1 项目要求

1. 使用 flex & bison 完成对 PCAT 语言的语法树的建立, 并将语法树打印出来
2. 具有语法报错功能并提示错误位置

2 实现原理

这部分的实现主要包含两个部分，词法解析和文法解析。

2.1 词法解析

词法解析主要由 flex 实现，代码见 tokenizer.h、tokenizer.l。

flex 是一个自动化工具，可以按照定义好的规则自动生成一个 C 函数 yylex()，也称为扫描器。这个 C 函数把文本串作为输入，a 按照定义好的规则分析文本串中的字符，找到符合规则的一些字符序列后，就执行在规则中定义好的动作。我们定义了行位置变量 ln 和列位置变量 col，以便记录位置。

对于关键词（AND|ELSIF|LOOP|PROGRAM|VAR|ARRAY|END|MOD|READ|WHILE|BEGIN|EXIT|NOT|RECO
相应的操作是输出当前行列位置，判断为关键词，增加记录的行号/列号，并保存特定的 token 信息来构建语法树。

对于关键词（AND|ELSIF|LOOP|PROGRAM|VAR|ARRAY|END|MOD|READ|WHILE|BEGIN|EXIT|NOT|RECO
相应的操作是输出当前行列位置，判断为关键词，增加记录的行号/列号，并保存特定的 token 信息来构建语法树。

2.2 文法分析

文法分析主要由 bison 实现，代码详见 main.y。

bison 是一个语法分析器生成器，可以把一个上下文无关文法的描述通过 LALR(1) 转化成可以分析该文法的 C 或 C++ 程序。在进行语法分析时，主要是利用其生成的 yyparse 函数，对 FILE* 类型的 ffin 进行语法解析，当发现解析错误时，会调用 yyerror 函数。这部分主要有三个步骤，文法构建、语法树分析、语法查错。

2.2.1 文法构建

PCAT 语言的文法构建参见其说明文件 (第 2 页)。在进行构建时，要进行两个转化：

形如 TOKEN{, TOKEN} 的通过新建一个 TOKEN_block 来表示，其中 TOKEN_block 定义为：

12 Complete Concrete Syntax

```
program         -> PROGRAM IS body ';'
body            -> {declaration} BEGIN {statement} END
declaration     -> VAR {var-decl}
                -> TYPE {type-decl}
                -> PROCEDURE {procedure-decl}
var-decl        -> ID { ',' ID } [ ':' type ] ':=' expression ';'
type-decl       -> ID IS type ';'
procedure-decl  -> ID formal-params [':' type] IS body ';'
type            -> ID
                -> ARRAY OF type
                -> RECORD component {component} END
component       -> ID ':' type ';'
formal-params   -> '(' fp-section {';' fp-section } ')'
                -> '(' ')'
fp-section      -> ID {',' ID} ':' type
statement       -> lvalue ':=' expression ';'
                -> ID actual-params ';'
                -> READ '(' lvalue {',' lvalue} ')' ';'
                -> WRITE write-params ';'
                -> IF expression THEN {statement}
                   {ELSIF expression THEN {statement}}
                   [ELSE {statement}] END ';'
                -> WHILE expression DO {statement} END ';'
                -> LOOP {statement} END ';'
                -> FOR ID ':=' expression TO expression [ BY expression ] DO {statement} END ';'
                -> EXIT ';'
                -> RETURN [expression] ';'
write-params    -> '(' write-expr {',' write-expr } ')'
                -> '(' ')'
write-expr      -> STRING
                -> expression
expression      -> number
                -> l-value
                -> '(' expression ')'
                -> unary-op expression
                -> expression binary-op expression
                -> ID actual-params
                -> ID comp-values
                -> ID array-values
l-value         -> ID
                -> l-value '[' expression ']'
                -> l-value '.' ID
actual-params   -> '(' expression {',' expression} ')'
                -> '(' ')'
comp-values     -> '{' ID ':=' expression { ';' ID ':=' expression} '}'
array-values    -> '[<' array-value { ',' array-value } '>]'
array-value     -> [ expression 'OF' ] expression
number          -> INTEGER | REAL
unary-op        -> '+' | '-' | NOT
binary-op       -> '+' | '-' |  '*' | '/' | DIV | MOD | OR | AND
                -> '>' | '<' | '=' | '>=' | '<=' | '<>'
```

$$
\begin{aligned}
TOKEN\_block &-> TOKEN\_block',' TOKEN \\
&-> TOKEN
\end{aligned}
$$

形如 [TOKEN] 的通过新建一个 TOKEN_opt 来表示，其中 TOKEN_opt 定义为：

$$
\begin{aligned}
TOKEN\_opt &-> TOKEN \\
&->
\end{aligned}
$$

这样，就可以将 PCAT 的文法转成 bison 接受的格式，bison 会根据这个进行自动分析，在匹配到对应的文法时，执行一定的内容，我们在这里加入语法树的构建即可。

2.2.2 语法树构建

根据上述的定义，在程序中，每个语法树的节点也就是每个符号，对应了一个类。例如在 syntax.h 的实现中，class Number 对应 number 类，class UnaryOpExpr 对应 unary-op 类，等等。如果在语法里，符号 a 能产生符号 b，那么在 a 定义的类就包含一个 b 对象。当 bison 产生的语法分析器执行 action（写在 main.y 的语法定义部分）时，语法树就被建立起来。

2.2.3 语法查错

在 syntax.h 的最后，对于 Program 对应的节点，调用 void print(int ident) 把整个语法树打印出来。打印的内容详见每一个 class 中的 void print 函数代码。

3 小组分工

吴韫聪（代码之 phrase 部分 + 报告）、朱天歌（代码之语法树部分 + 报告）、兰石懿（测试）、王禹程（测试）