电子科技大学

计算机专业类课程

实验报告

课程名称：编译原理

学　　院：计算机科学与工程

专　　业：计算机科学与技术

学生姓名：蔡与望

学　　号：2020010801024

指导教师：陈昆

日　　期：　　2023 年 5 月 8 日

电 子 科 技 大 学

实　验　报　告

实验一

一、实验名称：词法分析

二、实验学时：4

三、实验内容和目的：

1. 理解词法分析的原理和编程实现方式。
2. 熟悉状态转换图。
3. 对求 n!的极小语言进行词法分析，输出二元式文件（*.dyd）和错误信息文件（*.err）。

四、实验原理：

4.1 词法分析

编译器将源程序看成一个很长的字符串，首先对它进行从左到右的扫描，并进行分析，识别出符合词法规则的单词，如基本字、标识符、常数、运算符和界符等。这些符号以固定的编码提供给后续的语法分析进一步处理。

如果在词法分析过程中发现不符合词法规则的非法单词符号，则做出词法出错处理，给出相应的出错信息。

4.2 源程序

源程序是一个求 n!的极小语言。一个测试程序如下：

```
begin
   integer k;
   integer function F(n);
      begin
         integer n;
         if n<=0 then F:=1
         else F:=n*F(n-1)
      end;
   read(m);
```

```
    k:=F(m);
    write(k)
end
```

4.3 二元式文件

二元式文件包含了源程序的所有识别到的单词的列表。

每个二元式的形式为“符号 种别”，例如“foo 10”。符号是这个单词的内容，例如标识符单词的符号就是标识符的名字，常数单词的符号就是数字；种别是这个单词的内部编码，见下表。

符号	种别	符号	种别	符号	种别
begin	1	标识符	10	*	19
end	2	常数	11	:=	20
integer	3	=	12	(	21
if	4	<>	13	)	22
then	5	<=	14	;	23
else	6	<	15	EOL	24
function	7	>=	16	EOF	25
read	8	>	17		
write	9	-	18		

引入种别对照表，是为了便于后续语法分析时 match 函数的实现。在实际编程实现时，我们还可以为每个符号编上助记符，也就是使用宏定义或枚举给每个数字添加别名，以提高程序的可读性。

除了程序实际使用的 23 种符号，编码表的最后还额外添加了行结束符（EOL）和文件结束符（EOL）。行结束符的引入，是为了让词法和语法分析器能够知道自己正处于哪一行，从而在报错时能够精确到行号。文件结束符则是让后续的分析器知道自己的前驱分析器是否正常结束工作，在哪里结束工作。

4.4 错误信息文件

错误信息文件包含了源程序的所有词法错误。

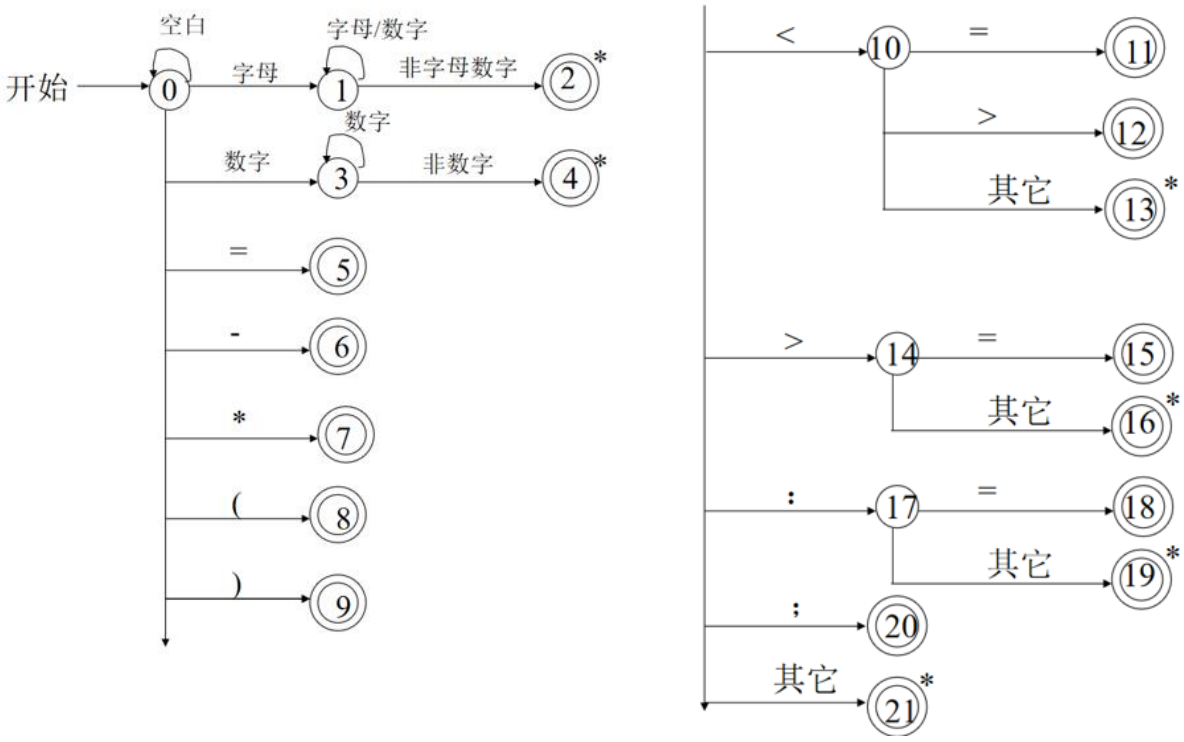
每个错误信息的形式为“LINE:行号 错误性质”。在词法分析中，共有 3 种错误类型：

1. 非法字符，即未在编码表中出现的首字符，无法识别的字符，例如“@”。
2. 冒号不匹配，即不以“:=”形式出现的单独的冒号。
3. 标识符长度溢出，即标识符的名字超过了最大长度，如 16 个字符。

4.5 状态转换图

状态转换图是词法分析的核心。在词法分析器从左到右扫描的过程中，它根据接收到的字符，沿箭头的指示在各个状态圆圈之间转换。如果到达了终止状态，就说明成功识别到了一个单词。

该语言的状态转换图如下：



状态转换图的每个状态都对应一小段程序。分支状态（如 10）对应 if 语句，循环状态（如 1）对应 while 语句，终态（如 2）对应 return 语句。星号（*）表示回退一个字符。

五、实验器材（设备、元器件）

笔记本电脑。

六、实验步骤：

1. 读入源程序。
2. 根据状态转换图，从左到右扫描源程序字符串，识别出所有单词。
3. 将识别到的单词写入二元式文件。
4. 将发现的错误写入错误信息文件。
5. 检查二元式文件内容是否正确。
6. 检查错误信息文件是否包含所有的错误。

七、实验数据及结果分析：

7.1 读入源程序

```
private static readSource() {
  return readFileSync("input/source.pas").toString().trim();
}
```

7.2 识别单词

```
private getNextToken(): Token {
  while (this.cursor.current === " ") {
```

```
    this.cursor.consume();
  }

  const initial = this.cursor.consume();

  if (Lexer.isLetter(initial)) {
    let value = initial;
    while (
      Lexer.isLetter(this.cursor.current) ||
      Lexer.isDigit(this.cursor.current)
    ) {
      value += this.cursor.consume();
    }

    const keywordType = Lexer.getKeywordType(value);
    if (keywordType !== undefined) {
      return { type: keywordType, value };
    }

    if (value.length <= MAX_IDENTIFIER_LENGTH) {
      return { type: TokenType.IDENTIFIER, value };
    }

    throw new Error(
      `Line ${this.line}: Identifier name '${value}' exceeds ${MAX_IDENTIFIER_LENGTH}
characters`
    );
  }

  if (Lexer.isDigit(initial)) {
    let value = initial;
    while (Lexer.isDigit(this.cursor.current)) {
      value += this.cursor.consume();
    }
    return { type: TokenType.CONSTANT, value };
  }

  if (initial === "=") {
    return { type: TokenType.EQUAL, value: "=" };
  }

  if (initial === "-") {
    return { type: TokenType.SUBTRACT, value: "-" };
  }
```

```
if (initial === "*") {
  return { type: TokenType.MULTIPLY, value: "*" };
}

if (initial === "(") {
  return { type: TokenType.LEFT_PARENTHESES, value: "(" };
}

if (initial === ")") {
  return { type: TokenType.RIGHT_PARENTHESES, value: ")" };
}

if (initial === "<") {
  if (this.cursor.current === "=") {
    this.cursor.consume();
    return { type: TokenType.LESS_THAN_OR_EQUAL, value: "<=" };
  }

  if (this.cursor.current === ">") {
    this.cursor.consume();
    return { type: TokenType.NOT_EQUAL, value: "<>" };
  }

  return { type: TokenType.LESS_THAN, value: "<" };
}

if (initial === ">") {
  if (this.cursor.current === "=") {
    this.cursor.consume();
    return { type: TokenType.GREATER_THAN_OR_EQUAL, value: ">=" };
  }

  return { type: TokenType.GREATER_THAN, value: ">" };
}

if (initial === ":") {
  if (this.cursor.current === "=") {
    this.cursor.consume();
    return { type: TokenType.ASSIGN, value: ":=" };
  }

  throw new Error(`Line ${this.line}: Misused colon`);
}
```

```
  if (initial === ";") {
    return { type: TokenType.SEMICOLON, value: ";" };
  }

  if (initial === "\n") {
    this.line++;
    return { type: TokenType.END_OF_LINE, value: "EOLN" };
  }

  throw new Error(`Line ${this.line}: Invalid character '${initial}'`);
}
```

7.3 写入二元式文件

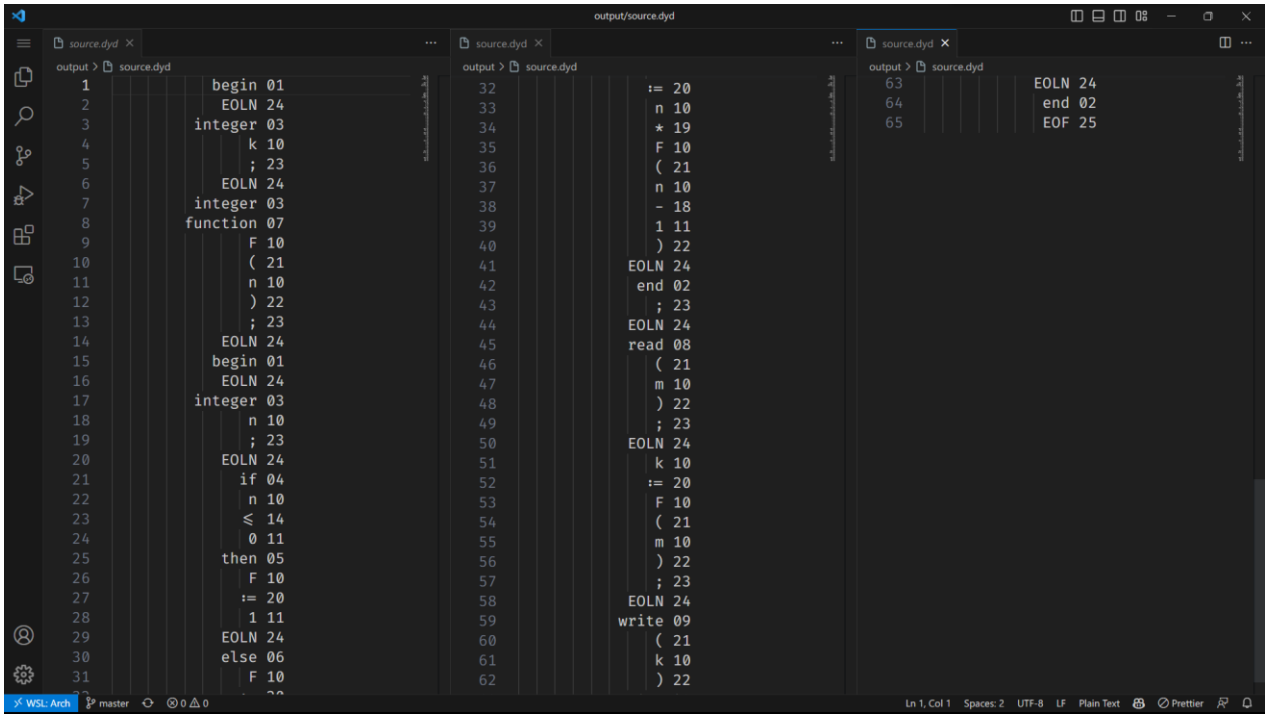
```
private static writeTokens(tokens: Token[]) {
  const text = tokens
    .map((token) => {
      const { type, value } = token;
      return [value.padStart(16), type.toString().padStart(2, "0")].join(" ");
    })
    .join("\n");
  writeFileSync("output/source.dyd", text);
}
```

7.4 写入错误信息文件

```
private static writeErrors(errors: string[]) {
  const text = errors.join("\n");
  writeFileSync("output/source.err", text);
}
```

7.5 检查二元式文件

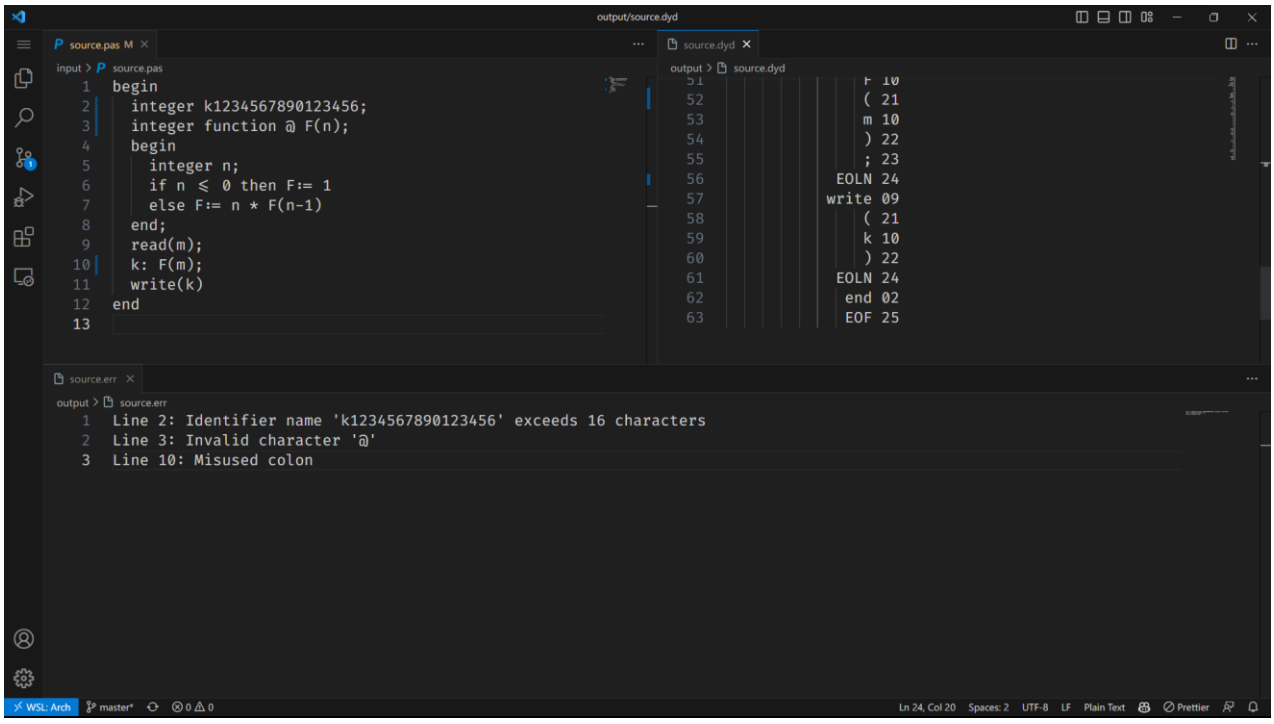
在词法正确的情况下，运行词法分析器。然后打开二元式文件 source.dyd，内容如下：



由此可知，词法分析器识别出了源程序中所有的 65 个符号，并且能够正确地添加行结束符和文件结束符。

7.6 检查错误信息文件

在源程序中引入所有三种类型的错误。把第 2 行改为“integer k1234567890123456”，第 3 行改为“integer function @ F(n)”，第 10 行改为“k: F(m)”。运行后打开错误信息文件 source.err，内容如下：



由此可知，词法分析器识别出了所有三种错误，能够精确到错误的行数，并且提供了高描述性的错误信息。错误的单词没有被写入二元式文件内。

八、实验结论、心得体会和改进建议：

本实验中编写的词法分析器能够正确识别源程序的所有单词，给出现的错误提供高描述性的错误信息和所在行数，完成了实验的所有要求。

通过本次实验，我进一步熟悉了词法分析的原理，对如何使用编程语言实现状态转换图有了亲身的实践。同时我意识到，精确、友好的报错可能和词法分析本身一样重要，这样才能帮助源程序的开发者快速定位错误，履行好编译器的责任。

电 子 科 技 大 学

实　验　报　告

实验二

一、实验名称：语法分析

二、实验学时：4

三、实验内容和目的：

1. 理解语法分析的原理和编程实现方式。
2. 理解递归下降法。
3. 对词法分析生成的二元式文件进行语法分析，输出二元式文件（*.dys）、变量表（*.var）、过程表（*.pro）和错误信息文件（*.err）。

四、实验原理：

4.1 语法分析

语法分析是对词法分析识别出来的符号流（也可看成符号串），按语法规则进行分析，识别出各类语法单位，如表达式、短语、子句、句子和程序等，以便后续步骤进行分析与合成。

语法分析通常是一种结构分析，分析的结果形成一棵语法树（分析树）。如果在分析过程中发现不符合语法规则的符号串，将做出语法出错处理，给出相应的出错信息。

4.2 二元式文件

二元式文件内含有能够被正确解析的单词。如果程序完全正确，那么*.dys 文件就和*.dyd 文件完全一致。

4.3 变量表

变量表中存放程序识别出的所有声明的实参和形参。每个变量拥有下列属性：

1. 变量名：标识符的名字。
2. 所属过程：在哪个过程中被声明。如果是顶级变量，则所属过程为“main”。
3. 分类：实参记为 0，形参记为 1。
4. 类型：变量的类型，在此极小语言中总为“integer”。
5. 层次：在第几层过程嵌套中被声明。
6. 位置：相对于第一个变量的偏移量。第一个变量为 0。

4.4 过程表

过程表中存放程序识别出的所有声明的过程。每个过程拥有下列属性：

1. 过程名：标识符的名字。
2. 类型：过程的返回值类型，在此极小语言中总为“integer”。
3. 层次：过程本身是第几层嵌套。
4. 首变量位置：过程中第一个声明的变量在变量表中的位置。
5. 末变量位置：过程中最后一个声明的变量在变量表中的位置。

4.5 错误信息文件

如果在词法分析结束后，错误信息文件不为空，说明源程序含有词法错误。那么，就没有必要进行语法分析，主控程序可以提前终止。

如果无词法错误，则语法分析将所有的语法错误写入该文件，格式与词法错误相同。

语法错误可以分为 3 类：

1. 缺少符号：在应该出现某符号的位置，出现了另一个符号。
2. 匹配错误：左括号没有右括号与之匹配。
3. 符号无定义或重复定义：使用了未定义的变量，或者在同过程内重复定义同名变量。

4.6 递归下降法

消除公共左因子和左递归后，文法如下：

PG -> SUBPG eof

SUBPG -> begin DCLS EXES end

DCLS -> DCL DCLS'

DCLS' -> DCL DCLS' | e

DCL -> integer DCL';

DCL' -> VARDCL | PRODCL

VARDCL -> ID

VAR -> ID

PRODCL -> function PRONAMEDCL (PARAMDCL); PROBODY

PRONAMEDCL -> ID

PRONAME -> ID

PARAMDCL -> ID

PARAM -> ID

PROBODY -> begin DCLS EXES end

EXES -> EXE EXES'

EXES' -> ; EXE EXES' | e

EXE -> READ | WRITE | ASSIGN | CONDITION

READ -> read (VAR)

WRITE -> write (VAR)

ASSIGN -> VAR := ARITH | PRONAME := ARITH

ARITH -> TERM ARITH'

ARITH' -> - TERM ARITH' | e

TERM -> FACTOR TERM'

TERM' -> * FACTOR TERM' | e

FACTOR -> VAR | CONST | CALL

CALL -> PRONAME (ARITH)

CONDITION = if CDT then EXE else EXE

CDT -> ARITH OP ARITH

OP -> = | <> | < | <= | > | >=

对于其中的每一条，都编写一个对应的函数。终结符使用 match 函数匹配，非终结符调用对应的函数。空产生式意味着如果没有匹配成功，也不需要报错。

五、实验器材（设备、元器件）

笔记本电脑。

六、实验步骤：

1. 读入二元式文件。
2. 根据递归下降文法，从上到下尝试匹配所有符号。
3. 匹配失败则报错。如果不是重大错误，就尝试恢复。
4. 写入二元式、变量表、过程表和错误信息文件。

七、实验数据及结果分析：

7.1 读入二元式文件

```
private static readTokens() {
  const text = readFileSync("output/source.dyd").toString().trim();

  const tokens: Token[] = [];

  for (const line of text.split("\n")) {
    const [value, type] = line.trim().split(" ");

    if (!type || !value) {
      continue;
    }

    tokens.push({ type: +type, value });
  }

  return tokens;
}
```

7.2 递归下降文法

由于函数过多，此处挑选若干典型展示。

按顺序匹配，例如函数声明（PRODCL）：

```
private parseProcedureDeclaration() {
  this.match(TokenType.FUNCTION);
  this.parseProcedureNameDeclaration();
  this.match(TokenType.LEFT_PARENTHESES);
  this.parseParameterDeclaration();
  this.match(TokenType.RIGHT_PARENTHESES);
  this.match(TokenType.SEMICOLON);
  this.parseProcedureBody();
}
```

含有空产生式，例如算术表达式（ARITH）：

```
private parseArithmeticExpression_() {
  if (this.hasType(TokenType.SUBTRACT)) {
    this.match(TokenType.SUBTRACT);
    this.parseTerm();
    this.parseArithmeticExpression_();
  }
}
```

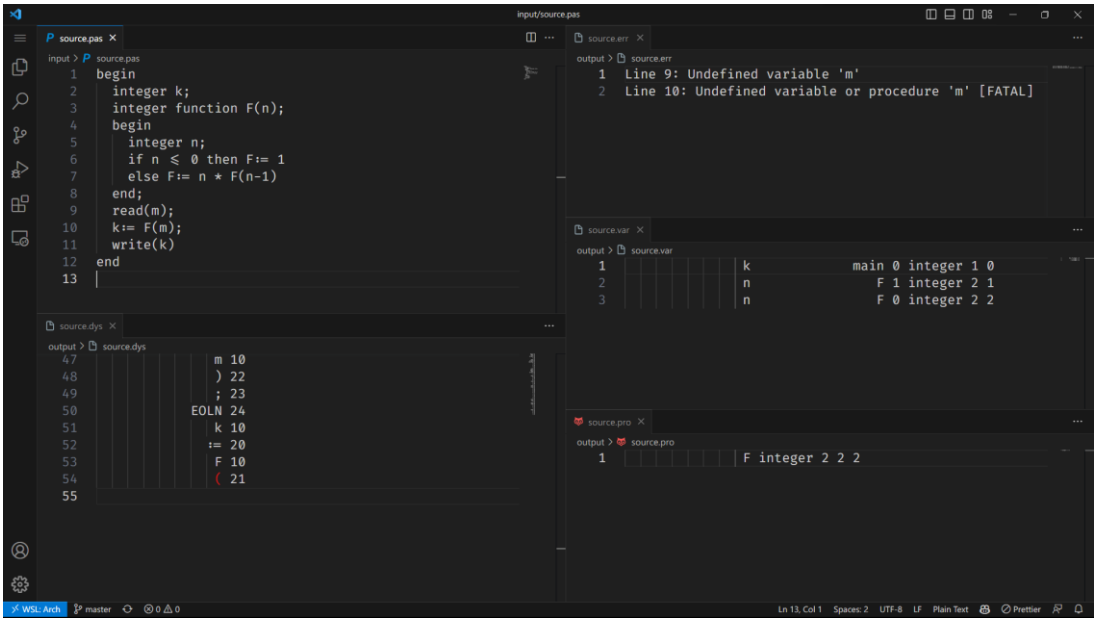
带有副作用，例如变量声明（VARDCL）：

```
private parseVariableDeclaration() {
  const { value } = this.match(TokenType.IDENTIFIER);
  this.registerVariable(value);
}
```

7.3 错误信息

7.3.1 测试一

样例程序中含有一个语法错误，即变量 m 未定义。运行后打开四个输出文件，内容如下：



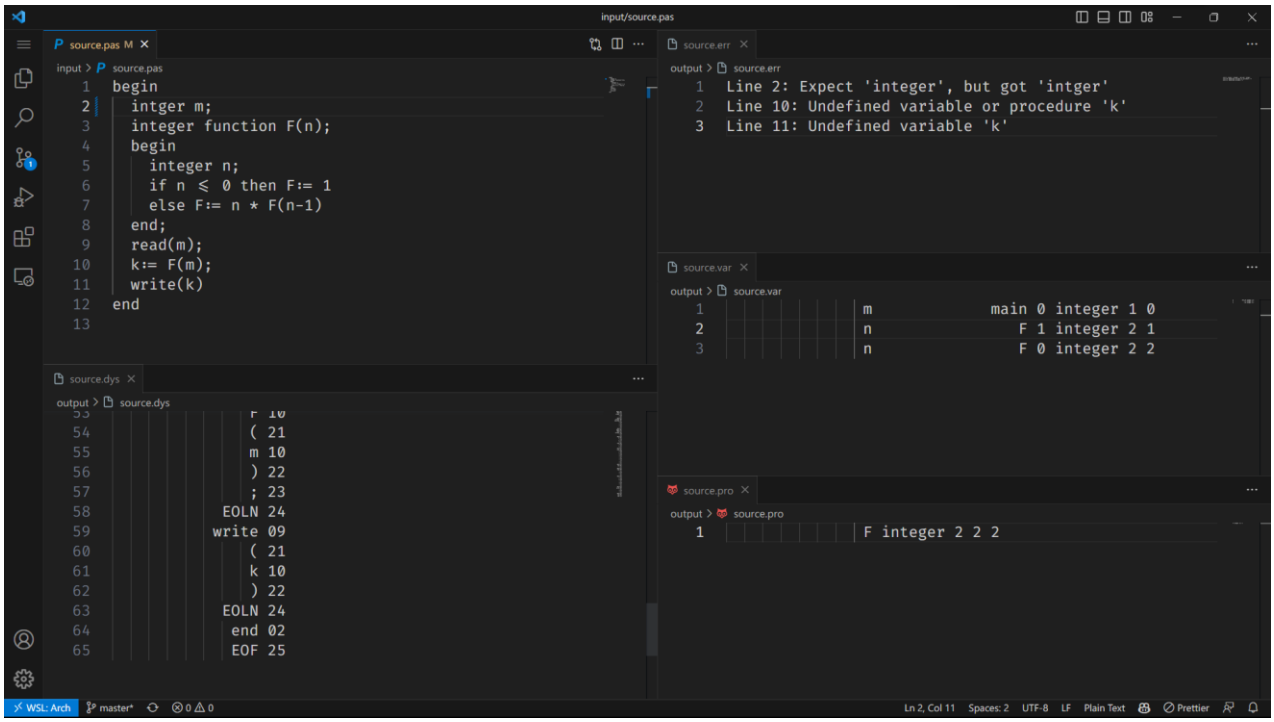
语法分析器在第 9 行和第 10 行都发现了未定义的变量 m。第 9 行的错误并不致命，因为 read 函数接收的肯定是变量。语法分析器即使发现了不认识的标识符，也知道它应当是一个变量，所以报错后可以跳过它继续检查。

但第 10 行的错误是致命的，因为函数调用可以接收一个算术表达式。语法分析器无法确定跳过的是变量还是函数调用的函数名，所以也就无法继续分析。

所以，二元式文件也在第 10 行的 m 之前提前结束。变量表和过程表都正确识别。

7.3.2 测试二

将样例程序的第 2 行改为“intger m”，再次运行观察结果。



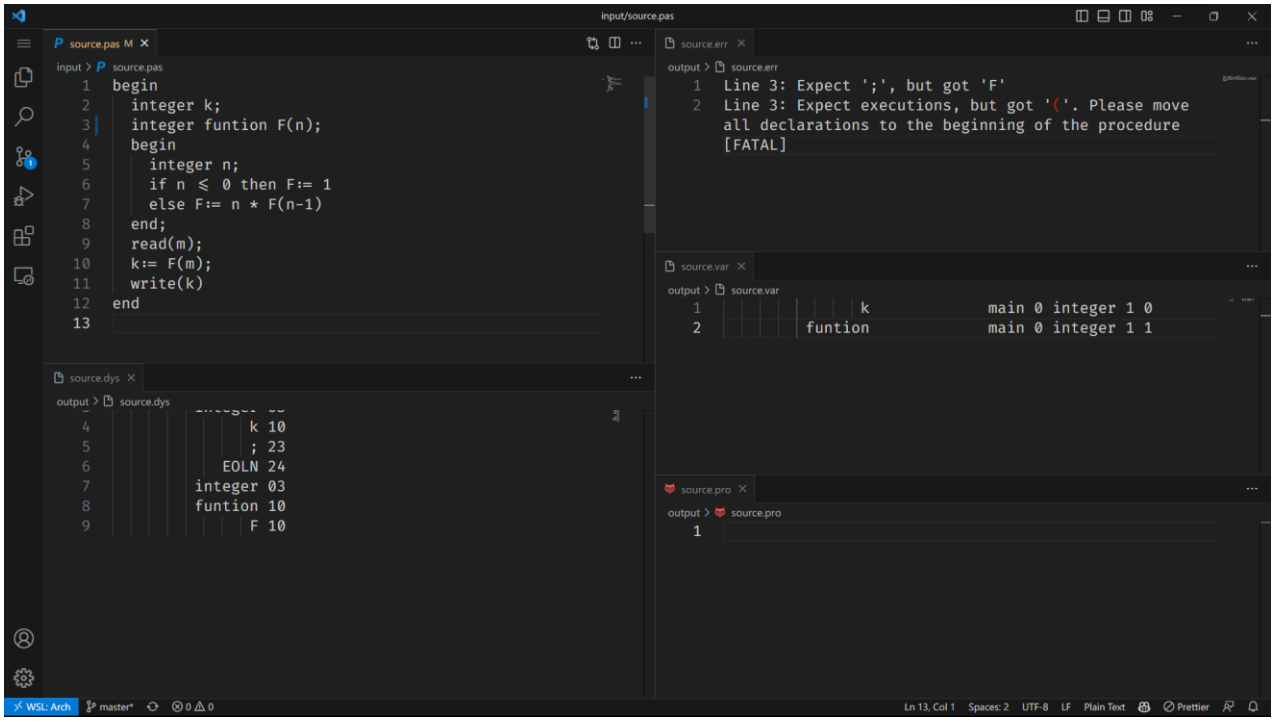
语法分析器在第 2 行报错，因为在期待出现“integer”的地方出现了“intger”。但由于该处后续跟的单词必定是声明的变量，所以变量 m 依旧能够被成功定义。

第 10 行和第 11 行也正确报出了变量 k 未定义的错误。

由于没有出现致命错误，所以二元式文件与词法分析的结果完全一致。变量表和过程表也符合预期。

7.3.3 测试三

将样例程序的第 3 行改为“integer funtion F(n)”，再次运行观察结果。



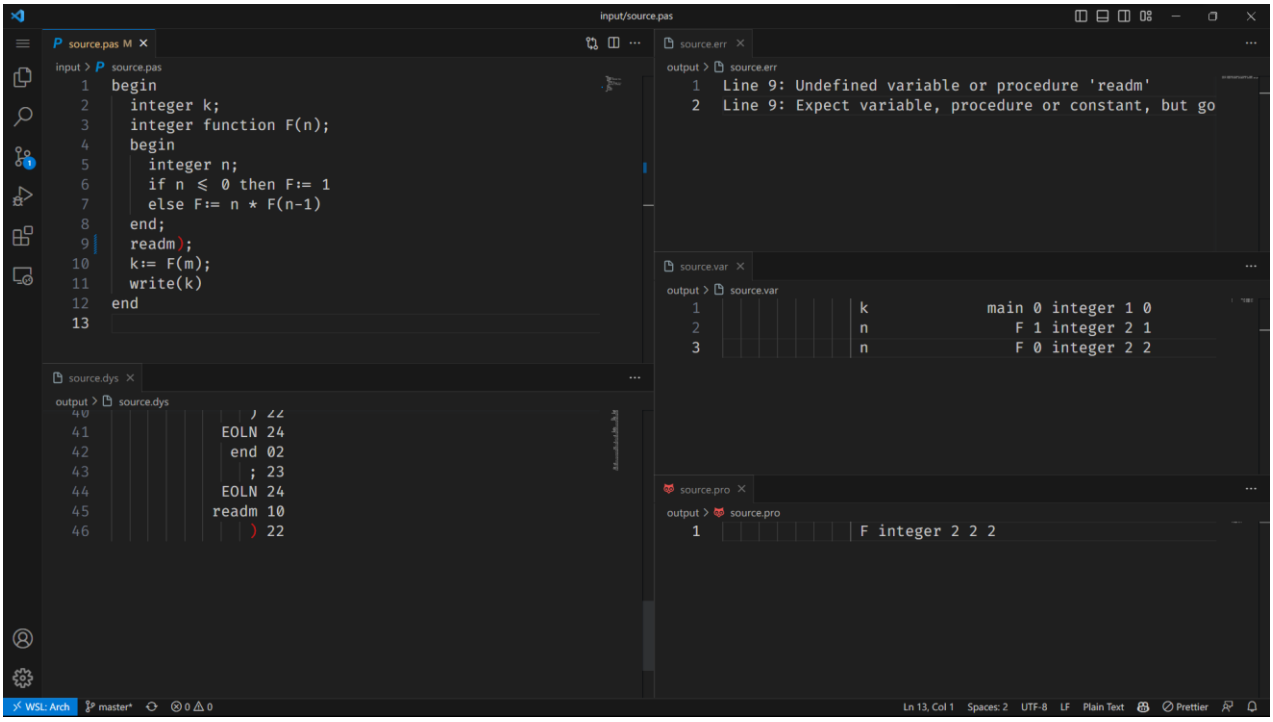
语法分析器在第 3 行报出了匹配错误，因为它误认为 funtion 是一个标识符，从而进一步认为这是一个变量声明语句。于是，它期待 funtion 后面应该跟上“;”，但实际上接收到了“F”。

第 3 行的另一个错误是致命的，所以一行报了两个错误。当语法分析器跳过“F”后，下一个接收到了“(”。因为所有的声明语句都以“integer”开头，所以语法分析器排除了声明语句的可能，那就说明这是执行语句。但执行语句只能以“read”、“write”、标识符或“if”开头，所以发生了错误。并且，就算跳过了“(”，分析器也不知道接下来应该采取哪一条文法进行分析，所以这是致命的错误。

相应地，二元式文件在“F”之后提前结束，k 和 funtion 注册为了变量，没有过程被注册。

7.3.4 测试四

将样例程序的第 9 行改为“readm)”，再次运行观察结果。

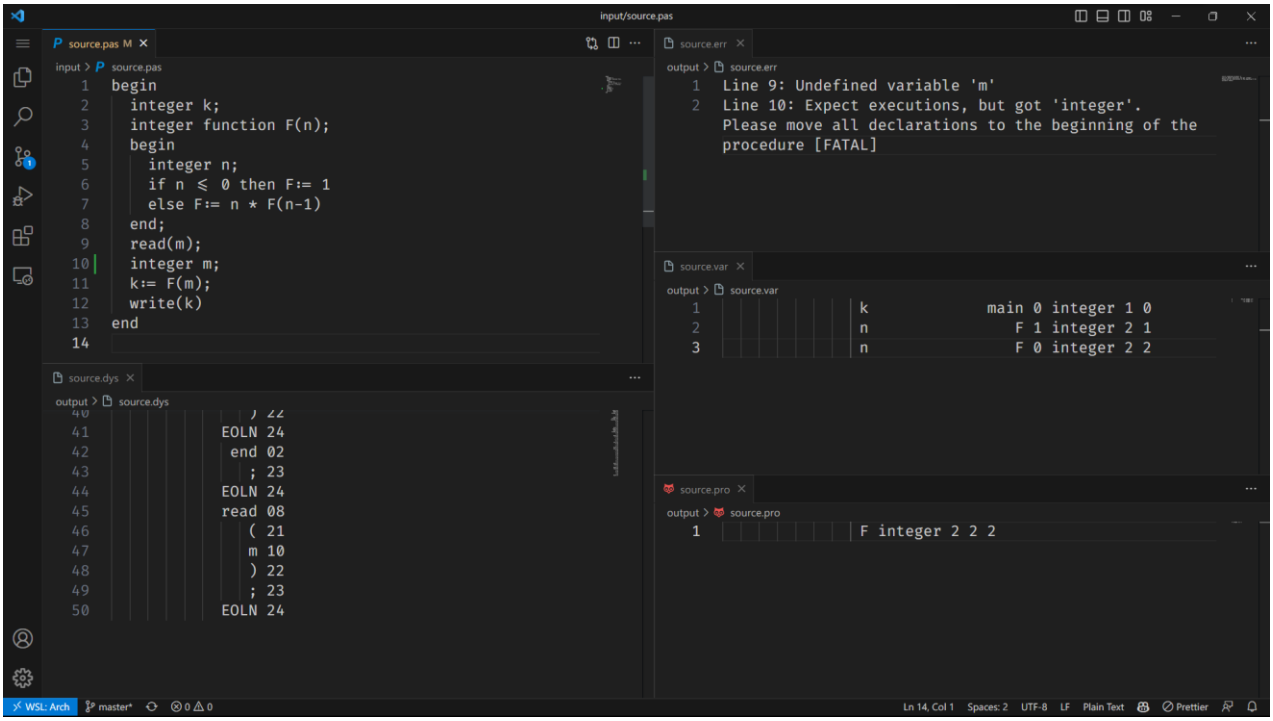


语法分析器在第 9 行接收到了“readm”，误认为是标识符，继而报错该标识符未被定义。也因此，该语句被认为是赋值语句，所以下一个符号应当是“:=”，但报错由于在同一行所以未被报出。最后，应当是算术表达式的地方出现了“;”，成为了致命错误，被语法分析器报出。

二元式文件在“;”之前提前终止，变量表与过程表均符合预期。

7.3.5 测试五

将样例程序的第 9 行后新增一行“integer m”，再次运行观察结果。



语法分析器在第 9 行发现了未定义的变量 m，正确报出了错误。由于所有声明语句都应该出现在执

行语句之前，所以语法分析器在第 10 行报出了致命错误，并提示“请将所有声明语句提到过程的开头”。

二元式文件在声明语句前终止，变量表和过程表均符合预期。

八、实验结论、心得体会和改进建议：

本实验中实现的语法分析器能够使用递归下降法，对词法分析得到的单词流逐个分析，并且尽可能多地报出错误，实现了实验的所有要求。

通过本实验，我巩固了语法分析的原理，对递归下降法及其编程实现有了自己的认识，并且亲身实践实现了一个递归下降语法分析器。同时，我也再次认识到，递归下降本身是较简单的，但是报出人性化的错误信息才是难点，也是一个好的编译器的显著特征。

在实验之前，指导老师可以预先给出样例程序的输出，这样可以让学生更好的理解“层次”“第一个变量”等概念，也能让学生对实验的要求和任务有一个预先的大致把握。