PA2报告

韩晓宇18300750006

刘明君18307130064

1 代码核心逻辑

1.1 bison代码结构

首先简单介绍下代码结构

```
%{
  文字块，该部分的内容将直接复制到生成的代码文件的开头，以便它们在使用yyparse定义之前使用。
%}

定义部分，此部分主要包括选项、文字块、注释、声明符号、语义值数据类型的集合、指定开始符号及其它声明等等。

%%
语法规则
%%

文字块，该部分的内容将直接逐字复制到生成的代码文件末尾。该部分主要用于对之前一些声明了的函数进行实现。
```

这个PA中，我们主要的目标是完成`cool.y`中的语法规则部分，同时我们需要在定义部分中增添一些内容。

1.2 定义部分

为了完成我们接下来语法规则的编写，我们需要在定义部分定义一些表达式，具体的格式如下

```
%type <example_types> example
```

`example`是我们需要定义的表达式，而`example_types`则是该表达式的类型，具体类型已在代码中的`%union`列举。

1.3 cool语法结构

在cool语法之中，自上到下总共有5种结构，分别为program，class，feature，formal，expression，其中靠后的结构会包含在靠前的结构中的某些句法规则里，如

$$\begin{aligned} feature \quad ::= \quad & \text{ID}(\ [\ formal\ [\![, formal]\!]^*\ ]\ ) : \text{TYPE}\ \{\ expr\ \} \\ | \quad & \text{ID} : \text{TYPE}\ [\ \texttt{<-}\ expr\ ] \end{aligned}$$

可以看到feature的句法规则中包含了formal和expr。

在下文中我们会按照自上到下的顺序依次介绍每一种结构的实现方式。

1.4 代码结构

首先我们可以观察代码中所给的部分结构实现，即program和class结构，通过观察以及对于bison语言的manual阅读，我们可以大致总结出语法规则部分代码的写法。

```
example:
example结构的一个句法规则
{
    此处填写在遇到该规则后需要做什么，通常是调用某一函数来构造分析树的结点
}
| example结构的另一个句法规则
{
    此处填写在遇到该规则后需要做什么，通常是调用某一函数来构造分析树的结点
}
;
```

以上只是大致语法规则，我们还需要注意几点

1. $$为结果符号，在归约后其值由规约右部的符号值综合确定。
2. 以`CLASS TYPEID INHERITS TYPEID '{' feature_list '}' ';'`为例，`CLASS`和`INHERITS`定义时直接使用`%token`，因此直接匹配对应文字，而`TYPEID`及与其同类的`OBJECTID`定义时使用了`%token<symbol>`，因此匹配了`TYPEID xxx`及`OBJECTID yyy`，并将`xxx`和`yyy`作为其值，在此处我们就可以用`$2`和`$4`分别访问到第一个和第二个`TYPEID`的值，而`feature_list`的值可以用`$6`访问。
3. `class`结构的代码中调用了`class_`函数，该函数定义在`cool-tree.aps`文件中，文件中的其它函数与该函数一样，都是在建分析树的时候需要用到的函数，不过对于不同的语法结构及表达式可能会需要调用到不同的函数，同时，第二条中的值可以作为函数的参数。
4. `xxx_list`表明其为`xxx`结构组成的列表，以如下规则为例

$$class \quad ::= \quad \textbf{class}\ \text{TYPE}\ [\textbf{inherits}\ \text{TYPE}]\ \{\ [\![feature;]\!]^*\}$$

此处使用`feature_list`表明一个由`feature`结构构成的列表，在代码中利用递归来实现，这样可以更加方便地构造出每种结构的句法规则，不过需要注意列表是否可以为空，以及列表中是否有一些分隔符等条件，例如图中所示的`feature_list`便可以为空，且每个`feature`结构都由`;`隔开，但不同的列表要求不同，实现时需要注意。

1.5 核心思路

根据以上总结出来的代码规则，以及参考cool语言的manual，我们可以很容易地构造出每种结构的句法规则。

由于program和feature代码中已经实现，此处不再进一步介绍，下面介绍其它结构。

1.5.1 feature_list / feature

对于`feature_list`，上文有所介绍，其第一种情况为空，归约时让自己为空`feature`即可，具体方法为使用`$$ = nil_Features();`，第二种情况即为非空列表，根据句法规则我们可知，利用递归的方式可以定义该列表为`feature_list feature ';'`，处理上使用`append_Features`函数将`single_Feature`放入队列中即可。

对于`feature`，我们参考以下句法规则进行构造

$$\begin{aligned} feature \quad ::= \quad & \text{ID}(\ [\ formal\ [\![, formal]\!]^*\ ]\ ) : \text{TYPE}\ \{\ expr\ \} \\ | \quad & \text{ID} : \text{TYPE}\ [\ \texttt{<-}\ expr\ ] \end{aligned}$$

其中对于`[formal[[,formal]]*]`，我们构造`formal_list`来表示，由于`formal_list`是可选参数，我们使用两条规则分别实现有和无两种情况。同理，第二个句法规则中`<- expr`也是可选规则，用两条规则分别实现。

需要注意的是，根据`cool-parse.cc`中的定义，在构造对象前，我们需要使用`SET_NODELOC`来确保我们的行号不会出错。

在整个句法规则中，我们使用了`method()`和`attr()`函数，具体参数参考`cool-tree.aps`中的定义，不多赘述。

1.5.2 formal_list/formal

`formal_list`与`feature_list`基本相同，不过由于其中至少有一个元素，因此第一条规则并非为空`formal`，而是`formal`，后续第二条规则根据上述1.5.1中的图可知为`formal_list ',' formal`。

对于`formal`，其规则非常简单

$$formal \quad ::= \quad \text{ID} : \text{TYPE}$$

处理中使用`formal()`即可。

1.5.3 expression

$$
\begin{aligned}
expr \quad ::= \quad & \text{ID} \ \texttt{<-} \ expr \\
| \quad & expr[@\text{TYPE}].\text{ID}(\ [\ expr\ [\![, expr]\!]^*\ ]\ ) \\
| \quad & \text{ID}(\ [\ expr\ [\![, expr]\!]^*\ ]\ ) \\
| \quad & \textbf{if}\ expr\ \textbf{then}\ expr\ \textbf{else}\ expr\ \textbf{fi} \\
| \quad & \textbf{while}\ expr\ \textbf{loop}\ expr\ \textbf{pool} \\
| \quad & \{\ [\![expr;]\!]^+\} \\
| \quad & \textbf{let}\ \text{ID} : \text{TYPE}\ [\ \texttt{<-}\ expr\ ]\ [\![, \text{ID} : \text{TYPE}\ [\ \texttt{<-}\ expr\ ]]\!]^*\ \textbf{in}\ expr \\
| \quad & \textbf{case}\ expr\ \textbf{of}\ [\![\text{ID} : \text{TYPE} \texttt{=>} expr;]\!]^+\textbf{esac} \\
| \quad & \textbf{new}\ \text{TYPE} \\
| \quad & \textbf{isvoid}\ expr \\
| \quad & expr + expr \\
| \quad & expr - expr \\
| \quad & expr * expr \\
| \quad & expr / expr \\
| \quad & \tilde{}\,expr \\
| \quad & expr < expr \\
| \quad & expr <= expr \\
| \quad & expr = expr \\
| \quad & \textbf{not}\ expr \\
| \quad & (expr) \\
| \quad & \text{ID} \\
| \quad & \text{integer} \\
| \quad & \text{string} \\
| \quad & \textbf{true} \\
| \quad & \textbf{false}
\end{aligned}
$$

对于`[expr[,expr]]*]`，我们构造了`expression_list`来简化设计。（这边到时候根据实际情况来写，如果要改代码的话）

对于`[[expr;]]+`，我们构造了`blkexpression`来简化设计。

对于let语句，由于其中有两部分都属于可有可无，因此我们专门构造了`lets`来包含其所有可能的情况，这样可以简化`expression`规则的表达，在`lets`的一条规则中我们利用了递归的方式解决了句法规则中的`[[,ID:TYPE[ <- expr ]]]*`语句，对于其中的两条非递归规则，我们根据manual使用了`%prec flag`让语句中的let去匹配一句代码中最后面的`in`，用来消除歧义。

对于case语句，我们构造了`cases`来简化设计，其中针对`[[ID:TYPE => expr;]]+`，我们构造了`acase`用来表示case中的一种情况。

对于之中的其它句法规则，我们直接根据其含义进行构造，并调用`cool-parse.aps`中的函数即可。

1.6 行号处理

bison会自动把行号设置成constructor里面第一个参数的行号。cool的manual中并没有明确指出所有语法规则需要对应的行号，需要使用实例`line_no2.cool`和标准程序进行对照。

得到的结果为需要特别设置的是一些二元运算符，需要把行号设置到符号上面；dispatch需要设置到`.`上面，no_expr需要设置为0.其他的都不需要动。

1.7 移进/归约冲突问题

1.7.1 左递归/右递归

在递归的语法规则中，例如

```
words : word
        words word
```

有左递归和右递归两种写法，上面是一个左递归的例子，下面是右递归的例子。

```
words : word
        word words
```

在LL语法分析中，不允许使用左递归的语法规则。但是在Bison中使用LR算法进行语法分析，LR算法同时允许左递归和右递归的语法规则，并且相较而言，左递归的写法更好。比如在解析

```
word word word word $end
```

时，左递归会进行如下操作：

```
压栈 word
归约 words
压栈 words word
归约 words
压栈 words word
归约 words
压栈 words word
归约 words
```

而右递归需要：

```
压栈          word
压栈          word word
压栈          word word word
压栈          word word word word
看到$end，进行归约
              word word word words
              word word words
              word words
              words
```

可以发现当输入串变长时，左递归需要的栈深度是有限的，而右递归需要把所有符号入栈，需要更大的栈空间。

1.7.2 左因子

相较而言，左因子会在LR语法分析中导致移进/归约冲突，例如cool语言中嵌套的`let`语句：

```
let x <- 1,
y <-
let z <- 1 in w <- 1
in u <- 1
```

此时语义层面没有歧义，正确的解释应该是外层的`let`定义了`x,y`两个变量，和后面的`in`匹配，其中`y`的赋值为`let z <- 1 in w <- 1`。但是语法层面，不能唯一确定第一个`let`要和最远处的`in`匹配。这时就会出现移进/归约冲突。需要上文所述的`%prec flag`进行消歧义。

2 测试编写思路

2.1 good.cl

可编译的测试分为两种：正常的代码和用于测试边界情况的代码。这些代码应能够正常被parser解析，输出对应的抽象语法树。.

在`tests`文件夹中，正常的代码包括`arith.cool`,`atoi.cool`,`book_list.cl.cool`,`code.cl`,`good2.cl`,`hairyscary.cool`,`io.cool`,`life.cool`,`new_complex.col`,`palindrome.cool`,`sort_list.cl.cool`。

用于测试边界情况的代码有：`let.cool`用于测试`let`的嵌套；`line_num2.cool`包括了所有语法，每个语法的所有部分都独立作为一行，用于测试行号；`no_expr.cool`用于测试constructor `no_expr()`生成节点的行号。

2.2 bad.cl

`bad.cl`为不符合语法的一些测试。parser对于这些代码，应该在命令行中返回遇到的前50个语法错误，不输出抽象语法树。

我们构造了`bad1.cl`和`bad2.cl`进行测试。

其中`bad1.cl`包含了类的一些错误定义，如：保留字错误、类的变量名首字母小写、类定义的括号不匹配等。`bad2.cl`中测试了feature、let、表达式块的错误恢复，如：feature中某个类名首字母小写、let中某个类名首字母小写、let中某个对象名首字母大写、表达式的运算中对象名首字母大写、不合法的表达式等。发生了这些错误时，程序都可以从下一个同级语法中恢复。