编译器设计实践项目WHI语言编译器WHI\_Compiler文档

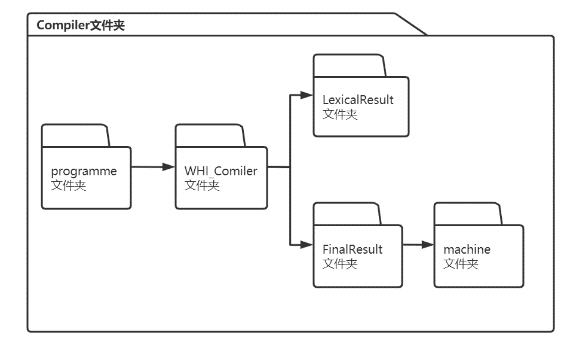
1. 项目简介

本项目是根据已有的WHI语言实现的编译器，WHI\_Compiler。此编译器能够完成WHI语言的编译功能，并且编译产生的生成代码能够在简单栈式抽象机上运行。此项目的输入为WHI语言源程序，输出为两个文件：词法分析结果文件和目标代码生成文件。本文将根据以下几个方面来说明本项目：

1. 项目定义
2. 源语言WHI语言简介
3. 目标语言SSAM抽象机指令简介
4. WHI语言编译器WHI\_Compiler设计与实现
5. WHI语言编译器WHI\_Compiler系统测试
6. 项目定义

项目输入：WHI 语言文件p.txt

项目输出：词法分析结果文件p\_LexResult.txt、目标代码文件p\_FinalResult.txt

项目结构：

Compiler文件夹为工程总文件夹，其下有5个文件夹，分别为：源程序文件夹Programme、词法分析结果文件夹LexicalResult、生成目标代码文件夹FinalResult、WHI编译器项目文件夹WHI\_Compiler和SSAM抽象机项目文件夹machine。具体如下图所示：

1. 源语言WHI语言简介

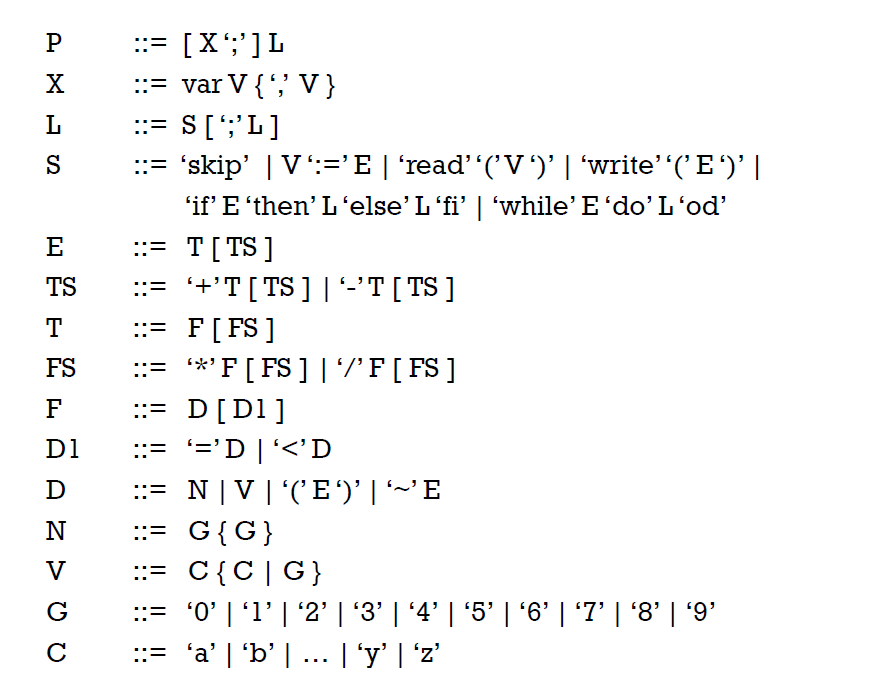
WHI语言是一种基于BNF范式的无二义性语言。在BNF范式文法中，约定[a]代表候选式a是可以有也可以无的；{a}代表候选式a可以出现0次或多次，即用正规式表示的a\*；(a1|a2)x表示x是a1，a2的公共因子，即a1x|a2x。WHI语言文法定义如下：

非终结符集合：

终结符集合：

开始符：

规则集合:



其中，C识别是否为字母；G识别是否为数字；V识别是否为标识符，鉴于文法的定义，标识符只存在变量标识符一种，因而V识别是否为变量；N识别是否为数字（非负整数）； E、TS、T、F、D1、D识别是否为表达式；L、S识别语句，X识别变量声明，P为开始符。

1. 目标语言SSAM抽象机指令简介

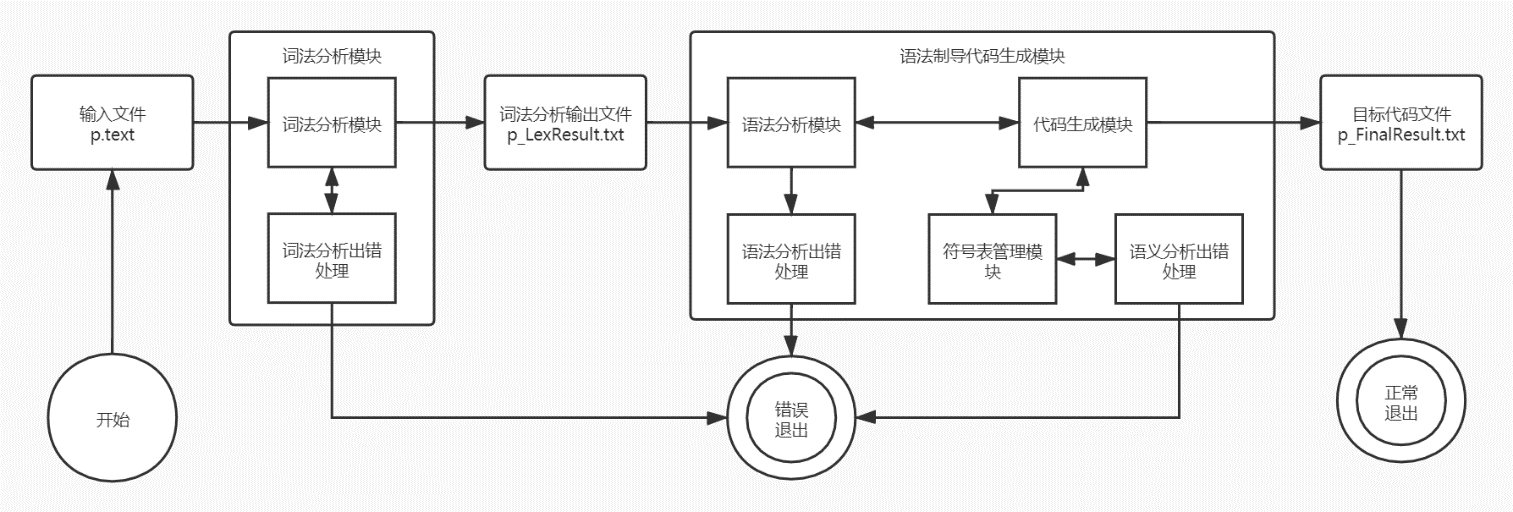
SSAM抽象机为简单栈式抽象机，变量值存放在栈中，大多数指令从栈中取得操作数，并将结果放回栈中。栈顶指针为T，当前指令指针为P。栈顶为运算区，用来保存运算结果。

SSAM抽象机指令结构分为操作符和操作数两部分。其中每条指令都有操作符，但部分指令没有操作数。SSAM指令集如下表所示：

|  |  |  |  |
| --- | --- | --- | --- |
| 指令格式 | | 指令含义 | 指令类型 |
| 操作符 | 操作数 |
| add | 无 | 栈顶和次栈顶相加，结果放回栈顶 | 算数运算类指令 |
| sub | 无 | 栈顶减次栈顶，结果放回栈顶 |
| mul | 无 | 栈顶和次栈顶相乘，结果放回栈顶 |
| div | 无 | 栈顶除以不为0的次栈顶，结果放回栈顶 |
| equ | 无 | 判断栈顶和次栈顶是否相等，结果放回栈顶 | 逻辑运算类指令 |
| lth | 无 | 判断栈顶是否小于次栈顶，结果放回栈顶 |
| lit | a | 将a赋值给栈顶变量，栈顶指针+1 | 赋值指令 |
| lod | a | 将栈中的第a个变量的值赋值给栈顶变量，栈顶指针+1 |
| sto | a | 将栈顶变量的值赋值给栈中的第a个变量，栈顶指针-1 |
| int | a | 栈顶指针+a | 指针指令 |
| jmp | a | 跳转到第a条指令 | 跳转指令 |
| jpc | a | 如果栈顶变量为0，跳转到第a条指令 |
| red | a | 输出数据并存入地址a | 访存指令 |
| wrt | 无 | 弹出栈顶变量并输出其值 |
| swp | 无 | 交换栈顶变量和次栈顶变量值 | 交换指令 |
| nop | 无 | 空操作 | 空指令 |

1. WHI语言编译器WHI\_Compiler设计与实现

一个编译程序将经过以下几个步骤来完成对源语言到目标语言的翻译：词法分析、语法分析、语义分析及中间代码生成、代码优化及目标代码生成。由于WHI语言和SSAM抽象机指令相对简单，可以省略中间代码生成和代码优化的部分。因此WHI\_Compiler将经过词法分析、语法分析、语义分析及代码生成三个步骤来编译WHI语言。在这三个步骤中，使用出错处理和符号表来进行异常事件的管理。

因此WHI\_Compiler主要由三个部分组成：输入输出模块、词法分析模块和语法制导代码生成模块。各个模块与子模块及其关系如下图所示：

WHI\_Compiler 结构图

以下具体说明各部分的用途：

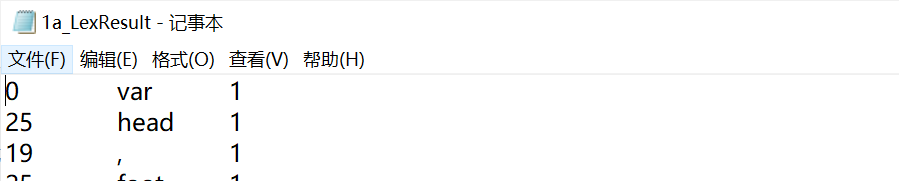
1. 输入输出模块

输入输出模块主要负责系统IO和模块间的交互，主要管理三个文件：源程序文件（系统输入文件p.text）、词法分析输出文件（p\_LexResult.txt）、目标代码文件（系统输出文件p\_FinalResult.txt）。其中源程序文件作为词法分析模块输入文件，存放WHI语言程序；词法分析输出文件存放词法分析得到的单词（Token）流，此文件同时作为语法制导代码生成模块的输入；目标代码文件作为语法制导代码生成模块的输出，存放翻译所得的SSAM抽象机指令。其中两个输出文件的格式如下：

**词法分析输出文件：各个字段之间使用制表符’\t’间隔**

|  |  |  |
| --- | --- | --- |
| Token编码 | Token值 | Token所在行号（从1开始） |

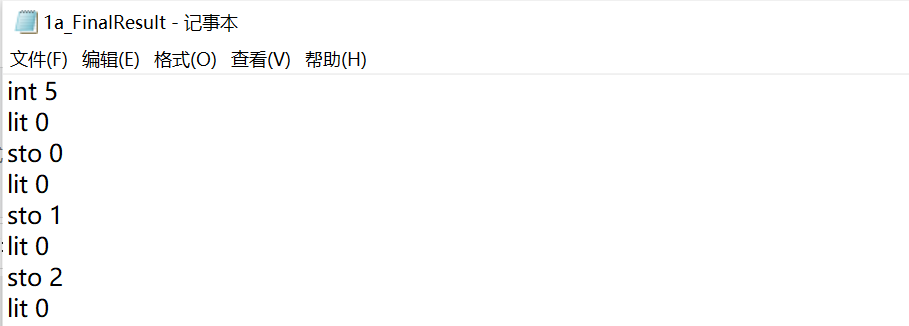
具体文件示例：



**目标代码生成文件：各个字段之间使用制表符’\t’间隔**

|  |  |
| --- | --- |
| 指令操作码 | 指令操作数（如果有） |

具体文件示例：





1. 词法分析模块

词法分析模块包含两个子模块：词法分析子模块和词法分析出错处理子模块，具体如下所示：

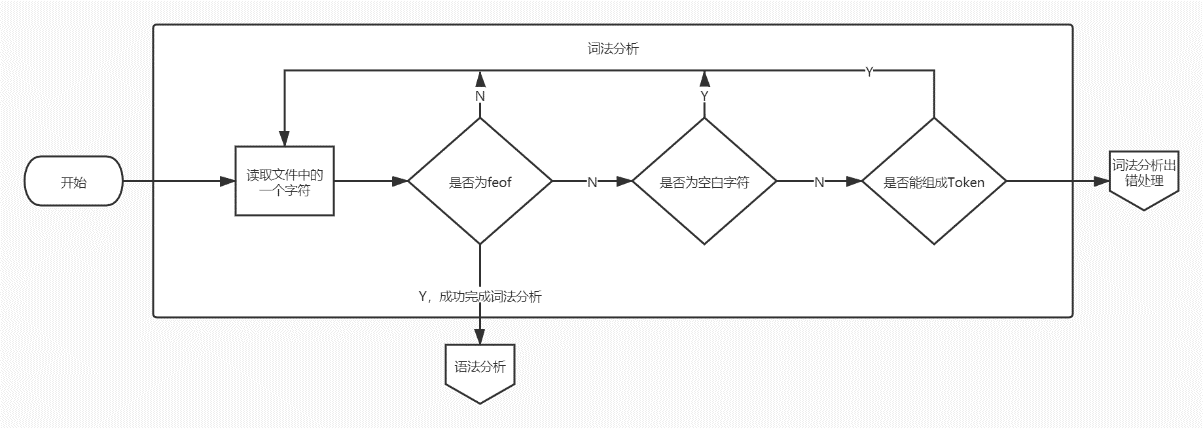
1. 词法分析模块
   1. 模块定义

**输入**：p.text

**输出**：p\_LexResult.txt

功能：将源程序字符流通过文法DFA识别转换为Token流。

具体流程如下图所示：

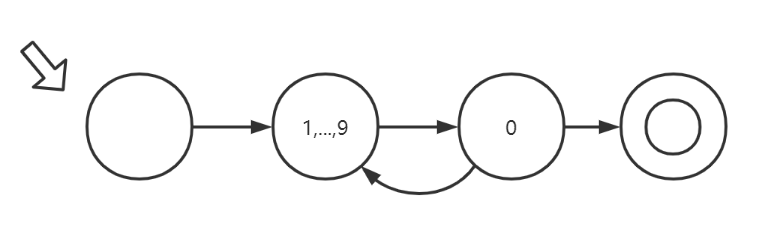


词法分析流程图

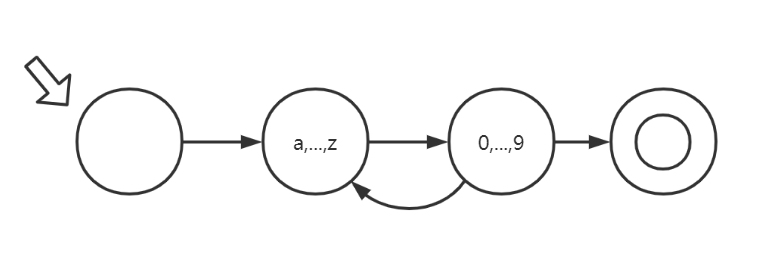
* 1. 模块原理

词法分析的基本原理是将文法转换成一个最小化的确定性有穷自动机（DFA）,根据DFA来构造一个词法分析程序。由于词法分析只识别Token,而不必考虑Token之间的关系是否符合文法，因此只需考虑和Token定义相关文法。在WHI语言中，Token的类型分为以下几种：常数、标识符、关键字、运算符、界符、空白符号（\n,\t,\0）和文件结束符，其中后两种符号不必作为Token流的输出。因此，只需将定义以上符号的文法转换为DFA即可。各种符号对应的DFA如下所示：

**常数：**

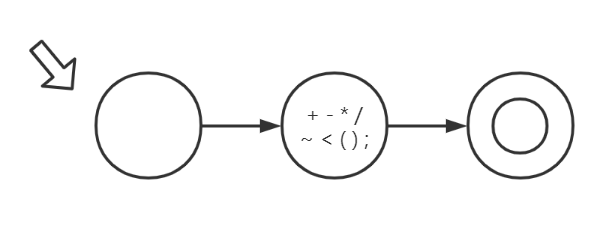


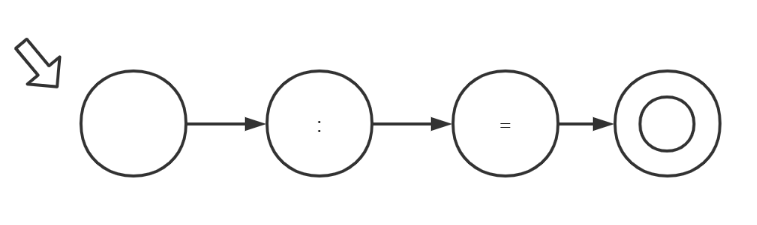
**标识符和关键字：**



识别出标识符后，要查关键字表判断是否为关键字，标识符不能使用关键字已经定义的符号串。

**运算符和界符：**





随后根据DFA构造词法分析程序，具体在模块实现中说明。

* 1. 模块实现

根据DFA构造词法分析程序，首先定义Token数据结构，以及相关的变量和常量；其次根据DFA构造词法分析函数，判断所输入的字符串流是否符合文法对Token的定义；最后将符合文法的Token输出，如果出现错误转到词法分析出错处理模块。

* + 1. 重要的数据结构，变量及常量

**Token定义：**

为了方便管理，首先将Token编码：关键字和运算符按照一字一种编码；标识符和常数按照一类一种编码，具体如下表所示。

|  |  |  |
| --- | --- | --- |
| Token 编码 | Token值 | Token类别 |
| 0 | var | 关键字Keyword |
| 1 | skip | 关键字Keyword |
| 2 | read | 关键字Keyword |
| 3 | write | 关键字Keyword |
| 4 | if | 关键字Keyword |
| 5 | then | 关键字Keyword |
| 6 | else | 关键字Keyword |
| 7 | fi | 关键字Keyword |
| 8 | while | 关键字Keyword |
| 9 | do | 关键字Keyword |
| 10 | od | 关键字Keyword |
| 11 | + | 算术运算符Arithmetic Operator |
| 12 | - | 算术运算符Arithmetic Operator |
| 13 | \* | 算术运算符Arithmetic Operator |
| 14 | / | 算术运算符Arithmetic Operator |
| 15 | ~ | 算术运算符Arithmetic Operator |
| 16 | := | 赋值运算符Assigning Operator |
| 17 | < | 逻辑运算符Logical Operator |
| 18 | = | 逻辑运算符Logical Operator |
| 19 | , | 界符comma |
| 20 | ; | 界符semicolons |
| 21 | ( | 界符Left Brackets |
| 22 | ) | 界符Right Brackets |
| 23 | \n,\t,制表符 | 未输出 |
| 24 | EOF | 未输出 |
| 25 | 示例：abc123 | 标识符Identifier |

根据此表在代码构造关键字表kwList和Token编码表TokenKind，如下所示：

const char kwList[11][10]

{

"var","skip","read","write","if","then","else","fi","while","do","od"

};

enum TokenKind

{

kw\_var,//0

kw\_skip,//1

kw\_read,//2

kw\_write,//3

kw\_if,//4

kw\_then,//5

kw\_else,//6

kw\_fi,//7

kw\_while,//8

kw\_do,//9

kw\_od,//10

add,//11

sub,//12

multip,//13

division,//14

negation,//15

valuation,//:=16

less,//<17

equal,//=18

comma,//,19

semicolons,//;20

left\_brkt,//(21

right\_brkt,//)22

nul,//23. 空格，回车，制表符

eof,//24 文件结尾符

identifier,//25. 标识符

number//26.数字

};

定义数据类型Token：

struct Token

{

TokenKind type;

char lexeme[MAX\_TK\_LEN + 1];

int row;

LexErr lex\_err;

};

其中MAX\_TK\_LEN为可识别的最大符号串长度：#define MAX\_TK\_LEN 100

定义存放Token的数组tkarray以及计数器tk\_num：

Token tkarray[1000];

int tk\_num = 0;

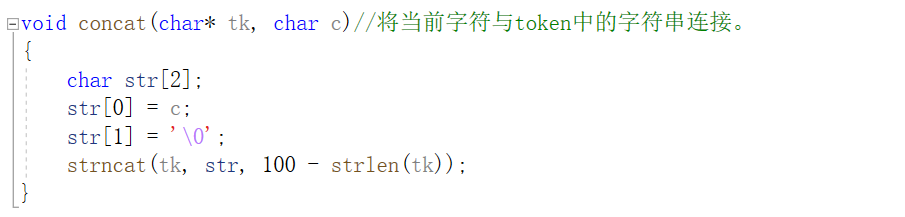
* + 1. 函数定义与关键代码与关键代码

词法分析函数主要分为三类：符号操作函数，DFA识别函数和输出函数。其中符号操作函数用来进行符号识别、合并等操作；DFA识别函数用来根据DFA识别符号串是否为Token；输出函数负责将识别到的Token流输出到屏幕和文件，具体定义和关键代码如下：

**符号操作函数**

|  |  |
| --- | --- |
| 函数原型 | 函数说明 |
| int letter(char c) | 判断字符c是否为字母 |
| int digital(char c) | 判断字符c是否为常数 |
| void concat(char\* tk, char c) | 合并字符串tk和字符c |

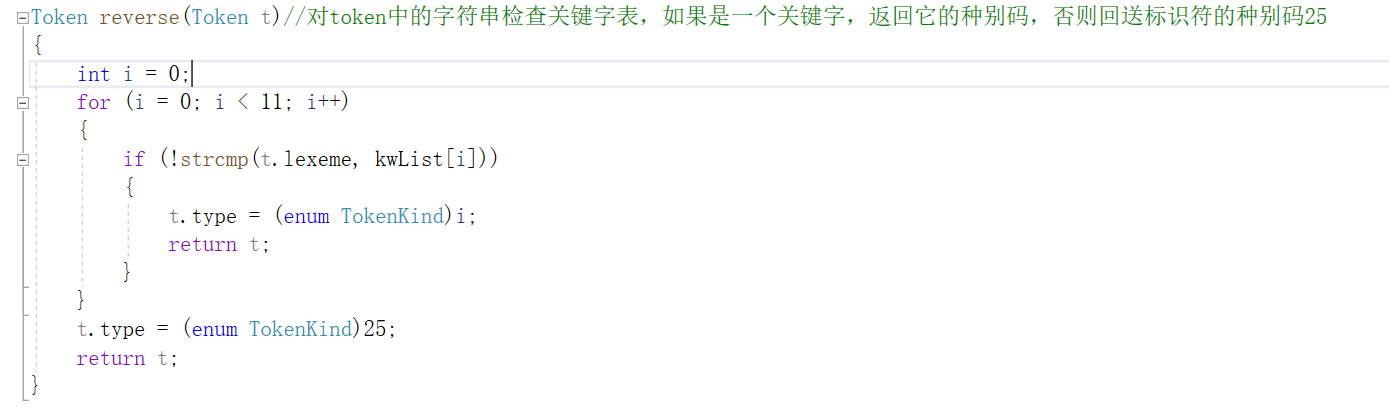
concat函数代码如下：



**DFA识别函数**

|  |  |
| --- | --- |
| 函数原型 | 函数说明 |
| Token reverse(Token t) | 判断t是否为关键字中的Token |
| int LexAys(FILE\* fpr, FILE\* fpw) | 词法分析 |

reverse函数代码如下：



LexAys函数伪代码代码(识别标识符的部分)如下：

int LexAys(FILE\* fpr, FILE\* fpw)

{

从文件中循环读取字符c：

从零开始记录行号；

当c为‘EOF’时：

词法分析完毕，退出此函数；

当c不为‘\n’或‘\t’或‘ ’时：

识别是否为标识符，如果是则输出，退出循环；

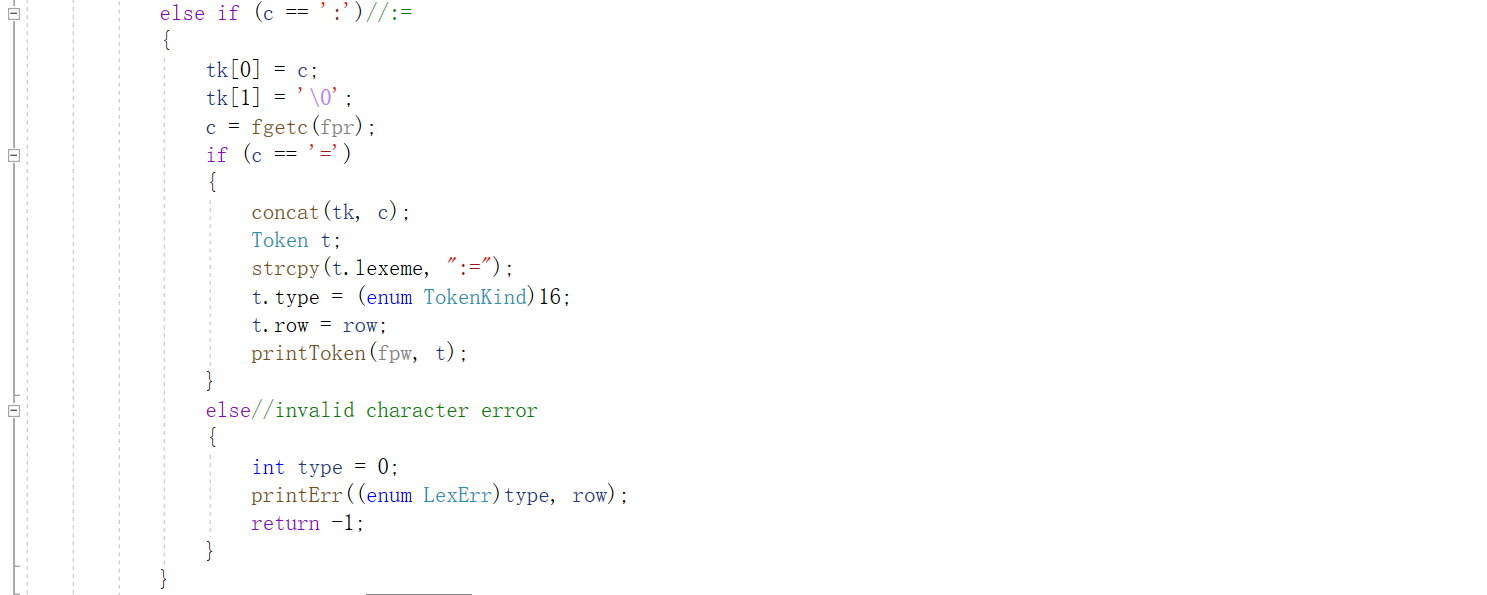
识别是否为常数，如果是则输出，退出循环；

识别是否为运算符中的符号，如果是则输出，退出循环；

如果不为以上任意一种Token，报错，退出函数。

}

其中识别运算符:=的代码如下所示：



**输出函数**

|  |  |
| --- | --- |
| 函数原型 | 函数说明 |
| int printToken(FILE\* f, Token t) | 输出Token |

函数部分代码如下：



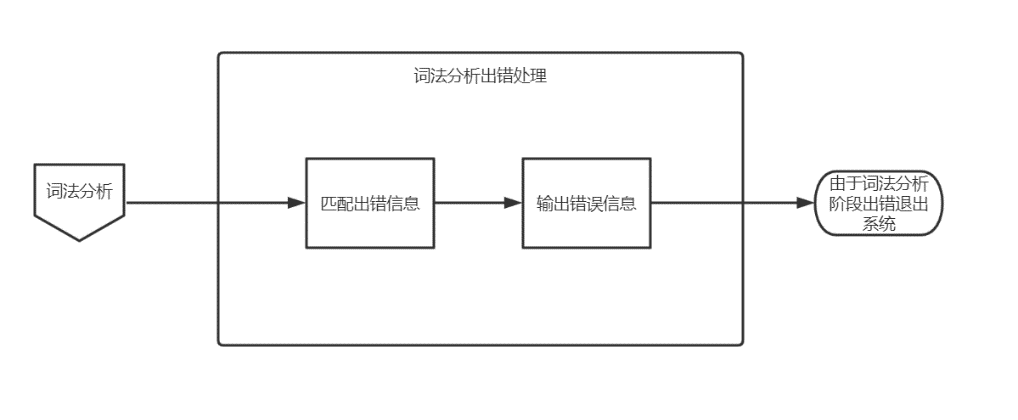
1. 词法分析出错处理模块
   1. 模块定义

输入：出错类型和行号

输出：出错信息

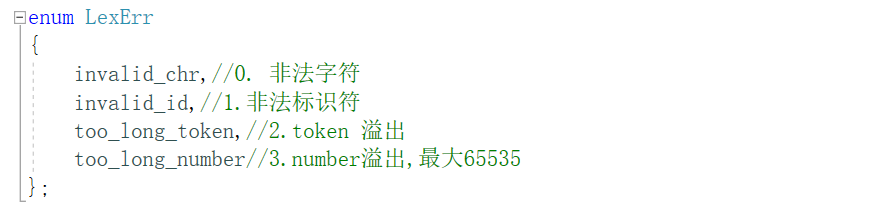
功能：提示源程序在词法分析阶段出错以及出错相关信息

流程图如下所示：



* 1. 模块实现
     1. 重要的数据类型，变量及常量

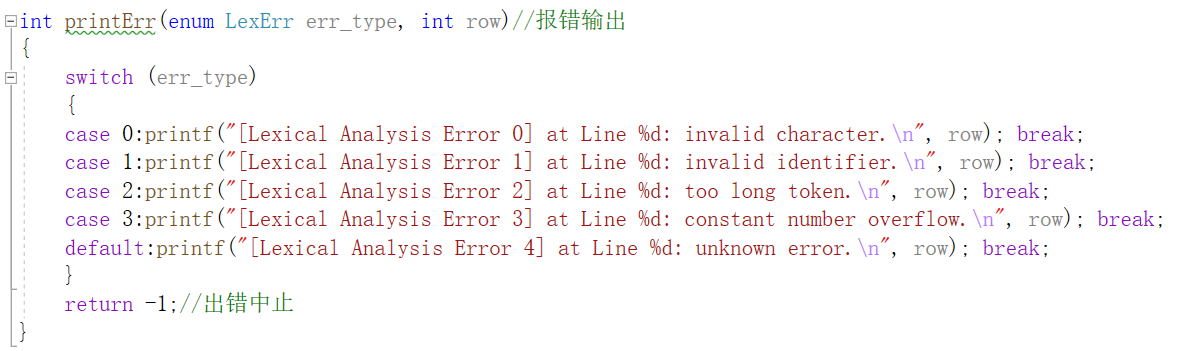
错误类型：根据词法分析的过程，定义了以下几种错误类型：



* + 1. 函数定义与关键代码与关键代码

|  |  |
| --- | --- |
| 函数原型 | 函数说明 |
| int printErr(enum LexErr err\_type, int row) | 输出出错类型和行号 |

printErr函数代码如下：



* + 1. 模块运行流程

1. 语法制导代码生成模块

语法制导代码生成模块包含5个子模块：语法分析模块、语法分析出错处理模块、代码生成模块、语义分析出错处理模块和符号表管理模块，具体如下所示：

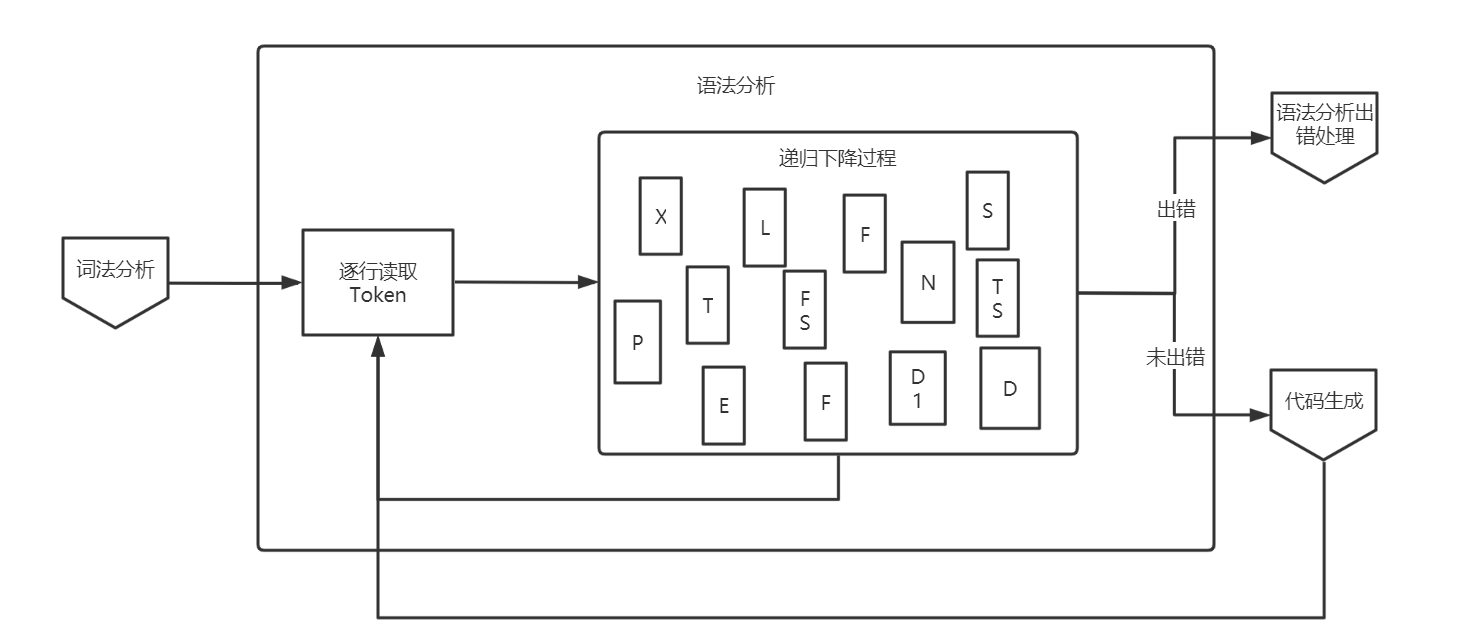
1. 语法分析模块
   1. 模块定义

**输入**：p\_LexResult.txt

**输出**：语法分析的状态信息：成功为1，出错为0

功能：对词法分析得到的Token流进行语法分析，判断Token之间的先后关系是否符合文法对语法的定义，如果符合，进入代码生成模块生成相关代码，如果不符合，转到语法分析出错处理模块提示出错。

流程图如下所示：



* 1. 模块原理

WHI文法是BNF范式的文法，无二义性和左递归，满足递归下降语法分析的条件，因此本项目采用递归下降语法分析作为语法分析的方法。递归下降语法分析的基本思想是：根据每一个非终结符构建一个递归下降子过程，在每个子过程中根据文法自上而下地判断Token流是否符合语法，通过子过程之间的递归调用来实现语法分析。如果遇到不确定进入哪个子过程的情况，只需向前看一个Token就能判断进入子过程，从而避免了在直接自上而下语法分析中需要回溯的情况，提升了分析效率。

* 1. 模块实现
     1. 重要的数据类型，变量及常量

语法分析结束状态：int state\_parsing

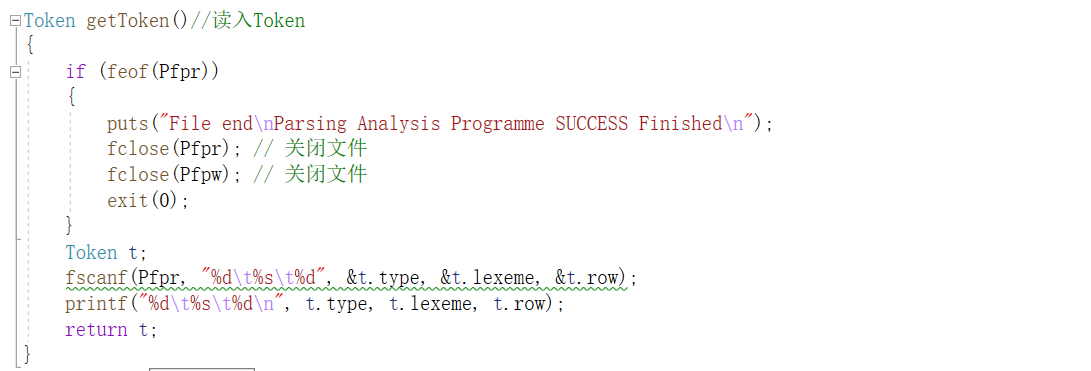


* + 1. 函数定义与关键代码

语法分析函数分为三类：Token输入函数、语法分析顶层函数和各个非终结符的递归下降函数。

**Token输入函数**从文件中读取Token，并存储在数组中，具体定义如下所示：

|  |  |
| --- | --- |
| 函数原型 | 函数说明 |
| Token getToken() | 从p\_LexResult.txt文件中读取Token |



如果读取到文件结束符feof,代表语法分析结束；否则逐行读取文件，返回该行的Token。

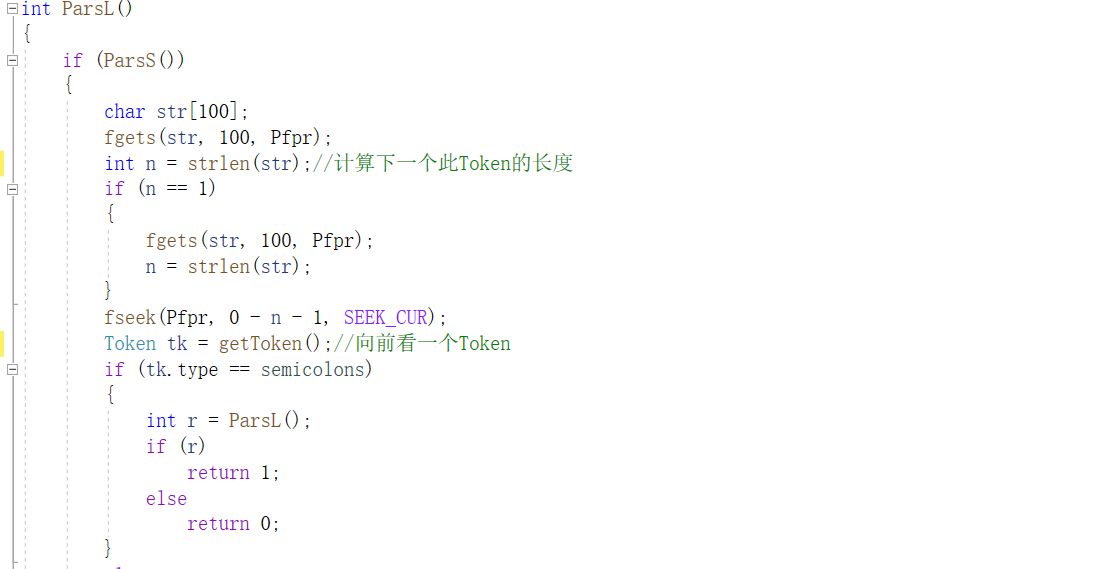
**语法分析顶层函数**和各个**非终结符的递归下降函数**用来进行递归下降语法分析，具体定义如下所示：

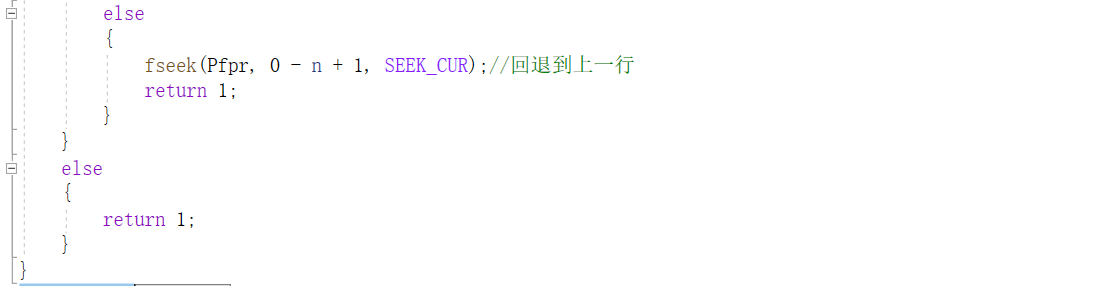
|  |  |
| --- | --- |
| 函数原型 | 函数说明 |
| int ParsV(Token& t) | 非终结符V递归下降子过程 |
| int ParsN(Token& t) | 非终结符N递归下降子过程 |
| int ParsD1() | 非终结符D1递归下降子过程 |
| int ParsD() | 非终结符D递归下降子过程 |
| int ParsF() | 非终结符F递归下降子过程 |
| int ParsFS() | 非终结符FS递归下降子过程 |
| int ParsT() | 非终结符T递归下降子过程 |
| int ParsTS() | 非终结符TS递归下降子过程 |
| int ParsE() | 非终结符E递归下降子过程 |
| int ParsS() | 非终结符S递归下降子过程 |
| int ParsX() | 非终结符X递归下降子过程 |
| int ParsL() | 非终结符L递归下降子过程 |
| int ParsP() | 非终结符P递归下降子过程 |
| int ParsAys() | 语法分析函数 |

ParsAys函数代码如下：



由于递归下降各个子过程实现方法相同，因此仅展示ParsL函数的代码：





由于文法：L := S[‘;’L]，在L的递归下降子过程中，在识别S后不确定是否再继续识别‘;’，因此需要向前看一个Token，判断这个Token是不是‘;’如果是，继续识别；如果不是，则完成L的语法分析。由于是直接从文件中读入Token流，向前看一个Token就提前输入了一个Token，影响之后的语法分析，因此需要使用fseek()函数把文件指针移到提前输入的Token之前。由于fseek()函数的参数为文件指针和文件指针移动的大小，所以使用fgets()和strlen()函数提前识别提前输入的Token在文件中所占用的长度。在getToken()函数中，使用的输入函数是fscanf函数，由于这两个输入函数的机制不同，因此二者在移动文件指针时所需要的大小是不同的。

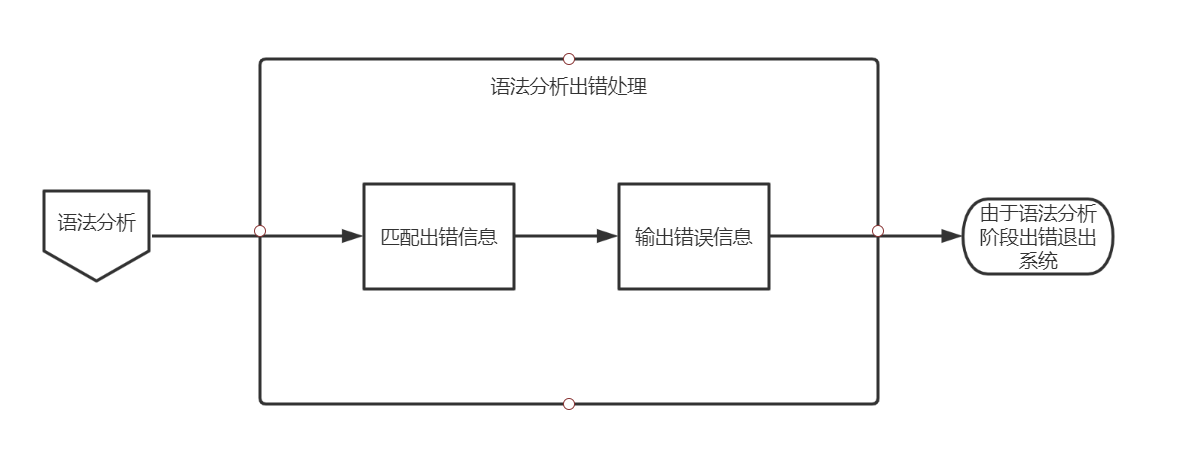
1. 语法分析出错处理模块
   1. 模块定义

**输入**：语法分析错误类型和出错行号

**输出**：语法分析错误信息

功能：提示源程序在语法分析阶段出错，并在屏幕上输出语法分析错误

流程图如下：



* 1. 模块实现
     1. 重要的数据类型，变量及常量

根据递归下降语法分析过程，定义以下语法分析错误类型：

|  |  |
| --- | --- |
| Enum元素 | 说明 |
| excepted\_identifier | expected an identifier |
| excepted\_number | expected a number |
| expected\_right\_brkt | expected a ')' |
| expected\_left\_brkt | expected a '(' or a '~' |
| expected\_equal\_less | expected a '=' or a '<' |
| expected\_multip\_divi | expected a '\*' or a '/' |
| expected\_add\_sub | expected a '+' or a '- |
| excepted\_skip | expected 'skip' or 'read' or 'write' or 'if' or 'while' or an identifier. |
| excepted\_then | expected 'then' |
| excepted\_else | expected 'else' |
| excepted\_fi | expected 'fi' |
| excepted\_do | expected 'do' |
| excepted\_od | expected 'od' |
| excepted\_valuation | expected a ':=' |
| excepted\_var | expected 'var |
| expected\_semicolons | expected ';' |

代码如下：



* + 1. 函数定义与关键代码

语法分析出错处理函数定义如下表所示：

|  |  |
| --- | --- |
| 函数原型 | 函数说明 |
| void ParsError(int row, ParsErr err) | 语法分析出错处理 |

代码如下：

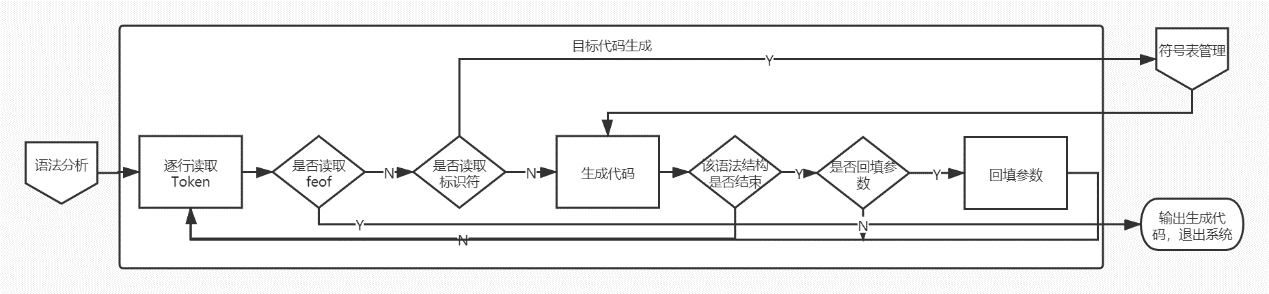


1. 代码生成模块
   1. 模块定义

**输入**：语法分析中各个递归下降子过程分析成功信息，当前运算或语句信息。 **输出**：相应的SSAM抽象机指令

功能：判断是否符合语义，如果符合，生成目标代码；否则转到语义分析出错处理模块。

流程图如下所示：



* 1. 模块原理

本项目采用语法制导的代码生成，具体方法为在一个语法结构识别结束后，在递归下降语法分析中，根据属性文法规定的语义动作，插入代码生成的相应代码。在WHI文法中有以下几种语法结构变量声明、赋值语句、表达式、read语句、write语句、if分支语句和while循环语句。每种语法结构生成不同代码，如下文所示：

**变量声明**

文法:X:=var V { ‘,’ V }

语义动作:

计算变量个数

检查符号表中是否已存在

如果存在，报错

如果不存在，向符号表中加入该变量

为这些变量赋初值0

为变量分配存储空间

生成代码示例

var x , x1;

|  |  |
| --- | --- |
| 0 | int 2 |
| 1 | lit 0 |
| 2 | sto 0 |
| 3 | lit 0 |
| 4 | sto 1 |

**赋值语句**

文法：S->V:=E

语义动作：

在符号表中查询变量V,返回地址，如果不存在则报错

将表达式E的结果加载到栈顶

将表达式E的结果存储在V相应的地址中

生成代码示例：

x:=x1

|  |  |
| --- | --- |
| 5 | lod 1 |
| 6 | sto 0 |

**表达式**

文法：

E->T[TS]

TS->’+’T[TS]|’-’T[TS]

T->F[FS]

FS->’\*’[FS]|’/’F[FS]

F->D[D1]

D1->’=’D|’<’D

D->N|V|’(‘E’)’|’~’E

语义动作：

加载变量（查找符号表）或常数

生成运算对应指令

生成代码示例：

x+1

|  |  |
| --- | --- |
| 0 | lod 0 |
| 1 | lit 1 |
| 2 | add |

**read语句**

文法：S ->read(V)

语义动作：

在符号表中查询变量V,返回地址，如果不存在则报错

将键盘读入的值存储在变量V地址中

生成代码示例：

read (x)

|  |  |
| --- | --- |
| 7 | read 0 |

**write语句**

文法：S ->write(E)

语义动作：

将表达式E的值输出

生成代码示例：

write (x)

|  |  |
| --- | --- |
| 8 | lod 0 |
| 9 | write 0 |

**if分支语句**

文法：S-> if E then L1 else L2 fi

语义动作：

加载表达式E的值

判断E的值，如果等于0，跳转到else,否则顺序执行

生成L1代码

跳转到fi

将当前指令地址回填到2

生成L2代码

将当前指令地址回填到4

生成代码示例：

if x< 10 then

x:= 3

else

x:= 2

fi

|  |  |
| --- | --- |
| 10 | lod 0 |
| 11 | lit 10 |
| 12 | lth |
| 13 | jpc 17 |
| 14 | lit 3 |
| 15 | sto 0 |
| 16 | jmp 19 |
| 17 | lit 2 |
| 18 | sto 0 |

**while循环语句**

文法：S ->while E do L od

语义动作：

加载表达式E的值

判断E的值，如果等于0，跳转到od,否则顺序执行

生成L代码

跳转到 while

将当前指令地址回填到2

生成代码示例：

while x<x1 do

x :=x+1;

x1 :=x+x1

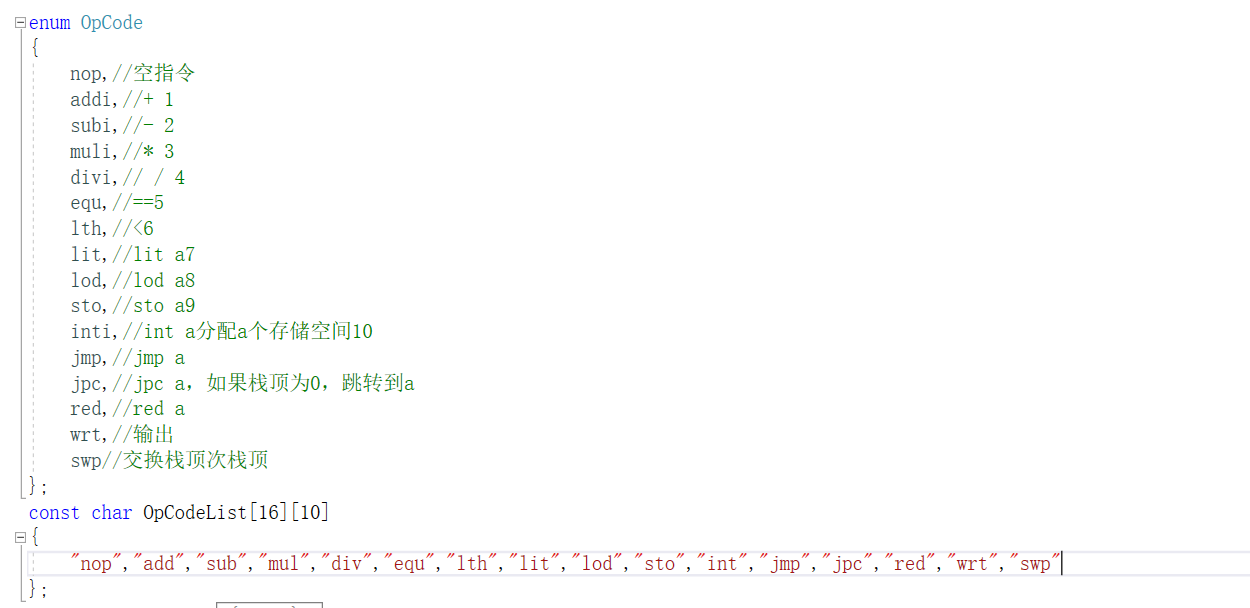
od;

|  |  |
| --- | --- |
| 19 | lod 0 |
| 20 | lod 1 |
| 21 | lth |
| 22 | jpc 32 |
| 23 | lit 1 |
| 24 | sto 0 |
| 25 | add |
| 26 | sto 0 |
| 27 | lod 0 |
| 28 | lod 1 |
| 29 | add |
| 30 | sto 1 |
| 31 | jmp 19 |

* 1. 模块实现
     1. 重要的数据类型，变量及常量

定义指令格式：指令由操作码和操作数两部分组成，部分指令无操作数，无操作数的指令操作数为-1，表示占位。具体指令集已在[SSAM抽象机简介](#SSAM抽象机指令集)中展示。

根据指令集定义**操作码变量与常量**：其中OpCode用于选择生成的指令的操作码（存放在OpCodeList中），但由于部分指令操作码和C语言关键字相同，因此在OpCode中对部分指令码的部分做出变形：如int改为inti。



定义**指令类型**：



定义指令数组，用于存放生成的指令：



定义当前指令地址pc：

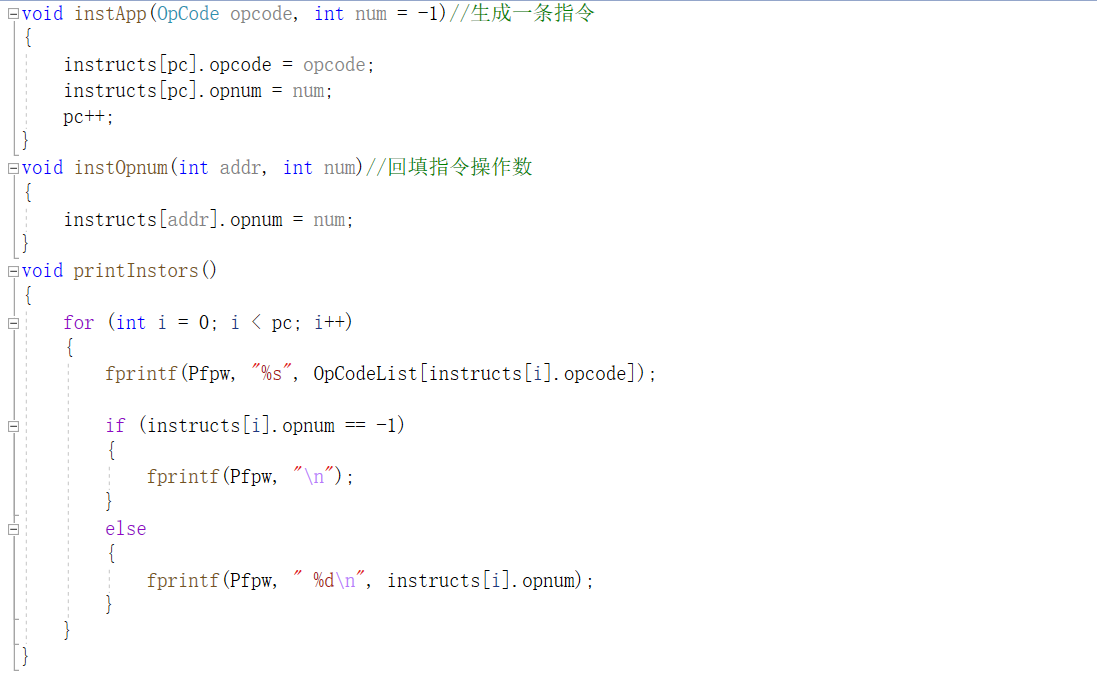


* + 1. 函数定义与关键代码

在代码生成中，有三个函数：指令生成函数，回填指令函数和输出函数。指令生成函数用来生成指令，但对于部分指令而言，如jmp指令，生成指令时不确定指令的操作数，只能在语法结构识别完成后才能确定，在这种情况下使用回填指令函数回填相应指令的操作数。因此在生成指令时，需要存储当前指令地址pc，以便于进行回填操作。这三个函数定义如下：

|  |  |
| --- | --- |
| 函数原型 | 函数说明 |
| void instApp(OpCode opcode, int num = -1) | 生成一条指令 |
| void instOpnum(int addr, int num) | 回填指令操作数 |
| void printInstors() | 输出指令到文件 |

代码如下：

* + 1. 模块使用示例

代码生成的代码是直接在语法分析中完成的，以下是一个例子：

 If语句的代码生成：

在生成if条件判断语句后的跳转指令时，由于不确定then后的语句的长度，所以需要在then执行完毕，else处回填if处条件判断失败后的地址。同样在生成then执行完毕的跳转指令时，不确定else后的语句长度，所以要在else执行完毕后，在fi处回填跳转指令的地址，完成相应的跳转功能。

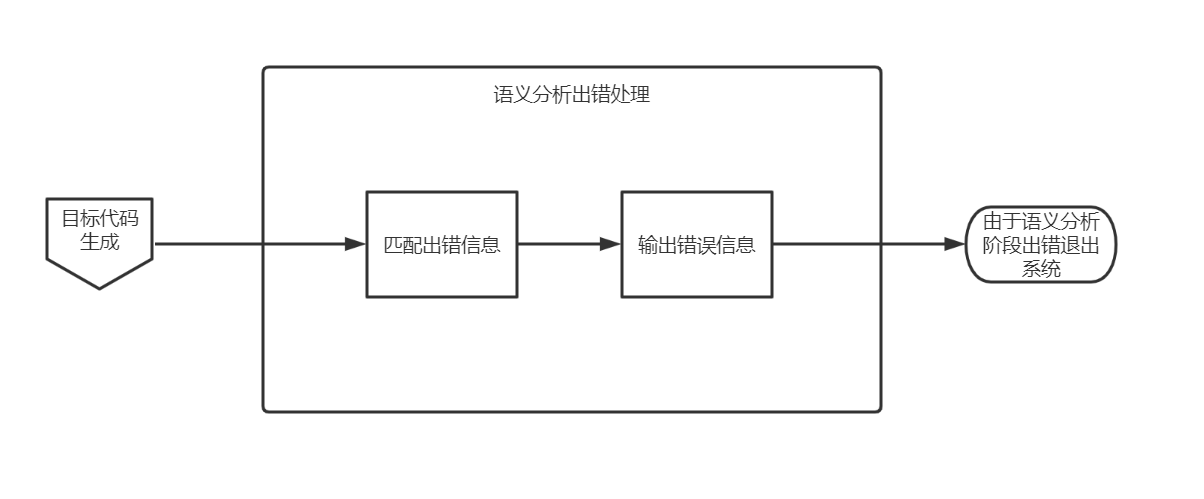
1. 语义分析出错处理模块
   1. 模块定义

**输入**：语义分析错误类型和出错行号

**输出**：语义分析错误信息

功能：提示源程序在语义分析阶段出错，并在屏幕上输出语义分析错误

流程图如下所示：



* 1. 模块实现
     1. 重要的数据类型，变量及常量

语义分析出错仅包含标识符的相关错误：

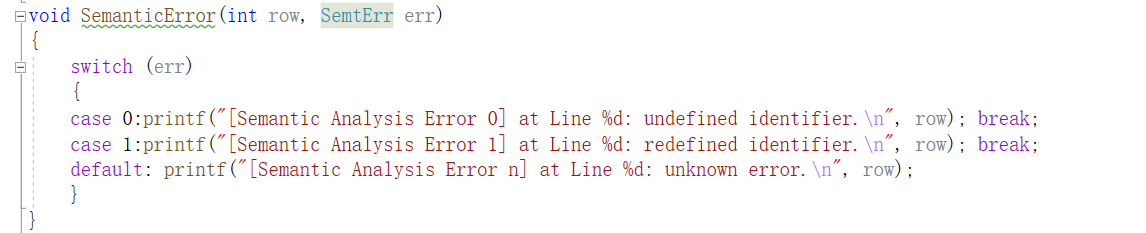
在符号表中找不到符号：标识符未定义错误

标识符重定义错误



* + 1. 函数定义与关键代码

|  |  |
| --- | --- |
| 函数原型 | 函数说明 |
| void SemanticError(int row, SemtErr err) | 语义分析出错处理 |

 代码如下：

1. 符号表管理模块
   1. 模块定义

**输入**：标识符

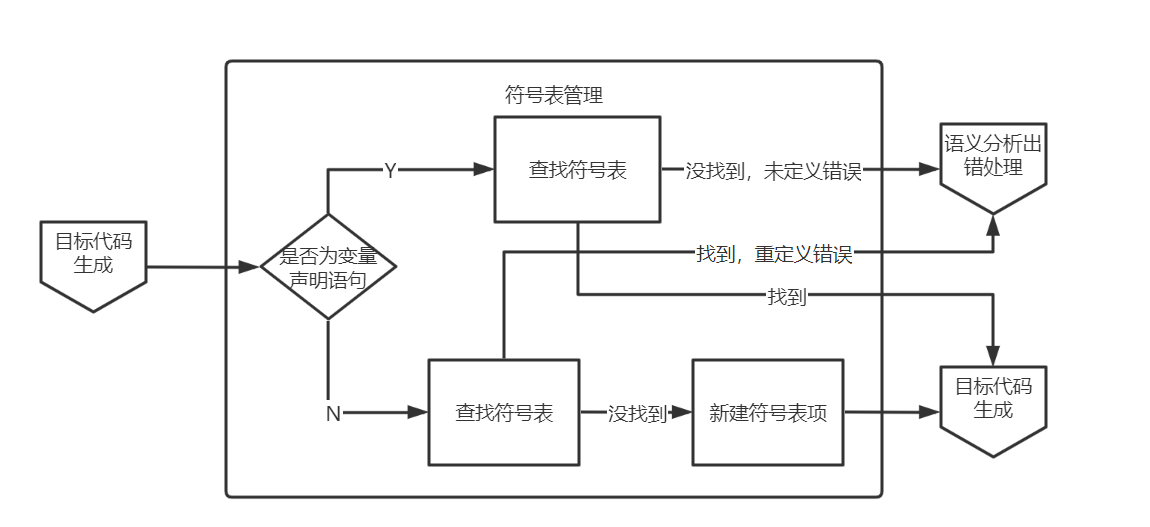
**输出**：符号表管理信息

功能：

为新声明的标识符新建一个符号表项，判断标识符是否重定义，如果重定义，转到语义分析出错处理模块报错。

在标识符使用时判断是否已声明，如果未声明转到语义分析出错处理模块报错。

流程图如下所示：



* 1. 模块实现
     1. 重要的数据类型，变量及常量

在WHI文法中，标识符的种类只有变量名一种，所以只需要一个变量符号表即可满足需求。符号表是一个结构体数组，数组的元素为符号表现结构体，记录了符号名，在符号表中的位置（地址）和符号值，具体为：

****

符号表定义如下，并将符号数目置０



* + 1. 函数定义与关键代码

对符号表中的操作有两种：一是为新标识符分配符号表的空间；二是在符号表中查找符号是否存在。

函数定义如下：

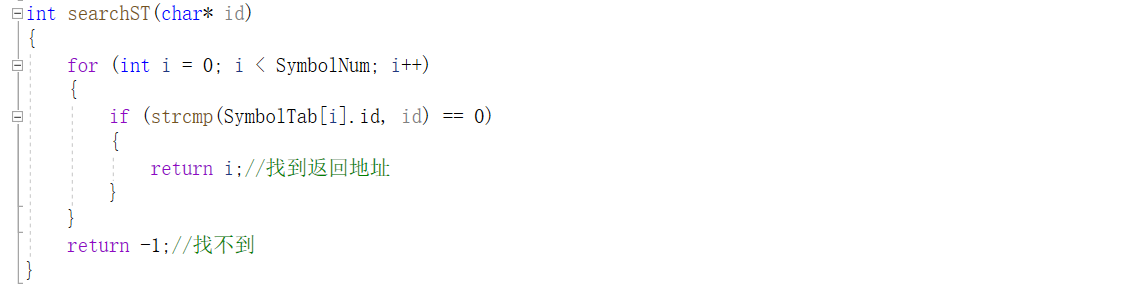
|  |  |
| --- | --- |
| 函数原型 | 函数说明 |
| void mallocVar(char\* id, int a, int v) | 在符号表中新加入一个变量 |
| int searchST(char\* id) | 搜索符号表 |

函数代码如下：

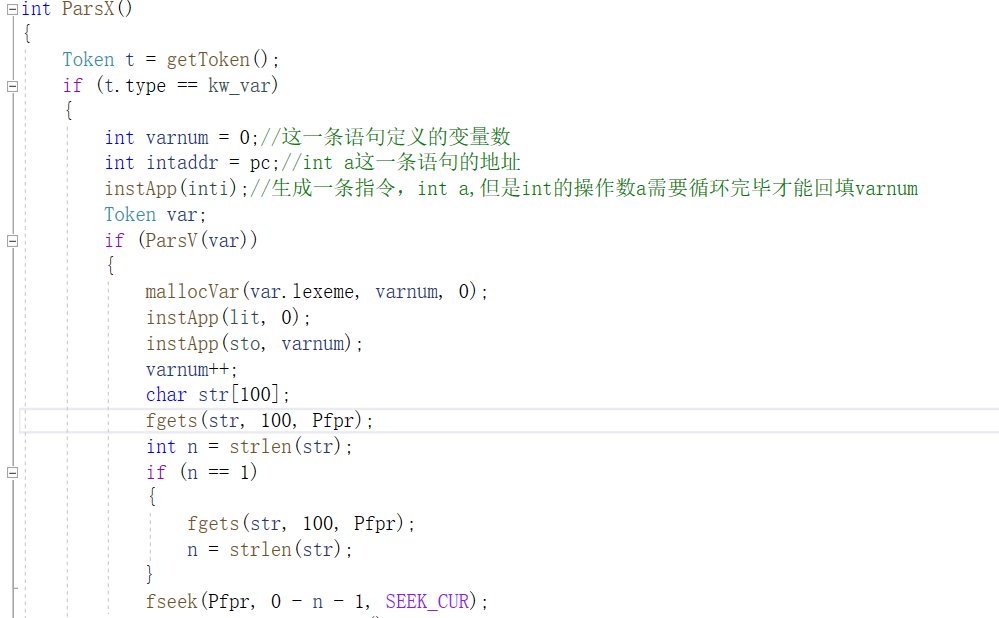
分配空间函数：在符号表中建立一个新表项，记录符号的相关信息。



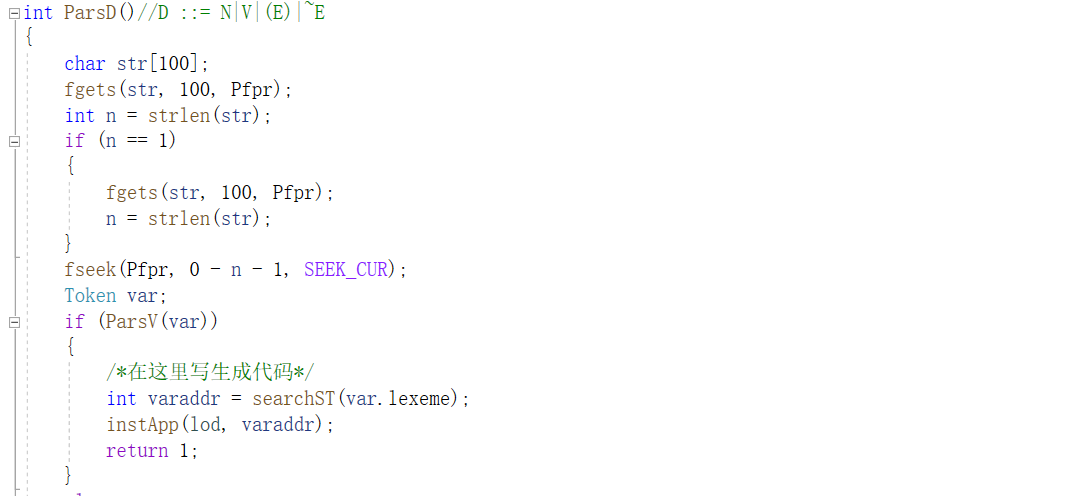
查找函数：在符号表中顺序查找所查询的符号是否存在。



* + 1. 模块使用示例



在变量定义语句中，为新定义的符号分配符号表空间



在赋值语句中，要先查找变量是否已定义，才能生成赋值代码。

* 1. 函数原型和说明

1. WHI语言编译器WHI\_Compiler系统测试
2. 系统测试说明：
   1. 源程序：源程序文件为WHI语言编写的程序文件，存放在”Programme”文件加内。为方便说明，在系统测试说明中源程序文件名为p.txt，具体如下图所示。



源程序文件夹Programme



测试程序文件

* 1. 输出文件：输出文件为两个文件，分别为词法分析结果文件和最终生成代码文件。其中，词法分析结果文件存放在“LexicalResult”文件夹，并约定文件名为“p\_LexResult”；最终生成代码文件存放在“”文件夹，并约定文件名为“”，具体如下图所示。



词法分析结果文件夹



词法分析结果文件



最终生成代码文件夹



最终生成代码文件

1. 程序1：鸡兔同笼问题(1a.txt)
   1. 源程序
      1. 输入：头和脚的个数
      2. 输出：鸡（cock）和兔（rabbit）的个数
      3. 源程序代码：

var head, foot, cock, rabbit, n;

n := 0;

read (head);

read (foot);

cock := 1;

while (cock < head) + (cock = head) do

rabbit := head - cock;

if (cock \* 2 + rabbit \* 4) = foot then

write(cock);

write(rabbit);

n := n+1

else

skip

fi;

cock := cock + 1

od;

if n = 0 then write(0) else skip fi

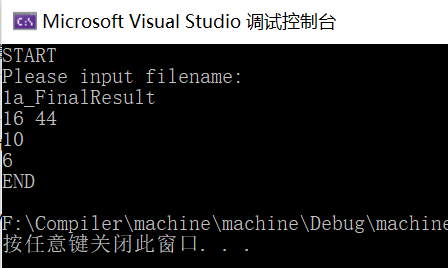
* 1. 词法分析器结果：1a\_LexResult.txt

|  |  |  |
| --- | --- | --- |
| 0 | var | 1 |
| 25 | head | 1 |
| 19 | , | 1 |
| 25 | foot | 1 |
| 19 | , | 1 |
| 25 | cock | 1 |
| 19 | , | 1 |
| 25 | rabbit | 1 |
| 19 | , | 1 |
| 25 | n | 1 |
| 20 | ; | 1 |
| 25 | n | 3 |
| 16 | := | 3 |
| 26 | 0 | 3 |
| 20 | ; | 3 |
| 2 | read | 4 |
| 21 | ( | 4 |
| 25 | head | 4 |
| 22 | ) | 4 |
| 20 | ; | 4 |
| 2 | read | 5 |
| 21 | ( | 5 |
| 25 | foot | 5 |
| 22 | ) | 5 |
| 20 | ; | 5 |
| 25 | cock | 6 |
| 16 | := | 6 |
| 26 | 1 | 6 |
| 20 | ; | 6 |
| 8 | while | 7 |
| 21 | ( | 7 |
| 25 | cock | 7 |
| 17 | < | 7 |
| 25 | head | 7 |
| 22 | ) | 7 |
| 11 | + | 7 |
| 21 | ( | 7 |
| 25 | cock | 7 |
| 18 | = | 7 |
| 25 | head | 7 |
| 22 | ) | 7 |
| 9 | do | 7 |
| 25 | rabbit | 8 |
| 16 | := | 8 |
| 25 | head | 8 |
| 12 | - | 8 |
| 25 | cock | 8 |
| 20 | ; | 8 |
| 4 | if | 9 |
| 21 | ( | 9 |
| 25 | cock | 9 |
| 13 | \* | 9 |
| 26 | 2 | 9 |
| 11 | + | 9 |
| 25 | rabbit | 9 |
| 13 | \* | 9 |
| 26 | 4 | 9 |
| 22 | ) | 9 |
| 18 | = | 9 |
| 25 | foot | 9 |
| 5 | then | 9 |
| 3 | write | 10 |
| 21 | ( | 10 |
| 25 | cock | 10 |
| 22 | ) | 10 |
| 20 | ; | 10 |
| 3 | write | 11 |
| 21 | ( | 11 |
| 25 | rabbit | 11 |
| 22 | ) | 11 |
| 20 | ; | 11 |
| 25 | n | 12 |
| 16 | := | 12 |
| 25 | n | 12 |
| 11 | + | 12 |
| 26 | 1 | 12 |
| 6 | else | 13 |
| 1 | skip | 14 |
| 7 | fi | 15 |
| 20 | ; | 15 |
| 25 | cock | 16 |
| 16 | := | 16 |
| 25 | cock | 16 |
| 11 | + | 16 |
| 26 | 1 | 16 |
| 10 | od | 17 |
| 20 | ; | 17 |
| 4 | if | 18 |
| 25 | n | 18 |
| 18 | = | 18 |
| 26 | 0 | 18 |
| 5 | then | 18 |
| 3 | write | 18 |
| 21 | ( | 18 |
| 26 | 0 | 18 |
| 22 | ) | 18 |
| 6 | else | 18 |
| 1 | skip | 18 |
| 7 | fi | 18 |

* 1. 生成代码1a\_FinalResult.txt

|  |  |
| --- | --- |
| int | 5 |
| lit | 0 |
| sto | 0 |
| lit | 0 |
| sto | 1 |
| lit | 0 |
| sto | 2 |
| lit | 0 |
| sto | 3 |
| lit | 0 |
| sto | 4 |
| lit | 0 |
| sto | 4 |
| red | 0 |
| red | 1 |
| lit | 1 |
| sto | 2 |
| lod | 2 |
| lod | 0 |
| lth |  |
| lod | 2 |
| lod | 0 |
| equ |  |
| add |  |
| jpc | 54 |
| lod | 0 |
| lod | 2 |
| sub |  |
| sto | 3 |
| lod | 2 |
| lit | 2 |
| mul |  |
| lod | 3 |
| lit | 4 |
| mul |  |
| add |  |
| lod | 1 |
| equ |  |
| jpc | 48 |
| lod | 2 |
| wrt |  |
| lod | 3 |
| wrt |  |
| lod | 4 |
| lit | 1 |
| add |  |
| sto | 4 |
| jmp | 49 |
| nop |  |
| lod | 2 |
| lit | 1 |
| add |  |
| sto | 2 |
| jmp | 17 |
| lod | 4 |
| lit | 0 |
| equ |  |
| jpc | 61 |
| lit | 0 |
| wrt |  |
| jmp | 62 |
| nop |  |

* 1. 生成代码运行



如图所示，START代表程序运行开始；1a\_FinalResult为编译好的程序作为简单栈式抽象机的输入；16 44为程序的两个输入，分别为头的个数和脚的个数；10 16为程序输出，分别为计算所得的鸡的个数和兔的个数；最后END代表程序运行结束。

1. 程序2 :最大公因数和最小公倍数问题（2b.txt）
   1. 源程序
      1. 输入：两个非负整数
      2. 输出：这两个数的最大公因数和最小公倍数
      3. 源程序代码：

var x0, y0, x, y, g, m, temp;

read(x);

read(y);

x0 := x;

y0 := y;

x := x-x/y\*y;

while ~(x = 0) do

temp := x;

x := y;

y := temp;

x := x-x/y\*y

od;

g := y;

m := x0\*y0/g;

write(g);

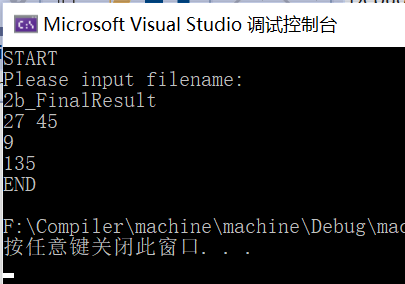
write(m)

* 1. 词法分析器结果：2b\_LexResult.txt

|  |  |  |
| --- | --- | --- |
| 0 | var | 1 |
| 25 | x0 | 1 |
| 19 | , | 1 |
| 25 | y0 | 1 |
| 19 | , | 1 |
| 25 | x | 1 |
| 19 | , | 1 |
| 25 | y | 1 |
| 19 | , | 1 |
| 25 | g | 1 |
| 19 | , | 1 |
| 25 | m | 1 |
| 19 | , | 1 |
| 25 | temp | 1 |
| 20 | ; | 1 |
| 2 | read | 3 |
| 21 | ( | 3 |
| 25 | x | 3 |
| 22 | ) | 3 |
| 20 | ; | 3 |
| 2 | read | 4 |
| 21 | ( | 4 |
| 25 | y | 4 |
| 22 | ) | 4 |
| 20 | ; | 4 |
| 25 | x0 | 5 |
| 16 | := | 5 |
| 25 | x | 5 |
| 20 | ; | 5 |
| 25 | y0 | 6 |
| 16 | := | 6 |
| 25 | y | 6 |
| 20 | ; | 6 |
| 25 | x | 7 |
| 16 | := | 7 |
| 25 | x | 7 |
| 12 | - | 7 |
| 25 | x | 7 |
| 14 | / | 7 |
| 25 | y | 7 |
| 13 | \* | 7 |
| 25 | y | 7 |
| 20 | ; | 7 |
| 8 | while | 9 |
| 15 | ~ | 9 |
| 21 | ( | 9 |
| 25 | x | 9 |
| 18 | = | 9 |
| 26 | 0 | 9 |
| 22 | ) | 9 |
| 9 | do | 9 |
| 25 | temp | 10 |
| 16 | := | 10 |
| 25 | x | 10 |
| 20 | ; | 10 |
| 25 | x | 11 |
| 16 | := | 11 |
| 25 | y | 11 |
| 20 | ; | 11 |
| 25 | y | 12 |
| 16 | := | 12 |
| 25 | temp | 12 |
| 20 | ; | 12 |
| 25 | x | 13 |
| 16 | := | 13 |
| 25 | x | 13 |
| 12 | - | 13 |
| 25 | x | 13 |
| 14 | / | 13 |
| 25 | y | 13 |
| 13 | \* | 13 |
| 25 | y | 13 |
| 10 | od | 14 |
| 20 | ; | 14 |
| 25 | g | 15 |
| 16 | := | 15 |
| 25 | y | 15 |
| 20 | ; | 15 |
| 25 | m | 16 |
| 16 | := | 16 |
| 25 | x0 | 16 |
| 13 | \* | 16 |
| 25 | y0 | 16 |
| 14 | / | 16 |
| 25 | g | 16 |
| 20 | ; | 16 |
| 3 | write | 17 |
| 21 | ( | 17 |
| 25 | g | 17 |
| 22 | ) | 17 |
| 20 | ; | 17 |
| 3 | write | 18 |
| 21 | ( | 18 |
| 25 | m | 18 |
| 22 | ) | 18 |

* 1. 生成代码2b\_FinalResult.txt

|  |  |
| --- | --- |
| int | 7 |
| lit | 0 |
| sto | 0 |
| lit | 0 |
| sto | 1 |
| lit | 0 |
| sto | 2 |
| lit | 0 |
| sto | 3 |
| lit | 0 |
| sto | 4 |
| lit | 0 |
| sto | 5 |
| lit | 0 |
| sto | 6 |
| red | 2 |
| red | 3 |
| lod | 2 |
| sto | 0 |
| lod | 3 |
| sto | 1 |
| lod | 2 |
| lod | 2 |
| lod | 3 |
| div |  |
| lod | 3 |
| mul |  |
| sub |  |
| sto | 2 |
| lod | 2 |
| lit | 0 |
| equ |  |
| lit | 0 |
| equ |  |
| jpc | 50 |
| lod | 2 |
| sto | 6 |
| lod | 3 |
| sto | 2 |
| lod | 6 |
| sto | 3 |
| lod | 2 |
| lod | 2 |
| lod | 3 |
| div |  |
| lod | 3 |
| mul |  |
| sub |  |
| sto | 2 |
| jmp | 29 |
| lod | 3 |
| sto | 4 |
| lod | 0 |
| lod | 1 |
| mul |  |
| lod | 4 |
| div |  |
| sto | 5 |
| lod | 4 |
| wrt |  |
| lod | 5 |
| wrt |  |

* 1. 生成代码运行结果

如图所示，START代表程序运行开始；2b\_FinalResult为编译好的程序作为简单栈式抽象机的输入；27 45为程序的两个输入，为两个非负整数；9 135为程序输出，分别为计算所得的最大公因数和最小公倍数；最后END代表程序运行结束。

1. 程序3：
   1. 源程序：判断质数（3a.txt）
      1. 输入：一个非负整数
      2. 输出：如果这个数是质数，输出1，否则输出0
      3. 源程序代码：

var m, n, i, f;

read(m);

if (m < 1) + (m = 1) then

f := 0

else

f := 1;

if m = 2 then

skip

else

i := 2;

while i < m do

if (m - m/i\*i) = 0 then

f := 0

else

skip

fi;

i := i+1

od

fi

fi;

write(f)

* 1. 词法分析器结果：3a\_LexResult.txt生成代码

|  |  |  |
| --- | --- | --- |
| 0 | var | 1 |
| 25 | m | 1 |
| 19 | , | 1 |
| 25 | n | 1 |
| 19 | , | 1 |
| 25 | i | 1 |
| 19 | , | 1 |
| 25 | f | 1 |
| 20 | ; | 1 |
| 2 | read | 3 |
| 21 | ( | 3 |
| 25 | m | 3 |
| 22 | ) | 3 |
| 20 | ; | 3 |
| 4 | if | 4 |
| 21 | ( | 4 |
| 25 | m | 4 |
| 17 | < | 4 |
| 26 | 1 | 4 |
| 22 | ) | 4 |
| 11 | + | 4 |
| 21 | ( | 4 |
| 25 | m | 4 |
| 18 | = | 4 |
| 26 | 1 | 4 |
| 22 | ) | 4 |
| 5 | then | 4 |
| 25 | f | 5 |
| 16 | := | 5 |
| 26 | 0 | 5 |
| 6 | else | 6 |
| 25 | f | 7 |
| 16 | := | 7 |
| 26 | 1 | 7 |
| 20 | ; | 7 |
| 4 | if | 8 |
| 25 | m | 8 |
| 18 | = | 8 |
| 26 | 2 | 8 |
| 5 | then | 8 |
| 1 | skip | 9 |
| 6 | else | 10 |
| 25 | i | 11 |
| 16 | := | 11 |
| 26 | 2 | 11 |
| 20 | ; | 11 |
| 8 | while | 12 |
| 25 | i | 12 |
| 17 | < | 12 |
| 25 | m | 12 |
| 9 | do | 12 |
| 4 | if | 13 |
| 21 | ( | 13 |
| 25 | m | 13 |
| 12 | - | 13 |
| 25 | m | 13 |
| 14 | / | 13 |
| 25 | i | 13 |
| 13 | \* | 13 |
| 25 | i | 13 |
| 22 | ) | 13 |
| 18 | = | 13 |
| 26 | 0 | 13 |
| 5 | then | 13 |
| 25 | f | 14 |
| 16 | := | 14 |
| 26 | 0 | 14 |
| 6 | else | 15 |
| 1 | skip | 16 |
| 7 | fi | 17 |
| 20 | ; | 17 |
| 25 | i | 18 |
| 16 | := | 18 |
| 25 | i | 18 |
| 11 | + | 18 |
| 26 | 1 | 18 |
| 10 | od | 19 |
| 7 | fi | 20 |
| 7 | fi | 21 |
| 20 | ; | 21 |
| 3 | write | 22 |
| 21 | ( | 22 |
| 25 | f | 22 |
| 22 | ) | 22 |

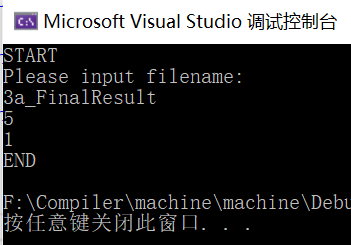
* 1. 生成代码. 3a\_FinalResult txt

|  |  |
| --- | --- |
| int | 4 |
| lit | 0 |
| sto | 0 |
| lit | 0 |
| sto | 1 |
| lit | 0 |
| sto | 2 |
| lit | 0 |
| sto | 3 |
| red | 0 |
| lod | 0 |
| lit | 1 |
| lth |  |
| lod | 0 |
| lit | 1 |
| equ |  |
| add |  |
| jpc | 21 |
| lit | 0 |
| sto | 3 |
| jmp | 54 |
| lit | 1 |
| sto | 3 |
| lod | 0 |
| lit | 2 |
| equ |  |
| jpc | 29 |
| nop |  |
| jmp | 54 |
| lit | 2 |
| sto | 2 |
| lod | 2 |
| lod | 0 |
| lth |  |
| jpc | 54 |
| lod | 0 |
| lod | 0 |
| lod | 2 |
| div |  |
| lod | 2 |
| mul |  |
| sub |  |
| lit | 0 |
| equ |  |
| jpc | 48 |
| lit | 0 |
| sto | 3 |
| jmp | 49 |
| nop |  |
| lod | 2 |
| lit | 1 |
| add |  |
| sto | 2 |
| jmp | 31 |
| lod | 3 |
| wrt |  |

* 1. 生成代码运行结果



如图所示，START代表程序运行开始；3a\_FinalResult为编译好的程序作为简单栈式抽象机的输入；8为程序的输入，为一个非负整数；0为程序输出，代表8不是一个质数；最后END代表程序运行结束。



如图所示，START代表程序运行开始；3a\_FinalResult为编译好的程序作为简单栈式抽象机的输入；5为程序的输入，为一个非负整数；1为程序输出，代表5是一个质数；最后END代表程序运行结束。