手把手教你构建 C 语言编译器（6）- 函数定义

Table of Contents

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。本章讲解函数定义相关的内容。

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

这是上一章的 EBNF 方法中与函数定义相关的内容。

```
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
```

解析函数的定义

上一章的代码中，我们已经知道了什么时候开始解析函数的定义，相关的代码如下：

```
...
if (token == '(') {
    current_id[Class] = Fun;
    current_id[Value] = (int)(text + 1); // the memory address of funct
    function_declaration();
} else {
...
```

即在这断代码之前，我们已经为当前的标识符（identifier）设置了正确的类型，上面这断代码为当前的标识符设置了正确的类别（Fun），以及该函数在代码段（text segment）中的位置。接下来开始解析函数定义相关的内容：`parameter_decl` 及 `body_decl`。

函数参数与汇编代码

现在我们要回忆如何将“函数”转换成对应的汇编代码，因为这决定了在解析时我们需要哪些相关的信息。考虑下列函数：

```
int demo(int param_a, int *param_b) {
    int local_1;
    char local_2;

    ...
}
```

那么它应该被转换成什么样的汇编代码呢？在思考这个问题之前，我们需要了解当 `demo` 函数被调用时，计算机的栈的状态，如下（参照第三章讲解的虚拟机）：

```
|    ....       | high address
+---------------+
| arg: param_a  |    new_bp + 3
+---------------+
| arg: param_b  |    new_bp + 2
+---------------+
|return address |    new_bp + 1
+---------------+
| old BP        | <- new BP
+---------------+
| local_1       |    new_bp - 1
+---------------+
| local_2       |    new_bp - 2
+---------------+
|    ....       |  low address
```

这里最为重要的一点是，无论是函数的参数（如 `param_a`）还是函数的局部变量（如 `local_1`）都是存放在计算机的 **栈** 上的。因此，与存放在 **数据段** 中的全局变量不同，在函数内访问它们是通过 `new_bp` 指针和对应的位移量进行的。因此，在解析的过程中，我们需要知道参数的个数，各个参数的位移量。

函数定义的解析

这相当于是整个函数定义的语法解析的框架，代码如下：

```
void function_declaration() {
    // type func_name (...) {...}
    //               | this part

    match('(');
    function_parameter();
    match(')');
    match('{');
```

```
    function_body();
    //match('}');                   //  ①

    // ②
    // unwind local variable declarations for all local variables.
    current_id = symbols;
    while (current_id[Token]) {
        if (current_id[Class] == Loc) {
            current_id[Class] = current_id[BClass];
            current_id[Type]  = current_id[BType];
            current_id[Value] = current_id[BValue];
        }
        current_id = current_id + IdSize;
    }
}
```

其中①中我们没有消耗最后的`}`字符。这么做的原因是：`variable_decl`与`function_decl`是放在一起解析的，而`variable_decl`是以字符`;`结束的。而`function_decl`是以字符`}`结束的，若在此通过`match`消耗了';'字符，那么外层的`while`循环就没法准确地知道函数定义已经结束。所以我们将结束符的解析放在了外层的`while`循环中。

而②中的代码是用于将符号表中的信息恢复成全局的信息。这是因为，局部变量是可以和全局变量同名的，一旦同名，在函数体内局部变量就会覆盖全局变量，出了函数体，全局变量就恢复了原先的作用。这段代码线性地遍历所有标识符，并将保存在`BXXX`中的信息还原。

解析参数

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

解析函数的参数就是解析以逗号分隔的一个个标识符，同时记录它们的位置与类型。

```
int index_of_bp; // index of bp pointer on stack

void function_parameter() {
    int type;
    int params;
    params = 0;
    while (token != ')') {
        // ①

        // int name, ...
        type = INT;
        if (token == Int) {
            match(Int);
        } else if (token == Char) {
            type = CHAR;
            match(Char);
        }

        // pointer type
        while (token == Mul) {
            match(Mul);
            type = type + PTR;
        }

        // parameter name
        if (token != Id) {
            printf("%d: bad parameter declaration\n", line);
            exit(-1);
        }
        if (current_id[Class] == Loc) {
            printf("%d: duplicate parameter declaration\n", line);
            exit(-1);
```

```
        }

        match(Id);

        //②
        // store the local variable
        current_id[BClass] = current_id[Class]; current_id[Class]  = Lc
        current_id[BType]  = current_id[Type];  current_id[Type]   = ty
        current_id[BValue] = current_id[Value]; current_id[Value]  = pa

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

    // ③
    index_of_bp = params+1;
}
```

其中①与全局变量定义的解析十分一样，用于解析该参数的类型。

而②则与上节中提到的“局部变量覆盖全局变量”相关，先将全局变量的信息保存（无论是是否真的在全局中用到了这个变量）在 `BXXX` 中，再赋上局部变量相关的信息，如 `Value` 中存放的是参数的位置（是第几个参数）。

③则与汇编代码的生成有关，`index_of_bp` 就是前文提到的 `new_bp` 的位置。

函数体的解析

我们实现的 C 语言与现代的 C 语言不太一致，我们需要所有的变量定义出现在所有的语句之前。函数体的代码如下：

```
void function_body() {
    // type func_name (...) {...}
    //                   -->|   |<--

    // ... {
    // 1. local declarations
    // 2. statements
    // }

    int pos_local; // position of local variables on the stack.
    int type;
    pos_local = index_of_bp;

    // ①
    while (token == Int || token == Char) {
        // local variable declaration, just like global ones.
        basetype = (token == Int) ? INT : CHAR;
        match(token);

        while (token != ';') {
            type = basetype;
            while (token == Mul) {
                match(Mul);
                type = type + PTR;
            }

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

```
            match(Id);

            // store the local variable
            current_id[BClass] = current_id[Class]; current_id[Class]
            current_id[BType]  = current_id[Type];  current_id[Type]
            current_id[BValue] = current_id[Value]; current_id[Value]

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

    // ②
    // save the stack size for local variables
    *++text = ENT;
    *++text = pos_local - index_of_bp;

    // statements
    while (token != '}') {
        statement();
    }

    // emit code for leaving the sub function
    *++text = LEV;
}
```

其中①用于解析函数体内的局部变量的定义，代码与全局的变量定义几乎一样。

而②则用于生成汇编代码，我们在第三章的虚拟机中提到过，我们需要在栈上为局部变量预留空间，这两行代码起的就是这个作用。

代码

本章的代码可以在 Github 上下载，也可以直接 clone

```
git clone -b step-4 https://github.com/lotabout/write-a-C-interpreter
```

本章的代码依旧无法运行，还有两个重要函数没有完成：`statement` 及 `expression`，感兴趣的话可以尝试自己实现它们。

小结

本章中我们用了不多的代码完成了函数定义的解析。大部分的代码依旧是用于解析变量：参数和局部变量，而它们的逻辑和全局变量的解析几乎一致，最大的区别就是保存的信息不同。

当然，要理解函数定义的解析过程，最重要的是理解我们会为函数生成怎样的汇编代码，因为这决定了我们需要从解析中获取什么样的信息（例如参数的位置，个数等），而这些可能需要你重新回顾一下“虚拟机”这一章，或是重新学习学习汇编相关的知识。

下一章中我们将讲解语句的解析，敬请期待。