第2章 实现一个简单的编译器

信息与软件工程学院

邓伏虎

实现一个简单的编译器

- 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
```

本章内容

2.1 高级语言的定义

2.2 词法分析

2.3 语法分析

2.4 语义分析

2.5 中间代码生成

2.1 高级语言的定义

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

语句的类型?

赋值语句的一般形式: 变量 = 表达式/变量;

循环语句的一般形式: while 表达式 do 语句;

条件语句的一般形式: if 表达式 语句;

if 表达式 语句1 else 语句2;

2.1 高级语言的定义

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

表达式的种类?

算术表达式 布尔表达式 逻辑表达式

算术表达式构成: 左部表达式/变量 算术运算符 右部表达式/变量

布尔表达式构成: 左部表达式/变量 比较运算符 右部表达式/变量

逻辑表达式构成: 左部表达式/变量 逻辑运算符 右部表达式/变量

2.1 高级语言的定义

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

变量的种类?

标识符

变量：以字母开头，后由字母、数字下划线构成的字符串

关键字：又成为保留字，具有特定含义的名字

常量：整数

浮点数？字符串常量？

运算符：+ - * /

分界符：；　（）　{} 空格 回车等

2.1 高级语言的定义

词法分析器

处理词法规则，也就是降格后的语法规则。

用于识别标识符、关键字、运算符、分界符、常量等单词

对于语法规则来说，这些单词符号具有特定的含义，不可拆分为更小的语法成分，统称为“终结符”

那些语法规则中还可以拆分为更小的语法成分的符号统称为“非终结符”，如“语句”、“表达式”等

2.1 高级语言的定义

语言：是一定的群体用来进行信息交流的工具。

高级程序设计语言：是人与计算机系统之间的信息交流工具。

- 单词：是按照一定的规则由字符组成的串。词法规则

  标识符 → [_a-zA-Z]+[_a-zA-Z0-9]...

- 语句：是按照一定的规则由单词组成的串。语法规则

  while语句→ while(布尔表达式)语句

- 程序：是语句的集合。

2.1 高级语言的定义

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

语法定义

对于赋值语句　　赋值语句 → 标识符 = 表达式;

assign_stmt → identifier = expr;

assign_stmt → ID = expr

2.1 高级语言的定义

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

语法定义

对于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

或者

while_stmt → WHILE (bool_expr) stmt

2.1 高级语言的定义

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

语法定义

对于布尔表达式

bool_expr → expr > expr

| expr < expr

| expr == expr

2.1 高级语言的定义

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

语法定义

算术表达式

expr → primary_expr

| primary_expr + expr

| primary_expr - expr

primary_expr → ID | NUMBER

2.1 高级语言的定义

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

语法定义

程序及语句

program → stmt | program stmt

stmt → while_stmt | assign_stmt

2.1 高级语言的定义

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

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

2.2 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：带标记的逻辑分组（源程序的子串）

实现方法：

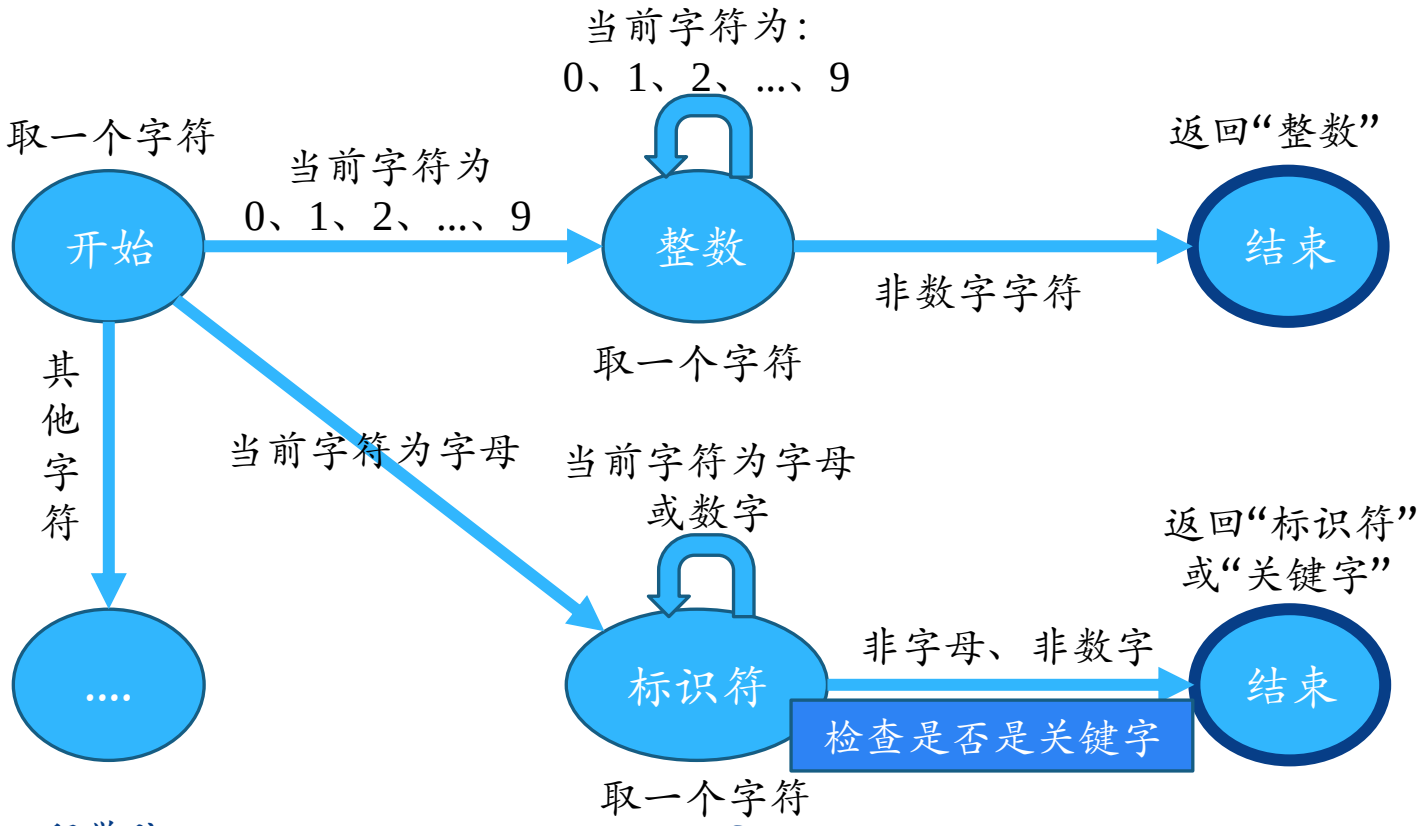
状态转移图法（手动实现）

正则表达式法（自动实现）

2.2 词法分析

状态转移图法

根据当前输入字符，确定跳转到相应的状态。



2.2 词法分析

状态转移图法

所需函数:

取一个字符	getChar()
判断当前符号	checkSymbol()
关键字检查	checkKeywords()
将当前符号加入到已识别的字符串中	addChar()
跳过空格/注释	getNonBlank()

2.3 语法分析

语法分析目的

确定输入程序是否满足语法规则

生成完整的语法分析树

语法分析前提

已经给出了相应的语法规则

已经给出了相应的源代码

语法规则：
rule 1
rule 2
...
rule n

如何对应？

源代码：
main()
{
...
}

2.3 语法分析

语法分析方法

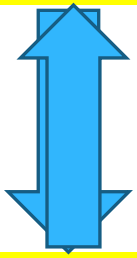
从语法规则出发

检查语法规则能否推导出源程序

从源程序出发

检查源程序是否符合语法规则

语法规则：
rule 1
rule 2
...
rule n



源代码：
main()
{
...
}

(1)program → stmt | program stmt　　(2)stmt → while_stmt | assign_stmt
(3)while_stmt → WHILE (bool_expr) stmt　(4)assign_stmt → ID = expr ;
(5)bool_expr → expr > expr | expr < expr | expr == expr
(6)expr → primary_expr | expr + expr | expr - expr
(7)primary_expr → ID | NUMBER

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

从语法规则出发

program => stmt
=> 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 (ID < ID) ID = ID + expr;
=> WHILE (ID < ID) ID = ID + primary_expr;
=> WHILE (ID < ID) ID = ID + NUMBER;

(1)program → stmt | program stmt　　(2)stmt → while_stmt | assign_stmt
(3)while_stmt → WHILE (bool_expr) stmt　(4)assign_stmt → ID = expr ;
(5)bool_expr → expr > expr | expr < expr | expr == expr
(6)expr → primary_expr | expr + expr | expr - expr
(7)primary_expr → ID | NUMBER

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

从源程序出发

词法分析	while (a < b) a = a + 1;
WHILE	**WHILE (ID < ID) ID = ID + NUMBER;**
WHILE(	**(ID < ID) ID = ID + NUMBER;**
WHILE(ID	**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;**

(1)program → stmt | program stmt　　(2)stmt → while_stmt | assign_stmt
(3)while_stmt → WHILE (bool_expr) stmt　(4)assign_stmt → ID = expr ;
(5)bool_expr → expr > expr | expr < expr | expr == expr
(6)expr → primary_expr | expr + expr | expr - expr
(7)primary_expr → ID | NUMBER

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

从源程序出发

词法分析	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** = primary_expr	**+ NUMBER;**
WHILE(**bool_expr**) **ID** = expr	**+ NUMBER;**
WHILE(**bool_expr**) **ID** = expr +	**NUMBER;**
WHILE(**bool_expr**) **ID** = expr + NUMBER	**;**
WHILE(**bool_expr**) **ID** = expr + primary_expr	**;**
WHILE(**bool_expr**) **ID** = expr + expr	**;**
WHILE(**bool_expr**) **ID** = expr	**;**

(1)program → stmt | program stmt　　(2)stmt → while_stmt | assign_stmt
(3)while_stmt → WHILE (bool_expr) stmt　(4)assign_stmt → ID = expr ;
(5)bool_expr → expr > expr | expr < expr | expr == expr
(6)expr → primary_expr | expr + expr | expr - expr
(7)primary_expr → ID | NUMBER

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

从源程序出发

词法分析	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	
stmt	
program	

2.3 语法分析

递归下降分析法

从语法规则出发

为每一个非终结符构造一个对应的解析函数

语法规则右侧为其左侧非终结符所对应的“函数体”

在函数体中

若当前符号为终结符 对应于 从输入串中识别到该终结符

若当前符号为非终结符 对应于 调用相应解析函数

语法规则中的 ‘|’ 对应于 “if-else”语句

2.3 语法分析

递归下降分析法

若当前符号为终结符 对应于 从输入串中识别到该终结符

涉及到的函数：终结符匹配函数 读入下一个单词

```
assign_stmt → ID = expr;
if (match(ID))          //match()功能：检查当前符号与参数是否一致
    advance();          //advance()功能：读入下一个单词

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

2.3 语法分析

递归下降分析法

若当前符号为非终结符 对应于 调用相应解析函数

涉及到的函数： 调用该非终结符对应的解析函数

assign_stmt → ID = expr;

对于非终结符 expr, 调用相应的解析函数

analyse_expr(); //expr的解析函数

2.3 语法分析

递归下降分析法

语法规则中的 ‘|’ 对应于 “if-else” 语句

primary_expr → ID | NUMBER

```
if(match(ID))
    advance();
else if (match(NUMBER))
    advance();
else
    error(“.....”); return 0;
```

stmt → while_stmt | assign_stmt

以非终结符开始的多个规则选哪一个规则？如何编写？

2.3 语法分析

递归下降分析法

语法规则中的 ‘|’ 对应于 “if-else” 语句

stmt → while_stmt | assign_stmt

以非终结符开始的多个规则
选哪一个规则？如何编写？

```
if(analyse_while_stmt()){
    return 1;
}else if (analyse_assign_stmt){
    return 1;
}else
    error(“.....”); return 0;
```

蒙一个?

2.3 语法分析

递归下降分析法

语法规则中的 ‘|’ 对应于 “if-else” 语句

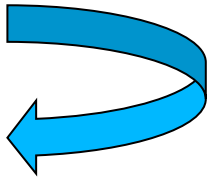
while_stmt → WHILE (bool_expr) stmt
assign_stmt → ID = expr;

while 语句总是以 WHILE 开始
赋值语句总是以 ID 开始

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

程序语句规则： program → stmt | program stmt

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



递归调用，死循环！

program → stmt | program stmt　　=>　　program → stmt stmt stmt ... stmt

改写规则：

program → stmt {stmt}

大括号表示其中的语法成份可以重复0次或多次

```
void analyse_program(){
    analyse_stmt();
    while(  nextToken ! = EOF  ) {
        analyse_stmt();
    }
}
```

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();                                 //读入下一个字符，并设置为当前分析符号
    if (!match('=')) {                         //匹配赋值符号
        printf("ERROR: Expect '=' .\n");
        return 0;
    }
    advance();
    analyse_expr();                            //调用非终结符expr对应的解析函数
    if (!match(';')) {                         //匹配分号
        printf("ERROR:  Expect ';' .\n");
        return 0;
    }
    advance();
    return 1;
}
```

$E \rightarrow TE_1$

$E_1 \rightarrow +TE_1 \mid \varepsilon$

$T \rightarrow FT_1$

$T_1 \rightarrow *FT_1 \mid \varepsilon$

$F \rightarrow (E) \mid i$

```
void E( )
{
   T( );
   E1( );
   return 1;
}
```

```
void T()
{
   F( );
   T1( );
   return 1;
}
```

```
void T1( )
{
  if( sym = = '*' )
  {
        advance( );
        F( );
        T1( );
   }
   return 1;
}
```

```
void F( )
{
   if( sym = = '(' )   {
        advance( );
        E( );
        if( sym = = ')' )  advance( );  return 1;
        else error( );
   }
   else if( sym = = 'i' )  advance( );  return 1;
   else error( );
}
```

```
void E1( )
{
  if ( sym = = '+' )
  {
        advance( );
        T( );
        E1( );
   }
   return 1;
}
```

输入串： i+i*i

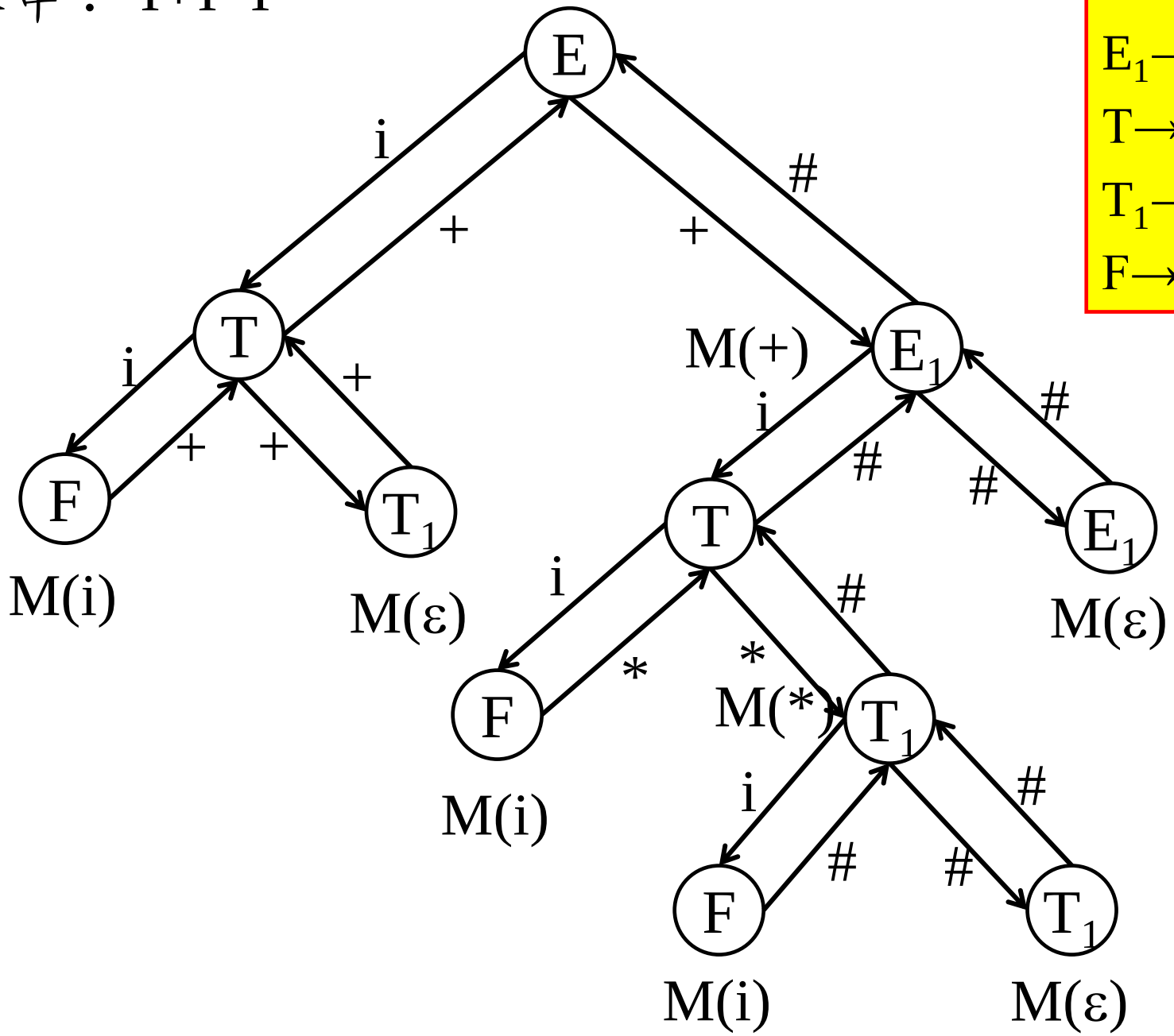
$E \rightarrow TE_1$

$E_1 \rightarrow +TE_1 \mid \varepsilon$

$T \rightarrow FT_1$

$T_1 \rightarrow *FT_1 \mid \varepsilon$

$F \rightarrow (E) \mid i$



2.3 语法分析

建立抽象语法树

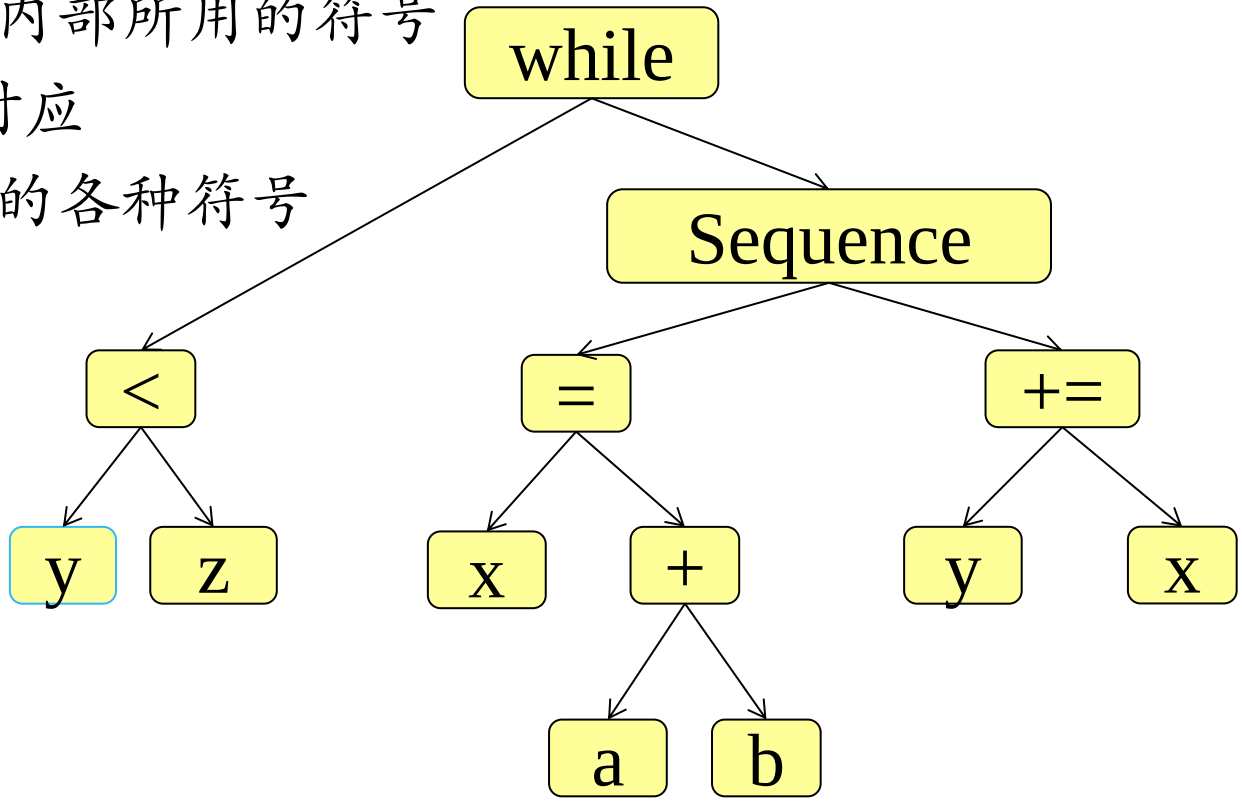
抽象语法树是源代码的抽象语法结构的抽象树状表示形式

不保留语法规则内部所用的符号

与具体的语法树相对应

包括语法规则中的各种符号

```
while y < z
    x = a + b
    y += x
```



2.3 语法分析

建立抽象语法树

节点设计

```
struct _ast{
        左子树节点指针
        右子树节点指针
        节点类型
        保存整数信息
        保存字符串信息
        ...
        ...
}
```

```
typedef struct _ast;
typedef struct _ast *past;

struct _ast{
        past left;
        past right;
        char* nodetype;
        int ivalue;
        char* svalue
        ...
        ...
}
```

2.3 语法分析

建立抽象语法树

与节点相关函数的设计

创建抽象语法树节点	past newAstNode()
创建数值节点	past newNum(int value)
创建标识符节点	past newVarRef(char* name)
创建表达式节点	past newExpr(int opr, past left, past right)

2.3 语法分析

建立抽象语法树

建立表达式 a - 4 + c的抽象语法树

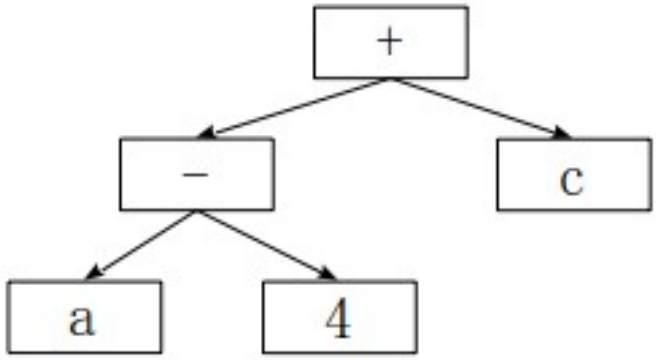
P1 = newVarRef (Pa);

P2 = newNum(4);

P3 = newExpr('-', P1, P2);

P4 = newVarRef(Pc);

P5 = newExpr('+', P3, P4);



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

2.3 语法分析

符号表

作用：登记符号的属性值，查找符号的属性，检查其合法性

属性列表：

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”对于程序设计语言至关重要!

2.4 语义分析

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”?

```
int a;
int b;
int c;
c = a + b;
```

```
float a;
float b;
float c;
c = a + b;
```

2.4 语义分析

数据类型

基本(初等)数据类型： 数值数据,逻辑数据,字符数据,指针类型等。

int, float, char, string, bool

复合数据类型： 数组、结构、表、栈、树等。

array, struct, table, stack, tree

抽象数据类型: Ada的包(Package),C++的类(Class)等。

2.4 语义分析

数据类型

计算机如何存储数字?

整数? 负数? 浮点数?

IEEE 浮点标准754格式

JAVA语言

数据类型

2.4 语义分析

数据类型

强类型语言：要求变量的实用要严格符合定义，所有变量都必须先定义再使用，一旦某一个变量被定义类型，不经强制转换，类型不会发生变化。

包含：JAVA, .net, python, C++

弱类型语言：某一个变量被定义类型，但可以根据环境变化自动进行转换，不需要经过强制转换。

包含：VB, PHP, JavaScript

2.4 语义分析

类型检查

对数据对象的类型和使用的操作是否匹配开展一致性检查

静态检查：编译时进行，保证程序正确、有效

动态检查：运行时进行，影响可靠性、效率低

语言按类型分类：无类型语言、弱类型语言、强类型语言

2.4 语义分析

类型转换

某种类型的值转换为另一种类型的值

隐式（自动）转换 和 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

用户参与控制

2.4 语义分析

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）

2.5 中间代码生成

三地址代码

一般形式为　x = y op z 或者 x = op y 或者 x = y

操作码: op

x, y和z 通常既可代表地址，也可以表示其对应的值

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

2.5 中间代码生成

三地址代码

（1）一元运算类赋值语句　x = op y

其中op为一元运算符，如减号、逻辑否定、移位符、类型转换符号等

（2）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

（3）复制类复制语句　x = y

2.5 中间代码生成

三地址代码

（4）条件跳转语句　if x rop y goto L 或者 if a goto L

其中rop为关系运算符 <、<=、 >、>=、==、<>

//如果x和y满足关系 rop时或者a为true时，执行标号为L的语句，否则顺序执行下一条语句

（5）无条件跳转语句　goto L　　//无条件跳转到标号为L的语句

2.5 中间代码生成

三地址代码

if a<b then a:=a+b else a:=a-b

```
100:   if a < b goto 102
101:   goto  105
102:   t1 = a + b
103:   a = t1
104:   goto  107
105:   t2 = a - b
106:   a = t2
107:  XXX
```

2.5 中间代码生成

目标代码

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

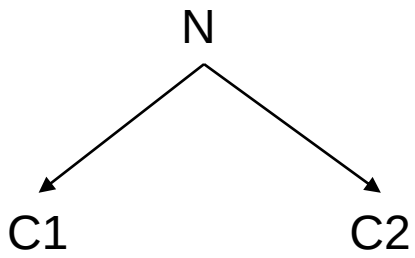
```
define i32 @func() #0 {
  %A = alloca i32, align 4
  %B = alloca i32, align 4
  %C = alloca i32, align 4
  %D = alloca i32, align 4
  store i32 1, i32* %A, align 4
  store i32 2, i32* %B, align 4
  store i32 3, i32* %C, align 4
  store i32 4, i32* %D, align 4
  %1 = load i32* %B, align 4
  %2 = sub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

2.5 中间代码生成

从语法树到三地址代码

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

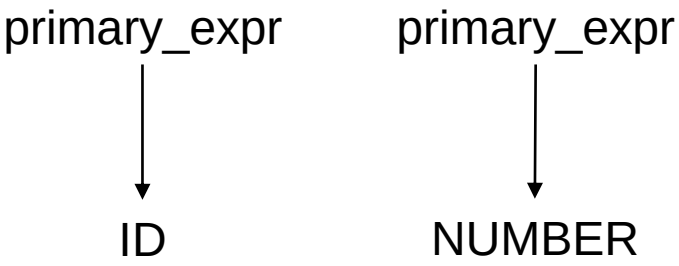
$N \rightarrow C_1\ C_2$



2.5 中间代码生成

基本表达式节点翻译方案

primary_expr → ID | NUMBER



基本表达式如果是“变量”

节点中的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”

节点中nodeType为“intValue”

三地址代码中需要常数的值，即“ivalue”

2.5 中间代码生成

基本表达式节点翻译方案

primary_expr → ID | NUMBER

设计函数 int genPrimaryExpr(past node, char* operand)

其中参数node 指向需要翻译的节点

参数operand 指向该节点的变量名或整数值对应的字符串

2.5 中间代码生成

```
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;
}
```

2.5 中间代码生成

表达式节点翻译方案

expr → left right

设计函数 int genExpr(past node)

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

若子树是基本表达式，调用genPrimaryExpr得到操作数

若子树是表达式，递归调用表达式生成子树的代码

如何获取该子树的结果?

约定：该子树的结果保存在当前的临时变量中

```
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”, genTemVarNum(), loperand, oper,
        roperand);

            return 1;
        }
    }
    return -1;
}
```