“Those with knowledge have great strength.”

48 Developing a new language / writing compiler

Believe it or not, developing a new language is one of the easiest things in programming as we’ve got so many tools for developing compliers.

48.1 Secrets

Developing a new language refers to developing new grammar. Grammar refers to rules of the language.

For example, following is the part of grammar for `enum` of C:

enum-specifier:
    `enum` *identifer { enumerator-list}*
    `enum` *identifer*

enumerator-list:
    enumerator
    enumerator-list, enumerator

enumerator:
    identifier
    identifer = constant-expression

So you need to write your new language's grammar first. By the way, you must decide the data types, keywords and operators too. After preparing grammar you may need to produce a complier for your language to emphasize the merits of your language.

48.2 Writing a compiler

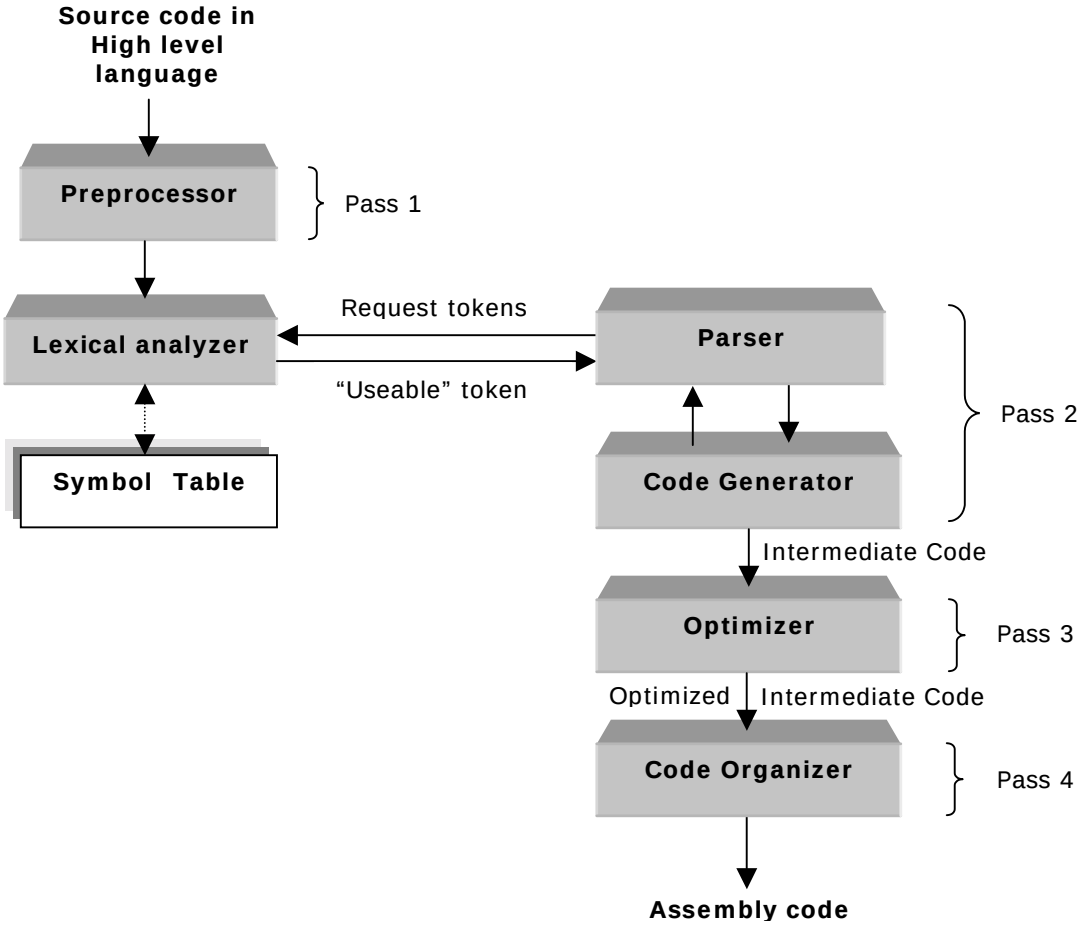
48.2.1 Compiler

First of all we must know what a compiler is and how it differs from Assembler and Linker.

- Compiler is the one which produces assembly listing (.ASM files) for a given file in high level language. In its first phase, it checks for the syntax and correctness.
- Assembler is the one which produces object (.OBJ) file for a given Assembly file.

- Linker is the one which links various object (.OBJ) files and produces executable files (.EXE or .COM).

Nowadays, we have certain integrated compilers that are able to produce the executable files directly for a given file in high-level language



48.2.2 Compiler Secrets

Let's see how our Turbo C compiler works! Understanding the functioning of an existing compiler will help us to write our own compiler.

Let's see how our `hello.c` program is been compiled by Turbo C.

```
int main( void )
{
   char *str = "Hello!\n";
   printf("%s", str);
   return( 0 );
}
```

Compile the `hello.c` program using command line compiler `tcc` with `-S` switch to get assembly listing as

```
c:>tcc –S hello.c
```

It will produce `hello.asm` file.

```
      ifndef      ??version
?debug      macro
      endm
$comm macro name,dist,size,count
      comm  dist name:BYTE:count*size
      endm
      else
$comm macro name,dist,size,count
      comm  dist name[size]:BYTE:count
      endm
      endif
      ?debug      S "hello.c"
      ?debug      C E9EA402E2B0768656C6C6F2E63
_TEXT segment byte public 'CODE'
_TEXT ends
DGROUP      group _DATA,_BSS
      assume      cs:_TEXT,ds:DGROUP
_DATA segment word public 'DATA'
d@    label byte
d@w   label word
_DATA ends
_BSS  segment word public 'BSS'
b@    label byte
b@w   label word
_BSS  ends
_TEXT segment byte public 'CODE'
   ;
   ;  int main( void )
   ;
      assume      cs:_TEXT
_main proc  near
      push  bp
      mov   bp,sp
```

```
        sub   sp,2
    ;
    ;  {
    ;     char *str = "Hello!\n";
    ;
        mov   word ptr [bp-2],offset DGROUP:s@
    ;
    ;     printf("%s", str);
    ;
        push  word ptr [bp-2]
        mov   ax,offset DGROUP:s@+8
        push  ax
        call  near ptr _printf
        pop   cx
        pop   cx
    ;
    ;     return( 0 );
    ;
        xor   ax,ax
        jmp   short @1@58
@1@58:
    ;
    ;  }
    ;
        mov   sp,bp
        pop   bp
        ret
_main endp
        ?debug      C E9
_TEXT ends
_DATA segment word public 'DATA'
s@    label byte
        db    'Hello!'
        db    10
        db    0
        db    '%s'
        db    0
_DATA ends
_TEXT segment byte public 'CODE'
_TEXT ends
        extrn _printf:near
        public      _main
_s@   equ   s@
        end
```

Here you can see how each C statement has been converted to equivalent assembly. The C statements are commented out with semicolon (;) in assembly file. I hope this might give you an idea about how high level statements are converted to equivalent assembly by compiler. Assembly file produced by the compiler can be assembled with the available assembler or with your own assembler.

48.3 Compiler-writing tools

As I pointed out, writing a compiler is a bit tough. You need to parse or split the character into meaningful tokens, check grammar and produce assembly listing. A compiler-writing tool would help us to write our own compiler without much overhead. Lex and YACC (Yet Another Compiler-Compiler) are the most famous compiler-writing utilities. Once Lex and YACC were available only to UNIX, but now we've got DOS versions too. DOS versions of lex and YACC are on CD.

A typical compiler's *source structure discovering* task can be divided into

1. Split the source file into tokens. It is a function of lexical analyzer.
2. Find the hierarchical structure of the program. It is a function of parser.

48.3.1 lex

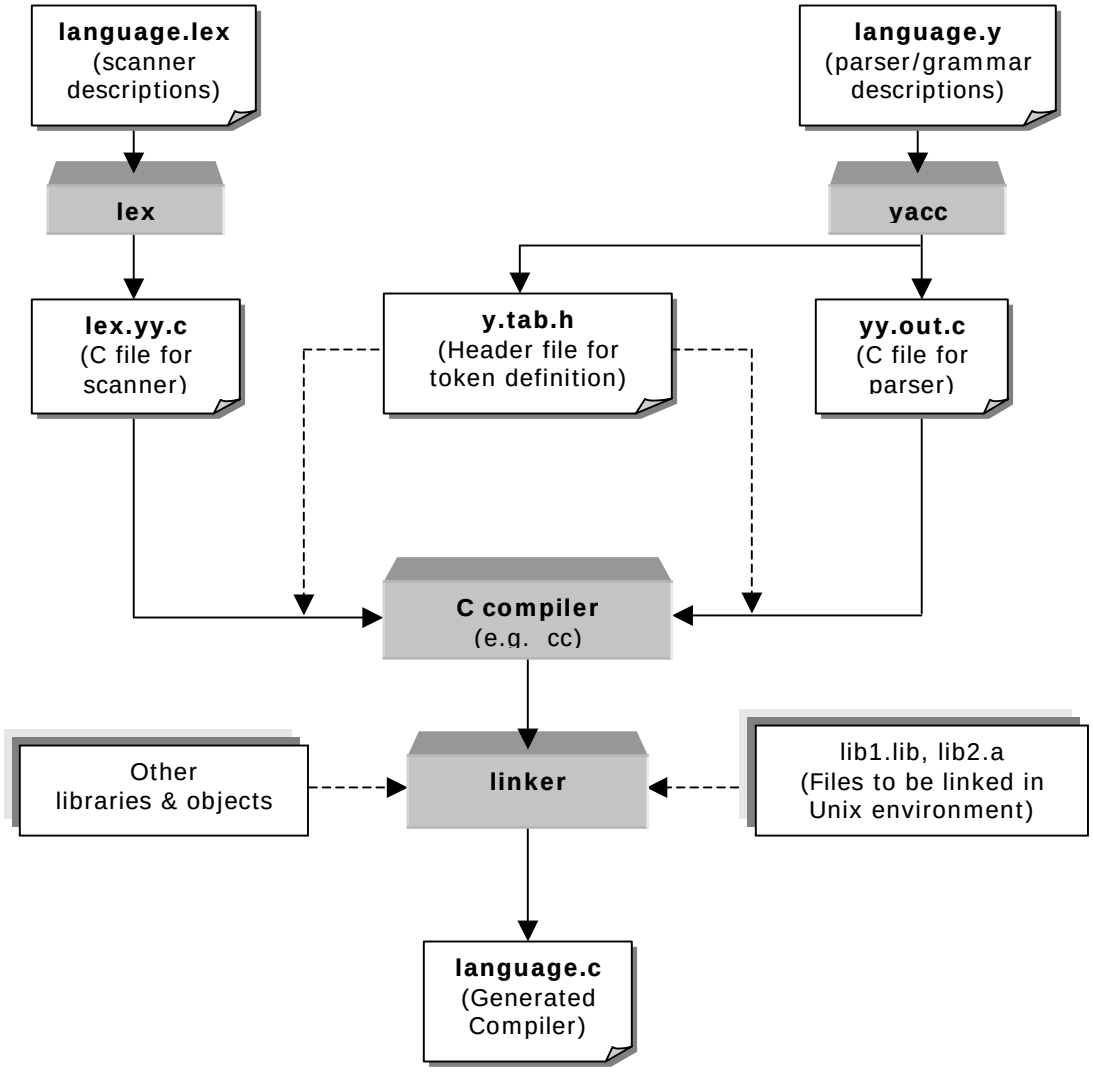
The lexical analyzer phase of a compiler is often referred as scanner or tokenizer, and it translates the input into a form that is more usable by the rest of the compiler phases. lex is a lexical anlyzer generator, which means it produces a C file that can be used as a lexical analyzer for the given (new) language.

48.3.2 YACC

YACC is a utility that translates the given *grammar* into a bottom-up parser. That is it would produce a C file that can be used as parser for your language. In otherwords, YACC will produce a compiler code for your new language, if you provide the grammar! It is really a nice tool for developing compiler in an easy and neat manner. **Berkeley YACC for MS-DOS** by **Jeff Jenness** & **Stephen C. Trier** is a clone of UNIX's YACC and it is a gift to the people who are working under DOS. **Wido Kruijtzer** also developed another **Berkeley YACC for MS-DOS** version. More information on YACC, how to input the grammar etc are available on CD.

48.3.3 Creating Compiler with lex & YACC

The following diagram shows how lex & YACC are used in UNIX environment to produce a compiler for a new *language*.



With little bit of creativity and compiler-writing utilities, hope you might come out with a new language!