Intermediate Code Generation

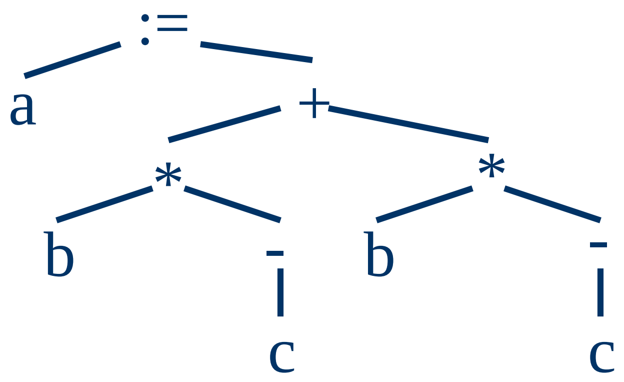
Intermediate Code Generation

- Intermediate languages
- Declarations
- Expressions
- Statements

Intermediate Languages

a := b * - c + b * - c

- Syntax tree



- Postfix notation

  a b c - * b c - * + :=

- Three-address code

MIPS Processors

- MIPS is a load-store architecture, which means that only load and store instructions access memory
- Computation instructions operate only on values in registers

SPIM Simulator

- SPIM is a software simulator that runs programs written for MIPS R2000/R3000 processors
- SPIM's name is just MIPS spelled backwards
- SPIM can read and immediately execute MIPS assembly language files or MIPS executable files
- SPIM contains a debugger and provides a few operating system-like services

MIPS Registers

Name	Number	Usage
$zero	0	constant 0
$v0~$v1	2~3	return value of a function
$a0~$a3	4~7	arguments
$t0~$t7	8~15	temporary (not preserved across call)
$s0~$s7	16~23	saved temporary (preserved across call)
$t8~$t9	24~25	temporary (not preserved across call)
$k0~$k1	26~27	reserved for OS kernel
$gp	28	pointer to global area
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address
$f0~f31		registers for floating-point

Addressing Modes

Format	Address computation
register	contents of register
imm	immediate
imm(register)	contents of (immediate + contents of register)
label	address of label

Load, Store and Move Instructions

li	rd, imm	$rd \leftarrow imm$
la	rd, label	$rd \leftarrow label$
lw	rd, imm(rs)	$rd \leftarrow imm(rs)$
sw	rd, imm(rs)	$imm(rs) \leftarrow rd$
move	rd, rs	$rd \leftarrow rs$

Arithmetic Instructions

add	rd, rs, rt	rd ← rs + rt
sub	rd, rs, rt	rd ← rs – rt
mul	rd, rs, rt	rd ← rs * rt
div	rd, rs, rt	rd ← rs / rt
rem	rd, rs, rt	rd ← rs % rt
neg	rd, rs	rd ← - rs

Branch Instructions

beq	rs, rt, label	branch to label if rs == rt
bne	rs, rt, label	branch to label if rs != rt
bgt	rs, rt, label	branch to label if rs > rt
bge	rs, rt, label	branch to label if rs >= rt
blt	rs, rt, label	branch to label if rs < rt
ble	rs, rt, label	branch to label if rs <= rt
b	label	branch to label

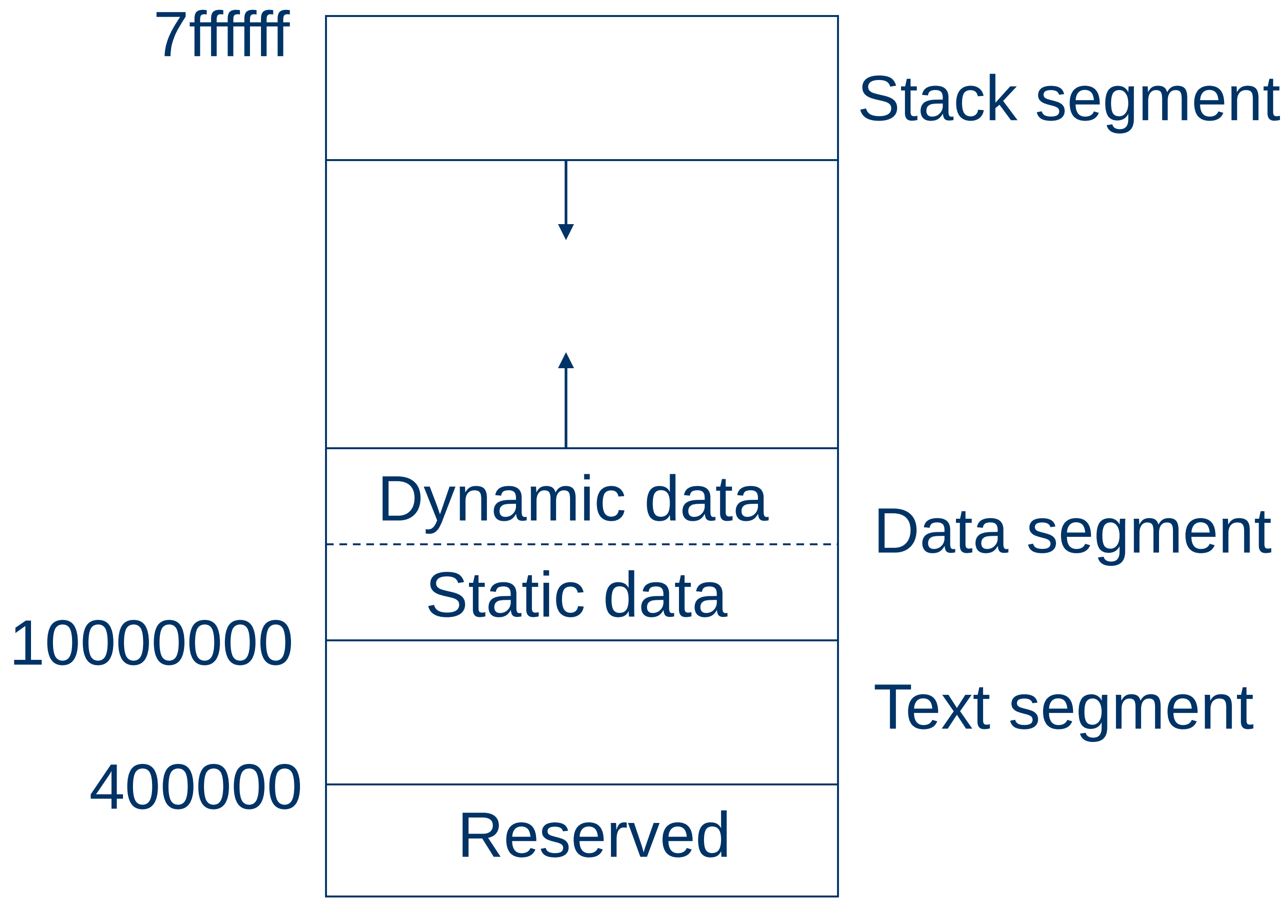
Assembler Syntax

- Comments in assembler files begin with a sharp sign (#) and continue to the end of the line
- Identifiers are a sequence of alphanumeric characters, underbars (_), and dots (.) that do not begin with a number
- Opcodes are reserved words that cannot be used as identifiers

Assembler Syntax

- Labels are declared by putting them at the beginning of a line followed by a colon
- Numbers are base 10 by default. If they are preceded by 0x, they are interpreted as hexadecimal

Memory Layout



Assembler Directives

.text

Subsequent items are put in the user text segment. These items may only be instructions.

.data

Subsequent items are stored in the data segment.

.word n

Store the 32-bit value n in current memory word

System Calls

- SPIM provides a small set of operating system-like services through the system call (syscall) instruction
- To request a service, a program loads the system call code into register $v0 and arguments into registers $a0~$a3
- System calls that return values put their results in register $v0

System Call Code

Service	Code	Arguments	Result
print_int	$v0= 1	$a0=interger	
read_int	$v0= 5		integer in $v0
exit	$v0=10		

Counters: Registers and Labels

- Counter reg maintains the temporary registers and is initialized to zero.
- The counter reg is incremented after allocating a register and is decremented after reclaiming a register.
- Counter label maintains the labels for control of flow and is initialized to one.
- The counter label is incremented after allocating a label and label is never reclaimed.

Declarations

Inherited: S.reg, S.label
Synthsized: S.nreg, S.nlabel

P $\rightarrow$ {emit(“.data”);} D
        {emit(“.text”); S.reg = 0; S.label = 1;} S

D $\rightarrow$ T {L.in := T.type;} L D | ε

T $\rightarrow$ **int** {T.type := integer;}

L $\rightarrow$ {L_1.in := L.in;} L_1 ‘,’
        id {if L.in = integer then
                emit(**id**.text || “: ” || “.word 0”);}

L $\rightarrow$ **id** {if L.in = integer then
                emit(**id**.text || “: ” || “.word 0”);}

An Example

```
main() {
    int n;
    int s;
    int i;

    read n;
```

```
if ( n < 1 ) {
        write -1;
        return;
   } else {
        s = 0;
   } fi
```

```
    i = 1;
    while ( i <= n) {
        s = s + i;
        i = i + 1;
    }
    write s;
    return;
}
```

An Example

```
        .data
n:      .word   0
s:      .word   0
i:      .word   0
```

Assignments

Inherited: E.reg   Synthsized: E.nreg, E.place

$S \rightarrow$ **id** := {E.reg = S.reg;} E
    {emit("la" || E.nreg || ", " || **id**.label);
    emit("sw" || E.place || ", " || "0(" || E.nreg || ")");
    S.nreg = E.nreg - 1; S.nlable = S.label;}

$E \rightarrow$ {E_1.reg = E.reg;} E_1 + {E_2.reg = E_1.nreg;} E_2
    {emit("add" || E_1.place || ", "
        || E_1.place || ", " || E_2.place);
    E.nreg = E_2.nreg - 1; E.place := E_1.place;}

Assignments

E $\rightarrow$ - {E_1.reg = E.reg;} E_1
{emit("neg" || E_1.place || ", " || E_1.place);
E.nreg = E_1.nreg; E.place := E_1.place;}

E $\rightarrow$ ({E_1.reg = E.reg;} E_1)
{E.nreg = E_1.nreg; E.place := E_1.place;}

E $\rightarrow$ **id** {emit("la" || E.reg || ", " || **id**.label);
emit("lw" || E.reg || ", " || "0(" || E.reg || ")");
E.place := E.reg; E.nreg = E.reg + 1;}

E $\rightarrow$ num {emit("li" || E.reg || ", " || num.value);
E.place := E.reg; E.nreg = E.reg + 1;}

An Example

```
main() {
    int n;
    int s;
    int i;

    read n;
```

```
if ( n < 1 ) {
        write -1;
        return;
   } else {
        s = 0;
   } fi
```

```
    i = 1;
    while ( i <= n) {
        s = s + i;
        i = i + 1;
    }
    write s;
    return;
}
```

An Example

```
# s := 0;
li      $t0, 0
la      $t1, s
sw      $t0, 0($t1)
```

```
# s := s + i;
la      $t0, s
lw      $t0, 0($t0)
la      $t1, i
lw      $t1, 0($t1)
add     $t0, $t0, $t1
la      $t1, s
sw      $t0, 0($t1)
```

24

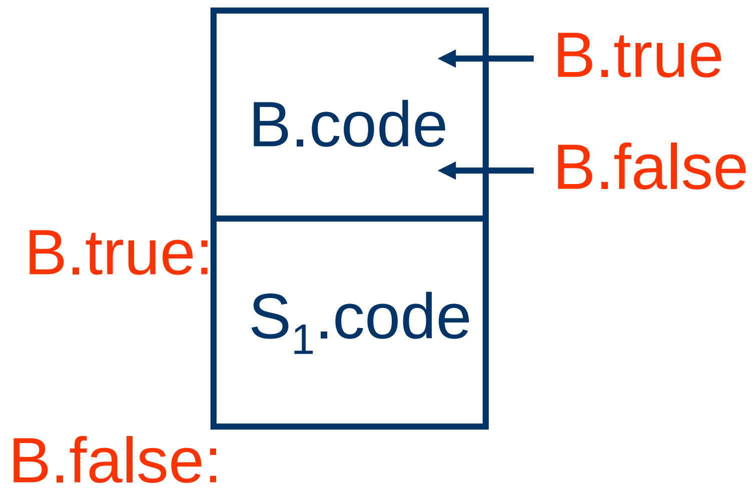
Flow-of-Control Statements

```
S → if B then S₁
  | if B then S₁ else S₂
  | while B do S₁
  | switch E begin
        case V₁: S₁
          …
        case Vₙ₋₁: Sₙ₋₁
        default: Sₙ
      end
```

Inherited: B.reg, B.label, B.true, B.false

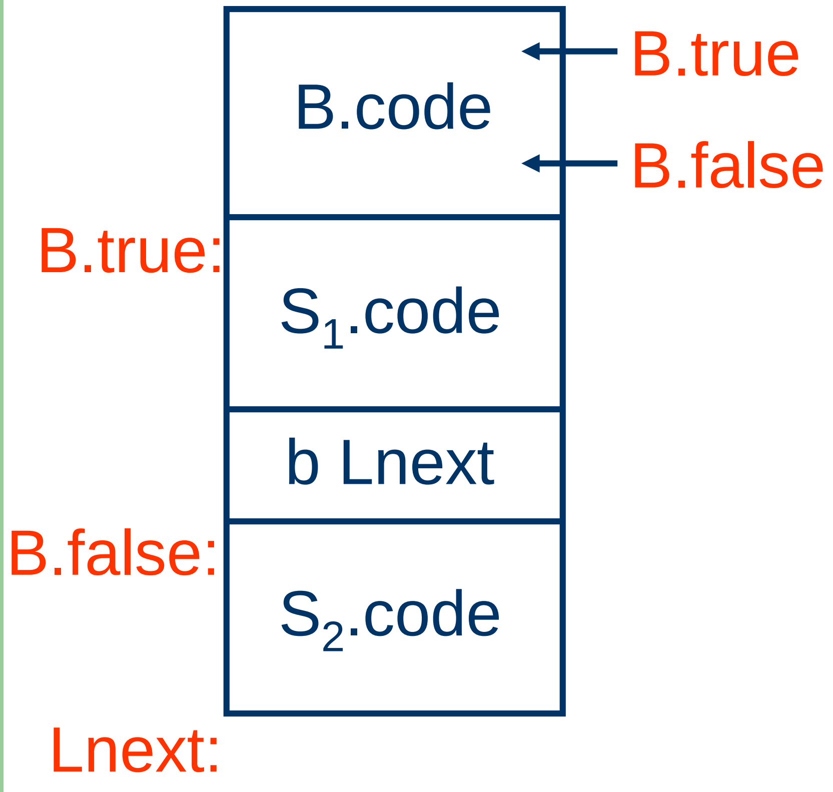
Synthsized: B.nreg, B.nlabel

Conditional Statements



```
S → if {B.true := S.label++;
         B.false := S.label++;
         B.reg := S.reg;
         B.label := S.label;}
    B then
         {emit(B.true || “:”);
          S_1.reg = B.nreg;
          S_1.label = B.nlabel;}
    S_1 {emit(B.false || “:”);
          S.nreg := S_1.nreg;
          S.nlabel := S_1.nlabel;}
```

Conditional Statements



$S \rightarrow$ **if**

{B.true := S.label++;
B.false := S.label++;
Lnext := S.label++;
B.reg := S.reg; B.label := S.label;}

B **then**

{emit(B.true || “:”);
S_1.reg = B.nreg; S_1.label = B.nlabel;}

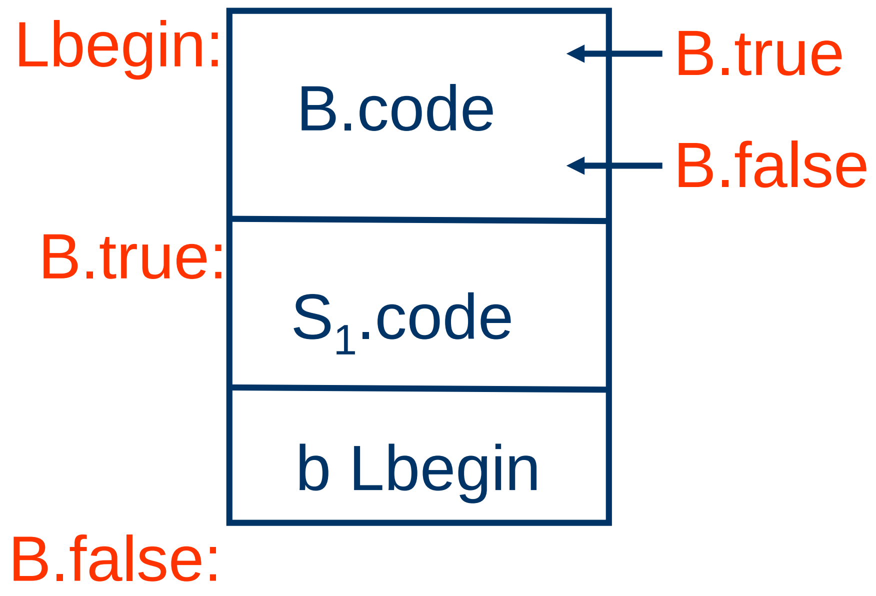
S_1 **else**

{emit(“b” || Lnext); emit(B.false || “:”);
S_2.reg := S_1.nreg;
S_2.label := S_1.nlabel;}

S_2

{emit(Lnext || “:”);
S.nreg := S_2.nreg;
S.nlabel := S_2.nlabel;}

Loop Statements



S $\rightarrow$ {Lbegin := S.label++;
emit(Lbegin || “:”);}
while
{B.true := S.label++;
B.false := S.label++;
B.reg := S.reg; B.label := S.label;}
B **do**
{emit(B.true || “:”);
S_1.reg = B.nreg;
S_1.label = B.nlabel;}
S_1
{emit(“b” || Lbegin);
emit(B.false || “:”);
S.nreg := S_1.nreg;
S.nlabel := S_1.nlabel;}

Boolean Expressions

B $\rightarrow$ {B_1.true := B.true; B_1.false := B.label++;
 B_1.reg = B.reg; B_1.label = B.label;}
B_1 **or**
{emit(B_1.false || ":");
 B_2.true := B.true; B_2.false := B.false;
 B_2.reg := B_1.nreg; B_2.label := B_1.nlabel;}
B_2
{B.nreg := B_2.nreg; B.nlabel := B_2.nlabel;}

Boolean Expressions

```
B → {B₁.true := B.label++; B₁.false := B.false;
     B₁.reg = B.reg; B₁.label = B.label;}
    B₁ and
    {emit(B₁.true || ":");
     B₂.true := B.true;  B₂.false := B.false;
     B₂.reg := B₁.nreg; B₂.label := B₁.nlabel;}
    B₂
    {B.nreg := B₂.nreg; B.nlabel := B₂.nlabel;}
```

$B \rightarrow \{B_1.true := B.label{+}{+};\ B_1.false := B.false;$
$B_1.reg = B.reg;\ B_1.label = B.label;\}$
B_1 **and**
$\{emit(B_1.true \mid\mid \text{“:”});$
$B_2.true := B.true;\ B_2.false := B.false;$
$B_2.reg := B_1.nreg;\ B_2.label := B_1.nlabel;\}$
B_2
$\{B.nreg := B_2.nreg;\ B.nlabel := B_2.nlabel;\}$

Boolean Expressions

```
B → not
        {B_1.true := B.false; B_1.false := B.true;
         B_1.reg = B.reg; B_1.label = B.label;}
        B_1
        {B.nreg := B_1.nreg; B.nlabel := B_1.nlabel;}
B → "("
        {B_1.true := B.true; B_1.false := B.false;
         B_1.reg = B.reg; B_1.label = B.label;}
        B_1 ")"
        {B.nreg := B_1.nreg; B.nlabel := B_1.nlabel;}
```

Boolean Expressions

B $\rightarrow$ {E_1.reg = B.reg;} E_1 "<" {E_2.reg = E_1.nreg;} E_2

{emit("blt" || E_1.place || ", " || E_2.place || ", " || B.true);

emit("b" || B.false);

B.nreg = E_2.nreg - 2; B.nlabel := B.label;}

B $\rightarrow$ **true**

{emit("b" || B.true);

B.nreg := B.reg; B.nlabel := B.label;}

B $\rightarrow$ **false**

{emit("b" || B.false);

B.nreg := B.reg; B.nlabel := B.label;}

An Example

a < b **or** c < d **and** e < f   B.true := L1; B.false := L2;

```
     la $t0, a                   lw $t1, 0($t1)
     lw $t0, 0($t0)              blt $t0, $t1, L4
     la $t1, b                   b  L2
     lw $t1, 0($t1)          L4: la $t0, e
     blt $t0, $t1, L1            lw $t0, 0($t0)
     b  L3                       la $t1, f
L3:  la $t0, c                   lw $t1, 0($t1)
     lw $t0, 0($t0)              blt $t0, $t1, L1
     la $t1, d                   b  L2
```

An Example

```
while a < b do
   if c < d then
      x := y + z
   else
      x := y - z
```

Lbegin := L1
B_1.true := L2
B_1.false := L3
B_2.true := L4
B_1.false := L5
Lnext := L6

```
L1:  # while
     la $t0, a
     lw $t0, 0($t0)
     la $t1, b
     lw $t1, 0($t1)
     blt $t0, $t1, L2
     b  L3
L2:  # body
     la $t0, c
     lw $t0, 0($t0)
     la $t1, d
     lw $t1, 0($t1)
     blt $t0, $t1, L4
     b  L5
```

An Example

```
while a < b do
   if c < d then
      x := y + z
   else
      x := y - z
```

Lbegin := L1
B_1.true := L2
B_1.false := L3
B_2.true := L4
B_1.false := L5
Lnext := L6

```
L4:  # then
     la $t0, y
     lw $t0, 0($t0)
     la $t1, z
     lw $t1, 0($t1)
     add $t0, $t0, $t1
     la $t1, x
     sw $t0, 0($t1)
     b  L6
```

An Example

```
while a < b do
  if c < d then
    x := y + z
  else
    x := y - z
```

$Lbegin := L1$ $B_2.true := L4$
$B_1.true := L2$ $B_1.false := L5$
$B_1.false := L3$ $Lnext := L6$

```
L5:  # else
     la $t0, y
     lw $t0, 0($t0)
     la $t1, z
     lw $t1, 0($t1)
     sub $t0, $t0, $t1
     la $t1, x
     sw $t0, 0($t1)
L6:  # end if
     b  L2
L3:  # end while
```

An Example

```
main() {
   int n;
   int s;
   int i;

   read n;

   if ( n < 1 ) {
         write -1;
         return;
   } else {
         s = 0;
   } fi

   i = 1;
   while ( i <= n) {
      s = s + i;
      i = i + 1;
   }
   write s;
   return;
}
```

An Example

```
        .data
n:      .word   0
s:      .word   0
i:      .word   0
        .text
main:
        li      $v0, 5
        syscall
        la      $t0, n
        sw      $v0, 0($t0)
```

```
        la      $t0, n
        lw      $t0, 0($t0)
        li      $t1, 1
        blt     $t0, $t1, L1
        b       L2
L1:     # then
        li      $t0, 1
        neg     $t0, $t0
        move    $a0, $t0
        li      $v0, 1
        syscall
```

An Example

```
        li      $v0, 1
        syscall
        b       L3
L2:     # else
        li      $t0, 0
        la      $t1, s
        sw      $t0, 0($t1)
L3:     # end if
        li      $t0, 1
        la      $t1, I
        sw      $t0, 0($t1)
```

```
L4:     #  while
        la      $t0, i
        lw      $t0, 0($t0)
        la      $t1, n
        lw      $t1, 0($t1)
        ble     $t0, $t1, L5
        b       L6
L5:     #  body
        la      $t0, s
        lw      $t0, 0($t0)
```

39

An Example

```
        la      $t1, i
        lw      $t1, 0($t1)
        add     $t0, $t0, $t1
        la      $t1, s
        sw      $t0, 0($t1)
        la      $t0, i
        lw      $t0, 0($t0)
        li      $t1, 1
        add     $t0, $t0, $t1
        la      $t1, i
        sw      $t0, 0($t1)
```

```
        b       L7
L6:     # end while
        la      $t0, s
        lw      $t0, 0($t0)
        move    $a0, $t0
        li      $v0, 1
        syscall
        li      $v0, 10
        syscall
```

Case Statements

- Conditional goto’s
  - less than 10 cases
- Jump table
  - more than 10 cases
  - dense value range
- Hash table
  - more than 10 cases
  - sparse value range

Conditional Goto's

```
        code  to  evaluate
             E  into  t
        b  test
  L1:   code  for  S1
        b  next
           …
Ln-1:   code  for  Sn-1
        b  next
  Ln:   code  for  Sn
        b  next
```

```
test:  if  t  =  V1 b  L1
             …
       if  t  =  Vn-1 b Ln-1
       b  Ln
next:
```

Jump Table

```
code to evaluate E into t
if t < Vmin b Ldefault
if t > Vmax b Ldefault
i := t - Vmin
L := jumpTable[i]
b L
```

Hash Table

```
code to evaluate E into t
i := hash(t)
L := hashTable[i]
b L
```

Procedure Calls

```
S → call id "(" Elist ")"
    {for each item p in Elist.queue do
        emit("lw" || newArgReg() || ", " || "0(" || p
            || ")");
     emit("bal" id.place);}
Elist → Elist₁ "," E
    {insert(Elist₁.queue, E.place);
     Elist.queue := Elist₁.queue;}
Elist → E
    {Elist.queue := {}; insert(Elist.queue, E.place);}
```