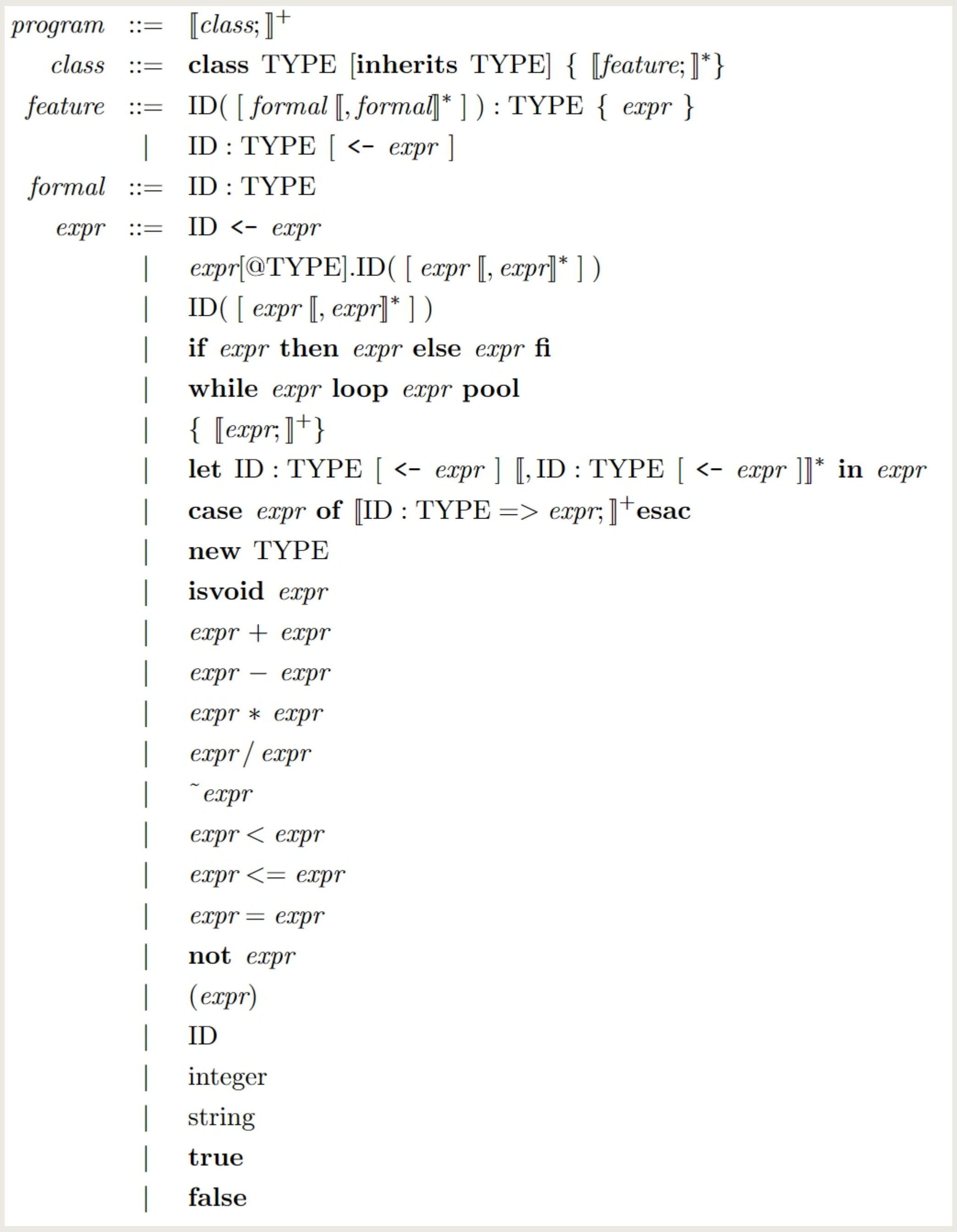
Programming Assignment III

15-李晓畅-凌之寒

1.1 语法分析器实现与 cool.y 代码简述

在 cool-manual 中给出了 COOL 语言的语法结构：



```
program ::= [[class;]]+
  class ::= class TYPE [inherits TYPE] { [[feature;]]* }
feature ::= ID( [ formal [[, formal]]* ] ) : TYPE { expr }
          | ID : TYPE [ <- expr ]
 formal ::= ID : TYPE
   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
```

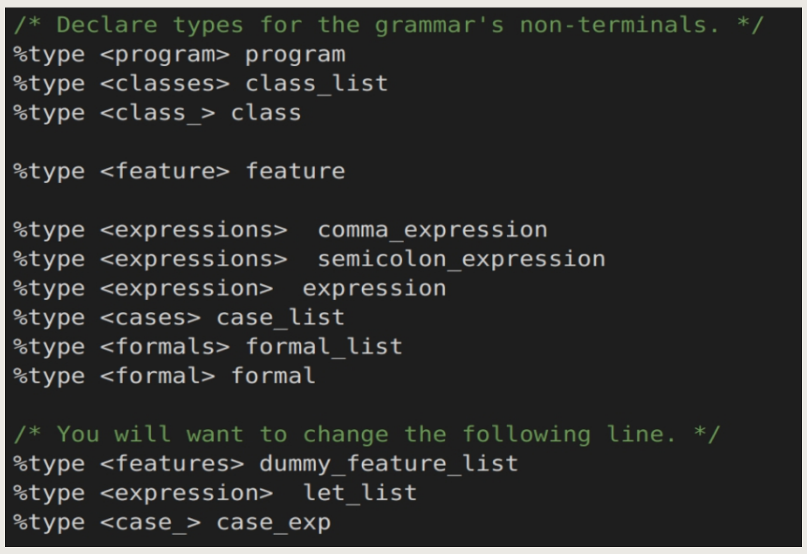
大体来说，除了将上面的语法结构转化为 bison 表达外，额外的工作就只有消除可能的冲突。另外需要注意的是：

在实现每条语法时需要额外考虑错误的情况；

上述的语法结构包含正则表达式，而这需要额外的方式解析成 bison 可以理解的形式；

在消除冲突时可以考虑添加优先级要求。

对应到设计上，我们为每一个语法结构设计一组对应的表达，同时为其中的一些正则表达式，尤其是闭包表达式，添加额外的规则识别即可。如下图声明的非终结符所示：



```
/* Declare types for the grammar's non-terminals. */
%type <program> program
%type <classes> class_list
%type <class_> class

%type <feature> feature

%type <expressions>  comma_expression
%type <expressions>  semicolon_expression
%type <expression>  expression
%type <cases> case_list
%type <formals> formal_list
%type <formal> formal

/* You will want to change the following line. */
%type <features> dummy_feature_list
%type <expression>  let_list
%type <case_> case_exp
```

总之大体上同参考中的图片一一对应。

1.1.1 优先级声明

```
/* Precedence declarations go here. */
%left ASSIGN
%left NOT
%nonassoc '=' LE
%nonassoc IN
%left '<'
%left '+' '-'
%left '*' '/'
%left ISVOID
%left '~'
%left '@'
%left '.'
```

有的其实不必要，但是符合使用习惯；有的是为了解决规约冲突添加的，比如 IN 就是为了解决一个移进规约冲突设计的。

1.1.2 Feature 和 formal 实现

Program 和 Class 已经由框架给出，并指点我们应该构造一个 dummy_feature_list 的非终结符来识别 feature 的克林闭包，一个 class_list 来识别 calss 的非空闭包。这实际上给出了实现词法分析器时会用到的几种方式。

下面以 Feature 非终结符的识别为例解释：

空 feature 的情况可以单独列出在之前解决更方便：

```
class
: CLASS TYPEID '{' '}' ';'
{ $$ = class_($2,idtable.add_string("Object"),nil_Features(),
stringtable.add_string(curr_filename)); }
```

好吧为什么我非要这样说是因为我们在后面遇到了类似情况报的冲突都是这样处理的，其他解决方法不是完全不行但这样比较方便理解，或许并非完全依照语法结构最方便。T_T

在消除空 feature 后上文提及的 feature 只会以大于一个的形式出现，因而只会有两种情况，即只包含一个 feature 和一个 feature 列表，一个分号和一个 feature。如下所示：

```
/* Feature list may be empty, but no empty features in list. */
dummy_feature_list:
feature ';'
{ $$ = single_Features($1); }
| dummy_feature_list feature ';'
{ $$ = append_Features($1,single_Features($2));}
```

解析 feature：

```
feature : OBJECTID ':' TYPEID
{ $$ = attr($1, $3, no_expr());}
| OBJECTID ':' TYPEID ASSIGN expression
{ $$ = attr($1, $3, $5);}
| OBJECTID '(' ')' ':' TYPEID '{' expression '}'
{ $$ = method($1, nil_Formals(), $5, $7);}
| OBJECTID '(' formal_list ')' ':' TYPEID '{' expression '}'
{ $$ = method($1, $3, $6, $8);}
```

参考语法结构，formal 列表不能为空，所以这里同上面的 class 不一样，直接写 formal_list 就可以了。

所以在 formal_list 这里的构造其实同上面类似就可以了。

```
formal_list : /*it should not be empty!*/
formal
{ $$ = single_Formals($1);}
| formal_list ',' formal
{ $$ = append_Formals($1, single_Formals($3));}

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

Formal 表达式直接翻译成 CFG 就可以了。

总结来说就是大体上直接翻译即可，处理闭包我们做了特殊处理，克林闭包我们的设计都是将空表达式的匹配提前，其余同非空闭包即可。

1.1.3 Expression 实现

Expression 的实现大体可以分为几个部分：

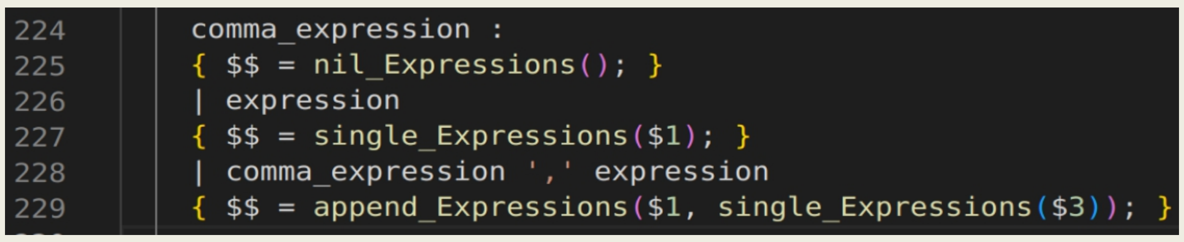
逗号表达式`comma_expression :`；

分号表达式`semicolon_expression`；

`let_list`let 表达式；

`case_list` `case_exp`case 列表及单个情况共同组成的 case 表达式。

上面提到这些包含正则表达式的列表实现上类似于：



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

Comma_expression 比较特殊可以允许为空，但其他总之我们在实现上都不允许在这一层面为空，而是将其在上面一层侦测到空。

以及包含或由上面组成的表达式和一些可以直接翻译的表达式组成的 expression，大概这个样子：

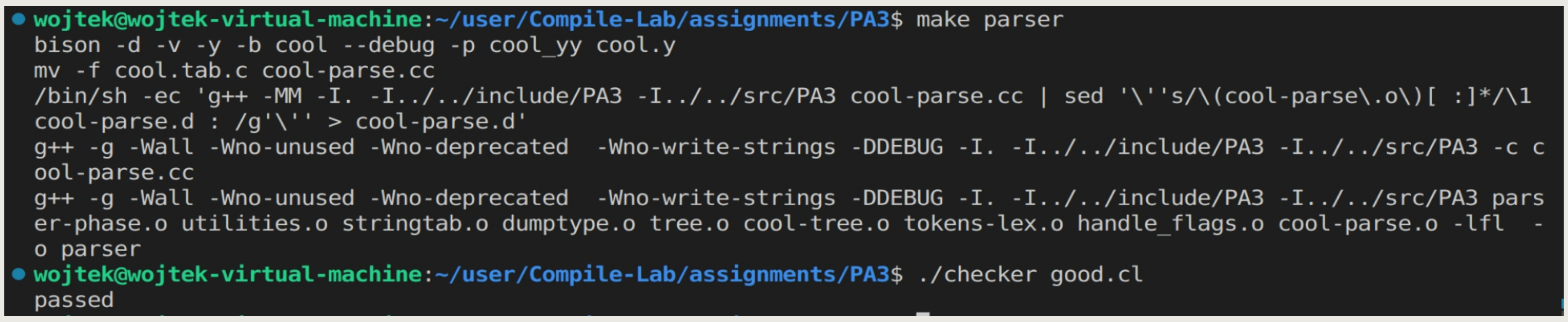
```
expression : OBJECTID ASSIGN expression
{ $$ = assign($1, $3);}
/* function */
| expression '.' OBJECTID '(' comma_expression ')'
{ $$ = dispatch($1, $3, $5);}
| expression '@' TYPEID '.' OBJECTID '(' comma_expression ')'
{ $$ = static_dispatch($1, $3, $5, $7);}
| OBJECTID '(' comma_expression ')'
{ $$ = dispatch(object(idtable.add_string("self")), $1, $3);}
/* if */
| IF expression THEN expression ELSE expression FI
{ $$ = cond($2, $4, $6);}
/* while */
| WHILE expression LOOP expression POOL
{ $$ = loop($2, $4);}
/* {block} */
| '{' semicolon_expression '}'
{ $$ = block($2);}
/* let */
| LET let_list
{ $$ = $2;}
/* case */
| CASE expression OF case_list ESAC
{ $$ = typcase($2, $4);}
/* other boring things OMG */
| NEW TYPEID
{ $$ = new_($2);}
| ISVOID expression
```

共同实现了 expression,转换方法同上面提及的一致。

具体代码可参附录 1。

1.1.4 验证

按照顺序执行指令：



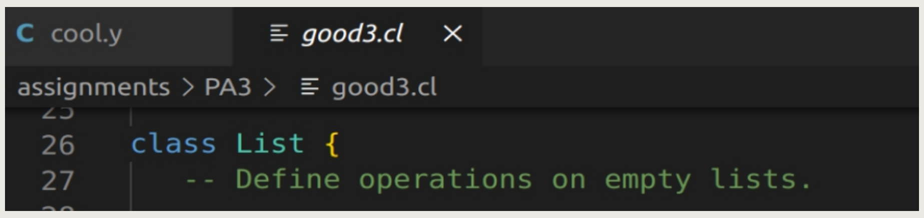
```
wojtek@wojtek-virtual-machine:~/user/Compile-Lab/assignments/PA3$ make parser
bison -d -v -y -b cool --debug -p cool_yy cool.y
mv -f cool.tab.c cool-parse.cc
/bin/sh -ec 'g++ -MM -I. -I../../include/PA3 -I../../src/PA3 cool-parse.cc | sed '\''s/\(cool-parse\.o\)[ :]*/\1
cool-parse.d : /g'\'' > cool-parse.d'
g++ -g -Wall -Wno-unused -Wno-deprecated  -Wno-write-strings -DDEBUG -I. -I../../include/PA3 -I../../src/PA3 -c c
ool-parse.cc
g++ -g -Wall -Wno-unused -Wno-deprecated  -Wno-write-strings -DDEBUG -I. -I../../include/PA3 -I../../src/PA3 pars
er-phase.o utilities.o stringtab.o dumptype.o tree.o cool-tree.o tokens-lex.o handle_flags.o cool-parse.o -lfl  -
o parser
wojtek@wojtek-virtual-machine:~/user/Compile-Lab/assignments/PA3$ ./checker good.cl
passed
```

结果同预期一致，总之就是没什么问题，可以正确对样例进行语法分析。

2. 测试样例构建

2.1 good.cl 构造

其实我也不知道什么是好，所以在这一样例的构造上我们实际上主要参考了官方结构给出的一些代码，比如这段：

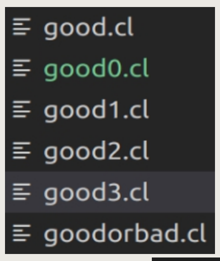


cool.y　good3.cl ×

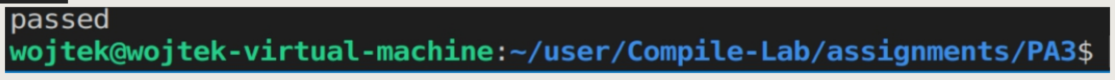
assignments > PA3 > good3.cl

```
26  class List {
27     -- Define operations on empty lists.
```

官方用于解释如何构造 list 的代码就是我们的一个测试。另一些事例代码同样在我们的测试程序范围内。



```
good.cl
good0.cl
good1.cl
good2.cl
good3.cl
goodorbad.cl
```

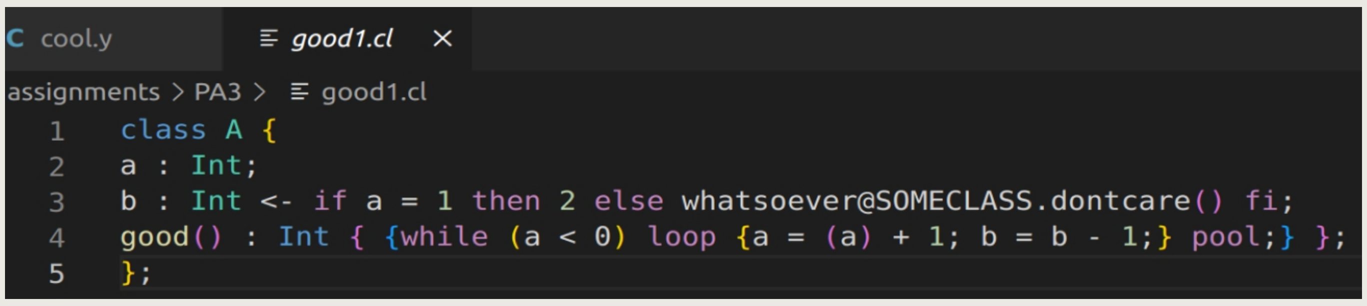
正确通过。

```
passed
wojtek@wojtek-virtual-machine:~/user/Compile-Lab/assignments/PA3$
```

这大概可以说明我们可以处理几乎所以常见的表达了。

我们就不为 good.cl 单独附加代码了，因为绝大多数您都可以在示例中找到。

除了这些外，我们还为示例代码未涉及的一些不太常用的语法构造了样例，比如：

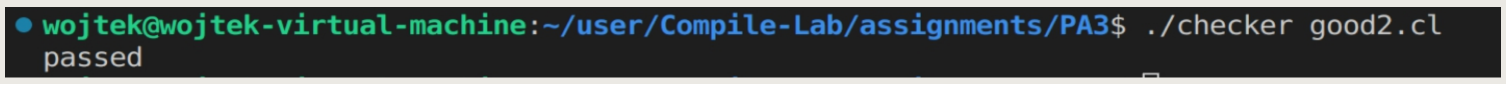


cool.y　good1.cl ×

assignments > PA3 > good1.cl

```
class A {
a : Int;
b : Int <- if a = 1 then 2 else whatsoever@SOMECLASS.dontcare() fi;
good() : Int { {while (a < 0) loop {a = (a) + 1; b = b - 1;} pool;} };
};
```

同样可以通过。

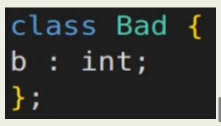


```
wojtek@wojtek-virtual-machine:~/user/Compile-Lab/assignments/PA3$ ./checker good2.cl
passed
```

2.1 bad.cl 构造

样例的 bad.cl 包含关于类本身很多问题的检查，在此不做赘述。与之对应，我们将为其他部分进行检查，包括：

比如对于 expression,我们构造了很多样例包括：



```
class Bad {
b : int;
};
```

匹配不上；

```
--An empty block is the worst thing I have ever seen.

class Bad {
b : Int <- if a = 1 then 2 else {} fi;
};
```

```
--Notgood is good

class Bad {
b : Int <- if a = 1 then 2 else a@notgood.dontcare() fi;
};
```

```
--An empty block is the worst thing I have ever seen.

class Bad {
b : Int <- if a = 1 then 2 else {} fi;
};
```

```
--Why are you doing this? {I mean the semicolomn;}

class Bad {
b : Int <- if a = 1 then 2 else {c = 99999} fi;
};
```

等等。其他具体的错误情况可参考附录 2。

客观说，这些错误我自己写的时候也会犯 qwq。

3. 对规约冲突的处理

在实验中我们同样遇到了一些移进/规约冲突，为此总结了两种比较有效的处理方法：

其一，规定优先级，比如：

```
let_list
: OBJECTID ':' TYPEID IN expression
```

原始设计中因为 IN 不是算符，所以我们没有为它设计优先级规定，但随后发现冲突，

其原因是显然的，因为 IN 之后单个 expression 和多个 expression 列表的情况我们是无法区分的。

解决方案也是简单的，添加一个：`%nonassoc IN`即可。

其二，将对某些规则的处理提前，比如：

我们原本将 feature_list 实现为可以为空的，形如：

```
dummy_feature_list:
{  $$ = nil_Features(); }
```

但这会出现冲突，于是我们将对空的识别提前到前一状态，比如：

```
class
: CLASS TYPEID '{' '}' ';'
{ $$ = class_($2,idtable.add_string("Object"),nil_Features(),
stringtable.add_string(curr_filename)); }
| CLASS TYPEID INHERITS TYPEID '{' '}' ';'
{ $$ = class_($2,$4,nil_Features(),stringtable.add_string(curr_filename)); }
```

就可以很好处理这种情况了。

4.实验总结

本次实验中我们运用 bison 工具实现了 COOL 语言的语法分析器。

bison 是一个语法分析器的生成器。简单来说，通过给定语法的产生式开始，bison 会通过算法，最终构造得到动作表，然后利用这个动作表去解析句子。具体来说，bison 读取用户提供的语法的产生式，生成一个 C 语言格式的动作表，并将其包含进一个名为 yyparse 的 C 函数，这个函数的作用就是利用这个动作表来解析 token 流，而这个 token 流是由 flex 生成的词法分析器扫描源程序得到的。

可以看出，COOL 简洁的语法结构同其精巧的设计密不可分。我们实际上可以将其描述为仅仅五个非终结符之间的状态转移构成的 CFG，很难相信这是一个包含很多现代编程语言特性、支持面向对象的语言。其简单的根源在于，同书写上方便相比，语法规定更多考虑了将类似的表达归为同一状态，很多不符合一般习惯的语法有助于减少状态数量，比如将{ /*balabala*/ }解释为一个表达式而非一个类似于 C 语言中的代码课实际上就允许我们将对其的识别归为对 expression 的识别。

总之在实际分析语法结构，并转化为 bison 表达的过程中，我们深刻体会到精巧设计的意义。

另一方面，在处理冲突的过程中我们最终可以总结出常用的解决方法，这体现了一种对问题本质的理解，启发我们探索问题根本的特性。

附录

1. 附录 1 cool.y 详细代码

```
/*
*  cool.y
*              Parser definition for the COOL language.
*
*/
%{
  #include <iostream>
  #include "cool-tree.h"
  #include "stringtab.h"
  #include "utilities.h"

  extern char *curr_filename;


  /* Locations */
  #define YYLTYPE int              /* the type of locations */
  #define cool_yylloc curr_lineno  /* use the curr_lineno from the lexer
  for the location of tokens */

    extern int node_lineno;          /* set before constructing a tree node
    to whatever you want the line number
    for the tree node to be */


      #define YYLLOC_DEFAULT(Current, Rhs, N)         \
      Current = Rhs[1];                             \
      node_lineno = Current;


    #define SET_NODELOC(Current)  \
    node_lineno = Current;

    /* IMPORTANT NOTE ON LINE NUMBERS
    *********************************
    * The above definitions and macros cause every terminal in your grammar to
    * have the line number supplied by the lexer. The only task you have to
    * implement for line numbers to work correctly, is to use SET_NODELOC()
    * before constructing any constructs from non-terminals in your grammar.
    * Example: Consider you are matching on the following very restrictive
    * (fictional) construct that matches a plus between two integer constants.
    * (SUCH A RULE SHOULD NOT BE  PART OF YOUR PARSER):

    plus_consts : INT_CONST '+' INT_CONST

    * where INT_CONST is a terminal for an integer constant. Now, a correct
    * action for this rule that attaches the correct line number to plus_const
    * would look like the following:

    plus_consts : INT_CONST '+' INT_CONST
    {
      // Set the line number of the current non-terminal:
      // ***********************************************
      // You can access the line numbers of the i'th item with @i, just
      // like you acess the value of the i'th exporession with $i.
      //
      // Here, we choose the line number of the last INT_CONST (@3) as the
      // line number of the resulting expression (@$). You are free to pick
      // any reasonable line as the line number of non-terminals. If you
      // omit the statement @$=..., bison has default rules for deciding which
      // line number to use. Check the manual for details if you are interested.
      @$ = @3;


      // Observe that we call SET_NODELOC(@3); this will set the global variable
      // node_lineno to @3. Since the constructor call "plus" uses the value of
      // this global, the plus node will now have the correct line number.
      SET_NODELOC(@3);

      // construct the result node:
      $$ = plus(int_const($1), int_const($3));
    }

    */



    void yyerror(char *s);        /*  defined below; called for each parse error */
    extern int yylex();           /*  the entry point to the lexer  */

    /************************************************************************/
    /*                DONT CHANGE ANYTHING IN THIS SECTION                  */

    Program ast_root;         /* the result of the parse  */
    Classes parse_results;        /* for use in semantic analysis */
    int omerrs = 0;               /* number of errors in lexing and parsing */
    %}

    /* A union of all the types that can be the result of parsing actions. */
    %union {
      Boolean boolean;
      Symbol symbol;
      Program program;
      Class_ class_;
      Classes classes;
      Feature feature;
      Features features;
      Formal formal;
      Formals formals;
          Case case_;
          Cases cases;
          Expression expression;
          Expressions expressions;
          char *error_msg;
        }

        /*
        Declare the terminals; a few have types for associated lexemes.
        The token ERROR is never used in the parser; thus, it is a parse
        error when the lexer returns it.

        The integer following token declaration is the numeric constant used
        to represent that token internally.  Typically, Bison generates these
```

```
on its own, but we give explicit numbers to prevent version parity
problems (bison 1.25 and earlier start at 258, later versions -- at
257)
*/
%token CLASS 258 ELSE 259 FI 260 IF 261 IN 262
%token INHERITS 263 LET 264 LOOP 265 POOL 266 THEN 267 WHILE 268
%token CASE 269 ESAC 270 OF 271 DARROW 272 NEW 273 ISVOID 274
%token <symbol>  STR_CONST 275 INT_CONST 276
%token <boolean> BOOL_CONST 277
%token <symbol>  TYPEID 278 OBJECTID 279
%token ASSIGN 280 NOT 281 LE 282 ERROR 283

/*  DON'T CHANGE ANYTHING ABOVE THIS LINE, OR YOUR PARSER WONT WORK       */
/**************************************************************************/

/* Complete the nonterminal list below, giving a type for the semantic
value of each non terminal. (See section 3.6 in the bison
documentation for details). */

/* Declare types for the grammar's non-terminals. */
%type <program> program
%type <classes> class_list
%type <class_> class

%type <feature> feature

%type <expressions>  comma_expression
%type <expressions>  semicolon_expression
%type <expression>  expression
%type <cases> case_list
%type <formals> formal_list
%type <formal> formal

/* You will want to change the following line. */
%type <features> dummy_feature_list
%type <expression>  let_list
%type <case_> case_exp

/* Precedence declarations go here. */
%left ASSIGN
%left NOT
%nonassoc '=' LE
%nonassoc IN
%left '<'
%left '+' '-'
%left '*' '/'
%left ISVOID
%left '~'
%left '@'
%left '.'

%%
/*
Save the root of the abstract syntax tree in a global variable.
*/
program : class_list    { @$ = @1; ast_root = program($1); };

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

/* If no parent is specified, the class inherits from the Object class. */
```

```
class
: CLASS TYPEID '{' '}' ';'
{ $$ = class_($2,idtable.add_string("Object"),nil_Features(),
stringtable.add_string(curr_filename)); }
| CLASS TYPEID INHERITS TYPEID '{' '}' ';'
{ $$ = class_($2,$4,nil_Features(),stringtable.add_string(curr_filename)); }
| CLASS TYPEID '{' dummy_feature_list '}' ';'
{ $$ = class_($2,idtable.add_string("Object"),$4,
stringtable.add_string(curr_filename)); }
| CLASS TYPEID INHERITS TYPEID '{' dummy_feature_list '}' ';'
{ $$ = class_($2,$4,$6,stringtable.add_string(curr_filename)); }

/* Feature list may be empty, but no empty features in list. */
dummy_feature_list:
{  $$ = nil_Features(); }
| feature ';'
{ $$ = single_Features($1); }
| dummy_feature_list feature ';'
{ $$ = append_Features($1,single_Features($2));}

feature : OBJECTID ':' TYPEID
{ $$ = attr($1, $3, no_expr());}
| OBJECTID ':' TYPEID ASSIGN expression
{ $$ = attr($1, $3, $5);}
| OBJECTID '(' ')' ':' TYPEID '{' expression '}'
{ $$ = method($1, nil_Formals(), $5, $7);}
| OBJECTID '(' formal_list ')' ':' TYPEID '{' expression '}'
{ $$ = method($1, $3, $6, $8);}

formal_list : /*it should not be empty!*/
formal
{ $$ = single_Formals($1);}
| formal_list ',' formal
{ $$ = append_Formals($1, single_Formals($3));}

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

// exp_list : '{' exp_list  exp ';' '}'
// { $$ = append_Expressions( $2, single_Expressions($3));}
// | exp ';'
// { $$ = single_Expressions($1);}

comma_expression :
{ $$ = nil_Expressions(); }
| expression
{ $$ = single_Expressions($1); }
| comma_expression ',' expression
{ $$ = append_Expressions($1, single_Expressions($3)); }

expression : OBJECTID ASSIGN expression
{ $$ = assign($1, $3);}
/* function */
| expression '.' OBJECTID '(' comma_expression ')'
{ $$ = dispatch($1, $3, $5);}
| expression '@' TYPEID '.' OBJECTID '(' comma_expression ')'
{ $$ = static_dispatch($1, $3, $5, $7);}
| OBJECTID '(' comma_expression ')'
{ $$ = dispatch(object(idtable.add_string("self")), $1, $3);}
/* if */
| IF expression THEN expression ELSE expression FI
{ $$ = cond($2, $4, $6);}
/* while */
| WHILE expression LOOP expression POOL
{ $$ = loop($2, $4);}
/* {block} */
| '{' semicolon_expression '}'
{ $$ = block($2);}
/* let */
```

```
            | LET let_list
            { $$ = $2;}
            /* case */
            | CASE expression OF case_list ESAC
            { $$ = typcase($2, $4);}
            /* other boring things OMG */
            | NEW TYPEID
            { $$ = new_($2);}
            | ISVOID expression
            { $$ = isvoid($2);}
            | expression '+' expression
            { $$ = plus($1, $3);}
            | expression '-' expression
            { $$ = sub($1, $3);}
            | expression '*' expression
            { $$ = mul($1, $3);}
            | expression '/' expression
            { $$ = divide($1, $3);}
            | '~' expression
            { $$ = neg($2);}
            | expression '<' expression
            { $$ = lt($1, $3);}
            | expression LE expression
            { $$ = leq($1, $3);}
            | expression '=' expression
            { $$ = eq($1, $3);}
            | NOT expression
            { $$ = comp($2);}
            | '(' expression ')'
            { $$ = $2;}
            | OBJECTID
            { $$ = object($1);}
            | INT_CONST
            { $$ = int_const($1);}
            | STR_CONST
            { $$ = string_const($1);}
            | BOOL_CONST
            { $$ = bool_const($1);}

            semicolon_expression
            : expression ';'
            { $$ = single_Expressions($1); }
            | semicolon_expression expression ';'
            { $$ = append_Expressions($1, single_Expressions($2)); }

            let_list
            : OBJECTID ':' TYPEID IN expression
            { $$ = let($1, $3, no_expr(), $5); }
            | OBJECTID ':' TYPEID ASSIGN expression IN expression
            { $$ = let($1, $3, $5, $7); }
            | OBJECTID ':' TYPEID ',' let_list
            { $$ = let($1, $3, no_expr(), $5); }
            | OBJECTID ':' TYPEID ASSIGN expression ',' let_list
            { $$ = let($1, $3, $5, $7); }
            | error ',' let_list
            { $$ = $3; }

            case_list
            : case_exp
            { $$ = single_Cases($1); }
            | case_list case_exp
            { $$ = append_Cases($1, single_Cases($2)); }

            case_exp
            : OBJECTID ':' TYPEID DARROW expression ';'
            { $$ = branch($1, $3, $5); }

            //------------------------------------------

```

```
      //-----------------------------------------

      /* end of grammar */
      %%

      /* This function is called automatically when Bison detects a parse error. */
      void yyerror(char *s)
      {
        extern int curr_lineno;

        cerr << "\"" << curr_filename << "\", line " << curr_lineno << ": " \
        << s << " at or near ";
        print_cool_token(yychar);
        cerr << endl;
        omerrs++;

        if(omerrs>50) {fprintf(stdout, "More than 50 errors\n"); exit(1);}
      }


```

2. 附录 2 bad.cl 详细代码

```
--I have to say that Im getting bored of this.
--It just cant continue parsing after an error.
--So we need to make 1 thousand badX.cl(s)??? bruh

class Bad {
b : int;
};

--Notgood is good

class Bad {
b : Int <- if a = 1 then 2 else a@notgood.dontcare() fi;
};

--An empty block is the worst thing I have ever seen.

class Bad {
b : Int <- if a = 1 then 2 else {} fi;
};

--Why are you doing this? {I mean the semicolomn;}

class Bad {
b : Int <- if a = 1 then 2 else {c = 99999} fi;
};

--Swimming pool

class Bad {
a() : Int {
  while (a < 0) loop {a = (a) + 1; b = b - 1;}
};
};

--Who put that ';' over there ?!?

class Bad {
a() : Int {
  while (a < 0) loop {a = (a) + 1; b = b - 1;} pool;
};


--Are you doing this on purpose or by mistake?

```

```
class While {
};

class goodguy {
a : Esac;
};

--So we are back in the mine~

class Creeper {
explode(player : Player, power : Int) : Int { player.setlifestate("dead") };
};

class Bad {
a : Creeper;
me : Player;
enrage(b : Creeper) : Int {a.explode(me, b.explode(me.mine, yourbloodpressure))};
};

--Im quiting.

class Bad {
funny : Int <- a(a(a(a(a(a))))));
};

--Bye

class Bad {
funny : Int <- a(a(a(a(a(a))));
};
```