**Reading a file**

In order to read a file we must do the following:

1. Load the FAT.
2. Find the root directory (which is located at second cluster).
3. Find the file entry on the directory.
4. Get the pointer to the first cluster of the file, in the directories entry.
5. Go to the cluster pointed to.
6. Read cluster.
7. Go to the cluster's FAT entry. And read the pointer.
8. If pointer is not equal to 0xFFF (end of file) go to step 3.


# Practical Compiler Development - Part 1

This is a rather informal introduction to development of a hobby compiler. The more formal chapters on compiler development will be given in later tutorials. By the end of this tutorial you will be able to create a simple interpreter . This can be easily converted into a compiler. The requirements for following this tutorial are:

1. A reasonable grasp of C / C++ / Python / C# / Java
2. An elementary knowledge of basic data structures and recursion

**Compiler defined**

A compiler is defined as a program that converts a source file in a specified form to an output file that can be fed to an assembler or simulated by a machine (virtual / real). Most of the compiler convert the given source file into assembly language that can be assembled by an assembler of the target machine. For example to see the code generated by your source program in Turbo-C++ , type the following:

tcc -S source.c

**Phases of a compiler**

*Lexical analysis* : The lexical analysis phase of the compiler essentially does the tokenizing . This phase takes a stream of text as input and converts it into tokens required for the parser . For example the lexical analyzer splits the C source into if , else , while , int etc . The lexical analyzer can be constructed using a hard coded DFA or by using a generator like lex. In the next part of this tutorial I will describe how to write your own lex like generator . In this tutorial we will use an ad hoc lexical analyzer which will be sufficient for the purpose of demonstration. In fact you will be surprised to see that this sort of approach is taken in small hobby compilers like small-C.

*Syntax Analysis* : The lexemes must be arranged in such form that reflects the syntactic structure of the language . This

is called syntax analysis or parsing . In this tutorial we will discuss only a form of top down parsing known as recursive descent parsing with one look ahead . Every programming language has a grammar which is usually represented in BNF form. A context-free grammar of a programming language is defined by the following:

1. A set of terminal symbols that cannot be substituted.
2. A set of non terminals which can be substituted by terminal or other non terminals.
3. A set of Productions that define the grammar (of the form NT -> A | B , i.e. LHS should have only one nonterminal).
4. A start symbol from which everything begins.

e.g. A grammar for the string *aaaa*.. any number of times is:

`S -> "a" S` { terminals are enclosed in " " and non terminals in capital letters }

{ | used to denote OR }

`S -> nothing`

ie this means that S can be an "a" followed by S itself or S can be nothing.

Lets see how aaa is derived:

```
S->aS
S->aaS ( using S -> aS)
S ->aaaS (using S->aS)
S->aaa (using S-> nothing).
```

Now lets see how a recursive descent parser is constructed for this grammar. This is easily done by replacing each non terminal by function call and performing a match for the terminal:

```
void parseS()
{
match_current_input_token_for("a");
if( not_matched) error();
if(end_of_input) return ;
parseS();
}
```

In order to construct a recursive descent parser like the above the grammar should follow these conditions:

1. For each production of the form `S -> A | C | E`, the first sets of each non-terminal in each of this production must be disjoint:
   ie `FIRST (A) intersection FIRST (C) intersection FIRST(E) = NULL`
2. The first sets and the follow sets of a non-terminal must be disjoint. i.e.
   `FIRST(A) intersection FOLLOW (A) = NULL`
   `FIRST(C) intersection FOLLOW ( C ) = NULL`
   and so on...

This implies that the first set should not be equal to the follow set of the non- terminal.

The first set of a terminal is the symbol itself. The first set of the non terminal is the set of terminals with which the non terminal can begin with. The follow set of a non terminal is the set of terminals which can come after the non terminal.

Now lets find out the First n Follow for the following grammar:

```
S -> "a" A | C D
A-> "b" A |"c"
C -> "e"| nothing
D-> "d" | "g" D
```

```
FIRST(S) = FIRST ( "a" A) union FIRST(CD)
= "a" union FIRST( C )
= "a" union FIRST( C ) union FIRST (D) {since C can be nothing, first of D must be included}
= "a" union FIRST ( "e" ) union FIRST ("d") union FIRST ("g")
= "a" union "e" union "d" union "g"
= { a , e , d , g }
```

Similarly `FOLLOW(C) = FIRST (D) = { d , g }` etc...

Computing the first and follow of the rest of the symbols in the grammar is left as an exercise to you . These rules have to followed in order to avoid conflicts in the predictive parsing table. There will be more about advanced parsing methods in later chapters.

As a final example, grammar for the 'if' statement:

```
S -> IFCONDTION
IFCONDITON -> "if" EXPRESSION "then" STATEMENT
EXPRESSION -> TERM "+" TERM
TERM -> FACTOR *FACTOR
FACTOR -> id
EXPRESSION -> "(" EXPRESSION ")"
STATEMENT -> id ";" STATEMENT | nothing
```

It can be seen that right recursion in many cases is unneeded and can be removed using a simple loop, as right recursion leads to tail recursion. E.g.
The grammar `S -> aS | nothing` can be more compactly written as `S-> (a)*`

The parser reduces to:

```
parseS()
{
while(current_token == "a" ) advance_input_pointer();
}
```

Left recursion will lead to an infinite loop and should be removed. For example the grammar:

```
S -> S B | D
```

can be re written as

```
S -> D S_DASH
S_DASH -> B S_DASH
```

Common factor in grammars should also be removed. For example, the grammar:

```
S ->" if" condition "then" statement
S -> "if" condition "then" statement "else" statement
```

can be modified into:

```
S->AB
A -> " if" condition "then" statement
B -> "else" statement | nothing
```

*Semantic Analysis*: Syntax analysis does not tell anything about the meaning of the statement under consideration. It simply checks whether the syntax is correct. E.g.

```
const int i = 20;
i++;
```

The above lines are syntactically correct but semantically wrong. The semantics are sometimes given using attribute grammars. More about this in later chapters.

*Code Generation*: This is the final phase where the assembly code to the target machine is generated. I have skipped many other phases which are part of a more mature compiler. The purpose of this tutorial is to give quick introduction.

In most of the hobby compilers the major component is the parser and code generation and semantic analysis is done when each non terminal is encountered in the parser.

As an example of developing a simple interpreter, I have provided the source code of a simple basic-like interpreter. I have used the mingw port of gcc for compiling the source.

Available along with the magazine are the source files to this interpreter.