第六章 中间代码生成

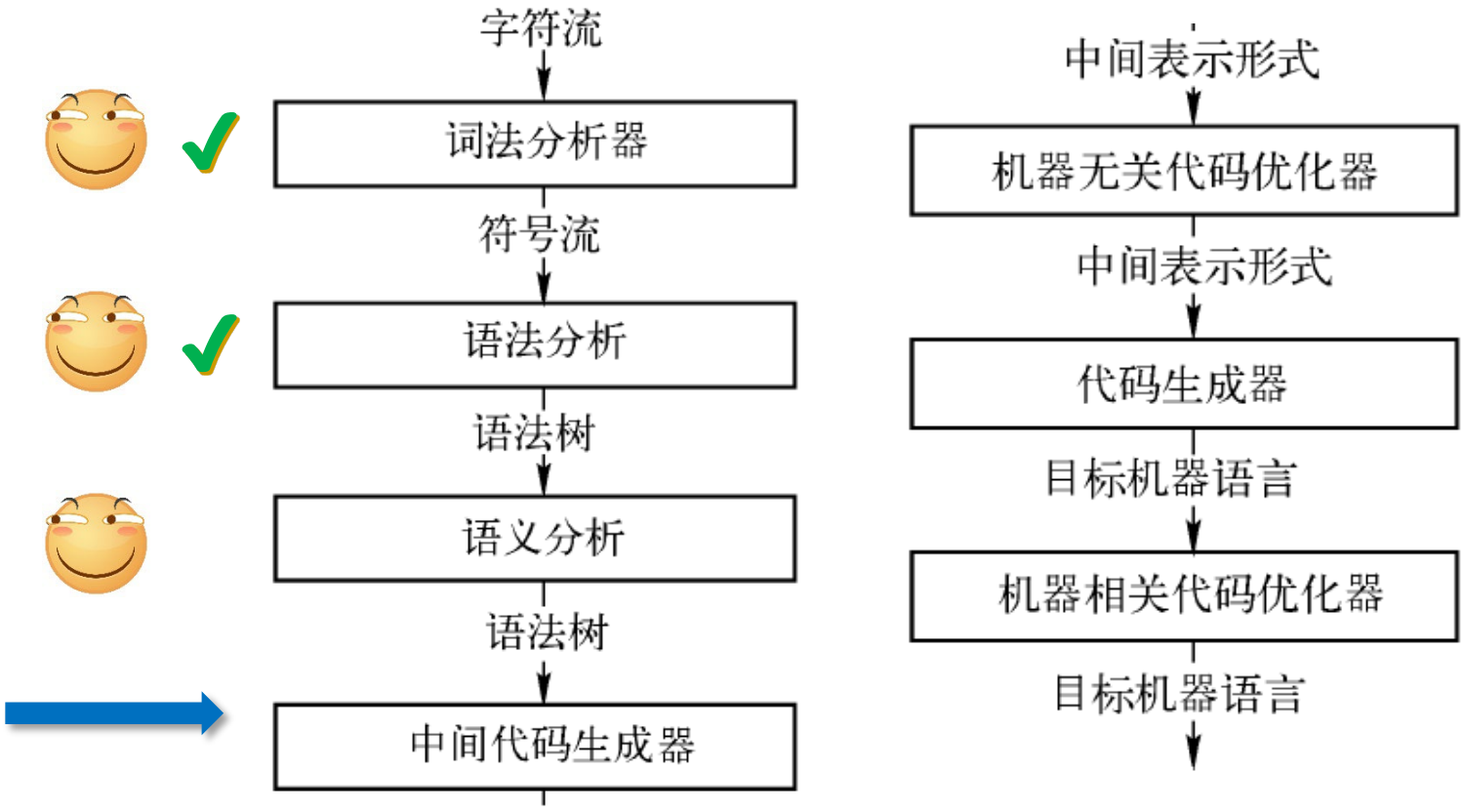
《编译原理》

谭添

南京大学计算机系

2026年春季

闯关进度

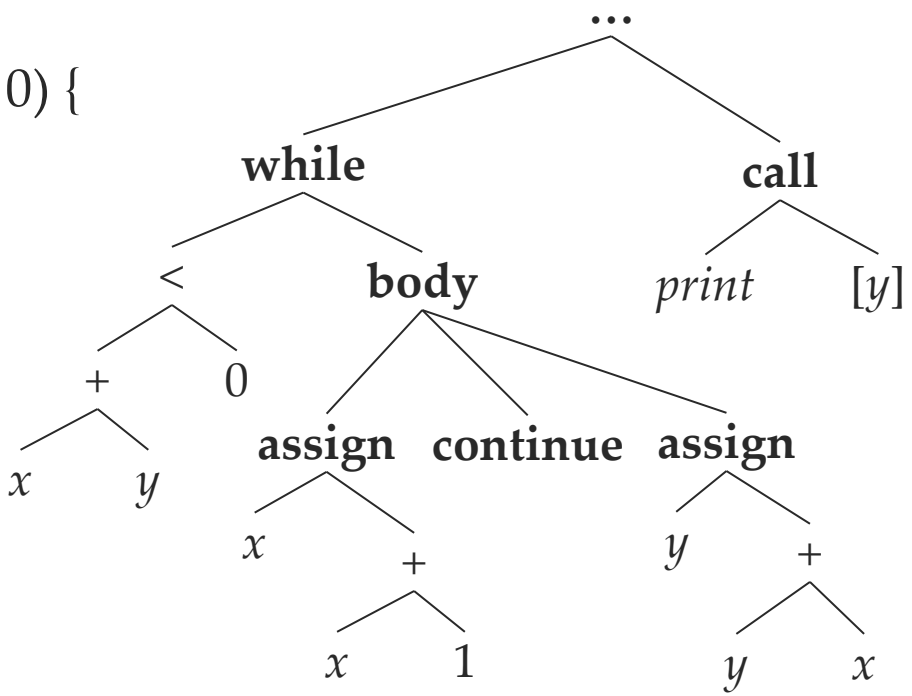

字符流
词法分析器
符号流
语法分析
语法树
语义分析
语法树
中间代码生成器
中间表示形式
机器无关代码优化器
中间表示形式
代码生成器
目标机器语言
机器相关代码优化器
目标机器语言

一个例子

```
while (x + y < 0) {
    x = x + 1;
    continue;
    y = y + x;
}
print(y);
```

AST表示

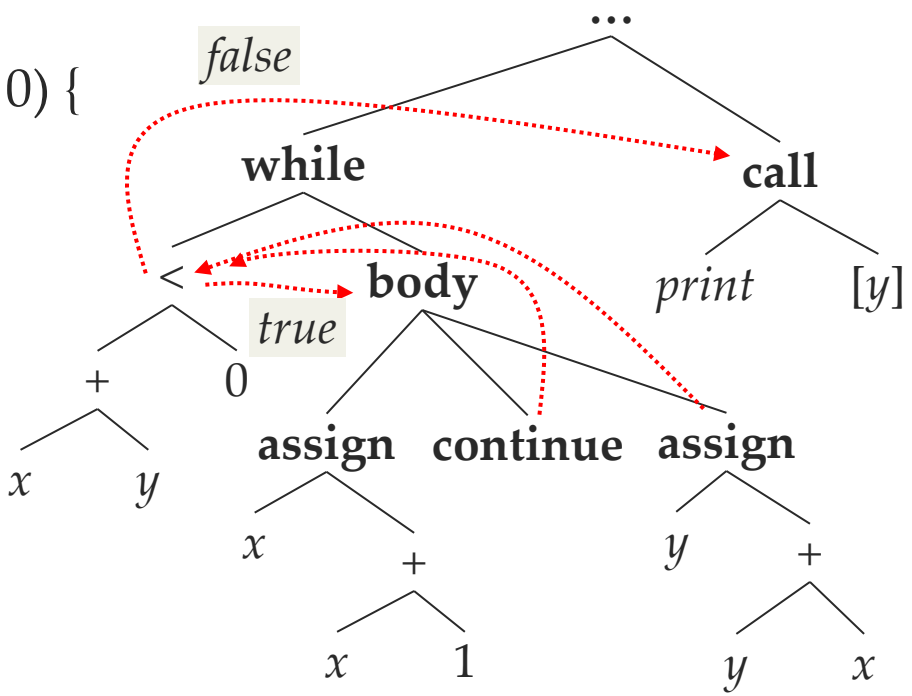
```
while (x + y < 0) {
    x = x + 1;
    continue;
    y = y + x;
}
print(y);
```



- 高级的程序表示 (贴近源语言层面)
- 语言相关 (与源语言语法紧密关联)
- 适合一些类型检查

AST表示

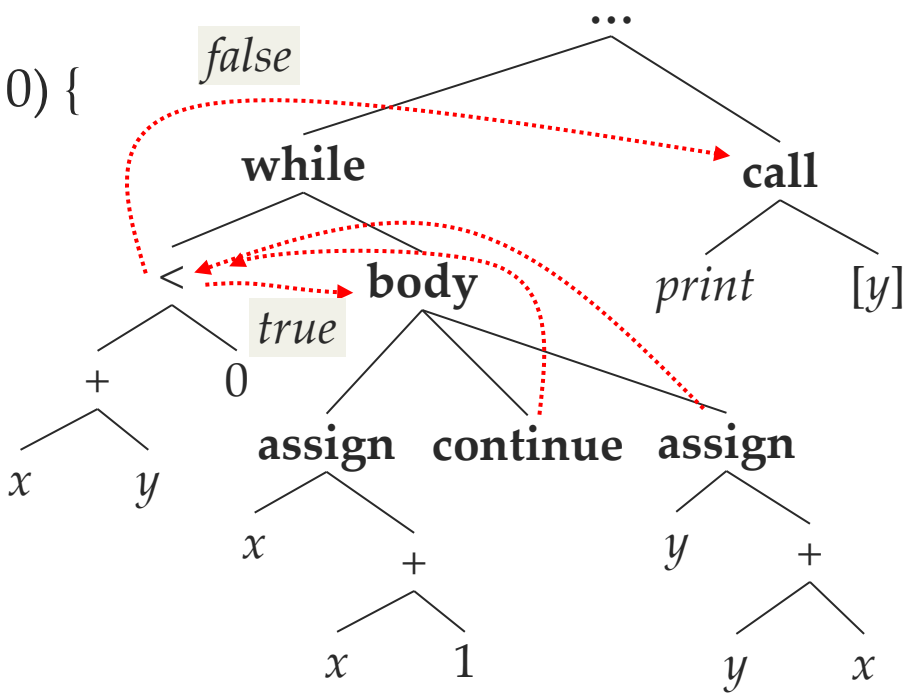
```
while (x + y < 0) {
    x = x + 1;
    continue;
    y = y + x;
}
print(y);
```



- 高级的程序表示 (贴近源语言层面)
- 语言相关 (与源语言语法紧密关联)
- 适合一些类型检查
- 不紧凑、缺少控制流信息 (不利于分析与优化😳)

中间代码表示

```
while (x + y < 0) {
    x = x + 1;
    continue;
    y = y + x;
}
print(y);
```




```
label L1
    t = x + y;
    if t < 0 goto L2
    else goto L3;
label L2
    x = x + 1;
    goto L1;
    y = y + x;
    goto L1;
label L3
    print(y);
```

三地址代码

- 紧凑、明晰的控制流信息 (易于分析与优化)
- 与具体语言无关 (通用)
- 与具体机器无关 (通用++)

编译器前端的逻辑结构 (1)

- 前端是对源语言进行分析并产生中间表示
- 处理与源语言相关的细节，与目标机器无关

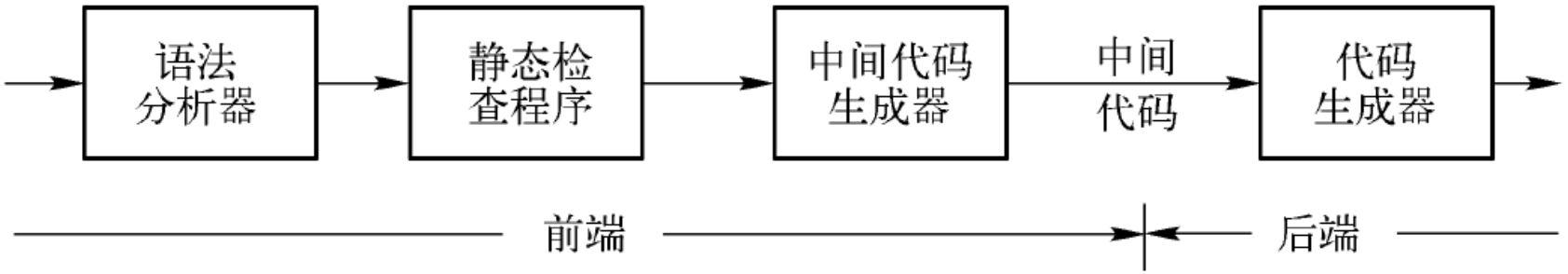
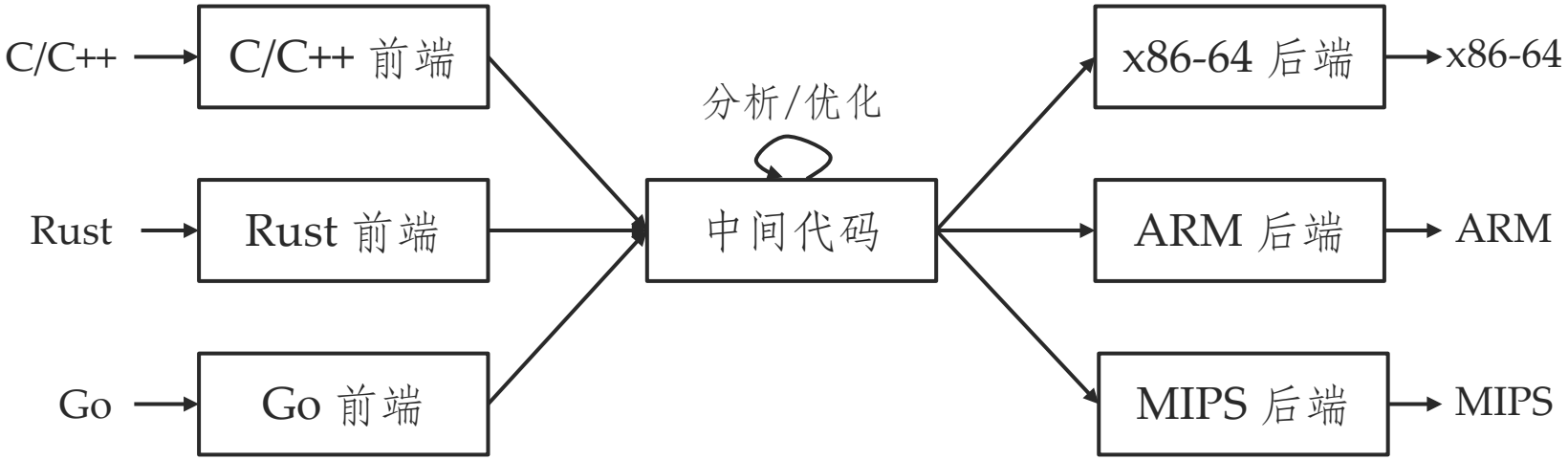


图 6-1　一个编译器前端的逻辑结构

编译器前端的逻辑结构 (2)

- 前端后端分开的好处
  - 不同的源语言、不同的机器可以得到不同的编译器组合
  - 可复用的分析/优化的逻辑



例: LLVM

本章内容

- 中间代码表示
  - 三地址代码：$x = y\ op\ z$
- 类型检查
  - 类型、类型检查、表达式的翻译
- 中间代码生成
  - 控制流、回填

中间代码表示及其好处

- 形式
  - 多种中间表示 (Intermediate Representations, **IR**s)，不同层次
  - **抽象语法树**
  - **三地址代码**
- **重定位**
  - 为新的机器建编译器，只需要做从中间代码到新的目标代码的翻译器 (前端独立)
- 高层次的优化
  - 优化与源语言和目标机器都无关

中间代码的实现

- 静态类型检查和中间代码生成的过程都可以用语法制导的翻译来描述和实现
- 对于抽象语法树这种中间表示的生成，第五章已经介绍过

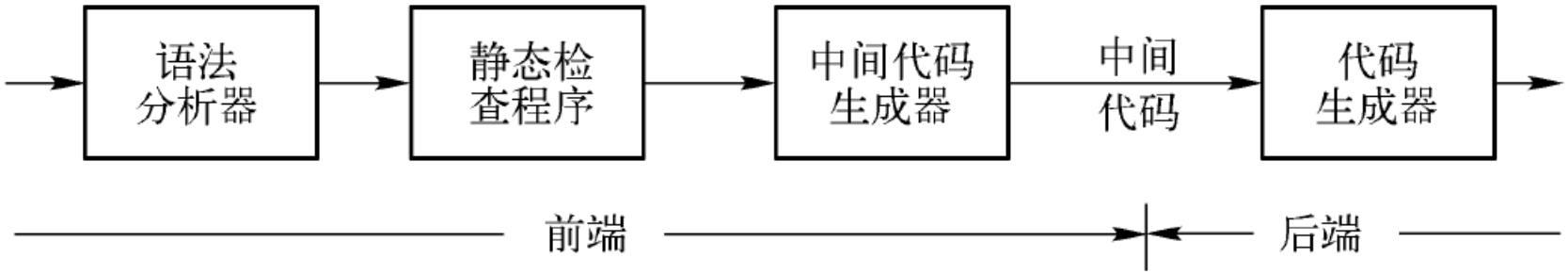
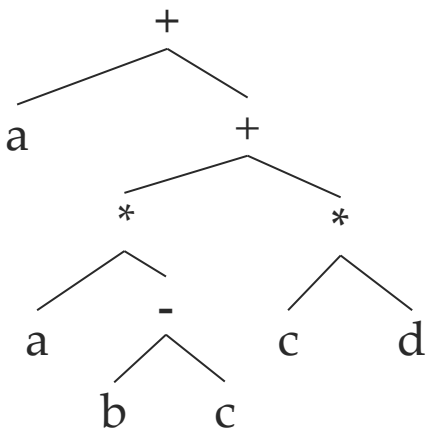


图 6-1　一个编译器前端的逻辑结构

生成抽象语法树的语法制导定义

- a + a * (b – c) + c * d的抽象语法树



	PRODUCTION	SEMANTIC RULES
1)	$E \to E_1 + T$	$E.node =$ **new** $Node('+', E_1.node, T.node)$
2)	$E \to E_1 - T$	$E.node =$ **new** $Node('-', E_1.node, T.node)$
3)	$E \to T$	$E.node = T.node$
4)	$T \to (E)$	$T.node = E.node$
5)	$T \to$ **id**	$T.node =$ **new** $Leaf($**id**, **id**.$entry)$
6)	$T \to$ **num**	$T.node =$ **new** $Leaf($**num**, **num**.$val)$

三地址代码 (1)

- 目标
  - 接近大多数目标机器的执行模型 (机器码)
  - 支持大多数目标机器提供的数据类型和操作
  - 提供有限度的、高于机器码的抽象表达能力，更容易表达出大多数 (命令式) 高级语言的特性
- 特征
  - 以指令为单位
  - 每条指令只有有限数量的运算分量 (通常≤3)
  - 支持基本数据类型及其运算
  - 支持复合数据类型 (如数组、结构体) 及其操作
  - 支持函数

三地址代码 (2)

- 每条指令右侧最多有一个运算符

  一般情况可以写成$x = y\ op\ z$

  - 接近机器码 (易于从机器角度优化)
  - 大大减少组合情况的数量 (易于处理)

- 允许的**运算分量**
  - 变量：源程序中的名字作为三地址代码的地址
  - 常量：源程序中出现或生成的常量
  - 编译器生成的临时变量 (也可与源程序变量等同视之)

三地址代码 (3)

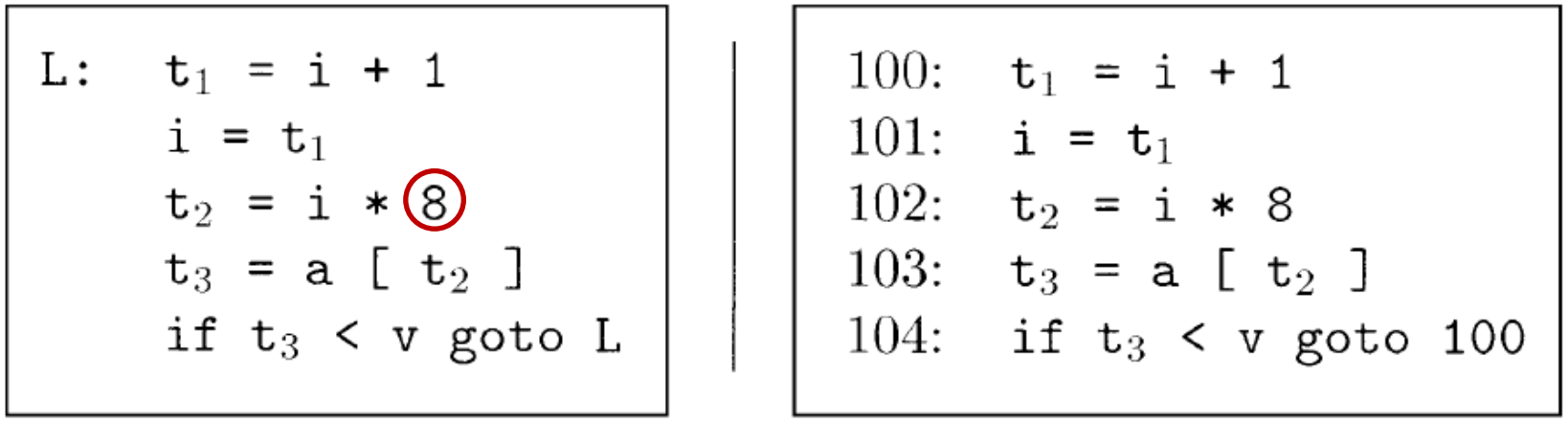
- 指令集合 (1)
  - 复制指令: $x = y$
  - 运算/赋值指令: $x = y\ op\ z$　　$x = op\ y$
  - 无条件转移指令: goto L
  - 条件转移指令: if x goto L　　if not x goto L
  - 条件转移指令: if x *relop* y goto L

三地址代码 (4)

- 指令集合 (2)
  - 过程调用/返回
    - param x_1 // 设置参数
    - param x_2
    - …
    - param x_n
    - call p, n // 调用子过程p，n为参数个数
  - 带下标的复制指令：$x = y[i]$ $x[i] = y$
    - 注意：i表示离开数组位置第i个字节，而不是数组的第i个元素
  - 地址/指针赋值指令
    - $x = \&y$ $x = *y$ $*x = y$

例子

- 语句
  - do
    i = i + 1;
    while (a[i] < v);



a) 符号标号　　b) 位置号

三地址指令的四元式表示方法

- 在实现时，可使用**四元式**/三元式/间接三元式/**静态单赋值**来表示三地址指令
- **四元式**：可以实现为记录 (或结构)
  - 格式 (字段)：　　*op*　　arg_1　　arg_2　　*result*
  - *op*：运算符的内部编码
  - arg_1, arg_2, *result*是地址
  - $x = y + z$　　+　　y　　z　　x
- 单目运算符不使用arg_2
- param运算不使用arg_2和*result*
- 条件转移/非条件转移将目标标号放在*result*字段

四元式的例子

- 赋值语句：a = b * – c + b * – c

```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
 a = t5
```

a) 三地址代码

	op	arg_1	arg_2	*result*
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
	...			

b) 四元式

图 6-10　三地址代码及其四元式表示

数组操作的四元式表示

- 读数组：$x = y[i]$
- 写数组：$x[i] = y$

op	arg_1	arg_2	*result*
=[]	y	i	x
[]=	i	y	x

静态单赋值 (SSA)

- Static Single Assignment (SSA) 是一种特殊的三地址代码，其所有变量在代码中只被赋值一次

```
x = a;
y = x + 1;
x = b;
z = x * 2;
```

```
x1 = a;
y = x1 + 1;
x2 = b;
z = x2 * 2;
```

SSA的构造

- 基本构造思路
  - 为每个变量维护一个计数器
  - 从函数入口开始遍历函数体
  - 遇到变量赋值时，为其生成新名字，并替换
  - 将新变量名传播到后续相应的使用处，并替换

```
x = a;          x1 = a;
y = x + 1;      y = x1 + 1;
x = b;          x2 = b;
z = x * 2;      z = x2 * 2;
```

- 通常只针对函数内的变量(即局部变量)计算SSA
- 全局变量的SSA在实际当中难以计算

SSA对分支的处理

- 在分支汇合处插入φ语句
- 对于同一个变量x在不同路径中被赋值的情况，使用$x_i=\varphi(x_j, x_k, \ldots)$来合并不同的赋值

```
if (flag)
    x = –1;
else
    x = 1;
y = x * a;
```

```
if (flag)
    x1 = –1;
else
    x2 = 1;
x3 = φ(x1, x2);
y = x3 * a;
```

SSA对循环的处理

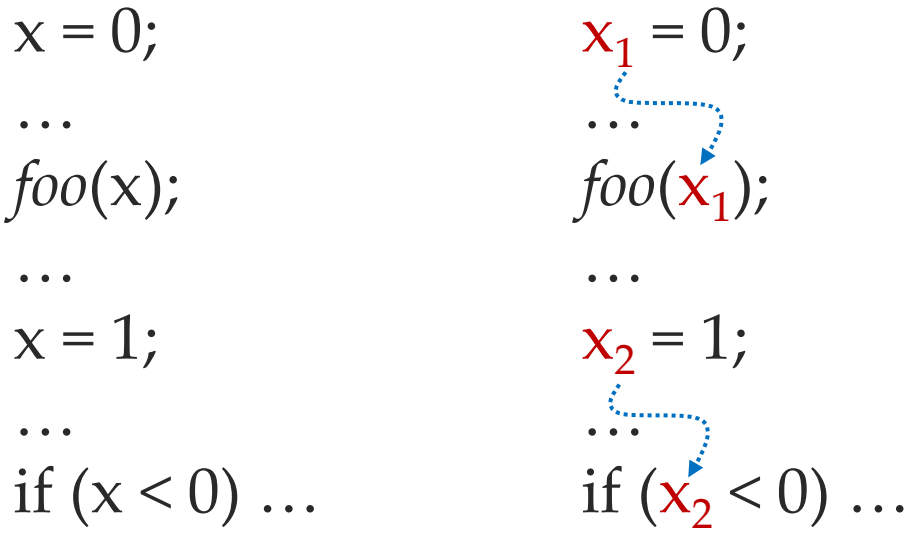
- 循环被翻译成分支与跳转语句的组合
- 直接按照相应语句的方式处理

```
x = j;
while (i < 0) {
3: b = x;
   x = k;
5: c = x;
   …
}
y = x * a;
```

```
100: x1 = j;
101: x3 = φ(x1, x2);
102: if i ≥ 0 goto 122
       b = x3;
103: x2 = k;
       c = x2;
       …
121: goto 101
122: y = x3 * a;
```

SSA的作用

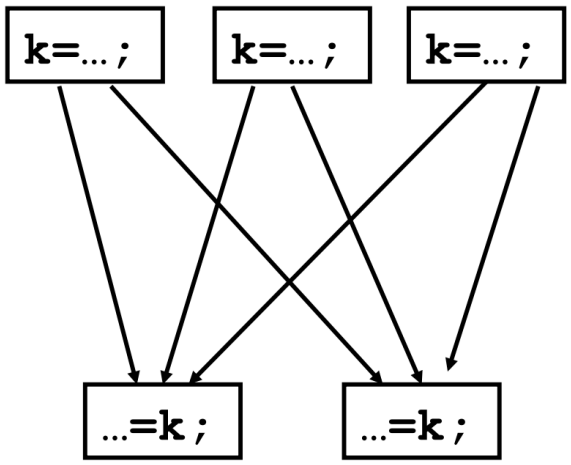
- 每个变量只被赋值一次，相当于都变成const变量
- 简化了数据流分析和某些优化
- 使得定义-使用链 (def-use chain) 易于计算
  - 关联每个变量的定义 (赋值) 及其相应的使用
  - 许多分析和优化所需的关键信息
  - SSA形式中，定义-使用关系非常清晰，且可以线性复杂度进行计算



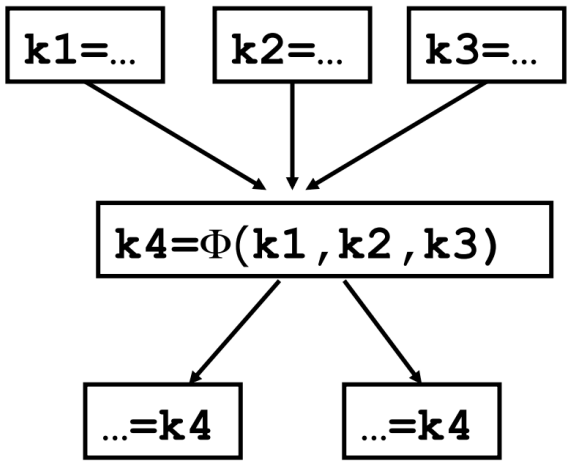
SSA的作用

- 每个变量只被赋值一次，相当于都变成const变量
- 简化了数据流分析和某些优化
- 使得定义-使用链 (def-use chain) 易于计算
  - 关联每个变量的定义 (赋值) 及其相应的使用
  - 许多分析和优化所需的关键信息
  - SSA形式中，定义-使用关系非常清晰，且可以线性复杂度进行计算

|Defs| × |Uses|



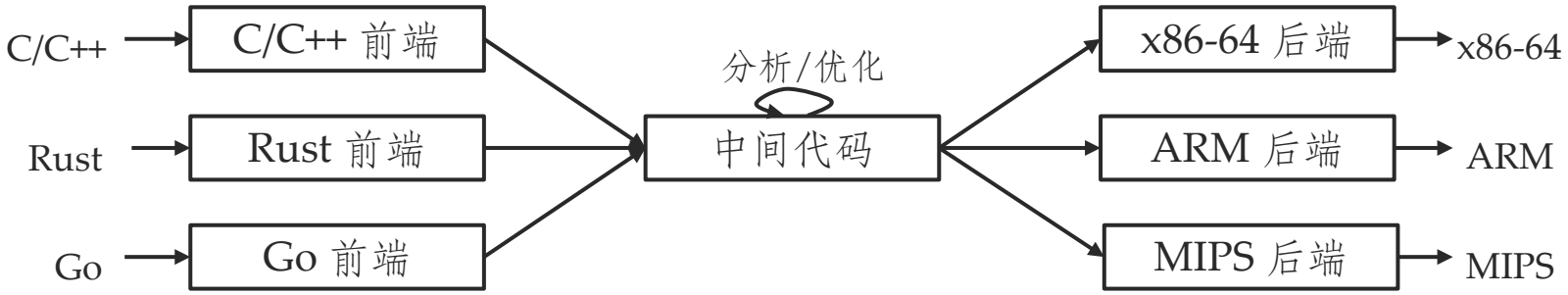
|Defs| + |Uses|



广泛使用于现代编译器中(如LLVM)

回顾

- 中间代码 (Intermediate Representation, IR)
  - 增强编译器各组件 (前端、后端、分析/优化) 的可复用性



- 三地址代码 (Three-Address Code, TAC)
  - 广泛使用的一种中间表示，抽象层次接近机器码
  - 每条指令最多 3 个运算分量，具有明晰的控制流信息
  - 易于分析和处理
- 静态单赋值 (Static Single Assignment, SSA)
  - 每个变量在代码中最多只有1次赋值
  - 可简化数据流分析和某些优化

例子

- 语句
  - do
    i = i + 1;
    while (a[i] < v);

```
L:   t1 = i + 1
     i = t1
     t2 = i * 8
     t3 = a [ t2 ]
     if t3 < v goto L
```

a) 符号标号

```
100:  t1 = i + 1
101:  i = t1
102:  t2 = i * 8
103:  t3 = a [ t2 ]
104:  if t3 < v goto 100
```

b) 位置号

类型和声明

- **类型检查** (Type checking)
  - 利用一组规则来检查运算分量的类型和运算符的预期类型是否匹配
- 类型信息的用途
  - 查错、确定名字 (对应的数据) 所需的内存空间、计算数组元素的地址、类型转换、选择正确的运算符
- 本节的内容
  - 确定名字的类型
  - 变量的存储空间布局 (相对地址)

类型

- 基本类型
  - 程序设计语言中的原子类型
    - 如：boolean, char, integer, float, void, …
  - 通常这些类型的运算都有对应的机器指令
- 复合类型
  - 由基本类型或其它复合组合而成的类型，为程序设计语言提供更强的抽象和表达能力
    - 如：int [2][3] (数组), struct { int a[10]; float f; } (结构体), boolean foo(int x) (函数), …
- **类型表达式** (Type expression)：表示类型的表达式
  - 基本类型：类型名字
  - 复合类型：通过类型构造算子作用于类型表达式得到

数组类型

- 表示同类型数据的聚合
- 类型构造算子array，有两个参数
  - 数字：表示数组的长度
  - 类型：表示数组元素的类型
- 如：int [2][3]
  - 对应类型表达式 array(2, array(3, integer))

记录类型

- 表示不同类型数据的聚合 (结构体、类)
- 类型构造算子record，有多组字段
  - 字段名：字段名字，可用于在记录中引用该字段
  - 类型：字段对应数据的类型
- 记录的基本构造算子是笛卡尔积×
  - 如果s, t是类型表达式，其笛卡尔积$s \times t$也是类型表达式
  - 在记录类型的构造中起组合作用
    - 组合字段名与相应类型
    - 组合多组字段
- 如： struct { int a[10]; float f; } st;
  - 对应类型表达式 record((a × array(10, int)) × (f × float))

函数类型

- 表示程序中函数的类型
- 类型构造算子→，接收参数类型与返回值类型，并构造出函数类型
- 如：int foo(float x, long[5] y) { … }
  - 对应类型表达式 (float × array(5, long)) → int

类型等价性

- 判断两个类型是否等价
  - 类型检查的基础
  - 不同的语言有不同的类型等价的定义
  - 当语言允许为自定义类型命名时，主要有两类等价性判定方式
- **名等价** (Name Equivalence)
  - 类型表达式t与u等价当且仅当它们对应的类型名字相同
- **结构等价** (Structure Equivalence)
  - 对于基本类型，比较它们名字是否相同
  - 对于复合类型，比较类型构造算子；若相同，(递归) 比较构造算子的各参数分量

类型的声明

- 处理基本类型、数组类型或记录类型的文法
  - $D \rightarrow T\ \mathrm{id};\ D \mid \varepsilon$
  - $T \rightarrow B\ C \mid \mathrm{record}\ '\{'\ D\ '\}'$
  - $B \rightarrow \mathrm{int} \mid \mathrm{float}$
  - $C \rightarrow \varepsilon \mid [\mathrm{num}]\ C$

- 应用该文法及其对应的语法制导定义，除了得到类型表达式之外，还得进行各种类型的存储布局

局部变量的存储布局

- 变量的类型可以确定变量需要的内存
  - 即类型的**宽度**(该类型一个对象所需的存储单元的数量)
  - 可变大小的数据结构只需要考虑指针
- 函数的局部变量总是分配在连续的区间
  - 因此给每个变量分配一个相对于这个区间开始处的相对地址(偏移量)

内存补齐（Padding）

- 提升内存访问效率的一种技术
- 现代计算机体系架构的内存通常不允许任意寻址
  - 内存以行×列的方式组织
  - 内存访问以“行”的宽度为单位
    - 行宽为w字节时，内存值允许访问地址为kw~k(w+1)这一区间的数据
    - 例如w=8，CPU可以访问0~7, 32~39，不能访问5~12, 17~24
  - 支持任意寻址会显著增加内存复杂度
- 内存（通常）不能“跨行”访问

内存补齐（Padding）

- 内存（通常）不能“跨行”访问
- 为了内存数据能被高效读取，编译器给数据布局时会使用“补齐”的技术
  - 若某数据d的宽度为u字节且小于行宽（w字节），则在d之后填充w-u字节的无用数据
  - 使数据能够以行宽为单位在内存中排列
  - 否则即使是对于小于行宽的数据，可能需要二次内存访问加上拼接操作才能读取到

局部变量的存储布局

- 变量的类型可以确定变量需要的内存
  - 即类型的**宽度**(该类型一个对象所需的存储单元的数量)
  - 可变大小的数据结构只需要考虑指针
- 函数的局部变量总是分配在连续的区间
  - 因此给每个变量分配一个相对于这个区间开始处的相对地址(偏移量)
- 变量的类型信息保存在符号表中

计算T的类型和宽度的SDT

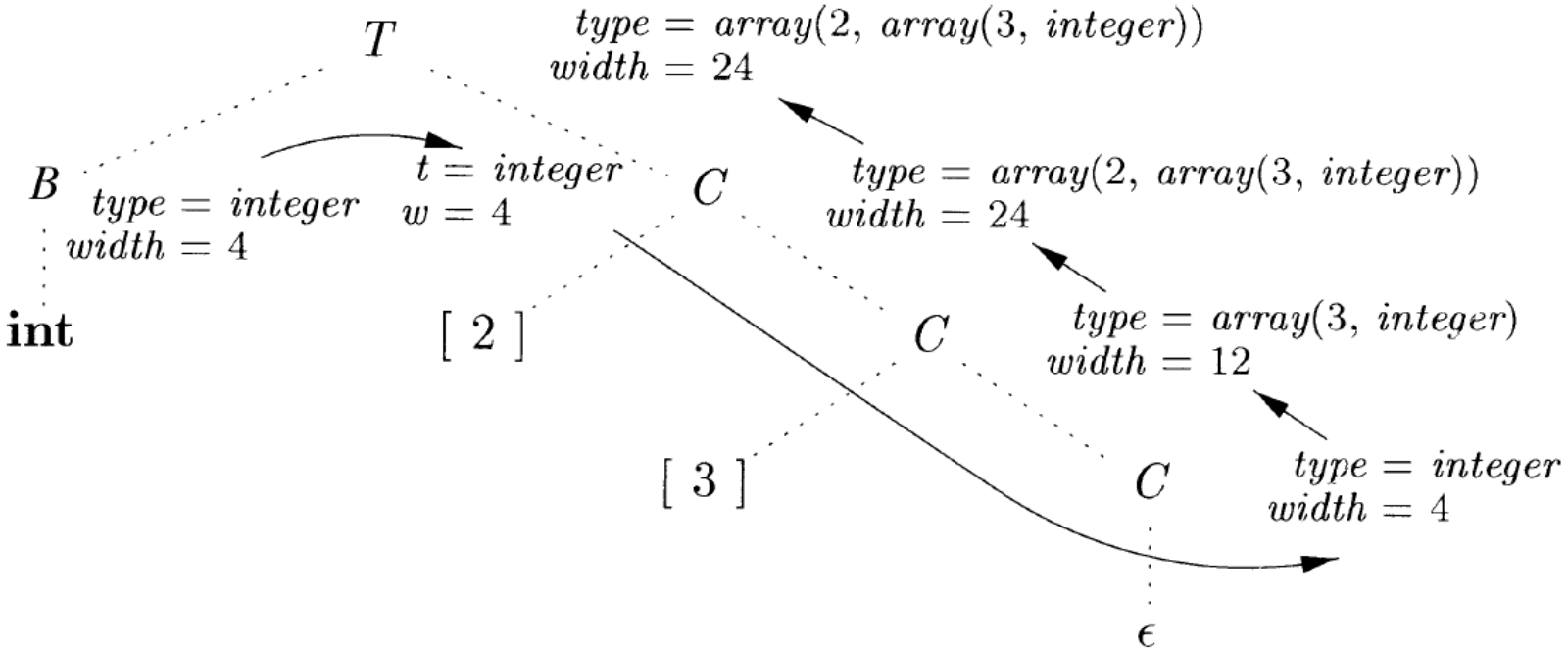
- 综合属性： *type, width*
  - 全局变量*t*和*w*用于将类型和宽度信息从*B*传递到*C* → ε
  - 相当于*C*的继承属性 (也可以把*t*和*w*替换为*C.t*和*C.w*)

$T \rightarrow B$		$\{ t = B.type;\ w = B.width; \}$
	C	$\{T.type{=}C.type;\ T.width{=}C.width; \}$
$B \rightarrow$	**int**	$\{ B.type = integer;\ B.width = 4; \}$
$B \rightarrow$	**float**	$\{ B.type = float;\ B.width = 8; \}$
$C \rightarrow$	ϵ	$\{ C.type = t;\ C.width = w; \}$
$C \rightarrow$	[**num**] C_1	$\{ C.type = array(\mathbf{num}.value,\ C_1.type);$ $C.width = \mathbf{num}.value \times C_1.width; \}$

SDT运行的例子

- 输入： int [2][3]

T	$\rightarrow$	B	$\{ t = B.type;\ w = B.width; \}$
		C	$\{T.type{=}C.type;\ T.width{=}C.width; \}$
B	$\rightarrow$	**int**	$\{ B.type = integer;\ B.width = 4; \}$
B	$\rightarrow$	**float**	$\{ B.type = float;\ B.width = 8; \}$
C	$\rightarrow$	ϵ	$\{ C.type = t;\ C.width = w; \}$
C	$\rightarrow$	[**num**] C_1	$\{ C.type = array(\mathbf{num}.value,\ C_1.type);$ $C.width = \mathbf{num}.value \times C_1.width; \}$



实际的变量声明

产生式规则：$F \rightarrow$ id

- *F.type* = lookupIDTable(**id.*name***)->*type*;

```
int x = 1;

char foo() {
    char x = 'a';
    …
}

float bar(int p) {
    float x = 3.14;
    …
}
```

实际的变量声明

产生式规则：$F \rightarrow id$

- *F.type* = lookupIDTable(**id.*name***)->*type*;

```
int x = 1;

char foo() {
    char x = 'a';
    …
}

float bar(int p) {
    float x = 3.14;
    …
}
```

全局变量符号表	
名字	信息 (类型、偏移…)
x	int
…	…

foo() 符号表	
名字	信息 (类型、偏移…)
x	char
…	…

bar(int) 符号表	
名字	信息 (类型、偏移…)
x	float
…	…

声明序列的SDT (1)

- 在处理一个过程/函数时，局部变量应该放到该过程/函数对应的符号表中
- 这些变量的内存布局独立
  - 相对地址从0开始，变量的放置和声明的顺序相同
- SDT的处理方法
  - 变量*offset*记录当前可用的相对地址
  - 每分配一个变量，*offset*增加相应的值 (加宽度)
- *top.put*(*id.lexeme, T.type, offset*)
  - *top*指向当前函数的符号表
  - 在符号表中创建条目，记录标识符的类型和偏移量

声明序列的SDT (2)

- *top.put*(*id.lexeme, T.type, offset*)
  - *top*指向当前函数的符号表
  - 在符号表中创建条目，记录标识符的类型和偏移量

```
float bar(int p) {
    float x = 3.14;
    int y = 1;
    char z = 'a';
    …
}
```

top →

bar(int) 符号表

名字	类型	偏移
x	float	0
y	int	8
z	char	12
…	…	…

声明序列的SDT (3)

$$
\begin{array}{lll}
P \rightarrow & & \{\ offset = 0;\ \} \\
 & D & \\
D \rightarrow & T\ \mathbf{id}\ ; & \{\ top.put(\mathbf{id}.lexeme,\ T.type,\ offset); \\
 & & \quad offset\ =\ offset + T.width;\ \} \\
 & D_1 & \\
D \rightarrow & \epsilon &
\end{array}
$$

```
float bar() {
    float x;
    int y;
    char z;
    …
}
```

top →

bar() 符号表		
名字	类型	偏移
x	float	0
y	int	8
z	char	12
…	…	…

实际的变量声明

```
float bar(int p) {
    float x;
    record {
        int a;
        char x;
    } r;
    boolean b;
    …
}
```

记录和类中的字段 (1)

- 记录变量声明的翻译方案
- 约定
  - 一个记录中各个字段的名字必须互不相同
  - 字段名的偏移量 (相对地址)，是相对于该记录的数据区字段而言的
- 记录类型使用一个专用的符号表，对其各个字段的类型和相对地址进行单独编码
- 记录类型record(*t*)：record是类型构造算子，*t*是符号表对象，保存该记录类型各个字段的信息

记录和类中的字段 (2)

- 记录类型使用一个专用的符号表，对其各个字段的类型和相对地址进行单独编码
- 记录类型record(*t*)：record是类型构造算子，*t*是符号表对象，保存该记录类型各个字段的信息

```
float bar(int p) {
    float x;
    record {
        int a;
        char x;
    } r;
    boolean b;
    …
}
```

bar(int)::r 符号表		
名字	类型	偏移
a	int	0
x	char	4

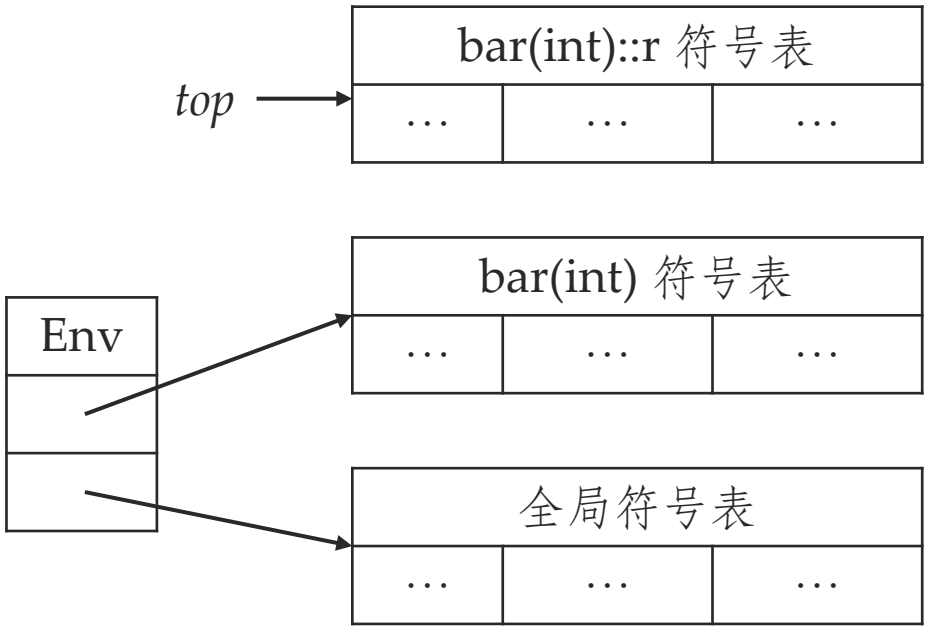
bar(int) 符号表		
名字	类型	偏移
x	float	0
b	boolean	?

Env环境

- 当程序中的作用域发生嵌套时，用一个栈Env辅助维护各作用域对应的符号表，栈中存储指向各符号表的指针
- 进入一个新作用域时，保存上一作用域 (压栈)
- 从一个作用域退出时，恢复上一作用域 (出栈)

```
int x;

float bar(int p) {
    float x;
    record {
        int a;
        char x;
    } r;
    boolean b;
    …
}
```



记录和类中的字段 (3)

$T \rightarrow$ **record** '{' { *Env.push*(*top*); *top* = **new** *Env*(); *Stack.push*(*offset*); *offset* = 0; }

D '}' { *T.type* = *record*(*top*); *T.width* = *offset*; *top* = *Env.pop*(); *offset* = *Stack.pop*(); }

$D \rightarrow T$ **id** ; { *top.put*(**id**.*lexeme*, *T.type*, *offset*); *offset* = *offset* + *T.width*; }

```
float bar(int p) {
    float x;
    record {
        int a;
        char x;
    } r;
    boolean b;
    …
}
```

bar(int)::r 符号表		
名字	类型	偏移
a	int	0
x	char	4

bar(int) 符号表		
名字	类型	偏移
x	float	0
r	record(r)	8
b	boolean	**13**

记录类型存储方式可以推广到类

回顾

- 类型与类型表达式
  - 基本类型 (如integer, float, boolean)
  - 复合类型 (数组、记录、函数)
  - 类型等价性

- 局部变量的存储布局
  - 变量声明语句的SDD
  - 各类型变量存储空间的计算
  - 各变量相对存储位置 (偏移量) 的计算

接下来：生成三地址码语句

表达式代码的SDD

- 将表达式翻译成三地址代码的SDD
  - *code*表示代码
  - *addr*表示存放表达式结果的地址
  - new *Temp*()生成临时变量
  - *gen*()生成指令

产生式	语义规则
$S \rightarrow \mathbf{id} = E\ ;$	$S.code = E.code \,\|\|\, gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \,\|\|\, E_2.code \,\|\|\, gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ - E_1$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \,\|\|\, gen(E.addr\ '='\ '\mathbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \mathbf{id}$	$E.addr = top.get(\mathbf{id}.lexeme)$ $E.code = ''$

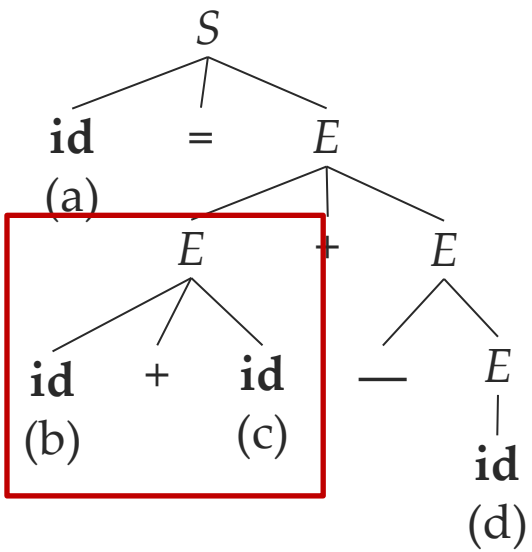
为每一次计算引入一个临时变量存储计算结果

a = b + c + –d

```
t1 = b + c
t2 = minus d
t3 = t1 + t2
a = t3
```

表达式代码的SDD

a = b + c + –d



S.*code*:

产生式	语义规则
$S \rightarrow \mathbf{id} = E\ ;$	$S.code = E.code \mid\mid gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \mid\mid E_2.code \mid\mid gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ - E_1$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \mid\mid gen(E.addr\ '='\ '\mathbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \mathbf{id}$	$E.addr = top.get(\mathbf{id}.lexeme)$ $E.code = ''$

增量式翻译方案

- 类似于上一章中所述的边扫描边生成
- *gen*不仅构造新的三地址指令，还要将它添加到至今为止已生成的指令序列之后
- 不需要*code*指令保存已有的代码，而是对*gen*的连续调用生成一个指令序列

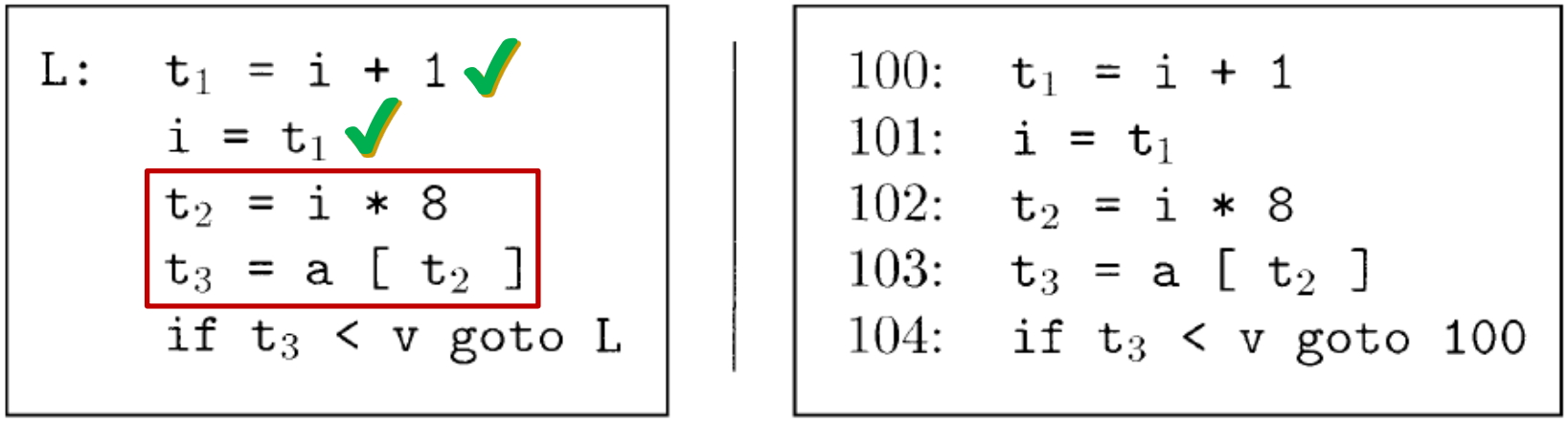
$S \rightarrow$	**id** = E ;	{ *gen*(*top.get*(**id**.*lexeme*) '=' E.*addr*); }
$E \rightarrow$	E_1 + E_2	{ E.*addr* = **new** *Temp*(); *gen*(E.*addr* '=' E_1.*addr* '+' E_2.*addr*); }
|	- E_1	{ E.*addr* = **new** *Temp*(); *gen*(E.*addr* '=' '**minus**' E_1.*addr*); }
|	(E_1)	{ E.*addr* = E_1.*addr*; }
|	**id**	{ E.*addr* = *top.get*(**id**.*lexeme*); }

例子

- 语句
  - do

    i = i + 1;

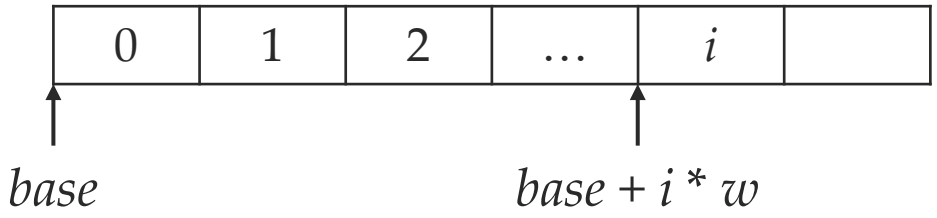
    while (a[i] < v);



a) 符号标号　　b) 位置号

数组元素的寻址 (1)

- 一维数组 ($A[i]$) 寻址：假设数组元素被存放在连续的存储空间中，元素从0到$n-1$编号，第i个元素的地址为
  - $base + i * w$



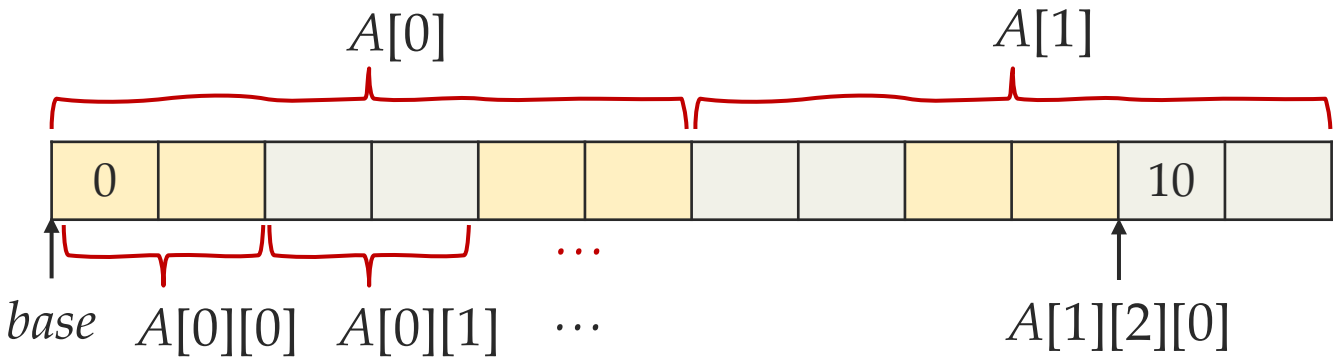
数组元素的寻址 (2)

- k维数组 ($A[i_1][i_2]\ldots[i_k]$)寻址：假设数组按行存放，首先存放$A[0][i_2]\ldots[i_k]$，然后存$A[1][i_2]\ldots[i_k]$，…，那么$A[i_1][i_2]\ldots[i_k]$的地址为
  - $base + i_1 * w_1 + i_2 * w_2 + \ldots + i_k * w_k$
  - 其中$base, w$的值可以从符号表中找到

数组: int A[2][3][2];
寻址: int A[1][2][0];

$base + 1 \times 24 + 2 \times 8 + 0 \times 4 = base + 40$

$w_1 = 3 \times 2 \times |\text{int}| = 24$
$w_2 = 2 \times |\text{int}| = 8$
$w_3 = |\text{int}| = 4$



数组引用的翻译

- 为数组引用生成代码要解决的主要问题
  - 数组引用的文法和地址计算相关联
- 假定数组编号从0开始，基于宽度来计算相对地址

- 数组引用相关文法
  - 非终结符号L生成数组名，加上一个下标表达式序列

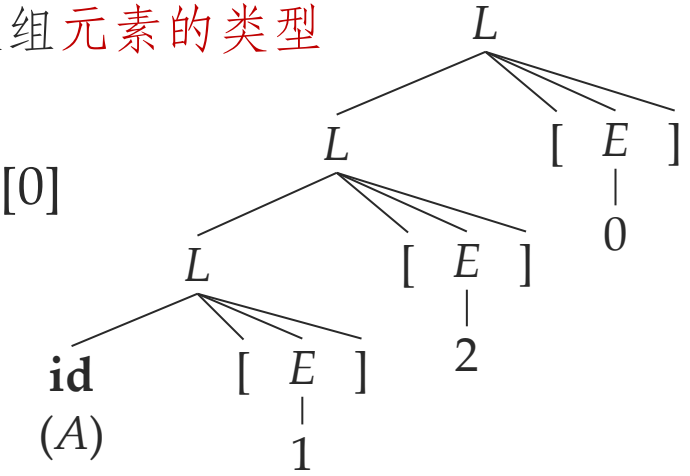
$$L \rightarrow L[E] \mid \mathbf{id}[E]$$

数组引用生成代码的翻译方案 (1)

- 非终结符号L的三个综合属性 $L \rightarrow L[E] \mid \mathbf{id}[E]$
  - *L.array*是一个指向数组名字对应的符号表条目的指针 (*L.array.base*为该数组的基地址)
  - *L.addr*指示一个临时变量，计算数组引用的偏移量

    $base + i_1 * w_1 + i_2 * w_2 + \ldots + i_k * w_k$
  - *L.type*是L生成的子数组的类型
    - 对于任何 (子) 数组类型*L.type*，其宽度由*L.type.width*给出，*L.type.elem*给出其数组元素的类型

$A[1][2][0]$

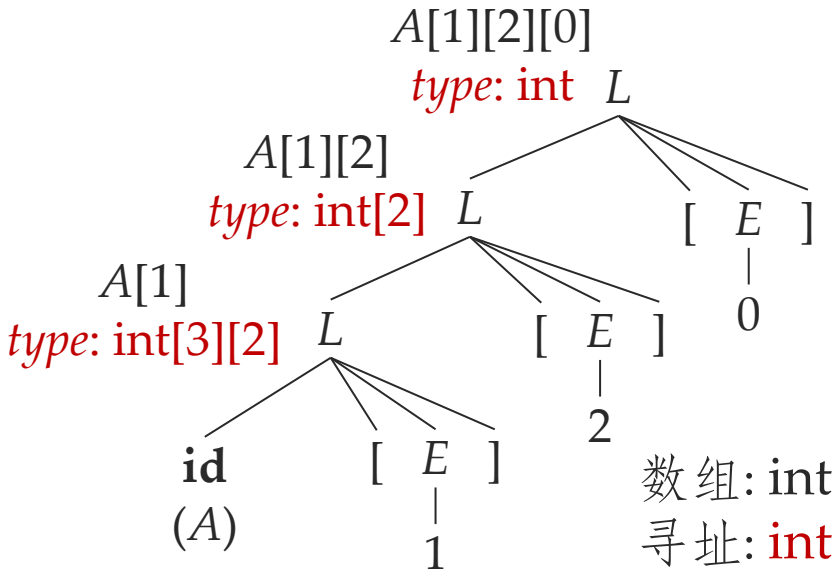


数组引用生成代码的翻译方案 (2)

- 核心是确定数组引用的地址

$$L \rightarrow \mathbf{id}\ [\ E\ ] \quad \{\ L.array = top.get(\mathbf{id}.lexeme);$$
$$L.type = L.array.type.elem;$$
$$L.addr = \mathbf{new}\ Temp\,();$$
$$gen(L.addr\ '='\ E.addr\ '*'\ L.type.width);\ \}$$

$$\mid\ L_1\ [\ E\ ] \quad \{\ L.array = L_1.array;$$
$$L.type = L_1.type.elem;$$
$$t = \mathbf{new}\ Temp\,();$$
$$L.addr = \mathbf{new}\ Temp\,();$$
$$gen(t\ '='\ E.addr\ '*'\ L.type.width);$$
$$gen(L.addr\ '='\ L_1.addr\ '+'\ t);\ \}$$



L.array

数组: int A[2][3][2]
寻址: int A[1][2][0]

符号表		
名字	type	base
...	...	...
A	int[2][3][2]	8

数组引用生成代码的翻译方案 (3)

- L的代码只计算了偏移量
- 数组元素的存放地址应该根据偏移量进一步计算，即L的数组基址加上偏移量
- 使用三地址指令$x = a[i]$

$$
\begin{array}{lll}
E \rightarrow E_1 + E_2 & \{\ E.addr = \textbf{new}\ Temp\,(); \\
 & \quad gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);\ \} \\
\mid\ \textbf{id} & \{\ E.addr = top.get(\textbf{id}.lexeme);\ \} \\
\mid\ L & \{\ E.addr = \textbf{new}\ Temp\,(); \\
 & \quad gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']');\ \}
\end{array}
$$

数组引用生成代码的翻译方案 (4)

- 使用三地址指令$a[i] = x$

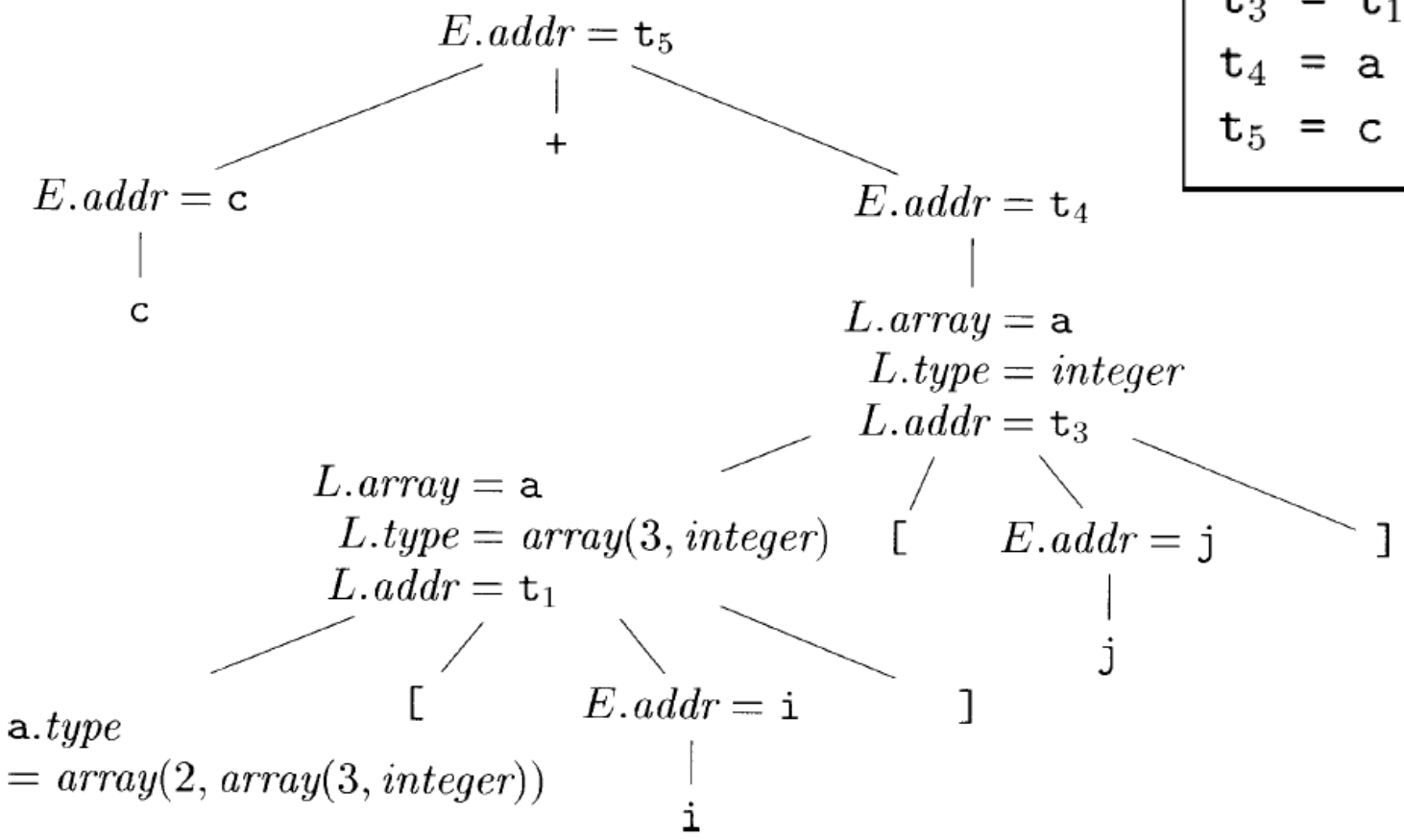
$$S \rightarrow \mathbf{id} = E\ ; \quad \{\ gen(\ top.get(\mathbf{id}.lexeme)\ '='\ E.addr);\ \}$$

$$\mid\ L = E\ ; \quad \{\ gen(L.array\ .base\ '['\ L.addr\ ']'\ '='\ E.addr);\ \}$$

例子

- 表达式：c + a[i][j]

```
t1 = i * 12
t2 = j * 4
t3 = t1 + t2
t4 = a [ t3 ]
t5 = c + t4
```



类型系统 (1)

- 类型系统
  - 给每一个组成部分赋予一个类型表达式
  - 通过一组逻辑规则来表达类型表达式必须满足的条件
  - 可发现错误、提高代码效率、确定临时变量的大小

类型系统 (2)

- 强类型/弱类型：根据类型互操作性/类型系统严格性区分
  - 强类型：不允许无关类型数据间的操作
  - 弱类型：通过隐式类型转换，允许无关类型间的操作

  也有说法按照类型系统严格性进行强弱区分，如C/C++允许通过指针操作绕过常规类型检查，因此被认为是弱类型语言
- 静态类型/动态类型：根据类型检查时机区分 (非绝对)
  - 静态类型：静态 (编译期) 进行类型检查
  - 动态类型：动态 (运行时) 进行类型检查
  - Gradual Typing: 混合，部分静态，部分动态

	静态类型	动态类型
强类型	Java, Haskell, …	Python, Ruby, …
弱类型	C, C++, …	JavaScript, VB, …

反面教材

```
> typeof NaN
"number"
> 9999999999999999
10000000000000000
> 0.5+0.1==0.6
true
> 0.1+0.2==0.3
false
> Math.max()
-Infinity
> Math.min()
Infinity
> []+[]
""
> []+{}
"[object Object]"
> {}+[]
0
> true+true+true===3
true
> true-true
0
```

```
> true==1
true
> true===1
false
> (!+[]+[]+![]).length
9
> 9+"1"
"91"
> 91-"1"
90
> []==0
true
```

https://elbruno.com/2018/07/01/humor-thanks-for-inventing-javascript/

类型规则 (Type Inference Rule)

- 用分式记法表达推导规则

$$\frac{\text{前置条件}}{\text{后置条件}}$$

读作“若前置条件成立，可推出后置条件”

$$\frac{\text{大明是小明的爸爸} \quad \text{老明是大明的爸爸}}{\text{老明是小明的爷爷}}$$

- 类型判决式 (type judgement)

$e: T$

读作“表达式e的类型为T”

$$\frac{e_1: integer \quad e_2: integer}{e_1 + e_2: integer}$$

- 无条件成立的规则

$$\frac{}{0: integer} \qquad \frac{}{\text{true}: boolean}$$

- 简洁 (易读易写)
- 抽象 (通用、独立于具体语言)

标识符的类型规则 (1)

```
void foo(int x, int y) {
    …
    {
        float x, y;
        …
        z = x + y;
    }
    …
}
```

$$\frac{\mathrm{x}: integer?\ float? \qquad \mathrm{y}: integer?\ float?}{\mathrm{x}+\mathrm{y}: integer?\ float?}$$

标识符的类型规则 (2)

- 为了推导标识符相关的类型，需要在类型规则中引入上下文，通常用Γ表示 (也称作类型环境)

$\Gamma \vdash e : T$

读作“在上下文Γ中，表达式e的类型为T”

```
void foo(int x, int y) {
    …
    {
        float x, y;
        …
        z = x + y;
    }
    …
}
```

$$\frac{\Gamma \vdash \mathrm{x} : float \quad \Gamma \vdash \mathrm{y} : float}{\Gamma \vdash \mathrm{x} + \mathrm{y} : float}$$

*top*所指的符号表可用作Γ

函数/运算符的重载 (1)

```
char foo(int x) { … }

boolean foo(float x) { … }

int bar(int p) {
    …
    n = foo(p);
    …
}
```

$$\frac{\text{p: } integer}{\text{foo(p): } char?\ boolean?}$$

函数/运算符的重载 (2)

- 通过查看参数来解决函数重载问题

```
char foo(int x) { … }

boolean foo(float x) { … }

int bar(int p) {
    …
    n = foo(p);
    …
}
```

全局符号表	
名字	类型集合
foo	{int→char, float→boolean}
bar	{int→int}

$$\frac{\Gamma \vdash e : S \qquad \Gamma \vdash \text{foo具有类型} S \to T}{\Gamma \vdash \text{foo}(e) : T}$$

- $E \rightarrow f(E_1)$

{ if $f.typeset = \{ s_i \rightarrow t_i \mid 1 <= i <= n \}$ and $E_1.type = s_k$
then $E.type = t_k$
}

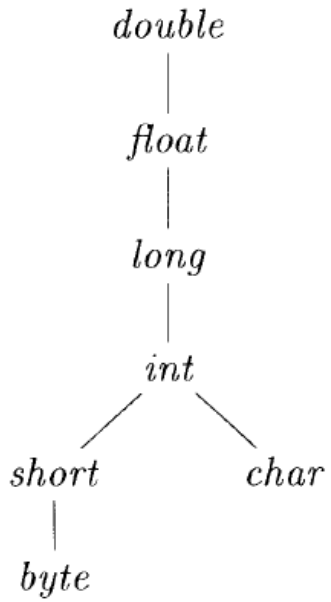
类型转换

- 假设在表达式$x * i$中，x为浮点数，i为整数，则结果应该是浮点数
  - x和i使用不同的二进制表示方式
  - 浮点*和整数*使用不同的指令
  - $t_1 = (\text{float})\ i \quad t_2 = x\ \text{fmul}\ t_1$
- 类型转换比较简单时的SDT
  - $E \rightarrow E_1 + E_2$

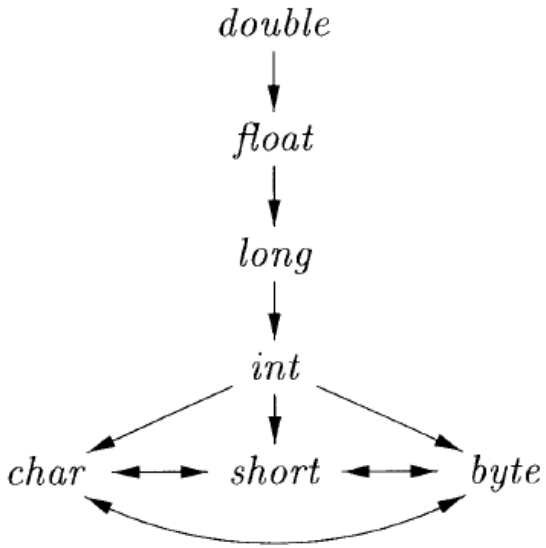
```
{   if (E1.type = integer and E2.type = integer) E.type = integer;
    else if (E1.type = float or E2.type = float) E.type = float;
    …
}
```

类型拓宽widening和窄化narrowing

- Java的类型转换规则
- 编译器自动完成的转换为**隐式转换**，程序员用代码指定的转换为**显式转换**



a) 拓宽类型转换



b) 窄化类型转换

拓宽通常不丢失信息

窄化有可能丢失信息

处理类型转换的SDT

- 函数*max*求两个参数在拓宽层次结构中的最小公共祖先
- 函数*widen*生成必要的类型转换代码

```
Addr widen(Addr a, Type t, Type w)
        if ( t = w ) return a;
        else if ( t = integer and w = float ) {
                temp = new Temp();
                gen(temp '=' '(float)' a);
                return temp;
        }
        else error;
}
```

宽 *op* 宽 → 宽
窄 *op* 窄 → 窄
宽 *op* 窄 / 窄 *op* 宽 → 宽 *op* 宽 →宽

$$
\begin{aligned}
E \rightarrow E_1 + E_2 \quad \{\ & E.type = max(E_1.type, E_2.type); \\
& a_1 = widen(E_1.addr, E_1.type, E.type); \\
& a_2 = widen(E_2.addr, E_2.type, E.type); \\
& E.addr = \textbf{new}\ Temp\,(); \\
& gen(E.addr\ '='\ a_1\ '+'\ a_2);\ \}
\end{aligned}
$$

回顾

- 数据流语句的翻译
  - $S \rightarrow \mathbf{id} = E\ ;$
  - $E \rightarrow E_1 + E_2$　　二元运算 (类型转换)
    - $\mid$ $-E_1$　　一元运算
    - $\mid$ $(\ E_1\ )$　　赋值
    - $\mid$ **id**　　赋值
    - $\mid$ L　　数组访问
  - $L \rightarrow \mathbf{id}\ [\ E\ ]$
    - $\mid$ $L_1\ [\ E\ ]$

接下来：控制流语句的翻译

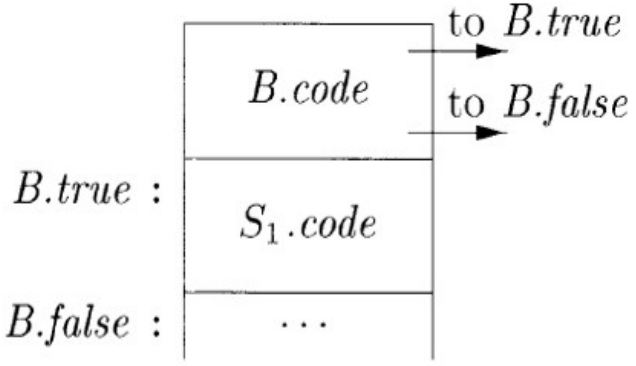
布尔表达式的控制流翻译

- 生成的代码执行时跳转到两个标号之一
  - 表达式的值为真时，跳转到*B.true*
  - 表达式的值为假时，跳转到*B.false*

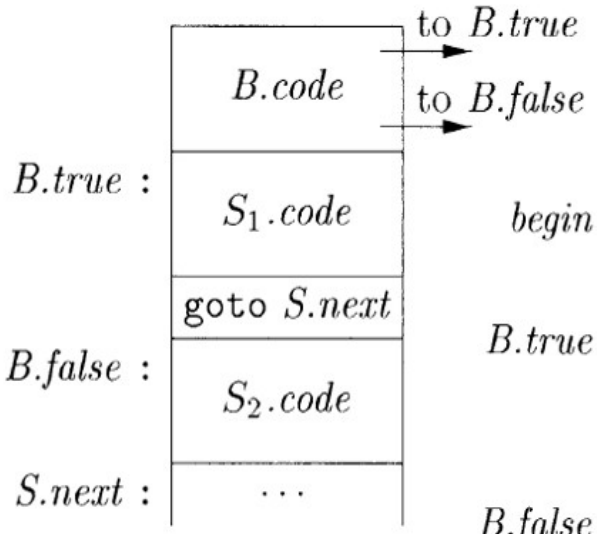
- *B.true*和*B.false*是两个继承属性，根据*B*所在的上下文指向不同的位置
  - 如果*B*是if语句条件表达式，分别指向then和else分支；如果没有else分支，则*B.false*指向if语句的下一条指令
  - 如果*B*是while语句的条件表达式，分别指向循环体的开头和循环出口处

控制流语句的翻译

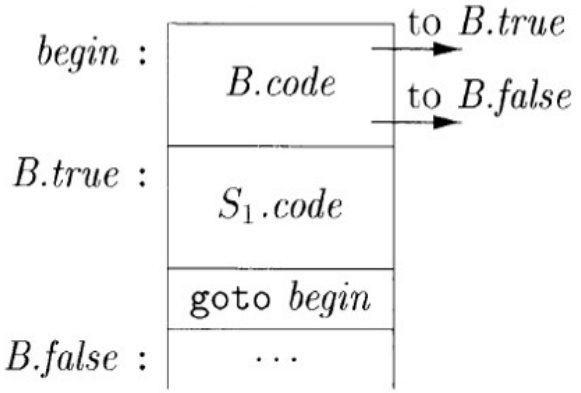
- 控制流语句
  - $S \rightarrow$ **if** $(B)\ S_1$
  - $S \rightarrow$ **if** $(B)\ S_1$ **else** S_2
  - $S \rightarrow$ **while** $(B)\ S_1$
- 继承属性
  - *B.true*：B为真时的跳转目标
  - *B.false*：B为假时的跳转目标
  - *S.next*：S执行完毕时的跳转目标



a) if



b) if-else



c) while

语法制导的定义 (1)

产生式	语义规则
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$
$S \rightarrow$ **assign**	$S.code =$ **assign**$.code$
$S \rightarrow$ **if** (B) S_1	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$
$S \rightarrow$ **if** (B) S_1 **else** S_2	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen('\texttt{goto}'\ S.next)$ $\mid\mid label(B.false) \mid\mid S_2.code$

语法制导的定义 (2)

$S \rightarrow \textbf{while}\ (\ B\ )\ S_1$	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \mid\mid B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen('\texttt{goto}'\ begin)$
$S \rightarrow S_1\ S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \mid\mid label(S_1.next) \mid\mid S_2.code$

- 增量式生成代码

  S → **while (**

  { *begin* = *newlabel*(); *B.true* = *newlabel*(); *B.false* = *S.next*; *gen*(*begin* ':'); }

  B) { S_1.*next* = *begin*; *gen*(*B.true* ':'); } S_1 { *gen*('goto' *begin*); }

布尔表达式的翻译

- 布尔表达式可以用于改变控制流/计算逻辑值
- 文法 $B \rightarrow B \,\|\, B \mid B \,\&\&\, B \mid !\, B \mid (\, B\,) \mid E\ \textbf{rel}\ E \mid \textbf{true} \mid \textbf{false}$
- 语义
  - $B_1 \,\|\, B_2$中B_1为真**或**B_2为真时，整个表达式为真
  - $B_1 \,\&\&\, B_2$中B_1为真**且**B_2为真时，整个表达式为真
- 短路求值
  - 通过跳转指令实现控制流，逻辑运算符本身不出现
  - $B_1 \,\|\, B_2$中B_1为真时，无需计算B_2，整个表达式为真，因此，当B_1为真时应该跳过B_2的代码
  - $B_1 \,\&\&\, B_2$中B_1为假时，无需计算B_2，整个表达式为假，因此，当B_1为假时应该跳过B_2的代码

短路代码的例子

- 语句
  - if (x < 100 ‖ (x > 200 && x != y)) x = 0;

- 代码
  - **if** x < 100 **goto** L_2
  - **if not** x > 200 **goto** L_1
  - **if not** x != y **goto** L_1
  - L_2: x = 0
  - L_1: 接下来的代码

布尔表达式的代码的SDD (1)

产生式	语义规则
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ // 短路 $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \;\&\&\; B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ // 短路 $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$
$B \rightarrow ! B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$

布尔表达式的代码的SDD (2)

$B \rightarrow E_1 \ \mathbf{rel} \ E_2$	$B.code = E_1.code \ \|\| \ E_2.code$ $\|\| \ gen('\texttt{if}' \ E_1.addr \ \mathbf{rel}.op \ E_2.addr \ '\texttt{goto}' \ B.true)$ $\|\| \ gen('\texttt{goto}' \ B.false)$
$B \rightarrow \mathbf{true}$	$B.code = gen('\texttt{goto}' \ B.true)$
$B \rightarrow \mathbf{false}$	$B.code = gen('\texttt{goto}' \ B.false)$

```
if (false)
    x = 0;
y = 1;
```

```
        goto L2
L1:  x = 0;
L2:  y = 1;
```

B → E (隐式类型转换)
if E.addr != 0 goto B.true
goto B.false

x = a || b;

布尔表达式代码的例子

- if (x < 100 ‖ x > 200 && x != y) x = 0; 的代码

```
        if x < 100 goto L2
        goto L3
L3:     if x > 200 goto L4
        goto L1
L4:     if x != y goto L2
        goto L1
L2:     x = 0
L1:
```

生成的中间代码

```
        if x < 100 goto L2
        if not x > 200 goto L1
        if not x != y goto L1
L2:     x = 0
L1:     接下来的代码
```

优化过的中间代码

布尔值和跳转代码 (1)

- 程序中出现布尔表达式也可能是求值：x = a < b
- 处理方法
  - 建立表达式的语法树，根据表达式的不同角色来处理
- 文法
  - $S \rightarrow \text{id} = B; \mid \text{if } (B)\ S \mid \text{while } (B)\ S \mid S\ S$
  - $B \rightarrow B \parallel B \mid B \,\&\&\, B \mid !\, B \mid E \text{ rel } E \mid \ldots$
- 根据B的语法树结点所在的位置
  - $S \rightarrow \text{while } (\ B\ )\ S_1$中的$B$，生成跳转代码
  - 对于$S \rightarrow \text{id} = B$，生成计算右值的代码

布尔值和跳转代码 (2)

- 对于$S \rightarrow \text{id} = B$，生成计算右值的代码
  - *B.true*: id = **true**
  - *B.false*: id = **false**

- 例： x = a < b

		if a < b **goto** L_3
		goto L_2
B.true	L_3:	x = **true**
		goto L_1
B.false	L_2:	x = **false**
	L_1:	接下来的代码

回顾

- 控制流语句的翻译　　标号的运用
  - $S \rightarrow$ **if** $(B)\ S_1$
  - $S \rightarrow$ **if** $(B)\ S_1$ **else** S_2
  - $S \rightarrow$ **while** $(B)\ S_1$

- 布尔表达式的翻译
  - $B \rightarrow B \parallel B \mid B$ **&&** B　　短路求值
  - $\mid\ !B \mid (\ B\ )$
  - $\mid E$ **rel** E
  - $\mid$ **true** $\mid$ **false**

指令的标号与索引

- 标号 (label)
  - 标识程序位置的符号
  - 并**不能**直接定位跳转目标指令
  - 可**提前生成**，合适时插入到相应位置

- 索引 (index)
  - 通常指令用数组存储，指令*i*的索引为其在数组中的下标
  - 可**直接定位**跳转目标指令
  - **无法**提前生成

if a < b **goto** L_2
goto L_1
L_2: x = 2333
L_1: 接下来的代码

if (a < b) x = 2333;

101: **if** a < b **goto** 103
102: **goto** 104
103: x = 2333
104: 接下来的代码

- 标号转换为索引
  - 先生成基于标号的三地址代码
  - 再遍历三地址码将标号替换为对应索引

需遍历两趟，能否**一趟**完成？

一趟完成

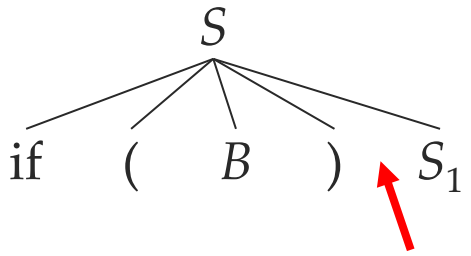
- 基于标号的翻译
  - 继承属性，自顶向下
  - 生成新标号并向下(右)传递，生成跳转语句时填入
  - 经过(生成)跳转目标时插入生成代码中

- 新思路
  - 综合属性，自底向上
  - 经过(生成)跳转目标时，记录其索引
  - 将跳转目标索引填入到相应跳转语句中

此时跳转语句早已生成，所以是“回头”填入跳转目标

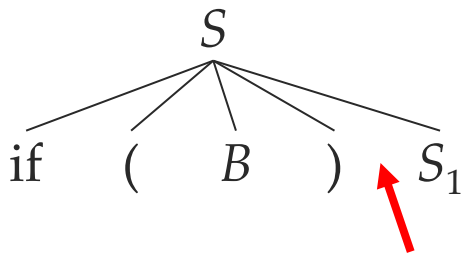
$S \rightarrow$ if (B) S_1

$$
\begin{aligned}
B.true &= newlabel() \\
B.false &= S_1.next = S.next \\
S.code &= B.code \,||\, label(B.true) \,||\, S_1.code
\end{aligned}
$$



回填 (1)

- 为布尔表达式和控制流语句生成目标代码
  - 关键问题：某些跳转指令应该跳转到哪里？
- 基本思想
  - 翻译过程中，若遇到跳转目标未知的情况，则先生成跳转指令坯，备用 goto ___ / if … goto ___
  - 将指令坯并向父结点传递 (综合属性)
  - 翻译过程中，若遇到某条指令是跳转目标，则记录其索引，备用 如 $S \rightarrow$ if (B) ▲ S_1
  - 当父结点收集齐跳转指令坯及其跳转目标索引时，将索引填入指令坯



回填 (2)

- 属性

基于标号		回填 [用索引表示指令 (坯)]	
B.true	*B*为true时跳转目标标号	*B.truelist*	*B*为true时执行的跳转指令
B.false	*B*为false时跳转目标标号	*B.falselist*	*B*为false时执行的跳转指令
S.next	*S*之后下条指令标号	*S.nextlist*	跳转到*S*下条指令的指令

同一列表中跳转指令的目标都相同

- 辅助函数
  - *backpatch*(*p*, *i*)：将*i*作为跳转目标插入*p*的所有指令中

```
if (a < b) x = 2333;
```

```
101:  if a < b goto 103
102:  goto 104
103:  x = 2333
104:  接下来的代码
```

控制转移语句的回填 (1)

- 语句
  - $S \rightarrow$ **if** (B) S | **if** (B) S **else** S | **while** (B) S
  - | { L } | A
  - $L \rightarrow L\ S \mid S$

- 在文法中引入非终结符号 (SDT)
  - M: 在适当的时候获取将要生成指令的索引
  - N: 在适当的位置生成跳转指令坯

控制转移语句的回填 (2)

- M：用$M.instr$记录跳转目标指令的索引
- N：生成goto指令坯，$N.nextlist$包含该指令索引

1) $S \rightarrow \textbf{if}\ (\ B\ )\ M\ S_1$ { $backpatch(B.truelist,\ M.instr)$;
   $S.nextlist\ =\ merge(B.falselist,\ S_1.nextlist)$; }

S → if (B) ▲ S_1

2) $S \rightarrow \textbf{if}\ (\ B\ )\ M_1\ S_1\ N\ \textbf{else}\ M_2\ S_2$
{ $backpatch(B.truelist,\ M_1.instr)$;
$backpatch(B.falselist,\ M_2.instr)$;
$temp\ =\ merge(S_1.nextlist,\ N.nextlist)$;
$S.nextlist\ =\ merge(temp,\ S_2.nextlist)$; }

S → if (B) ▲ S_1 else ▲ S_2

makelist(*i*)：创建一个包含跳转指令 (其索引为*i*) 的列表

6) $M \rightarrow \epsilon$ { $M.instr\ =\ nextinstr$; }

7) $N \rightarrow \epsilon$ { $N.nextlist\ =\ makelist(nextinstr)$;
$gen('\texttt{goto}\ \_')$; }

回填和非回填方法的比较 (1)

1) $S \rightarrow \textbf{if}\ (\ B\ )\ M\ S_1$ { $backpatch(B.truelist,\ M.instr)$;
$S.nextlist\ =\ merge(B.falselist,\ S_1.nextlist)$; }

2) $S \rightarrow \textbf{if}\ (\ B\ )\ M_1\ S_1\ N\ \textbf{else}\ M_2\ S_2$
{ $backpatch(B.truelist,\ M_1.instr)$;
$backpatch(B.falselist,\ M_2.instr)$;
$temp\ =\ merge(S_1.nextlist,\ N.nextlist)$;
$S.nextlist\ =\ merge(temp,\ S_2.nextlist)$; }

S → if (B) ▲ S_1

S → if (B) ▲ S_1 else ▲ S_2

$S \rightarrow \textbf{if}\ (\ B\ )\ S_1$	$B.true\ =\ newlabel()$ $B.false\ =\ S_1.next\ =\ S.next$ $S.code\ =\ B.code \mid\mid label(B.true) \mid\mid S_1.code$
$S \rightarrow \textbf{if}\ (\ B\ )\ S_1\ \textbf{else}\ S_2$	$B.true\ =\ newlabel()$ $B.false\ =\ newlabel()$ $S_1.next\ =\ S_2.next\ =\ S.next$ $S.code\ =\ B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen('\texttt{goto}'\ S.next)$ $\mid\mid label(B.false) \mid\mid S_2.code$

回填获取跳转目标的位置
就是非回填插入标号的位置

控制转移语句的回填 (3)

3) $S \rightarrow$ **while** M_1 (B) M_2 S_1

$S \rightarrow$ while ▲ (B) ▲ S_1

{ $backpatch(S_1.nextlist,\ M_1.instr)$;
$backpatch(B.truelist,\ M_2.instr)$;
$S.nextlist\ =\ B.falselist$;
$gen('\texttt{goto}'\ M_1.instr)$; }

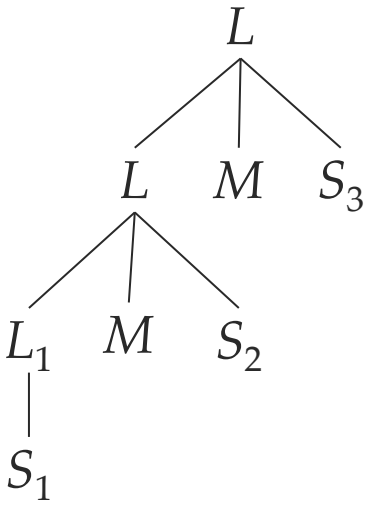
4) $S \rightarrow \{\ L\ \}$ { $S.nextlist\ =\ L.nextlist$; }

5) $S \rightarrow A\ ;$ { $S.nextlist\ =$ **null**; }

8) $L \rightarrow L_1\ M\ S$ { $backpatch(L_1.nextlist,\ M.instr)$;
$L.nextlist\ =\ S.nextlist$; }

$L \rightarrow L_1$ ▲ S

9) $L \rightarrow S$ { $L.nextlist\ =\ S.nextlist$; }



回填和非回填方法的比较 (2)

3) $S \rightarrow$ **while** M_1 (B) M_2 S_1

{ $backpatch(S_1.nextlist,\ M_1.instr)$;
$backpatch(B.truelist,\ M_2.instr)$;
$S.nextlist\ =\ B.falselist$;
$gen('\texttt{goto}'\ M_1.instr)$; }

8) $L \rightarrow L_1\ M\ S$ { $backpatch(L_1.nextlist,\ M.instr)$;
$L.nextlist\ =\ S.nextlist$; }

$S \rightarrow$ **while** (B) S_1	$begin\ =\ newlabel()$ $B.true\ =\ newlabel()$ $B.false\ =\ S.next$ $S_1.next\ =\ begin$ $S.code\ =\ label(begin)\ \|\|\ B.code$ $\|\|\ label(B.true)\ \|\|\ S_1.code$ $\|\|\ gen('\texttt{goto}'\ begin)$
$S \rightarrow S_1\ S_2$	$S_1.next\ =\ newlabel()$ $S_2.next\ =\ S.next$ $S.code\ =\ S_1.code\ \|\|\ label(S_1.next)\ \|\|\ S_2.code$

布尔表达式的回填翻译 (1)

辅助函数

- *makelist*(*i*)：创建一个包含跳转指令 (其索引为*i*) 的列表
- *merge*(*p1*, *p2*)：将*p1*和*p2*指向的索引列表合并然后返回

1)	$B \rightarrow B_1$ `||` M B_2	{ $backpatch(B_1.falselist, M.instr)$; $B.truelist = merge(B_1.truelist, B_2.truelist)$; $B.falselist = B_2.falselist$; }
2)	$B \rightarrow B_1$ `&&` M B_2	{ $backpatch(B_1.truelist, M.instr)$; $B.truelist = B_2.truelist$; $B.falselist = merge(B_1.falselist, B_2.falselist)$; }
3)	$B \rightarrow$ `!` B_1	{ $B.truelist = B_1.falselist$; $B.falselist = B_1.truelist$; }
4)	$B \rightarrow$ `(` B_1 `)`	{ $B.truelist = B_1.truelist$; $B.falselist = B_1.falselist$; }
5)	$B \rightarrow E_1$ **rel** E_2	{ $B.truelist = makelist(nextinstr)$; $B.falselist = makelist(nextinstr + 1)$; $gen('\texttt{if}'\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ '\texttt{goto}\ \_')$; $gen('\texttt{goto}\ \_')$; }
6)	$B \rightarrow$ **true**	{ $B.truelist = makelist(nextinstr)$; $gen('\texttt{goto}\ \_')$; }
7)	$B \rightarrow$ **false**	{ $B.falselist = makelist(nextinstr)$; $gen('\texttt{goto}\ \_')$; }
8)	$M \rightarrow \epsilon$	{ $M.instr = nextinstr$; }

回填和非回填方法的比较 (1)

$B \rightarrow E_1 \ \mathbf{rel} \ E_2$	$B.code = E_1.code \ \|\| \ E_2.code$ $\|\| \ gen('\texttt{if}' \ E_1.addr \ \mathbf{rel}.op \ E_2.addr \ '\texttt{goto}' \ B.true)$ $\|\| \ gen('\texttt{goto}' \ B.false)$

5) $B \rightarrow E_1 \ \mathbf{rel} \ E_2$

$\{ \ B.truelist = makelist(nextinstr);$
$B.falselist = makelist(nextinstr + 1);$
$gen('\texttt{if}' \ E_1.addr \ \mathbf{rel}.op \ E_2.addr \ '\texttt{goto} \ \_');$
$gen('\texttt{goto} \ \_'); \ \}$

- 比较
  - 生成指令坯，然后加入相应的*list*
  - 原来跳转到*B.true*的指令，现在加入到*B.truelist*中
  - 原来跳转到*B.false*的指令，现在加入到*B.falselist*中

回填和非回填方法的比较 (2)

$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$

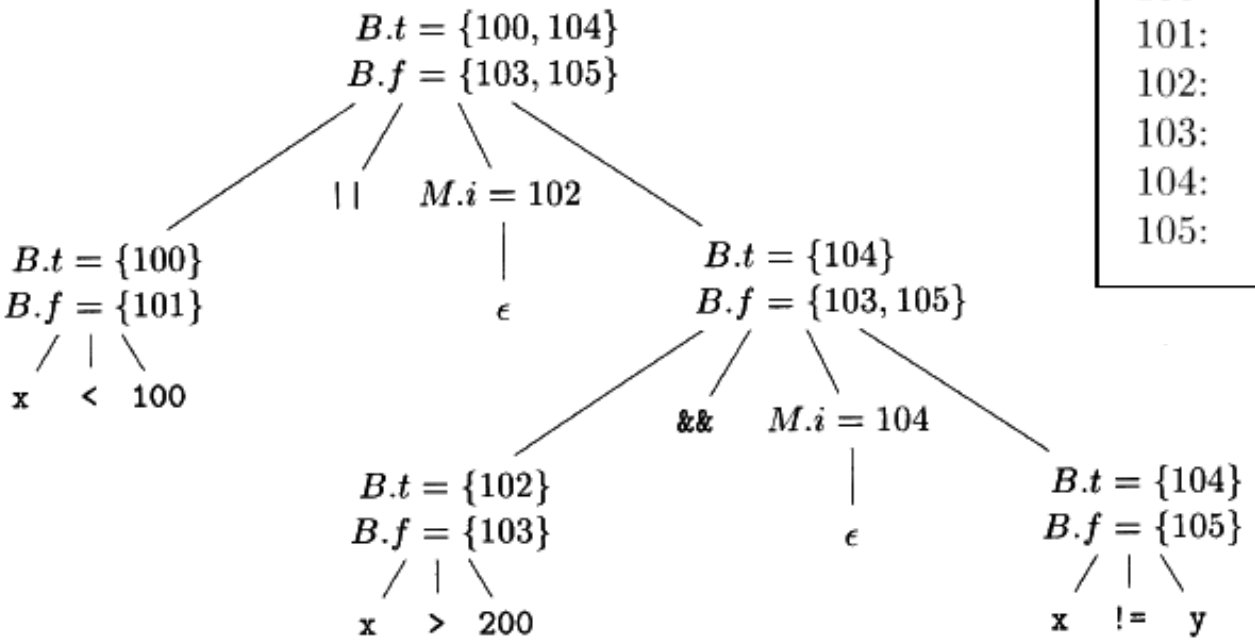
1) $B \rightarrow B_1 \mid\mid M\ B_2$ { $backpatch(B_1.falselist, M.instr)$;
$B.truelist = merge(B_1.truelist, B_2.truelist)$;
$B.falselist = B_2.falselist$; }

- *true/false*属性的赋值，在回填方案中对应为相应的*truelist/falselist*的赋值

布尔表达式的回填例子

- x < 100 || x > 200 && x != y

```
100:  if x < 100 goto _
101:  goto _
102:  if x > 200 goto _
103:  goto _
104:  if x != y goto _
105:  goto _
```




```
100:  if x < 100 goto _
101:  goto 102
102:  if x > 200 goto 104
103:  goto _
104:  if x != y goto _
105:  goto _
```

Break、Continue的处理

- 虽然break、continue在语法上是一个独立的句子，但是它的代码和外围语句相关

- 方法：(break语句)
  - 跟踪外围循环语句*S*
  - 生成一个跳转指令坯
  - 将这个指令坯的索引加入到*S*的*nextlist*中

总结

- 中间代码表示
  - 中间代码表示的作用
  - 三地址代码
- 数据流语句的翻译
  - 变量声明
  - 基本数据类型的运算与数组访问
  - 类型转换
- 控制流语句的翻译
  - 控制流语句
  - 布尔表达式
  - 回填