编译技术申优文档

18373466 战晨曦

本申优文档将尽可能地去介绍实现编译器的整个过程，以及在实现编译器的过程中，我遇到了什么样的困难，以及我是如何解决的。由于笔者并没有做什么拿得出手的优化，所以本文档将基本不会涉及优化方面的内容。

词法分析部分

在词法分析部分，通过通读题目，可以发现我们要做的其实就是把一个字符流转化成 `Token` 流，其中每个 `Token` 是输入字符流中最小的有意义的单位。做了词法分析，我们以后就可以以 `Token` 为单位去解读整个输入流。

这里值得一提的是，`Token` 流在做完语法分析之后就再也不会用到的，且 `Token` 的最关键的信息就是这个 `Token` 是什么类型的，所以完全没有必要为每种 `Token` 建立一个类，我们只需要给 `Token` 类设置一个字段，该字段存储本 `Token` 的类型就可以了。对于类型集合，使用一个枚举类进行存储即可。

学过理论课的话可以知道，程序设计语言中的 `Token` 基本上都可以用正则文法去表示，而正则文法对应着正则表达式，所以如果我们使用带有正则表达式工具包的语言去开发编译器，那么其实是可以考虑使用它的。我们可以用一个指针 `cur` 记录当前已经词法分析到哪儿了，然后从 `cur` 开始往后看，用正则表达式的 `lookingAt` 方法（匹配前缀）看某个正则表达式是否能够匹配得到，匹配得到的话就新建一个 `Token` ，存储一下 `Token` 值和类型，把 `cur` 移动到剩余部分的开头。

在使用正则表达式做词法分析时可能会遇到如下问题：

- 怎么去表示一些**比较特殊的** `Token` ，比方说单行注释和多行注释。首先我们思考一下，这个问题很可能是一个比较经典的问题，因为删除注释这种需求还是蛮常见的，所以网上很有可能会有现成的正则表达式，拿过来测试一下之后可以直接使用。
- 在理论课上我们也学过，词法分析阶段可能会出现**多个正则表达式都能匹配待分析部分的某个前缀**的情况，比方说 `const` 是关键字，但其也满足标识符的正则表达式。遇到这种情况，可以考虑理论课上学过**最长匹配原则**，取最长的 `pattern` 。如果最长的 `pattern` 一样长，如果这些 `pattern` 里有关键字，则关键字优先。

在词法分析阶段，我曾经修复过如下 bug，有需要的同学可以参考一下：

- 错误实现最长匹配。
- 多行注释处理错误，多行注释中出现**不认识的终结符**导致失配。
- 忘记处理**标识符中的下划线**。
- 遇到多行注释时**行号统计错误**。
- 正则表达式使用错误
    - 错误使用了负向断言。
    - 一些特殊符号的**转义**。
    - **优先级**问题。
    - 其他各种对 `Java` 正则表达式的误使用。要多读文档！

语法分析部分

在语法分析部分，我们要以 `Token` 流为基础，去识别出来各种语法成分，得到一颗抽象语法树，或者是暂时只做语法成分的识别，等到代码生成的时候把生成代码的操作插到每个递归子程序中。

语法分析部分在实现时我使用的是递归下降分析法，主要的问题在于我们要约定好子程序之间的接口，不然会出现：

- 子程序 `A` 调用子程序 `B`，`A` 期望 `B` 读第一个要分析的 `Token`，但 `B` 又期望 `A` 在调用它之前已经把待分析的 `Token` 读进来了，所以俩程序都没读，所以分析错误了。
- 子程序 `A` 调用子程序 `B`，`A` 想先读入第一个 `Token`，再调用 `B`，而 `B` 又想自己读入，这样俩程序都读入了，分析同样错误。

我在设计子程序之间的接口时，做了两个约定：

- `A` 调用 `B` 的子程序之前，`A` 一定把 `B` 要分析的第一个 `Token` 读进来了。
- 离开 `B` 的子程序之前，`B` 一定把分析完当前语法成分之后的下一个 `Token` 读进来了。

始终保持这个约定，则很多 bug 可以解决。

下面是解决一些递归下降分析法的痛点：

- 对于表达式等文法，会发现存在左递归，需要把 `E -> E + T` 型文法改成 `E -> T {+ T}` 的非左递归文法。
- 对于 `Stmt` 的分析，存在不容易判断应该调用分析哪种 `Stmt` 的分析方法的情况，传统的预读法是往后偷看1个甚至多个 `Token`，但有时候遇到分析到底是 `LVal`（`stmt1`）还是 `LVal = exp`（`stmt2`）（这是两种不同的 `Stmt`），这个时候我们不知道要预读多少个才行。在这时，我们其实可以考虑直接调用 `LVal` 的分析方法，即一次往后看一个 `LVal`，然后再看下一个符号是不是 `=`，如果是的话，我们可以把指针回退到分析 `LVal` 之前的状态，调用 `stmt2` 的分析方法；如果不是的话，就回退回去，调用 `stmt1` 的分析方法。我个人认为这也是一种有目的的预读。

下面记录一下语法分析部分遇到的bug：

- 分析子程序之间的接口问题。
- 左递归文法分析得到的输出和我消除左递归后输出不一致。
- 对于比较难的几个预读判断（`Stmt` 里面的 `[Exp];` 和 `LVal` 等）情况没有分析对。
- 有一些 `;` 和 `,` 漏解析/多解析的问题。
- 第二次写的时候遇到了多/少换行问题。
- 有些结点的某些域可能是空，需要特判。

错误处理部分

在错误处理部分，我们需要对一些常见的词法错误、语法错误以及语义错误进行识别并报错。

词法和语法错误比较简单，一般不太会有问题，关键是语义错误。在本次作业中，我们需要用符号表去辅助我们查找语义错误。

在这里，第一个难点就是如何去设计符号表。由于这个时候大部分班级的理论课没讲到代码生成，所以可能对符号表的理解比较浅。我个人认为，与其犹豫不知道咋设计符号表，不妨就从简处理。

- 符号表里面按理说应该存地址什么的，但是现在还没有地址的概念（因为没生成代码），所以可以暂时不要管，只存最基础的作用域层次、符号名字、符号类型即可，这个可以看作是**符号表项基类**。
- 考虑到对于函数和数组这种特殊的符号来说，它们都有自己特异的信息，比方说函数有参数以及参数类型，数组有维度，这种信息它们应该自己维护，所以每种特定的符号的符号表项应该是子类，继承自符号表项基类，这样做就比较容易加入新符号，修改表项也不会带来毁灭性的打击。
- 在设计完符号表项之后，一定要新建一个符号表类，然后把对符号表的各种插入删除查询操作都封装进去，不要仅仅用一个 `List` 或者 `Map` 之类的裸容器维护，要建立好抽象。

第二个难点应该就是如何去识别**跨语法成分的语义错误**，比方说 `break` 和 `continue` 不在循环里面之类的。这种错误实现的时候，一个简单的实现方法是设计一个全局标志位，然后递归往下分析。比方说，分析到 `while` ，就把全局标志位设置好（可以用累加器实现），离开 `while` 的时候减少一个全局标志位，当遇到 `break` 的时候，看全局标志位就知道自己现在在不在循环里面。再例如，在 `void` 函数中`return value`，也可以设计全局函数标志位，标志当前函数的返回值类型，在遇到 `return` 的时候检查是否匹配即可。

分享一下错误处理部分我遇到的 bug：

- 函数参数一维数组第一维省略，但我访问了 `null` 。
- `Block` 的作用域层次变化维护错误，应该把所有 `blockItem` 都分析完再把这层符号表弹出去。
- 未定义名字漏查。
- 某些域（比如函数参数表）可能为空导致的空指针异常。
- 迭代器删除。
- 调用错误处理方法处理正确程序，导致后面的一小部分忘记用来维护符号表。
- `g` 类错误理解偏差， `int func(){ return; }`不属于此类错误。
- `break` 和 `continue` 应该在函数结点就开始检查，不能在 `Block` 才开始检查。
- `void` 返回值的函数作为实参。
- `return` 相关的错误行号输出有问题，有的要输出 `return` 的行号，有的输出 `}` 的行号。
- `LVal` 的理解：不是`a[size1][size2]`，而是 `a[index1][index2]`，即表示的是一个单元而不是一个数组。
- 接上一条，实参`LVal` 的类型除了看名字之外，还得看后面有几对中括号，以确定该 `LVal` 的真实维数。

代码生成部分

在代码生成部分，课程组要求我们设计中间代码，然后把中间代码转化成目标代码。我相信大部分同学是第一次写编译器，所以可能对中间代码没什么感觉，更别谈设计了。这个时候大家可以采用“需要啥指令就设计啥指令”的思路去设计中间代码，先写一版能用的，然后大概就知道中间代码是怎么回事儿了。

由于中间代码是对目标代码的抽象，所以在设计中间代码的时候**不要去考虑细节**，比方说不要去想“我应该把传递的参数放到哪个地方”、“栈指针是怎么维护的”之类的问题，在中间代码阶段你可以简单生成一条 `passParam op` 这样的指令来“占位”，说明在这里我们需要做这个事情，但具体怎么做，等到生成目标代码的时候再把这些细节给实现出来。

另外，在写这部分的时候，一定要时刻提醒自己：**我现在只需要管编译期的事情，运行时怎么样我不关心**，不然很可能一直卡着写不出来。

关于符号表，由于要生成代码，所以符号表里面需要存储关于**变量的地址**以及**占用空间大小**的信息。对于**常量**，还需要保存所有**初始值**，之所以这么做，是因为可能用常量表示数组大小，而数组大小是编译期就应该知道的。另外，可以以函数为单位，去维护每个函数的局部符号表，注意及时删除过期符号。

关于寄存器和内存操作，在最开始做代码生成的时候，我们可以只用几个特定的寄存器，每次需要用值就 `lw` 进来，计算，然后把结果 `sw` 回去，每次计算只用固定的三个寄存器，除此之外只用 `$ra`，`$v0`， `$sp`， `$gp`等有特殊用途的寄存器。等到整个功能实现了，再想如何对寄存器做分配。

关于数组，这里有一个难点在于分析 `LVal` 的时候，有时候应该返回数组元素地址（等号左边），有时候应该返回数组元素值（等号右边），这里可以参考错误处理时使用的标志位的技巧，比方说分析赋值语句的时候，分析等号左边的内容，就把标志位给置位，这样在 `LVal` 里面就可以根据标志位判断到底该返回什么东西。

关于指针，有一个坑点是，指针本身是一个变量，这个变量其有一个**存储地址addr**，而这个变量的值呢，恰好也是一个**地址ptr**，所以想要得到“指针指向的那块内存区域的地址ptr”时，正确姿势是把这个指针当成普通变量，把它的值从addr里拿出来，就是ptr了。若要拿ptr表示的那块内存的值，则要以0为基地址，以ptr为偏移量进行load。

代码生成部分的bug非常多，下面我列举一下自己遇到的一点bug，希望能有用：

- getint读入之后要把值写到内存里面。
- printf输出要把值放到 `$a0` 里面。
- 数组不管是全局还是局部，都应该是**从低地址向高地址生长。**
- `f(a, g(b))` 调用时，在加载 `g` 的参数时，有可能会把 `a` 覆盖掉。
- `mips` 指令一般是 `name res op1 op2` 的格式，不要把结果当成第三个。
- 全局变量相对 `$gp` 正偏移，局部变量相对 `$sp` 负偏移。
- 数组的某个确定元素传参时应该传值而不是地址。
- 没有显式 `return` 的函数要手动 `return`。
- `and` 是按位与不是逻辑与。

测试

关于测试，想举一个例子说明一下，以代码生成1为例子，先把每种语法成分对应的需求写出来，然后用加法原理和乘法原理做排列组合：

变量定义：

- 类型上：
  - 局部变量定义
  - 局部常量定义
  - 全局变量定义
  - 全局常量定义
- 数量上：
  - 同一行内多个定义
- 初始值上：
  - 有无初值
    - 有的话是正负数，表达式，函数调用
- 名字重载作用域是否正确

表达式：

- 加减乘除取模是否全部实现了
- 指令的寄存器顺序是否正确
- `UnaryExp` 的连续符号
- 溢出?

函数定义：

- 有无参数
  - 0
  - 1
  - 多
- 有无返回值
  - void
  - int
- 返回值形式
  - 变量
  - 常数
  - 表达式
- 函数内部有无定义变量

函数调用：

- 实参类型：
  - 局部变量
  - 全局变量
  - 常数
  - 表达式
  - 函数返回值
    - 无参数函数
    - 单参数函数
    - 多参数函数
- 多层连续调用
- 函数参数压栈顺序
- 参数求值顺序（从左往右）

读入：

- 读给局部变量，全局变量

输出：

- 输出普通字符，空白符，转义字符（`\n`），表达式，函数调用；参数计算的顺序

```c
// #include <stdio.h>

const int global_const1 = 7;
const int global_const2 = -9, global_const3 = -+global_const1;

int global_var1 = 2;
int global_var2 = -21, global_var3 = -global_var2;
int global_var4 = global_var1 * global_var3;
int global_var5;

int overload_var = 10;

int init_tool() {
    return 345;
}

int test_overload() {
    return overload_var;
}

void non_return_non_param() {
    int a = 0;
    a = a + 1;
    printf("call non_return_non_param: %d\n", a);
}

void non_return_one_param(int a) {
    a = a + 1;
    printf("call non_return_one_param: %d\n", a);
}

void non_return_multi_params(int a, int b, int c) {
```

```c
    int res = a + b + c;
    printf("call non_return_multi_params: %d\n", res);
}

int has_return_non_param() {
    return 1;
}

int has_return_one_param(int a) {
    int useless = a;
    a = useless - 1;
    return a;
}

int has_return_multi_params(int a, int b, int c) {
    int useless = 10;
    useless = useless + a;
    return a - b + c;
}

int calculate_from_left_to_right(int a, int b) {
    return a - b;
}

int modify_global() {
    global_var1 = global_var1 + 3;
    return global_var1;
}

int in_func(int a, int b) {
    return a / b;
}

int out_func(int a) {
    int b = 3;
    return in_func(a, b);
}

int main() {
    int local_main_var1 = 2;
    int local_main_var2 = -9, local_main_var3, local_main_var4 =
local_main_var1;
    int temp;

    const int local_main_const1 = 3;
    const int local_main_const2 = -123, local_main_const3 = -
local_main_const2;

    // test global var init
    printf("------------test global var init start------------\n");
    printf("global_var_value: %d %d %d %d %d\n", global_var1, global_var2,
global_var3, global_var4, global_var5);

    printf("-------------test global var init end-------------\n\n");


    // test global const init
    printf("-----------test global const init start-----------\n");
```

```
    printf("global_const_value: %d %d %d\n", global_const1, global_const2, global_const3);
    printf("------------test global const init end------------\n\n");

    // test overload
    printf("----------------test overload start---------------\n");
    int overload_var = 7;
    printf("overload in main: %d\n", overload_var);
    printf("overload in global: %d\n", test_overload());
    {
        int overload_var = 8;
        printf("overload in main's sub-block: %d\n", overload_var);
    }
    printf("-----------------test overload end----------------\n\n");

    // test calculate
    printf("---------------test calculate start---------------\n");
    int res_add = local_main_var1 + local_main_var2;
    int res_sub = local_main_var1 - local_main_var2; // order
    int res_mul = local_main_var1 * local_main_var2;
    int res_div = local_main_var1 / local_main_var2;
    int res_mod = local_main_var1 % local_main_var2;
    printf("test calculate: %d %d %d %d %d\n", res_add, res_sub, res_mul, res_div, res_mod);
    int res = -(local_main_var1 +- local_main_var2) * (-local_main_const1 / +local_main_var1);
    printf("test mix calculate: %d\n", res);
    printf("-----------------test calculate end---------------\n\n");

    //test function call
    printf("-------------test function call start-------------\n");
    non_return_non_param();
    non_return_one_param(global_const1);
    non_return_multi_params(local_main_var1 - 1, global_var1, global_const1);
    printf("call has_return_non_param: %d\n", has_return_non_param());
    printf("call has_return_one_param: %d\n", has_return_one_param(has_return_non_param()));
    printf("call has_return_multi_params: %d\n", has_return_multi_params(local_main_var1, global_var1, global_const1));
    printf("call calculate_from_left_to_right: %d\n", calculate_from_left_to_right(
                modify_global(), global_var1));
    printf("call multi-call function: %d\n", out_func(1 + 5));
    printf("--------------test function call end--------------\n\n");

    // test getint()
    printf("-----------------test io start--------------------\n");
    temp = getint();
    global_var5 = getint();
    printf("temp is:\t%d\n", temp);
    printf("global_var5 is:\t%d\n", global_var5);
    printf("------------------test io end---------------------\n\n");

    return 0;
}
```

这样就可以比较系统地构造出一份测试数据。