MP1&MP2 实验报告

PB14011009 邹易澄

总述：

MP1 和 MP2 实验是实现 COOL 语言编译器的两个关键步骤。其中，MP1 完成了 COOL 编译器的前端，包括词法分析和语法分析，并生成了抽象语法树。MP2 以 MP1 生成的 AST 为基础，实现去除面向对象特性后的中间代码生成。用到的工具有 lex，bison，llvm，主要语言是 C++。

以下内容包括**设计细节**和**主要问题**两部分。

设计细节：

MP1.2

1. 参考 cool-manual，对常用的表达式或者 token 进行正规定义。

2. 处理标识符，关键字，运算符，空白符，无效字符以及整数常量的 token。

如类名标识符：<INITIAL>{typeid}　{yylval.symbol = idtable.add_string(yytext);return (TYPEID);}
其中，<INITIAL>表示在初始状态下识别后续的类名，之后通过 stringtab.h 里定义的方法，将类名标识符加入 idtable 中，通过 yylyal 返回。不考虑类名过长的情况。

其余的处理方案类似，详见 cool.flex 里的实现。

3. 记录行号，每一次读到换行符\n，使 curr_lineno 加 1。

4. 处理注释。

注释要考虑的错误处理较复杂，故单独处理，增加一个 COMMENT 的状态。每当在初始状态下识别到(*，则进入 COMMENT 状态。

通过引入对 comment 的计数来实现嵌套注释，当 comment 计数为 0 时，进入 INITIAL 状态。每读到(*，comment 加 1；每读到 *)，comment 减 1。若在 COMMENT 状态下读到 EOF (即文件结束符)，comment 置为 0，返回错误信息。

行注释较为简单，遇到--符号，则直到换行或者文件末尾，都为注释内容。

在 INITIAL 状态读到*)，表示遇到未匹配的注释符，返回错误信息。

5. 处理字符串常量。

字符串常量的错误处理较复杂，故单独处理，增加一个 STRING 的状态。每当在 INITIAL 状态下识别到 “，则进入 STRING 状态，并做一些初始化。

```
 /* 遇到“，表明字符串常量开始*/
<INITIAL>[\"] {
  BEGIN(STRING);
  string_buf_ptr = string_buf;  //初始化string_buf_ptr的指针,指向string_buf的第一个字符
  string_buf_valid = MAX_STR_CONST;  //string_buf剩余空间为最大
  string_error = false;    //无字符串错误
  null_error = false;      //无空字符错误
}
```

若在字符串里遇到 EOF，直接返回错误信息，并进入 INITIAL 状态。

若字符串中存在转义符，对转义符后出现的每一种情况都进行考虑，并单独处理。

字符串常量的识别用到了一个 string_buf 字符串数组，作为读取的字符串常量的缓冲区。

遇到 null 字符，无论是否在转义符之后，都生成错误信息，等到字符串结束后，再返回此错误信息。

string_buf 大小有上限，如果字符串大小超过此上限，则生成错误信息，等到字符串结束才返回错误信息，这在自定义的函数 string_write 里实现。

```
/* 填充字符串缓冲数组，形成一个string token */
int string_write(char *str, unsigned int len) {
  if (len < string_buf_valid) {
    strncpy(string_buf_ptr, str, len);
    string_buf_ptr += len;
    string_buf_valid -= len;
    return 0;
  } else {  //若剩余空间不够，则生成错误信息
    if(!null_error){    //若已经有了null错误信息，不再生成字符串过长的错误信息
      string_error = true;
      yylval.error_msg = "String constant too long";
      }
    return -1;
  }
}
```

如果遇到字符串过长又包含 NULL 字符，优先返回 NULL 的错误信息。所以设置了一个 null_error 的 BOOL 量，若已有 NULL 的错误，忽略字符串过长的信息。

一旦进入 STRING 状态，初始化 string_error 为 false，表示无字符串错误。之后如果遇到字符串过长或者包含无效字符 NULL，则置为 true。

若字符串结束 (即再次遇到”)，如果 string_error 为 true，返回错误信息，否则返回此 string_buf，写入 stringtable，作为一个 token。

若字符串里面包含非转义的换行，直接返回错误信息，进入 INITIAL 状态，直接开始下一行的词法分析。

MP1.3

1. 参考 COOL 的语法层次结构，定义 LALR 文法中的非终结符。在 bison 中声明它们的类型。

2. 参考 cool-manual 中的运算符优先级关系和左右结合关系，声明各算符的 precedence.

3. 参考 COOL 的语法层次结构，写一个 LALR 文法，并完成构建 AST 的语法制导定义。生成 AST 各结点的接口在 cool-tree.h 中已经定义。

其中，let 的语法本身具有二义性，在 bison 中表现为移进归约冲突。我通过对 IN 关键字添加 precedence 来解决此冲突，IN 应该是右结合的关系。

由于设计文法时考虑到避免移进归约冲突，最后生成的 parser 不包含移进归约冲突，我也没有多关注 bison 对于冲突出现后的缺省处理。

4. 实现粒度较细的错误恢复，使 parser 能够在检测到错误后继续分析。

以类错误处理为例。如果在一个 class 定义中遇到错误，这个类定义能被正确终止，并且能继续下一个类的分析。我在 class 这一层次进行错误的处理，它的产生式的第一个终结符是 CLASS。如果在类定义时出现错误，它不会管里面的 feature，而是一直读到这个类的结束，并规约成一个 class。并进行下一个 class 的分析。其中，我还特别考虑了 class 结尾丢失分号的情况。不过，此方案还是有些不足之处，

因为它假设类的结尾必有“;”或“}”符号。如果有人粗心大意在类结尾处把}和；都丢了，它会吃掉下一个 class 的分析。

```
// If no parent is specified, the class inherits from the Object class.
class  :  CLASS TYPEID '{' feature_list '}' ';'
                { $$ = class_($2,idtable.add_string("Object"),$4,
                                  stringtable.add_string(curr_filename)); }
        | CLASS TYPEID INHERITS TYPEID '{' feature_list '}' ';'
                { $$ = class_($2,$4,$6,stringtable.add_string(curr_filename)); }
        | CLASS TYPEID '{' feature_list '}' error
                { yyerrok; }
        | CLASS TYPEID INHERITS TYPEID '{' feature_list '}' error
                { yyerrok; }
        | CLASS error '{' error '}' ';'
                {}
        | CLASS error '{' error '}'
                {}
        ;
```

还有一种情况，如果在定义一个类的时候，把 class 关键字拼错，那就不能规约为一个 class 了。要解决这一问题，必须在 class_list 中定义 class 关键字拼错的处理方式。这里也默认类结尾必然有；或}符号。不然，parser 会吃掉下一个 class 的分析。还有一个问题是，如果在下一个 class 之前有“；”或者“}”符号，parser 会报多余的错误。

```
class_list
        :  class            // single class
                { $$ = single_Classes($1); }
        | class_list class // several classes
                { $$ = append_Classes($1,single_Classes($2)); }
        | error ';'
                {}
        | error '}'
                {}
        | class_list error ';'
                {}
        | class_list error '}'
                {}
        ;
```

类似地，parser 能从 feature、let 语句、{...}中的表达式中发生的错误中恢复，并跳到下一个对应层次的语法结构继续解析。具体特性如下：

1. 对于 feature_list 中的错误，除了能够识别属性错误(包括';'丢失)，还能进一步对方法中的 expr 进行粒度更细的错误恢复。

2. 对于{}中的错误，能够做到嵌套识别。如{ {a;} {+a;};}。

3. 对于 let，reference 的 parser 中似乎没有对它进行粒度更细的错误恢复。我实现的 parser 可以处理嵌套的 let 错误并恢复分析。

除了基本要求，我还实现了以下部分的错误恢复。

1. 对于 feature 中的 formal, 能正确识别其中的错误。如 a (b:Int, c): Int 中，能识别 c 处的错误并恢复分析。

2. 对于参数传递时用到的表达式 multi_expr，能正确识别其中的错误。如 a (b+w,+3)中, 能识别'+'的错误并恢复分析。

MP2

MP2 的实现思想并不特别困难，难点在于它的工程量。

1. 先用示例 cgen-1-ref 产生 IR，观察它的实现特性。

2. 先生成 main 类。通过 cgen-1-ref，得知它先 call 了 main_main 方法，其次调用 printf 打印 main_main 的返回值，再返回 0。

3. 搭好 Main_main 方法的框架，初始化 environment，为 Main_main 中的 codegen 做准备，并在最后定义 abort 标签，使得 main 方法中所有的错误都跳转至这个标签。

4. 实现 `Int` 和 `Bool` 的常数。作为 LLVM 的 `i32` 和 `i1` 生成它们。实现算术和比较运算。

5. 实现 block 表达式，这里需要用一个循环来遍历并生成各个 expr，最后返回的是最后一个 expr 的值。

6. 实现 `let`。用 `add_local` 在环境中记录变量名和内存地址的绑定关系。当 let 表达式结束，要删除这些变量名，因为这些变量名是局部的。

7. 实现赋值语句。 分清赋值的左边（LHS）和右边（RHS）。

8. 实现 loop。仿照 cegn-1-ref，先定义 loop 开始的标识，接着计算 cond 表达式，然后进行条件跳转。接着定义 true 和 false 的标识，完成每一个 block 的代码生成。

9. 实现 cond。仿照 cgen-1-ref，首先要判断 cond 表达式的返回类型，其次类比 loop 进行代码生成，关键还是在于 block 的布局，以及 IR 的生成顺序。

loop 和 cond 在实现的时候，要特别注意用简单的例子观察 ref 的 IR 生成结果。这样才能发现它的框架。

其中，我用到了流回退方式，使 cond 获得它的返回类型又不会生成两遍一样的 then_expr 代码。

10. 最后, 实现运行时的除 0 错误处理。其实，这本质上也是一种 cond 的代码生成。

主要问题：

MP1.2

1. 如何处理 comment 和 string 如此复杂的情况?

我在网上查看了 Lex 的手册，发现可以定义一些状态，用有限状态机的思想，来处理 commet 和 string 的识别和错误处理。如，我可以定义 COMMENT 状态，并且可以适时用 begin 来切换这些状态。这样，分状态处理就具有可行性了。

2. 做实验时，若干困扰我的小问题。

(1) 如果遇到“\<<EOF>>，会出现不明输出：\#1....。

Solution：
添加了如下语句后，输出就正常：

```
<STRING>[\\]          {}
```

因为之前的分析里面没有处理转义符\后直接 EOF 的情况，所以会出现这种问题。

(2) 有时候处理多文件会有莫名其妙的错误。

Solution：
原来在处理注释的时候，打开新文件后没有将 comment 归 0。所以它在处理新文件的时候，一旦进入 COMMENT 状态，就会出错。

MP1.3

1. let 表达式的行号显示有问题。

Solution:
虽然实验并没有特别要求行号正确，但是，我还是努力靠近 Reference 的 parser。具体现象是，当 let 语句中缺少赋值动作时，let_expr 右句型的文法动作是 * $$ = let($2,$4,no_expr(),$5)，此时会出现行号错误的问题。解决方法如下：
定义一个 none_expr。

```
| LET OBJECTID ':' TYPEID none_expr let_expr
        {  $$ = let($2,$4,$5,$6); }

none_expr :
                {  $$ = no_expr(); }
        ;
```

2. 类错误恢复一直达不到满意的效果。

No Solution.
Reference 里的 parser 对类的处理能力非常强，它总能以 class 关键字开始恢复分析。而我的 parser 默认类的结尾会有“；”或“}”符号。一旦下一个类的 class 关键字之前没有“；”或“}”，这个类会被吞掉而不进行分析。

3. 产生式结尾出错（如类结尾丢失分号），此时 parser 已读取下一个可能有用的 token，如何回退？

Solution：
通过咨询别的同学，我知道了可以用 yyerrok 这一个命令，回退一个 token，并恢复分析。当时，我想用 yyerrok 来处理问题二，但是 yyerrok 只回退出现 error 的 token，如果我显式地加入 CLASS 关键字，这个 CLASS 是 match 状态，无法回退。

MP2

1. 处理 cond 时，如何提前获知 if 表达式的类型，并且不用多次生成代码？

if 表达式有 then_expr 和 else_expr，我们已经假定这两个表达式有相同的类型。但是需要提前知道其中一个的类型，为之分配一个虚拟寄存器，然后 then 和 else 表达式将自己的结果存入该寄存器中（我想不通为何 MP2 没有提供 phi 的代码生成）。然后，一旦调用 code 方法获得 then_expr 的类型，就直接将 then_expr 的代码生成了，之后 then_expr 的 block 部分又会将代码生成一遍。

Solution：
使用 std::fpos 来记录之前的输出流，调用 code 方法获得 if 的类型之后，用 seekup(fpos)方法流回退。

2. valueprinter 提供的 Value_Printer_Counter 是 static 类型的，外部无法访问。导致用 ValuePrinter 标号时，无法手动修改标号值。

Solution：

我发现 ValuePrinter 提供了两个接口，其中一个调用 make_fresh_operand 方法会用到那个计数值。我于是调用另一个接口。并且在 CgenEnvironment 中定义了一个方法来代替 make_fresh_operand，此方法用到了 CgenEnvironment 中的 private 属性 tmp_counter。这么做的好处在于，采用流回退之前，我可以保存标号值，流回退之后恢复下一个标号，显得更加 elegant。不会出现%tmp1 之后突然出现%tmp3 的情况。

3. let 的实现比较曲折。一开始我没有搞清楚 let 的 init 部分包含多个声明的情况。还写了一个循环来获取 init 的每一个部分。结果生成的 IR 每次有问题。

Solution：

我仔细了解了一下 let 的 cool-tree 结构，才发现 init 部分是递归关系。只要再次调用 body->code()，便能继续 let 的代码生成。

总结：

通过 MP1 和 MP2 的实验，我对编译的前端和中间代码生成部分有了深刻的理解。中间的过程有些曲折，我常常咨询助教和同学。他们往往能提供一些好的思路和方法。在面对一个较大的工程时，首先要熟知里面的工具，清楚整个框架，才能着手去做。当然，编译的原理还是要熟知于心的，只有通过理论与实践结合，才能得到最大的进步。

参考文献：

https://guoxing.gitbooks.io/compiler-f2016/content/
里面有各种资源连接