitespace	a space, a tab or a newline is a whitespace.
Lexeme	A sequence of characters in the source program that matches the pattern for a token.
Token	A pair consisting of a token name and an optional attribute value.
Pattern	A description of the form that the lexemes of a token may take.
Symbol Table	The data structures that are used by parser and lexer to hold information.
Lexer	Scanner which finds the token and return to the parser.
Parser	Which handles the token.

3 Constructs of the Language

The program consists of 2 parts:

- The 'file inclusion' part
- The 'function definition' part

3.1 File Inclusion

File inclusion is done using the keyword **include**.

Syntax:
include "*file name*"

File names should not contain any extensions. Hence unique files are included. The include directive can be placed anywhere in the program and every file that is to be included should in a separate line.

Example: include "stdio"

The directive includes the file "stdio" to the current scope.

3.2 Function

Function can be started by specifying the function name and parameter list for the function enclosed within parenthesis. Every function starts with the special character '{' and ends with '}'.The function definition is very similar to 'C'

Syntax:
functionName (parameterList)

{

...

body

...

}

The parameter list can be zero or more. The elements of the list should be separated by comma (,). The function body can contain any number of basic statements like assignment, increment, decrement that resemble the 'C' statements. The body can also include the relational operations, looping operations and function calls also.

Example: main(int argc, char **argv){ s = p + v; a++; }

or

```
main(int argc, char **argv)
 {
    s = p + v;
    a++;
       ...
}
```

Every statement with in the function body should end with a semicolon (;) similar to 'C' language syntax. The *return value* of the function lies in its name only.

The details of the various types of statements in our language construction is dealt in the next section.

4 Syntax of Statements

The following are the various types of syntax that are used:

4.1 Operators and Symbols

The following **operators** are supported in the language. It includes the basic arithmetic operators.

+	to perform binary addition of two variables.
-	to perform binary subtraction of two variables.
*	to perform binary multiplication of two variables.
/	to perform binary division of two variables.
,	(comma) to separate the variable declarations and also to separate assignment statements.
=	to assign the rvalue obtained by an expression to the lvalue.
∧	to perform the binary power operation.

Relational Operators:
The following operators on two variables or expressions returns a Boolean value. These operators are used to test the various conditions in the language.

Operator	Operation	Notation
==	checks for equivalence of two variables or expressions.	EE
!=	checks if left variable or expression is equal to or not to the right.	NE
>	checks if left variable or expression is greater than to the right.	GT
<	checks if left variable or expression is less than or equal to the right.	LT
>=	checks if left variable or expression is greater than or equal to the right.	GE
<=	checks if left variable or expression is less than or equal to the right.	LE

The following **symbols** are used:

; to terminate a statement in the program.
( ) to specify a function call and the precedence of evaluation.
{ to open the block statement.
} to close the block statement.

4.2 Variables

The language supports all the characters (both uppercase and lowercase) of the English Language. It also supports the integer values. Any variable can be declared on the lines of variable declaration in 'C'.

- The variable can have both the characters and integers in it.
- The variable should start with an English character and not by an integer.
- The variable can be of any length.
- Keywords of the language cannot be used as variables.

4.3 Commenting

Commenting in the language is similar to the commenting styles in C language.

Syntax:

/**comment* */

Everything between /∗ and ∗/ also considered as comment it is not parsed.

4.4 Declaration

The type declaration of variables used in the program are not needed and the variables can be used directly, although some of the basic data types (integer,

float, character) are provided. The variable is casted according to the rvalue used in the program. Also multiple declaration of variables in a single line separated by commas are possible. The declarations are to be terminated by a semicolon to mark the end.

Keyword	Meaning	Example statement
int	Integer	int a,b;
float	Floating point	float fl;
char	character	char ch;

The declaration of a new variable is made at its first assignment.
Example: a = 3;

This declares the variable 'a' and assigns to it the value 3.

4.5 Assignment

The assignment statements are implemented based on the syntax of assignments done in 'C' Language. The 'rvalue' is assigned to the 'lvalue'. The left value can consist only one variable and not a number as a single value cannot be assigned to two or more variables on left. The right value can consist of a variable, number or a combination of both to form a compound expression.

All assignment statements should terminate with a semicolon.

The following assignment statements are supported:

- Direct Assignment as 'a = b;'
- Evaluation of expressions within '( )', that has the highest order.

4.6 Relational Condition Checking - The '*if-else*' construct

The language uses the following syntax to perform the relational condition checking.

Simple IF syntax:

if(*condition*)

{

statements

}

IF-ELSE syntax:
if(*condition*)

{

statement1

}

else

{

statement2

}

The *if* keyword tests for the Boolean value of the condition. The condition uses the relational operators between two variables or expressions. If the condition is true, then the *statements1* are executed. In the second case, if the condition results false then the keyword *else* is parsed to execute the *statements2*.

The statements1 and statements2 can inturn contain any number of nested '*if-else*' conditions.

4.7 Looping Construct

The looping construct is implement in the language using the keywords **for** and.

Syntax:
for(*initialization*;*condition*;*incrementation*;)
{

statements

}

Looping of the *statements* is done only if the *condition* is satisfied.
Example

```
for(i = 0 ; i<= n ; i = i + 1;)
{

        ...
        a = a + b;
        ...
}
```

The statements can in turn contain any number of nested looping structures.

4.8 Nesting of Statements

The language supports nesting of statements.

The statements containing the '*if-else*' and '*for-loop*' can be nested. Any number of statements can be used in them. Also the nested can be to any level in depth. The 'if' can nest 'if' or 'loop' or both. Similarly, 'loop' can nest 'loop' or 'if' or both.

Thus any of the *nesting* combinations works well with the language.

4.9 Program

The language supports to have *any number of* 'functions', 'file inclusions', 'conditional executions', 'looping constructs' and 'statements' in the program. Each construct is supposed to be terminated by appropriate terminator.

5 Transition Diagrams

Transition diagram has collection of nodes called as states. Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that one of the several patterns.

5.1 Transition diagram for relational operators

The transition starts from state 0(initial state). After scanning the symbol <(less then) it goes to state '1' there are three options from state '1' namely

2, 3 and 4 shown in diagram, where state 2, 3 and 4 are final states which returns respective tokens. Similar operation in other states, it is clear in below diagram.

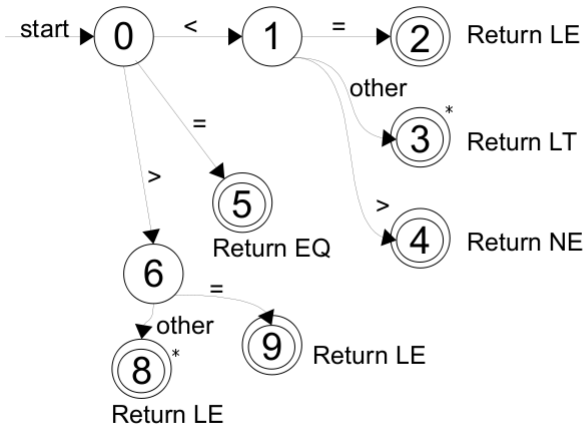


Figure 1: Transition diagram for relational operators.

5.2 Transition diagram for reserved words and identifiers

keywords like *if* or *then* are reserved, so they are not identifiers even though they look like identifiers.To recognise the identifier the transition diagram as shown in fig:2.

5.2.1 Steps to follow to recognise keywords

- Install the reserved word in symbol table initially.
- Create separate transition diagram for each keywords.

5.3 Transition diagram for unsigned number

Transition diagram for token *number* is as shown in Figure 3.

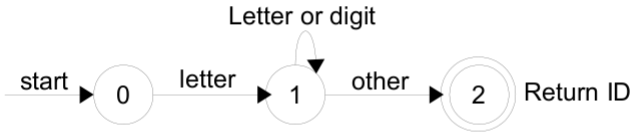


Figure 2: Transition diagram for recognising identifiers.

6 Grammar of our language

Since our language is subset of 'C',the program is collection of blocks. Each block may contain one or more statements. The block is opened by the character { and the block terminates by }. The production is as shown below.

$$
\begin{array}{rcl}
\textit{program} & \rightarrow & \textit{block} \\
\textit{block} & \rightarrow & \text{'\{'}\ \textit{stmts}\ \text{'\}'} \\
\textit{stmts} & \rightarrow & \textit{stmts1}\ \ \textit{stmt} \\
 & | & \varepsilon
\end{array}
$$

It is clear that statement has vast option it may start with expression or may *if* statement or it may loop construct. All this options are grouped in *FIRST(stmt)*. Let us look at productions of *stmt*.

$$
\begin{array}{rcl}
\textit{stmt} & \rightarrow & \textit{expr}\ ; \\
 & | & \textit{IF} \\
 & | & \textit{FOR} \\
 & | & \textit{block}
\end{array}
$$

For the above productions *IF* and *FOR* are still non terminals, they again have the following grammar.

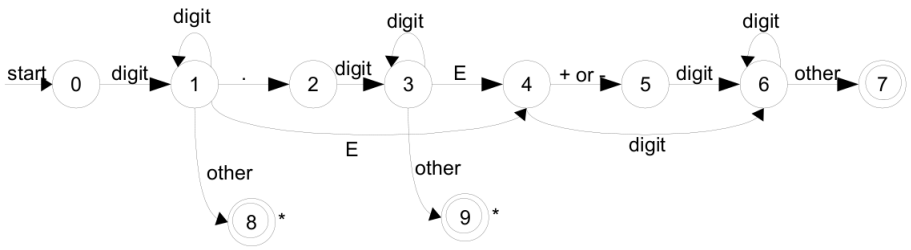


Figure 3: Transition diagram for unsigned number.

$$
\begin{array}{rcl}
IF & \rightarrow & \textbf{if } (\ expr\ )\ stmt1 \\
 & | & \textbf{if } (\ expr\ )\ stmt1\ \textbf{else}\ stmt2 \\
FOR & \rightarrow & \textbf{for } (\ expr1\ ;\ expr2\ ;\ expr3\ ;\ )\ \{\ stmt\ \}
\end{array}
$$

The production of *expr* contains arithmetic expression or relation expression which the language supports.

$$
\begin{array}{rcl}
expr & \rightarrow & rel = expr1 \\
 & | & rel
\end{array}
$$

$$
\begin{array}{rcl}
rel & \rightarrow & \textbf{id} < \textbf{id} \\
 & | & \textbf{id} <= \textbf{id} \\
 & | & \textbf{id} > \textbf{id} \\
 & | & \textbf{id} >= \textbf{id} \\
 & | & \textbf{id} != \textbf{id} \\
 & | & \textbf{id}
\end{array}
$$

For a detailed description of all the grammars used in our language, please refer to the section *Grammars* at last.

7 Tokens and Symbol Table

A token is identified by two parts - <tokenName, attribute>. TokenName is given by a value as given by the macro definition and it identifies a token type. Attribute is the lexeme itself. A token is returned from the lexer

everytime a parser is in need of it.

Symbol table are data structure that are used by compilers to hold information about source program constructs. An entry in the symbol table resembles a token. A set of predefined macros are used for the general constructs of the language. The symbol table is a linked list that has 2 visible fields. One is the token ID that specifies what type of lexeme it is storing n the other is the attribute or the lexeme itself. The last field is a pointer to the next entry of the symbol table. Our symbol table is divided into two parts:

- First part consists of the keywords or reserve words of our language. It also includes all the ASCII symbols and operators. It is initialized at start by a call to installation of keywords. This part is fixed and unchangeable at all times.
- Second part consists of all the user defined identifiers. Insertion of these identifiers into the symbol table starts from IDHEAD. On arrival of a new lexeme, it is looked up in the symbol table for its entry. If that particular entry is found, the corresponding token is returned otherwise a new entry in the symbol table is created and token is returned. During lookup, the entire symbol table needs to be searched and in the case of a new entry traversal is made from IDHEAD.

Some symbol table entries are as shown bellow.

id	**attribute**
INT	260
FL	261
CH	262
IN	268
IF	270
EL	271
FR	275
EQ	279
NE	280

8 Implementation of *lexer*

The lexer keeps global variables to keep track of its input. It uses a file pointer, that is initialized to the start of the source program. A variable 'ch' gets the characters from the input stream using the file pointer. Also the

maximum size of the lexeme is fixed to the macro value MAX_SIZE. Finally, the value in the macro IDHEAD is used to initialize the symbol table, from which the entries of the identifiers into it starts.

The lexer maintains a temporary buffer into which the input stream is written. It is then checked with the symbol table for any entry in the same name. If so, the symbol table returns a token from its entry else a new entry is made and that token is returned. The buffer is discarded at the time of returning the token to the parser.

The following comparisions are done for getting the lexemes:

- First a check is made for the whitespaces. If any whitespace is found it is suitably discarded and the lexer starts scanning its next input. Only in the case of a newline, the lexer increments the global newline count by one.
- If the current character (ch) is a delimiter, then it is checked for the comments and relational operators. The automata described above for relational operators is used. A lookahead character (lach) is kept that points one character ahead of the current character. The following situations may arise depending on the contents of 'ch' and 'lach':

  - If ch = '/' and lach '=' *: It indicates the existence of a comment. Therefore, the comment continues till '*/' and hence everything scanned is discarded until ch = '*' and lach = '/'.
  - if ch = '=' and lach = '=': A token for the relational operator '==' is returned to the parser.
  - if ch = '<' and lach = '=': A token for the relational operator '<=' is returned to the parser. If lach is anything other than '=', then a token for the relational operator '<' is returned to the parser and one character from input stream, that is, lach is retracted back as per the automata (denoted by a * in automata for retraction).
  - if ch = '>' and lach = '=': A token for the relational operator '>=' is returned to the parser. If lach is anything other than '=', then a token for the relational operator '>' is returned to the parser and lach is retracted back as per the automata.
  - if ch = '!' and lach = '=': A token for the relational operator '!=' is returned to the parser.

- The character is now checked for the existence of any of the operators (+, -, *, /, {, }, (,), ;, , , =) and a token corresponding to the operator is retracted back.

- The lexer now for numbers as per the automata of numbers. If the current character is within 0-9, a flag is set saying it as an integer number. Successive numbers are entered into a buffer. If the current character becomes '.', and if next character is a digit, then it is a float number and the flag is set to float. Successive numbers are entered into the buffer. But if the next character is not a digit, then its an error and lexing is continued from first. Also, if the current character is not a '.' and not a delimiter, then it is an error. Finally, if no error occurs, a token is returned to the parser, depending on the flag value.

- The lexer checks for identifiers. If the current character is within a-z or A-Z and not a digit, then it is the start of an identifier. A new character is scanned everytime and filled into a buffer until the new character is not a delimiter. If the current character is '.', then it is an error, as no dots can appear in an identifier of a C subset language. Finally, a pointer is retracted. The identifier is checked in the symbol table for its entry. If there is no entry in the symbol table, a new entry is made and a token for the identifier is returned to the parser otherwise a token from the existing symbol table entry is returned to the parser.

9 Implementation of *Parser*

The parser uses the first components of the tokens produced by the lexical analyzer to create the tree like intermediate representation that depicts the grammatical structure if the token stream. A recursive descent top-down parsing technique is followed to parse the input.

The function *void match (char *string)* matches the parameter with the next lexeme. If it does not match, an explicit error handling is done by printing the required 'string' from the grammar.

The parser consists of a global lookahead token. It is initialised at start by a call to lexer, that returns a token. Further updations of the lookahead token is done in the call to 'match', where the current lexeme expected from the grammar is checked with the lexeme in the input string, that is held in the lookahead token. After a successful match, the lookahead is updated with a new token by a call to the lexer. This process continues till the end of file is reached.

A grammar like
A → aB
B → b
is implemented using function call for each of the non-terminals. The terminals defined in the FIRST (α), like FIRST (A) = a, guide the parsing actions, that is, depending on the current terminal symbol in the FIRST (α) leads to different paths or productions during parsing. The above grammar is implemented from the start function S as,

```
S ()
{
A (); /* A function call to non-terminal A */
}

A ()
{
match ("a"); /* A match is made for the terminal 'a' */
B (); /* A function call to non-terminal B */
}

B ()
{
match ("b"); /* A similar match for terminal 'b' */
}
```

This implementation results in an infinite recursion if the grammar is left recursive, that is a production calling its own production at start like A → AB results in an infinite recursion. Also if two or more productions have the same starting terminals, then it needs to be factored as the starting terminals or the FIRST (α) guides the parsing actions. Therefore, grammars that are implemented should be left recursion eliminated and left factored. Such an implementation is done during our parsing and the grammar used is shown in the section of *Grammars* at last.

Finally, a parse tree is generated alongwith parsing by constructing tree nodes explicitly while the 'match' is done for a lexeme.

10 Example program

Consider the example program *test1.cs*.

```
include "Hello"

int main (12, a)

{
    if (a == b)
    {
        S = ed + c;
    }
    else
    {
        x = y;
    }
    for(i=0;i<=100;i = i +1;)
    {
        print();
    }
}
```

Figure 4: Example program

The above program involves both *if-else* and *for-loop* constructs. Where *int main()* considered as a function. We have *Makefile* to create **a.out**.After executing *make* we can call a.out with program as the parameter.

```
prabhakar@prabhakar-laptop:~/Desktop/handwrittenLexParse-Updated$ ./a.out test1.cs
line 7: parse.c: ifGrammar matched...
line 11: parse.c: elseGrammar matched...
End of token stream...
prabhakar@prabhakar-laptop:~/Desktop/handwrittenLexParse-Updated$
```

Figure 5: Output on the terminal after parsing.

Once *a.out* is executed the file *dotty.dot* is created which can be used as input to *dotty* or *kgraphviewer* to view the explicit parse tree. The parse tree for the *test1.cs* is as shown below.

11 Parse Tree

The parse tree pictorially shows how the start symbol of the grammar derives a string in the language. A typical parse tree for our language is as given in the figure of typical Parse Tree.

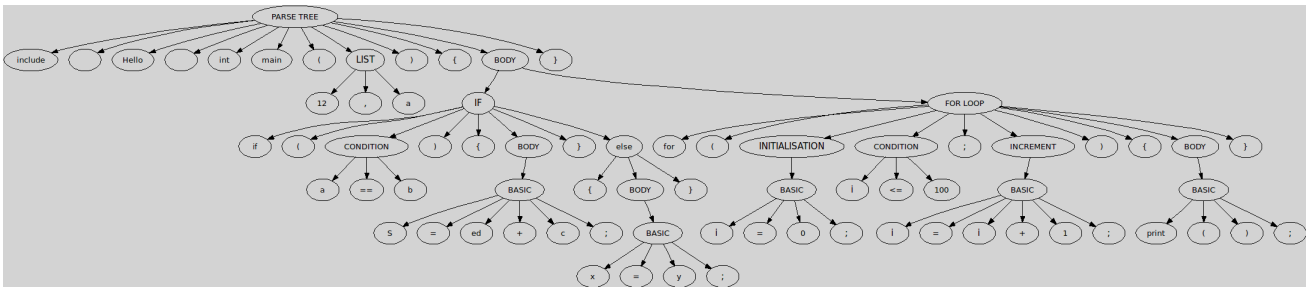


Figure 6: Parse tree for the program test1.cs

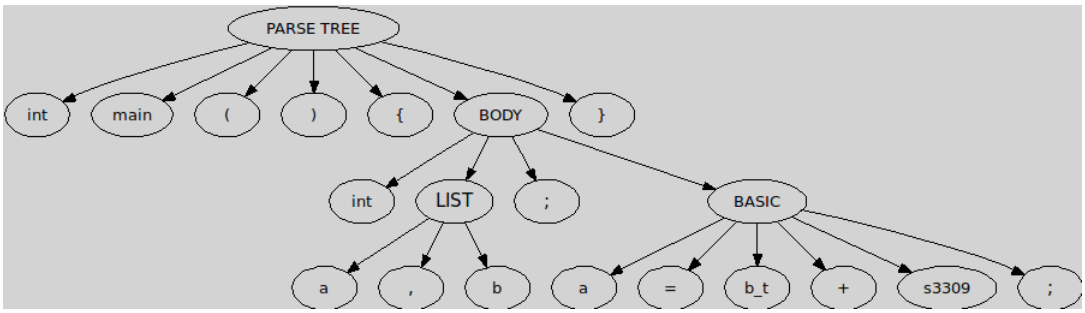


Figure 7: Typical Parse Tree.

12 Grammars

The following are the left recursion eliminated, left factored grammars used in the implementation. The first production is the program start. All productions are self explanatory and all non-terminals are capitalized.

S :
PROGRAM

PROGRAM :
INCLUDE FUNCTIONS

INCLUDE :
include ”IDENTIFIER” INCLUDE | ϵ

FUNCTIONS :
KEYWORD IDENTIFIER (LIST) BODY FUNCTIONS | ϵ

BODY :
{ STATEMENT } BODY | ϵ

DECLARATION :
KEYWORD LIST ; DECLARATION | ϵ

STATEMENT :
IF | FOR | STATEMENT'

STATEMENT' :
BASICSTATEMENT | ϵ

FOR :
for (BASICSTATEMENT; CONDITION; BASICSTATEMENT) BODY

IF :
if CONDITION BODY ELSE
ELSE :
else BODY

BASICSTATEMENT :
IDENTIFIER = EXPR ; BASICSTATEMENT
| IDENTIFIER (LIST) ; BASICSTATEMENT | ϵ | ;

CONDITION :
IDENTIFIER RELCOND IDENTIFIER

KEYWORD :
int | float | char | ϵ

LIST :
IDENTIFIER SUBLIST | ϵ
SUBLIST :
, IDENTIFIER SUBLIST | ϵ

EXPR :
TERM SUBEXPR
SUBEXPR :
+ TERM SUBEXPR | - TERM SUBEXPR | ϵ
TERM :

FACTOR SUBTERM
SUBTERM :
* FACTOR SUBTERM | / FACTOR SUBTERM | ϵ
FACTOR :
(EXPR) | IDENTIFIER

RELCOND :
IDENTIFIER < IDENTIFIER
| IDENTIFIER <= IDENTIFIER
| IDENTIFIER > IDENTIFIER
| IDENTIFIER >= IDENTIFIER
| IDENTIFIER == IDENTIFIER
| IDENTIFIER != IDENTIFIER
| IDENTIFIER

IDENTIFIER :
id

13 References

- Lex and Yacc by John R. Levine, Tony Mason, Doug Brown - OReilly Publications
- A Compact Guide to Lex and Yacc by Tom Niemann at epaperpress.com
- Compilers Principles Techniques and Tools by Afred V.Aho, Monica S.lam, Ravi Sethi,Jeffrey D.Ullman
- Wikipaedia