2.1.1 **Lexical Analyzer**  
As the first phase of the compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and procedure as output a sequence of tokens of each lexeme in the source program. The stream of tokens is sent to the parser of the syntax analyzer. It is common for the lexical analyzer to interact with the symbol table as well. When the lexical analyzer discovers a lexeme constitution an identifier, it needs to enter that lexeme into the symbol table. In some cases, information regarding the kind of identifier may be read from the symbol table by the lexical analyzer to assist it in determining the proper token it must pass to the parser.  
These interactions are suggested in figure X.X commonly, the interaction is implemented by having the parser call the lexical analyzer. The call, suggested from the input by the *getNextToken* command, causes the lexical analyzer to read characters from the input stream until it can identify the next lexeme And produce for it the next token, which it returns to the parser.

Lexical Analyzer

Parser

Symbol   
Table

**Token**

*getNextToken()*

**Source program**

**To semantic analyzer**

Figure. X.x Interaction between the lexical analyzer and the parser

Since the lexical analyzer is the part of the compiler that reads the source text, it may perform certain other tasks besides identification of lexemes. One such task is striping out comments and *whitespaces* (blank, newline, tab and maybe other characters that are used to separate tokens in the input). Another task is correlating error messages generated by the compiler with the source program. For instance, the lexical analyzer may keep track of the number of newline characters seen, so it can associate a line number with each error message. In some compilers, the lexical analyzer makes a copy of the source program with the error message instead of the appropriate position. If the source program used a macro-preprocessor, the expansion of macros may also be performed by the lexical analyzer.  
Sometimes lexical analyzer is divided into a cascade of two processes.

a) *Scanning* consist of simple processes that so not require tokenization of the input, such as deletion of comments and compaction of consecutive whitespace character into one.  
  
b) *Lexical analyzer* proper is the more complex portion, which produces tokens from the output of the scanner.

2.1.2 **Lexical Analysis versus Parsing**

There are a number of reasons why the analysis portion of the compiler is normally separated into lexical analysis and parsing (syntax analysis) phases.  
1. Simplify the design is the most important consideration. The separation of lexical and syntactic analysis allows us to simplify at least one of these tasks. For example, a parser that had to deal with comments than one that can assume comments and whitespaces have already been removed by the lexical analyzer. If we are designing a new language, separating lexical and syntactic concerns can lead to a cleaner overall language design.  
2. Compiler efficiently is improved A separate lexical analyzer allow us to apply specialized techniques that serve only the lexical task, not the job of parsing. In addition specialized buffering techniques for reading input characters can speed up the compiler significantly.  
3. Compiler portability is enhanced. Input-device-specific peculiarities can be restricted to the lexical analyzer.

2.1.3 **Specification of tokens**

Regular expressions are an important notation for specifying lexeme patterns. While they cannot express all possible patterns, they are very effective in specifying those types of patterns that we actually need for tokens. In this section we shall see the formal notation for the regular expressions, and in section X.X we shall see how these expressions are used in the lexical analyzer generator.  
2.1.2.3 **Strings and Languages**

An *alphabet* is any finite of symbols. Typical examples of symbols are letters, digits and punctuations. The set {0,1} is the *binary language* ASCII is an example of an alphabet, it is used in many software systems.  
A *string* over an alphabet is a finite sequence of symbols drawn from that alphabet.  
A *language* is any countable set of strings over some fixed alphabet. This definition is very broad. Abstract languages like ÷∅, the *empty set* or {ε}, the set contains only an empty string, are languages under this definition. So the two are the set of all Well-formed C program an the set of all grammatically correct English sentences, although the later tow languages are difficult to specify exactly. Note that the definition of "language" does no require that any meaning be ascribe to the strings in the language. Methods for defining the "meaning" are described in Chapter X.X

**Operation on languages**

**??????????????????????????????????????????????????????**

**Regular expressions**

|  |  |  |
| --- | --- | --- |
| **EXPRESSION** | **MATCHES** | **EXAMPLE** |
| c | the non-operation character c | a |
| \c | character c lateral | \\* |
| "g" | string s literally | "\*\*" |
| . | any character but newline | a.\*b |
| ^ | beginning of newline | ^abc |
| $ | end of a line | abc$ |
| [s] | any one of the characters in string s | [abc] |
| [^s] | any one characters not in string s | [^abc] |
| r\* | zero or more strings matching r | a\* |
| r+ | one or more strings matching r | a+ |
| r? | zero or one r | a? |
| r{m,n} | between m and n occurrences of r | a{1,5} |
| r1r2 | an r1 followed r2 | ab |
| r1|r2 | an r1 or an r2 | a|b |
| (r) | same as r | (a|b) |
| r1/r2 | r1 when followed by r2 | abc/123 |

Figure X.X. lex regular expressions

**The Lexical-Analyzer Generator Lex**

Figure X.X suggests how Lex is used. An input file, which we call lex.l(RME.l), is written in the Lex language and describe the lexical analyzer to be generated. The Lex compiler transforms lex.l to a C program, in a file that is always named lex.yy.c the latter file is compiled by the C compiler into a file called a.out. The C-compiler output is a working lexical analyzer that can take a stream of input characters and produce a stream of tokens. The normal use of the compiled program, referred to as a/out in Fig. X.X is as subroutine of the parser. It has a C function that returns an integer, which is a code for one of the tokens name. The attribute value, whether it be another numeric code, a pointer to the symbol table. Or nothing, is placed in a global variable **yylval** which is shared between the lexical analyzer and the semantic parser, thereby making it simple to return bath the name and the attribute value of the token.

lex.l

lex.yy.c

Lex compiler

lex.yy.c

a.out

C compiler

a.out

Sequence of tokens

a.out

Figure X.X Creating a lexical analyzer with Lex.

**Structure of the Lex program**A lex program has the following form:

declarations

%%

Transactions rules

%%

Auxiliary functions

The declaration section includes declaration of variables, *manifest constant*, ( identifiers declared to stand for a constant e.g., the name of a token) and regular definitions.

The translation rules each have a form of

Pattern {action}

Each pattern is a regular expression, which may use the regular definitions of the declaration section. The actions are fragments of code, typically written in C.  
The third section holds whatever additional functions are used in the action. Alternatively, thesr functions can be compiled separately and loaded whit the lexical analyzer.

The lexical analyzer created by the Lex behaves in concert with the parser as follows. When called by the parser, lexical analyzer begins reading its remaining input, one character at a time, until it finds the longest *prefix* of the input that matches one of the patterns Pi It the then executes the associated action Ai. Typically, Ai will return to the parser, but if it does not (e.g., because Pi describes whitespaces or comments), then the lexical analyzer proceeds to find additional lexemes, until one of the corresponding actions causes a return to the parser. The lexical analyzer returns a single value, the token name, to the parser, but uses the shared, integer variable **yylval** to pass information about the lexeme found, if needed.

**Syntax Analysis**

In this section, we examine the way the parser fits into a typical compiler. We then look typical grammars for arithmetic expressions. Grammars foe expressions suffice for illustrating the essence of parsing, since parsing techniques for expressions carry over to most programming constructs.  
This section ends with a discussion of error handling, since the parser must respond gracefully to finding that its input cannot be generated by its grammar.

**The role of the Parser**In our compiler model, the parser obtains a string of tokens from the lexical analyzer, as shown in figure X.X, and verifies that the string of tokens names can be generated by the grammar for the source language. We expect the parser to report any syntax errors in the intelligible fashion and to recover from commonly occurring errors to continue processing the remaining of the program. Conceptually, for well-formed program, the parser constructs a parse tree and passes into the rest of the compiler for further processing. In fact the parse can be interspersed with parsing, as we shall see. Thus, the parser and the rest of the front end could well be implemented by a single module.

**Intermediate representation**

**Parse tree**

Rest of front end

*getNextToken()*

Parser

**Token**

Lexical Analyzer

Figure. X.x Position of the parser in the compiler model.

**Source program**

Symbol   
Table

There are tree general types of parsers for grammars: universal , top-down and bottom-up. Universal parsing methods such as the Coke-Younger-Kasami algorithm and Earley's can pars any grammar. However these general methods are too inefficient to use in production compilers.

The methods commonly used in compilers can be classified as being , top-down or bottom-up. As implied by thire names, top-down methods build parse trees from the top(root) to bottom(leaves), while bottom-up method start from the leaves and work their way up to the root. In either case, the input to the parser is scanned from the left to right, one symbol at a time.

The most efficient top-down and bottom-up methods work only for subclasses of grammars, but several of this classes, particularly LL and LR grammars, are expressive enough to describe most of the syntactic constructs in modern programming languages.

In this project we assume that the output of the parser is some representation of the pars tree for the stream of tokens that comes from the lexical analyzer. In practice there are a number of tasks that can be conducted during parsing, such as collecting information about various tokens into the symbol table, performing type checking and other kinds of semantic analysis, and generating intermediate code. We have lamped all of this activities into the "rest of the front end" box in the Figure.

**Syntax error handling**

The remainder of this section considers the nature of syntactic errors and general strategies for error recovery.

If a compiler had to process only correct programs, its design and implementation would be simplified greatly. Whoever a compiler is expected to assist the programmer in locating and tracking down errors that inevitably creep into programs, despite the programmer's best efforts. Strikingly, few languages have been designed with error handling in mind, even though errors are so commonplace. Most programming languages specifications do not describe how the compiler should respond to errors; error handling is left to the compiler designer.

Common programming errors can occur at many difference levels.

* *Lexical* errors include misspelling of identifiers

e.g., the use of an identifier *elipseSize* instead of *ellipseSize,* and missing quotes around text intended as a string.

* *Syntactic* errors include misplaced semicolon or extra or missing braces. As another example, in C or Java, the appearance of a case statement without an enclosing switch is a syntactic error.
* *Semantic* error includes type mismatches between operators and operands.
* *Logical* errors can be anything from incorrect reasoning on the part of the programmer to the use in a C program of the assignment operator '=' instead of the comparison operator '=='.

The precision of parsing method allows synthetic errors to be detected very efficiently. Several parsing methods, such as the LL and LR methods, detect an error as soon as possible; that is , when the stream of tokens from the lexical analyzer cannot be parsed further according to the grammar for the language. More precisely , they have the *viable-prefix property,* meaning that they detect that an error has occurred as soon as they see a prefix of the input that cannot be completed to form a string in the language.

Another reason for emphasizing error recovery during parsing is that many errors appears syntactic, whatever the cause, and are exposed when parsing detected efficiently; however, accurate detected of semantic and logical errors at compile time is in general difficult task.

The error handler in the parser has goal that are simple to state but challenging to realize.

* Report the presence of errors clearly and accurately.
* Recover from each error quickly enough to detect subsequent errors.
* Add minimal overhead to the processing of correct programs.

Fortunately, common errors are simple ones, and a relatively straightforward error-handling mechanism often suffices.

**The Parser Generator Bison**

Bison is an LARL (1) - parser generator which is part of the Open Software foundation's GNU systems. It is upwards compatible with the LALR (1)-parser generator Yacc. Bison allows one to input ambiguous grammars and eliminate the resulting parser. In addition, bison has a rudimentary mechanism for handling errors.

Parsers generates by bison have an interface to C. the call of a user define C function can be associated with these reduction of a production. There is a predefined s-attribution; for every symbol of the grammar there is precisely one synthesized attribute. Its occurrence of the left side of a production is indicated by '$$', its occurrence of the symbol on a right side is indicated by ' $i' the standard type of this attribute for all symbols is int, but other type convertion exist, including different conversions for different symbols.

Parsers generated by Bison are called *yyparese.*  They expect to find a scanner called *yylex*. This may be generated by Flex or written by hand.[compiler design]

**Bison - Input**

The input file for bison has the following format.

%{

C Declarations

}%

Bison Declarations

%%

Grammar rules

%%

User-Defined C code

The *C declarations* part introduces types and variables, which are used in the semantic rules. The macro definitions and the necessary includes are also found here.

The *bison declaration part* lists the terminal part and the non-terminal alphabet and may assign individual types of the attribute of grammar symbols. The precedence of the associatively rules for operators can also be given here. Example:

%token REGISTER VAR MESH DIM VOID

%token<string> SOUTH WEST EAST N0RTH

%left ADD MIN OR MUL DIV AND MOD

%right NOT

The listing of the operators determines the precedence, operators in the same line have the same precedence, those introduced later have hither precedence then does introduced earlier. Neg is introduce as a level of precedence, in order to include the unary minus. This can only be distinguish from the binary minus by virtue of its syntactic position.

We now give an example of the use of different attribute types. Let us suppose that an abstract syntax tree is to be constructed under the control of the syntax analysis. Then the node for a statement would have pointers to its component trees. A union declaration in the bison declaration part will then introduce the set of possible attributes types.

%union {

int code;

double real;

char \*string;

NODE node;

}

One of the attributes types introduced in this way will then be assigned to each attribute of the non-terminals.

Example:

The *grammar rules* list the production and their associated semantic action.

expr: expr ADD expr { $$ = makenode(ADD,$1,$3,NULL,NULL,0,NULL);}

|expr MIN expr { $$ = makenode(MIN,$1,$3,NULL,NULL,0,NULL);}

|expr MUL expr { $$ = makenode(MUL,$1,$3,NULL,NULL,0,NULL);}

|expr DIV expr { $$ = makenode(DIV,$1,$3,NULL,NULL,0,NULL);}

|expr MOD expr { $$ = makenode(MOD,$1,$3,NULL,NULL,0,NULL);}

|expr LES expr { $$ = makenode(LES,$1,$3,NULL,NULL,0,NULL);}

|expr LEQ expr { $$ = makenode(LEQ,$1,$3,NULL,NULL,0,NULL);}

|expr EQU expr { $$ = makenode(EQU,$1,$3,NULL,NULL,0,NULL);}

|expr NEQ expr { $$ = makenode(NEQ,$1,$3,NULL,NULL,0,NULL);}

|expr GRE expr { $$ = makenode(GRE,$1,$3,NULL,NULL,0,NULL);}

|expr GEQ expr { $$ = makenode(GEQ,$1,$3,NULL,NULL,0,NULL);}

|expr AND AND expr { $$ = makenode(AND,$1,$4,NULL,NULL,0,NULL);}

|expr OR OR expr { $$ = makenode(OR ,$1,$4,NULL,NULL,0,NULL);}

| '(' expr ')' { $$ = $2; }

| MIN atom %prec DUMMY { $$ = makenode(MIN,$2,NULL,NULL,NULL,0,NULL); }

| NOT atom { $$ =makenode(NOT,$2,NULL,NULL,NULL,0,NULL); }

| atom { $$ = $1; }

;

Figure X.x a grammar for variable-free, arithmetic expressions.

As previously stated, the ambiguous of this grammar are resolved using the associatively rules and the precedence.

The *user-defined C code* contains the definitions of the actions, which accrue in the grammar rules, which may include the definition of a scanner **yylex** and other necessary functions

A translator can be constructed using Baison in the manner illustrated in figure X.X. First, a file, say rme.y, containing a Bison specification of the translator is prepared. The command *bison RME.y* transforms the file RME.y into C program called y.tab.c using the LALR method outlined in Algorithm. The program y.tab.c is a representation of the LALR parser written in C, along with other C routines that the user may have been prepared. By compiling the y.tab.c file we obtain the desired object program a.out that performs the translation specified by the original Bison program.[Compilers- Prin tech and tools]

Bison specification rme.y

y.tab.c

Bison compiler

y.tab.c

a.out

C compiler

input

Syntax tree output

a.out

Figure X.X Creating an input/output translator with Yacc

a.out

[**The Bison Parser Algorithm**](http://dinosaur.compilertools.net/bison/index.html#SEC68)

As Bison reads tokens, it pushes them onto a stack along with their semantic values. The stack is called the parser stack. Pushing a token is traditionally called shifting.

For example, suppose the infix calculator has read `1 + 5 \*', with a `3' to come. The stack will have four elements, one for each token that was shifted.

But the stack does not always have an element for each token read. When the last n tokens and groupings shifted match the components of a grammar rule, they can be combined according to that rule. This is called reduction. Those tokens and groupings are replaced on the stack by a single grouping whose symbol is the result (left hand side) of that rule. Running the rule's action is part of the process of reduction, because this is what computes the semantic value of the resulting grouping.

For example, if the infix calculator's parsers stack contains this:

1 + 5 \* 3

and the next input token is a newline character, then the last three elements can be reduced to 15 via the rule:

expr: expr '\*' expr;

Then the stack contains just these three elements:

1 + 15

At this point, another reduction can be made, resulting in the single value 16. Then the newline token can be shifted.

The parser tries, by shifts and reductions, to reduce the entire input down to a single grouping whose symbol is the grammar's start-symbol

This kind of parser is known in the literature as a bottom-up parser.

[<http://www.gnu.org/software/bison/manual/>]

[<http://dinosaur.compilertools.net/bison/bison_8.html>]

**Bison error handling**

If the parser generator by bison finds syntax error it calls a routine yyerror, which issue an error message and normally ends the syntax analysis. However, bison supports the following simple mechanism for handling syntax errors. There is a predefined symbol error. This symbol may accrue at any position on the right sides of production. The parser generated by bison reports an occurrence of this symbol if it has no legal transition. If the current state contains an item in which an error follows the period then a transition under error is executed. If this production leads to a reduction, the associated action can be used for error handling. The predefined routing yyerrok authorizes the parser to continue the analysis.