How to Write a Simple Parser by Ashim Gupta

This is a simple tutorial which gives certain simple yet necessary steps required to write a Parser. It won't cover too much details about Programming Languages constructs.
The Compiler Design Tools used in this tutorial are LEX (Lexical Analysis) YACC (Yet another Compiler Compiler).

Certain Basic Concepts and Definitions:

What is a Parser ?

A Parser for a Grammar is a program which takes in the Language string as it's input and produces either a corresponding Parse tree or an Error.

What is the Syntax of a Language?

The Rules which tells whether a string is a valid Program or not are called the Syntax.

What is the Semantics of a Language?

The Rules which gives meaning to programs are called the Semantics of a Language.

What are tokens ?

When a string representing a program is broken into sequence of substrings,such that each substring represents a constant,identifier,operator , keyword etc of the language, these substrings are called the tokens of the Language.

What is the Lexical Analysis ?

The Function of a lexical Analyzer is to read the input stream representing the Source program, one character ata a time and to translate it into valid tokens.

How can we represent a token in a language ?

The Tokens in a Language are represented by a set of Regular Expressions. A regular expression specifies a set of strings to be matched. It contains text characters and operator characters.The Advantage of using regular expression is that a recognizer can be automatically generated.

How are the tokens recognized ?

The tokens which are represented by an Regular Expressions are recognized in an input string by means of a state transition Diagram and Finite Automata.

Are Lexical Analysis and Parsing two different Passes ?

These two can form two different passes of a Parser. The Lexical analysis can store all the recognized tokens in an intermediate file and give it to the Parser as an input. However it is more convenient to have the lexical Analyzer as a coroutine or a subroutine which the Parser calls whenever it requires a token.

How do we write the Regular Expressions ?

The following are the most general notations used for expressing a R.E.

Symbol	Description
|	OR (alternation)
()	Group of Subexpression
*	0 or more Occurrences
?	0 or 1 Occurrence
+	1 or more Occurrences
{n,m}	n-m Occurrences

Suppose we want to express the 'C' identifiers as a regular Expression:-
    identifier=letter(letter|digit)*
Where letter denotes the character set a-z & A-Z
In LEX we can write it as   [a-zA-Z][a-zA-Z0-9]*

The Rules for writting R.E are:-
* An operator character may be turned into a text character by enclosing it in quotes, or by preceding it with a \ (backslash).
* a/b matches "a" but only if followed by b (the b is not matched)
* a$ matches "a" only if "a" occurs at the end of a line
* ^a matches "a" only if "a" occurs at the beginning of a line
* [abc] matches any charcter that is an "a", "b" or "c"
* [^abc] matches any charcter but "a", "b" and "c".
* ab?c matches abc and ac
* Within the square brackets most operators lose their special meanings except "\" and "-". the "^" which takes there special meaning.
* "\n" always matches newline, with or without the quotes. If you want to match the character "\" followed by "n", use \\n.

What are the Advantages of using Context-Free grammars ?

o It is precise and easy to understand.
o It is easier to determine syntatic ambiguities and conflicts in the grammar.

If Context-free grammars can represent every regular expression ,Why do one needs R.E at all?

o Regular Expression are Simpler than Context-free grammars.
o It is easier to construct a recognizer for R.E than Context-Free grammar.
o Breaking the Syntatic structure into Lexical & non-Lexical parts provide better front end for the Parser.
o R.E are most powerfull in describing the lexical constructs like identifiers, keywords etc while Context-free grammars in representing the nested or block structures of the Language.

What are the Parse Trees ?

Parse trees are the Graphical representation of the grammar which filters out
the choice for replacement order of the Production rules.
e.g
    for a production P->ABC
the parse tree would be

```
          P
        /  \\
      /     \ \
    A       B  C
```

What are Terminals and non-Terminals in a grammar ?

Terminals:- All the basic symbols or tokens of which the language is composed of are called Terminals. In a Parse Tree the Leafs represents the Terminal Symbol.
Non-Terminals:- These are syntactic variables in the grammar which represents a set of strings the grammar is composed of. In a Parse tree all the inner nodes represents the Non-Terminal symbols.

What are Ambiguious Grammars ?

A Grammar that produces more than one Parse Tree for the same sentences or the Production rules in a grammar is said to be ambiguous.
e.g consider a simple mathematical expression
E->E*E
This can have two Parse tree according to assocciativity of the operator
'*'

```
          E                         E
         / \                       / \
       *    E                    E    *
      / \                             / \
    E   E                           E   E
```

What is bottom up Parsing ?

The Parsing method in which the Parse tree is constructed from the input language string begining from the leaves and going up to the root node.
Bottom-Up parsing is also called shift-reduce parsing due to it's implementation.
The YACC supports shift-reduce pasing.
e.g Suppose there is a grammar G having a production E
E->E*E
and an input string x*y.
The left hand side of any production are called Handles. thus the handle for this example is E.
The shift action is simply pushing an input symbol on a stack. When the R.H.S of a production is matched the stack elements are popped and replaced by the corresponding Handle. This is the reduce action.
Thus in the above example , The parser shifts the input token 'x' onto the stack. Then again it shifts the token '*' on the top of the stack. Still the production is not satisfied so it shifts the next token 'y' too. Now the production E is matched so it pops all the three tokens from the stack and replaces it with the handle 'E'. Any action that is specified with the rule is carried out.
If the input string reaches the end of file /line and no error has occurred then the parser executes the 'Accept' action signifing successfull completion of parsing. Otherwise it executes an 'Error' action.

What is the need of Operator precedence ?

The shift reduce Parsing has a basic limitation. A grammar which can represent a left-sentential parse tree as well as right-sentential parse tree cannot be handled by shift reduce parsing. Such a grammar ought to have two non-terminals in the production rule. So the Terminal sandwitched between
these two non-terminals must have some associativity and precedence. This will help the parser to understand which non-terminal would be expanded first.

What is a LR Parser ?

The LR means Left-to- Right signifying the parser which reads the input string from left to right. An LR parser can be written for almost all Programming constructs.
An LR parser consists of two parts:- Driver Routine & Parsing Table. The Driver routine is same for all the Parsers ,only the Parsing Table changes. The Parsing Table is essentially a form of representing the State- Transition Diagram for the language. It consists the entries for all possible States and the input symbols. In each state there in a predetermined next state depending upon the input symbol. If there is any duplicate entry or two next states for the same symbol, then there is an ambiguity in the grammar.

What is LALR Parser ?

LALR is Look-ahead LR parser. It differs from LR in the fact that it will look ahead one symbol in the input string before going for a reduce action. Look- ahead helps in knowing if the complete rule has been matched or not.
e.g Consider a grammar G with production

P->AB|ABC
When the Parser shifts the Symbol B it can reduce to P . But if the next Symbol was C then it has not matched the complete rule. A LALR parser will shift one extra token and then take a decision to reduce or shift.

LEX-the Lexical Analyzer

The Lex utility generates a 'C' code which is nothing but a yylex() function which can be used as an interface to YACC. A good amount of details on Lex can be obtained from the Man Pages itself. A Practical approach to certain fundamentals are given here.

The General Format of a Lex File consists of three sections:

1. Definitions
2. Rules
3. User Subroutines

Definitions consists of any external 'C' definitions used in the lex actions or subroutines . e.g all preprocessor directives like #include, #define macros etc. These are simply copied to the lex.yy.c file. The other type of definitions are Lex definitions which are essentially the lex substitution strings,lex start states and lex table size declarations.The Rules is the basic part which specifies the regular expressions and their corresponding actions. The User Subroutines are the function definitions of the functions that are used in the Lex actions.

Things to remember:
1. If there is no R.E for the input string , it will be copied to the standard output.
2. The Lex resolves the ambiguity in case of matching by choosing the longest match first or by choosing the rule given first.
3. All the matched expressions are contained in yytext whose length is yyleng.

When shall we use the Lex Substitution Strings ?
Lex Substitution Strings are of importance when the regular expressions become very large and unreadable. In that case it is better to break them into smaller substitution strings and then use them in the Rules Sections. Another use is when a particular Regular Expression appears quite often in number of Rules.

When shall one use the Start states ?
Start states helps in resolving the reduce -reduce error in the Parser. It is particularly important when one wants to return different tokens for the same Regular Expression depending upon what is the previous scanned token.
e.g suppose we have two tokens /keywords CustName & ItemName followed by a string. If the BNF is such that the string reduces to two non terminals then there is reduce/reduce conflict. To avoid this it will be better to return two different tokens for the string depending upon whether CustName was scanned previously or ItemName.

How can one determine correct Table Size Declarations?
There is indeed no hard and fast rule for the above. The best method is not to give any sizes initially. The default table sizes are sufficient for small applications. However if the limits are crossed then Lex itself will generate errors messages giving info on the size violations. Then one can experiment by incrementally increasing the corresponding sizes.

When does it become important to give 'C' code to identify a token instead of a R.E?
Sometimes one might notice that after adding one more R.E to the Lex rules , the number of Lex states increases tremendously. This happens when the particular R.E has a subexpression which clashes with other R.E. Huge number of states implies more case statements and hence reduces the execution speed of yylex functions. Thus it is more economical to identify an initial yet deterministic part of the token and then use input() & unput() functions to identify the remaining part of the token.
One nice example is that of a 'C' comment (/* comment */). After identifying the '/*' one can write a simple 'C' action to identify the remaining token.

How do we redirect the Stdin to a particular FILE * ?
Normally the Lex takes the input from the Standard input through a macro definition

```
#define  lex_input()    (((yytchar=yysptr>yysbuf?U(*--yysptr):getc(yyin))==10?
 (yylineno++,yytchar):yytchar)==EOF?0:yytchar)
```

To redirect the yyyin from stdin just do the following

```
      FILE * fp=fopen("Filename","r");
      yyin=fp;
```

Another leagal but bad workaround is to redefine this Macro in the definitions section by replacing yyin with fp. However this will always give a Warning during compilation.

What is the significance of the yywrap() function ?
The yywrap() function is executed when the lexical analyzer reaches the end of the input file. It is generally used to print a summary of lexical analysis or to open another file for reading. The yywrap() function should return 0 if it has arranged for additional input, 1 if the end of the input has been reached.

YACC-Yet Another Compiler-Compiler

Yacc is the Utility which generates the function 'yyparse' which is indeed the Parser. Yacc describes a context free , LALR(1) grammar and supports both bottom-up and top-down parsing.The general format for the YACC file is very similar to that of the Lex file.

1. Declarations
2. Grammar Rules
3. Subroutines

In Declarations apart from the legal 'C' declarations there are few Yacc specific declarations which begins with a %sign.

1. %union It defines the Stack type for the Parser. It is a union of various datas/structures/ objects.

2. %token These are the terminals returned by the yylex function to the yacc. A token can also have type associated with it for good type checking and syntax directed translation. A type of a token can be specified as %token <stack member> tokenName.

3. %type The type of a non-terminal symbol in the Grammar rule can be specified with this. The format is %type <stack member> non-terminal.

4. %noassoc Specifies that there is no associativity of a terminal symbol.

5. %left Specifies the left associativity of a Terminal Symbol

6. %right Specifies the right assocoativity of a Terminal Symbol.

7. %start Specifies the L.H.S non-terminal symbol of a production rule which should be taken as the starting point of the grammar rules.

8. %prec Changes the precedence level associated with a particular rule to that of the following token name or literal.
   The grammar rules are specified as follows:
   Context-free grammar production-
   p->AbC
   Yacc Rule-
   p : A b C { /* 'C' actions */}

The general style for coding the rules is to have all Terminals in upper-case and all non-terminals in lower-case.
To facilitates a proper syntax directed translation the Yacc has something called pseudo-variables which forms a bridge between the values of terminal/non-terminals and the actions. These pseudo variables are $$,$1,$2,$3...... The $$ is the L.H.S value of the rule whereas $1 is the first R.H.S value of the rule and so is $2 etc. The default type for pseudo variables is integer unless they are specified by %type , %token <type> etc.

How are Recursions handled in the grammar rule ?
Recursions are of two types left or right recursions. The left recursion is of form

```
list :  item    { /* first item */ }
     |  list  item  { /* rest of the items */ }
```

The right recursion is of form

```
list  : item  { /* first item */ }
      | item list  { /* rest of items */ }
```

In right Recursion the Parser is a bit bigger then that of left recursion and the items are matched from right to left.

How are symbol table or data structures built in the actions ?
For a proper syntax directed translation it is important to make full use of the pseudo variables. One must have structures/classes defined of all the productions which can form a proper abstraction. The yystack should then be a union of pointers of all such structures/classes. The reason why pointers should be used instead of structures is to save space and also to avoid copying structures when the rule is reduced. Since the stack is always updated i.e on reduction the stack elements are popped and repaced by L.H.S so any data that was refered gets lost.

EXAMPLE

The above description gives a fairly good sight into the fundamentals of Parsing,Lex and YACC.   A small yet complete example with detailed description is given in the next Page. It also tells about commonly seen errors and ways to resolve them.

click for example ..

Object-Oriented YACC

An Object Oriented version of YACC. This modified version allows for multiple instances of same parser which can be used concurrently or in parallel.

click for ooyacc

LINKS

Introduction to Syntax Directed Parsing
A page by Parsifal Software, discusses about syntax Directed translation.

Parsing Techniques - A Practical Guide
A free book by Dick Grune and Ceriel J.H. Jacobs, Vrije Universiteit, Amsterdam, The Netherlands. A very good source , must read this!!

Examples of Lex and Yacc

Home Next

Contact : For Any suggestion/comments drop a mail
ashim

HOME	Practical-Management.com