

# 手把手教你构建 C 语言编译器

## (3) - 词法分析器

### Table of Contents

本章我们要讲解如何构建词法分析器。

手把手教你构建 C 语言编译器系列共有10个部分：

1. 手把手教你构建 C 语言编译器 (0) ——前言
2. 手把手教你构建 C 语言编译器 (1) ——设计
3. 手把手教你构建 C 语言编译器 (2) ——虚拟机
4. 手把手教你构建 C 语言编译器 (3) ——词法分析器
5. 手把手教你构建 C 语言编译器 (4) ——递归下降
6. 手把手教你构建 C 语言编译器 (5) ——变量定义
7. 手把手教你构建 C 语言编译器 (6) ——函数定义
8. 手把手教你构建 C 语言编译器 (7) ——语句
9. 手把手教你构建 C 语言编译器 (8) ——表达式
10. 手把手教你构建 C 语言编译器 (9) ——总结

## 什么是词法分析器

---

简而言之，词法分析器用于对源码字符串做预处理，以减少语法分析器的复杂程度。

词法分析器以源码字符串为输入，输出为标记流（token stream），即一连串的标志，每个标志通常包括： (token, token value) 即标志本身和标志的值。例如，源码中若包含一个数字 '998'，词法分析器将输出 (Number, 998)，即（数字，998）。再例如：

```
2 + 3 * (4 - 5)
=>
(Number, 2) Add (Number, 3) Multiply Left-Bracket (Number, 4) Subtract
```

通过词法分析器的预处理，语法分析器的复杂度会大大降低，这点在后面的语法分析器我们就能体会。

# 词法分析器与编译器

要是深入词法分析器，你就会发现，它的本质上也是编译器。我们的编译器是以标记流为输入，输出汇编代码，而词法分析器则是以源码字符串为输入，输出标记流。



在这个前提下，我们可以这样认为：直接从源代码编译成汇编代码是很困难的，因为输入的字符串比较难处理。所以我们先编写一个较为简单的编译器（词法分析器）来将字符串转换成标记流，而标记流对于语法分析器而言就容易处理得多了。

## 词法分析器的实现

---

由于词法分析的工作很常见，但又枯燥且容易出错，所以人们已经开发出了许多工具来生成词法分析器，如 `lex`, `flex`。这些工具允许我们通过正则表达式来识别标记。

这里注意的是，我们并不会一次性地将所有源码全部转换成标记流，原因有二：

1. 字符串转换成标记流有时是有状态的，即与代码的上下文是有关系的。
2. 保存所有的标记流没有意义且浪费空间。

所以实际的处理方法是提供一个函数（即前几篇中提到的 `next()`），每次调用该函数则返回下一个标记。

## 支持的标记

在全局中添加如下定义：

```
// tokens and classes (operators last and in precedence order)
enum {
    Num = 128, Fun, Sys, Glo, Loc, Id,
```

```
Char, Else, Enum, If, Int, Return, Sizeof, While,  
Assign, Cond, Lor, Lan, Or, Xor, And, Eq, Ne, Lt, Gt, Le, Ge, Shl, Sh  
};
```

这些就是我们要支持的标记符。例如，我们会将 `=` 解析为 `Assign`；将 `==` 解析为 `Eq`；将 `!=` 解析为 `Ne` 等等。

所以这里我们会有这样的印象，一个标记 (token) 可能包含多个字符，且多数情况下如此。而词法分析器能减小语法分析复杂度的原因，正是因为它相当于通过一定的编码（更多的标记）来压缩了源码字符串。

当然，上面这些标记是有顺序的，跟它们在 C 语言中的优先级有关，如 `*(Mul)` 的优先级就要高于 `+(Add)`。它们的具体使用在后面的语法分析中会提到。

最后要注意的是还有一些字符，它们自己就构成了标记，如右方括号 `]` 或波浪号 `~` 等。我们不另外处理它们的原因是：

1. 它们是单字符的，即并不是多个字符共同构成标记（如 `==` 需要两个字符）；
2. 它们不涉及优先级关系。

## 词法分析器的框架

即 `next()` 函数的主体：

```
void next() {
```

```
char *last_pos;
int hash;

while (token = *src) {
    ++src;
    // parse token here
}
return;
}
```

这里的一个问题是，为什么要用 `while` 循环呢？这就涉及到编译器（记得我们说过词法分析器也是某种意义上的编译器）的一个问题：如何处理错误？

对词法分析器而言，若碰到了我们一个我们不认识的字符该怎么处理？一般处理的方法有两种：

1. 指出错误发生的位置，并退出整个程序
2. 指出错误发生的位置，跳过当前错误并继续编译

这个 `while` 循环的作用就是跳过这些我们不识别的字符，我们同时还用它来处理空白字符。我们知道，C 语言中空格是用来作为分隔用的，并不作为语法的一部分。因此在实现中我们将它作为“不识别”的字符，这个 `while` 循环可以用来跳过它。

## 换行符

换行符和空格类似，但有一点不同，每次遇到换行符，我们需要将当前的行号加一：

```
// parse token here
...

if (token == '\n') {
    ++line;
}
...
```

## 宏定义

C 语言的宏定义以字符 `#` 开头，如 `# include <stdio.h>`。我们的编译器并不支持宏定义，所以直接跳过它们。

```
else if (token == '#') {
    // skip macro, because we will not support it
    while (*src != 0 && *src != '\n') {
        src++;
    }
}
```

## 标识符与符号表

标识符 (identifier) 可以理解为变量名。对于语法分析而言，我们并不关心一个变量具体叫什么名字，而只关心这个变量名代表的唯一标识。例如 `int a;` 定义了变量 `a`，而之后的语句 `a = 10`，我们需要知道这两个 `a` 指向的是同一个变量。

基于这个理由，词法分析器会把扫描到的标识符全都保存到一张表中，遇到新的标识符就去查这张表，如果标识符已经存在，就返回它的唯一标识。

那么我们怎么表示标识符呢？如下：

```
struct identifier {  
    int token;  
    int hash;  
    char * name;  
    int class;  
    int type;  
    int value;  
    int Bclass;  
    int Btype;  
    int Bvalue;  
}
```

这里解释一下具体的含义：

1. `token`：该标识符返回的标记，理论上所有的变量返回的标记都应该是 `Id`，但实际上由于我们还将在符号表中加入关键字如 `if`，`while` 等，它们都有对应的标记。
2. `hash`：顾名思义，就是这个标识符的哈希值，用于标识符的快速比较。
3. `name`：存放标识符本身的字符串。
4. `class`：该标识符的类别，如数字，全局变量或局部变量等。
5. `type`：标识符的类型，即如果它是个变量，变量是 `int` 型、`char` 型还是指针型。
6. `value`：存放这个标识符的值，如标识符是函数，则存放函数的地址。
7. `BXXXX`：C 语言中标识符可以是全局的也可以是局部的，当局部标识符的名字与全局标识符相同时，用作保存全局标识符的信息。

由上可以看出，我们实现的词法分析器与传统意义上的词法分析器不太相同。传统意义上的符号表只需要知道标识符的唯一标识即可，而我们还存放了一些只有语法分析器才会得到的信息，如 `type`。

由于我们的目标是能自举，而我们定义的语法不支持 `struct`，故而使用下列方式。

Symbol table:

```
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
.. |token|hash|name|type|class|value|btype|bclass|bvalue| ..
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
      |<---          one single identifier          --->|
```

即用一个整型数组来保存相关的ID信息。每个ID占用数组中的9个空间，分析标识符的相关代码如下：

```
int token_val;           // value of current token (mainly for num
int *current_id,         // current parsed ID
    *symbols;            // symbol table

// fields of identifier
enum {Token, Hash, Name, Type, Class, Value, BType, BClass, BValue, IdS

void next() {
    ...

    else if ((token >= 'a' && token <= 'z') || (token >= 'A' && tok

        // parse identifier
        last_pos = src - 1;
        hash = token;
```



```

while ((*src >= 'a' && *src <= 'z') || (*src >= 'A' && *src
    hash = hash * 147 + *src;
    src++;
}

// look for existing identifier, linear search
current_id = symbols;
while (current_id[Token]) {
    if (current_id[Hash] == hash && !memcmp((char *)current
        //found one, return
        token = current_id[Token];
        return;
    }
    current_id = current_id + IdSize;
}

// store new ID
current_id[Name] = (int)last_pos;
current_id[Hash] = hash;
token = current_id[Token] = Id;
return;
}
...
}

```

查找已有标识符的方法是线性查找 `symbols` 表。

## 数字

数字中较为复杂的一点是需要支持十进制、十六进制及八进制。逻辑也较为直接，可能唯一不好理解的是获取十六进制的值相关的代码。

```
token_val = token_val * 16 + (token & 15) + (token >= 'A' ? 9 : 0);
```

这里要注意的是在ASCII码中，字符a对应的十六进制值是 61，A是 41，故通过 (token & 15) 可以得到个位数的值。其它就不多说了，这里这样写的目的是装B（其实是抄c4的源代码的）。

```
void next() {
    ...

    else if (token >= '0' && token <= '9') {
        // parse number, three kinds: dec(123) hex(0x123) oct(017)
        token_val = token - '0';
        if (token_val > 0) {
            // dec, starts with [1-9]
            while (*src >= '0' && *src <= '9') {
                token_val = token_val*10 + *src++ - '0';
            }
        } else {
            // starts with number 0
            if (*src == 'x' || *src == 'X') {
                //hex
                token = *++src;
                while ((token >= '0' && token <= '9') || (token >=
                    token_val = token_val * 16 + (token & 15) + (token >=
                    token = *++src;
                }
            } else {
                // oct
                while (*src >= '0' && *src <= '7') {
                    token_val = token_val*8 + *src++ - '0';
                }
            }
        }

        token = Num;
        return;
    }
```

```
...  
}
```

## 字符串

在分析时，如果分析到字符串，我们需要将它存放到前一篇文章中说的 `data` 段中。然后返回它在 `data` 段中的地址。另一个特殊的地方是我们需要支持转义符。例如用 `\n` 表示换行符。由于本编译器的目的是达到自己编译自己，所以代码中并没有支持除 `\n` 的转义符，如 `\t`，`\r` 等，但仍支持 `\a` 表示字符 `a` 的语法，如 `\"` 表示 `"`。

在分析时，我们将同时分析单个字符如 `'a'` 和字符串如 `"a string"`。若得到的是单个字符，我们以 `Num` 的形式返回。相关代码如下：

```
void next() {  
    ...  
  
    else if (token == '"' || token == '\\') {  
        // parse string literal, currently, the only supported escape  
        // character is '\n', store the string literal into data.  
        last_pos = data;  
        while (*src != 0 && *src != token) {  
            token_val = *src++;  
            if (token_val == '\\') {  
                // escape character  
                token_val = *src++;  
                if (token_val == 'n') {  
                    token_val = '\n';  
                }  
            }  
        }  
    }  
}
```

```

        if (token == '"') {
            *data++ = token_val;
        }
    }

    src++;
    // if it is a single character, return Num token
    if (token == '"') {
        token_val = (int)last_pos;
    } else {
        token = Num;
    }

    return;
}
}

```

## 注释

在我们的C语言中，只支持 `//` 类型的注释，不支持 `/* comments */` 的注释。

```

void next() {
    ...

    else if (token == '/') {
        if (*src == '/') {
            // skip comments
            while (*src != 0 && *src != '\n') {
                ++src;
            }
        } else {
            // divide operator
            token = Div;
            return;
        }
    }
}

```

```
        }  
    }  
  
    ...  
}
```

这里我们要额外介绍 `lookahead` 的概念，即提前看多个字符。上述代码中我们看到，除了跳过注释，我们还可能返回除号 `/ (Div)` 标记。

提前看字符的原理是：有一个或多个标记是以同样的字符开头的（如本小节中的注释与除号），因此只凭当前的字符我们并无法确定具体应该解释成哪一个标记，所以只能再向前查看字符，如本例需向前查看一个字符，若是 `/` 则说明是注释，反之则是除号。

我们之前说过，词法分析器本质上也是编译器，其实提前看字符的概念也存在于编译器，只是这时就是提前看  $k$  个“标记”而不是“字符”了。平时听到的 `LL(k)` 中的 `k` 就是需要向前看的标记的个数了。

另外，我们用词法分析器将源码转换成标记流，能减小语法分析复杂度，原因之一就是减少了语法分析器需要“向前看”的字符个数。

## 其它

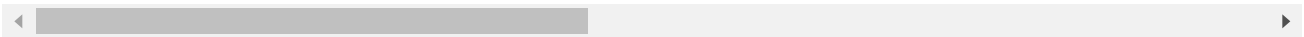
其它的标记的解析就相对容易一些了，我们直接贴上代码：

```
void next() {
    ...

    else if (token == '=') {
        // parse '==' and '='
        if (*src == '=') {
            src++;
            token = Eq;
        } else {
            token = Assign;
        }
        return;
    }
    else if (token == '+') {
        // parse '+' and '++'
        if (*src == '+') {
            src++;
            token = Inc;
        } else {
            token = Add;
        }
        return;
    }
    else if (token == '-') {
        // parse '-' and '--'
        if (*src == '-') {
            src++;
            token = Dec;
        } else {
            token = Sub;
        }
        return;
    }
    else if (token == '!') {
        // parse '!='
        if (*src == '=') {
            src++;
            token = Ne;
        }
        return;
    }
}
```

```
else if (token == '<') {
    // parse '<=', '<<' or '<'
    if (*src == '=') {
        src++;
        token = Le;
    } else if (*src == '<') {
        src++;
        token = Shl;
    } else {
        token = Lt;
    }
    return;
}
else if (token == '>') {
    // parse '>=', '>>' or '>'
    if (*src == '=') {
        src++;
        token = Ge;
    } else if (*src == '>') {
        src++;
        token = Shr;
    } else {
        token = Gt;
    }
    return;
}
else if (token == '|') {
    // parse '|' or '||'
    if (*src == '|') {
        src++;
        token = Lor;
    } else {
        token = Or;
    }
    return;
}
else if (token == '&') {
    // parse '&' and '&&'
    if (*src == '&') {
        src++;
        token = Lan;
    }
}
```

```
        } else {
            token = And;
        }
        return;
    }
    else if (token == '^') {
        token = Xor;
        return;
    }
    else if (token == '%') {
        token = Mod;
        return;
    }
    else if (token == '*') {
        token = Mul;
        return;
    }
    else if (token == '[') {
        token = Brak;
        return;
    }
    else if (token == '?') {
        token = Cond;
        return;
    }
    else if (token == '~' || token == ';' || token == '{' || token
        // directly return the character as token;
        return;
    }
    ...
}
```



代码较多，但主要逻辑就是向前看一个字符来确定真正的标记。

## 关键字与内置函数



虽然上面写完了词法分析器，但还有一个问题需要考虑，那就是“关键字”，例如 `if`，`while`，`return` 等。它们不能被作为普通的标识符，因为有特殊的含义。

一般有两种处理方法：

1. 词法分析器中直接解析这些关键字。
2. 在语法分析前将关键字提前加入符号表。

这里我们就采用第二种方法，将它们加入符号表，并提前为它们赋予必要的信息（还记得前面说的标识符 `Token` 字段吗？）。这样当源代码中出现关键字时，它们会被解析成标识符，但由于符号表中已经有了相关的信息，我们就能知道它们是特殊的关键字。

内置函数的行为也和关键字类似，不同的只是赋值的信息，在 `main` 函数中进行初始化如下：

```
// types of variable/function
enum { CHAR, INT, PTR };
int *idmain;                      // the `main` function

void main() {
    ...

    src = "char else enum if int return sizeof while "
          "open read close printf malloc memset memcmp exit void main";


    // add keywords to symbol table
    i = Char;
    while (i <= While) {
        next();
    }
}
```

```
        current_id[Token] = i++;
    }

    // add library to symbol table
    i = OPEN;
    while (i <= EXIT) {
        next();
        current_id[Class] = Sys;
        current_id[Type] = INT;
        current_id[Value] = i++;
    }

    next(); current_id[Token] = Char; // handle void type
    next(); idmain = current_id; // keep track of main

    ...
    program();
}
```



## 代码

---

本章的代码可以在 [Github](#) 上下载，也可以直接 clone

```
git clone -b step-2 https://github.com/lotabout/write-a-C-interpretor
```

上面的代码运行后会出现 ‘Segmentation Falt’，这是正常的，因为它会尝试运行我们上一章创建的虚拟机，但其中并没有任何汇编代码。

## 小结

---

本章我们为我们的编译器构建了词法分析器，通过本章的学习，我认为有几个要点需要强调：

1. 词法分析器的作用是对源码字符串进行预处理，作用是减小语法分析器的复杂程度。
2. 词法分析器本身可以认为是一个编译器，输入是源码，输出是标记流。
3. `lookahead(k)` 的概念，即向前看 `k` 个字符或标记。
4. 词法分析中如何处理标识符与符号表。

下一章中，我们将介绍递归下降的语法分析器。我们下一章见。

0 Comments   三点水    Disqus' Privacy Policy

 Login ▾

 Favorite    Tweet    Share

Sort by Best ▾



Start the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS 

Name

Be the first to comment.

 Subscribe    Add Disqus to your siteAdd  
DisqusAdd

 Do Not Sell My Data