Syntax Analyser for C

Aishwarya Rajan - 13CO109
Anagh Singh - 13CO110

February 22, 2016

Department of Computer Science and Engineering, NITK

Abstract

Syntax analysis or parsing is the second phase of a compiler.

A lexical analyzer can identify tokens with the help of regular expressions and pattern rules. But a lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions. Regular expressions cannot check balancing tokens, such as parenthesis. Therefore, this phase uses context-free grammar (CFG), which is recognized by push-down automata. It is roughly the equivalent of checking that some ordinary text written in a natural language (e.g. English) is grammatically correct (without worrying about meaning).

The purpose of syntax analysis or parsing is to check that we have a valid sequence of tokens. Tokens are valid sequence of symbols, keywords, identifiers etc. Note that this sequence need not be meaningful; as far as syntax goes, a phrase such as "true + 3" is valid but it doesn't make any sense in most programming languages.

Contents

List of Figures

Listings

1 Introduction

The purpose of syntax analysis or parsing is to check that we have a valid sequence of tokens. Tokens are valid sequence of symbols, keywords, identifiers etc. Note that this sequence need not be meaningful; as far as syntax goes, a phrase such as "true + 3" is valid but it doesn't make any sense in most programming languages.

The parser takes the tokens produced during the lexical analysis stage, and attempts to build some kind of in-memory structure to represent that input. Frequently, that structure is an "abstract syntax tree" (AST).
The parser needs to be able to handle the infinite number of possible valid programs that may be presented to it. The usual way to define the language is to specify a grammar. A grammar is a set of rules (or productions) that specifies the syntax of the language (i.e. what is a valid sentence in the language).
There can be more than one grammar for a given language. Furthermore, it is easier to build parsers for some grammars than for others.

1.1 Syntactical Analysis

1.1.1 Definition

Syntactic analysis, or parsing, is needed to determine if the series of tokens given are appropriate in a language - that is, whether or not the sentence has the right shape/form. However, not all syntactically valid sentences are meaningful, further semantic analysis has to be applied for this. For syntactic analysis, context-free grammars and the associated parsing techniques are powerful enough to be used - this overall process is called parsing.

1.1.2 Tree Usage

In syntactic analysis, parse trees are used to show the structure of the sentence, but they often contain redundant information due to implicit definitions (e.g., an assignment always has an assignment operator in it, so we can imply that), so syntax trees, which are compact representations are used instead. Trees are recursive structures, which complement CFGs nicely, as these are also recursive (unlike regular expressions).

There are many techniques for parsing algorithms (vs FSA-centred lexical analysis), and the two main classes of algorithms are top-down and bottom-up parsing.

1.1.3 Representation

Context-free grammars can be represented using Backus-Naur Form (BNF). BNF uses three classes of symbols: non-terminal symbols (phrases) enclosed by brackets $\langle()\rangle$, terminal symbols (tokens) that stand for themselves, and the metasymbol ::= - is defined to be.

As derivations are ambiguous, a more abstract structure is needed. Parse trees generalise derivations and provide structural information needed by the later stages of compilation.

1.1.4 Parse Trees

Parse trees over a grammar G is a labelled tree with a root node labelled with the start symbol (S), and then internal nodes labelled with non-terminals. Leaf nodes are labelled with terminals or . If an internal node is labelled with a non-terminal A, and has n children with labels $X_1, ..., X_n$ (terminals or non-terminals), then we can say that there is a grammar rule of the form $A \rightarrow X_1 ... X_n$. Parse trees also have optional node numbers.

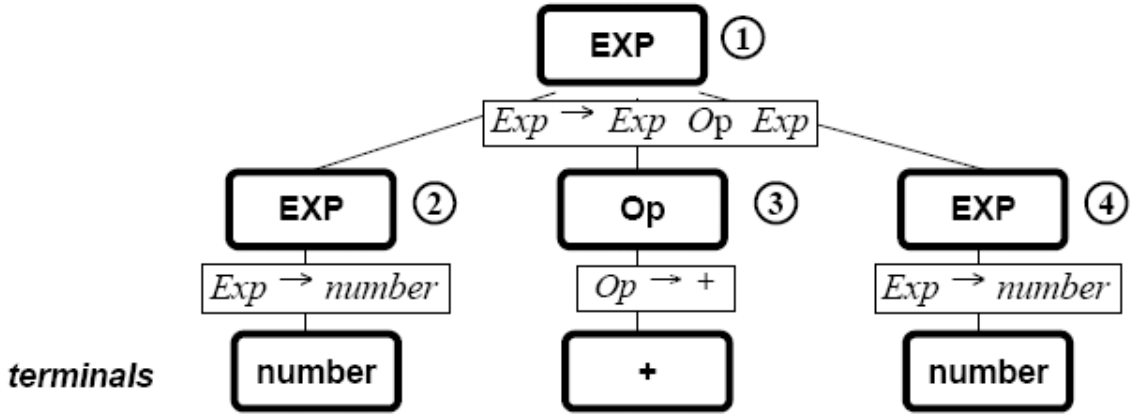

Figure 1: Parse Tree for Leftmost Derivation

1.1.5 Traversal of the tree

Traversing the tree can be done by three different forms of traversal. In preorder traversal, you visit the root and then do a preorder traversal of each of the children, in in-order traversal, an in-order traversal is done of the left sub-tree, the root is visited, and then an in-order traversal is done of the remaining subtrees. Finally, with postorder traversal, a postorder traversal is done of each of the children and the root visited.

1.1.6 Syntax Trees

Parse trees are often converted into a simplified form known as a syntax tree that eliminates wasteful information from the parse tree.
At this stage, treatment of errors is more difficult than in the scanner (tokeniser), as the scanner may pass problems to the parser (an error token). Error recovery typically isolates the error and continues parsing, and repair can be possible in simple cases. Generating meaningful error messages is important, however this can be difficult as the actual error may be far behind the current input token.

1.1.7 Ambiguous Grammars

Grammars are ambiguous if there exists a sentence with two different parse trees - this is particularly common in arithmetic expressions - does $3 + 4 \times 5 = 23$(the natural interpretation, that is

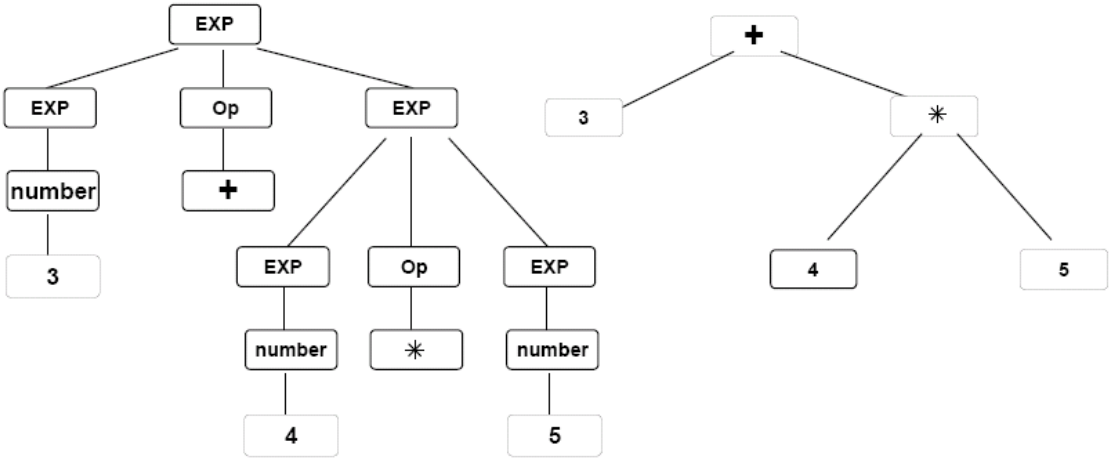

Figure 2: Parse Tree to Syntax Tree

$3+(4\times5))or35((3+4)\times5)$. Ambiguity can be inessential, where the parse trees both give the same syntax tree (typical for associative operations), but still a disambiguation rule is required. However, there is no computable function to remove ambiguity from a grammar, it has to be done by hand, and the ambiguity problem is not decidable. Solutions to this include changing the grammar of the language to remove any ambiguity, or to introduce disambiguation rules.

A common disambiguation rule is precedence, where a precedence cascade is used to group operators in to different precedence levels.

1.1.8 Denoting Grammars - BNF

BNF does have limitations as there is no notation for repetition and optionality, nor is there any nesting of alternatives - auxiliary non-terminals are used instead, or a rule is expanded. Each rule also has no explicit terminator.

1.1.9 Extended BNF

Extended BNF (EBNF) was developed to work around these restrictions.

- In EBNF, terminals are in double quotes "", and non-terminals are written without $\langle\rangle$.
- Rules have the form non-terminal = sequence or non-terminal = sequence |...| sequence, and a period "." is used as the terminator for each rule.
- p is used for 0 more occurrences of p, $[p]$ stands for 0 or 1 occurances of p and $(p|q|r)$ stands for exactly one of p, q, or r.
- However, notation for repetition does not fully specify the parse tree (left or right associativity isn't specified).

Syntax diagrams can be used for graphical representations of EBNF rules. Non-terminals are represented in a square/rectangular box, and terminals in round/oval boxes. Sequences and choices are represented by arrows, and the whole diagram is labelled with the left-hand side non-terminal. e.g., for

$$factor \rightarrow (exp) | \text{“}number\text{”}.$$

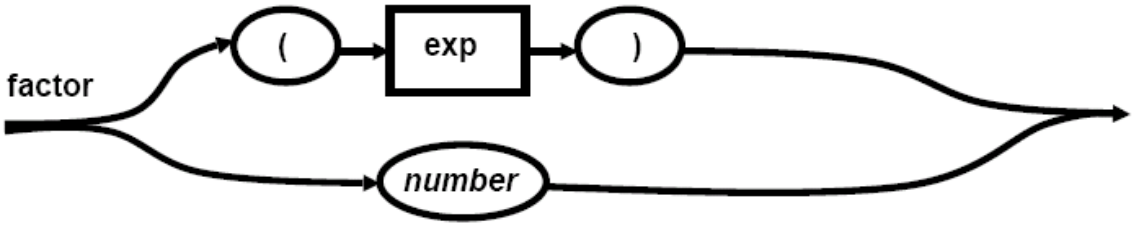

Figure 3: Factoring

1.2 Top Down Parsing

Top down parsing can be broken down into two classes: backtracking parsers, which try to apply a grammar rule and backtrack if it fails; and predictive parsers, which try to predict the next nonterminal in the input using one or more tokens lookahead. Backtracking parsers can handle a larger class of grammars, but predictive parsers are faster. We consider only predictive parsers here.

There are two classes of predictive parsers:

- Recursive-descent parsers, which are quite versatile and appropriate for a hand-written parser, and were the first type of parser to be developed;
- LL(1) parsing - left-to-right, leftmost derivation, 1 symbol lookahead, a type of parser which is no longer in use. It does demonstrate parsing using a stack, and helps to formalise the problems with recursive descent. There is a more general class of LL(k) parsers.

1.2.1 Recursive Descent

The basic idea here is that each nonterminal is recognised by a procedure. This choice corresponds to a case or if statement. Strings of terminals and nonterminals within a choice match the input and calls to other procedures. Recursive descent has a one token lookahead, from which the choice of appropriate matching procedure is made.

The code for recursive descent follows the EBNF form of the grammar rule, however the procedures must not use left recursion, which will cause the choice to call a matching procedure in an infinite loop.

There are problems with recursive descent, such as converting BNF rules to EBNF, ambiguity

in choice and empty productions (must look on further at tokens that can appear after the current non-terminal). We could also consider adding some form of early error detection, so time is not wasted deriving non-terminals if the lookahead token is not a legal symbol.

Therefore, we can improve recursive descent by defining a set $First(\alpha)$ as the set of tokens that can legally begin α, and a set Follow(A) as the set of tokens that can legally come after the nonterminal A. We can then use $First(\alpha)$ and $First(\beta)$ to decide between $A \to \alpha$ and $A \to \beta$, and then use Follow(A) to decide whether or not $A \to \epsilon$ can be applied.

1.2.2 LL(1) Parsing

LL(1) parsing is top-down parsing using a stack as the memory. At the beginning, the start symbol is put onto the stack, and then two basic actions are available: Generate, which replaces a non-terminal A at the top of the stack by string using the grammar rule $A \to \alpha$; and Match, which matches a token on the top of the stack with the next input token (and, in case of success, pop both). The appropriate action is selected using a parsing table.

With the LL(1) parsing table, when a token is at the top of the stack, there is either a successful match or an error (no ambiguity). When there is a nonterminal A at the top, a lookahead is used to choose a production to replace A.

A grammar is an LL(1) grammar if the associate LL(1) parsing table has at most one production in each table entry. A LL(1) grammar is never ambiguous, so if a grammar is ambiguous, disambiguating rules can be used in simple cases. Some consequences of this are for rules of the form $A \to \alpha|\beta$, then α and β can not derive strings beginning with the same token a and at most one of α or β can derive ϵ.

With LL(1) parsing, the list of generated actions match the steps in a leftmost derivation. Each node can be constructed as terminals, or nonterminals are pushed onto the stack. If the parser is not used as a recogniser, then the stack should contain pointers to tree nodes. LL(1) parsing has to generate a syntax tree.

In LL(1) parsing, left recursion and choice are still problematic in similar ways to that of recursive descent parsing. EBNF will not help, so it's easier to keep the grammar as BNF. Instead, two techniques can be used: left recursion removal and left factoring. This is not a foolproof solution, but it usually helps and it can be automated.

To construct a syntax tree in LL(1) parsing, it takes an extra stack to manipulate the syntax tree nodes. Additional symbols are needed in the grammar rules to provide synchronisation between the two stacks, and the usual way of doing this is using the hash (#). In LL(1) parsers, sets of synchronising tokens are kept in an additional stack or built directly into the parsing table.

1.3 Bottom Up Parsing

Top-down parsing works by tracing out the leftmost derivations, whereas bottom-up parsing works by doing a reverse rightmost derivation.

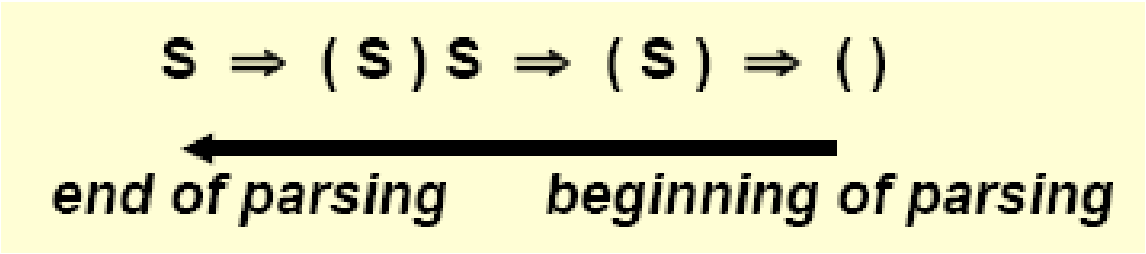

Figure 4: Reverse Rightmost Derivation

The right sentential form are the rightmost derivations of a string, and a parse of a string eats tokens left to right to reverse the rightmost derivation. A handle is a string defined by the expansion of a non-terminal in right sentential form.

Bottom-up parsing works by starting with an empty stack and having two operations: shift, putting the next input token onto the stack; and reduce, replacing the right-hand side of a grammar rule with its left-hand side. We augment the start state S with a rule $Z \rightarrow S$ (some books refer to this as $S' \rightarrow S$), making Z our new start symbol, so when only Z is on the stack and the input is empty, we accept. The parse stack is maintained where tokens are shifted onto it until we have a handle on top of the stack, whereupon we shall reduce it by reversing the expansion.

In this process, we can observe that when a rightmost non-terminal is expanded, the terminal to the right of it is in its Follow set - we need this information later on, and is also important to note that the Follow set can include $.

At any moment, a right sentential form is split between the stack and the input; each reduce action produces the next right sentential form. Given $A \rightarrow \alpha$, α is reduced to A if the stack content is a viable prefix of the right sentential form. Appropriate handles are at the base of the expansion triangles in the following right sentential form.

1.3.1 LR(0)

An LR(0) item is a way of monitoring progress towards a handle. It is represented by a production rule together with a dot. The right-hand side has part behind the dot and part in front. This says that the parse has matched a substring derivable by the component to the left of the dot and is now prepared to match something on the input stream defined by that component to the right of the dot. A production rule can therefore give rise to several configuration items, with the dots in varying places. When the dot is to the extreme left, we call this an initial item, and when the dot is to the extreme right, we call this a completed item, and these are typically underlined.

LR(0) parsing belongs to a more general class of parsers, called LR parsers. NDFAs are built to describe parsing in progressively increasing levels of detail (similar to recursive descent parsing). The DFA, built from subset construction, builds the parse tree in a bottom-up fashion and waits for a complete right-hand side with several right-hand sides sharing a prefix considered in parallel. A stack is used to store partial right-hand sides (the handles) and remember visited states.

The right-hand side is traced following arrows (going from left-hand side to right-hand side), and then removed from the stack (going against the arrows). The uncovered state and left-hand side non-terminal then define the new state.

1.3.2 LALR(1)

The class of languages that LALR(1) can parse is between that of LR(1) and SLR(1), and in reality is enough to implement most of the languages available today. It uses a DFA based on LR(0) items and sets of look-aheads which are often proper subsets of the Follow sets allowed by the SLR(1) parser. The LALR(1) parsing tables are often more compact than the LR(1) ones.

We define the core of a state in an LR(1) DFA as the set of all LR(1) items reduced to LR(0) by dropping the lookahead.

The LALR(1) algorithm is the same as the LR(1) algorithm, and an LALR(1) grammar leads to unambiguous LALR(1) parsing. If the grammar is at least LR(1), then the LALR(1) parser can not contain any shift-reduce conflicts. The LALR(1) parser could be less efficient than the LR(1) parser, however - as it makes additional reductions before throwing error.

Another technique to construct the LALR(1) DFA using the LR(0) DFA instead of the LR(1) DFA is known as propagating lookaheads.

1.4 Error Recovery

Similarly to LL(1) parsers, there are three possible actions for error recovery:

- Pop a state from the stack
- Pop tokens from the input until a good one is seen (for which the parse can continue)
- Push a new state onto the stack

A good method is to pop states from the stack until a nonempty Goto entry is found, then, if there is a Goto state that matches the input, go - if there is a choice, prefer shift to reduce, and out of many reductions, choose the most specific one. Otherwise, scan the input until at least one of the Goto states like the token or the input is empty.

This could lead to infinite loops, however. To avoid loops, we can only allow shift actions from a Goto state, or, if a reduce action Rx is applied, set a flag and store the following sequence of states produced purely by reduction. If a loop occurs (that is, Rx is produced again), states are popped from the stack until the first occurence of Rx is removed, and a shift action immediately resets the flag.

The method Yacc uses for error recovery is $NT_i \rightarrow error$. The Goto entries of NT_i will then be used for error recovery. In case of an error, states are popped from the stack until one of NT_i is seen. error is then added to the input as its first token, and you continue going as if nothing has happened (or choose to discard the original lookahead symbol). All erroneous input tokens are discarded without producing error messages until a sequence of 3 tokens is shifted legally onto the stack.

1.5 The YACC Script

Yacc is a computer program for the Unix operating system. It is a LALR parser generator, generating a parser, the part of a compiler that tries to make syntactic sense of the source code, specifically a LALR parser, based on an analytic grammar written in a notation similar to BNF. Yacc itself used to be available as the default parser generator on most Unix systems, though it has since been supplanted as the default by more recent, largely compatible, programs.

YACC is an acronym for "Yet Another Compiler Compiler". It is a LALR parser generator, generating a parser, the part of a compiler that tries to make syntactic sense of the source code, specifically a LALR parser, based on an analytic grammar written in a notation similar to BNF.

The input to Yacc is a grammar with snippets of C code (called "actions") attached to its rules. Its output is a shift-reduce parser in C that executes the C snippets associated with each rule as soon as the rule is recognized. Typical actions involve the construction of parse trees.

Yacc produces only a parser (phrase analyzer); for full syntactic analysis this requires an external lexical analyzer to perform the first tokenization stage (word analysis), which is then followed by the parsing stage proper. Lexical analyzer generators, such as Lex or Flex are widely available. The IEEE POSIX P1003.2 standard defines the functionality and requirements for both Lex and Yacc.

1.5.1 Basic Specifications

Names refer to either tokens or nonterminal symbols. Yacc requires token names to be declared as such. In addition, for reasons discussed in Section 3, it is often desirable to include the lexical analyzer as part of the specification file; it may be useful to include other programs as well. Thus, every specification file consists of three sections: the declarations, (grammar) rules, and programs. The sections are separated by double percent "%%" marks. (The percent "%" is generally used in Yacc specifications as an escape character.)
In other words, a full specification file looks like

```
declarations
%%
rules
%%
programs
```

The declaration section may be empty. Moreover, if the programs section is omitted, the second thus, the smallest legal Yacc specification is

```
%%
rules
```

Blanks, tabs, and newlines are ignored except that they may not appear in names or multi-character reserved symbols. Comments may appear wherever a name is legal; they are enclosed in /* . . . */, as in C and PL/I.

1.5.2 Actions

With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.

An action is an arbitrary C statement, and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements, enclosed in curly braces "" and "". For example,

```
A       :       '('  B  ')'
                        {       hello( 1, "abc" );  }
```

and

```
XXX     :       YYY  ZZZ
                        {       printf("a message\n");
                                flag = 25;   }
```

are grammar rules with actions.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks "%" and "%". These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example,

```
\%{   int variable = 0;   \%}
```

could be placed in the declarations section, making variable accessible to all of the actions. The Yacc parser uses only names beginning in "yy"; we should avoid such names.

1.5.3 Lexical Analysis

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called *yylex*. The function returns an integer, the token number, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable *yylval*.

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by Yacc, or chosen by the user. In either case, the "# define" mechanism of C is used to allow the lexical analyzer to return these numbers symbolically.

To assign a token number to a token (including literals), the first appearance of the token name or literal in the declarations section can be immediately followed by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined by this mechanism retain their default definition. It is important that all token numbers be distinct.

For historical reasons, the endmarker must have token number 0 or negative. This token number cannot be redefined by the user; thus, all lexical analyzers should be prepared to return 0 or negative as a token number upon reaching the end of their input.

1.5.4 Parsing

Yacc turns the specification file into a C program, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex, and will not be discussed here (see the references for more information). The parser itself, however, is relatively simple, and understanding how it works, while not strictly necessary, will nevertheless make treatment of error recovery and ambiguities much more comprehensible.

The parser produced by Yacc consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the lookahead token). The current state is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read.

The machine has only four actions available to it, called shift, reduce, accept, and error. A move of the parser is done as follows:

- Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls yylex to obtain the next token.
- Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off of the stack, and in the lookahead token being processed or left alone.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack, running in parallel with it, holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable yylval is copied onto the value stack. After the return from the user code, the reduction is carried out. When the goto action is done, the external variable yyval is copied onto the value stack. The pseudo-variables $1, $2, etc., refer to the value stack.

The other two parser actions are conceptually much simpler. The accept action indicates that the entire input has been seen and that it matches the specification. This action appears only when the lookahead token is the endmarker, and indicates that the parser has successfully done its job. The error action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen, together with the lookahead token, cannot be followed by anything that would result in a legal input.

1.5.5 Ambiguity and Conflicts

A set of grammar rules is ambiguous if there is some input string that can be structured in two or more different ways. For example, the grammar rule

```
expr    :       expr  '-'  expr
```

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule

does not completely specify the way that all complex inputs should be structured. For example, if the input is

```
expr - expr - expr
```

the rule allows this input to be structured as either

```
( expr - expr ) - expr
```

or as

```
expr - ( expr - expr )
```

When there are shift/reduce or reduce/reduce conflicts, Yacc still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing which choice to make in a given situation is called a disambiguating rule.

Yacc invokes two disambiguating rules by default:

- In a shift/reduce conflict, the default is to do the shift.
- In a reduce/reduce conflict, the default is to reduce by the earlier grammar rule (in the input sequence).

Rule 1 implies that reductions are deferred whenever there is a choice, in favor of shifts. Rule 2 gives the user rather crude control over the behavior of the parser in this situation, but reduce/reduce conflicts should be avoided whenever possible.

1.5.6 Precedence

There is one common situation where the rules given above for resolving conflicts are not sufficient; this is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of precedence levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the form

```
expr : expr OP expr
```

and

```
expr : UNARY expr
```

for all binary and unary operators desired. This creates a very ambiguous grammar, with many parsing conflicts. As disambiguating rules, the user specifies the precedence, or binding strength, of all the operators, and the associativity of the binary operators. This information is sufficient to allow Yacc to resolve the parsing conflicts in accordance with these rules, and construct a parser that realizes the desired precedences and associativities.

The precedences and associativities are attached to tokens in the declarations section. This is

done by a series of lines beginning with a Yacc keyword: %left, %right, or %nonassoc, followed by a list of tokens. All of the tokens on the same line

are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence or binding strength. Thus,

```
\%left  '+'  '-'
\%left  '*'  '/'
```

describes the precedence and associativity of the four arithmetic operators. Plus and minus are left associative, and have lower precedence than star and slash, which are also left associative.

1.5.7 Error Handling

Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, for example, it may be necessary to reclaim parse tree storage, delete or alter symbol table entries, and, typically, set switches to avoid generating any further output. It is seldom acceptable to stop all processing when an error is found; it is more useful to continue scanning the input to find further syntax errors. This leads to the problem of getting the parser "restarted" after an error. A general class of algorithms to do this involves discarding a number of tokens from the input string, and attempting to adjust the parser so that input can continue.

To allow the user some control over this process, Yacc provides a simple, but reasonably general, feature. The token name "error" is reserved for error handling. This name can be used in grammar rules; in effect, it suggests places where errors are expected, and recovery might take place. The parser pops its stack until it enters a state where the token "error" is legal. It then behaves as if the token "error" were the current lookahead token, and performs the action encountered. The lookahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

In order to prevent a cascade of error messages, the parser, after detecting an error, remains in error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted. Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc

1.5.8 YACC Environment

When the user inputs a specification to Yacc, the output is a file of C programs, called y.tab.c on most systems (due to local file system conventions, the names may differ from installation to installation). The function produced by Yacc is called yyparse; it is an integer valued function. When it is called, it in turn repeatedly calls yylex, the lexical analyzer supplied by the user to obtain input tokens. Eventually, either an error is detected, in which case (if no error recovery is possible) yyparse returns the value 1, or the lexical analyzer returns the endmarker token and the parser accepts. In this case, yyparse returns the value 0.

The user must provide a certain amount of environment for this parser in order to obtain a working program. For example, as with every C program, a program called main must be defined, that

eventually calls yyparse. In addition, a routine called yyerror prints a message when a syntax error is detected.

The argument to yyerror is a string containing an error message, usually the string "syntax error". The average application will want to do better than this. Ordinarily, the program should keep track of the input line number, and print it along with the message when a syntax error is detected. The external integer variable yychar contains the lookahead token number at the time the error was detected; this may be of some interest in giving better diagnostics. Since the main program is probably supplied by the user (to read arguments, etc.) the Yacc library is useful only in small projects, or in the earliest stages of larger ones.

The external integer variable yydebug is normally set to 0. If it is set to a nonzero value, the parser will output a verbose description of its actions, including a discussion of which input symbols have been read, and what the parser actions are. Depending on the operating environment, it may be possible to set this variable by using a debugging system.

1.6 C Program

This section contains the C progam which we provide as input to our scanner.

This is another program that we provide as input:

2 Design of Program

2.1 Code

This section contains our actual scanner. The code for it is:

2.2 Explanation

Given lex.y file is code for our lexical analyser which is takes a C program as input and scan and analyses that code and works as given below.

2.2.1 Alphabet keywords

They are already predefined in their respective section.

2.2.2 To keep track of code blocks

This will take a section which is between/before or after comments (excludes comments).

2.2.3 To resolve nested comments

We resolved this situtation by keeping integer track of the depth of comments.

2.2.4 String errors

These are solved by checking last character of given string to check if it has ended or not.

3 Test Cases

S.No	Test case	Steps	Expected Results	Status
1	Identifier	Enter an identifier Enter an identifier Enter an identifier Enter an identifier	Var1 VALID IDENTIFIER 1dr INVALID IDENTIFIER _var VALID IDENTIFIER V1:b INVALID IDENTIFIER	Working
2	Integer Constant(int)	Enter an integer Enter an integer Enter an integer Enter an integer Enter an integer Enter an integer	3 VALID INTEGER 2.0 INVALID INTEGER 23,333 INVALID INTEGER -55 VALID INTEGER +10 VALID INTEGER 12938365675 INVALID INTEGER 09 INVALID INTEGER	Working
3	String Constant	Enter a string Enter a string Enter a string Enter a string	Horse VALID STRING Horse INVALID STRING Horse INVALID STRING 100, Rs VALID STRING	Working
4	Character Constant	Enter a character Enter a character Enter a character Enter a character Enter a character Enter a character Enter a character	'A' VALID CHARACTER n INVALID CHARACTER 'kg' INVALID CHARACTER ''VALID CHARACTER 'VALID CHARACTER '*iN*' INVALID CHARACTER '9' VALID CHARACTER	Working
5	Relational Operators	Enter an R-operator Enter an R-operator Enter an R-operator Enter an R-operator Enter an R-operator Enter an R-operator	>VALID R-OPERATOR >= VALID R-OPERATOR = INVALID R-OPERATOR == VALID R-OPERATOR != VALID R-OPERATOR - INVALID R-OPERATOR	Working
6	Real Constant	Enter a real constant Enter a real constant Enter a real constant Enter a real constant Enter a real constant Enter a real constant Enter a real constant Enter a real constant Enter a real constant	5.6 VALID CONSTANT 3.4.5 INVALID CONSTANT 34 INVALID CONSTANT 34 456 INVALID CONSTANT 34,477 INVALID CONSTANT -364.8 VALID CONSTANT +ie INVALID CONSTANT +3.0 VALID CONSTANT 314159E-5L VALID CONSTANT	Working

7	Keywords	Enter a keyword Enter a keyword Enter a keyword Enter a keyword Enter a keyword Enter a keyword Enter a keyword	double VALID KEYWORD main INVALID KEYWORD case VALID KEYWORD INT INVALID KEYWORD while VALID KEYWORD yes! INVALID KEYWORD auto VALID KEYWORD	Working
8	Nested Comments	Enter a comment Enter a comment Enter a comment	/*hello /*world*/ VALID /*hi /*hello*/*/ INVALID /*yes*/*/ INVALID	
9	Delimiters	Enter a delimiter Enter a delimiter Enter a delimiter Enter a delimiter Enter a delimiter Enter a delimiter Enter a delimiter Enter a delimiter Enter a delimiter	{ VALID ; VALID : VALID , VALID # VALID [VALID) VALID INVALID . INVALID	Working
10	Unary Operators	Enter a U-operator Enter a U-operator Enter a U-operator Enter a U-operator Enter a U-operator Enter a U-operator Enter a U-operator Enter a U-operator Enter a U-operator	++ VALID + VALID sizeof VALID * VALID & VALID VALID ! VALID / INVALID . INVALID	Working
11	Binary Operators	Enter a B-Operator Enter a B-Operator Enter a B-Operator Enter a B-Operator Enter a B-Operator Enter a B-Operator Enter a B-Operator Enter a B-Operator	+ VALID - VALID / VALID * VALID —— VALID ++ INVALID ! INVALID { INVALID	Working
12	Ternary Operator	Enter a T-Operator Enter a T-Operator	?: VALID !: INVALID	Working

4 Testing With Errors

Figure 5: Errors in Nested Comments

5 Implementation

For implementation of this given code we used following technique:

5.1 To keep track of blocks

```
int comment=0,bracCount=0
comment (\/\/.*)  //line comment
comstart (\/\*)   //multi line comment starts
comend (\*\/)  //multi line comment ends
```

5.2 Checking for comment related errors

```
{comment}          {if(comment<=0) fprintf(yyout,"\n//Line Comment ")}
{comstart}         {comment++
                   if(comment>1)
                   fprintf(yyout,"\nERROR: Nested Comment")
                   else
                   fprintf(yyout,"\n/*ML Comment Sarts ")
          }
{comend}           {   if(comment>0)
                   {
                   comment
                   fprintf(yyout,"\n*/ML Comment Ends ")
                   }
                   else
                   fprintf(yyout,"\n\tERROR: */ found before /* ")
                   }
```

6 Results

Figure 6: Successful Run Part 2

Figure 7: Successful Run Part 2

References

[1] En.wikibooks.org, (2016). Compiler Construction/Lexical analysis - Wikibooks, open books for an open world. [online] Available at: https://en.wikibooks.org/wiki/Compiler_Construction/Syntax_analysis [Accessed 22 Feb. 2016].

[2] Cs.fsu.edu, (2016). [online] Available at: http://www.cs.fsu.edu/ xyuan/-cop5621/lect3_syntax.ppt [Accessed 28 Jan. 2016].

[3] dragonbook.stanford.edu/lecture-notes/05-Syntax-Analysis.pdf

[4] cecs.wright.edu/ tkprasad/courses/cs780/L3Lexing.pdf

[5] quex.sourceforge.net/

[6] http://www.pling.org.uk/cs/lsa.html

[7] www.cs.nyu.edu/courses/spring11/G22.2130-001/lecture6.pdf