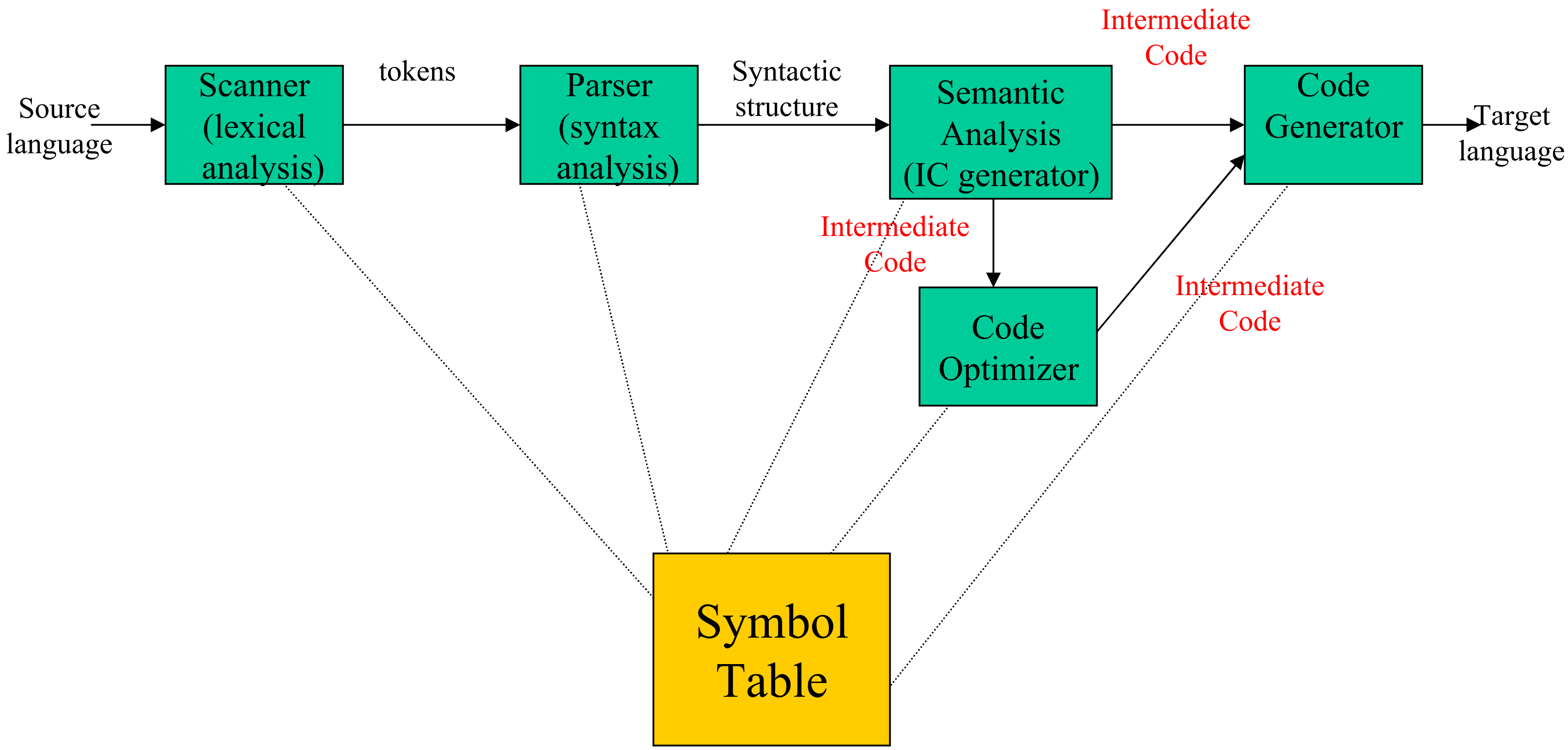
Lecture 8: Intermediate Code

CS 540

Spring 2009

Compiler Architecture



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

t1 ← a[i,j+2]

- Medium

t1 ← j + 2

t2 ← i * 20

t3 ← t1 + t2

t4 ← 4 * t3

t5 ← addr a

t6 ← t5 + t 4

t7 ← *t6

- Low

r1 ←[fp-4]

r2 ← r1 + 2

r3 ← [fp-8]

r4 ← r3*20

r5 ← r4 + r2

r6 ← 4 * r5

r7 ← fp – 216

f1 ← [r7+r6]

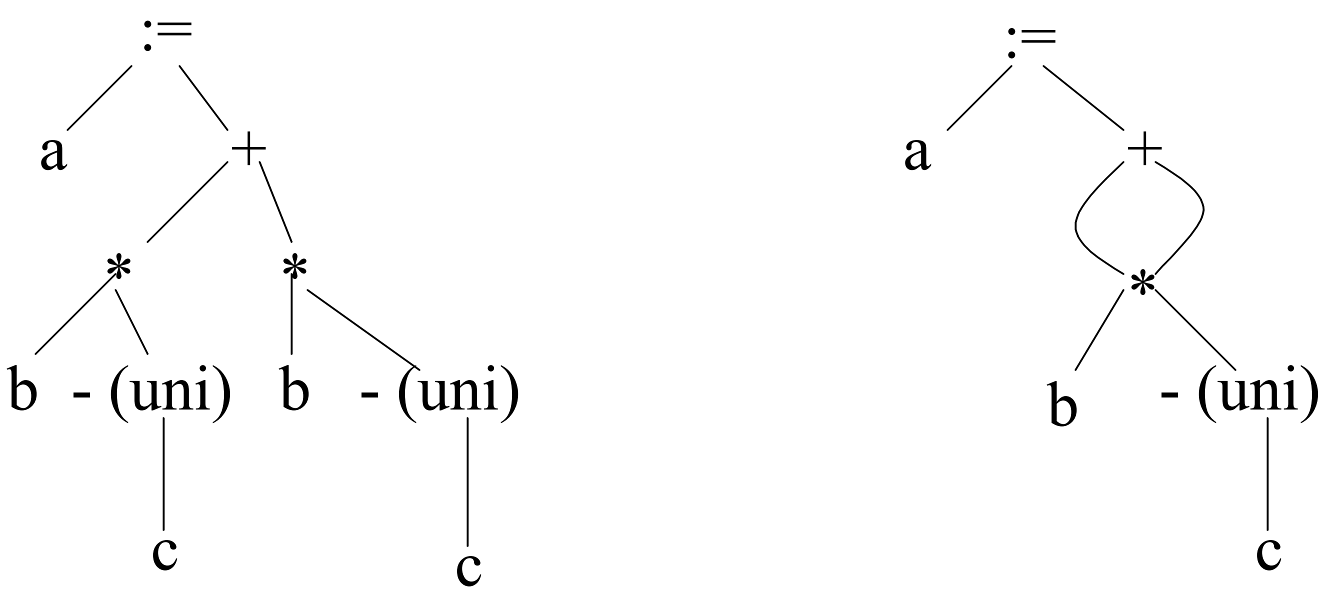
Intermediate Languages Types

- 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 *-c + b*-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

Format	***Address =***
(register)	contents of register
imm	immediate
imm(register)	immediate + contents of register
symbol	address of symbol
symbol +/- imm	address of symbol + or - immediate
symbol +/- imm(register)	address of symbol + or – (immediate + contents of register)

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

```
lw $t0,b
```

b
t0

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

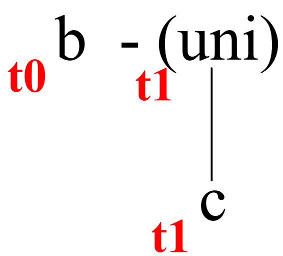
```
lw $t0,b
lw $t1,c
```

b
t0

c
t1

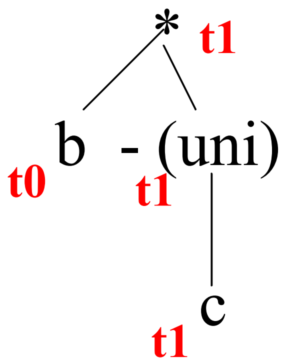
a := b *-c + b*-c

```
lw $t0,b
lw $t1,c
neg $t1,$t1
```

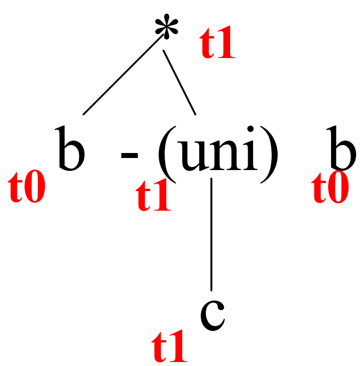


a := b *-c + b*-c

```
lw $t0,b
lw $t1,c
neg $t1,$t1
mul $t1, $t1,$t0
```

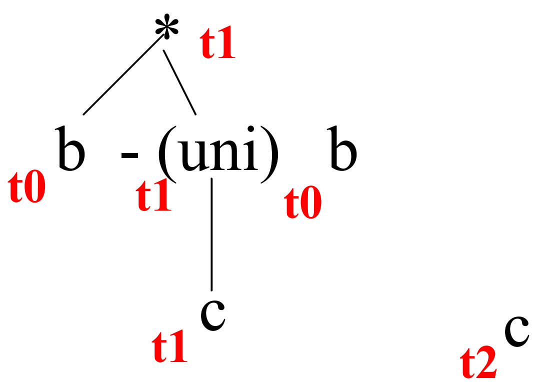


a := b *-c + b*-c



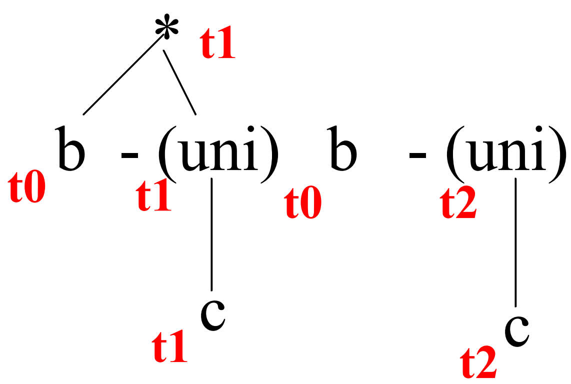

```
lw $t0,b
lw $t1,c
neg $t1,$t1
mul $t1, $t1,$t0
lw $t0,b
```

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



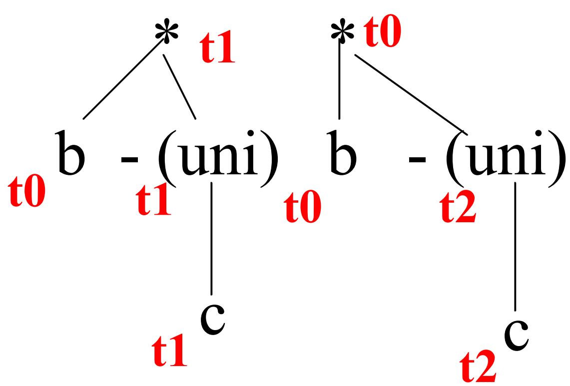

```
lw $t0,b
lw $t1,c
neg $t1,$t1
mul $t1, $t1,$t0
lw $t0,b
lw $t2,c
```

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



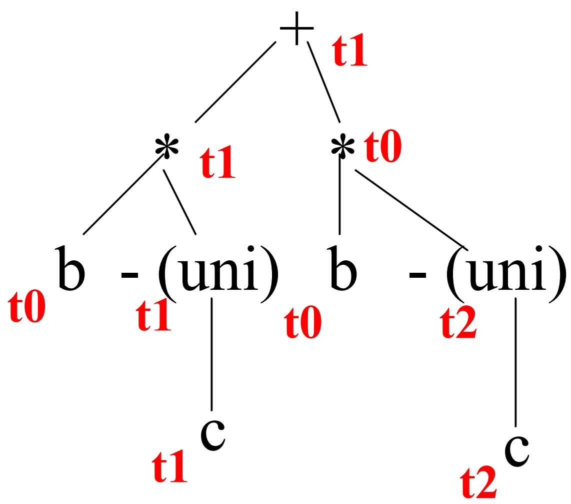

```
lw $t0,b
lw $t1,c
neg $t1,$t1
mul $t1, $t1,$t0
lw $t0,b
lw $t2,c
neg $t2,$t2
```

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



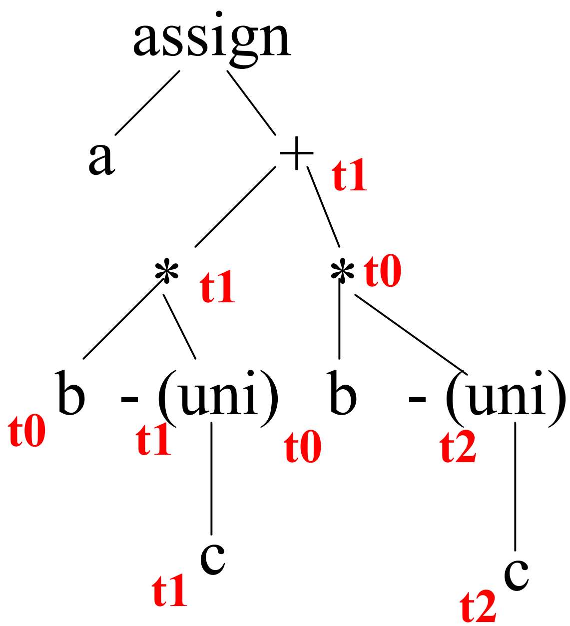

```
lw $t0,b
lw $t1,c
neg $t1,$t1
mul $t1, $t1,$t0
lw $t0,b
lw $t2,c
neg $t2,$t0
mul $t0,$t0,$t2
```

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
```

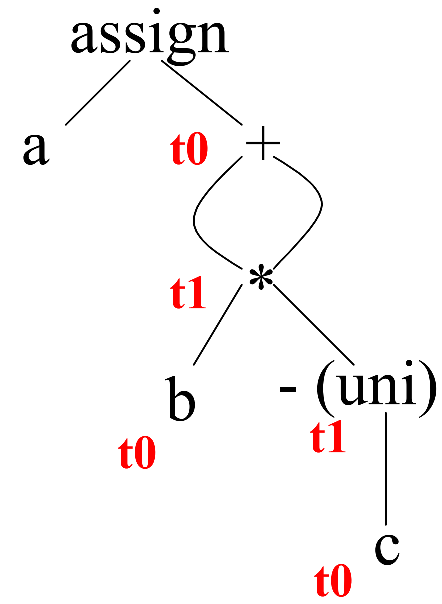
a := b *-c + b*-c




```
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 * -c + b * -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 – temp1 = temp2 xxx temp3, where xxx is a condition (gt, ge, lt, le, eq) – temp1 is 0 for false, non-zero for true.

  - **`sgt reg1,reg2,reg3`**
  - **`slt reg1,reg2,reg3`**
  - …

More Spim

- Jumps
  - **`b label`** - unconditional branch to label
  - **`bxxx temp, label`** – conditional branch to label, **`xxx`** = condition such as eqz, neq, …
- Procedure statement
  - **`jal label`** – jump and save return address
  - **`jr register`** – jump to address stored in register

Control Flow

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

```
     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 → beqz $t2,L26

loop body → addi $t0,$t0,1 / sw $t0,x

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 l1
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
      ...
      addi $t2,$t2,1       # increment j
      b l3
l4:
```

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,l5          # if there is
                         # remainder
    li $t4,1             # divides=1 in t4
l5:
  …
   bnez $t4,l6           # if divides = 0 not prime
   print i
l6:
```

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
```

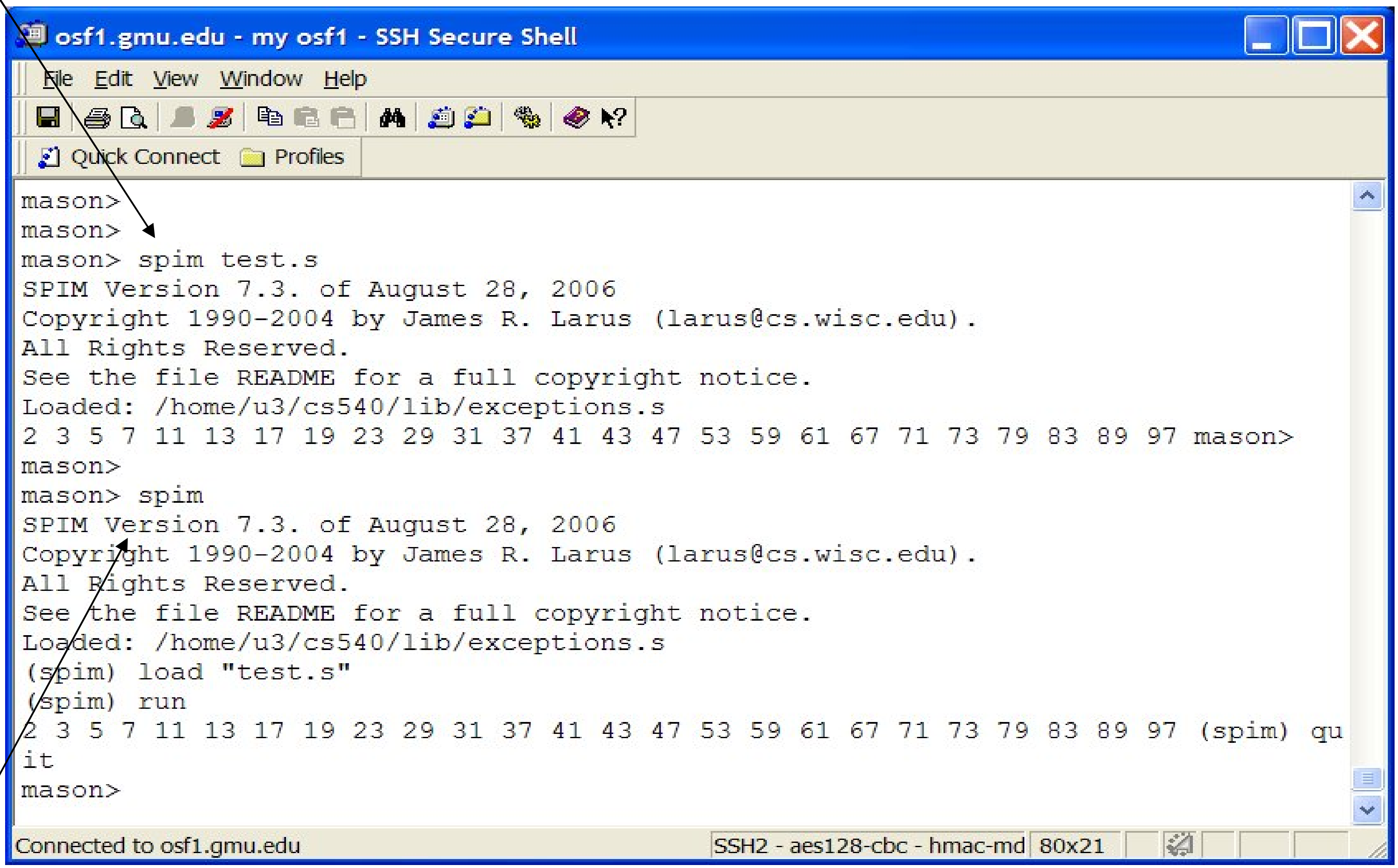
Entire program

```
        .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
```

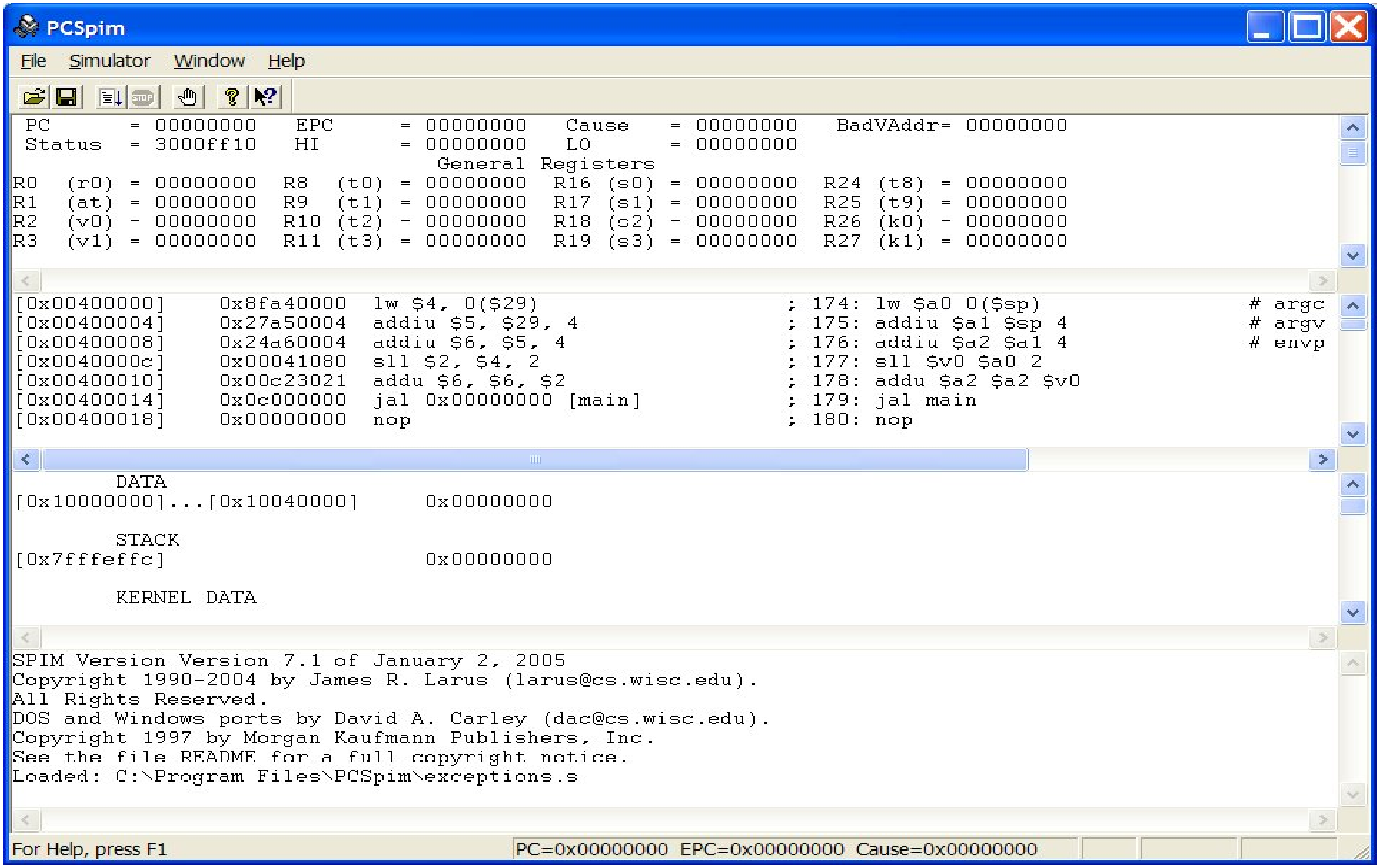
inner loop

can run by providing an input file



can also use more interactively

PC SPIM


PCSpim
File Simulator Window Help
PC = 00000000 EPC = 00000000 Cause = 00000000 BadVAddr= 00000000
Status = 3000ff10 HI = 00000000 LO = 00000000
General Registers
R0 (r0) = 00000000 R8 (t0) = 00000000 R16 (s0) = 00000000 R24 (t8) = 00000000
R1 (at) = 00000000 R9 (t1) = 00000000 R17 (s1) = 00000000 R25 (t9) = 00000000
R2 (v0) = 00000000 R10 (t2) = 00000000 R18 (s2) = 00000000 R26 (k0) = 00000000
R3 (v1) = 00000000 R11 (t3) = 00000000 R19 (s3) = 00000000 R27 (k1) = 00000000
[0x00400000] 0x8fa40000 lw $4, 0($29) ; 174: lw $a0 0($sp) # argc
[0x00400004] 0x27a50004 addiu $5, $29, 4 ; 175: addiu $a1 $sp 4 # argv
[0x00400008] 0x24a60004 addiu $6, $5, 4 ; 176: addiu $a2 $a1 4 # envp
[0x0040000c] 0x00041080 sll $2, $4, 2 ; 177: sll $v0 $a0 2
[0x00400010] 0x00c23021 addu $6, $6, $2 ; 178: addu $a2 $a2 $v0
[0x00400014] 0x0c000000 jal 0x00000000 [main] ; 179: jal main
[0x00400018] 0x00000000 nop ; 180: nop
DATA
[0x10000000]...[0x10040000] 0x00000000
STACK
[0x7fffeffc] 0x00000000
KERNEL DATA
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. Carley (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
For Help, press F1
PC=0x00000000 EPC=0x00000000 Cause=0x00000000

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

allocate a
4 byte word for
each given initial
value

- Declarations generate code

in **.data** sections

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

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

Generate:

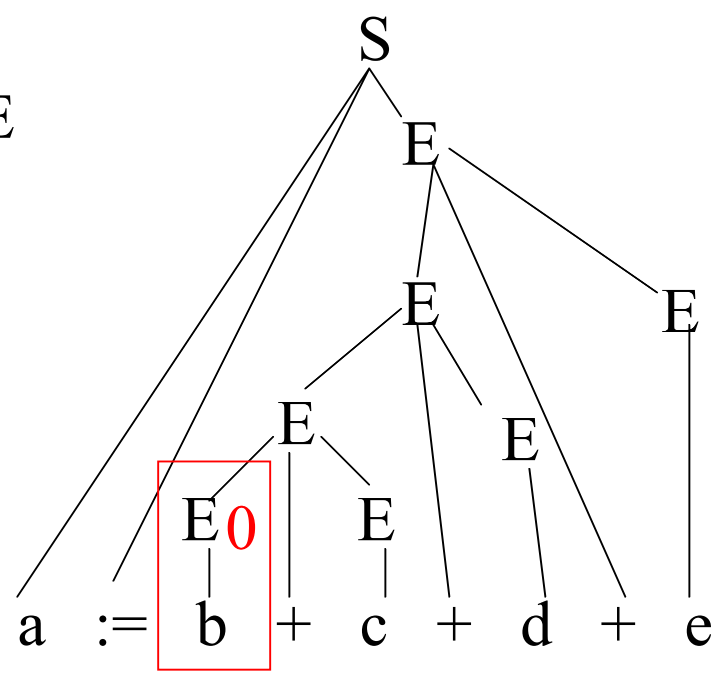
```
  lw $t0,b
```

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

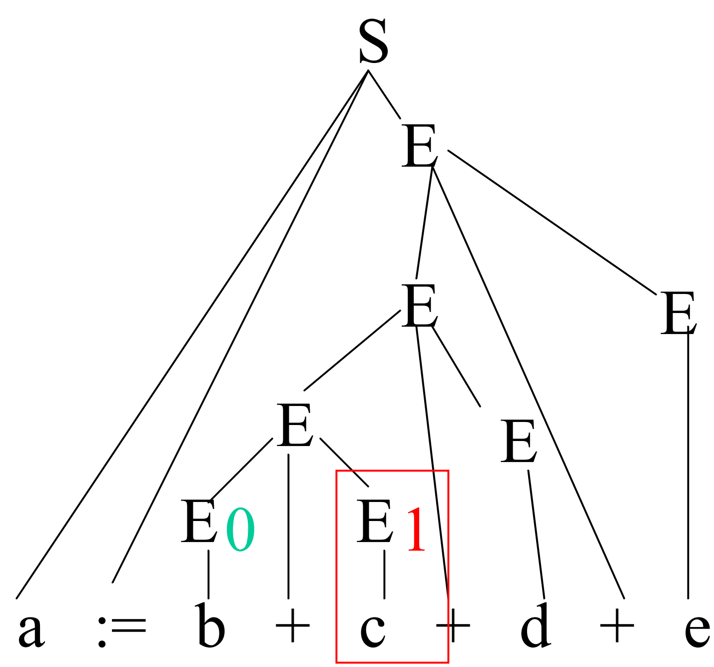
Expressions

Grammar:

S → id := E

E → E + E

E → id



Generate:

```
lw $t0,b
lw $t1,c
```

Each number corresponds to a temporary variable.

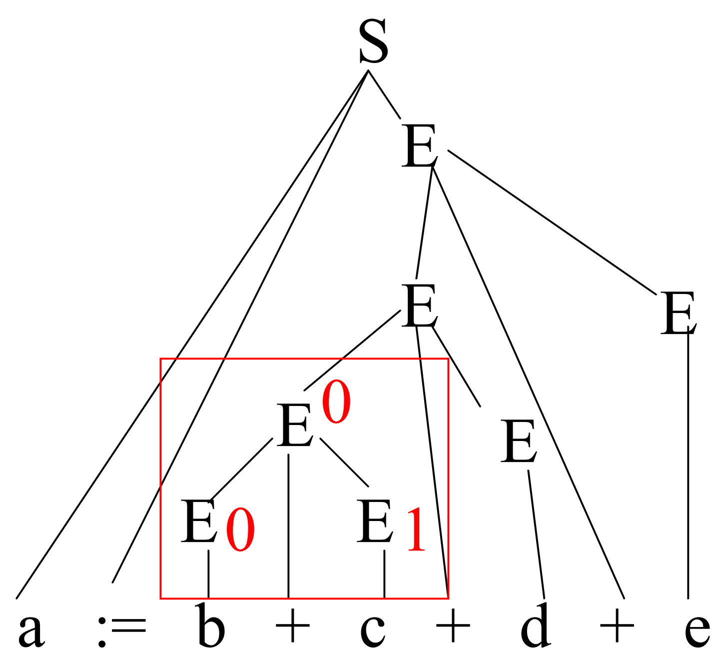
Expressions

Grammar:

S → id := E

E → E + E

E → id



Generate:

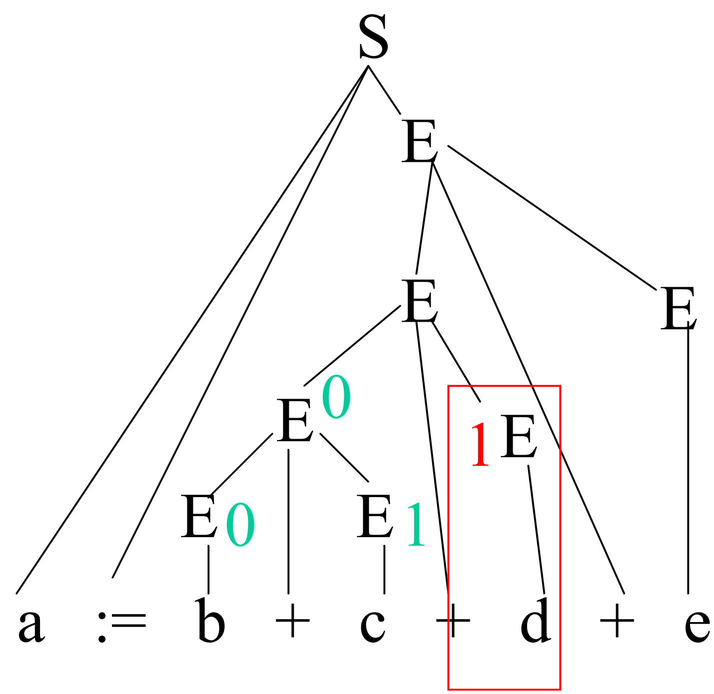
```
lw $t0,b
lw $t1,c
add $t0,$t0,$t1
```

Each number corresponds to a temporary variable.

Expressions

Grammar:

S → id := E
E → E + E
E → id



Generate:

```
lw $t0,b
lw $t1,c
add $t0,$t0,$t1
lw $t1,d
```

Each number corresponds to a temporary variable.

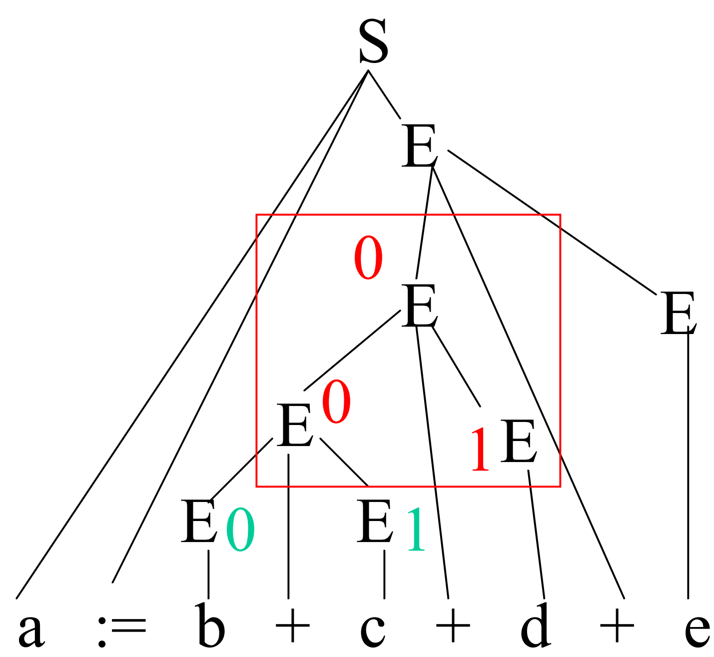
Expressions

Grammar:

S → id := E

E → E + E

E → id



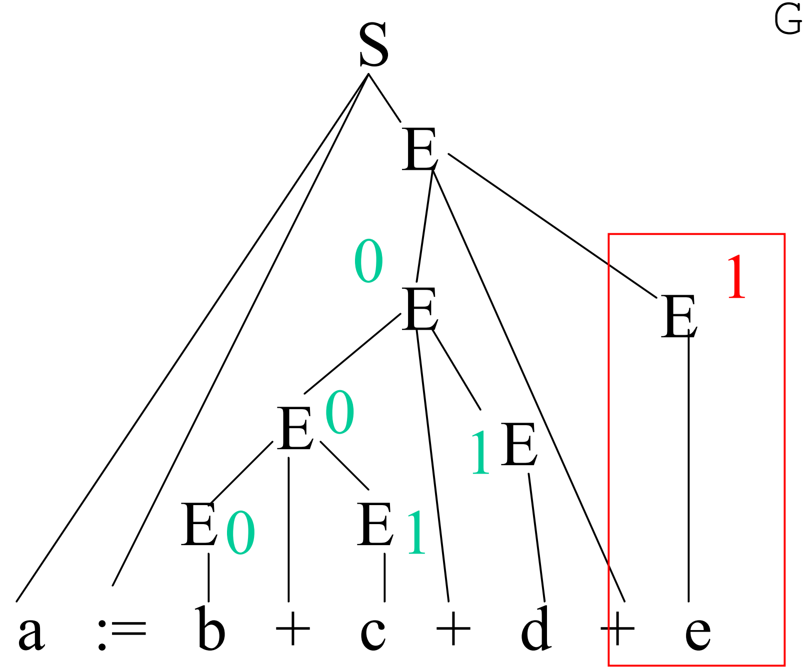

```
Generate:
  lw t0,b
  lw t1,c
  add $t0,$t0,$t1
  lw t1,d
  add $t0,$t0,$t1
```

Each number corresponds to a temporary variable.

Expressions

Grammar:

S → id := E
E → E + E
E → id




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

Each number corresponds to a temporary variable.

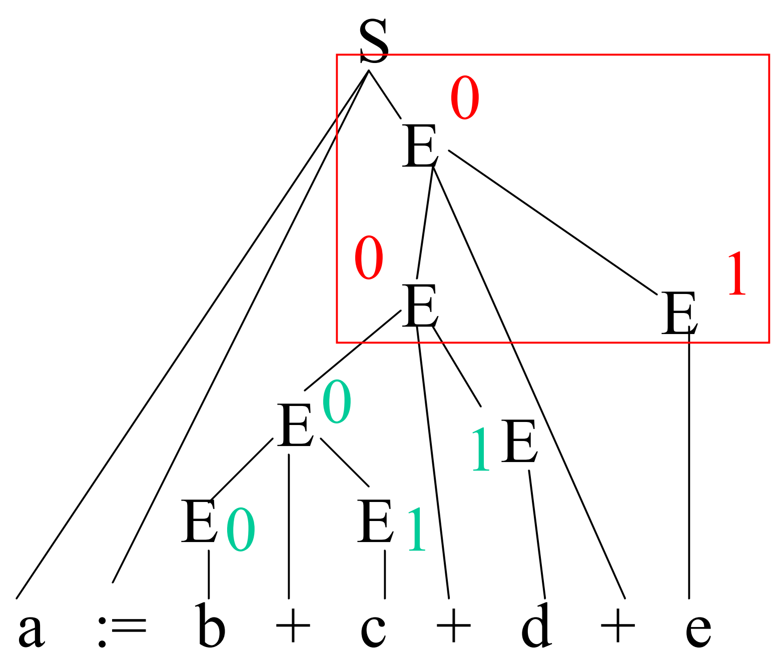
Expressions

Grammar:

S → id := E

E → E + E

E → id



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
```

Each number corresponds to a temporary variable.

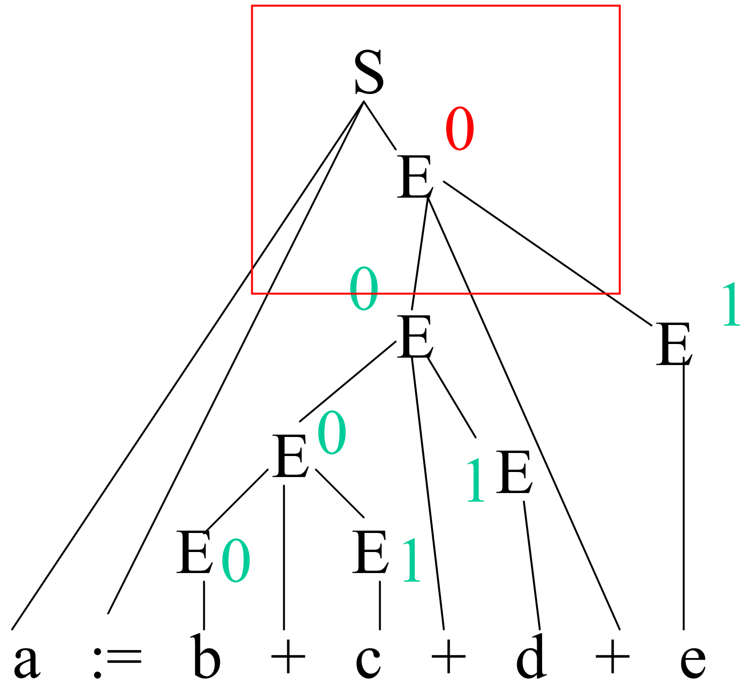
Expressions

Grammar:

S → id := E

E → E + E

E → id



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
sw $t0,a
```

Each number corresponds to a temporary variable.

Processing Expressions: SPIM

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)
```

b's offset in the record

  - 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: $e * s$
- Address of element a[i], assuming a starts at address *b* and $l <= i <= h$:

  $b + (i - l) * s$

a[*l*]	a[*l*+1]	a[*l*+2]	• • •	a[*h*]

b

Example

a[3..100] with element size 4

- Number of elements: 100 – 3 + 1 = 98
- Size of array: 98 * 4 = 392
- Address of element a[50], assuming a starts at address 100

  100 + (50 - 3) * 4 = 288

a[3]	a[4]	a[5]		a[100]

100 104

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]

Address	Row	Column
b + 0s	a[4,3]	a[4,3]
b + 1s	a[4,4]	a[5,3]
b + 2s	a[5,3]	a[6,3]
b + 3s	a[5,4]	a[4,4]
b + 4s	a[6,3]	a[5,4]
b + 5s	a[6,4]	a[6,4]

2-D Arrays: Row Major

- A[4..7,3..4]

	Address	Row
a[4,x]	b + 0s	a[4,3]
	b + 1s	a[4,4]
a[5,x]	b + 2s	a[5,3]
	b + 3s	a[5,4]
a[6,x]	b + 4s	a[6,3]
	b + 5s	a[6,4]
a[7,x]	b + 6s	a[7,3]
	b + 7s	a[7,4]

2-D arrays – Row major

a[$l_1..h_1$, $l_2..h_2$] with element size s

- Number of elements: $e = e_1 * e_2$, where $e_1 = (h_1 - l_1 + 1)$ and $e_2 = (h_2 - l_2 + 1)$
- Size of array: $e * s$
- Size of each dimension (stride):

  $d_1 = e_2 * d_2$

  $d_2 = s$
- Address of element a[i,j], assuming a starts at address b and $l_1 <= i <= h_1$ and $l_2 <= j <= h_2$:

  $b + (i - l_1) * d_1 + (j - l_2) * s$

Example

A[3…100,4…50] with elements size 4

- 98*47 = 4606 elements
- 4606 * 4 = 18424 bytes long
- $d_2 = 4$ and $d_1 = 47 * 4 = 188$
- If a starts at 100, a[5,5] is:

  100+(5-3) * 188 + (5 – 4) * 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*2
- Size of first dimension =
  s * 2 * 2

a[4,x]	b + 0s	a[4,3,8]	a[4,3,x]
	b + 1s	a[4,3,9]	
	b + 2s	a[4,4,8]	a[4,4,x]
	b + 3s	a[4,4,9]	
a[5,x]	b + 4s	a[5,3,8]	a[5,3,x]
	b + 5s	a[5,3,9]	
	b + 6s	a[5,4,8]	a[5,4,x]
	b + 7s	a[5,4,9]	
a[6,x]	b + 8s	a[6,3,8]	a[6,3,x]
	b + 9s	a[6,3,9]	
	b + 10s	a[6,4,8]	a[6,4,x]
	b + 11s	a[6,4,9]	
a[7,x]	b + 12s	a[7,3,8]	a[7,3,x]
	b + 13s	a[7,3,9]	
	b + 14s	a[7,4,8]	a[7,4,x]
	b + 15s	a[7,4,9]	

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: $e = e_1 * e_2 * e_3$, where $e_i = (h_i - l_i + 1)$
- Size of array: $e * s$
- Size of each dimension (stride):

  $d_1 = e_2 * d_2$

  $d_2 = e_3 * d_3$

  $d_3 = s$
- Address of element a[i,j,k], assuming a starts at address b and $l_1 <= i <= h_1$ and $l_2 <= j <= h_2$:

  $b + (i - l_1) * d_1 + (j - l_2) * d_2 + (k - l_3) * s$

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: $e = \Pi\ e_i$ where $e_i = (h_i - l_i + 1)$
- Size of array: $e * s$
- Size of each dimension (stride):

  $d_i = e_{i+1} * d_{i+1}$

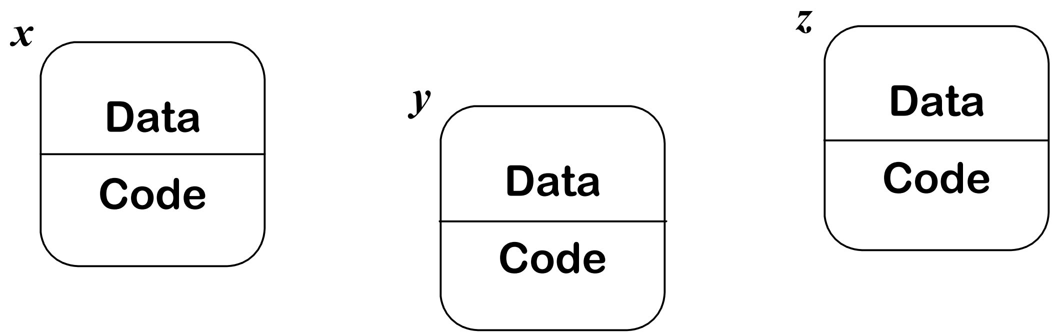
  $d_n = s$

- Address of element a[$i_1, \ldots, i_n$], assuming a starts at address b and $l_j <= i_j <= h_j$:

  $b + (i_1 - l_1) * d_1 + \ldots + (i_n - l_n) * d_n$

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

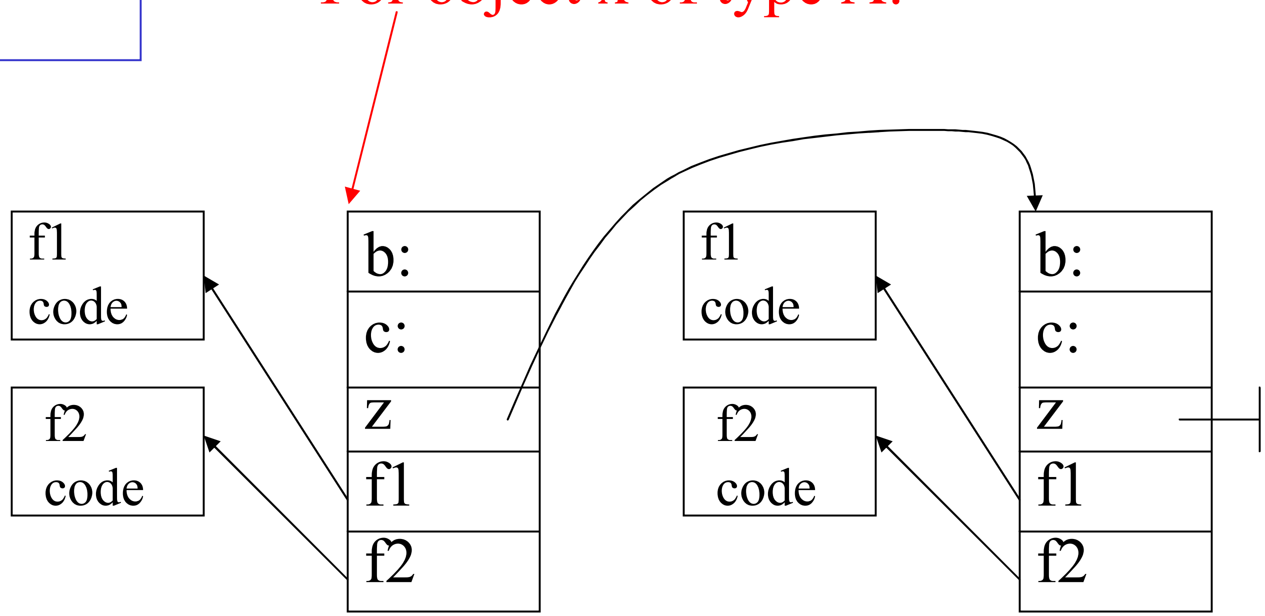
Objects

- 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

```
Class A {
   int b,c;
   A z;
   f1()
   f2()
}
```

Simplistic Object Representation

For object x of type A:

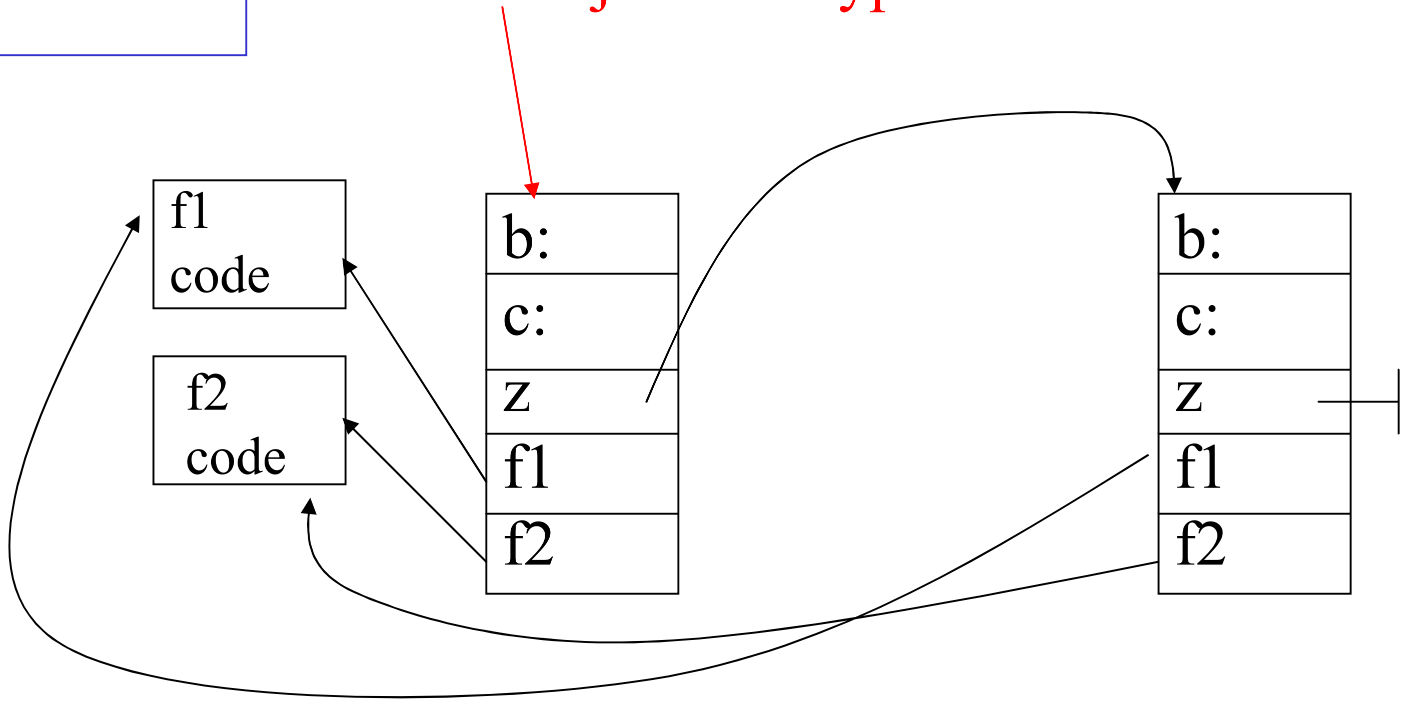


Each object gets copies of all attributes and methods

```
Class A {
  int b,c;
  A z;
  f1()
  f2()
}
```

Better Representation

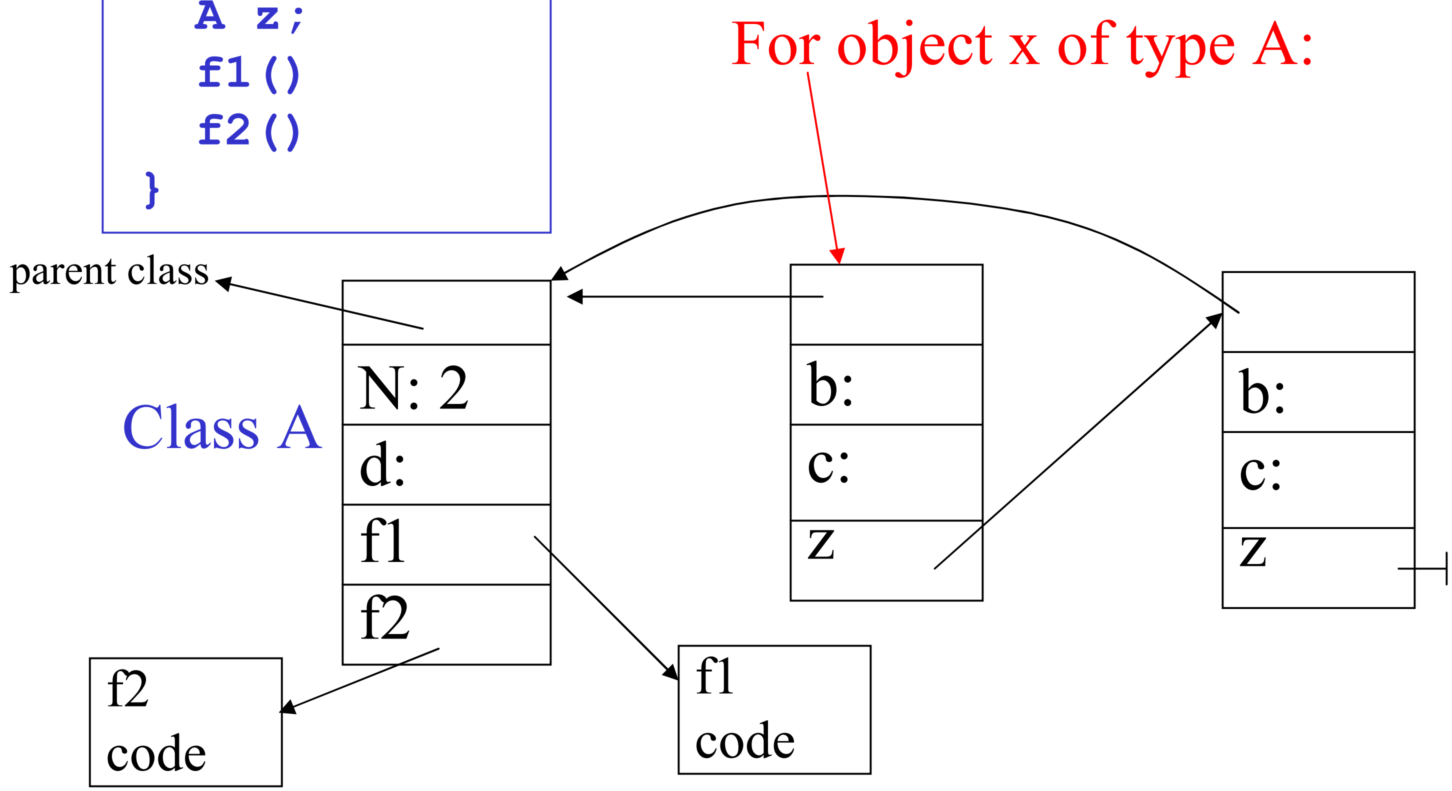
For object x of type A:



Objects share methods

More typically:

```
Class A {
  int b,c;
  static int d
  A z;
  f1()
  f2()
}
```



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

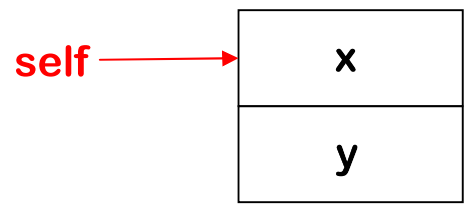
Object Variables

- 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

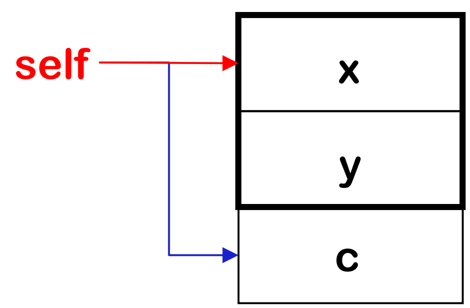
Dealing with Single Inheritance

- Use prefixing of storage for objects

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




```
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

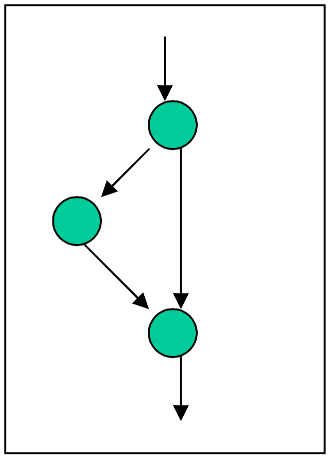
if (y > 0) then begin

…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

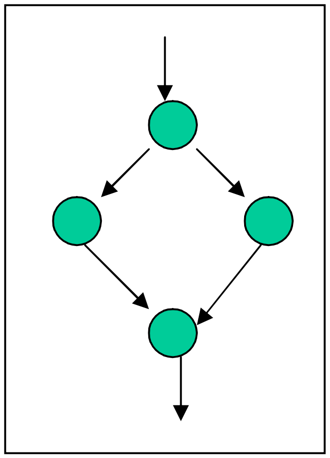
… body-1 …

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

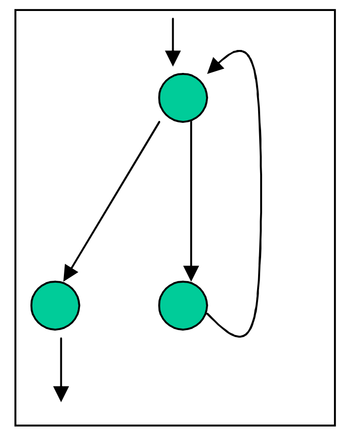
Looping constructs

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

for_stmt → FOR id = start TO stop

```
{ code to eval start ($4) and stop ($6)done
   get two new label names
   output code to initialize id = start
   output label1
   output code to compare id to stop
   output conditional branch to label2}
```

stmts END

```
{ increment id (and save)
  unconditional branch to label1
  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.