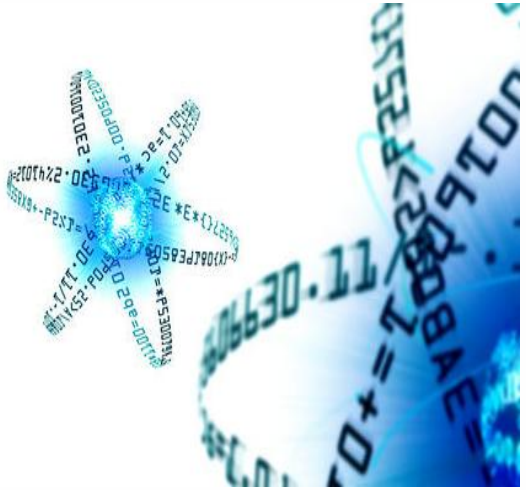
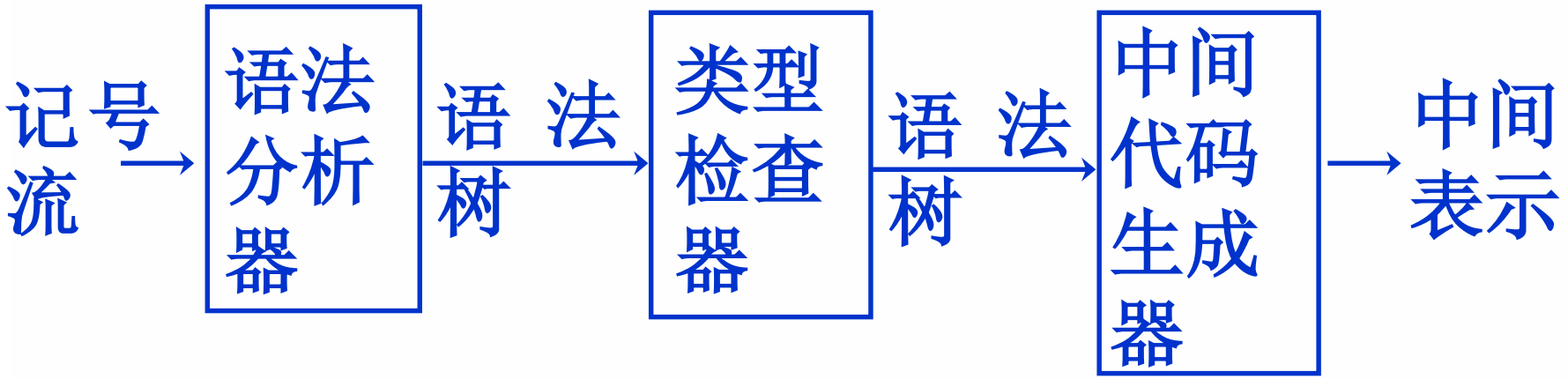
编译原理

Compilers – Principles, Techniques and Tools



第六章 类 型 检 查



本章内容

- 静态检查中最典型的部分 — 类型检查
  - 设计语法制导的类型检查器
- 忽略其它的静态检查：控制流检查、唯一性检查、关联名字检查

简单类型检查器的说明

类型检查——声明语句

$D \rightarrow D ; D$

$D \rightarrow \text{id} : T$ {*addtype* (id.*entry*, *T.type*)}

addtype：把类型信息填入符号表

简单类型检查器的说明

类型检查——声明语句

$D \rightarrow D ; D$

$D \rightarrow \text{id} : T \ \{addtype(\text{id}.entry, T.type)\}$

$T \rightarrow \text{boolean} \quad \{T.type = boolean\}$

$T \rightarrow \text{integer} \quad \{T.type = integer\}$

$T \rightarrow \uparrow T_1 \quad \{T.type = pointer(T_1.type)\}$

简单类型检查器的说明

类型检查——声明语句

$D \rightarrow D ; D$

$D \rightarrow \text{id} : T \quad \{addtype(\text{id}.entry, T.type)\}$

$T \rightarrow \text{boolean} \quad \{T.type = boolean\}$

$T \rightarrow \text{integer} \quad \{T.type = integer\}$

$T \rightarrow \uparrow T_1 \quad \{T.type = pointer(T_1.type)\}$

$T \rightarrow \text{array [num] of } T_1$

$\{T.type = array(\text{num}.val, T_1.type)\}$

简单类型检查器的说明

类型检查——声明语句

$D \rightarrow D ; D$

$D \rightarrow \text{id} : T \quad \{addtype(\text{id}.entry, T.type)\}$

$T \rightarrow \text{boolean} \quad \{T.type = boolean\}$

$T \rightarrow \text{integer} \quad \{T.type = integer\}$

$T \rightarrow \uparrow T_1 \quad \{T.type = pointer(T_1.type)\}$

$T \rightarrow \text{array [num] of } T_1$

$\{T.type = array(\text{num}.val, T_1.type)\}$

$T \rightarrow T_1 \text{ ‘}\rightarrow\text{’ } T_2 \quad \{T.type = T_1.type \rightarrow T_2.type\}$

简单类型检查器的说明

类型检查——表达式

$E \rightarrow$ truth　　{*E.type* = *boolean* }

$E \rightarrow$ num　　{*E.type* = *integer*}

$E \rightarrow$ id　　{*E.type* = *lookup*(id.*entry*)}

简单类型检查器的说明

类型检查——表达式

$E \rightarrow$ truth {*E.type* = *boolean* }

$E \rightarrow$ num {*E.type* = *integer*}

$E \rightarrow$ id {*E.type* = *lookup*(id.*entry*)}

$E \rightarrow E_1$ mod E_2

{*E.type* = if E_1.*type* == *integer* and

E_2. *type* == *integer* then *integer*

else *type_error* }

简单类型检查器的说明

类型检查——表达式

$E \rightarrow E_1 [E_2]$ {*E.type* = if E_2. *type* == *integer* and E_1. *type* == *array*(*s*, *t*) then *t* else *type_error* }

简单类型检查器的说明

类型检查——表达式

$E \rightarrow E_1\,[E_2\,]$ {*E.type* = if E_2. *type* == *integer* and E_1. *type* == *array*(*s*, *t*) then *t* else *type_error* }

$E \rightarrow E_1\uparrow$ {*E.type* = if E_1.*type* == *pointer*(*t*) then *t* else *type_error* }

简单类型检查器的说明

类型检查——表达式

$E \rightarrow E_1\,[E_2\,]$ {*E.type* = if E_2. *type* == *integer* and
E_1. *type* == *array*(*s*, *t*) then *t*
else *type_error* }

$E \rightarrow E_1\uparrow$ {*E.type* = if E_1.*type* == *pointer*(*t*) then *t*
else *type_error* }

$E \rightarrow E_1\,(E_2\,)$ {*E*. *type* = if E_2 . *type* == *s* and
E_1. *type* == $s \rightarrow t$ then *t*
else *type_error* }

简单类型检查器的说明

类型检查——语句

$S \rightarrow$ id := E { if (*id.type* == *E.type* && *E.type* $\in$ {*boolean*, *integer*}) *S.type* = *void*;

else *S.type* = *type_error*;}

简单类型检查器的说明

类型检查——语句

$S \rightarrow$ id := E { if (*id.type* == *E.type* && *E.type* $\in$ {*boolean*, *integer*}) *S.type* = *void*;

else *S.type* = *type_error*;}

$S \rightarrow$ if E then S_1 {*S. type* = if *E. type* == *boolean*

then S_1. *type*

else *type_error* }

简单类型检查器的说明

类型检查——语句

$S \rightarrow$ **while** E **do** S_1

{*S.type* = **if** *E.type* == *boolean* **then** S_1. *type*

else *type_error* }

简单类型检查器的说明

类型检查——语句

$S \rightarrow$ while E do S_1

{$S.type$ = if $E.type$ == $boolean$ then $S_1.type$
else $type\_error$ }

$S \rightarrow S_1; S_2$

{$S.type$ = if $S_1.type$ == $void$ and
$S_2.type$ == $void$ then $void$
else $type\_error$ }

简单类型检查器的说明

类型检查——程序

$P \rightarrow D; S$

{*P*. *type* = if *S*. *type* == *void* then *void*
else *type_error* }

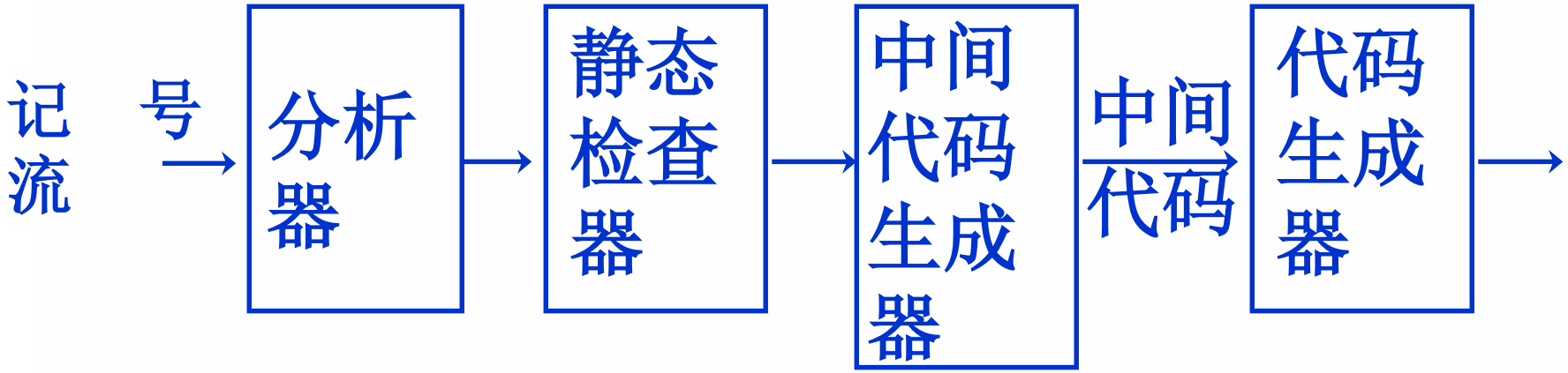
简单类型检查器的说明

类型转换

$E \rightarrow E_1$ op E_2

{*E.type* = if E_1.*type* == *integer* and E_2.*type* == *integer*
then *integer*
else if E_1.*type* == *integer* and E_2.*type* == *real*
then *real*
else if E_1.*type* == *real* and E_2.*type* == *integer*
then *real*
else if E_1.*type* == *real* and E_2.*type* == *real*
then *real*
else *type_error* }

第七章　中间代码生成



本章内容

- 介绍几种常用的中间表示：后缀表示、图形表示和三地址代码
- 用语法制导定义和翻译方案来说明源语言的各种构造怎样被翻译成中间形式

7.1 中 间 语 言

7.1.1 后缀表示

$E \rightarrow E\ op\ E \mid uop\ E \mid (E) \mid id \mid num$

表达式*E*的后缀表示可以如下归纳定义：

表达式*E*	后缀式*E′*
id	id
num	num
$E_1\ op\ E_2$	$E_1'\ E_2'\ op$
$uop\ E$	$E'\ uop$
(E)	E'

7.1 中 间 语 言

- 后缀表示不需要括号

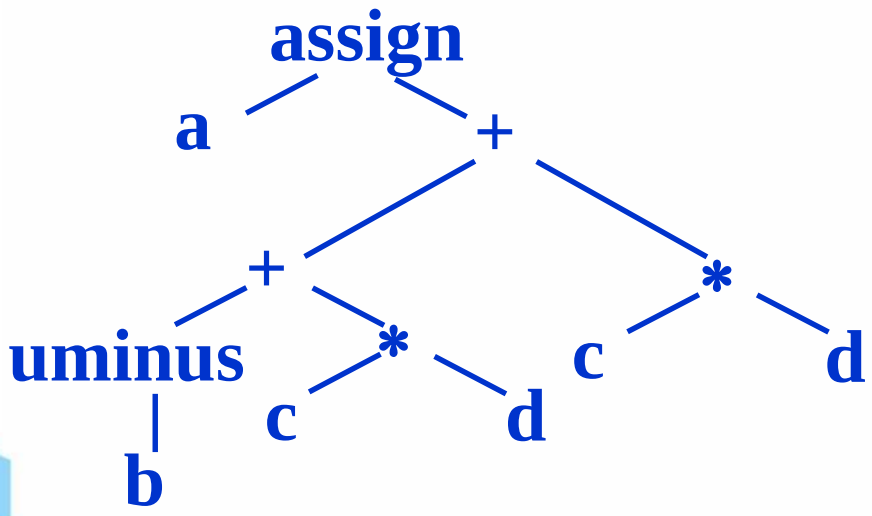
  (8 – 5) + 2 的后缀表示是8 5 –2 +

- 后缀表示的最大优点是便于计算机处理表达式

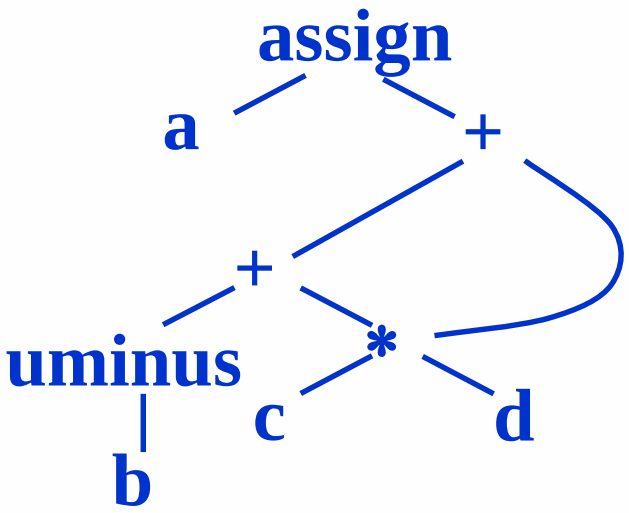
计算栈	输入串
	8 5 –2 +
8	5 –2 +
8 5	–2 +
3	2 +
3 2	+
5	

7.1.2 图形表示

- 语法树是一种图形化的中间表示
- 有向无环图也是一种中间表示



(a) 语法树



(b) DAG

a = (–b + c*d) + c*d的图形表示

7.1 中 间 语 言

构造赋值语句语法树的语法制导定义

产 生 式	语 义 规 则
$S \rightarrow \text{id} = E$	$S.nptr = mkNode(\text{'assign'}, mkLeaf(\text{id}, \text{id}.entry), E.nptr)$
$E \rightarrow E_1 + E_2$	$E.nptr = mkNode(\text{'+'}, E_1.nptr, E_2.nptr)$
$E \rightarrow E_1 * E_2$	$E.nptr = mkNode(\text{'*'}, E_1.nptr, E_2.nptr)$
$E \rightarrow -E_1$	$E.nptr = mkUNode(\text{'uminus'}, E_1.nptr)$
$E \rightarrow (E_1)$	$E.nptr = E_1.nptr$
$F \rightarrow \text{id}$	$E.nptr = mkLeaf(\text{id}, \text{id}.entry)$

7.1 中 间 语 言

7.1.3 三地址代码

一般形式：$x = y\ op\ z$

- 例 表达式$x + y * z$翻译成的三地址语句序列是

  $t_1 = y * z$

  $t_2 = x + t_1$

7.1 中 间 语 言

- 三地址代码是语法树或DAG的一种线性表示
- 例 a = (–b + c*d) + c*d

  语法树的代码

$t_1 = -b$

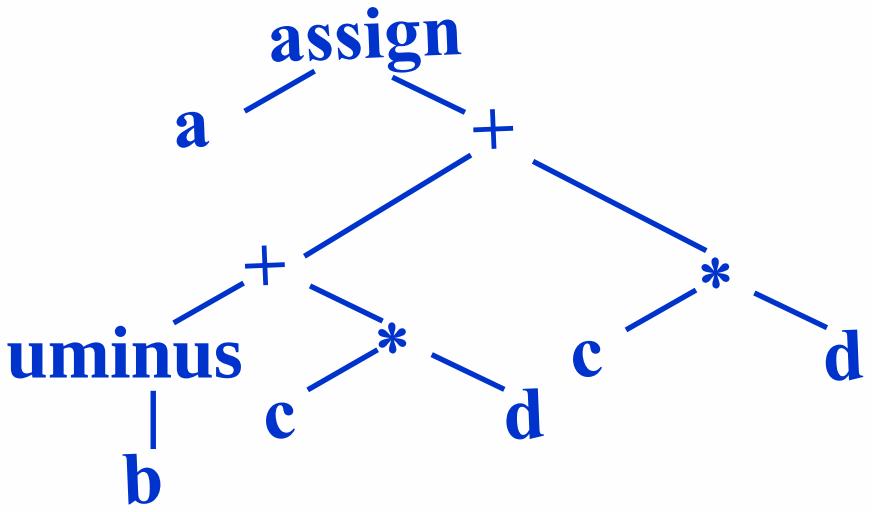
$t_2 = c * d$

$t_3 = t_1 + t_2$

$t_4 = c * d$

$t_5 = t_3 + t_4$

$a = t_5$



7.1 中 间 语 言

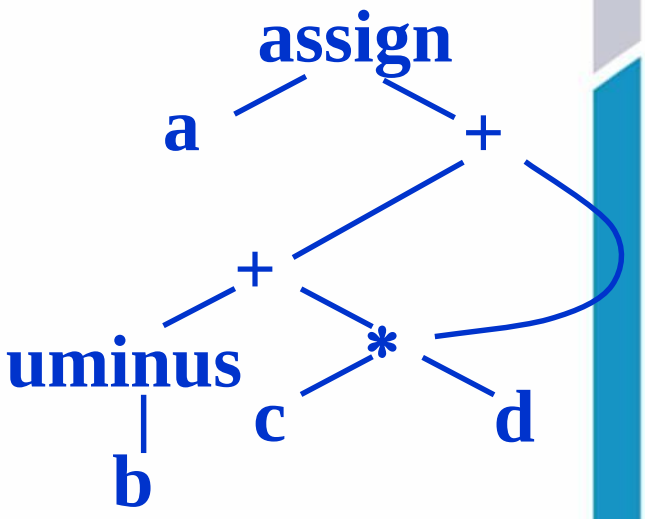
- 三地址代码是语法树或DAG的一种线性表示
- 例 a = (–b + c*d) + c*d

语法树的代码

$t_1 = -b$
$t_2 = c * d$
$t_3 = t_1 + t_2$
$t_4 = c * d$
$t_5 = t_3 + t_4$
$a = t_5$

DAG的代码

$t_1 = -b$
$t_2 = c * d$
$t_3 = t_1 + t_2$
$t_4 = t_3 + t_2$
$a = t_4$



7.1 中 间 语 言

本书常用的三地址语句

- 赋值语句$x = y\ op\ z$，　$x = op\ y$，　$x = y$
- 无条件转移goto L
- 条件转移if x *relop* y goto L
- 过程调用param x 和call p , n
- 过程返回 return y
- 索引赋值$x = y[i]$和 $x[i] = y$
- 地址和指针赋值$x = \&y$，$x = *y$和$*x = y$

7.2 声 明 语 句

本节介绍

- 为局部名字建立符号表条目
- 为它分配存储单元
- 符号表中包含名字的类型和分配给它的存储单元的相对地址等信息

7.2 声 明 语 句

7.2.1 过程中的声明

7.2 声 明 语 句

计算被声明名字的类型和相对地址

$P \rightarrow$ {*offset* = 0} D; S

$D \rightarrow D$; D

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

$T \rightarrow$ integer {T.*type* = *integer*; T.*width* = 4 }

$T \rightarrow$ real {T.*type* = *real*; T.*width* = 8 }

$T \rightarrow$ array [num] of T_1
{T.*type* = *array* (num.*val*, T_1.*type*);
T.width = num.*val* $\times$ T_1.width}

$T \rightarrow \uparrow T_1$ {T.*type* = *pointer* (T_1.*type*); T.*width* = 4 }

7.2 声 明 语 句

计算被声明名字的类型和相对地址

$P \rightarrow$ {*offset* = 0} *D*; *S*

$D \rightarrow D$; D

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

$T \rightarrow$ integer {*T*.*type* = *integer*; *T*.*width* = 4 }

$T \rightarrow$ real {*T*.*type* = *real*; *T*.*width* = 8 }

$T \rightarrow$ array [num] of T_1
{*T*.*type* = *array* (num.*val*, T_1.*type*);
T.width = num.*val* $\times$ T_1.width}

$T \rightarrow \uparrow T_1$ {*T*.*type* = *pointer* (T_1.*type*); *T*.*width* = 4 }

7.2 声 明 语 句

7.2.2 作用域信息的保存

- 所讨论语言的文法

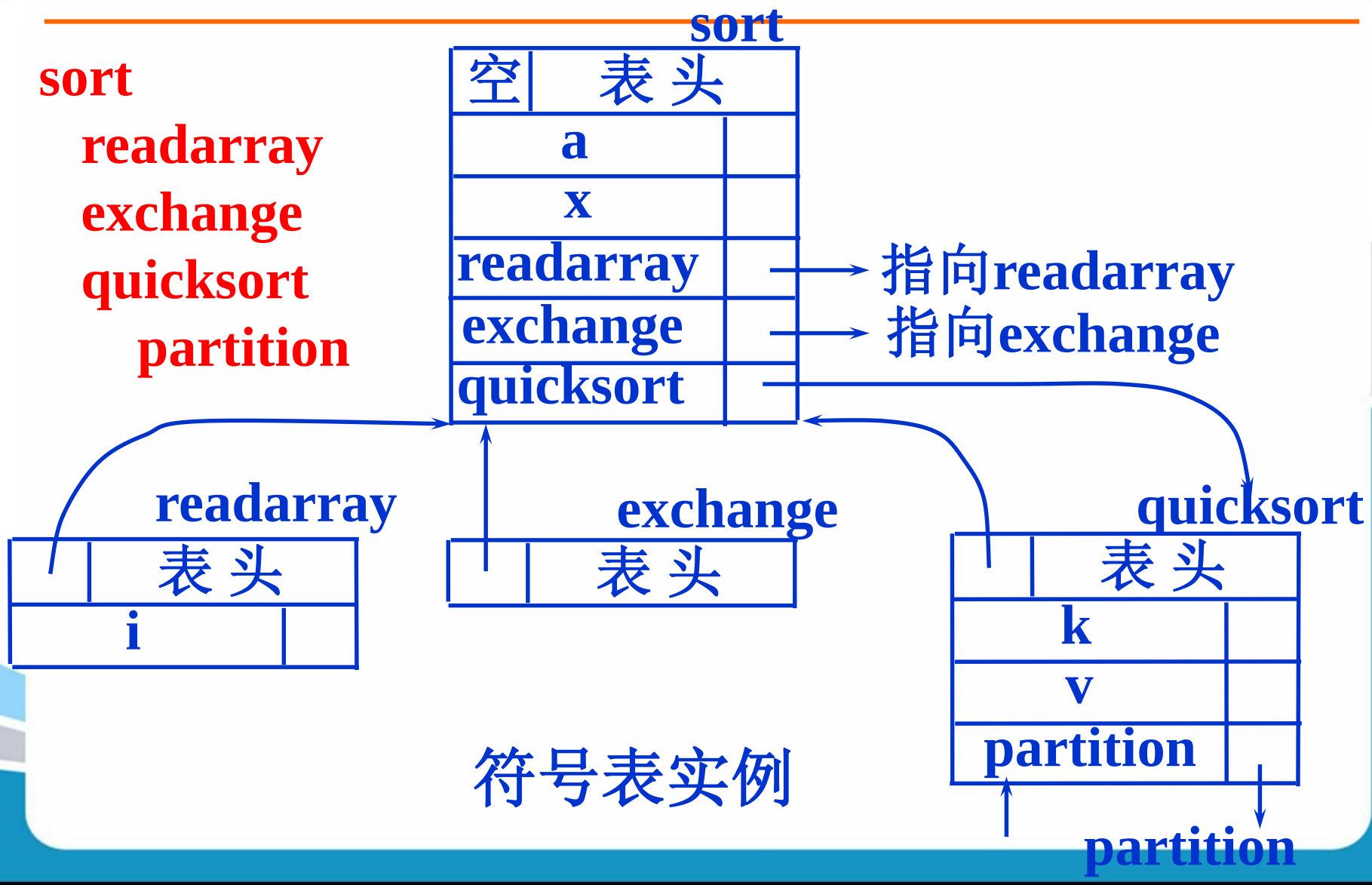
$P \rightarrow D; S$

$D \rightarrow D ; D \mid \text{id} : T \mid$

proc id ; D ; S

```
sort
    var a:…; x:…;
    readarray
        var i:…;
    exchange
    quicksort
        var k, v:…;
        partition
            var i, j:…;
```

7.2 声 明 语 句



符号表实例

7.2 声 明 语 句

- 符号表的特点
  - 各过程有各自的符号表
  - 符号表之间有双向链
  - 构造符号表时需要符号表栈
  - 构造符号表需要活动记录栈
- 语义动作用到的函数

  mkTable(previous)

  enter(table, name, type, offset)

  addWidth(table, width)

  enterProc(table, name, newtable)

```
sort
    var a:…; x:…;
    readarray
        var i:…;
    exchange
    quicksort
        var k, v:…;
        partition
            var i, j:…;
```

7.2 声 明 语 句

$P \rightarrow M\ D;\ S$ {*addWidth* (*top* (*tblptr*), *top* (*offset*));
pop(*tblptr*); *pop* (*offset*) }

$M \rightarrow \varepsilon$ {*t* = *mkTable* (*nil*);
push(*t*, *tblprt*); *push* (0, *offset*) }

$D \rightarrow D_1\ ;\ D_2$

$D \rightarrow$ proc id ; $N\ D_1$; S {*t* = *top*(*tblptr*);
addWidth(*t*, *top*(*offset*)); *pop*(*tblptr*); *pop*(*offset*);
enterProc(*top*(*tblptr*), id.*lexeme*, *t*) }

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

$N \rightarrow \varepsilon$ {*t* = *mkTable*(*top*(*tblptr*));
push(*t*, *tblptr*); *push*(0, *offset*) }

7.2 声 明 语 句

$P \rightarrow M\ D;\ S$ {*addWidth* (*top* (*tblptr*), *top* (*offset*));
pop(*tblptr*); *pop* (*offset*) }

$M \rightarrow \varepsilon$ {*t* = *mkTable* (*nil*);
push(*t*, *tblprt*); *push* (0, *offset*) }

$D \rightarrow D_1\ ;\ D_2$

$D \rightarrow$ proc id ; $N\ D_1;\ S$ {*t* = *top*(*tblptr*);
addWidth(*t*, *top*(*offset*)); *pop*(*tblptr*); *pop*(*offset*);
enterProc(*top*(*tblptr*), id.*lexeme*, *t*) }

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

$N \rightarrow \varepsilon$ {*t* = *mkTable*(*top*(*tblptr*));
push(*t*, *tblptr*); *push*(0, *offset*) }

7.2 声 明 语 句

$P \rightarrow M\ D;\ S$ {*addWidth* (*top* (*tblptr*), *top* (*offset*));
pop(*tblptr*); *pop* (*offset*) }

$M \rightarrow \varepsilon$ {*t* = *mkTable* (*nil*);
push(*t*, *tblprt*); *push* (0, *offset*) }

$D \rightarrow D_1\ ;\ D_2$

$D \rightarrow$ proc id ; $N\ D_1$; S {*t* = *top*(*tblptr*);
addWidth(*t*, *top*(*offset*)); *pop*(*tblptr*); *pop*(*offset*);
enterProc(*top*(*tblptr*), id.*lexeme*, *t*) }

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

$N \rightarrow \varepsilon$ {*t* = *mkTable*(*top*(*tblptr*));
push(*t*, *tblptr*); *push*(0, *offset*) }

7.3 赋 值 语 句

7.3.1 符号表中的名字

$S \rightarrow$ id := E {*p* = *lookup*(id.*lexeme*);
if *p* != *nil* then
emit (*p*, '=', *E.place*)
else *error* }

$E \rightarrow E_1 + E_2$
{*E.place* = *newTemp*();
emit (*E.place*, '=', E_1.*place*, '+', E_2.*place*) }

7.3 赋 值 语 句

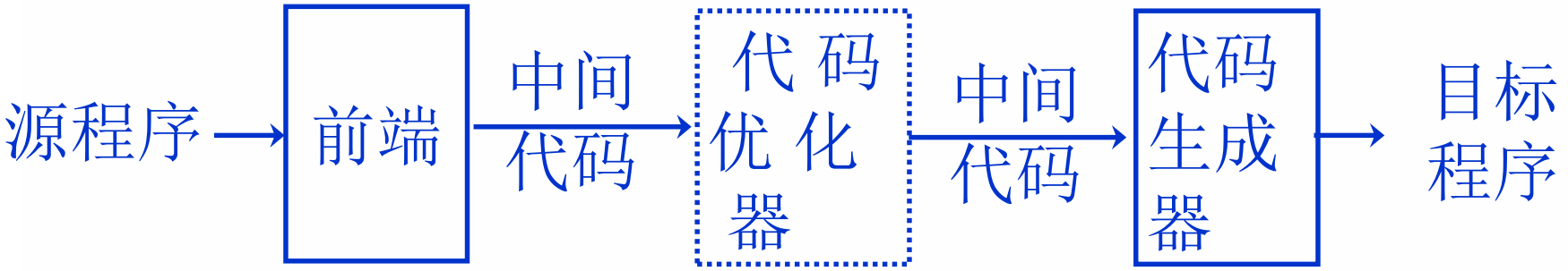
7.3.1 符号表中的名字

$E \rightarrow -E_1$ {*E.place* = *newTemp*();
emit (*E.place*, '=', 'uminus', E_1.*place*) }

$E \rightarrow (E_1)$ {*E.place* = E_1.*place* }

$E \rightarrow$ id {*p* = *lookup*(id.*lexeme*);
if *p* != *nil* then *E.place* = *p* else *error* }

第八章 代 码 生 成



本章内容

- 一个简单的代码生成算法
- 涉及存储管理，指令选择，寄存器分配和计算次序选择等基本问题

8.1 代码生成器的设计中的问题

8.1.1 目标程序

- 绝对机器语言程序
  - 目标程序将装入到内存的固定地方
- 可重定位目标模块
  - 代码中含重定位信息，以适应重定位要求

8.1 代码生成器的设计中的问题

8.1.1 目标程序

- 绝对机器语言程序
  - 目标程序将装入到内存的固定地方
- 可重定位目标模块
  - 代码中含重定位信息，以适应重定位要求
  - 允许程序模块分别编译
  - 调用其它先前编译好的程序模块

8.1 代码生成器的设计中的问题

8.1.1 目标程序

- 绝对机器语言程序
- 可重定位目标模块
  - 代码中含重定位信息，以适应重定位要求
  - 允许程序模块分别编译
  - 调用其它先前编译好的程序模块
- 汇编语言程序
  - 免去编译器重复汇编器的工作
  - 从教学角度，增加可读性

8.1 代码生成器的设计中的问题

8.1.2 指令的选择

- 目标机器指令系统的性质决定了指令选择的难易程度，指令系统的统一性和完备性是重要的因素
- 指令的速度和机器特点是另一些重要的因素

8.1 代码生成器的设计中的问题

- 若不考虑目标程序的效率，指令的选择是直截了当的

- 例　三地址语句x = y + z (x，y和z都静态分配)

```
MOV     y,     R0     /* 把y装入寄存器R0 */
ADD     z,     R0     /* 把z加到R0上 */
MOV     R0,    x      /* 把R0存入x中 */
```

逐条语句地产生代码，常常得到低质量的代码

8.1 代码生成器的设计中的问题

语句序列 a = b + c

d = a + e

的代码如下

```
MOV     b,   R0
ADD     c,   R0
MOV     R0,  a
MOV     a,   R0
ADD     e,   R0
MOV     R0,  d
```

8.1 代码生成器的设计中的问题

语句序列 a = b + c

d = a + e

的代码如下

```
MOV     b,   R0
ADD     c,   R0
MOV     R0,  a
MOV     a,   R0    —— 多余的指令
ADD     e,   R0
MOV     R0,  d
```

8.1 代码生成器的设计中的问题

语句序列 a = b + c

d = a + e

的代码如下

```
MOV     b,   R0
ADD     c,   R0
MOV     R0,  a
MOV     a,   R0    —— 多余的指令
ADD     e,   R0    —— 若a不再使用，
MOV     R0,  d     —— 第三条指令也多余
```

8.1 代码生成器的设计中的问题

8.1.3 寄存器分配

运算对象处于寄存器和处于内存相比，指令要短一些，执行也快一些

- 寄存器分配

  选择驻留在寄存器中的一组变量
- 寄存器指派

  挑选变量要驻留的具体寄存器

8.2 目 标 机 器

8.2.1 目标机器的指令系统

- 选择可作为几种微机代表的寄存器机器
- 四个字节组成一个字，有*n*个通用寄存器
  R0, R1, …, Rn-1
- 二地址指令： **op 源，目的**

  MOV {源传到目的}

  ADD {源加到目的}

  SUB {目的减去源}

8.2 目 标 机 器

- 地址模式和它们的汇编语言形式及附加代价

模式	形式	地址	附加代价
绝对地址	M	M	1
寄存器	R	R	0
变址	*c*(R)	*c* + *contents*(R)	1
间接寄存器	*R	*contents*(R)	0
间接变址	**c*(R)	*contents*(*c* + *contents*(*R*))	1
直接量	#*c*	*c*	1

8.2.2 指令的代价

- 指令代价简化为

  1 + 指令的源和目的地址模式的附加代价

指令	代价
MOV R0，R1	1
MOV R5，M	2
ADD #1，R3	2
SUB 4(R0), *12(R1)	3

8.3 基本块和流图

- 怎样为三地址语句序列生成目标代码？

先给出本节用例

```
prod = 0;
i = 1;
do {
    prod = prod + a[i] * b[i];
    i = i +1;
} while (i <= 20);
```

其三地址代码见右边

(1)prod = 0
(2) i = 1
(3) $t_1 = 4 * i$
(4) $t_2 = a[t_1]$
(5) $t_3 = 4 * i$
(6) $t_4 = b[t_3]$
(7) $t_5 = t_2 * t_4$
(8) $t_6 = prod + t_5$
(9) $prod = t_6$
(10) $t_7 = i + 1$
(11) $i = t_7$
(12) if i <= 20 goto (3)

8.3 基本块和流图

8.3.1 基本块

- 基本块
  - 连续的语句序列，控制流从它的开始进入，并从它的末尾离开

- 流图
  - 再用有向边表示基本块之间的控制流信息，就能得到程序的流图

(1)prod = 0
(2) i = 1
(3) $t_1 = 4 * i$
(4) $t_2 = a[t_1]$
(5) $t_3 = 4 * i$
(6) $t_4 = b[t_3]$
(7) $t_5 = t_2 * t_4$
(8) $t_6 = prod + t_5$
(9) $prod = t_6$
(10) $t_7 = i + 1$
(11) $i = t_7$
(12) if i <= 20 goto (3)

8.3 基本块和流图

(1) prod = 0
(2) i = 1
(3) $t_1 = 4 * i$
(4) $t_2 = a[t_1]$
(5) $t_3 = 4 * i$
(6) $t_4 = b[t_3]$
(7) $t_5 = t_2 * t_4$
(8) $t_6 = prod + t_5$
(9) $prod = t_6$
(10) $t_7 = i + 1$
(11) $i = t_7$
(12) if i <= 20 goto (3)

B_1:
(1) prod = 0
(2) i = 1

B_2:
(3) $t_1 = 4 * i$
(4) $t_2 = a[t_1]$
(5) $t_3 = 4 * i$
(6) $t_4 = b[t_3]$
(7) $t_5 = t_2 * t_4$
(8) $t_6 = prod + t_5$
(9) $prod = t_6$
(10) $t_7 = i + 1$
(11) $i = t_7$
(12) if i <= 20 goto (3)

一个简单的代码生成器

- 为单个块生成代码
  - 依次考虑三地址指令
  - 进行寄存器组织和管理（最大限度地利用）
    - 执行一个运算时，部分或全部分量在寄存器中
    - 寄存器存放单个组块内的临时变量
    - 寄存器可以存放全局值
    - 运行时刻的存储管理

一个简单的代码生成器

- 为单个块生成代码
  - 每个寄存器都有寄存器描述符
    - 跟踪哪些变量的当前值存放在此寄存器
  - 每个变量都有地址描述符
    - 跟踪哪些位置上可找到该变量的当前值。位置可以指一个寄存器、一个内存地址、一个栈中的位置

	R1	R2	R3	a	b	c	d	t	u	v
t=a-b				a	b	c	d			
LD R1, a										
LD R2, b										
SUB R2,R1, R2	a	t		a,R1	b	c	d	R2		

一个简单的代码生成器

- 代码生成算法
  - **getReg(*I*)**：为每个与三地址指令*I*有关的内存位置选择寄存器
  - 运算的机器指令：**x = y + z**
    - **getReg(x=y+z)**为**x**、**y**、**z**选择寄存器，记为R_x、R_y、R_z
    - 如果**y**不在R_y的寄存器描述符中，则生成指令“**LD** R_y**, y’**”，其中**y’**是存放**y**的内存位置之一(**y’**可以根据**y**的地址描述符得到)。
    - **z**的处理跟**y**的处理类似
    - 生成指令“**ADD** R_x**,** R_y**,** R_z”

一个简单的代码生成器

- 代码生成算法
  - 管理寄存器和地址描述符
    - LD R, x
      - 修改R的寄存器描述符，使之只包含x
      - 把R加入到x的地址描述符中
      - 从任何不同于x的变量的地址描述符中删除R

一个简单的代码生成器

- 代码生成算法
  - 管理寄存器和地址描述符
    - ADD R_x, R_y, R_z
      - 改变R_x的寄存器描述符，使之只包含x
      - 改变x的地址描述符，使之只包含R_x
      - 从任何不同于x的变量的地址描述符中删除R_x

一个简单的代码生成器

t=a-b
u=a-c
v=t+u
a=d
d=v+u

- 代码生成算法

	R1	R2	R3	a	b	c	d	t	u	v
t=a-b				**a**	**b**	**c**	**d**			

LD R1, a
LD R2, b
SUB R2,R1, R2
	a	**t**		**a,R1**	**b**	**c**	**d**	**R2**		

一个简单的代码生成器

t=a-b
u=a-c
v=t+u
a=d
d=v+u

- 代码生成算法

	R1	R2	R3	a	b	c	d	t	u	v

t=a-b
LD R1, a
LD R2, b
SUB R2,R1, R2
	a	**t**		**a,R1**	**b**	**c**	**d**	**R2**		

u=a-c
LD R3, c
SUB R1, R1, R3
	u	**t**	**c**	**a**	**b**	**c,R3**	**d**	**R2**	**R1**	

一个简单的代码生成器

t=a-b
u=a-c
v=t+u
a=d
d=v+u

- 代码生成算法

	R1	R2	R3	a	b	c	d	t	u	v
t=a-b				a	b	c	d			

LD R1, a
LD R2, b
SUB R2,R1, R2
	a	t		a,R1	b	c	d	R2		

u=a-c										
LD R3, c SUB R1, R1, R3	u	t	c	a	b	c,R3	d	R2	R1	
v=t+u										
ADD R3, R2, R1	u	t	v	a	b	c	d	R2	R1	R3

…… ……

一个简单的代码生成器

t=a-b
u=a-c
v=t+u
a=d
d=v+u

- 代码生成算法

	R1	R2	R3	a	b	c	d	t	u	v
t=a-b				a	b	c	d			

LD R1, a
LD R2, b
SUB R2,R1, R2
	a	t		a,R1	b	c	d	R2		

u=a-c										
LD R3, c SUB R1, R1, R3	u	t	c	a	b	c,R3	d	R2	R1	
v=t+u										
ADD R3, R2, R1	u	t	v	a	b	c	d	R2	R1	R3
…….				…….						
exit										
ST a, R2 ST d, R1	d	a	v	a,R2	b	c	d,R1			R3

一个简单的代码生成器

- 代码生成算法
  - 函数getReg：以x=y+z作一般性的例子
    - 为y选择R_y
      - 如果y在一个寄存器中，就选择这个寄存器

一个简单的代码生成器

- 代码生成算法
  - 函数getReg：以x=y+z作一般性的例子
    - 为y选择R_y
      - 如果y在一个寄存器中，就选择这个寄存器
      - 如果y不在一个寄存器中，但存在空寄存器，则选择该空寄存器

一个简单的代码生成器

- 代码生成算法
  - 函数getReg：以x=y+z作一般性的例子
    - 为y选择R_y
      - 如果y在一个寄存器中，就选择这个寄存器
      - 如果y不在一个寄存器中，但存在空寄存器，则选择该空寄存器
      - 如果y不在一个寄存器中，也不存在空寄存器
        - 设R是一个候选寄存器，v是R的寄存器描述符中的一个变量

一个简单的代码生成器

- 代码生成算法
  - 函数getReg：以x=y+z作一般性的例子
    - 为y选择R_y
      - 如果y在一个寄存器中，就选择这个寄存器
      - 如果y不在一个寄存器中，但存在空寄存器，则选择该空寄存器
      - 如果y不在一个寄存器中，也不存在空寄存器
        - 设R是一个候选寄存器，v是R的寄存器描述符中的一个变量

          (1)如果v的地址描述符说v还保存在R之外的地方，则选择R

          (2)如果v是x，且不同时是z，则选择R(因为原来的x不再使用)

          (3)如果v在以后不再使用，则选择R

          (4)如果前面3个条件都不满足，调用“ST v, R”，将R的值复制到内存位置上

一个简单的代码生成器

- 代码生成算法
  - 函数getReg：以x=y+z作一般性的例子
    - 为x选择R_x
      - 与为y选择R_y相同，只是有下列不同：
        - 对于只存放了x的寄存器R_x来说，选择R_x安全
        - 如果y在当前指令之后不再使用，且R_y仅存有y，则选择R_y作R_x；对z和Rz有类似的选择。