Lecture 8: Intermediate Code

CS 540
Spring 2009
Compiler Architecture

Scanner (lexical analysis) → tokens → Parser (syntax analysis) → Syntactic structure → Semantic Analysis (IC generator) → Intermediate Code → Code Generator → Target language

Symbol Table

Source language

Intermediate Code

Code Optimizer
Intermediate Code

• Similar terms: *Intermediate representation, intermediate language*
• Ties the front and back ends together
• Language and Machine neutral
• Many forms
• Level depends on how being processed
• More than one intermediate language may be used by a compiler
Intermediate language levels

- High
  \[
t_1 \leftarrow a[i,j+2]
  \]

- Medium
  \[
  t_1 \leftarrow j + 2 \\
  t_2 \leftarrow i \times 20 \\
  t_3 \leftarrow t_1 + t_2 \\
  t_4 \leftarrow 4 \times t_3 \\
  t_5 \leftarrow \text{addr } a \\
  t_6 \leftarrow t_5 + t_4 \\
  t_7 \leftarrow *t_6
  \]

- Low
  \[
  r_1 \leftarrow [fp-4] \\
  r_2 \leftarrow r_1 + 2 \\
  r_3 \leftarrow [fp-8] \\
  r_4 \leftarrow r_3 \times 20 \\
  r_5 \leftarrow r_4 + r_2 \\
  r_6 \leftarrow 4 \times r_5 \\
  r_7 \leftarrow fp - 216 \\
  f_1 \leftarrow [r_7 + r_6]
  \]
Intermediate Languages Types

• Graphical IRs: Abstract Syntax trees, DAGs, Control Flow Graphs

• Linear IRs:
  – Stack based (postfix)
  – Three address code (quadruples)
Graphical IRs

- Abstract Syntax Trees (AST) – retain essential structure of the parse tree, eliminating unneeded nodes.
- Directed Acyclic Graphs (DAG) – compacted AST to avoid duplication – smaller footprint as well
- Control flow graphs (CFG) – explicitly model control flow
ASTs and DAGs:
\[ a := b \ast-c + b\ast-c \]
Linearized IC

• Stack based (one address) – compact
  push 2
  push y
  multiply
  push x
  subtract

• Three address (quadruples) – up to three operands, one operator
  t1 <- 2
  t2 <- y
  t3 <- t1 * t2
  t4 <- x
  t5 <- t4 - t1
SPIM

• Three address code
• We are going to use a subset as a mid-level intermediate code

• Loading/Storing
  – lw register,addr - moves value into register
  – li register,num - moves constant into register
  – la register,addr - moves address of variable into register
  – sw register,addr - stores value from register
# Spim Addressing Modes

<table>
<thead>
<tr>
<th>Format</th>
<th>Address =</th>
</tr>
</thead>
<tbody>
<tr>
<td>(register)</td>
<td>contents of register</td>
</tr>
<tr>
<td>imm</td>
<td>immediate</td>
</tr>
<tr>
<td>imm(register)</td>
<td>immediate + contents of register</td>
</tr>
<tr>
<td>symbol</td>
<td>address of symbol</td>
</tr>
<tr>
<td>symbol +/- imm</td>
<td>address of symbol + or - immediate</td>
</tr>
<tr>
<td>symbol +/- imm(register)</td>
<td>address of symbol + or – (immediate + contents of register)</td>
</tr>
</tbody>
</table>

We typically only use some of these in our intermediate code
Examples

li $t2,5 – load the value 5 into register t2
lw $t3,x – load value stored at location labeled ‘x’ into register t3
la $t3,x – load address of location labeled ‘x’ into register t3
lw $t0,($t2) – load value stored at address stored in register t2 into register t0
lw $t1,8($t2) – load value stored at address stored in register 2 + 8 into register t1
• Lots of registers – we will primarily use 8 ($t0 - $t7) for intermediate code generation

• Binary arithmetic operators – work done in registers (reg1 = reg2 op reg3) – reg3 can be a constant
  – add reg1, reg2, reg3
  – sub reg1, reg2, reg3
  – mul reg1, reg2, reg3
  – div reg1, reg2, reg3

• Unary arithmetic operators (reg1 = op reg2)
  – neg reg1, reg2
\[ a := b * -c + b * -c \]

\[ \text{lw } $t0, b \]
\[ a := b \times -c + b \times -c \]

\[
\begin{align*}
\text{lw } & \$t0,b \\
\text{lw } & \$t1,c
\end{align*}
\]
\[ a := b \times -c + b \times -c \]

\[
\begin{align*}
\text{lw} & \quad \$t0, b \\
\text{lw} & \quad \$t1, c \\
\text{neg} & \quad \$t1, \$t1
\end{align*}
\]
\[ a := b \times -c + b \times -c \]

```
lw $t0, b
lw $t1, c
neg $t1, $t1
mul $t1, $t1, $t0
```
\[ a := b \times -c + b \times -c \]

\[
\begin{align*}
\text{lw } &\$t0, b \\
\text{lw } &\$t1, c \\
\text{neg } &\$t1, $t1 \\
\text{mul } &\$t1, $t1, $t0 \\
\text{lw } &\$t0, b
\end{align*}
\]
\[ a := b \times -c + b \times -c \]

```
lw $t0,b
lw $t1,c
neg $t1,$t1
mul $t1, $t1,$t0
lw $t0,b
lw $t2,c
```
a := b *-c + b*-*c

\[
\begin{align*}
&\text{lw } $t0,b \\
&\text{lw } $t1,c \\
&\text{neg } $t1,$t1 \\
&\text{mul } $t1, $t1,$t0 \\
&\text{lw } $t0,b \\
&\text{lw } $t2,c \\
&\text{neg } $t2,$t2
\end{align*}
\]
\[ a := b \times -c + b \times -c \]

```
lw $t0,b
lw $t1,c
neg $t1,$t1
mul $t1, $t1,$t0
lw $t0,b
lw $t2,c
neg $t2,$t0
mul $t0,$t0,$t2
```
a := b *-c + b*-c

lw $t0,b
lw $t1,c
neg $t1,$t1
mul $t1, $t1,$t0
lw $t0,b
lw $t2,c
neg $t2,$t0
mul $t0,$t0,$t2
add $t1,$t0,$t1
a := b *-c + b*-c

```
lw $t0,b
lw $t1,c
neg $t1,$t1
mul $t1, $t1,$t0
lw $t0,b
lw $t2,c
neg $t2,$t0
mul $t0,$t0,$t2
add $t1,$t0,$t1
sw $t1,a
```
\[ a := b \times -c + b \times -c \]

```
lw $t0, b
lw $t1, c
neg $t1,$t1
mul $t1,$t1,$t0
add $t0,$t1,$t1
sw $t0,a
```
• **Comparison operators**

  set condition – \( \text{temp1} = \text{temp2} \ xxx \text{temp3} \), where \( xxx \) is a condition (gt, ge, lt, le, eq) – \( \text{temp1} \) is 0 for false, non-zero for true.

  - \texttt{sgt reg1,reg2,reg3}
  - \texttt{slt reg1,reg2,reg3}
  - ...
More Spim

• Jumps
  - \texttt{b label} - unconditional branch to label
  - \texttt{bxxx temp, label} – conditional branch to label, \texttt{xxx} = condition such as eqz, neq, ...

• Procedure statement
  - \texttt{jal label} – jump and save return address
  - \texttt{jr register} – jump to address stored in register
Control Flow

```plaintext
while x <= 100 do
    x := x + 1
end while
```

```assembly
lw $t0, x
li $t1, 100
L25: sle $t2, $t0, $t1
    beqz $t2, L26
    addi $t0, $t0, 1
    sw $t0, x
b L25

L26:
```

branch if false

loop body
Example: Generating Prime Numbers

print 2  print blank
for i = 3 to 100
    divides = 0
    for j = 2 to i/2
        if j divides i evenly then divides = 1
    end for
    if divides = 0 then print i  print blank
end for
exit
Loops

print 2  print blank
for i = 3 to 100
    divides = 0
    for j = 2 to i/2
        if j divides i evenly then divides = 1
    end for
    if divides = 0 then print i  print blank
end for
exit
Outer Loop: for i = 3 to 100

li $t0, 3  # variable i in t0
li $t1,100 # max loop counter in t1
l1:    sle $t7,$t0,$t1 # i <= 100
    beqz $t7, l2
    ...
    addi $t0,$t0,1 # increment i
    b 11

l2:
Inner Loop:  for j = 2 to i/2

li $t2,2           # j = 2 in t2
div $t3,$t0,2      # i/2 in t3
l3:   sle $t7,$t2,$t3   # j <= i/2
      beqz $t7,l4    # j <= i/2
   ...          # increment j
      addi $t2,$t2,1
      b 13

14:
Conditional Statements

print 2  print blank
for i = 3 to 100
    divides = 0
    for j = 2 to i/2
        if j divides i evenly then divides = 1
    end for
    if divides = 0 then print i  print blank
end for
exit
if j divides i evenly then divides = 1

rem $t7,$t0,$t2 # remainder of i/j
bnez $t7,15 # if there is

li $t4,1 # remainder

15:

... li $t4,1 # divides=1 in t4

bnez $t4,16 # if divides = 0 not prime
print i

16:
SPIM System Calls

- Write(i)
  
  ```
  li $v0,1
  lw $a0,I
  syscall
  ```

- Read(i)
  
  ```
  li $v0,5
  syscall
  sw $v0,i
  ```

- Exiting
  
  ```
  li $v0,10
  syscall
  ```
Example: Generating Prime Numbers

print 2  print blank
for i = 3 to 100
    divides = 0
    for j = 2 to i/2
        if j divides i evenly then divides = 1
    end for
    if divides = 0 then print i  print blank
end for
exit
.data

blank: .asciiz " "

.text

li $v0, 1
li $a0, 2
syscall    # print 2
li $v0, 4
la $a0, blank  # print blank
syscall

li $v0, 1
lw $a0, i
syscall    # print I

li $v0, 10
syscall    # exit
```
.data
blank: .asciiz " "
.text
main:
    li $v0,1
    li $a0,2
    syscall
    li $v0,4
    la $a0,blank
    syscall
    li $t0,3    # i in t0
    li $t1,100  # max in t1
l1:  sle $t7,$t0,$t1
    beqz $t7,l2
    li $t4,0
    li $t2,2   # jj in t2
    div $t3,$t0,2 # max in t3
l3:  sle $t7,$t2,$t3
    beqz $t7,l4
    rem $t7,$t0,$t2
    bnez $t7,l5
    li $t4,1
l5:   addi $t2,$t2,1
       b l3     #end of inner loop
l4:
    bnez $t4,l6
    li $v0,1
    move $a0,$t0
    syscall  # print i
    li $v0,4
    la $a0,blank
    syscall
l6:   addi $t0,$t0,1
       b l1  
       #end of outer loop
l2:   li $v0,10
       syscall
```

Entire program
can run by providing an input file

can also use more interactively
PC SPIM

SPIM Version Version 7.1 of January 2, 2005
Copyright 1990-2004 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
DOS and Windows ports by David A. Corley (dac@cs.wisc.edu).
Copyright 1997 by Morgan Kaufmann Publishers, Inc.
See the file README for a full copyright notice.
Loaded: C:\Program Files\PCSpim\exceptions.s
Notes

• Spim requires a main: label as starting location
• Data must be prefixed by “.data”
• Executable code must be prefixed by “.text”
• Data and code can be interspersed
• You can’t have variable names (i.e. labels) that are the same as opcodes – in particular, b and j are not good names (branch and jump)
Generating Intermediate Code

• Just as with typechecking, we need to use the syntax of the input to generate the output.
  – Declarations
  – Expressions
  – Control flow
  – Procedure call/return

Next week
Processing Declarations

- Global variables vs. local variables
- Binding name to storage location
- Basic types: integer, boolean …
- Composite types: records, arrays …
- Tied to expression code generation
In SPIM

- Declarations generate code in `.data` sections

  var_name1: .word 0
  var_name2: .word 29,10
  var_name3: .space 40

allocate a 4 byte word for each given initial value

Can also allocate a large space
Issues in Processing Expressions

• Generation of correct code
• Type checking/conversions
• Address calculation for constructed types (arrays, records, etc.)
• Expressions in control structures
Expressions

Grammar:
S → id := E
E → E + E
E → id

As we parse, generate IC for the given input. Use attributes to pass information about temporary variables up the tree.

Generate:
lw $t0, b
Expressions

Grammar:

\[ S \rightarrow \text{id} := E \]
\[ E \rightarrow E + E \]
\[ E \rightarrow \text{id} \]

Generate:

\[ \text{lw } $t0,b \]
\[ \text{lw } $t1,c \]

Each number corresponds to a temporary variable.
Expressions

Grammar:
\[ S \rightarrow \text{id := E} \]
\[ E \rightarrow E + E \]
\[ E \rightarrow \text{id} \]

Generate:
\[ \text{lw \ $t0,b} \]
\[ \text{lw \ $t1,c} \]
\[ \text{add \ $t0,$t0,$t1} \]

Each number corresponds to a temporary variable.
Expressions

Grammar:
- $S \rightarrow \text{id} := E$
- $E \rightarrow E + E$
- $E \rightarrow \text{id}$

$\text{a} := b + c + d + e$

Generate:
- \text{lw $t0,b}$
- \text{lw $t1,c$}
- \text{add $t0,$t0,$t1}$
- \text{lw $t1,d$}

Each number corresponds to a temporary variable.
Expressions

Grammar:
\[ S \rightarrow \text{id} := E \]
\[ E \rightarrow E + E \]
\[ E \rightarrow \text{id} \]

Each number corresponds to a temporary variable.

Generate:
\[ \text{lw t0,b} \]
\[ \text{lw t1,c} \]
\[ \text{add $t0,$t0,$t1} \]
\[ \text{lw t1,d} \]
\[ \text{add $t0,$t0,$t1} \]
Expressions

Grammar:
- $S \rightarrow id := E$
- $E \rightarrow E + E$
- $E \rightarrow id$

Each number corresponds to a temporary variable.

Generate:
- lw $t0,b$
- lw $t1,c$
- add $t0,$t0,$t1$
- lw $t1,d$
- add $t0,$t0,$t1$
- lw $t1,e$
Expressions

Grammar:
- S → id := E
- E → E + E
- E → id

Each number corresponds to a temporary variable.

Generate:
- lw $t0, b$
- lw $t1, c$
- add $t0, t0, t1$
- lw $t1, d$
- add $t0, t0, t1$
- lw $t1, e$
- add $t0, t0, t1$
Expressions

Grammar:
\[ S \rightarrow \text{id} := E \]
\[ E \rightarrow E + E \]
\[ E \rightarrow \text{id} \]

Each number corresponds to a temporary variable.

```
S 0
  E 0
    E 0
      E 0
    E 0
      E 1
    E 1
      E 1
      E 1
  E 1
    E 1
      E 1
      E 1

lw $t0,b
lw $t1,c
add $t0,$t0,$t1
lw $t1,d
add $t0,$t0,$t1
lw $t1,e
add $t0,$t0,$t1
sw $t0,a
```
## Processing Expressions: SPIM

<table>
<thead>
<tr>
<th>Production</th>
<th>Action</th>
</tr>
</thead>
</table>
| **S → id := E** | ```
{ 
  printf("sw $t%d,%s\n",$3.reg,$1);
  free_reg($3.reg);
}
``` |
| **E → E + E** | ```
{ 
  $$\.reg = $1\.reg;
  printf("add $t%d, $t%d, $t%d\n",$$\.reg, $1\.reg, $3\.reg);
  free_reg($3\.reg);
}
``` |
| **E → id** | ```
{ 
  p := lookup($1);
  $$\.reg = get_register();
  printf("lw $t%d,%s\n", $$\.reg,$1);
}
``` |
What about constructed types?

- For basic types, we may be able to just load the value.
- When processing declarations for constructed types, need to keep enough information to generate code that finds the appropriate data at runtime
  - Records
  - Arrays
  - …
Records

• Typical implementation: allocate a block large enough to hold all record fields
  
  ```
  struct s{
    type1 field-1;
    ...
    typen field-n;
  } data_object;
  ```

• Boundary issues

• Field names – address will be offset from record address
Records in Spim

- Allocate enough space to hold all of the elements.
- Multiple ways to do this
- Record holding 3 (uninitialized) four-byte integers named a, b, c:

```
record: .space 12
```

OR

```
record_a: .word 0
record_b: .word 0
record_c: .word 0
```

convert to scalar
Records in Spim

• Address calculations:
  – Version 1: base address + offset
  Ex: to get contents of record.b:
    ```
    la $t0, record
    add $t0, $t0, 4
    lw $t1, ($t0)
    ```
  – Version 2: similar to scalars
1-D arrays

\[ a[l..h] \] with element size \( s \)

- Number of elements: \( e = h - l + 1 \)
- Size of array: \( e \cdot s \)
- Address of element \( a[i] \), assuming \( a \) starts at address \( b \) and \( l \leq i \leq h \):
  \[ b + (i - l) \cdot s \]

<table>
<thead>
<tr>
<th>( a[l] )</th>
<th>( a[l+1] )</th>
<th>( a[l+2] )</th>
<th>( \cdots )</th>
<th>( a[h] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( b )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

a[3..100] with element size 4

• Number of elements: \(100 - 3 + 1 = 98\)

• Size of array: \(98 \times 4 = 392\)

• Address of element a[50], assuming a starts at address 100

\[100 + (50 - 3) \times 4 = 288\]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>104</td>
<td></td>
</tr>
</tbody>
</table>
1-D arrays in SPIM

a[10] <- assuming C-style arrays in the HL language

• Allocation

```
.data
a:   .word 0,1,2,3,4,5,6,7,8,9
```

• Address calculation:

```
#calculate the address of a[y] word size elements
la  $t0, a
lw  $t2,y
mul $t2,$t2,4     # multiply by word size
add $t0,$t0,$t2    #t0 holds address of a[y]
lw  $t2,($t0)      #t2 hold a[y]
```
Arrays

• Typical implementation: large block of storage of appropriate size

• Row major vs. column major

• Consider $a[4..6,3..4]$

<table>
<thead>
<tr>
<th>Address</th>
<th>Row</th>
<th>Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b + 0s$</td>
<td>$a[4,3]$</td>
<td>$a[4,3]$</td>
</tr>
<tr>
<td>$b + 1s$</td>
<td>$a[4,4]$</td>
<td>$a[5,3]$</td>
</tr>
<tr>
<td>$b + 2s$</td>
<td>$a[5,3]$</td>
<td>$a[6,3]$</td>
</tr>
<tr>
<td>$b + 3s$</td>
<td>$a[5,4]$</td>
<td>$a[4,4]$</td>
</tr>
<tr>
<td>$b + 4s$</td>
<td>$a[6,3]$</td>
<td>$a[5,4]$</td>
</tr>
<tr>
<td>$b + 5s$</td>
<td>$a[6,4]$</td>
<td>$a[6,4]$</td>
</tr>
</tbody>
</table>
2-D Arrays: Row Major

- $A[4..7,3..4]$
2-D arrays – Row major

\[ a[l_1..h_1, l_2..h_2] \] with element size \( s \)

- Number of elements: \( e = e_1 \times e_2 \), where \( e_1 = (h_1 - l_1 + 1) \) and \( e_2 = (h_2 - l_2 + 1) \)

- Size of array: \( e \times s \)

- Size of each dimension (stride):
  \[ d_1 = e_2 \times d_2 \]
  \[ d_2 = s \]

- Address of element \( a[i,j] \), assuming \( a \) starts at address \( b \) and \( l_1 \leq i \leq h_1 \) and \( l_2 \leq j \leq h_2 \):
  \[ b + (i - l_1) \times d_1 + (j - l_2) \times s \]
Example

A[3...100,4...50] with elements size 4
• $98 \times 47 = 4606$ elements
• $4606 \times 4 = 18424$ bytes long
• $d_2 = 4$ and $d_1 = 47 \times 4 = 188$
• If a starts at 100, a[5,5] is:
  $100 + (5-3) \times 188 + (5 - 4) \times 4 = 720$
2-D arrays in SPIM

a[3,5] <- assuming C-style arrays

• Allocation
  .data
  a: .space 60  # 15 word-size elements * 4

• Address calculation:
  #calculate the address of a[x,y] word size elements
  la $t0,a
  lw $t1,x
  mul $t1,$t1,20  # stride = 5 * 4 = 20
  add $t0,$t0,$t1  # start of a[x,...]
  lw $t1,y
  mul $t1,$t1,4  # multiply by word size
  add $t0,$t0,$t1  #t0 holds address of a[y]
  lw $t1,($t0)  #t2 hold a[y]
### 3-D Arrays

- $a[4..7,3..4,8..9]$
- Size of third (rightmost) dimension = $s$
- Size of second dimension = $s \times 2$
- Size of first dimension = $s \times 2 \times 2$

<table>
<thead>
<tr>
<th></th>
<th>$b+0s$</th>
<th>$a[4,3,8]$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$b+1s$</td>
<td>$a[4,3,9]$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$b+2s$</td>
<td>$a[4,4,8]$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$b+3s$</td>
<td>$a[4,4,9]$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$b+4s$</td>
<td>$a[5,3,8]$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$b+5s$</td>
<td>$a[5,3,9]$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$b+6s$</td>
<td>$a[5,4,8]$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$b+7s$</td>
<td>$a[5,4,9]$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$b+8s$</td>
<td>$a[6,3,8]$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$b+9s$</td>
<td>$a[6,3,9]$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$b+10s$</td>
<td>$a[6,4,8]$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$b+11s$</td>
<td>$a[6,4,9]$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$b+12s$</td>
<td>$a[7,3,8]$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$b+13s$</td>
<td>$a[7,3,9]$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$b+14s$</td>
<td>$a[7,4,8]$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>$a[7,4,9]$</td>
<td></td>
</tr>
</tbody>
</table>

**Note:** The table above illustrates the indexing of a 3-D array, showing how each element is accessed based on the indices provided.
3-D arrays – Row major

\[ a[l_1..h_1, l_2..h_2, l_3..h_3] \] with element size \( s \)

- Number of elements: \( e = e_1 \times e_2 \times e_3 \), where \( e_i = (h_i - l_i + 1) \)
- Size of array: \( e \times s \)
- Size of each dimension (stride):
  \[
  \begin{align*}
  d_1 &= e_2 \times d_2 \\
  d_2 &= e_3 \times d_3 \\
  d_3 &= s
  \end{align*}
  \]
- Address of element \( a[i,j,k] \), assuming \( a \) starts at address \( b \) and \( l_1 \leq i \leq h_1 \) and \( l_2 \leq j \leq h_2 \):
  \[
  b + (i - l_1) \times d_1 + (j - l_2) \times d_2 + (k - l_3) \times s
  \]
Example

A[3…100,4…50,1..4] with elements size 4
• 98*47* 4 = 18424 elements
• 18424 * 4 = 73696 bytes long
• d_3 = 4, d_2 = 4 * 4 = 16 and d_1 = 16 * 47 = 752
• If a starts at 100, a[5,5,2] is:
  100+(5-3) * 752 + (5 – 4) * 16 + (2 – 1)*4 = 1624
N-D arrays – Row Major

\[a[l_1..h_1, \ldots l_n..h_n]\] with element size \(s\)

- Number of elements: \(e = \prod e_i\) where \(e_i = (h_i - l_i + 1)\)
- Size of array: \(e \times s\)
- Size of each dimension (stride):
  \[d_i = e_{i+1} \times d_{i+1}\]
  \(d_n = s\)
- Address of element \(a[i_1,\ldots,i_n]\), assuming \(a\) starts at address \(b\) and \(l_j \leq i_j \leq h_j:\)
  \[b + (i_1 - l_1) \times d_1 + \ldots + (i_n - l_n) \times d_n\]
An object is an abstract data type that encapsulates data, operations and internal state behind a simple, consistent interface.

The Concept:

Elaborating the concepts:

• Each **object** needs local storage for its attributes
  – Attributes are static (*lifetime of object*)
  – Access is through methods

• Some methods are public, others are private

• Object’s internal state leads to complex behavior
Objects

• Each object needs local storage for its attributes
  – Access is through methods
  – Heap allocate object records or “instances”
• Need consistent, fast access ➔ use known, constant offsets in objects
• Provision for initialization
• Class variables
• Inheritance
Each object gets copies of all attributes and methods.
Class A {
    int b,c;
    A z;
    f1()
    f2()
}

Better Representation

For object x of type A:

Objects share methods
More typically:

Objects share methods (and static attributes) via shared class object (can keep counter of objects N)
OOL Storage Layout

Class variables
• Static class storage accessible by global name (class C)
  – Method code put at fixed offset from start of class area
  – Static variables and class related bookkeeping

Object Variables
• Object storage is heap allocated at object creation
  – Fields at fixed offsets from start of object storage
• Methods
  – Code for methods is stored with the class
  – Methods accessed by offsets from code vector
    • Allows method references inline
  – Method local storage in object (no calls) or on stack
Dealing with Single Inheritance

• Use **prefixing** of storage for objects

```java
Class Point {
    int x, y;
}
```

```java
Class ColorPoint extends Point {
    Color c;
}
```

Multiple inheritance??
Processing Control Structures

• Constructs:
  – If
  – While
  – Repeat
  – For
  – case

• Label generation – all labels must be unique

• Nested control structures – need a stack
Conditional Examples

if (y > 0) then begin
  ...body...
end

lw $t0,y
li $t1,0
sgt $t2,$t0,$t1  # = 1 if true
beqz $t2,L2
...body...

L2:

Control Flow
Conditional Examples

if (y > 0) then begin
    ... body-1 ...
end else
    ...body-2 ...
end

lw $t0,y
li $t1,0
sgt $t2,$t0,$t1  # = 1 if true
beqz $t2,L2
...body-1...
b L3
L2:
    ...body-2 ...
L3:

Control Flow
Looping constructs

while x < 100 do

    ... body ...

end

L25: lw $t0,x
     li $t1,100
     sge $t2,$t0,$t1
     beqz $t2,L26
     ... body ...
     b L25

L26: Control Flow
Generating Conditionals

if_stmt → IF expr THEN
    { code to eval expr ($2) already done
      get two new label names
      output conditional ($2=false) branch to first label}
stmts ELSE
    { output unconditional branch to second label
      output first label }
stmts ENDIF
    { output second label }
Generating Loops

\[
\text{for\_stmt} \rightarrow \text{FOR id = start TO stop}
\]
\[
\{ \text{code to eval start ($4$) and stop ($6$)} \text{done}
\]
\[
\text{get two new label names}
\]
\[
\text{output code to initialize id = start}
\]
\[
\text{output label1}
\]
\[
\text{output code to compare id to stop}
\]
\[
\text{output conditional branch to label2}
\}

\[
\text{stmts END}
\]
\[
\{ \text{increment id (and save)}
\]
\[
\text{unconditional branch to label1}
\]
\[
\text{output label2}
\}

Nested conditionals

- Need a stack to keep track of correct labels
- Can implement own stack
  - push two new labels at start of statement
  - pop two labels when end statement
  - while generating code, use the two labels on the top of the stack
- Can use YACC
  - Give two tokens (like IF and THEN) label types.
  - At start of statement, when generate new labels, assign them to these tokens
  - When you need the numbers for generation, just use the value associated with the token.