一、参考编译器介绍

主要阅读了pl0编译器，介绍如下

1. 总体结构

- **词法分析器 (getsym)**: 读取源代码字符，并将其转换为符号和标识符，例如变量名和保留字。
- **语法分析器 (block, statement, expression, condition)**: 检查源代码的语法结构是否正确，并生成中间代码。
- **代码生成器 (gen)**: 将中间代码转换为可执行的机器代码。
- **解释器 (interpret)**: 执行生成的机器代码，并输出结果。

2. 接口设计

```
procedure error(n : integer);
```

主要作用是处理编译过程中遇到的错误。当编译器检测到源代码中的错误时，会调用 `error` 过程，并传递一个错误代码 `n` 作为参数。
`error` 过程的具体行为如下：

1. **打印错误信息**： `error` 过程会打印一个错误信息，格式为 `****` 后面跟着空格、错误位置指示 `^` 和错误代码 `n` 。错误代码 `n` 用于标识具体的错误类型。
2. **增加错误计数**： `error` 过程会将编译器内部的错误计数器 `err` 的值增加 1，用于统计编译过程中发现的错误数量。

```
procedure error( n : integer );
  begin
    writeln( '****', ' ':cc-1, '^', n:2 );
    err := err+1
  end; { error }
```

```
procedure getsym;
```

是词法分析的主过程，主要作用是读取源程序文件中的字符，并将其转换为符号和标识符。
`getsym` 过程是编译器词法分析阶段的核心，负责识别源代码中的单词和符号，并为后续的语法分析和代码生成提供输入。
`getsym` 过程的具体行为如下：

1. **跳过空白字符**：首先， `getsym` 过程会跳过源代码中的空白字符（例如空格、制表符等），直到遇到第一个非空白字符。

2. **识别标识符和保留字**：如果遇到的第一个非空白字符是字母， `getsym` 过程会读取后续的字符，直到遇到非字母或数字字符，从而形成一个标识符。然后， `getsym` 过程会检查这个标识符是否是保留字。如果是，它将符号设置为对应的保留字符号；如果不是，它将符号设置为 `ident` （标识符）。
3. **识别数字**：如果遇到的第一个非空白字符是数字， `getsym` 过程会读取后续的字符，直到遇到非数字字符，从而形成一个数字。然后， `getsym` 过程将符号设置为 `number` （数字）。
4. **识别其他符号**：如果遇到的第一个非空白字符是其他符号（例如 `+` ， `-` ， `*` ， `/` ， `(` ， `)` 等）， `getsym` 过程会读取该符号，并将其设置为对应的符号类型。
5. **更新变量**： `getsym` 过程会将识别出的符号存储在变量 `sym` 中，并将识别出的标识符存储在数组 `id` 中，将识别出的数字存储在变量 `num` 中。
6. **读取下一个字符**：最后， `getsym` 过程会调用 `getch` 过程来读取下一个字符，为下一次调用 `getsym` 做准备。

```
procedure getsym;
  var i,j,k : integer;
  procedure getch;
    begin
      if cc = ll  { get character to end of line }
      then begin { read next line }
             if eof(fin)
             then begin
                    writeln('program incomplete');
                    close(fin);
                    exit;
                  end;
             ll := 0;
             cc := 0;
             write(cx:4,' ');  { print code address }
             while not eoln(fin) do
               begin
                 ll := ll+1;
                 read(fin,ch);
                 write(ch);
                 line[ll] := ch
               end;
             writeln;
             readln(fin);
             ll := ll+1;
             line[ll] := ' ' { process end-line }
           end;
      cc := cc+1;
      ch := line[cc]
    end; { getch }
  begin { procedure getsym;   }
    while ch = ' ' do
      getch;
    if ch in ['a'..'z']
    then begin  { identifier of reserved word }
           k := 0;
           repeat
             if k < al
             then begin
                    k := k+1;
                    a[k] := ch
                  end;
             getch
```

```
      until not( ch in ['a'..'z','0'..'9']);
      if k >= kk        { kk : last identifier length }
      then kk := k
      else repeat
             a[kk] := ' ';
             kk := kk-1
           until kk = k;
      id := a;
      i := 1;
      j := norw;   { binary search reserved word table }
      repeat
        k := (i+j) div 2;
        if id <= word[k]
        then j := k-1;
        if id >= word[k]
        then i := k+1
      until i > j;
      if i-1 > j
      then sym := wsym[k]
      else sym := ident
    end
else if ch in ['0'..'9']
    then begin  { number }
           k := 0;
           num := 0;
           sym := number;
           repeat
             num := 10*num+(ord(ch)-ord('0'));
             k := k+1;
             getch
           until not( ch in ['0'..'9']);
           if k > nmax
           then error(30)
         end
    else if ch = ':'
         then begin
                getch;
                if ch = '='
                then begin
                       sym := becomes;
                       getch
                     end
                else sym := nul
```

```
                end
            else if ch = '<'
                then begin
                    getch;
                    if ch = '='
                    then begin
                        sym := leq;
                        getch
                    end
                    else if ch = '>'
                        then begin
                            sym := neq;
                            getch
                        end
                        else sym := lss
                end
            else if ch = '>'
                then begin
                    getch;
                    if ch = '='
                    then begin
                        sym := geq;
                        getch
                    end
                    else sym := gtr
                end
            else begin
                    sym := ssym[ch];
                    getch
                end
end; { getsym }
```

```
procedure gen( x: fct; y,z : integer );
```

编译器中的代码生成过程，主要作用是生成一条中间代码指令，并将其添加到编译器的代码数组 `code` 中。
每条中间代码指令由三个部分组成：

1. **操作码 (f)**: 表示指令要执行的操作，例如 `lit` (加载常数)，`opr` (执行运算)，`lod` (加载变量)，`sto` (存储变量)，`cal` (调用过程)，`int` (增加栈空间)，`jmp` (无条件跳转)，`jpc` (条件跳转)，`red` (读取)，`wrt` (写入) 等。
2. **层级 (l)**: 表示指令所涉及的变量的层级，用于处理嵌套过程。
3. **地址 (a)**: 表示指令的地址或操作数，具体含义取决于操作码。

4.

`gen` 过程的具体行为如下：

1. **检查代码数组是否溢出**：首先， `gen` 过程检查代码数组 `code` 的索引 `cx` 是否超过了数组的大小 `cxmax` 。如果超过了，说明代码数组溢出， `gen` 过程会打印错误信息并退出程序。
2. **生成中间代码指令**：如果代码数组没有溢出， `gen` 过程会创建一条新的中间代码指令，并将操作码 `x` 、层级 `y` 和地址 `z` 分别赋值给指令的三个部分。
3. **更新代码数组索引**：最后， `gen` 过程将代码数组索引 `cx` 增加 1，为生成下一条中间代码指令做准备。

```
procedure gen( x: fct; y,z : integer );
  begin
    if cx > cxmax
    then begin
           writeln('program too long');
           close(fin);
           exit
         end;
    with code[cx] do
      begin
        f := x;
        l := y;
        a := z
      end;
    cx := cx+1
  end; { gen }
```

`procedure test(s1, s2: symset; n: integer);`

编译器中的一个辅助过程，主要作用是检查当前读取的符号 `sym` 是否属于期望的符号集合 `s1` 。如果 `sym` 不属于 `s1` ，则说明发生了语法错误， `test` 过程会调用 `error` 过程报告错误，并跳过错误符号直到遇到期望的符号集合 `s2` 中的符号。

`test` 过程的具体行为如下：

1. **检查符号集合**： `test` 过程首先检查当前读取的符号 `sym` 是否属于期望的符号集合 `s1` 。
2. **处理错误**：如果 `sym` 不属于 `s1` ，则说明发生了语法错误。 `test` 过程会调用 `error` 过程报告错误，并打印错误代码 `n` 。
3. **跳过错误符号**： `test` 过程会继续读取符号，直到遇到期望的符号集合 `s2` 中的符号。这通常是通过循环调用 `getsym` 过程实现的。

4. **继续编译**：一旦遇到期望的符号， `test` 过程会结束，编译器可以继续进行后续的编译工作。

```
procedure test( s1,s2 :symset; n: integer );
  begin
    if not ( sym in s1 )
    then begin
           error(n);
           s1 := s1+s2;
           while not( sym in s1) do
             getsym
           end
  end; { test }
```

```
procedure interpret;
```

编译器中的解释器过程，主要作用是执行生成的中间代码，并输出程序的运行结果。解释器是编译器的一个重要组成部分，它负责将编译器生成的中间代码转换为机器代码，并在虚拟机上执行。
`interpret` 过程的具体行为如下：

1. **初始化栈空间**：解释器首先初始化栈空间，包括程序计数器 `p` 、基地址寄存器 `b` 和栈顶指针 `t` 。栈空间用于存储程序运行过程中的数据，例如变量值、常量值、临时值等。
2. **执行中间代码**：解释器使用循环结构逐条执行中间代码指令。每条指令由操作码、层级和地址三个部分组成。解释器根据操作码执行相应的操作，例如加载常数、执行运算、加载变量、存储变量、调用过程、增加栈空间、无条件跳转、条件跳转、读取、写入等。
3. **处理运算符**：解释器使用栈来存储运算符的操作数，并使用程序计数器来控制程序的执行流程。解释器支持基本的算术运算符和逻辑运算符，例如加法、减法、乘法、除法、取模、等于、不等于、小于、大于、小于等于、大于等于等。
4. **处理控制流**：解释器支持基本的控制流语句，例如条件跳转和循环跳转。解释器根据条件表达式或循环条件来决定是否跳转到指定的指令地址。
5. **输出结果**：解释器在执行过程中会输出程序的运行结果，例如变量的值、常量的值、临时值的值等。

```
procedure interpret;
  const stacksize = 500;
  var p,b,t: integer; { program-,base-,topstack-register }
      i : instruction;{ instruction register }
      s : array[1..stacksize] of integer; { data store }
  function base( l : integer ): integer;
    var b1 : integer;
    begin { find base l levels down }
      b1 := b;
      while l > 0 do
        begin
          b1 := s[b1];
          l := l-1
        end;
      base := b1
    end; { base }
  begin
    writeln( 'START PL/0' );
    t := 0;
    b := 1;
    p := 0;
    s[1] := 0;
    s[2] := 0;
    s[3] := 0;
    repeat
      i := code[p];
      p := p+1;
      with i do
        case f of
          lit : begin
                  t := t+1;
                  s[t]:= a;
                end;
          opr : case a of { operator }
                  0 : begin { return }
                        t := b-1;
                        p := s[t+3];
                        b := s[t+2];
                      end;
                  1 : s[t] := -s[t];
                  2 : begin
                        t := t-1;
                        s[t] := s[t]+s[t+1]
```

```
            end;
          3 : begin
                t := t-1;
                s[t] := s[t]-s[t+1]
            end;
          4 : begin
                t := t-1;
                s[t] := s[t]*s[t+1]
            end;
          5 : begin
                t := t-1;
                s[t] := s[t]div s[t+1]
            end;
          6 : s[t] := ord(odd(s[t]));
          8 : begin
                t := t-1;
                s[t] := ord(s[t]=s[t+1])
            end;
          9 : begin
                t := t-1;
                s[t] := ord(s[t]<>s[t+1])
            end;
          10: begin
                t := t-1;
                s[t] := ord(s[t]< s[t+1])
            end;
          11: begin
                t := t-1;
                s[t] := ord(s[t] >= s[t+1])
            end;
          12: begin
                t := t-1;
                s[t] := ord(s[t] > s[t+1])
            end;
          13: begin
                t := t-1;
                s[t] := ord(s[t] <= s[t+1])
            end;
        end;
  lod : begin
          t := t+1;
          s[t] := s[base(l)+a]
        end;
```

```
          sto : begin
                  s[base(l)+a] := s[t];  { writeln(s[t]); }
                  t := t-1
                end;
          cal : begin  { generate new block mark }
                  s[t+1] := base(l);
                  s[t+2] := b;
                  s[t+3] := p;
                  b := t+1;
                  p := a;
                end;
          int : t := t+a;
          jmp : p := a;
          jpc : begin
                  if s[t] = 0
                  then p := a;
                  t := t-1;
                end;
          red : begin
                  writeln('??:');
                  readln(s[base(l)+a]);
                end;
          wrt : begin
                  writeln(s[t]);
                  t := t+1
                end
        end { with,case }
    until p = 0;
    writeln('END PL/0');
  end; { interpret }
```

二、编译器总体设计

1. 总体结构

本编译器经历了词法分析、语法分析、语义分析和代码生成等阶段，最终选择llvm作为目标代码，具体文件组织结构如下

2. 接口设计

`Lexer` 词法分析器入口

`CompUnitParser` 语法分析构建语法树入口

`LLVMParser` 代码生成构建目标代码入口

`Token` 词法成分和错误处理入口

`Symbol` 符号入口

`SymbolTable` 符号表入口

`Compiler` 编译器总入口

以上均为各个阶段涉及的基础和顶层接口，具体信息见文件组织和各个阶段

3. 文件组织

目标代码为 `llvm`， 整体架构分为 `frontend` 和 `llvm` 层

`frontend` 下设词法 `lexer` 和语法 `symbol`

`llvm` 下设块结构 `llvmBlock` 和指令结构 `llvmInstruction`

```
D:.
|  Compiler.java
|  config.json
|  tree.txt
|
├─frontend
|  ├─lexer
|  |      Error.java
|  |      Lexer.java
|  |      Token.java
|  |      TokenCode.java
|  |
|  └─symbol
|      |  BType.java
|      |  BTypeParser.java
|      |  CompUnit.java
|      |  CompUnitParser.java
|      |  Ident.java
|      |  IdentParser.java
|      |  Symbol.java
|      |  SymbolCode.java
|      |  SymbolTable.java
|      |
|      ├─block
|      |      Block.java
|      |      BlockItem.java
|      |      BlockItemFactor.java
|      |      BlockItemParser.java
|      |      BlockParser.java
|      |
|      ├─decl
|      |  |  Decl.java
|      |  |  DeclFactor.java
|      |  |  DeclParser.java
|      |  |  StringConst.java
|      |  |  StringConstParser.java
|      |  |
|      |  ├─constDecl
|      |  |  |  ConstDecl.java
|      |  |  |  ConstDeclParser.java
|      |  |  |
|      |  |  └─constDef
|      |  |      |  ConstDef.java
```

```
|       |   |       |  ConstDefParser.java
|       |   |       |
|       |   |       └─constInitVal
|       |   |               ConstInitVal.java
|       |   |               ConstInitValArray.java
|       |   |               ConstInitValArrayParser.java
|       |   |               ConstInitValFactor.java
|       |   |               ConstInitValParser.java
|       |   |
|       |   └─varDecl
|       |       |  VarDecl.java
|       |       |  VarDeclParser.java
|       |       |
|       |       └─varDef
|       |           |  VarDef.java
|       |           |  VarDefFactor.java
|       |           |  VarDefParser.java
|       |           |  VarDefWithAssign.java
|       |           |  VarDefWithNull.java
|       |           |
|       |           └─varInitVal
|       |                   VarInitVal.java
|       |                   VarInitValArray.java
|       |                   VarInitValArrayParser.java
|       |                   VarInitValFactor.java
|       |                   VarInitValParser.java
|       |
|       ├─exp
|       |   |  ConstExp.java
|       |   |  ConstExpParser.java
|       |   |  Exp.java
|       |   |  ExpParser.java
|       |   |
|       |   └─expFactor
|       |       |  AddExp.java
|       |       |  AddExpParser.java
|       |       |  EqExp.java
|       |       |  EqExpParser.java
|       |       |  LAndExp.java
|       |       |  LAndExpParser.java
|       |       |  LOrExp.java
|       |       |  LOrExpParser.java
|       |       |  MulExp.java
```

```
|       |       |  MulExpParser.java
|       |       |  RelExp.java
|       |       |  RelExpParser.java
|       |       |
|       |       └─unaryExp
|       |           |  UnaryExp.java
|       |           |  UnaryExpParser.java
|       |           |
|       |           └─unaryExpFactor
|       |               |  UnaryExpFactor.java
|       |               |  UnaryExpFuncR.java
|       |               |  UnaryExpFuncRParser.java
|       |               |  UnaryExpOp.java
|       |               |  UnaryExpOpParser.java
|       |               |  UnaryOp.java
|       |               |  UnaryOpParser.java
|       |               |
|       |               └─primaryExp
|       |                   |  PrimaryExp.java
|       |                   |  PrimaryExpParser.java
|       |                   |
|       |                   └─primaryExpFactor
|       |                           Character.java
|       |                           CharacterParser.java
|       |                           LVal.java
|       |                           LValParser.java
|       |                           Number.java
|       |                           NumberParser.java
|       |                           PrimaryExpExp.java
|       |                           PrimaryExpExpParser.java
|       |                           PrimaryExpFactor.java
|       |
|       ├─func
|       |   |  FuncDef.java
|       |   |  FuncDefParser.java
|       |   |  MainFuncDef.java
|       |   |  MainFuncDefParser.java
|       |   |
|       |   ├─funcParam
|       |   |      FuncFParam.java
|       |   |      FuncFParamParser.java
|       |   |      FuncFParams.java
|       |   |      FuncFParamsParser.java
```

```
|       |   |       FuncRParams.java
|       |   |       FuncRParamsParser.java
|       |   |
|       |   └─funcType
|       |           FuncType.java
|       |           FuncTypeChar.java
|       |           FuncTypeFactor.java
|       |           FuncTypeInt.java
|       |           FuncTypeParser.java
|       |           FuncTypeVoid.java
|       |
|       └─stmt
|           |   Cond.java
|           |   CondParser.java
|           |   Stmt.java
|           |   StmtParser.java
|           |
|           └─stmtFactor
|               |   StmtFactor.java
|               |
|               ├─stmtBlock
|               |       StmtBlock.java
|               |
|               ├─stmtBreak
|               |       StmtBreak.java
|               |       StmtBreakParser.java
|               |
|               ├─stmtContinue
|               |       StmtContinue.java
|               |       StmtContinueParser.java
|               |
|               ├─stmtExp
|               |       StmtExp.java
|               |       StmtExpParser.java
|               |
|               ├─stmtFor
|               |       ForStmt.java
|               |       ForStmtParser.java
|               |       StmtFor.java
|               |       StmtForParser.java
|               |
|               ├─stmtIf
|               |       StmtIf.java
```

```
│               │      StmtIfParser.java
│               │
│               ├─stmtLVal
│               │      StmtLValAssignExp.java
│               │      StmtLValAssignExpParser.java
│               │      StmtLValAssignGetChar.java
│               │      StmtLValAssignGetCharParser.java
│               │      StmtLValAssignGetInt.java
│               │      StmtLValAssignGetIntParser.java
│               │
│               ├─stmtPrintf
│               │      StmtPrintf.java
│               │      StmtPrintfParser.java
│               │
│               └─stmtReturn
│                       StmtReturn.java
│                       StmtReturnParser.java
│
└─llvm
    │  Counter.java
    │  LLVMFunc.java
    │  LLVMGlobal.java
    │  LLVMModule.java
    │  LLVMParser.java
    │
    ├─llvmBlock
    │      BlockBlock.java
    │      ForBlock.java
    │      IfBlock.java
    │      InstructionsBlock.java
    │      LLVMBlock.java
    │
    └─llvmInstruction
            BreakInstruction.java
            ContinueInstruction.java
            DeclInstruction.java
            GetFuncInstruction.java
            LLVMInstruction.java
            LValExpInstruction.java
            PrintfInstruction.java
            ReturnInstruction.java
```

三、词法分析

词法分析的解析入口是 Lexer 类，作为词法分析的解析器， Lexer 接受文法语句输入流 inputStream ，分析词法的同时完成相应的错误处理

最终将文法语句解析成一个个符合词法要求的词法成分 Token ， 通过调用 getTokens() 方法，得到词法列表

```
public ArrayList<Token> getTokens() {
    return (ArrayList<Token>) tokens.clone();
}
```

Lexer 以行为单位对源程序语句进行分析，首先处理输入流中的注释

对于单行注释，跳过之后的内容，解析下一行

```
// 单行注释
Pattern patternLineComment = Pattern.compile("^//.*");
Matcher matcherLineComment = patternLineComment.matcher(line);
if (matcherLineComment.find()) {
    break;
}
```

对于多行注释，采用栈式匹配，匹配成功则继续解析，否则跳过当前行

```
// 多行注释开始
Pattern patternBlockCommentStart = Pattern.compile("^/\\*.*");
Matcher matcherBlockCommentStart = patternBlockCommentStart.matcher(line);
Pattern patternBlockCommentEnd1 = Pattern.compile(".*\\*/");
Matcher matcherBlockCommentEnd1 = patternBlockCommentEnd1.matcher(line);
if (matcherBlockCommentStart.find()) {
    blockComment = true;
    if (matcherBlockCommentEnd1.find()) {
        blockComment = false;
        start += matcherBlockCommentEnd1.end();
        line = rawLine.substring(start);
    } else {
        break;
    }
}
```

遍历的过程中，进行词法分析

```
// 词法分析
for (TokenCode tokenCode : TokenCode.values()) {
     Pattern pattern = tokenCode.getPattern();
     Matcher matcher = pattern.matcher(line);
     if (matcher.find()) {
         tokens.add(new Token(matcher.group(0), lineNumber, tokenCode));
         start += matcher.end();
         line = rawLine.substring(start);
         error = false;
         break;
     }
}
```

识别成功则词法归档，跳转识别下一句
否则进行词法阶段的错误处理

```
// 否则进行错误输出
if (error) {
    for (Error error1: Error.values()) {
        Pattern pattern = error1.getPattern();
        Matcher matcher = pattern.matcher(line);

        if (matcher.find()) {
            tokens.add(new Token(matcher.group(0), lineNumber));
            start += matcher.end();
            break;
        }
    }
}
```

对于词法的识别，采用了正则匹配的方式，所有需要解析的此法成分封装成枚举类 `TokenCode` ，对于每个词法，给出其正则匹配规则

```
public enum TokenCode {
    MAINTK("^main(?![_A-Za-z0-9])"),
    CONSTTK("^const(?![_A-Za-z0-9])"),
    INTTK("^int(?![_A-Za-z0-9])"),
    CHARTK("^char(?![_A-Za-z0-9])"),
    BREAKTK("^break(?![_A-Za-z0-9])"),
    CONTINUETK("^continue(?![_A-Za-z0-9])"),
    IFTK("^if(?![_A-Za-z0-9])"),
    ELSETK("^else(?![_A-Za-z0-9])"),
    FORTK("^for(?![_A-Za-z0-9])"),
    GETINTTK("^getint(?![_A-Za-z0-9])"),
    GETCHARTK("^getchar(?![_A-Za-z0-9])"),
    PRINTFTK("^printf(?![_A-Za-z0-9])"),
    RETURNTK("^return(?![_A-Za-z0-9])"),
    VOIDTK("^void(?![_A-Za-z0-9])"),
    IDENFR("^[_A-Za-z][_A-Za-z0-9]*"),
    INTCON("^[0-9]+"),
    STRCON("^\"([^\"\\\\]*(\\\\.[^\"\\\\]*)*)\""),
    CHRCON("^'([^'\\\\]*(\\\\.[^'\\\\]*)*)'"),
    LEQ("^<="),
    LSS("^<"),
    GEQ("^>="),
    GRE("^>"),
    EQL("^=="),
    NEQ("^!="),
    PLUS("^\\+"),
    MINU("^-"),
    MULT("^\\*"),
    DIV("^/"),
    NOT("^!"),
    AND("^&&"),
    OR("^\\|\\|"),
    MOD("^%"),
    ASSIGN("^="),
    SEMICN("^;"),
    COMMA("^,"),
    LPARENT("^\\("),
    RPARENT("^\\)"),
    LBRACK("^\\["),
    RBRACK("^]"),
    LBRACE("^\\{"),
    RBRACE("^}");
```

```
    private final Pattern pattern;

    TokenCode(String pattern) {
        this.pattern = Pattern.compile(pattern);
    }
}
```

^ 加在所有正则之前，匹配字符串的开始位置
(?![_A-Za-z0-9]) 是一个负向先行断言，确保后面跟的字符不是下划线、字母或数字。也就是说，在关键字后面不能有这些字符。
此外需要注意遍历的顺序问题，比如 EQL("^==") 必须放在 ASSIGN("^=") 前面，因为后者涵盖了前者的情况。

错误词法放在 Error 类中，目前只有 a 类错误

```
public enum Error {
    a("^&|\\|");

    private final Pattern pattern;
    Error(String pattern) {
        this.pattern = Pattern.compile(pattern);
    }
}
```

词法和错误都会以 Token 的形式构建， Token 类含有以下属性

```
public static ArrayList<Token> tokens;
private String name;
private int line;
private TokenCode tokenCode;
private static ArrayList<Token> errors;
```

词法会加入 tokens ， 错误则会进入 errors

四、语法分析

语法分析采用递归下降的方式，逐个解析，每一个 parser 类接受词法分析输出的 tokens ， 解析相应的语法成分。

递归下降的方式和语法类之间的关系由文法决定
比如文法 CompUnit → {Decl} {FuncDef} MainFuncDef 决定了编译单元解析类 CompUnitParser 的解析逻辑

```
public CompUnit parseCompUnit() {
    ArrayList<Decl> decls = parseDecls();
    ArrayList<FuncDef> funcDefs = parseFuncDefs();
    MainFuncDef mainFuncDef = parseMainFuncDef();
    return new CompUnit(decls, funcDefs, mainFuncDef);
}
```

而文法 `Decl → ConstDecl | VarDecl` 决定了声明解析类 `DelcParser` 的解析逻辑

```
public Decl parseDecl() {
    if (tokens.get(0).getCode().equals(TokenCode.CONSTTK)) {
        return new Decl(new ConstDeclParser(tokens).parseConstDecl());
    }
    return new Decl(new VarDeclParser(tokens).parseVarDecl());
}
```

对于终结符，无需再向下解析，直接返回语法成分即可，例如 `ident`

```
public Ident parseIdent() {
    Token token = tokens.remove(0);
    return new Ident(token);
}
```

通过递归下降，每一级 `parser` 将词法列表 `tokens` 传递给下一级，逐级解析相应的语法成分，最终得到一个符合文法架构的语法树
语法树通过语法成分表达文法关系，如对于常量声明文法 `ConstDecl → 'const' BType ConstDef { ',' ConstDef } ';'` ，在 `ConstDecl` 类中会有以下属性来表示。

```
private final String name = "<ConstDecl>";
private Token constTk; // 'const'
private BType btype; // BType
private ConstDef constDef; // ConstDef
private ArrayList<Token> commas;
private ArrayList<ConstDef> constDefs; // { ',' ConstDef }
private Token semicn; // ';'
```

通过重载 `Token` 类的构造方法，实现不同阶段错误处理的区分，最后用方法 `toError()` 统一处理

```
//词法分析 a
public Token(String name, int line) {
    this.error = name;
    if (name.equals("&")) {
        this.name = "&&";
        this.tokenCode = TokenCode.AND;
        this.error = "a";
    } else if (name.equals("|")){
        this.name = "||";
        this.tokenCode = TokenCode.OR;
        this.error = "a";
    }
    this.line = line;
    toError();
}
// 语法分析 i j k
public Token(Token pretoken, TokenCode tokenCode) {
    this.line = pretoken.line;;
    if (tokenCode.equals(TokenCode.SEMICN)) {
        this.error = "i";
    } else if (tokenCode.equals(TokenCode.RPARENT)) {
        this.error = "j";
    } else {
        this.error = "k";
    }
    this.toError();
}
```

五、语义分析

语义分析和语法分析结构大致相同，区别在于

1. 语法分析建立了语法树结构，语义分析需要建立符号表系统
2. 语法分析错误处理很简单，语义分析的错误处理较多而且涉及符号表的上下文关联

在语义分析中，首先建立了符号类 `Symbol` 和符号表类 `SybmbolTable`
为了兼容不同符号的详细信息，符号类中设计了很多属性，对于共性属性，会在构造符号时初始化，对于各种符号的详细属性，则会初始化为 `null` ，在不同构造方法中根据情况初始化。

```
private int line;
public String name;
private SymbolCode symbolCode;
private int array; // 数组维数
private ConstInitVal constInitVal = null; // 常量初值
private VarInitVal varInitVal = null; // 变量初值
private ArrayList<Symbol> funcFParams = null; // 函数参数
```

例如 `constInitVal` 属性，只会在 `const` 类的符号初始化时才会被赋值，其余情况均为 `null`

```
// Const
public Symbol(Integer line, String name, SymbolCode symbolCode,
              Integer array, ConstInitVal constInitVal) {
    this.line = line;
    this.name = name;
    this.symbolCode = symbolCode;
    this.array = array;
    this.constInitVal = constInitVal;
}
```

对于符号表，每张符号表会有一个父表 `parent` ，用来表示符号表之间的从属关系

```
private SymbolTable parent;

public SymbolTable(SymbolTable parent) {
    this.symbols = new HashMap<>();
    this.symbolArrayList = new ArrayList<>();
    this.parent = parent;
    if (parent != null) {
        this.inCycle = parent.isInCycle();
        this.field = parent.getField();
    }
}
```

对符号表各个属性进行说明：

1. `parent` ，父表索引，标识谁创建了此表，最上层的符号表父表为 `null`
2. `symbols` 和 `symbolArrayList` 都是用来存储符号的，map方便索引，而list方便格式化输出
3. `inClcle` 用来标识该符号表是否处于循环当中
4. `field` 用来标识改符号表的区号

此外，需要在递归下降的时候建立符号表系统，主要涉及 `symbolTable` 属性的传递和 `addSymbol` 方法的调用

上级语法或传递自己的符号表，或传递此基础上的子表，作为下级语法的符号表，例如

```
private SymbolTable symbolTable;
...
constExp = new ConstExpParser(tokens, symbolTable).parseConstExp();
```

同时，相比语法分析，需要在返回语法成分之前，将当前符号入表，从而逐级构建符号表系统

```
addSymbol(ident, lBrack, constExp, rBrack, assign, constInitval);
return new ConstDef(ident, lBrack, constExp, rBrack, assign, constInitval, symbolTable);
```

对于错误处理，一般是在符号入表的过程中进行处理，例如 `b` 类错误

```
// bError
if (this.symbolTable.bError(ident.getToken().getName())) {
    new Token("b", ident.getToken().getLine());
} else {
    this.symbolTable.addSymbol(symbol);
}
```

错误处理的过程会调用符号表中相应的方法查看上下文信息，如果发生错误，构建新的错误 `Token` ，否则当前符号入表

需要确保错误处理不会影响编译器正常解析的运行，类似于产生日志，但不能中断异常

六、代码生成

最终代码采用 `llvm`
经过词法分析、语法分析和语义分析，我们已经建立了语法树和符号表，接下来需要再次遍历语法树结构，逐级解析相关的代码成分

代码单元解析类 `LLVMParser` 接受编译单元 `CompUnit` 作为输入，同时也是整个语法树 的起点
每一级语法成分根据自己的文法进行解析，例如文法 `CompUnit → {Decl} {FuncDef} MainFuncDef`

```
public LLVMModule parseLLVMModule() {
    LLVMModule llvmModule = new LLVMModule();
    // 全局变量
    for (Decl decl : this.compUnit.getDecls()) {
       llvmModule.addGlobals(decl.parseLLVMGlobal());
    }
    // 函数
    for (FuncDef funcDef : this.compUnit.getFuncDefs()) {
        llvmModule.addFunc(funcDef.parseLLVMFunc());
    }
    // Main
    llvmModule.addFunc(this.compUnit.getMainFuncDef().parseLLVMFunc());
    return llvmModule;
}
```

类似语法分析的工作，会在相应的语法成分中增加解析 `llvm` 模块的函数

对于全局变量，会解析相应的全局变量 `LLVMGlobal` ，例如 `ConstDecl`

```
public ArrayList<LLVMGlobal> parseLLVMGlobal() {
    ArrayList<LLVMGlobal> globals = new ArrayList<>();
    globals.add(this.constDef.parseLLVMGlobal(this.btype));
    for (int i = 0; i < this.constDefs.size(); i++) {
        globals.add(this.constDefs.get(i).parseLLVMGlobal(this.btype));
    }
    return globals;
}
```

之后在每一个 `ConstDef` 中继续解析全局变量，最终返回具体的全局变量类

```
public LLVMGlobal parseLLVMGlobal(BType bType) {
    Symbol symbol = symbolTable.getSymbol(ident.getToken().getName());
    return new LLVMGlobal(symbolTable, symbol, constExp, constInitval);
}
```

`LLVMGlobal` 以每一条定义为单位，每一个 `def` 会产生一个 `global` ，最终按照顺序依次输出 `llvm` 指令串

对于函数，会先解析成一个函数模块 `LLVMFunc` ，例如 `FuncDef`

```
public LLVMFunc parseLLVMFunc() {
    Symbol symbol = symbolTable.getSymbol(ident.getToken().getName());
    BlockBlock blockBlock = this.block.parseLLVMBlock();
    return new LLVMFunc(symbolTable, symbol, funcFParams, blockBlock);
}
```

每一个函数模块 LLVMFunc 都会含有一个块型模块 LLVMBlock ， 在这里是 BlockBlcok ， 解析来源是语法成分中的 block

下面解释指令模块 LLVMInstruction 和块型模块 LLVMBlock

LLVMInstruction 是指令的统一接口，共有7个指令实现类

1. DeclInstruction 是声明型指令，涉及局部的常量和变量声明，下分到定义，每一个 def 产生一条定义指令
2. BreakInstruction 和 ContinueInstruction 分别是 break 和 continue 语句产生的指令，输出时会接受目标块的 label ， 生成相应的跳转指令
3. GetFuncInstruction 是 getint 和 getchar 语句产生的指令，输出时会生成对应的函数调用指令
4. LValExpInstruction 指令涵盖了 exp 和 lval=exp 两种语句，根据情况进行相应的运算和赋值
5. PrintfInstruction 指令对应 printf 语句，会格式化输出相应的字符串，同时处理转义字符
6. ReturnInstruction 指令对应 return 语句，会根据函数类型生成相应的返回指令

LLVMBlock 是块的统一接口，共有4个实现类

1. BlockBlock 针对 block 语法，其中存放若干个 LLVMBlock ， 输出是按照顺序格式化输出每一个块
2. IfBlock 针对 if 语法，其中存放条件判断 cond 、为真跳转块 ifBlock 以及为假跳转块 elseBlock ， 用于格式化输出 if 相应的上下文指令
3. ForBlock 针对 for 语法，其中包含一个循环体块 forBlock ， 格式化输出 for 有关的上下文指令
4. InstructionsBlock 针对指令型基本块，类似于块的终结符，由若干条 LLVMInstruction 组成，按顺序输出每条指令串

当解析函数的 block 时，会根据语法成分的不同，产生不同类型的模块，最后由这些模块共同组成函数的``BlockBlcok`

当遇到 if 、 for 和 block 类型时，会产生相应的模块类型

```
if (stmtFactor instanceof StmtIf) {
    blocks.add(((StmtIf) stmtFactor).parseLLVMBlock());
    continue;
}
if (stmtFactor instanceof StmtFor) {
    blocks.add(((StmtFor) stmtFactor).parseLLVMBlock());
    continue;
}
if (stmtFactor instanceof StmtBlock) {
    blocks.add(((StmtBlock) stmtFactor).parseLLVMBlock());
    continue;
}
```

除此之外，则需要产生 `InstructionsBlock` 类型
从此位置开始遍历

```
// 解析指令
ArrayList<LLVMInstruction> instructions = new ArrayList<>();
for (int j = i; j < this.blockItems.size(); j++) {
    BlockItemFactor blockItemFactorIn = this.blockItems.get(j).getBlockItemFactor();
```

当发现以上三种成分时，结束遍历

```
    if (blockItemFactorIn instanceof Stmt) {
        StmtFactor stmtFactorIn = ((Stmt) blockItemFactorIn).getStmtFactor();
        if (stmtFactorIn instanceof StmtIf || stmtFactorIn instanceof StmtFor ||
            stmtFactorIn instanceof StmtBlock) {
            break;
        }
```

否则逐条解析指令并添加到指令集中

```
        if (stmtFactorIn instanceof StmtReturn) {
            instructions.add(((StmtReturn) stmtFactorIn).parseLLVMInstructions());
        } else if (stmtFactorIn instanceof StmtPrintf) {
            instructions.add(((StmtPrintf) stmtFactorIn).parseLLVMInstructions());
        } else if (stmtFactorIn instanceof StmtLValAssignGetInt) {
            instructions.add(((StmtLValAssignGetInt) stmtFactorIn).parseLLVMInstruction());
        } else if (stmtFactorIn instanceof StmtLValAssignGetChar) {
            instructions.add(((StmtLValAssignGetChar) stmtFactorIn).parseLLVMInstruction());
        } else if (stmtFactorIn instanceof StmtLValAssignExp) {
            instructions.add(((StmtLValAssignExp) stmtFactorIn).parseLLVMInstruction());
        } else if (stmtFactorIn instanceof StmtExp) {
            instructions.add(((StmtExp) stmtFactorIn).parseLLVMInstruction());
        } else if (stmtFactorIn instanceof StmtBreak) {
            instructions.add(((StmtBreak) stmtFactorIn).parseLLVMInstruction());
        } else if (stmtFactorIn instanceof StmtContinue) {
            instructions.add(((StmtContinue) stmtFactorIn).parseLLVMInstruction());
        }
        } else {
            // decl
            DeclFactor declFactor = ((Decl) blockItemFactorIn).getDeclFactor();
            instructions.addAll(declFactor.parseLLVMInstructions());
        }
            i = j;
        }
```

遍历到末尾时，结束遍历，由指令集构建 `InstructionsBlock`

```
        blocks.add(new InstructionsBlock(instructions));
```

由此便可以建立出 `llvm` 的块-指令结构，之后只需将解析出的 `llvmModule` 中的成分按顺序格式化输出即可

```
for (LLVMGlobal llvmGlobal : llvmGlobals) {
    strings.addAll(llvmGlobal.getString());
}
strings.add("\n");
for (LLVMFunc llvmFunc : llvmFuncs) {
    strings.addAll(llvmFunc.getString(cnt));
}
```

另外，对于标识符的标号，编译器提供全局的计数器 `Counter` 来实现

```
private int cnt;
public Counter() {
    this.cnt = 0;
}
public int use() {
    return cnt++;
}
```