手把手教你构建 C 语言编译器（5）- 变量定义

Table of Contents

本章中我们用 EBNF 来大致描述我们实现的 C 语言的文法，并实现其中解析变量定义部分。

由于语法分析本身比较复杂，所以我们将它拆分成 3 个部分进行讲解，分别是：变量定义、函数定义、表达式。

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

EBNF 表示

EBNF 是对前一章提到的 BNF 的扩展，它的语法更容易理解，实现起来也更直观。但真正看起来还是很烦，如果不想看可以跳过。

```
program ::= {global_declaration}+

global_declaration ::= enum_decl | variable_decl | function_decl

enum_decl ::= 'enum' [id] '{' id ['=' 'num'] {',' id ['=' 'num'] '}'

variable_decl ::= type {'*'} id { ',' {'*'} id } ';'

function_decl ::= type {'*'} id '(' parameter_decl ')' '{' body_decl '}

parameter_decl ::= type {'*'} id {',' type {'*'} id}

body_decl ::= {variable_decl}, {statement}

statement ::= non_empty_statement | empty_statement

non_empty_statement ::= if_statement | while_statement | '{' statement
                     | 'return' expression | expression ';'

if_statement ::= 'if' '(' expression ')' statement ['else' non_empty_st

while_statement ::= 'while' '(' expression ')' non_empty_statement
```

其中 `expression` 相关的内容我们放到后面解释，主要原因是我们的语言不支持跨函数递归，而为了实现自举，实际上我们也不能

使用递归（亏我们说了一章的递归下降）。

P.S. 我是先写程序再总结上面的文法，所以实际上它们间的对应关系并不是特别明显。

解析变量的定义

本章要讲解的就是上节文法中的 `enum_decl` 和 `variable_decl` 部分。

program()

首先是之前定义过的 `program` 函数，将它改成：

```
void program() {
    // get next token
    next();
    while (token > 0) {
        global_declaration();
    }
}
```

我知道 `global_declaration` 函数还没有出现过，但没有关系，采用自顶向下的编写方法就是要不断地实现我们需要的内容。下面是 `global_declaration` 函数的内容：

global_declaration()

即全局的定义语句，包括变量定义，类型定义（只支持枚举）及函数定义。代码如下：

```
int basetype;    // the type of a declaration, make it global for conve
int expr_type;   // the type of an expression

void global_declaration() {
    // global_declaration ::= enum_decl | variable_decl | function_decl
    //
    // enum_decl ::= 'enum' [id] '{' id ['=' 'num'] {',' id ['=' 'num'}
    //
    // variable_decl ::= type {'*'} id { ',' {'*'} id } ';'
    //
    // function_decl ::= type {'*'} id '(' parameter_decl ')' '{' body_


    int type; // tmp, actual type for variable
    int i; // tmp

    basetype = INT;

    // parse enum, this should be treated alone.
    if (token == Enum) {
        // enum [id] { a = 10, b = 20, ... }
        match(Enum);
        if (token != '{') {
            match(Id); // skip the [id] part
        }
        if (token == '{') {
            // parse the assign part
            match('{');
            enum_declaration();
            match('}');
        }

        match(';');
        return;
    }
```

```
// parse type information
if (token == Int) {
    match(Int);
}
else if (token == Char) {
    match(Char);
    basetype = CHAR;
}

// parse the comma seperated variable declaration.
while (token != ';' && token != '}') {
    type = basetype;
    // parse pointer type, note that there may exist `int ****x;`
    while (token == Mul) {
        match(Mul);
        type = type + PTR;
    }

    if (token != Id) {
        // invalid declaration
        printf("%d: bad global declaration\n", line);
        exit(-1);
    }
    if (current_id[Class]) {
        // identifier exists
        printf("%d: duplicate global declaration\n", line);
        exit(-1);
    }
    match(Id);
    current_id[Type] = type;

    if (token == '(') {
        current_id[Class] = Fun;
        current_id[Value] = (int)(text + 1); // the memory address
        function_declaration();
    } else {
        // variable declaration
        current_id[Class] = Glo; // global variable
        current_id[Value] = (int)data; // assign memory address
        data = data + sizeof(int);
    }
```

```
            if (token == ',') {
                match(',');
            }
        }
        next();
    }
```

看了上面的代码，能大概理解吗？这里我们讲解其中的一些细节。

向前看标记：其中的 `if (token == xxx)` 语句就是用来向前查看标记以确定使用哪一个产生式，例如只要遇到 `enum` 我们就知道是需要解析枚举类型。而如果只解析到类型，如 `int identifier` 时我们并不能确定 `identifier` 是一个普通的变量还是一个函数，所以还需要继续查看后续的标记，如果遇到 `(` 则可以断定是函数了，反之则是变量。

变量类型的表示：我们的编译器支持指针类型，那意味着也支持指针的指针，如 `int **data;`。那么我们如何表示指针类型呢？前文中我们定义了支持的类型：

```
// types of variable/function
enum { CHAR, INT, PTR };
```

所以一个类型首先有基本类型，如 `CHAR` 或 `INT`，当它是一个指向基本类型的指针时，如 `int *data`，我们就将它的类型加上 `PTR` 即代码中的：`type = type + PTR;`。同理，如果是指针的指针，则再加上 `PTR`。

enum_declaration()

用于解析枚举类型的定义。主要的逻辑用于解析用逗号（`,`）分隔的变量，值得注意的是在编译器中如何保存枚举变量的信息。

即我们将该变量的类别设置成了 `Num`，这样它就成了全局的常量了，而注意到上节中，正常的全局变量的类别则是 `Glo`，类别信息在后面章节中解析 `expression` 会使用到。

```
void enum_declaration() {
    // parse enum [id] { a = 1, b = 3, ...}
    int i;
    i = 0;
    while (token != '}') {
        if (token != Id) {
            printf("%d: bad enum identifier %d\n", line, token);
            exit(-1);
        }
        next();
        if (token == Assign) {
            // like {a=10}
            next();
            if (token != Num) {
                printf("%d: bad enum initializer\n", line);
                exit(-1);
            }
            i = token_val;
            next();
        }

        current_id[Class] = Num;
        current_id[Type] = INT;
        current_id[Value] = i++;

        if (token == ',') {
            next();
```

```
        }
    }
}
```

其它

其中的 `function_declaration` 函数我们将放到下一章中讲解。`match` 函数是一个辅助函数：

```
void match(int tk) {
    if (token == tk) {
        next();
    } else {
        printf("%d: expected token: %d\n", line, tk);
        exit(-1);
    }
}
```

它将 `next` 函数包装起来，如果不是预期的标记则报错并退出。

代码

本章的代码可以在 Github 上下载，也可以直接 clone

```
git clone -b step-3 https://github.com/lotabout/write-a-C-interpreter
```

本章的代码还无法正常运行，因为还有许多功能没有实现，但如果有兴趣的话，可以自己先试着去实现它。

小结

本章的内容应该不难，除了开头的 EBNF 表达式可能相对不好理解一些，但如果你查看了 EBNF 的具体表示方法后就不难理解了。

剩下的内容就是按部就班地将 EBNF 的产生式转换成函数的过程，如果你理解了上一章中的内容，相信这部分也不难理解。

下一章中我们将介绍如何解析函数的定义，敬请期待。