编译原理PJ2：语法分析

20300240006 吴骁

20307130001 蔡哲飚

一.实验目的

1.学习bison和flex的配合与使用

2.通过flex与bison，分析目标PCAT语言，并生成目标语言的语法树

3.通过实验，学习在进行完词法分析后进行语法分析的具体实现

二.具体实现

2.1Bison的使用方法

Bison 文件的结构可以分为四个部分:

- Prologue：定义了一些 Bison 头文件和源文件中需要使用的函数和变量，一般通过 include 外部宏的方式实现。
- Bison declarations：声明了所有终结符和非终结符以及它们语义值的类型，同时指定了一些操作符的优先级。此外这里也包含了 Bison 的各种设置。
- Grammar rules：定义了语法分析的具体规则，即每个非终结符有哪些产生式，匹配到相应产生式时需要采取什么操作等等。
- Epilogue：用户可以在这里定义一些辅助函数，此部分代码会被直接复制到 `src/parser.cpp` 里。

```
%{
// Prologue
%}

// Bison declarations

%%
// Grammar rules
%%

// Epilogue
```

2.1.1Prologue

在本项目中，Prologue被分为两部分

```
%code requires {
  #include <string>
  #include "ast/ast.h"

  class Driver;
  namespace yy {
      class Lexer;
  }
}
```

这部分代码会直接被复制到`src/syntactic_parser.h`的头部,代码中include了`<string>`和`ast/ast.h`,`<string>`中的`std::string`所有终结符的语义值,`ast/ast.h`包含了所有非终结符的语法书节点。

```
%code top{
  #include <memory>
  #include "driver.h"
  #include "lexer.h"
  #include "utils/logger.h"

  using std::make_shared;
  using location_type = yy::Parser::location_type;
  using symbol_type = yy::Parser::symbol_type;

  static symbol_type yylex(yy::Lexer* p_lexer) {
      return p_lexer->ReadToken();
  }

  int yyFlexLexer::yylex() {
    Logger::Warn("Wrong yylex() called");
    return EXIT_FAILURE;
  }
}
```

这部分代码会被直接复制到`src/syntactic_parser.cpp`的头部。在这里我们提供了函数`yy::Parser::yylex()`的定义，在 Parser 调用 Lexer 读取 token 时需要使用。这里这样写的目的是为了修改 Flex 提供的函数`yylex()`的默认函数签名`int yyFlexLexer::yylex()`，因为 Bison 要求函数`yylex()`的返回值是一个`symbol_type`而不是`int`，但普通的函数重载不能修改返回值的类型。这里仍然提供了一个`int yyFlexLexer::yylex()`的定义，否则编译时会报错。

同时为了修改这个返回值的类型，我们需要在`src/lexer`中定义宏`src/lexer.h`中定义宏`YY_DECL`,使得Flex在生成`yylex()`的代码时，会使用我们所提供的函数签名。

2.1.2Bison declarations

在这里首先我们对Bison进行了一些设置

```
%skeleton "lalr1.cc"       // 使用 LALR1 语法分析器
%require "3.8"             // 指定 Bison 版本为 3.8+（部分特性仅 3.8+ 支持）
```

```
%header "src/parser.hpp"  // 生成 Parser 时将头文件分离，并将 Parser 的头文件保存在这个路径
%locations                // 提供 location 接口，以便 Lexer 记录当前分析到的位置

%define api.parser.class {Parser}          // 指定生成 Parser 的类名为 Parser
%define api.location.file "location.hpp"// 将 location 的头文件保存在这个路径（相对于 parser.hpp 位置）
%define api.token.constructor  // 提供 token 构造函数接口，以便 Lexer 返回 symbol_type 的 token
%define api.token.prefix {T_} // 为 token 名增加指定前缀
%define api.token.raw//因为 Lexer 返回的已经是 symbol_type 的 token 而非读取的原字符,这里不再需要转换
%define api.value.type variant // 指定 token 语义值类型为 variant，用于同时支持更广泛的类型，

%define parse.assert  // 通过运行时检查确保 token 被正常构造与析构，以保证上述 variant 类型被正确使用
%define parse.trace   // 启用 Debug 功能
%define parse.error verbose// 指定报错等级，verbose 级别将对语法分析时遇到的错误进行详细报错
%define parse.lac full  // 启用 lookahead correction (LAC)，提高对语法错误的处理能力

%lex-param {yy::Lexer* p_lexer}           // 函数 yy::Parser::yylex() 传入的参数
%parse-param {yy::Lexer* p_lexer} {Driver* p_driver// 函数 yy::Parser::Parser() 传入的参数
```

然后，我们声明了所有在语法分析时遇到的终结符和非终结符的语义值类型,`%token`为终结符，`%nterm`表示非终结符。

```
%token
//reserved keywords
<std::string>   AND
<std::string>   ARRAY
<std::string>   BEGIN
<std::string>   BY
<std::string>   DIV
<std::string>   DO
<std::string>   ELSE
<std::string>   ELSIF
<std::string>   END
<std::string>   EXIT
<std::string>   FOR
<std::string>   IF
<std::string>   IN
<std::string>   IS
<std::string>   LOOP
<std::string>   MOD
<std::string>   NOT
<std::string>   OF
<std::string>   OR
```

```
<std::string>   OUT
<std::string>   PROCEDURE
<std::string>   PROGRAM
<std::string>   READ
<std::string>   RECORD
<std::string>   RETURN
<std::string>   THEN
<std::string>   TO
<std::string>   TYPE
<std::string>   VAR
<std::string>   WHILE
<std::string>   WRITE

//operators
<std::string>   ASSIGN  ":="
<std::string>   PLUS    "+"
<std::string>   MINUS   "-"
<std::string>   MULT    "*"
<std::string>   DIVIDE  "/"
<std::string>   LT      "<"
<std::string>   LE      "<="
<std::string>   GT      ">"
<std::string>   GE      ">="
<std::string>   EQ      "="
<std::string>   NE      "<>"


//delimiters
<std::string>   LPAREN      "("
<std::string>   RPAREN      ")"
<std::string>   LBRACK      "["
<std::string>   RBRACK      "]"
<std::string>   LBRACE      "{"
<std::string>   RBRACE      "}"
<std::string>   COMMA       ","
<std::string>   COLON       ":"
<std::string>   SEMICOLON   ";"
<std::string>   DOT         "."
<std::string>   LSABRAC     "[<"
<std::string>   RSABRAC     ">]"
<std::string>   BACKSLASH   "\\"

//identifiers
<std::string>   ID          "identifier"

//constants
<std::string>   INTEGER     "integer"
<std::string>   REAL        "real"
<std::string>   STRING      "string"
```

```
;

%left OR;
%left AND;
%nonassoc EQ NE;
%nonassoc LT LE GT GE;
%left PLUS MINUS;
%left MULT DIVIDE MOD DIV;//binary operators
%right POS NEG NOT;//unary operators

%nterm
//program
<shared_ptr<Program> >                  program
<shared_ptr<Body> >                     body

//declarations
<shared_ptr<Declarations> >             declarations
<shared_ptr<Declarations> >             declaration
<shared_ptr<VarDeclarations> >          var_declarations
<shared_ptr<VarDeclaration> >           var_declaration
<shared_ptr<TypeDeclarations> >         type_declarations
<shared_ptr<TypeDeclaration> >          type_declaration
<shared_ptr<ProcedureDeclarations> >    procedure_declarations
<shared_ptr<ProcedureDeclaration> >     procedure_declaration
<shared_ptr<FormalParameters> >         formal_parameters
<shared_ptr<FormalParameter> >          formal_parameter
<shared_ptr<TypeAnnotation> >           type_annotation
<shared_ptr<Type> >                     type
<shared_ptr<Components> >               components
<shared_ptr<Component> >                component
<shared_ptr<Ids> >                      ids
<shared_ptr<Id> >                       id

//statements
<shared_ptr<Statements> >               statements
<shared_ptr<Statement>  >               statement
<shared_ptr<ReadParameters>>            read_parameters
<shared_ptr<ActualParameters> >         actual_parameters
<shared_ptr<WriteParameters> >          write_parameters
<shared_ptr<ElifSections> >             elif_sections
<shared_ptr<ElifSection> >              elif_section
<shared_ptr<ElseSection> >              else_section
<shared_ptr<ForStep> >                  for_step

//expressions
<shared_ptr<Expressions> >              expressions
<shared_ptr<Expression> >               expression
<shared_ptr<WriteExpressions> >         write_expressions
<shared_ptr<WriteExpression> >          write_expression
```

```
<shared_ptr<AssignExpressions> >          assign_expressions
<shared_ptr<AssignExpression> >           assign_expression
<shared_ptr<ArrayExpressions> >           array_expressions
<shared_ptr<ArrayExpression> >            array_expression
<shared_ptr<Number> >                     number
<shared_ptr<Integer> >                    integer
<shared_ptr<Real> >                       real
<shared_ptr<String> >                     string
<shared_ptr<Lvalues> >                    lvalues
<shared_ptr<Lvalue> >                     lvalue
<shared_ptr<ComponentValues> >            component_values
<shared_ptr<ArrayValues> >                array_values
;
```

对所有非终结符，我们都定义了一个AST节点，每个非终结符的类型是这个节点的`std::shared_ptr`,这样做的目的是不需要手动地进行内存管理，方便我们的操作。

同时在这段代码中，我们还指定了操作符的优先级和结合性，从上到下表示优先级依次提高，`%left`，`%right`，`%nonassoc` 分别表示左结合、右结合和非结合。

2.1.3Grammar rules

在这部分我们定义了语法分析的具体规则。即所有非终结符的产生式

```
%start program;

//program
program:
    PROGRAM IS body SEMICOLON{
        $$ = make_shared<Program>(@$,$body);
        p_driver->SetProgram($$);
    }
```

首先是程序的根结点program，program的产生式为：

```
program -> PROGRAM IS body ';'
```

在产生式所对应的规则中，我们使用`std::make_shared`来产生一个指向Program类的`std::shared_ptr`,传入的参数`@$`和`$body`分别为`$`的location和`body`的语义值，其中`$`表示当前非终结符（也就是`program`）。最后我们将这个指针赋值给`$$`，也就是`$`的语义值。

由于`$`是根结点`program`，因此我们将其保存到`Driver`类里。将来我们通过`Driver`打印语法树时，这就是我们语法树的根。

下面是declaration的产生式

```
declaration:
    VAR var_declarations{
        $$ = $2;
```

```
        if($$) $$->SetLocation(@$);
    }
|   PROCEDURE procedure_declarations{
        $$ = $2;
        if($$) $$->SetLocation(@$);
    }
|   TYPE type_declarations{
        $$ = $2;
        if($$) $$->SetLocation(@$);
    }
;
```

这里SetLocation函数的所用是更新当前节点的location，`$1` `$2`分别代表产生式的第一个和第二个语义值

下面是处理多个并列declaration的情况

```
declarations:
    %empty{
        $$ = make_shared<Declarations>(@$);
    }
| declarations declaration{
        $$ =$1;
        if($$) $$->InsertArray($2);
    }
;
```

这里是程序在处理多个并列语句时，我们通过递归的方式来描述这个产生式，递归的退出条件空产生式用`%empty`来进行表示，动作是新建一个Declarations对象，其中包含了一个`std::vector`，之后每分析道一个declaration，就通过InsertArray来将新的declaration的所有子节点移到这个vector中，这里declaration和declarations的语义值都是`std::shared_ptr<Declarations>`

下面是statement的产生式

```
statement:
    lvalue ASSIGN expression SEMICOLON{
        $$ = make_shared<AssignStatement>(@$,$lvalue,$expression);
    }
|   id LPAREN actual_parameters RPAREN SEMICOLON{
        $$ = make_shared<ProcedureCallStatement>(@$,$id,$actual_parameters);
    }
|   IF expression THEN statements elif_sections[elif] else_section[else] END SEMICOLON{
        $$ = make_shared<IfStatement>(@$,$expression,$statements,$elif,$else);
    }
|   READ LPAREN read_parameters RPAREN SEMICOLON{
        $$ = make_shared<ReadStatement>(@$,$read_parameters);
```

```
    }
|   WRITE LPAREN write_parameters RPAREN SEMICOLON{
        $$ = make_shared<WriteStatement>(@$,$write_parameters);
    }
|   WHILE expression DO statements END SEMICOLON{
        $$ = make_shared<WhileStatement>(@$,$expression,$statements);
    }
|   LOOP statements END SEMICOLON{
        $$ = make_shared<LoopStatement>(@$,$statements);
    }
|   FOR id ASSIGN expression[begin] TO expression[end] for_step[step] DO statements END
SEMICOLON{
        $$ = make_shared<ForStatement>(@$,$id,$begin,$end,$step,$statements);
    }
|   RETURN expression SEMICOLON{
        $$ = make_shared<ReturnStatement>(@$,$expression);
    }
|   RETURN SEMICOLON{
        $$ = make_shared<ReturnStatement>(@$);
    }

|   EXIT SEMICOLON{
        $$ = make_shared<ExitStatement>(@$);
    }
;
```

statement在不同的产生式里可以有不同的类型，这里利用了C++的多态性，在实现AST时，我们使用继承实现了多态，在产生式中的各种不同的类如AssignStatement，IfStatement都是Statement类的子类，所以可以将他们的指针直接赋值给基类指针。

下面是expression的产生式：

```
expression:
    lvalue{
        $$ = make_shared<LvalueExpression>(@$,$1);
    }
|   number{
        $$ = make_shared<NumberExpression>(@$,$1);
    }
|   LPAREN expression RPAREN{
        $$ = make_shared<ParenthesisExpression>(@$,$2);
    }
|   PLUS expression %prec POS{
        $$ = make_shared<UnaryExpression>(@$,make_shared<Operator>(@1, $1),$2);
    }
|   MINUS expression %prec NEG{
        $$ = make_shared<UnaryExpression>(@$,make_shared<Operator>(@1,$1),$2);
    }
|   NOT expression{
```

```
        $$ = make_shared<UnaryExpression>(@$,make_shared<Operator>(@1,$1),$2);
    }
|   expression PLUS expression {
        $$ = make_shared<BinaryExpression>(@$,$1,make_shared<Operator>(@2,$2),$3);
    }
|   expression MINUS expression{
        $$ = make_shared<BinaryExpression>(@$,$1,make_shared<Operator>(@2,$2),$3);
    }
|   expression MULT expression{
        $$ = make_shared<BinaryExpression>(@$,$1,make_shared<Operator>(@2,$2),$3);
    }
|   expression DIV expression{
        $$ = make_shared<BinaryExpression>(@$,$1,make_shared<Operator>(@2,$2),$3);
    }
|   expression DIVIDE expression{
        $$ = make_shared<BinaryExpression>(@$,$1,make_shared<Operator>(@2,$2),$3);
    }
|   expression MOD expression{
        $$ = make_shared<BinaryExpression>(@$,$1,make_shared<Operator>(@2,$2),$3);
    }
|   expression AND expression{
        $$ = make_shared<BinaryExpression>(@$,$1,make_shared<Operator>(@2,$2),$3);
    }
|   expression OR expression{
        $$ = make_shared<BinaryExpression>(@$,$1,make_shared<Operator>(@2,$2),$3);
    }
|   expression EQ expression{
        $$ = make_shared<BinaryExpression>(@$,$1,make_shared<Operator>(@2,$2),$3);
    }
|  expression NE expression{
        $$ = make_shared<BinaryExpression>(@$,$1,make_shared<Operator>(@2,$2),$3);
    }
|   expression LT expression{
        $$ = make_shared<BinaryExpression>(@$,$1,make_shared<Operator>(@2,$2),$3);
    }
|   expression LE expression{
        $$ = make_shared<BinaryExpression>(@$,$1,make_shared<Operator>(@2,$2),$3);
    }
|   expression GT expression{
        $$ = make_shared<BinaryExpression>(@$,$1,make_shared<Operator>(@2,$2),$3);
    }
|   expression GE expression{
        $$ = make_shared<BinaryExpression>(@$,$1,make_shared<Operator>(@2,$2),$3);
    }
|   id LPAREN actual_parameters RPAREN{
        $$ = make_shared<ProcedureCallExpression>(@$,$1,$3);
    }
|   id LBRACE component_values RBRACE{
        $$ = make_shared<RecordConstructExpression>(@$,$1,$3);
```

```
    }
|   id LSABRAC array_values RSABRAC{
        $$ = make_shared<ArrayConstructExpression>(@$,$1,$3);
    }
;
```

在这里，我们将单目运算符和双目运算符的产生式分开，为的是利用我们之前定义的运算符的优先级和结合性。同时，`PLUS expr %prec POS`表示对此式采用 `POS` 的优先级，`POS` 的优先级和 `NOT` 同级，通过这种方式我们就实现了对 `PLUS` 作为单目运算符时优先级的重定义。

剩余部分的语法规则与上述的语法规则大同小异不做赘述

2.1.4Epilogue

在这个部分我们定义了一个函数，用于实现语法分析器的报错

```
void yy::Parser::error(const location_type& l, const std::string& m) {
    Logger::Error(m,&l);
    Logger::Error(m,&l,p_driver->ofs());
}
```

当 Lexer 或 Parser 遇到词法错误或语法错误时，就会抛出一个 `std::syntax_error` 异常。程序捕获这个异常后，就会调用函数 `yy::Parser::error()`，然后调用函数 `Logger::Error()`，并传入参数 `l` 和 `m`，分别对应错误发生的位置和错误信息

2.2对Flex的修改

在使用Bison进行语法分析时，可以利用一些Bison的特性，所以我们可以对Flex做一些调整来方便功能的实现。

2.2.1位置

Bison提供了location的接口，我们现在可以使用Bison中的函数获得位置的追踪

```
#define YY_USER_ACTION location.step(); location += YYLeng();

auto& location = drv_.location();


NEWLINE                 \n
<INITIAL>{NEWLINE}              { location.lines(); }
<COMMENT>{NEWLINE}              { location.lines(); skip_COMMENTS(YYText(), location); }
```

`YY_USER_ACTION` 在每次读取一个 token 后执行一次。`loc.step()` 将当前 `location` 的 `begin` 设置为 `end`，`location += YYLeng()` 将当前 `loc`ation 的 `end.column` 增加当前 token 的长度。于是，我们就得到了当前 token 的始末位置。

对于换行的情况，我们在遇到换行符 `\n` 时利用 `loc.lines()` 将当前 `loc` 的 `end.line` 加 1，`end.column` 设置为 1，从而实现了行号的更新。skip_COMMENTS是为了保存注释内容所创立的，主要目的是在unterminated comments报错时能够在错误信息里打印注释内容。

2.2.2Token

Lexer 需要返回一个symbol_type 而不是 int,我们使用Bison生成的函数 `yy::Parser::make_<TOKEN>()`来生成需要的token类型，这个函数传入的参数是token的语义值和位置。

```
<INITIAL>{INTEGER}            { return make_INTEGER(YYText(), location); }
<INITIAL>{REAL}               { return make_REAL(YYText(), location); }
<INITIAL>{STRING}             { return make_STRING(YYText(), location); }
<INITIAL>{IDENTIFIER}         { return make_ID(YYText(), location); }
<INITIAL>{OPERATOR}           { return make_OPERATOR(YYText(), location); }
<INITIAL>{DELIMITER}          { return make_DELIMITER(YYText(), location); }
```

2.3错误检测与报错

Bison 的错误处理是通过捕获 `yy::Parser::syntax_error` 异常来实现的，这里 `yy::Parser::syntax_error` 是基于 `std::runtime_error` 的一个派生类。当分析过程中出现语法错误时，程序就会抛出一个 `yy::Parser::syntax_error` 异常，Bison 捕获后调用 `yy::Parser::error()` 函数进行报错。这个函数已经在 `src/parser.yy` 的 Epilogue 部分提供了定义。

2.3.1词法错误

我们让Flex在检测到词法错误时，同样也抛出一个`yy::Parser::syntax_error` 异常传入的参数是 token 的位置和报错信息。这样我们就利用了Bison语法错误机制来实现词法错误的处理

```
UNTERM_STRING          (\"[^\n"]*)
<INITIAL>{UNTERM_STRING}      { panic_UNTERM_STRING(YYText(), location); }
void panic_UNKNOWN_CHAR(const std::string& s, const location_type& location){
    throw yy::Parser::syntax_error(location,"syntax error, unknown character: " + s);
}

symbol_type make_ID(const std::string& s, const location_type& location) {
  if (s.size() > 255) {
    throw yy::Parser::syntax_error(
        location, "compile error, identifier is too long: " + s);
  }

  auto entry = make_keyword_table.find(s);
    if (entry != make_keyword_table.end()) {
        return entry->second(s, location);
    }
    return yy::Parser::make_ID(s, location);
}

symbol_type make_STRING(const std::string& s, const location_type& location) {
  if (s.size() > 257) {
    throw yy::Parser::syntax_error(
        location, "value error, string literal is too long: " + s);
  }
  if(s.find('\t') != std::string::npos){
```

```
    throw yy::Parser::syntax_error(location,"value error, invalid character found in
string: "+s);
  }
    return yy::Parser::make_STRING(s,location);
}

symbol_type make_INTEGER(const std::string& s, const location_type& location){
    try{
        std::stoi(s);
    }catch(const std::out_of_range& e){
        throw yy::Parser::syntax_error(location,"range error,integer out of range" +
s);
    }
    return yy::Parser::make_INTEGER(s,location);
}

void skip_COMMENTS(const std::string& s, const location_type& location){
    //do nothing
    static std::string comments_buffer;
    static location_type comments_location;

    if(s == "(*"){
        comments_buffer = s;
        comments_location = location;
    }else if(!s.empty()){
        comments_buffer += s;
        comments_location += location;
    }else{
        throw yy::Parser::syntax_error(comments_location,"syntax error, unterminated
comment: " + comments_buffer);
    }
}
```

2.3.2语法错误

当 Bison 遇到语法错误时，会自动抛出 `yy::Parser::syntax_error` 异常，因此不需要显式地写报错逻辑，只需要定义 `yy::Parser::error()` 函数。但是我们需要进行错误恢复，这就让我们需要对每一种非终结符可能出现的错误进行特殊的错误处理，在本项目中只对测试用例中出现的错误进行了处理。

例如，我们对program的错误恢复进行了一些特判：

```
program:
    PROGRAM IS body SEMICOLON{
        $$ = make_shared<Program>(@$,$body);
        p_driver->SetProgram($$);
    }
| error body SEMICOLON{
        $$ = make_shared<Program>(@$,$body);
        p_driver->SetProgram($$);
        yyerrok;
    }
| error {$$ = nullptr;yyerrok;yyclearin;}
;
```

这里，`error body SEMICOLON` 将语法错误视作一个 token（`error`）。通过这样的产生式可以让程序检测到错误后，能继续对后续输入进行语法分析。

动作里的 `yyerrok` 表示立即报错，并继续进行语法分析。继续分析时，程序默认会重新分析这次读入的token，但有时这会导致程序出现死循环，`yyclearin` 的作用就是丢弃这个 token，直接读取下一个 token，从而避免这种情形。

```
var_declaration:
    ids type_annotation ASSIGN expression SEMICOLON{
        $$ = make_shared<VarDeclaration>(@$,$ids,$type_annotation,$expression);
    }
| error SEMICOLON{
    $$ = nullptr;
    yyerrok;
    }
;

type_declaration:
    id IS type SEMICOLON{
        $$ = make_shared<TypeDeclaration>(@$,$id,$type);
    }
| error SEMICOLON{$$ = nullptr;yyerrok;}
;

component:
    id COLON type SEMICOLON{
        $$ = make_shared<Component>(@$,$id,$type);
    }
|   error SEMICOLON { $$ = nullptr; yyerrok;}
;

statement:
    ...
|   error SEMICOLON{
        $$ = nullptr;
```

```
        yyerrok;
    }
```

这里的错误恢复是在一条语句的结尾少一个分号时，舍弃接下来所有的 token，直到读到分号为止。读到分号后，错误恢复，继续进行接下来的语法分析。

```
procedure_declaration:
    id LPAREN formal_parameters RPAREN type_annotation IS body SEMICOLON{
        $$ = make_shared<ProcedureDeclaration>
(@$,$id,$formal_parameters,$type_annotation,$body);
    }
|   id LPAREN formal_parameters RPAREN type_annotation error body SEMICOLON{
        $$ = make_shared<ProcedureDeclaration>
(@$,$id,$formal_parameters,$type_annotation,$body);
        yyerrok;
    }
;
```

这部分是为了应对case14中procedure声明错误

2.4语法树的实现

我们将所有节点分为三种类型：节点 `Node`、有值节点 `ValueNode` 和数组节点 `Nodes`。我们先实现了所有节点的基类 `Node`，提供了节点的基本成员和方法，然后基于 `Node` 派生出 `ValueNode` 和 `Nodes`，分别提供有值节点和数组节点的额外成员和方法。所有其他类型的节点最终都继承自这三种节点。

2.4.1基础节点类

Node的实现如下：

```
class Node{
    public:
        explicit Node(const yy::location& location,const std::string name = "node") :
name_{name},location_{location} {}
        virtual ~Node() {}

        virtual void UpdateDepth(int depth);
        virtual void Print(std::ostream& os) const;

        std::string name() const { return name_; }
        yy::location loc() const { return location_; }
        void SetLocation(const yy::location& location) { location_ = location; }
        void SetDepth(int depth) { depth_ = depth; }

    protected:
        void PrintIndent(std::ostream& os) const;
        void PrintLocation(std::ostream& os) const;
        void PrintBase(std::ostream& os) const;
```

```
    private:
        std::string name_;
        yy::location location_;
        int depth_ = 0;
};
```

```
void Node::Print(std::ostream& os) const {
    PrintBase(os);
    os << std::endl;
}

void Node::UpdateDepth(int depth) {
    SetDepth(depth);
}

void Node::PrintIndent(std::ostream& os) const {
    os << std::string(depth_, '\t');
}

void Node::PrintLocation(std::ostream& os) const {
    os << "(" << location_.begin.line << "," << location_.begin.column << ")-("
       << location_.end.line << "," << location_.end.column << ")";
}

void Node::PrintBase(std::ostream& os) const {
    PrintIndent(os);
    os << name() << " ";
    PrintLocation(os);
}
```

Node的构造函数传入参数location和name，分别代表token所在的位置和Node的名称，方便子类修改自己的名称。UpdateDepth方便每个节点在规约时更新深度，其主要逻辑是传入当前节点的深度depth，更新自己的深度depth_，然后「递归」调用所有子节点（如果有）的UpdateDepth()，传入参数 depth + 1。

在最后打印语法分析树前，我们调用一次根结点的UpdateDepth(),根结点的深度为0，其每个字节点深度是父节点深度+1。

在PrintIndent函数中，每个节点的锁进是一个"\t"，我们通过缩进来表示节点的深度。在打印节点时，我们打印节点的name和location。

ValueNode的实现如下：

```
class ValueNode : public Node{
    public:
        explicit ValueNode(const yy::location& location,const std::string name =
"node",  const std::string& value = "")
            : Node{location,name}, value_{value} {}

        void Print(std::ostream& os) const override;
        virtual std::string value() const { return value_; }

    private:
        const std::string value_;
};
```

```
void ValueNode::Print(std::ostream& os) const {
    PrintBase(os);
    os << " " << value() << std::endl;
}
```

ValueNode由Node继承而来，相比于Node多了一个语义值value_，并对于value增加了一些方法。

Nodes的实现如下：

```
class Nodes : public Node{
    public:
        explicit Nodes(const yy::location& location,const std::string name = "nodes") :
Node{location,name} {}

        void Insert(shared_ptr<Node> p_node);
        void InsertArray(shared_ptr<Nodes> p_nodes);
        void UpdateDepth(int depth) override;
        void Print(std::ostream& os) const override;

    private:
        std::vector<shared_ptr<Node> > data_;
};
```

```
void Nodes::Insert(shared_ptr<Node> p_node) {
    if (p_node) {
        SetLocation(loc() + p_node->loc());
        data_.push_back(p_node);
    }
}

void Nodes::InsertArray(shared_ptr<Nodes> p_nodes) {
    if (p_nodes) {
        for (auto&& p_node : p_nodes->data_) {
```

```
            Insert(p_node);
        }
    }
}

void Nodes::UpdateDepth(int depth) {
    Node::UpdateDepth(depth);
    for (auto&& p_node : data_) {
        if (p_node) p_node->UpdateDepth(depth + 1);
    }
}

void Nodes::Print(std::ostream& os) const {
    if (data_.size()) {
        Node::Print(os);
        for (const auto& p_node : data_) {
            if (p_node) p_node->Print(os);
        }
    }
}
```

Nodes 比Node主要是多了一个数组成员data，包含了所有子节点的指针，并围绕 data新增和修改了相关方法，比如提供了插入节点的方法 Insert() 和插入另一个数组节点的所有子节点的方法 InsertArray()。

2.4.2具体节点类

Program类的实现如下：

```
class Program : public Node {
    public:
        explicit Program(const yy::location& location, shared_ptr<Body> p_body)
            : Node{location,"program"}, p_body_{p_body} {}

        void UpdateDepth(int depth) override;
        void Print(std::ostream& os) const override;

    private:
        shared_ptr<Body> p_body_;
};
```

```
void Program::UpdateDepth(int depth) {
  Node::UpdateDepth(depth);
  if (p_body_) p_body_->UpdateDepth(depth + 1);
}

void Program::Print(std::ostream& os) const {
  Node::Print(os);
  if (p_body_) p_body_->Print(os);
}
```

相比于Node，增加了一个指向Bode的指针，用来存储语法分析完成后的程序，同时覆盖对于Program的相关函数UpdateDepth()和Print()。Print()中，我们向下递归打印语法分析树的所有节点

Expression类的实现如下：

```
class Expression: public ValueNode {
    public:
        explicit Expression(const yy::location& location,const std::string name =
"expression", const std::string& value = "")
            : ValueNode{location,name, value} {}
};

class Expressions: public Nodes {
    public:
        explicit Expressions(const yy::location& location,const std::string name =
"expression list") : Nodes{location,name} {}
};
```

在这里只是修改了节点的名子，其余与Nodes和ValueNode并无区别

UnaryExpression类实现如下：

```
class UnaryExpression : public Expression {
    public:
        explicit UnaryExpression(const yy::location& location, shared_ptr<Operator>
p_operator, shared_ptr<Expression> p_expression)
            : Expression{location,"unary expression"}, p_operator_{p_operator},
p_expression_{p_expression} {}

        void UpdateDepth(int depth) override;
        void Print(std::ostream& os) const override;

        std::string value() const override;

    private:
        shared_ptr<Operator> p_operator_;
        shared_ptr<Expression> p_expression_;
```

```
};
```

```
void UnaryExpression::UpdateDepth(int depth) {
    Expression::UpdateDepth(depth);
    if (p_operator_) p_operator_->UpdateDepth(depth + 1);
    if (p_expression_) p_expression_->UpdateDepth(depth + 1);
}

void UnaryExpression::Print(std::ostream& os) const {
    Expression::Print(os);
    if (p_operator_) p_operator_->Print(os);
    if (p_expression_) p_expression_->Print(os);
}

std::string UnaryExpression::value() const {
    auto op = p_operator_ ? p_operator_->value() : "";
    auto expression = p_expression_? p_expression_->value() : "";
    return op + expression;
}
```

这里主要是对value的操作有一些区别，在单目运算中，我们需要一个operator和一个expression，我们分别得到这两个的语义值并将它们组合，就能得到一个单目运算式的语义值。

WriteExpression类实现如下

```
class WriteExpression : public Expression {
    public:
        using UnionPtr = std::variant<shared_ptr<Expression>, shared_ptr<String>>;

        explicit WriteExpression(const yy::location& location, UnionPtr
p_write_expression)
            : Expression{location,"write expression"},
p_write_expression_{p_write_expression} {}

        void UpdateDepth(int depth) override;
        void Print(std::ostream& os) const override;

        std::string value() const override;

    private:
        UnionPtr p_write_expression_;
};
```

```
void WriteExpression::UpdateDepth(int depth) {
    Expression::UpdateDepth(depth);
    auto visitor = Overloaded{
        [depth](auto&& p) {
            if(p) p->UpdateDepth(depth + 1);
```

```
        },
    };
    std::visit(visitor, p_write_expression_);
}

void WriteExpression::Print(std::ostream& os) const {
    Expression::Print(os);
    auto visitor = Overloaded{
        [&os](auto&& p){
            if (p) p->Print(os);
        },
    };
    std::visit(visitor, p_write_expression_);
}

std::string WriteExpression::value() const {
    auto visitor = Overloaded{
        [](const auto& p) {
            auto value = p ? p->value() : "";
            return value;
        },
    };
    return std::visit(visitor, p_write_expression_);
}
```

在WriteExpression中，我们使用了variant，WriteExpression保存的可能是一个String或者Expression的`std::shared_ptr`,在实际调用时动态判断保存的具体是哪一种类型，并进行相应的操作。下面的visitor实现了对于variant的访问。

其余类的实现方法与上述实现类似，不做赘述。

2.5文件组织结构

- bin/:二进制文件位置（构建时自动生成）
  - syntactic_parser:语法分析器的二进制文件
- build/:项目构建时产生的临时文件（构建时自动生成）
- output/:语法分析结果的存放
- src/:源代码
  - ast/：语法树节点的实现
  - base/:一些通用设置
  - utils/:工具，主要是一个日志生成器
  - driver.h语法分析器
  - driver.cpp
  - lexer.h:词法分析器头文件
  - lexer.cpp：词法分析器源文件（由Flex生成）
  - location.h:用于标识分析到的位置(项目构建时生成)
  - main.cpp:主程序

    - syntactic_parser.yy:语法分析器文件，用于Bison生成语法分析器
    - syntactic_parser.h：语法分析器头文件（Bison生成）
    - syntactic_parser.cpp:语法分析器源文件（Bison生成）
- tests/:测试用例
- Makefile
- test.sh:自动化测试脚本

2.6Makefile实现

首先，我们使用Flex生成词法分析器源文件

```
SRC_DIR   := src

LEX_IN    := $(SRC_DIR)/lexer.lex
LEX_SRC   := $(SRC_DIR)/lexer.cpp
LEX       := flex

$(LEX_SRC): $(LEX_IN)
  @echo + $@
  @$(LEX) -o $@ $<
```

然后，我们通过Bison生成词法分析器文件

```
YACC_IN   := $(SRC_DIR)/parser.yy
YACC_SRC  := $(SRC_DIR)/parser.cpp
YACC_H    := $(SRC_DIR)/parser.hpp $(SRC_DIR)/location.hpp
YACC      := bison

$(YACC_SRC): $(YACC_IN)
  @echo + $@
  @$(YACC) -o $@ -d $<

$(YACC_H): $(YACC_SRC)
```

然后，我们通过g++生成源代码的目标文件，并通过g++生成最终二进制文件

```
TARGET    := syntactic_parser
BIN_DIR   := bin
BUILD_DIR := build

SRCS      := $(YACC_SRC) $(LEX_SRC) $(shell find $(SRC_DIR) -name *.cpp)
OBJS      := $(SRCS:%=$(BUILD_DIR)/%.o)
DEPS      := $(OBJS:.o=.d)
CXX       := g++
CXXFLAGS  := -g -Wall -O3 -std=c++17 -MMD
MKDIR     := mkdir -p
```

```
$(BUILD_DIR)/%.cpp.o: %.cpp
  @echo + $@
  @$(MKDIR) $(dir $@)
  @$(CXX) $(CXXFLAGS) -c -o $@ $<

$(BIN_DIR)/$(TARGET): $(OBJS)
  @echo + $@
  @$(MKDIR) $(dir $@)
  @$(CXX) $(CXXFLAGS) -o $@ $^
# ...

-include $(DEPS)
```

项目运行时，我们执行二进制文件bin/syntactix_parser，变量INPUT是用户输入的测试文件名

```
OUT_DIR   := output

start: $(BIN_DIR)/$(TARGET)
  @$(MKDIR) $(OUT_DIR)
  @$< $(INPUT)
```

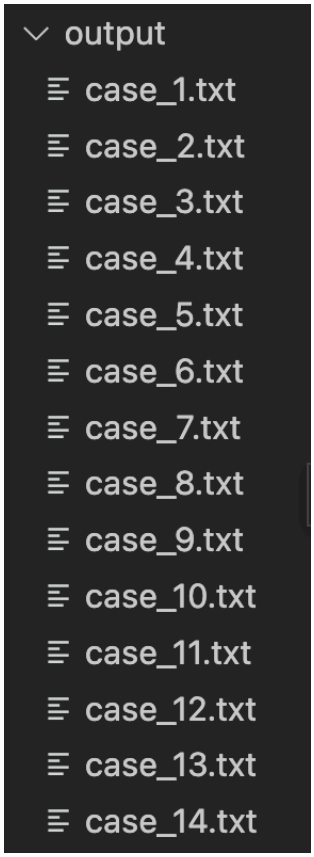
我们还编写了test.sh用于快速测试所有样例，具体文件如下

```
#!/bin/sh

make clean
for filename in tests/case_*.pcat; do
    make INPUT="$filename"
done
```

2.7成果展示

对项目的测试用例进行语法分析，得到的结果保存在output/case_1.txt到output/case_14.txt中



以case_1.txt为例，得到的结果如下：

```
# tests/case_1.pcat


program (1,1)-(8,5)
  body (1,11)-(8,4)
    declaration list (1,11)-(4,25)
      variable declaration (2,9)-(2,29)
        identifier list (2,9)-(2,13)
          identifier (2,9)-(2,10) i
          identifier (2,12)-(2,13) j
        type annotation (2,14)-(2,23)
          id type (2,16)-(2,23)
            identifier (2,16)-(2,23) INTEGER
        number expression (2,27)-(2,28) 1
          integer (2,27)-(2,28) 1
      variable declaration (3,9)-(3,25)
        identifier list (3,9)-(3,10)
          identifier (3,9)-(3,10) x
        type annotation (3,11)-(3,17)
          id type (3,13)-(3,17)
            identifier (3,13)-(3,17) REAL
        number expression (3,21)-(3,24) 2.0
          real (3,21)-(3,24) 2.0
      variable declaration (4,9)-(4,25)
        identifier list (4,9)-(4,10)
          identifier (4,9)-(4,10) y
        type annotation (4,11)-(4,17)
          id type (4,13)-(4,17)
            identifier (4,13)-(4,17) REAL
        number expression (4,21)-(4,24) 3.0
          real (4,21)-(4,24) 3.0
    statement list (5,6)-(7,36)
      write statement (6,5)-(6,36)
        write parameter list (6,12)-(6,34)
          write expression (6,12)-(6,18) "i = "
            string (6,12)-(6,18) "i = "
          write expression (6,20)-(6,21) i
            lvalue expression (6,20)-(6,21) i
              lvalue (6,20)-(6,21) i
          write expression (6,23)-(6,31) ", j = "
            string (6,23)-(6,31) ", j = "
          write expression (6,33)-(6,34) j
            lvalue expression (6,33)-(6,34) j
              lvalue (6,33)-(6,34) j
      write statement (7,5)-(7,36)
        write parameter list (7,12)-(7,34)
          write expression (7,12)-(7,18) "x = "
            string (7,12)-(7,18) "x = "
          write expression (7,20)-(7,21) x
            lvalue expression (7,20)-(7,21) x
              lvalue (7,20)-(7,21) x
          write expression (7,23)-(7,31) ", y = "
            string (7,23)-(7,31) ", y = "
          write expression (7,33)-(7,34) y
            lvalue expression (7,33)-(7,34) y
              lvalue (7,33)-(7,34) y

```

case_11得到的结果如下：

```
# tests/case_11.pcat

[ERROR] (2,1)-(2,9): syntax error, unexpected integer, expecting PROGRAM
[ERROR] (5,1)-(5,21): range error,integer out of range1111111111111111111
[ERROR] (14,1)-(14,10): value error, invalid character found in string: "abcde  g"
[ERROR] (17,1)-(17,12): value error, invalid character found in string: "abcde         g"
[ERROR] (23,1)-(23,259): value error, string literal is too long: "12345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456
[ERROR] (35,1)-(35,257): compile error, identifier is too long: x12345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456
[ERROR] (44,1)-(44,2): syntax error, unknown character: 
[ERROR] (50,1)-(50,5): syntax error, unterminated string literal: "abc
[ERROR] (53,1)-(53,9): syntax error, unterminated string literal: "abc(**)
[ERROR] (56,1)-(56,12): syntax error, unterminated string literal: "abc(*123*)
[ERROR] (74,1)-(74,11): syntax error, unterminated comment: (***123**


```

case_12到case_14得到的结果如下：

```
# tests/case_12.pcat

[ERROR] (7,5)-(7,10): syntax error, unexpected WRITE, expecting ;

program (1,1)-(8,5)
    body (1,11)-(8,4)
        declaration list (1,11)-(4,25)
            variable declaration (2,9)-(2,29)
                identifier list (2,9)-(2,13)
                    identifier (2,9)-(2,10) i
                    identifier (2,12)-(2,13) j
                type annotation (2,14)-(2,23)
                    id type (2,16)-(2,23)
                        identifier (2,16)-(2,23) INTEGER
                number expression (2,27)-(2,28) 1
                    integer (2,27)-(2,28) 1
            variable declaration (3,9)-(3,25)
                identifier list (3,9)-(3,10)
                    identifier (3,9)-(3,10) x
                type annotation (3,11)-(3,17)
                    id type (3,13)-(3,17)
                        identifier (3,13)-(3,17) REAL
                number expression (3,21)-(3,24) 2.0
                    real (3,21)-(3,24) 2.0
            variable declaration (4,9)-(4,25)
                identifier list (4,9)-(4,10)
                    identifier (4,9)-(4,10) y
                type annotation (4,11)-(4,17)
                    id type (4,13)-(4,17)
                        identifier (4,13)-(4,17) REAL
                number expression (4,21)-(4,24) 3.0
                    real (4,21)-(4,24) 3.0

```

```
# tests/case_13.pcat

[ERROR] (3,24)-(3,25): syntax error, unexpected .
[ERROR] (7,5)-(7,10): syntax error, unexpected WRITE, expecting ;

program (1,1)-(8,5)
    body (1,11)-(8,4)
        declaration list (1,11)-(4,25)
            variable declaration (2,9)-(2,29)
                identifier list (2,9)-(2,13)
                    identifier (2,9)-(2,10) i
                    identifier (2,12)-(2,13) j
                type annotation (2,14)-(2,23)
                    id type (2,16)-(2,23)
                        identifier (2,16)-(2,23) INTEGER
                number expression (2,27)-(2,28) 1
                    integer (2,27)-(2,28) 1
            variable declaration (4,9)-(4,25)
                identifier list (4,9)-(4,10)
                    identifier (4,9)-(4,10) y
                type annotation (4,11)-(4,17)
                    id type (4,13)-(4,17)
                        identifier (4,13)-(4,17) REAL
                number expression (4,21)-(4,24) 3.0
                    real (4,21)-(4,24) 3.0

```

```
# tests/case_14.pcat

[ERROR] (3,7)-(3,16): syntax error, unexpected PROCEDURE, expecting IS

program (1,1)-(12,5)
    body (1,11)-(12,4)
        declaration list (1,11)-(8,23)
            procedure declaration (2,15)-(7,11)
                identifier (2,15)-(2,18) FOO
                formal parameter list (2,19)-(2,34)
                    formal parameter (2,19)-(2,25)
                        identifier list (2,19)-(2,20)
                            identifier (2,19)-(2,20) X
                        id type (2,22)-(2,25)
                            identifier (2,22)-(2,25) INT
                    formal parameter (2,27)-(2,34)
                        identifier list (2,27)-(2,28)
                            identifier (2,27)-(2,28) Y
                        id type (2,30)-(2,34)
                            identifier (2,30)-(2,34) REAL
                type annotation (2,35)-(2,40)
                    id type (2,36)-(2,40)
                        identifier (2,36)-(2,40) REAL
                body (3,16)-(7,10)
                    declaration list (3,16)-(3,50)
                        procedure declaration (3,17)-(3,50)
                            identifier (3,17)-(3,20) BAR
                            body (3,25)-(3,49)
                                statement list (3,31)-(3,45)
                                    assign statement (3,32)-(3,45)
                                        lvalue (3,32)-(3,33) Y
                                        binary expression (3,37)-(3,44) X + 1.0
                                            lvalue expression (3,37)-(3,38) X
                                                lvalue (3,37)-(3,38) X
                                            operator (3,39)-(3,40) +
                                            number expression (3,41)-(3,44) 1.0
                                                real (3,41)-(3,44) 1.0
                    statement list (4,12)-(6,11)
                        procedure call statement (5,9)-(5,15)
                            identifier (5,9)-(5,12) BAR
                        return statement (6,2)-(6,11)
                            lvalue expression (6,9)-(6,10) Y
                                lvalue (6,9)-(6,10) Y
            variable declaration (8,9)-(8,23)
                identifier list (8,9)-(8,10)
                    identifier (8,9)-(8,10) C
                type annotation (8,11)-(8,17)
                    id type (8,13)-(8,17)
                        identifier (8,13)-(8,17) REAL
                number expression (8,21)-(8,22) 0
                    integer (8,21)-(8,22) 0
        statement list (9,6)-(11,42)
            assign statement (10,5)-(10,22)
                lvalue (10,5)-(10,6) C
                procedure call expression (10,10)-(10,21) FOO()
                    identifier (10,10)-(10,13) FOO
                    actual parameter list (10,14)-(10,20)
                        number expression (10,14)-(10,15) 3
                            integer (10,14)-(10,15) 3
                        number expression (10,17)-(10,20) 2.0
                            real (10,17)-(10,20) 2.0
            write statement (11,5)-(11,42)
                write parameter list (11,11)-(11,40)
                    write expression (11,11)-(11,17) "C = "
                        string (11,11)-(11,17) "C = "
                    write expression (11,19)-(11,20) C
                        lvalue expression (11,19)-(11,20) C
                            lvalue (11,19)-(11,20) C
                    write expression (11,22)-(11,40) " (should be 4.0)"
                        string (11,22)-(11,40) " (should be 4.0)"
```

三.项目完成情况

1、完成指标

（1）项目完成度 60分

- [x] 分析case1至10中的语法，并打印出语法树

- [x] 分析case11-14中出现的各种词法错误和语法错误，提供相应报错信息并提示错误的位置，同时打印出错误修复后的语法树

（2）项目报告及展示 40分

- [x] 撰写项目报告，并详细展现实现过程

2.成员分工

吴骁 55%：共同设计与讨论，并完成大部分代码

蔡哲飚 45%：共同设计与讨论，并完成大部分报告