《编译技术》

周尔强

第二章 实现一个简单编译器

实现一个简单编译器

- 词法分析
- 语法分析
- 语义分析
- 中间代码生成
- 中间代码优化
- 目标代码生成
- 目标代码优化

```
a = 1;
b = 10;
while (a < b)
    a = a + 1;
b = a + b;
```

```
100:  a = 1
101:  b = 10
102:  if a < b goto 104
103:  goto 107
104:  t1 = a + 1
105:  a = t1
106:  goto 102
107:  t2 = a + b
108:  b = t2
```

语言定义

代码考查

“C语言”代码片段

表达式种类?

语句种类?

```
a = 1;
b = 10;
while (a < b)
    a = a + 1;
b = a + b;
```

赋值表达式构成

变量 赋值号 表达式（整数/求和运算）

布尔表达式构成

变量 比较符号 变量

语言定义

赋值语句构成

赋值表达式 分号

while语句构成

while（ 布尔表达式 ） 语句

进一步考查表达式

```
a = 1;
b = 10;
while (a < b)
    a = a + 1;
b = a + b;
```

标识符

关键字？ 函数名？ 结构名？

变量：以字母开头，后由字母、数字下划线构成的字符串

常量：整数 浮点数？ 字符串常量？

运算：运算符 变量 常量 函数调用？ 数组元素？

语言定义

语言定义总结

语言 = 语法 + 语义

语法规则

较高级(抽象)的语法成份

语法规则

while语句→ while(布尔表达式)语句

较低级(具体)的语法成份

降格为词法规则

标识符→ [_a-zA-Z]+[_a-zA-Z0-9]*

标识符、关键字、符号、常量等

语言定义

词法分析器

处理词法规则，即降格的语法规则

需识别：标识符、关键字、符号、常量等

对于语法规则，这些符号已终结，不可再分

统称为“终结符”

非终结符：语法规则中可再分的语法成分

如：“语句”、“表达式”等

语言定义

```
a = 1;
b = 10;
while (a < b)
    a = a + 1;
b = a + b;
```

语法定义

关于 while 语句

while语句 → while (布尔表达式) 语句

while_stmt → while (bool_expr) stmt

WHILE_STMT → while (BOOL_EXPR) STMT

或

while_stmt → WHILE (bool_expr) stmt

语言定义

```
a = 1;
b = 10;
while (a < b)
    a = a + 1;
b = a + b;
```

赋值语句

赋值语句 → 标识符 = 表达式 ;

assign_stmt → identifier = expr ;

assign_stmt → ID = expr ;

布尔表达式

bool_expr → expr > expr
| expr < expr
| expr == expr

语言定义

表达式

```
primary_expr  → ID | NUMBER
expr  → primary_expr
      | primary_expr + expr
      | primary_expr - expr
```

程序及语句

```
program  →  stmt | program stmt
   stmt  →  while_stmt | assign_stmt
```

```
a = 1;
b = 10;
while (a < b)
    a = a + 1;
b = a + b;
```

语言定义

```
program   →   stmt | program stmt
stmt   →   while_stmt | assign_stmt
while_stmt → WHILE ( bool_expr ) stmt
assign_stmt   → ID = expr ;
bool_expr   → expr > expr
            |   expr < expr
            |   expr == expr
expr → primary_expr
     |   primary_expr + expr | primary_expr - expr
primary_expr   →  ID   |   NUMBER
```

QUESTIONS?

词法分析

词法分析器

功能：识别“终结符”

输入：字符串形式的源代码

输出：带有标记的逻辑分组（源程序的子串）

单词/子串类别 及 属性

实现方法

状态转移图法（手动实现/表驱动）

正则表达式法（第三章）

词法分析

词法分析器的本质

根据当前的输入字符，跳转到相应的分组/类别

状态转移图法

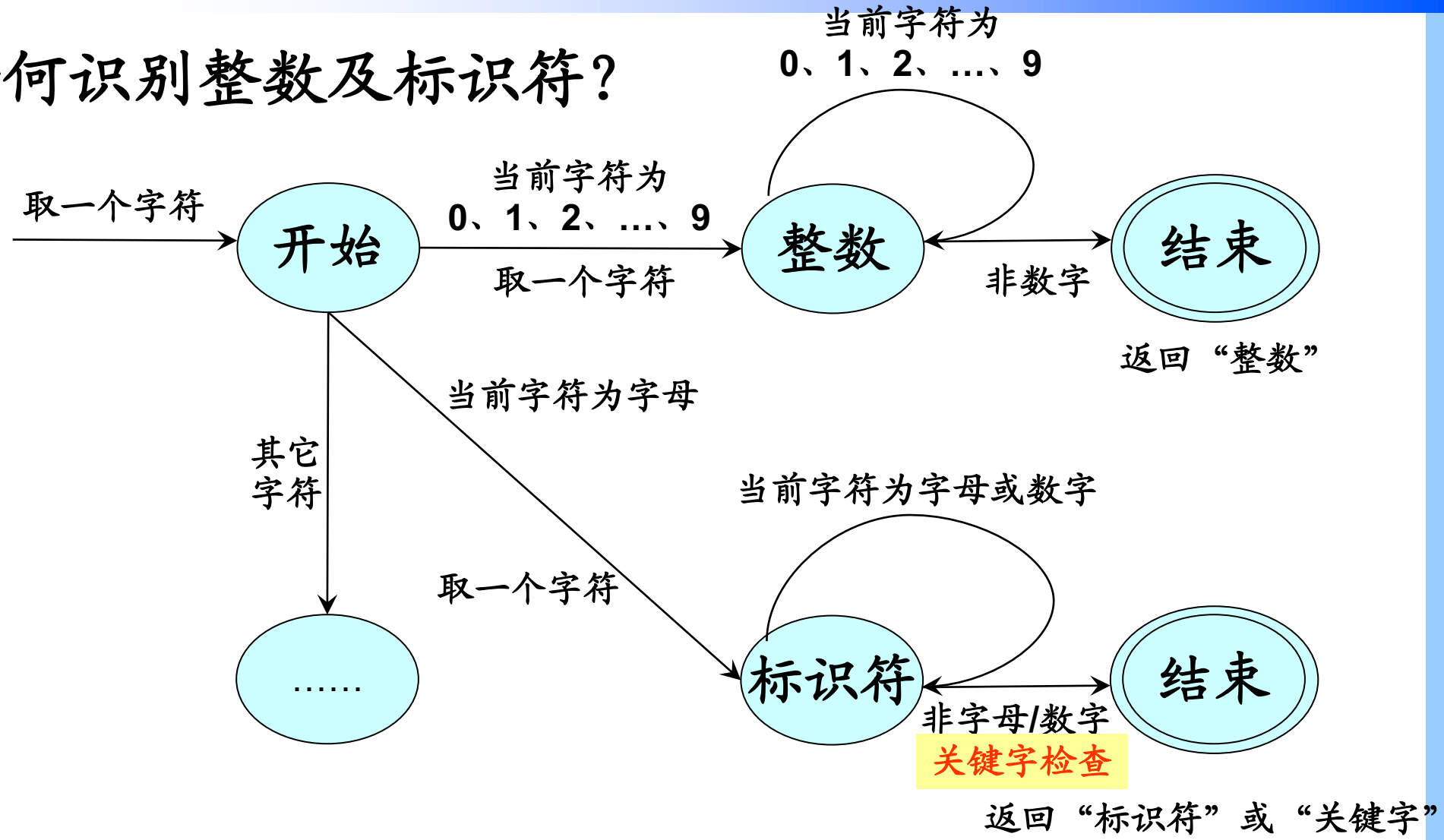
将该过程图形化

根据当前字符的跳转 ==> 图中的“边”

跳转到的类别 ==> 图中的“结点”

词法分析

如何识别整数及标识符？



词法分析

词法分析器所需函数

- 取一个字符 getChar
- 将当前字符加到当前已识别子串 addChar
- 跳过多余空格/注释 getNonBlank
- 关键字检查 checkKeywords
- 符号检查 checkSymbol

词法分析的实现

- 参教材 图2-3及相关代码

语法分析

语法分析的目的

确定输入程序在语法是否正确

生成完整的语法分析树，供后续分析使用

语法分析方法

已定义语法规则

已有符合该规则的源代码

语法规则 和 源程序 如何建立联系？

语法分析

语法规则 和 源程序 如何建立联系？

源程序是否符合语法规则？

由语法规则出发

检查语法规则是否能推导出源程序

由源程序出发

检查源程序是否符合语法规则

语言定义

```
stmt   →   while_stmt | assign_stmt
while_stmt → WHILE ( bool_expr ) stmt
assign_stmt   → ID = expr ;
bool_expr   → expr > expr
            |   expr < expr
            |   expr == expr
expr → primary_expr
     |   primary_expr + expr | primary_expr - expr
primary_expr   →  ID   |   NUMBER
```

```
stmt  →  while_stmt | assign_stmt
while_stmt → WHILE ( bool_expr ) stmt
assign_stmt  → ID = expr ;
primary_expr  → ID  |  NUMBER
```

```
bool_expr  → expr > expr
           |  expr < expr
           |  expr == expr
expr → primary_expr
     |  primary_expr + expr
     |  primary_expr - expr
```

从语法规则出发检验while语句

while_stmt ⇒ WHILE (bool_expr) stmt

⇒ WHILE (expr < expr) stmt

⇒ WHILE (primary_expr < expr) stmt

⇒ WHILE (ID < expr) stmt

⇒ WHILE (ID < primary_expr) stmt

⇒ WHILE (ID < ID) stmt

⇒ WHILE (ID < ID) assign_stmt

⇒ WHILE (ID < ID) ID = expr ;

⇒ WHILE (ID < ID) ID = primary_expr + expr ;

```
while (a < b)
    a = a + 1;
```

```
stmt  →  while_stmt | assign_stmt
while_stmt → WHILE ( bool_expr ) stmt
assign_stmt  → ID = expr ;
primary_expr  → ID  |  NUMBER
```

```
bool_expr  → expr > expr
           |  expr < expr
           |  expr == expr
expr → primary_expr
     |  primary_expr + expr
     |  primary_expr - expr
```

从语法规则出发检验while语句

while_stmt ⇒ WHILE (bool_expr) stmt

⇒ WHILE (expr < expr) stmt

⇒ ……

⇒ WHILE (ID < ID) ID = primary_expr + expr ;

⇒ WHILE (ID < ID) ID = ID + expr ;

⇒ WHILE (ID < ID) ID = ID + primary_expr ;

⇒ WHILE (ID < ID) ID = ID + NUMBER ;

while (a < b) a = a + 1 ;

```
while (a < b)
    a = a + 1;
```

QUESTIONS?

```
stmt  →  while_stmt | assign_stmt
while_stmt → WHILE ( bool_expr ) stmt
assign_stmt  → ID = expr ;
primary_expr  → ID  |  NUMBER
```

```
bool_expr  → expr > expr
           |  expr < expr
           |  expr == expr
expr → primary_expr
     |  primary_expr + expr
     |  primary_expr - expr
```

从源程序出发检验while语句

```
while (a < b)
    a = a + 1;
```

	WHILE (ID < ID) ID = ID + NUMBER ;
WHILE	(ID < ID) ID = ID + NUMBER ;
WHILE (	ID < ID) ID = ID + NUMBER ;
WHILE (ID	< ID) ID = ID + NUMBER ;
WHILE (primary_expr	< ID) ID = ID + NUMBER ;
WHILE (expr	< ID) ID = ID + NUMBER ;
WHILE (expr <	ID) ID = ID + NUMBER ;
WHILE (expr < ID	) ID = ID + NUMBER ;
WHILE (expr < primary_expr	) ID = ID + NUMBER ;
WHILE (expr < expr	) ID = ID + NUMBER ;
WHILE (bool_expr	) ID = ID + NUMBER ;
WHILE (bool_expr)	ID = ID + NUMBER ;

```
stmt  →  while_stmt | assign_stmt
while_stmt → WHILE ( bool_expr ) stmt
assign_stmt  → ID = expr ;
primary_expr  → ID  |  NUMBER
```

```
bool_expr  → expr > expr
           |  expr < expr
           |  expr == expr
expr → primary_expr
     |  primary_expr + expr
     |  primary_expr - expr
```

从源程序出发检验while语句

```
while (a < b)
    a = a + 1;
```

	WHILE (ID < ID) ID = ID + NUMBER ;
.....	.....
WHILE (bool_expr)	ID = ID + NUMBER ;
WHILE (bool_expr) ID	= ID + NUMBER ;
WHILE (bool_expr) ID =	ID + NUMBER ;
WHILE (bool_expr) ID = ID	+ NUMBER ;
WHILE (bool_expr) ID = primary_expr	+ NUMBER ;
WHILE (bool_expr) ID = primary_expr +	NUMBER ;
WHILE (bool_expr) ID = primary_expr + NUMBER	;
WHILE (bool_expr) ID = primary_expr + primary_expr	;
WHILE (bool_expr) ID = primary_expr + expr	;
WHILE (bool_expr) ID = expr	;

```
stmt  →  while_stmt | assign_stmt
while_stmt → WHILE ( bool_expr ) stmt
assign_stmt  → ID = expr ;
primary_expr  → ID  |  NUMBER
```

```
bool_expr  → expr > expr
           |  expr < expr
           |  expr == expr
expr → primary_expr
     |  primary_expr + expr
     |  primary_expr - expr
```

从源程序出发检验while语句

```
while (a < b)
    a = a + 1;
```

	WHILE (ID < ID) ID = ID + NUMBER ;
.....	.....
WHILE (bool_expr) ID = expr	;
WHILE (bool_expr) ID = expr ;	
WHILE (bool_expr) assign_stmt	
WHILE (bool_expr) stmt	
while_stmt	

while_stmt → WHILE (bool_expr) stmt

语法分析

递归下降分析法

每个 非终结符 对应一个解析函数

语法规则右侧为其左侧非终结符所对应的“函数体”

关于“函数体”

右侧终结符　　对应　从输入串中消耗该终结符

右侧非终结符　对应　函数调用

语法上规则中的‘|’

对应“if-else”语句

语法分析

根据右侧定义函数体

当遇到终结符时，编写语句

if（当前符号 与 该终结符 匹配）

读入下一个字符

源程序中的符号 与 规则中的符号

当遇到非终结符时

编写调用该非终结符对应的函数

语法分析

赋值语句规则: assign_stmt → ID = expr ;

左侧非终结符 assign_stmt 定义函数

```
void analyse_assign_stmt()
```

右侧开始为终结符ID，则编写

```
if ( match(ID) )
    advance();
```

ID之后是终结符 =，编写

```
if ( match('=') )
    advance();
```

match函数检查
当前符号 与 参数 是否匹配

advance函数 读入下一个单词

语法分析

赋值语句规则: assign_stmt → ID = expr ;

= 之后是非终结符 expr，编写

```
    analyse_expr(); //表达式分析函数
```

最后是终结符 ; 编写

```
    if ( match(';') )
        advance();
```

语法分析

赋值语句规则: assign_stmt → ID = expr ;

```
int analyse_assign_stmt()
{
    if ( ! match( ID ) ) { //匹配标识符
        printf("ERROR: Expect an ID.\n");
        return 0;
    }
    advance(); //将下一个待分析字符设为当前分析字符
    if ( ! match( '=' ) ) { //匹配赋值符号
        printf("ERROR: Expect '=' .\n");
        return 0;
    }
    advance(); //将下一个待分析字符设为当前分析字符
    analyse_expr(); //匹配表达式
    if ( ! match( ';' ) ) { //匹配分号
        printf("ERROR: Expect ';' .\n");
        return 0;
    }
    advance();
    return 1;
}
```

语法分析

```
program   →   stmt | program stmt
stmt   →   while_stmt | assign_stmt
while_stmt → WHILE ( bool_expr ) stmt
assign_stmt   → ID = expr ;
bool_expr   → expr > expr
            |   expr < expr
            |   expr == expr
expr → primary_expr + expr | primary_expr - expr
primary_expr   →  ID   |   NUMBER
```

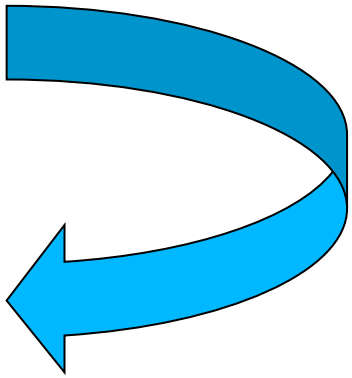
以非终结符开始的规则
选哪一个规则？如何编写？

语法分析

program → stmt | program stmt

按前述方法

```
void analyse_program()
{
    analyse_program();
    analyse_stmt();
}
```



递归调用，死循环！

如何改进？

语法分析

program → stmt | program stmt

该规则的含义

program表示程序，stmt表示语句

一条语句可组成一个程序

一个程序之后再跟一条语句仍然是程序

程序是由一个语句序列组成

规则改写　大括号表示其中的语法成份可以重复0次或多次

program → stmt { stmt }

语法分析

program → stmt { stmt }

改进后的方法

大括号之前的符号：编写调用该符号对应的函数

大括号内的重复部分：用while循环处理

```
void analyse_program() {
  analyse_stmt(); //对语句进行分析
  while( nex      循环条件？
    analyse_stmt();
  }
}
```

是不是所有规则都这样处理？

思考：

如何改进？

利用分析函数的返回值？

语法分析

stmt → while_stmt | assign_stmt

语句有两种形式

要么是 *while_stmt*

要么是 *assign_stmt*

不可能有其它，

因此可以猜测是哪一种！

如何使程序没有“回溯”？

```
int analyse_stmt() {
    if ( analyse_while_stmt() )
        return 1;
    else if ( analyse_assign_stmt() )
        return 1;
    else
        error("....."); return 0;
}
```

语法分析

stmt → while_stmt | assign_stmt

while_stmt → WHILE (bool_expr) stmt

assign_stmt → ID = expr ;

```
void analyse_stmt() {
    if ( match( xxx WHILE
            analyse_while_stmt();
    }else if ( match( xxxxx  ID
            analyse_assign_stmt();
    }else
            error(".....");
}
```

while语句只接受
以 “WHILE”开始的单词序列

赋值语句只接受
以 “ID”开始的单词序列

语法分析

```
program   →   stmt | program stmt
stmt   →   while_stmt | assign_stmt
while_stmt → WHILE ( bool_expr ) stmt
assign_stmt   → ID = expr ;
bool_expr   → expr > expr
            |   expr < expr
            |   expr == expr
expr → primary_expr + expr | primary_expr - expr
primary_expr   → ID   |   NUMBER
```

方法类似，参考教材

递归下降分析法

例：对下列文法构造分析程序

$$E \rightarrow TE_1$$

$$E_1 \rightarrow +\ TE_1 \mid \varepsilon$$

$$T \rightarrow FT_1$$

$$T_1 \rightarrow *\ F\ T_1 \mid \varepsilon$$

$$F \rightarrow (\ E\ ) \mid i$$

递归下降分析法

$E \rightarrow TE_1$
$E_1 \rightarrow + TE_1 \mid \varepsilon$
$T \rightarrow FT_1$
$T_1 \rightarrow * F T_1 \mid \varepsilon$
$F \rightarrow (E) \mid i$

```
int E( )
{
   return T() &&E1();
}

int T( )
{
    return F() && T1();
}
```

```
int E1( )
{
   if( match( '+') )
   {
       advance();
       return T() &&E1();
   }
   return 1;
}
```

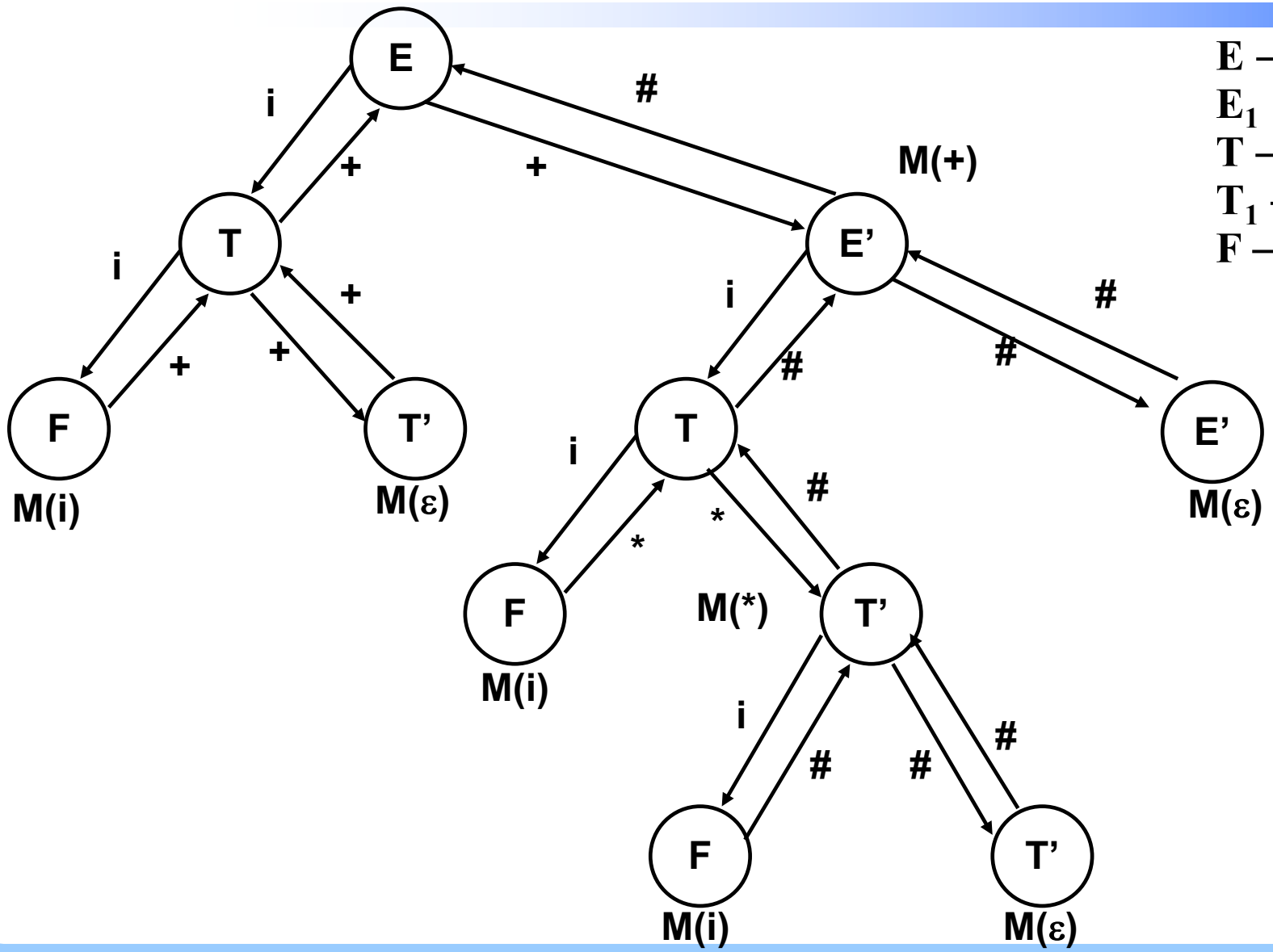
递归下降分析法

$E \rightarrow TE_1$
$E_1 \rightarrow +\ TE_1 \mid \varepsilon$
$T \rightarrow FT_1$
$T_1 \rightarrow *\ F\ T_1 \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid i$

```
int T1()
{
   if( match('*' ) )
   {
      advance();
      return F() && T1();
   }
}
```

```
int F() {
    if ( match( '(' ) )
    {
       advance();
       if (E() && match( ')' ))
       {   advance();   return 1;  }
       else return 0;
    }
    else if ( match( 'i'  ))
      {   advance();   return 1;  }
    else return 0;
}
```

串 i+i*i# 递归下降分析过程



$E \rightarrow TE_1$
$E_1 \rightarrow +TE_1 \mid \varepsilon$
$T \rightarrow FT_1$
$T_1 \rightarrow *FT_1 \mid \varepsilon$
$F \rightarrow (E) \mid i$

建立抽象语法树

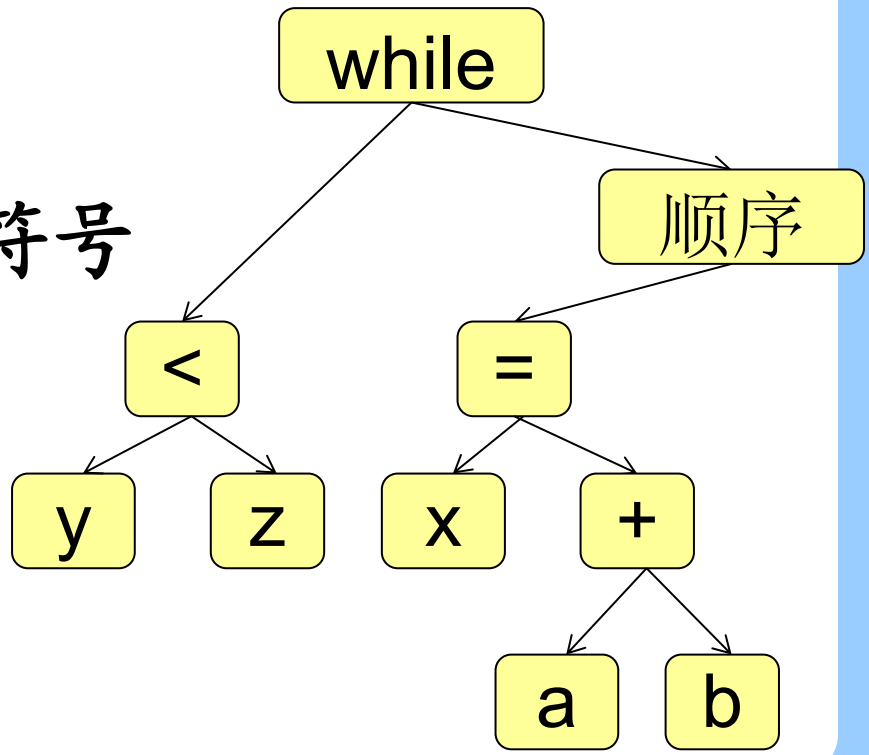
抽象语法树

是源代码的抽象语法结构的树状表现

不保留仅语法用的符号

与具体语法树相对应

包括语法规则中的各种符号



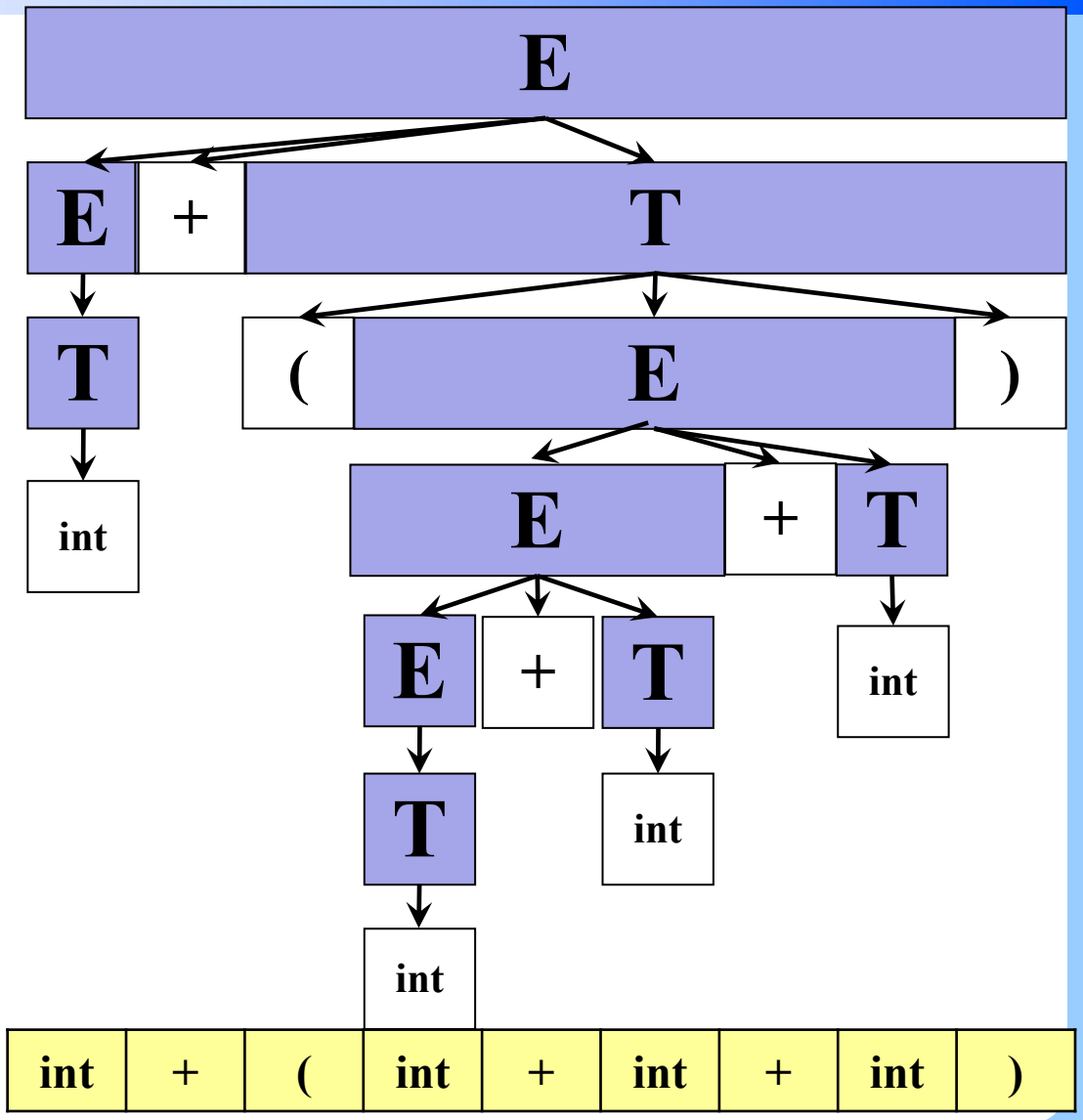
建立抽象语法树

$E \rightarrow T$

$E \rightarrow E + T$

$T \rightarrow int$

$T \rightarrow (E)$



建立抽象语法树

抽象语法树 结点类型 设计

```
typedef struct _ast ast;
typedef struct _ast *past;
struct _ast{
  int ivalue;
  char* svalue;
  char* nodeType;
  past left;
  past right;
};
```

结点类型

结点指针类型

保存整数信息

保存字符串信息

结点类型

指向左子树

指向右子树

建立抽象语法树

与 结点 相关函数设计

past newAstNode()　创建抽象语法树结点

past newNum(int value)　创建数值结点

结点中的ivalue被设置为value

past newVarRef(char* name)　创建标识符结点

结点中的svalue被设置为name

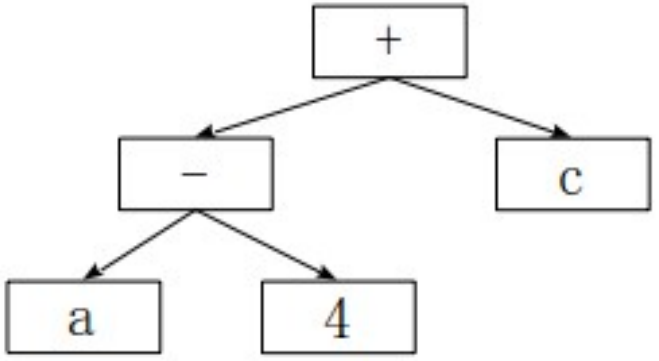
past newExpr(int oper, past left, past right)

创建一个表达式结点:ivalue表示运算符

left、right分别指向表达式的两个操作数

建立抽象语法树

建立了表达式a-4+c的抽象语法树




```
p1 = newVarRef( pa );
p2 = newNum( 4 );
p3 = newExpr( ‘-’, p1, p2);
p4 = newVarRef( pc );
p5 = newExpr( ‘+’, p3, p4);
```

pa是指向标识符 ‘a’ 的指针

pc是指向标识符 ‘c’ 的指针

a或c是什么类型？
局部变量？全局变量？

建立语法树时需要填表

语义分析

符号表的作用

登记符号属性值

查找符号的属性、检查其合法性

作为目标代码生成阶段地址分配的依据

符号（标识符）有哪些属性

变量：变量名、类型、值

函数：函数名、参数个数、参数类型，返回类型

“数据类型”对程序设计语言至关重要

语义分析 之 数据类型

“int a;” 意味着什么？

数据类型

是“数据”的抽象

定义了一组**值**和一组**操作**

为什么需要“数据类型”

实现“数据抽象”

不用关心存储单元中二进制位串的意义

C语言实现“分数”运算？

假如语言只支持位运算

“整数”运算如何进行？

语义分析 之 基本类型

基本数据类型

不用其它类型来定义的数据类型

int, float, char, string, bool

整数

支持不同长度的整数类型

java中4种有符号整数

byte, short, int, long

大多数计算机中

负整数是如何表示的？

二进制补码，为什么？

0的二进制反码有几种？

Java中的byte与C的char有什么区别？

正零和负零的原码为:

+0 : 0000 0000 0000 0000 0000 0000 0000 0000

−0 : 1000 0000 0000 0000 0000 0000 0000 0000

而反码为:

+0 : 0000 0000 0000 0000 0000 0000 0000 0000

−0 : 1111 1111 1111 1111 1111 1111 1111 1111

补码为:

+0 : 0000 0000 0000 0000 0000 0000 0000 0000

−0 : 1 0000 0000 0000 0000 0000 0000 0000 0000

-0的补码发生溢出，舍弃最高位:

0000 0000 0000 0000 0000 0000 0000 0000

语义分析 之 基本类型

浮点数

实数的模拟，保持“近似值”

语言支持

$$1.2345 = \underbrace{12345}_{\text{significand}} \times \underbrace{10}_{\text{base}}^{\overbrace{-4}^{\text{exponent}}}$$

两种浮点类型：float 和 double

float：四个字节

double：八个字节

表示为小数和指数

大多数计算机中

浮点数是如何表示的？

IEEE 浮点标准754格式

语义分析 之 基本类型

布尔类型

只有“真”和“假”两个值

C，Java，C#，C++ 中的布尔类型？

数值表达式是否可以当作布尔类型？

布尔类型如何表示？（占几位？）

C99，C++允许

Java，C#不允许

通常为一个字节。为什么？

因为很多计算机

不能有效访问单个二进制位

语义分析 之 基本类型

总结：内部类型有以下优点

基本表示（二进制表示）的不可见

编译时类型检查

运算无二义检查

精度控制

语义分析 之 类型检查

对数据对象

类型和使用的操作

是否匹配的一致性检查

静态检查和动态检查

①静态：编译时；使程序正确、有效

②动态：运行时；影响可靠性、效率低

编程方便

语义分析 之 类型检查

语言按类型分类

无类型 没有类型定义

弱类型 所有类型检查在编译时完成

强类型 全部或部分类型检查在运行时完成

语义分析 之 类型转换

某种类型的值转换为另一种类型的值

语言应该提供类型转换机制

隐式(自动)转换

显式(强制)转换

混合运算

表达式给变量值赋

实参向函数形参传值

函数返回值

语义分析 之 类型转换

FORTRAN

类型转换的要求和规则都是隐式的

ADA

类型转换的要求和规则都是显式的

C

类型转换的要求和规则有隐式、显式

语义分析 之 类型转换

两种转换方式

①拓展（扩大）：转换之后的类型值集合包含转换之前类型值集合（整型→实型）

②收缩（缩小）：若转换之前类型值集合包含转换之后类型值集合（实型→整型）

中间代码（三地址码）

三地址代码

一般形式为　　x = y op z

操作码：op

三个地址：x, y和z

对运算类操作

地址 y 和 z 指出两个运算对象

地址 x 用来存放运算结果

三地址码的常见形式

二元运算类赋值语句　x = y op z

其中op为二元运算符或逻辑运算符

一元运算类赋值语句　x = op z

其中op为一元运算符

如一元减uminus

逻辑否定not

移位符，类型转换符等

三地址码的常见形式

复制类赋值语句

x = y

无条件转移语句

goto L

转到标号为L的语句

三地址码的常见形式

条件转移语句

if x rop y goto L 或 **if a goto L**

rop为关系运算符

<、<=、==、>、>=、<>

若 **x** 和 **y** 满足关系**rop**，或 **a** 为**true**时

则执行标号为**L**的语句

否则顺序执行下一语句

三地址码的常见形式

翻译 A := -B*(C+D)

地址	代码
n	t_1 := -B
n+1	t_2 := C+D
n+2	t_3 := t_1* t_2
n+3	A := t_3

教材中的三地址代码中的地址
既代表地址，又表示其对应的值，类似于高级语言

三地址码的常见形式

实际编译器的三地址代码类似于汇编语言
即：把地址与值分开保存
要么是值，要么是地址

```
void func()
{
    int A = 1, B = 2, C = 3, D = 4;

    A = -B*(C+D);
}
```

clang -emit-llvm -S ./llvm_example.c

```
define i32 @func() #0 {
  %A = alloca i32, align 4
  %B = alloca i32, align 4
  %C = alloca i32, align 4
  %D = alloca i32, align 4
  store i32 1, i32* %A, align 4
  store i32 2, i32* %B, align 4
  store i32 3, i32* %C, align 4
  store i32 4, i32* %D, align 4
  %1 = load i32* %B, align 4
  %2 = sub nsw i32 0, %1
  %3 = load i32* %C, align 4
  %4 = load i32* %D, align 4
  %5 = add nsw i32 %3, %4
  %6 = mul nsw i32 %2, %5
  store i32 %6, i32* %A, align 4
  ret void
}
```

三地址码的生成

语法树的遍历

```
void visit(past N){
    for( 从左到右遍历 N 的每个子结点 C )
        visit( C );
    按照 N 的语义规则生成三地址代码;
}
```

三地址码的生成

基本表达式结点翻译方案

primary_expr → ID | NUMBER

基本表达式如果是“变量”

结点中nodeType为“varRef”

三地址代码中需要变量名，即“sValue”

基本表达式如果是“整数”

结点中nodeType为“intValue”

三地址代码中需要常数的值，即“iValue”

三地址码的生成

基本表达式结点翻译方案

primary_expr → ID | NUMBER

设计函数

```
int genPrimaryExpr(past node, char* operand)
```

其中输入参数node指向要翻译的结点

输出参数operand指向该结点的

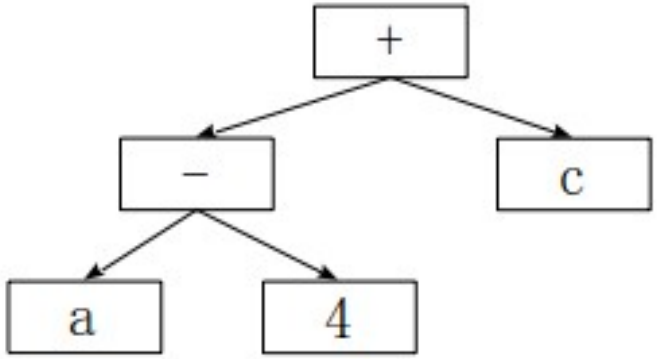
变量名 或 整数值对应的字符串

三地址码的生成

基本表达式结点翻译方案

```
int genPrimaryExpr(past node, char* operand)
{
    if(strcmp(node->nodeType, "intValue") == 0){
        if(operand != NULL)
            sprintf(operand, "%d", node->ivalue);
    }else if(strcmp(node->nodeType, "varRef") == 0){
        if(operand != NULL)
            sprintf(operand, "%s", node->svalue);
    }else{
        printf("ERROR: 发现不支持的运算类型");
        return -1;
    }
    return 1;
}
```

三地址码的生成



表达式结点翻译方案

left指向左子树，right指向右子树

子树可能是基本表达式

也可能是表达式

对genPrimaryExpr行调整:
加入对表达式的处理
使其可处理表达式的子树
参教材 genPrimaryExpr()

如果是基本表达式

调用genPrimaryExpr得到操作数

如果是表达式

如何获取该子树的结果?

递归调用自己生成子树对应的代码

约定：该子树的结果保存在当前的临时变量中

三地址码的生成

基本表达式结点翻译方案

```
int genExpr(past node){
    if(node == NULL) return -1;
    if( strcmp(node->nodeType, "expr") == 0){
        char loperand[50];    char roperand[50];        char oper[5];

        ltype = genPrimaryExpr(node->left, loperand);
        rtype = genPrimaryExpr(node->right, roperand);

        if( ltype == rtype && ltype != -1){
            sprintf(oper, "%c", node->ivalue);
            printf("  %d = %s %s %s\n",getTemVarNum(), loperand, oper, roperand);
            return 1;
    }}
    return -1;
}
```

THE END

QUESTIONS

实验1 词法分析

自学：正则表达式，

Lexical Analysis with Flex