完成 C 编译器

徐子越

November 2019

目录

1 作业要求

完成编译器的代码生成部分，中间代码生成和代码优化不作要求，生成的目标代码要求为汇编语言程序。

1. 对类型检查过的语法树进行遍历，实现临时变量分配、标号生成和目标代码生成，可参考第 8 章讲义中代码片段。
2. 生成目标程序（汇编程序）可借助部分 MASM32 的宏特性，如变量定义、输入、输出，但条件分支、循环、函数调用等 MASM32 宏语句不能用来直接进行目标码生成。
3. 例如给出一个计算 Fibonacci 简化 C 语言例程 tmp.c 和它编译后形成的 tmp.asm，tmp.asm 经 MASM32 编译后可形成可执行代码——输入一个整数 n，程序输出 fibonacci(0) fibonacci(n) 的值。大家可参考这个例子，利用类似例子调试自己的编译器程序。

2 作业思路

2.1 大体流程

根据作业要求，本次实验主要是在之前实验的基础上实现将 C 语言转换为对应汇编语言的操作，已达到实现 C 编译器的目的。

主要包括实现关于数据段（.data）以及代码段（.code）这两个汇编代码的主要部分，此外，还需要一些其他的辅助函数例如临时变量分配函数并且需要完善节点元素例如为了实现跳转语句加入的 label（取决于具体实现）。详细内容可以参考讲义 8 中最后的相关代码。

2.2 代码讲解

不妨以讲义 8 的内容为例，主函数代码如下：

frame

```
1  int main(int argc, char *argv[])
2  {
3          int n = 1;
4          lexer lexer;
5          parser parser;
6          if (parser.yycreate(&lexer)) {
7                  if (lexer.yycreate(&parser)) {
8                          lexer.yyin = new ifstream(argv[1]);
9                          lexer.yyout = new ofstream(argv[2]);
10                         n = parser.yyparse();
11                         parse_tree.get_label();
```

```
12                          parse_tree.gen_code(*lexer.yyout);
13                  }
14          }
15          getchar();
16          return n;
17  }
```

main

通过对 yyin/yyout 重赋值，以改变 lexer 默认的输入/输出目标，使用文件输入输出的方式进行交互，之后的两个函数 get_label 以及 get_label 分别是对语法树节点写标号并且生成相应代码。当然，这只是其中一种思路，可以将两个函数的作用合并到一起来实现。下面着重讲解两个函数的思路。

2.2.1 gen_label

该函数主要针对语句和表达式类型分为两类，下面为语句类型的贴标号函数：

frame

```
1  void tree::stmt_get_label(Node *t)
2  {
3    switch (t->kind_kind)
4    { …
5      case WHILE_STMT:
6      {
7        Node *e = t->children[0];
8        // 循环判定条件——布尔表达式
9        Node *s = t->children[1];
10       // 循环体
11       if (t->label.begin_label == "")
12         t->label.begin_label = new_label();
13         // 生成循环开始标号
14       s->label.next_label = t->label.begin_label;
15       // 循环体的“下一条语句”是循环开始标号——继续循环
16       s->label.begin_label = e->label.true_label = new_label();
17      // 生成循环体开始标号——也是循环条件真值出口
18       if (t->label.next_label == "")
19         t->label.next_label = new_label();
20         // 生成整个循环的下一条语句标号（循环结束）
```

```
21        e->label.false_label = t->label.next_label;
22        // 循环条件假值出口即循环结束
23        if (t->sibling)
24          t->sibling->label.begin_label = t->label.next_label;
25                // 兄弟结点的开始标号即这条语句的下一个标号
26        recursive_get_label(e);
27        // 递归地生成布尔表达式内标号
28        recursive_get_label(s);
29        // 递归地生成循环体内标号
30      }
31    }
32  }
```

stmt_get_label

下面为表达式类型的贴标号函数:

frame

```
1  void tree::expr_get_label(Node *t)      // 布尔表达式生成标号
2  {
3    if (t->type != Boolean)        return;
4    Node *e1 = t->children[0];             // 第一子表达式
5    Node *e2 = t->children[1];             // 第二子表达式
6    switch (t->attr.op)
7    {
8      case AND:
9        e1->label.true_label = new_label();
10       // 子表达式一为真，跳转到子表达式二代码
11       e2->label.true_label = t->label.true_label;
12       // 子表达式二也为真，与操作结果为真——真值出口是相同标号
13       e1->label.false_label = e2->label.false_label =
14       t->label.false_label;
15       // 三者假值出口是相同标号
16       break;
17     case OR:
18       e1->label.false_label = new_label();
19       e2->label.false_label = t->label.false_label;
```

```
20          e1->label.true_label = e2->label.true_label =
21          t->label.true_label;
22          break;
23        case NOT:
24          e1->label.true_label = t->label.false_label;
25          e1->label.false_label = t->label.true_label;
26          break;
27        }
28    if (e1)   recursive_get_label(e1);
29    if (e2)   recursive_get_label(e2);
30  }
```

expr_get_label

2.2.2 gen_code

代码生成函数主要由 3 个部分组成即汇编语言基本格式、数据段、代码段。需要说明的是汇编语言基本格式主要有需要包含的头文件、代码段开始与结束符号、退出函数等，根据自己的需要进行添加。下图为 gen_code 函数的主要内容：

frame

```
1  void tree::gen_code(ostream &out)
2  {
3    gen_header(out);
4    Node *p = root->children[0];
5    if (p->kind == DECL_NODE)
6      gen_decl(out, p);
7    out << endl << endl << "\t.code" << endl;
8    recursive_gen_code(out, root);
9    if (root->label.next_label != "")
10     out << root->label.next_label << ":" << endl;
11   out << "\tinvoke ExitProcess, 0" << endl;
12   out << "end " << root->label.begin_label << endl;
13 }
```

gen_code

该函数基本上将汇编代码的框架清楚的表示，首先将汇编函数的头部进行输出，之后将所有 C 语

言中所有与声明相关的语句转化为数据段的相关内容，之后再将代码段的相关内容进行输出，最后调用退出程序函数 ExitProcess 以及结束符 end，汇编程序完成。

下面为 gen_header 的相关代码：

frame

```
1  void tree::gen\_header(ostream &out)
2  {
3    out << "\t.586" << endl;
4    out << "\t.model flat, stdcall" << endl;
5    out << "\toption casemap :none" << endl;
6    out << endl;
7    out << "\tinclude \\masm32\\include\\windows.inc" << endl;
8    out << "\tinclude \\masm32\\include\\user32.inc" << endl;
9    out << "\tinclude \\masm32\\include\\kernel32.inc" << endl;
10   out << "\tinclude \\masm32\\include\\masm32.inc" << endl;
11   out << endl;
12   out << "\tincludelib \\masm32\\lib\\user32.lib" << endl;
13   out << "\tincludelib \\masm32\\lib\\kernel32.lib" << endl;
14   out << "\tincludelib \\masm32\\lib\\masm32.lib" << endl;
15 }
```

gen_header

即将汇编语言的“头部”进行输出，包含的头文件取决于你在汇编语言中要使用的函数。其中汇编语句涉及到的具体作用参见第二次实验指导书。

下面为 get_temp_var 的相关代码，用于计算需要申请的临时变量的数量：

frame

```
1  void tree::get_temp_var(Node *t)
2  {
3          if (t->kind != EXPR_NODE)
4                  return;
5          if (t->attr.op < PLUS || t->attr.op > OVER)
6                  return;
7
8          Node *arg1 = t->children[0];
9          Node *arg2 = t->children[1];
10         // 临时变量重用（收回不用的临时变量）
```

```
11          if (arg1->kind_kind == OP_EXPR)
12                  temp_var_seq--;
13          if (arg2 && arg2->kind_kind == OP_EXPR)
14                  tree::temp_var_seq--;
15          t->temp_var = tree::temp_var_seq;
16          // 分配临时变量
17          tree::temp_var_seq++;
18  }
```

get_temp_var

下面为 gen_decl 的相关代码:

frame

```
1  void tree::gen_decl(ostream &out, Node *t)
2  {
3    out << endl << endl << "\t.data" << endl;
4
5    for (; t->kind == DECL_NODE; t = t->sibling)
6    {
7      for (Node *p = t->children[1]; p; p = p->sibling)
8        if (p->type == Integer)
9          out << "\t\t_" << symtbl.getname(p->attr.symtbl_seq)
10         << " DWORD 0" << endl;
11       else if (p->type == Char)
12         out << "\t\t_" << symtbl.getname(p->attr.symtbl_seq)
13         << " BYTE 0" << endl;
14   }
15  for (int i = 0; i < temp_var_seq; i++)
16   {
17     out << "\t\tt" <<  i << " DWORD 0" << endl;
18   }
19
20   out << "\t\tbuffer BYTE 128 dup(0)" << endl;
21   out << "\t\tLF BYTE 13, 10, 0" << endl;
22 }
```

gen_decl

首先对语法树进行遍历，将所有声明语句的相关变量在数据段进行声明，需要注意的是从声明语句的第二个孩子开始遍历，最后再将之前统计好的临时变量进行声明。

下面为 recursive_gen_code 的相关代码：

frame

```
1
2  void tree::recursive_gen_code(ostream &out, Node *t)
3  {
4    if (t->kind == STMT_NODE)
5    {
6      stmt_gen_code(out, t);
7    }
8    else if (t->kind == EXPR_NODE && (t->kind_kind == OP_EXPR ||
9    t->kind_kind == NOT_EXPR))
10   {
11     expr_gen_code(out, t);
12   }
13 }
```

recursive_gen_code

该函数也是通过区分节点类型进行汇编语言的构建。下面为 expr_gen_code 的相关代码：

frame

```
1  void tree::expr_gen_code(ostream &out, Node *t)
2  {
3    Node *e1 = t->children[0];
4    Node *e2 = t->children[1];
5    switch (t->attr.op)
6    {
7    case PLUS:
8      out << "\tMOV eax, ";
9      if (e1->kind_kind == ID_EXPR)
10       out << "_" << symtbl.getname(e1->attr.symtbl_seq);
11     else if (e1->kind_kind == CONST_EXPR)
```

```
12        out << e1->attr.vali;
13      else out << "t" << e1->temp_var;
14      out << endl;
15      out << "\tADD eax, ";
16          if (e2->kind_kind == ID_EXPR)
17        out << "_" << symtbl.getname(e2->attr.symtbl_seq);
18      else if (e2->kind_kind == CONST_EXPR)
19        out << e2->attr.vali;
20      else out << "t" << e2->temp_var;
21      out << endl;
22      out << "\tMOV t" << t->temp_var << ", eax" << endl;
23      break;
24    case LT:
25      out << "\tMOV eax, ";
26      if (e1->kind_kind == ID_EXPR)
27        out << "_" << symtbl.getname(e1->attr.symtbl_seq);
28      else if (e1->kind_kind == CONST_EXPR)
29        out << e1->attr.vali;
30      else out << "t" << e1->temp_var;
31      out << endl;
32      out << "\tCMP eax, ";
33      if (e2->kind_kind == ID_EXPR)
34        out << "_" << symtbl.getname(e2->attr.symtbl_seq);
35      else if (e2->kind_kind == CONST_EXPR)
36        out << e2->attr.vali;
37      else out << "t" << e2->temp_var;
38      out << endl;
39      out << "\tjl " << t->label.true_label << endl;
40      out << "\tjmp " << t->label.false_label << endl;
41      break;
42   }
43 }
```

expr_gen_code

此处给出了两个例子，分别是加法和小于比较：

1. 加法（PLUS）：首先将第一个操作数的值给到 eax 寄存器中，此时需要区分第一个操作数节点的类型（标识符、常数或者操作符），分别对应不同操作。之后将 eax 的值也就是第一个操作数的值与第二个操作数相加，最后将相加后得到的数值保存在加号节点的临时变量中。

2. 小于（LT）：操作几乎与加法相同，主要注意的是更改了汇编语言以及使用了之前设定好的 label。

下面为 expr_gen_code 的相关代码：

frame

```
1  void tree::stmt_gen_code(ostream &out, Node *t)
2  {
3    if (t->kind_kind == COMP_STMT)
4    {
5      for (int i = 0; t->children[i]; i++)
6      {
7        recursive_gen_code(out, t->children[i]);
8        for (Node *p = t->children[i]->sibling; p; p = p->sibling)
9          recursive_gen_code(out, p);
10     }
11   }
12     else if (t->kind_kind == WHILE_STMT)
13   {
14     if (t->label.begin_label != "")
15       out << t->label.begin_label << ":" << endl;
16     recursive_gen_code(out, t->children[0]);
17     recursive_gen_code(out, t->children[1]);
18     out << "\tjmp " << t->label.begin_label << endl;
19   }
```

stmt_gen_code

此处给出了两个例子，分别是复合语句和 While 语句：

1. 复合语句（COMP_STMT）：对该节点进行遍历，递归打印相关汇编语言。

2. While 语句（WHILE_STMT）：首先打印出该语句的起始 label，之后的再将该节点的两个孩子进行打印，生成布尔表达式代码以及循环体代码，最后跳转回起始 label；

以上就是本次实验的全部内容