

# 实验报告

## PA2

### 1. cool.y 代码及解释

#### 1.1 bison基础知识

Bison是一个语法生成器，在 .y 文件中按指定的格式写好相应的规则(对应上下文无关文法)后，Bison就能生成对应的语法分析器。在给定 .cl 文件后，词法分析器先将其分析为token序列，再输入到语法分析器，语法分析器就能生成对应的抽象语法树(AST)。

#### 1.2 Cool 抽象语法树

为了构建一棵抽象语法树，我们有必要了解对于Cool语言如何构建抽象语法树。Cool语言提供了对应的AST package，主要对应的文件为 cool-tree.aps 、 cool-tree.cc 。以下内容参考[cool-tour 6. Abstract Syntax Trees](#)。

具体来讲，AST结点被分为几种类型(Phylum)，如 Program\_ 、 Class\_ 等，这些类型对应到Cool语法中的非终结符(结构)，我们在后面会看到。

AST提供了 List 结构用于构造含多个元素的类型，如 Classes 是多个 Class\_ 的列表。List 还提供了一系列方法，具体见cool-tour 6.2。

最重要的就是AST提供的各种构造函数。对于一个特定的语法，我们可以将组成该语法的元素提取出来，然后调用对应的构造函数以生成对应的AST结点。例如，plus() 函数接收两个 Expression 结点参数，返回一个 Expression 结点，从而在语法上实现了加法。这种构造是递归的，从而最终能构造出一棵以 program 为根的抽象语法树。

#### 1.3 Cool Syntax

具体的语法见[cool manual figure 1](#)。我们需要按照该语法，在 .y 文件中写出对应的文法(生成式和对应的action)。

## 1.3.1 Program

`program` 代表整个程序，也是文法的**开始符号**。其应由一系列 `class` 构成：

$$program ::= \llbracket class; \rrbracket^+$$

例如，

```
Class A{
...
};
Class B{
...
};
```

因此对应的文法为

```
program : class_list
{ @$ = @1; ast_root = program($1); } ;
```

生成 `class_list`，并将该结点作为AST树的根节点。

## 1.3.2 Class

`program` 生成的 `class_list` 的生成式为：

```
class_list: class
{ $$ = single_Classes($1); parse_results = $$; }
| class_list class
{ $$ = append_Classes($1, single_Classes($2)); parse_results = $$; }
;
```

注意 `program` 应该至少有一个 `class`。

`class` 代表一个类。由**类名**、**继承的类名**(可选，只能单继承)、**类的成员**(feature, 可为空，也可多个)构成：

$$class ::= \text{class TYPE} [\text{inherits TYPE}] \{ \llbracket feature; \rrbracket^* \}$$

例如：

```
// 例1: 不继承
class A{
    ...
}
// 例2: 有继承
class B inherits A{
    ...
}
```

`class` 的生成式为:

```
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)); } ;
```

这里两条生成式的区别在于是否含有Inherits部分。注意最后要加上分号与 `program` 的要求对应。另外, 引入了 `feature_list`, 代表类成员(**feature**)的列表。

### 1.3.3 Feature

首先, `feature_list` 的生成式为:

```
feature_list:
    { $$ = nil_Features(); }
    | feature_list feature
    { $$ = append_Features($1,single_Features($2)); } ;
```

与 `class_list` 不同的是, `feature_list` 可以生成空符号, 因为 `class` 语法中的 `feature` 部分可以为空 (\*和+的区别)。

`feature` 是类的成员, 进一步可分为方法(**Method,也称函数**)和属性(**Attribute**)。

$$feature ::= ID([formal \llbracket, formal \rrbracket^*]) : TYPE\{expr\} \\ | ID : TYPE[<- expr]$$

`feature` 的语法中, 上面一条语法代表的是类的**方法**, 由**方法名**、**方法参数**(为空, 或多个, 中间用逗号隔开)、**方法返回类型**、**方法内容**构成, 例如:

```
A(b:Int,c:String):Object{
    ...
}
```

这里引入 `formal_list`，代表参数(**formal**)的列表。  
其对应的生成式为：

```
feature : OBJECTID '(' formal_list ')' ':' TYPEID '{' expr '}' ';'
{ $$ = method($1,$3,$6,$8); }
```

注意最后有分号，对应 `class` 语法中的要求。第二条语法代表的是类的**属性**，由**属性名**、**属性类型**和**初始化**(可选)构成，例如：

```
a : String           // 例1: 无初始化
b : Int <- 1          // 例2: 有初始化
```

其对应的生成式为(与上面那条生成式合并)

```
feature : OBJECTID ':' TYPEID ';'
{ $$ = attr($1,$3,no_expr()); }
| OBJECTID ':' TYPEID ASSIGN expr ';'
{ $$ = attr($1,$3,$5); } ;
```

这里两条分别代表是否赋值(初始化)，以及同样结尾需要有分号。

## 1.3.4 Formal

`formal_list` 的生成式为

```
formal_list :
{ $$ = nil_Formals(); }
| formal
{ $$ = single_Formals($1); }
| formal_list ',' formal
{ $$ = append_Formals($1,single_Formals($3)); } ;
```

表示 `formal_list` 可以不含有参数、含有一个或多个参数，其中每两个参数间用逗号隔开。  
`formal` 代表方法(函数)的参数。其语法为

$$formal ::= ID : TYPE$$

表示每个参数的形式为**参数名：参数类型**。例如：

```
// 括号里的内容即formal_list
func(a : Int , b : String):Int{
    ...
}
```

对应的生成式为：

```
formal : OBJECTID ':' TYPEID { $$ = formal($1,$3); } ;
```

## 1.3.5 Expression

`expression` 是最常用的形式，代表任何可能的表达式。其语法如下：

```
expr ::= ID <- expr
      |  expr[@TYPE].ID( [ expr [ , expr ] * ] )
      |  ID( [ expr [ , expr ] * ] )
      |  if expr then expr else expr fi
      |  while expr loop expr pool
      |  { [expr; ] + }
      |  let ID : TYPE [ <- expr ] [ , ID : TYPE [ <- expr ] ] * in expr
      |  case expr of [ ID : TYPE => expr; ] + esac
      |  new TYPE
      |  isvoid expr
      |  expr + expr
      |  expr - expr
      |  expr * expr
      |  expr / expr
      |  ~expr
      |  expr < expr
      |  expr <= expr
      |  expr = expr
      |  not expr
      |  (expr)
      |  ID
      |  integer
      |  string
      |  true
      |  false
```

我们需要特殊处理的只有：**函数调用、表达式块、let表达式、case表达式**

$$expr ::= expr[@TYPE].ID([expr \[, expr \]] *) \\ | ID([expr, \[, expr \]] *)$$

```

| { [[expr;]]+ }
| let ID : TYPE [<- expr] [[, ID:TYPE [<- expr]]* in expr
| case expr of [[ID:TYPE => expr;]]+ esac

```

因为这些语法里都含有多个 `expr`，需要对应生成不同的 `expr_list` 来处理。  
 对于剩下的这些简单语法，可以直接写出其生成式：

```

expr : OBJECTID ASSIGN expr
    { $$ = assign($1,$3); }          // example : a <- 1
    | IF expr THEN expr ELSE expr FI
    { $$ = cond($2,$4,$6); }          // example : if a > 1 then
                                      // a <- 1 else a <- 2 fi
    | WHILE expr LOOP expr POOL
    { $$ = loop($2,$4); }              // example : while a < 10 LOOP
                                      // a <- a + 1 POOL
    | NEW TYPEID
    { $$ = new_($2); }                 // example : new A
    | ISVOID expr
    { $$ = isvoid($2); }               // example : isvoid(a)
    | expr '+' expr
    { $$ = plus($1,$3); }              // example : 1 + 2
    | expr '-' expr
    { $$ = sub($1,$3); }               // example : 2 - 1
    | expr '*' expr
    { $$ = mul($1,$3); }               // example : 1 * 2
    | expr '/' expr
    { $$ = divide($1,$3); }            // example : 4 / 2
    | '~' expr
    { $$ = neg($2); }                  // example : ~2
    | expr '<' expr
    { $$ = lt($1,$3); }                // example : 1 < 2
    | expr 'LE' expr
    { $$ = leq($1,$3); }               // example : 1 <= 2
    | expr '=' expr
    { $$ = eq($1,$3); }                // example : 2 = 2
    | NOT expr
    { $$ = comp($2); }                 // example : not a
    | '(' expr ')'
    { $$ = $2; }                       // example : (a)
    | OBJECTID
    { $$ = object($1); }               // example : a
    | INT_CONST
    { $$ = int_const($1); }            // example : 1
    | STR_CONST
    { $$ = string_const($1); }         // example : "hello"
    | BOOL_CONST

```

```
{ $$ = bool_const($1); }    // example : true
;
```

对于**函数调用语法**，需要引入新的 Expressions : `dispatch_exprs`。生成式为

```
expr: expr '.' OBJECTID '(' dispatch_exprs ')'
    { $$ = dispatch($1,$3,$5); }
| expr '@' TYPEID '.' OBJECTID '(' dispatch_exprs ')'
    { $$ = static_dispatch($1,$3,$5,$7); }
| OBJECTID '(' dispatch_exprs ')'
    { $$ = dispatch(object(idtable.add_string("self")), $1,$3); } ;
```

第一个生成式是普通调用；第二个生成式在Cool中称为**静态调用**，使得表达式可以调用其父类的方法(这个在语义分析阶段检查)；第三个生成式省略了调用的表达式，默认应为 `self`。(具体见4.2 `self dispatch`)

其中， `dispatch_exprs` 应写为

```
dispatch_exprs :
    { $$ = nil_Expressions(); }
| expr
    { $$ = single_Expressions($1); }
| dispatch_exprs ',' expr
    { $$ = append_Expressions($1,single_Expressions($3)); } ;
```

这与 `feature` 语法中对 `formal` 的要求一样，也是没有或多个，每两个间用逗号隔开。

第二个生成式是表达式块；其形式和 `program` 的语法是一样的，生成式为

```
expr : '{' block_exprs '}'
    { $$ = block($2); };
block_exprs : expr ';'
    { $$ = single_Expressions($1); }
| block_exprs expr ';'
    { $$ = append_Expressions($1,single_Expressions($2)); }
```

主要区别在于需要在 `expr` 后间添加分号，因为并不是每个普通 `expr` 后都有分号。

第三个生成式是 `let` 表达式。因为LET后面的内容有明显的重复内容，可以写为



```

expr : LET let_expr
{ $$ = $2; };
let_expr : OBJECTID ':' TYPEID ASSIGN expr IN expr
{ $$ = let($1,$3,$5,$7); }
| OBJECTID ':' TYPEID IN expr
{ $$ = let($1,$3,no_expr(),$5); }
| OBJECTID ':' TYPEID ',' let_expr
{ $$ = let($1,$3,no_expr(),$5); }
| OBJECTID ':' TYPEID ASSIGN expr ',' let_expr
{ $$ = let($1,$3,$5,$7); } ;

```

这里实际上 `let_expr` 是 `Expression` 而非 `Expressions`，因为 `let` 函数的要求中没有 `Expressions` 类别(按照 `Cool-tour` 的要求，`let` 表达式转化为嵌套的单个 `let` 表达式)。另外为了满足最后 `in expr` 的要求，这里只能写成右递归形式。(如果将 `IN expr` 写在 `expr` 的生成式中，`let_expr` 的单条生成式中就没有足够的信息构造 `let` 结点了)

第四个生成式是 `case` 表达式。`typcase` 构造函数需要 `Cases` 类型，生成式为

```

expr: CASE expr OF case_list ESAC
{ $$ = typcase($2,$4); };
case_list : case
{ $$ = single_Cases($1); }
| case_list case
{ $$ = append_Cases($1,single_Cases($2)); } ;
case : OBJECTID ':' TYPEID DARROW expr ';'
{ $$ = branch($1,$3,$5); } ;

```

注意每个 `case` 的生成式右部最后要加上分号。

## 1.3.6 错误处理

当给定的 `.cl` 文件出现语法上的错误时，我们应该能识别出，并采取对应正确的动作(如从哪里开始继续考察)。我们并不需要识别是哪种错误，[PA3 handout 5.Error Handling](#)指出，我们可以用 `error` 来指代错误情况。[Bison manual 6.Error Recovery](#)中指出，`error` 是一个terminal,当出现语法错误(syntax error)会自动生成。我们主要是针对部分错误指定正确的回应，而**大多数错误并不需要特意对应添加规则**。

需要注意的是，`error` 也是规则的一部分，它也会作为被归约的一部分。因此，**在子结点中出现的错误也作为在父结点中的错误，最基本的则是token错误**，这一点尤其需要注意。

1. 当class定义中出现错误，则应该考察下一个class。

```
class :error ';'
  {} ;
```

2. 当feature中出现错误，应考察下一个feature。

```
feature : error ';'
  {} ;
```

3. 当let表达式中出现错误，应考察下一个变量。这里，如果是最后一个变量的表达式出错了，就不需要考察下一个变量了。

```
let_expr:error ',' let_expr
  { $$ = $3; }
  | error IN expr
  {} ;
```

4. 当block内表达式出错，应该查看block内下一个表达式。

```
block_exprs: error ';'
  {}
```

以上4类错误是在PA3 handout中提到的。除此之外，还有一些类型的可能的错误如下，主要是针对有多个表达式的情况下的处理。相应的处理方式是经过与官方的parser的报错信息对比得到的。

1. 当调用(dispatch)内出现错误，经测试，此时直接考察下一个feature(我觉得，这有可能是因为我们定义了feature出错的情况，此时出错被归约到feature对应的生成式上，所以采取的动作是考察下一个feature。至少官方Parser的表现是这样)，调用内部及该feature剩余的内容不再考察。因此我们不需要添加任何错误规则。
2. 当case表达式的branch出错，应查看下一个branch。事实是官方parser确实直接考察下一个branch，且与默认表现一致，因此我们实际上也不需要添加任何错误规则。(加了错误规则后反而表现不一致了，具体原因未知...)
3. 当formal(函数定义的参数)中出错时，经测试，此时应跳过剩下的所有formals，考察函数内部。我的想法是，在feature的错误规则中添加一条：

```
feature: error '{' expr '}' ';'
  {}
```

这样就保证了能在错误后，考察大括号内部的表达式。

## 2. 测试用例

需要说明的一点是，由于语法生成阶段不涉及语义(Semantics)的检查，因此测试文件并不需要完全正确，比如返回值不需要匹配函数声明的类型、使用变量不需要声明等，因为对这些的检查在语义阶段完成。

### 2.1 good.cl

对于 `good.cl`，首先我考虑测试了全部的语法规则，保证能正确处理所有情况。我将对**每个语法的每种情况测试**尽量都写在不同的函数内，避免杂糅，这样找起错误来会方便很多。

我着重测试了一些嵌套的情况，如下：

#### 1. if...then...else...fi嵌套

```
if_then_else_nest_test(a:Int,b:Int):Int {  
    if a < b then a + b  
    else if b < a then a - b  
    else if a = b then a * b  
    else a / b  
    fi fi fi  
};
```

#### 2. let 嵌套

```
let_nest_test():Int {  
    let a:Int <-3 in let b:Int <-1 in let c:Int<-2,d:Int in a+b+c+d  
};
```

#### 3. block 嵌套

```
block_nest_test():Int {  
    {  
        {  
            1;  
        };  
    };  
    {  
        2;  
    };  
    3;  
}  
};
```

## 2.2 bad.cl

bad.cl 需要尽可能涵盖尽可能多的错误情况。相应的错误处理方法已经在[1.3.6 错误处理](#)给出了总结。相应的部分构造案例如下：

1. class错误(各个位置出错)

```

// error : keyword Class is misspelled
Crass t{

};

// error:  b is not a type identifier
Class b inherits A {
};

// error:  a is not a type identifier
Class C inherits a {
};

// error:  keyword inherits is misspelled
Class D inherts A {
};

// error:  closing brace is missing
Class E inherits A {
;

```

## 2. feature错误(分为属性错误和方法错误)

```

// error : feature error
class E {
    // attr
    A : Int;           // A is not object ID
    e : Int <- 1;
    C : int;           // int is not type ID
    f : Int <- 1;
    // method
    D():Int{           // D is not object ID

    };
};

```

## 3. formal错误(即参数)

```
// formal test
class F {
  f(Int a,b:Int,c:Int <- 1,d:Int):Int{
    // 1. incorrect formal style
    // 2. cannot assign initial value

  };
};
```

4. let错误，看是否会报多次错，从而说明是考察的下一个变量。

```
let_test():Int {
  let A in let b:Int,c,d:Int,e in let c:Int in let d:Int <- ? in c
};
```

5. block错误，看是否是考察的下一行。如果不是，应该只会报一次错，否则会报两次错。这里将两个表达式写在一行是为了检查错误发生后的继续情况。

```
block_test():Int {
  {
    1+1;A+B;
    2+2;
    C+D;
  }
};
```

6. case错误，这里将两个case写在一行是为了检查错误发生后的继续情况。

```
// see next feature
case_test():Int {
  case a of
    b : Int => b;c : Int -> c;
    d : Int => d;
    e : int => e;
  esac
};
```

7. dispatch错误，分静态和动态

```
// only first error, then next feature
dispatch_test():Int {
    f(?;b,c;d)
    f@B(?)
    g(?)
    e(?)
};
```

8. 表达式错误，测试一下parser对应的行为

```
// only first error, then next feature
expr_test():Int {
    A+B
    C+D
};
```

## 3. 测试结果及解释

### 3.1 good.cl

输入 `./checker good.cl` 的对比结果如下：

```
● breezer@ubuntu:~/CompilerPA/assignments/PA3$ ./checker good.cl
13c13
<      #3
---
>      #0
401c401
<      #87
---
>      #0
441c441
<                #91
---
>                #0
```

我们查看对应生成的AST树，区别在

```

1 #1
2 program
3 #1
4 class
5   A
6   Object
7   "good.cl"
8   (
9     #3
10    attr
11    a
12    String
13    #0
14    _no_expr
15    : _no_type
16    #4
17    attr
18    b
19    Int
20    #4
21    int
22    1
23    : _no_type
24    #6

```

```

1 #1
2 program
3 #1
4 class
5   A
6   Object
7   "good.cl"
8   (
9     #3
10    attr
11    a
12    String
13    #3
14    _no_expr
15    : _no_type
16    #4
17    attr
18    b
19    Int
20    #4
21    int
22    1
23    : _no_type
24    #6

```

这三处不同都是同样的情况，即行号不同(在 `tree.h` 中我们能看到所有AST树结点的公共父类 `tree_node`，它有一个属性就是行号 `line_number`)。查看了对应 `cool.y` 的代码后，对应的问题在于：当我们传入 `no_expr()` 作为参数时，官方的parser会将其行号标识为0，而我的会将行号标识为程序中对应的行号。

相应的处理见[4.3 关于传入no\\_expr的行号问题](#)。我个人认为，将 `no_expr` 的行号标记为0的理由在于因为 `no_expr()` 实际并不在原本的程序中出现，其行号本来就没有意义。

修改后，结果如下。可以看到得到了正确的结果。

```

breezer@ubuntu:~/CompilerPA/assignments/PA3$ ./checker good.cl
passed

```

## 3.2 bad.cl

输入 `./checker bad.cl` 的对比结果如下：

```

breezer@ubuntu:~/CompilerPA/assignments/PA3$ ./checker bad.cl
passed

```

能正确识别错误及对应行号。

## 3.3 最终parser对 good.cl 和 bad.cl 的输出结果

执行 `./myparser good.cl` 和 `./myparser bad.cl` 就能查看相应的结果，结果如下图( `good.cl` 因为结果太长，只给出部分)



```

_leq
#152
_int
1
: _no_type
#154
_int
1
: _no_type
: _no_type
#156
_eq
#155
_int
1
: _no_type
#157
_int
1
: _no_type
: _no_type
#158
_comp
#159
_int
1
: _no_type
: _no_type
#161
_int
1
: _no_type
#163
_isvoid
#164
_int
1
: _no_type
: _no_type
#165
_new
A
: _no_type
: _no_type
)

```

⊗ breezer@ubuntu:~/CompilerPA/assignments/PA3\$ ./myparser bad.cl

```

"bad.cl", line 15: syntax error at or near TYPEID = Crass
"bad.cl", line 24: syntax error at or near OBJECTID = a
"bad.cl", line 28: syntax error at or near OBJECTID = inherts
"bad.cl", line 37: syntax error at or near CLASS
"bad.cl", line 46: syntax error at or near TYPEID = C
"bad.cl", line 49: syntax error at or near TYPEID = D
"bad.cl", line 57: syntax error at or near TYPEID = Int
"bad.cl", line 59: syntax error at or near TYPEID = A
"bad.cl", line 65: syntax error at or near TYPEID = A
"bad.cl", line 65: syntax error at or near ','
"bad.cl", line 65: syntax error at or near IN
"bad.cl", line 65: syntax error at or near ERROR = ?
"bad.cl", line 71: syntax error at or near TYPEID = A
"bad.cl", line 73: syntax error at or near TYPEID = C
"bad.cl", line 80: syntax error at or near '-'
"bad.cl", line 81: syntax error at or near DARROW
"bad.cl", line 82: syntax error at or near OBJECTID = int
"bad.cl", line 88: syntax error at or near ERROR = ?
"bad.cl", line 96: syntax error at or near TYPEID = A
Compilation halted due to lex and parse errors

```

## 4. 遇到的问题及思路

### 4.1 Shift/Reduce Conflicts

在使用 `bison cpol.y` 时会提示有 `Shift/Reduce Conflicts`。因为没有进一步的提示，所以我甚至不知道冲突出在哪里。在 [Bison manual 8.2 Understanding your parser](#) 中，提到可以输入 `bison --report=state cool.y` 来将具体的冲突信息报告在 `cool.output` 中。结果如下：

```
State 106 conflicts: 9 shift/reduce
State 133 conflicts: 9 shift/reduce
State 146 conflicts: 9 shift/reduce
State 106

19 expr: expr . '.' OBJECTID '(' dispatch_exprs ')'
20     | expr . '@' TYPEID '.' OBJECTID '(' dispatch_exprs ')'
29     | expr . '+' expr
30     | expr . '-' expr
31     | expr . '*' expr
32     | expr . '/' expr
34     | expr . '<' expr
35     | expr . LE expr
36     | expr . '=' expr
54 let_expr: error IN expr .

LE      shift, and go to state 62
'<'    shift, and go to state 63
'='    shift, and go to state 64
'+'    shift, and go to state 65
'-'    shift, and go to state 66
'*'    shift, and go to state 67
'/'    shift, and go to state 68
'@'    shift, and go to state 69
'.'    shift, and go to state 70

LE      [reduce using rule 54 (let_expr)]
'<'    [reduce using rule 54 (let_expr)]
'='    [reduce using rule 54 (let_expr)]
'+'    [reduce using rule 54 (let_expr)]
'-'    [reduce using rule 54 (let_expr)]
'*'    [reduce using rule 54 (let_expr)]
'/'    [reduce using rule 54 (let_expr)]
'@'    [reduce using rule 54 (let_expr)]
'.'    [reduce using rule 54 (let_expr)]
$default reduce using rule 54 (let_expr)
```

另外两个状态的冲突类似。我们可以看到，冲突的原因是这些**运算符表达式**与**let表达式**产生了

**Shift/Reduce冲突**。而在[Bison manual 5.2 Shift/Reduce Conflicts](#)中提到，冲突很多时候是由于运算符(非运算符的优先级)问题。因此，我们有必要为 `in` 设定优先级。设置为：

```
%right IN
```

并将其置于所有优先级最顶部。这样，在同样的状态下，语法分析器就会优先选择归约到 `let_expr`，从而消除了冲突。

## 4.2 self dispatch

`dispatch`的有一种形式是没有调用`Object`的，此时默认调用者为`self`。但 `dispatch()` 构造函数的第一个参数总是要求为调用者的 `Expression`，且此时也不应该传入 `no_expr()`。

此时，我们可以仿照 `class` 对应生成式传入参数的方法：利用 `idtable.add_string()` 方法，在 `dispatch()` 的第一个参数处也可以传入 `idtable.add_string("self")`。但还有一个问题：这里 `add_string()` 返回的类型是一个 `Symbol`，但要求的参数类型是 `Expression`。

因此，我们还需要调用 `object()` 构造函数，该构造函数本来是用于将标识符构造为一个表达式的，其接收一个 `Symbol` 类型，将其转换为 `Expression` 类型，从而在这里能满足 `dispatch()` 函数的要求。

因此，最后构造函数的写法为：

```
dispatch(object(idtable.add_string("self")),...)
```

## 4.3 传入no\_expr的行号问题

问题描述见[3.1](#)。如果我们想与官方parser保持一致的话，我们需要在传入 `no_expr()` 作为参数的对应位置将该AST结点的行号设置为0。可以通过 `SET_NODELOC()` 函数来完成。该函数将决定当前结点行号的 `node_lineno` 设定为指定值。

但我们并不能简单地在构造函数前设置行号，因为这会导致该结点对应的子树的所有行号都为0，而我們需要的只是 `no_expr` 对应的结点行号为0。这又该如何着手呢？因为 `no_expr()` 并没有对应的实际参数。

我考虑的方法是，**直接对`no_expr()`函数本身进行修改**，具体如下：

```

Expression no_expr()
{
    int temp = node_lineno;
    node_lineno = 0;
    no_expr_class* noexpr = new no_expr_class();
    node_lineno = temp;
    return noexpr;
}

```

其中，需要声明 `extern int node_lineno;`。我们先保存原本行号，然后设置 `node_lineno` 为0，再构造 `no_expr` 对应的结点，然后恢复行号。

## 4.4 关于 `bad.cl` 的feature分号问题

我在 `bad.cl` 里添加这样一个错误示例，使得一个 `feature` 遗漏掉结尾的分号。

```
e : Int
```

此时的结果为

```

● breezer@ubuntu:~/CompilerPA/assignments/PA3$ ./checker bad.cl
8,9d7
< "bad.cl", line 57: syntax error at or near TYPEID = Int
< "bad.cl", line 59: syntax error at or near TYPEID = A

```

<pre> 1 "bad.cl", line 15: syntax error at or near TYPEID = Crass 2 "bad.cl", line 24: syntax error at or near OBJECTID = a 3 "bad.cl", line 28: syntax error at or near OBJECTID = inherts 4 "bad.cl", line 37: syntax error at or near CLASS 5 "bad.cl", line 46: syntax error at or near TYPEID = C 6 "bad.cl", line 49: syntax error at or near TYPEID = D 7 "bad.cl", line 53: syntax error at or near '}' 8 "bad.cl", line 65: syntax error at or near TYPEID = A 9 "bad.cl", line 65: syntax error at or near ',' 10 "bad.cl", line 65: syntax error at or near IN 11 "bad.cl", line 65: syntax error at or near ERROR = ? 12 "bad.cl", line 72: syntax error at or near TYPEID = A 13 "bad.cl", line 74: syntax error at or near TYPEID = C 14 "bad.cl", line 82: syntax error at or near '-' 15 "bad.cl", line 83: syntax error at or near DARROW 16 "bad.cl", line 84: syntax error at or near OBJECTID = int 17 "bad.cl", line 90: syntax error at or near ERROR = ? 18 "bad.cl", line 98: syntax error at or near TYPEID = A 19 Compilation halted due to lex and parse errors 20 </pre>	<pre> 1 "bad.cl", line 15: syntax error at or near TYPEID = Crass 2 "bad.cl", line 24: syntax error at or near OBJECTID = a 3 "bad.cl", line 28: syntax error at or near OBJECTID = inherts 4 "bad.cl", line 37: syntax error at or near CLASS 5 "bad.cl", line 46: syntax error at or near TYPEID = C 6 "bad.cl", line 49: syntax error at or near TYPEID = D 7 "bad.cl", line 53: syntax error at or near '}' 8 "bad.cl", line 57: syntax error at or near TYPEID = Int 9 "bad.cl", line 59: syntax error at or near TYPEID = A 10 "bad.cl", line 65: syntax error at or near TYPEID = A 11 "bad.cl", line 65: syntax error at or near ',' 12 "bad.cl", line 65: syntax error at or near IN 13 "bad.cl", line 65: syntax error at or near ERROR = ? 14 "bad.cl", line 72: syntax error at or near TYPEID = A 15 "bad.cl", line 74: syntax error at or near TYPEID = C 16 "bad.cl", line 82: syntax error at or near '-' 17 "bad.cl", line 83: syntax error at or near DARROW 18 "bad.cl", line 84: syntax error at or near OBJECTID = int 19 "bad.cl", line 90: syntax error at or near ERROR = ? 20 "bad.cl", line 98: syntax error at or near TYPEID = A 21 Compilation halted due to lex and parse errors 22 </pre>
--	--

可以看到，我的输出多了两行，对应到原来的 `bad.cl` 中，我认为我的报错更合理，因为官方parser直接跳过了下面的一整个class，进而忽略了这两处错误。具体原因我暂时不清楚。

## 4.5 表达式结点行号问题

我在 good.cl 的最后补充了这样一个示例：

```
multilines_test():Int {           // 第133行
    {                             // 第134行
        1                         // 第135行
        +                         // 第136行
        1;                       // 第137行
        2 +                       // 第138行
        2;                       // 第139行
        3                         // 第140行
        + 3;                     // 第141行
    }                             // 第142行
};                               // 第143行
```

主要是想测试当表达式跨越多行时，其行号为哪一行。结果如下：

The screenshot displays a debugger window with two panes. The left pane shows the source code of the `multilines_test` function, with line numbers 133 through 143. The right pane shows the corresponding AST nodes, with line numbers 614 through 654. Red boxes highlight the line numbers for the plus nodes: #136 for the first plus (line 136), #141 for the second plus (line 141), and #140 for the third plus (line 140).

Source Line	AST Line	AST Node
133	614	method
134	615	newline_test
135	616	Int
136	617	#134
137	618	block
138	619	plus
139	620	#135
140	621	int
141	622	1
142	623	: _no_type
143	624	#137
144	625	int
145	626	1
146	627	: _no_type
147	628	: _no_type
148	629	#138
149	630	plus
150	631	#138
151	632	int
152	633	2
153	634	: _no_type
154	635	#139
155	636	int
156	637	2
157	638	: _no_type
158	639	: _no_type
159	640	#141
160	641	plus
161	642	#140
162	643	int
163	644	3
164	645	: _no_type
165	646	#141
166	647	int
167	648	3
168	649	: _no_type
169	650	: _no_type
170	651	: _no_type
171	652	: _no_type
172	653	: _no_type
173	654	)

可以看到，三个表达式的行号在我的语法生成器(即bison默认设定下)，都为**表达式开始时的行号**(准确来说，bison默认将AST结点的(开始)行号设定为第一个参数的行号，参考[Bison manual 3.5.4 Default Action for Locations](#))，而官方parser的行号为**运算符对应的行号**。

在 cool.y 的Prologue部分有一大段注释来说明**如何正确设定结点的行号**。为了达到官方parser的效果，我们只需要在加法规则对应的action处进行一点修改：

```
expr '+' expr
{ SET_NODELOC(@2); $$ = plus($1,$3); }
```

即将该结点的行号设置为运算符的行号。需要注意的是，很多规则(如减法等二元运算符规则)都需要做同样的修改，限于篇幅不多展示。

## 5. 总结与感想

本次实验要通过自己编写上下文无关文法来实现一个语法生成器。从实现上来说，写一个与给定的Cool语法匹配的文法其实是不难的，难的是各种各样的细节，尤其是在**错误处理**这方面非常难以理解与实现，很难达到官方parser的水准。在实验过程中，起到帮助的一是各类手册，二是官方parser的表现。以下是我在实验过程中的一些思考：

- 对于Shift/Reduct Conflicts或Reduce/Reduct Conflicts，具体来讲是怎样的Conflict？
- 我们可以通过修改 `node_lineno` 来调整每个AST结点的行号，而对于那些没有具体规则对应的(内置)结点，要怎样修改其行号？(直接修改源函数!)
- 错误(error)具体来说是以怎样的形式检测的？又是如何影响文法的构造的？在出错的时候是应用的哪条错误规则...