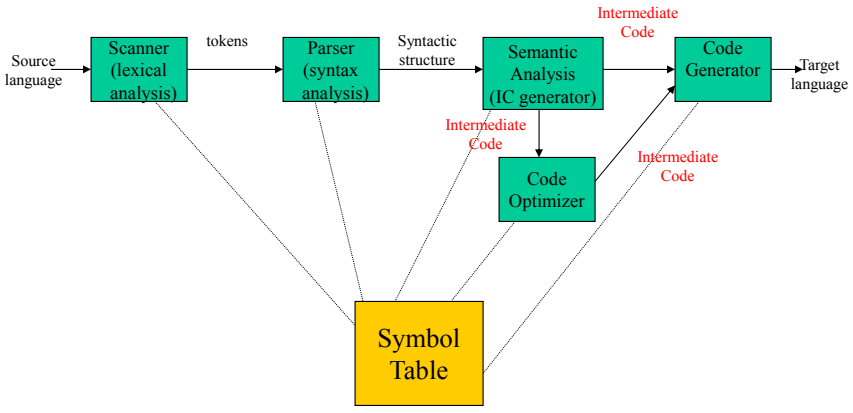
Lecture 5: Intermediate Code Generation

COSC 4316

Sam Houston State University

Partially based on *Introduction to Compiler Construction* by Thomas W. Parsons

Compiler Architecture



- Let us see where we are now.
  - We have tokenized the program and parsed it.
  - We know the structure of the program and of every statement in it,
  - and we have presumably established that it is free of grammatical errors.
  - It would appear that we are ready to start translating it.

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 Code Generation

- Intermediate Code Generation
  - Up to this point, the discussions related to parsing the input to verify syntax.
  - The next stage to consider is semantics – how is the input structured.
  - So far, the act of parsing involved managing states and what to push or pop to the stack – basically determining structure.
  - When considering semantics, the associated stages need to have attributes attached to them so that meaning becomes apparent.
  - Can attach a meaning to every production – first step towards translating the program.

Intermediate Code Generation

- Intermediate Code Generation
  - Some questions to consider:
    - Generate intermediate code during the course of the parse?
    - Produce an intermediate representation like a parse tree and then get the intermediate code?
    - Or go directly from the parse tree to the object code?
    - What's the scale of optimization?
    - Should the compiler be re-targetable? (e.g. different back ends for different target machines)
  - All these alternatives are based on the syntactic information uncovered during the parse

Our Approach

- We can apply a meaning to virtually every production
- e.g., the production E → E + E is used when the source program contains an addition.
  - Obviously, the programmer wants to add something.
  - So, the production itself tells us that the compiled program must do an addition at this point, and it tells us what to add.
- There must be a correspondence between productions and elementary computations (or between productions and some housekeeping action on the part of the compiler itself.)

Semantic Actions and Syntax-Directed Translation

- We can apply a meaning to virtually every production
- ***Syntax-directed translation*** – sequence of productions that guide the generation of intermediate code. We attach an appropriate interpretation to each production.
- For example
  - $E \rightarrow E_1 + E_2$ $\{E := E_1 + E_2\}$
  - $E \rightarrow E_1 * E_2$ $\{E := E_1 * E_2\}$
  - $E \rightarrow (E_1)$ $\{E := E_1\}$
  - $E \rightarrow$ **id** $\{E :=$ **id**.val$\}$

Semantic Actions and Syntax-Directed Translation

- Suppose we repeat the bottom up parse of **id + id * id** for **a+b*c**

Stack	Input	Production	Semantic Action
$	id + id * id$		
$id	+ id * id$	$E_1 \rightarrow$ **id**	$E_1 := a$
E_1	+ id * id$		
E_1 +	id * id$		
E_1 + id	* id$	$E_2 \rightarrow$ **id**	$E_2 := b$
E_1 + E_2	* id$		
E_1 + E_2 *	id$		
E_1 + E_2 * id	$	$E_3 \rightarrow$ **id**	$E_3 := c$
E_1 + E_2 * E_3	$	$E_4 \rightarrow E_2 * E_3$	$E_4 := E_2 * E_3$
E_1 + E_4	$	$E_5 \rightarrow E_1 + E_4$	$E_5 := E_1 + E_4$
E_5	$		

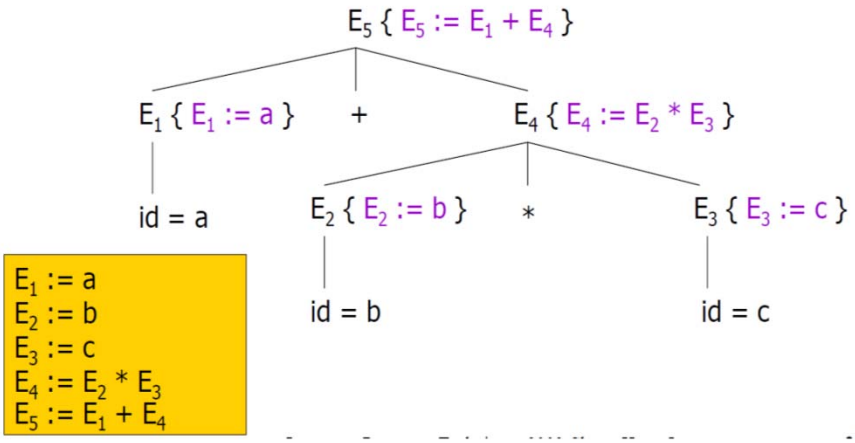
And the sequence of instructions:

E_1 := a
E_2 := b
E_3 := c
E_4 := E_2 * E_3
E_5 := E_1 + E_4

does in fact, compute **a+b*c**
(assuming we think of E_1 and the rest as representing memory locations instead of nonterminals.)

Semantic Actions and Syntax-Directed Translation

- It may help to look at the parse tree again

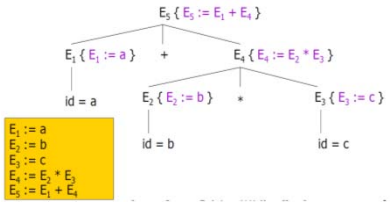


We have decorated the tree with the ***semantic actions*** that correspond to the productions.

The parse tree itself tells us how the expression arose out of the grammar and the ***decorations*** show us how the value of the expression is computed.

Semantic Actions and Syntax-Directed Translation

- It may help to look at the parse tree again



The computations or other operations attached to the productions impute meaning to each production, and so these operations are called ***semantic actions***

The information obtained by the semantic actions is associated with the symbols of the grammar

it is normally put in fields of records associated with the symbols; these fields are called ***attributes***

Note: as far as the parser is concerned, neither the semantic actions nor the attributes are a part of the grammar.

Attributes are only used as a device for bridging the gap between parsing and constructing an intermediate representation.

Semantic Actions and Syntax-Directed Translation

- Things we must take care of with the semantic actions:
  - making sure the variables are declared before use.
  - type checking
  - making sure actual and formal parameters are matched
- These things are called ***semantic analysis***
- So we can now have it both ways, we can put context dependent information and actions together into a language that is still context free.

Semantic Actions and Syntax-Directed Translation

- Of course there are still some unanswered questions:
  - What if the statement is a declaration like
    int i, j, k;
    where there is no executable code?
- The answers to some of these questions depend on how the parse is done and what the parser output is.
  - With a recursive descent parser, we can embed the semantic actions in the code for each nonterminal.
  - If a tree, we attach the semantic actions to the appropriate nodes

Intermediate Representations

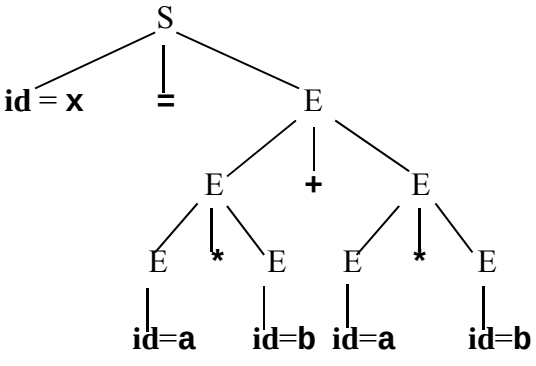
- Before we pursue these question, we must look at the form of output the parser takes
- We will look at several different representations
  - Syntax Trees
  - Directed Acyclic Graphs
  - Postfix notation
  - Three-Address Code
  - Other Forms.

Syntax Trees

- (Or abstract syntax tree)
- typically used when intermediate code is to be generated later (maybe after an optimization pass)
- Same general form as the parse tree, but the operators take the place of the non-terminals in the interior nodes.
  - Basically, the operators 'move up'.

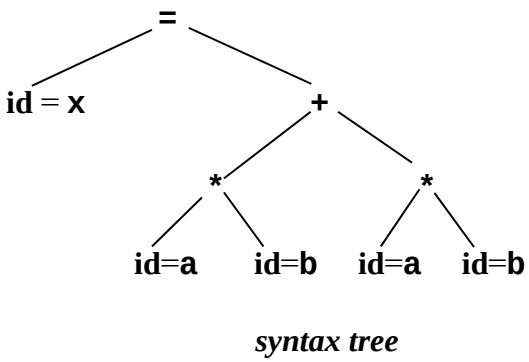
Syntax Tree

for the statement **`x = a*b+a*b`**



parse tree

emphasis is on the grammatical structure of the statement

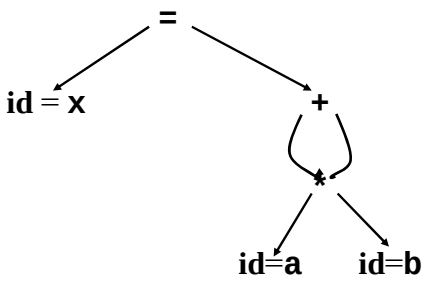


syntax tree

emphasis is on the actual computation to be performed

Directed Acyclic Graphs

- The ***directed acyclic graph*** (DAG) is a relative of a Syntax Tree.
- The difference is that nodes for variables or repeated subexpressions are merged.
- Before constructing a node for anything, the procedure first sees whether such a node is already in existence



Directed Acyclic Graphs

- The use of DAG's to eliminate redundant code is our first instance of optimization. We will see more optimization later.
- Redundant code really comes into play when we do array subscripts.
- When you start generating intermediate code, you will be amazed at how much is generated for array subscripts.

Postfix Notation

- Postfix notation, also know as reverse Polish notation, is another form of intermediate representation
  - Jan Lukasiewics, Polish mathematician and logician
- In postfix notation, every expression is rewritten with the operator at the end
- Very easy to generate from a Bottom-Up parse.

Postfix Notation

- You can also generate it from a Syntax Tree via a post-order traversal.
- The chief virtue of postfix is that it can be evaluated with the use of a stack.
  - Operands are pushed onto the stack
  - Operators pop the required number of operands from the stack, do the operation, and push the result onto the stack.
- Nested if statements can cause problems.

Postfix Notation

- Very easy to generate from a Bottom-Up parse. We can attach the semantic actions to the productions.

$S \rightarrow$ **i** = { output("lvalue", i.lexeme)} E { output("sto")}

$E \rightarrow E + E$	{ output("add"}
$E \rightarrow E * E$	{ output("mult"}
$E \rightarrow$ **(** E **)**	{ /* nothing */ }
$E \rightarrow$ **i**	{ output("rvalue", i.lexeme)}

$S \rightarrow \mathbf{i} = E$	{output ('=', **i**.*lexeme*)}
$E \rightarrow E + E$	{output ('+')}
$E \rightarrow E * E$	{output ('*')}
$E \rightarrow (E)$	{do nothing}
$E \rightarrow \mathbf{i}$	{output (**i**.*lexeme*)}

Postfix Notation

- The reason such a simple scheme works is this: In postfix notation, before we list an operator, we must first list all its operands.
- But this is exactly the sequence that is followed doing reductions in a bottom-up parse: we never reduce a production E op E until we have obtained the E's by reductions further down the tree.

Postfix Notation for **x = (a+b)*(c+d)**

Stack	Input	Production	Action
$	i = (i+i)*(i+i)$		
$i	= (i+i)*(i+i)$		
$i=	(i+i)*(i+i)$		lvalue x
$i=(	i+i)*(i+i)$		
$i=(i	+i)*(i+i)$	$E \rightarrow$ i	rvalue a
$i=(E	+i)*(i+i)$		
$i=(E+	i)*(i+i)$		
$i=(E+i	)*(i+i)$	$E \rightarrow$ i	rvalue b
$i=(E+E	)*(i+i)$	$E \rightarrow E + E$	add
$i=(E	)*(i+i)$		
$i=(E)	*(i+i)$	$E \rightarrow$ (E)	
$i=E	*(i+i)$		

Postfix Notation for **x = (a+b)*(c+d)**

Stack	Input	Production	Action
$i=E	*(i+i)$		
$i=E*	(i+i)$		
$i=E*(	i+i)$		
$i=E*(i	+i)$	$E \rightarrow$ i	rvalue c
$i=E*(E	+i)$		
$i=E*(E+	i)$		
$i=E*(E+i	)$	$E \rightarrow$ i	rvalue d
$i=E*(E+E	)$	$E \rightarrow E + E$	add
$i=E*(E	)$		
$i=E*(E)	$	$E \rightarrow$ (E)	
$i=E*E	$	$E \rightarrow E * E$	
$i=E	$	$S \rightarrow$ i = E	
$S	$		

Three-Address Code

- Three-address code (3AC) breaks the program down into elementary statements having no more than 3 variables and no more than one operator.
- Sample Statement: x = a + b * c
- 3AC Translation:
  - T := b * c
  - x := a + T
- Note: T is a temporary variable.

Three-Address Code

- The notation is a compromise; it has the general form of a high-level language, but the individual statements are simple enough that they map into assembly language in a reasonably straight forward manner.
- 3AC may be:
  - Generated from a traversal of a Syntax Tree or a DAG.
  - or it may be generated as intermediate code directly in the course of the parse.

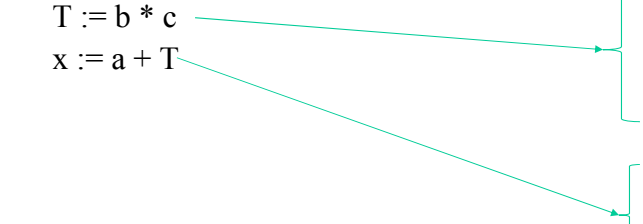
Example

Sample Statement: x = a + b * c

3AC Translation:

T := b * c

x := a + T



- x86 assembly code

```
mov     eax,[b]
cdq
imul    dword [c]
mov     [T], eax
mov     eax,[a]
add     eax,[T]
mov     [x], eax
```


Intermediate Languages

- Sometimes the Intermediate representation may be a language of its own.
- This helps uncouple the front end of the compiler from the back end.
- You can then have a front end for each language that generate the same intermediate language, and then one back end for each type of computer.

- Examples:
  - UNCOL (1961) -- UNiversal Compiler-Oriented Language.
  - P-Code (1981) -- UCSD -- based upon a p-code interpreter (they also built p-code compilers.)
  - GNU Intermediate Code -- gcc, g++, g77, gada,
    - -- a Lispish type intermediate language.
  - Java byte code (1995) – borrowed heavily from p-code
- Intermediate codes can be either interpreted or translated.

Bottom-Up Translation

- We need to keep track of the various elements or pieces of the intermediate representation we are using, so we can get at them when we need them.
- These elements will be attributes of symbols in the grammar.
  - For an identifier, the attribute will usually be its address in the symbol table.
  - For a non-terminal, the attribute will be some appropriate reference to part of the intermediate representation.

Bottom-Up Translation

- The most convenient way to keep track of these attributes is by keeping them in a stack (known as the *semantic stack*).
- In the case of bottom-up parsing, the semantic stack and the parser stack move in synchronism.
  - When we pop from the parse stack we pop the semantic stack, and when we push something onto the parse stack we will push something onto the semantic stack

Semantic Stack Example

- Some parsers put

AN EXTENDED EXAMPLE USING A MIPS SUBSET

Intermediate Code Generation

- Terms:
  - Syntax-directed translation – sequence of productions that guide the generation of intermediate code.
  - Semantic actions – computations or other operations that impute a meaning to each operation.
  - Semantic stack – a stack that maintains/holds the attributes associated with semantical parsing.
  - Synthesized attributes – attributes put together from things in the production itself.
  - Inherited Attributes – attributes transferred between siblings or from parent to child.

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]

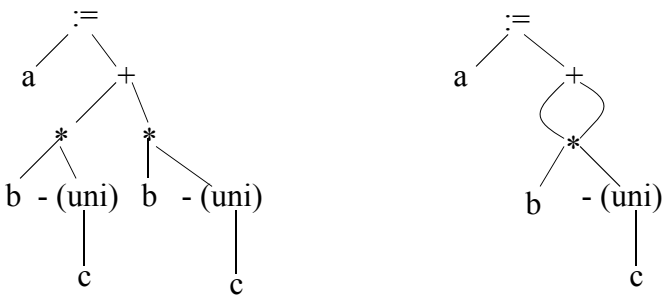
Intermediate Languages Types

- 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
```


MIPS Subset

- 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

MIPS 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`**

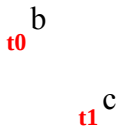
a := b *-c + b*-c

```
lw $t0,b
```

b
t0

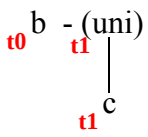
a := b *-c + b*-c

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

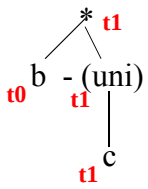


a := b *-c + b*-c

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

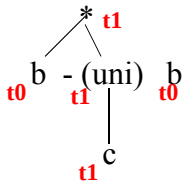


a := b *-c + b*-c



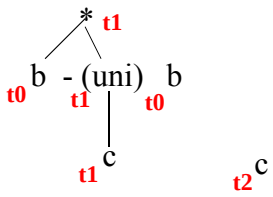

```
lw $t0,b
lw $t1,c
neg $t1,$t1
mul $t1, $t1,$t0
```


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



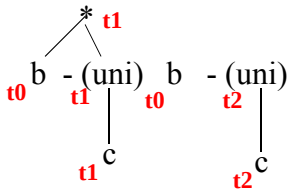

```
lw $t0,b
lw $t1,c
neg $t1,$t1
mul $t1, $t1,$t0
lw $t0,b
```

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



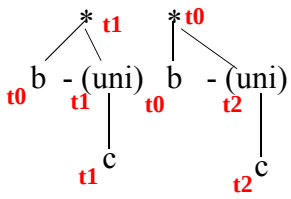

```
lw $t0,b
lw $t1,c
neg $t1,$t1
mul $t1, $t1,$t0
lw $t0,b
lw $t2,c
```


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



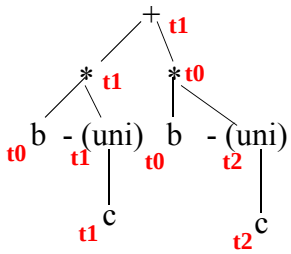

```
lw $t0,b
lw $t1,c
neg $t1,$t1
mul $t1, $t1,$t0
lw $t0,b
lw $t2,c
neg $t2,$t2
```

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



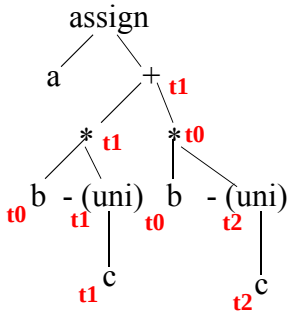

```
lw $t0,b
lw $t1,c
neg $t1,$t1
mul $t1, $t1,$t0
lw $t0,b
lw $t2,c
neg $t2,$t0
mul $t0,$t0,$t2
```


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
```

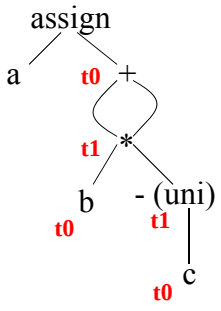
a := b *-c + b*-c




```
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 MIPS

- 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

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 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:
```


MIPS 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

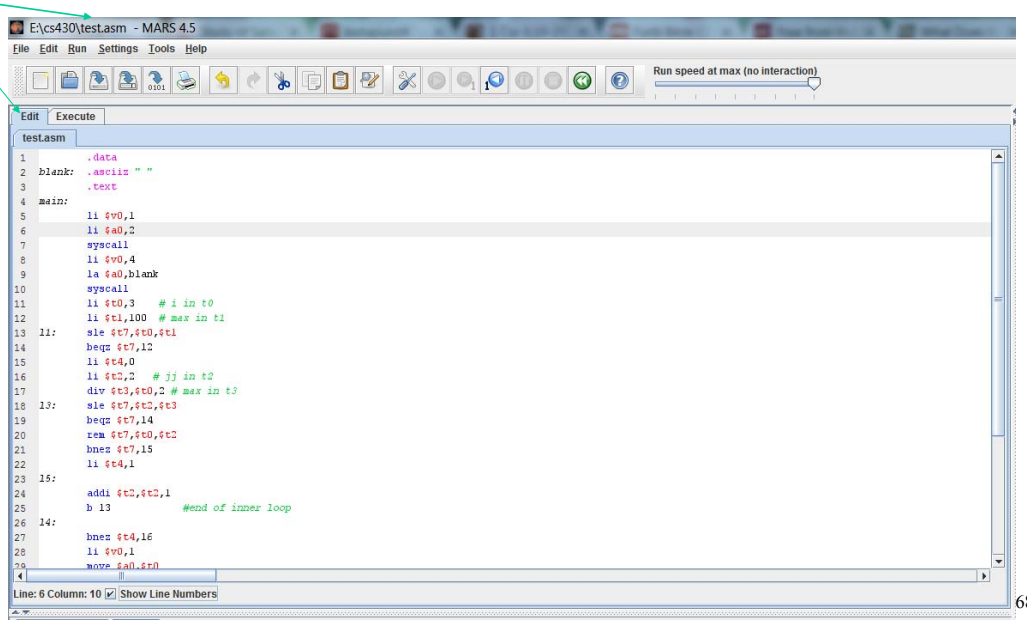

```
   .data
blank:  .asciiz " "
   .text
   li $v0,1
   li $a0,2
   syscall          # print 2
   li $v0,4
   la $a0,blank   # print blank
   syscall

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

   li $v0,10
   syscall     # exit
```

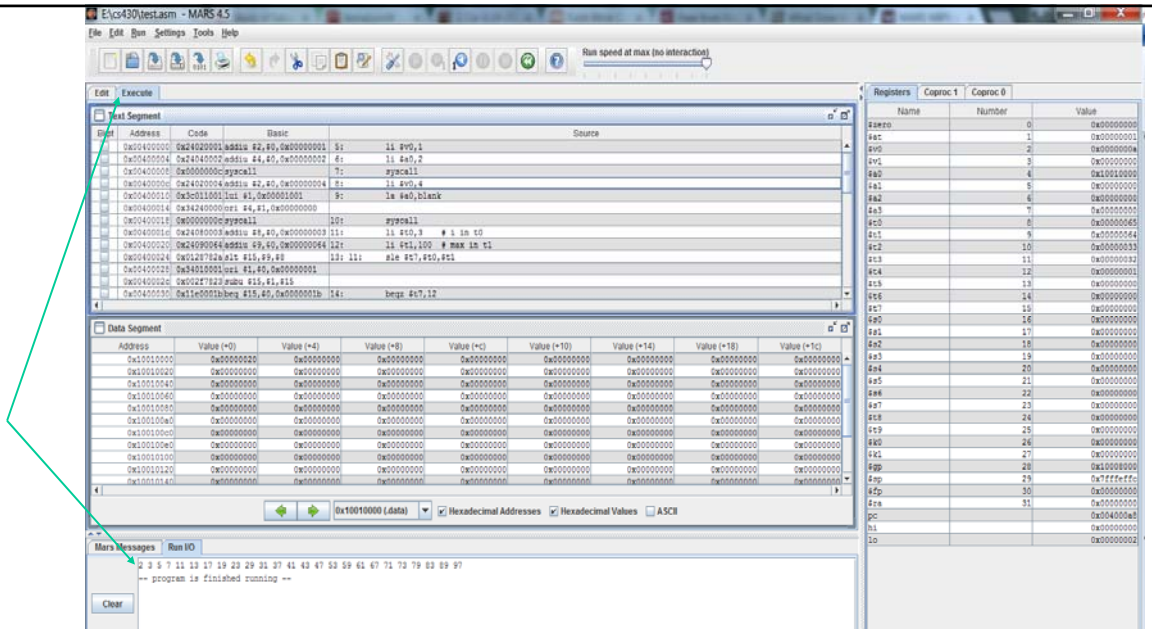
```
        .data
blank:          .asciiz " "
        .text
main:
        li $v0,1
        li $a0,2
        syscall
        li $v0,4
        la $a0,blank
        syscall
        li $t0,3    # i in t0
        li $t1,100  # max in t1
l1:     sle $t7,$t0,$t1
        beqz $t7,l2
        li $t4,0
        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
```

inner loop

Entire program

Can run this through the MARS simulator





Notes

- MIPS 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

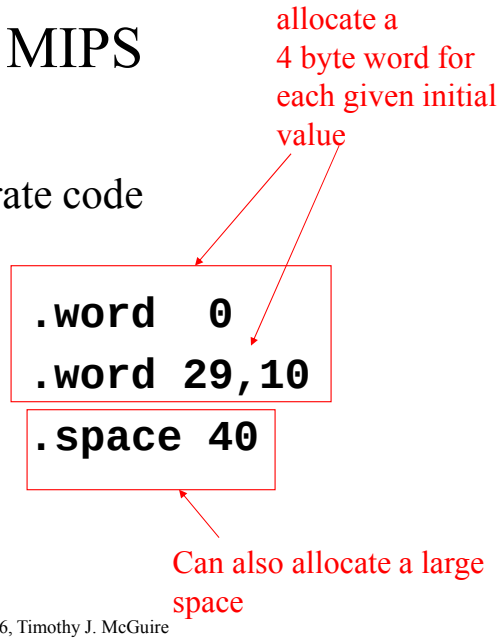
Processing Declarations

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

In MIPS

- Declarations generate code in **`.data`** sections

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

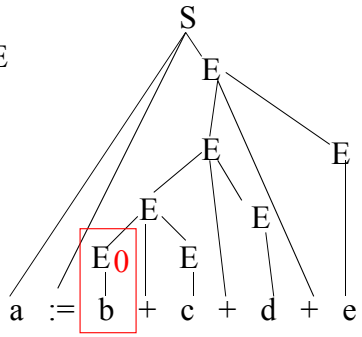


Issues in Processing Expressions

- 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

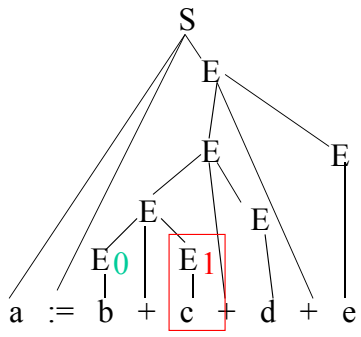



```
Generate:
  lw $t0,b
```

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

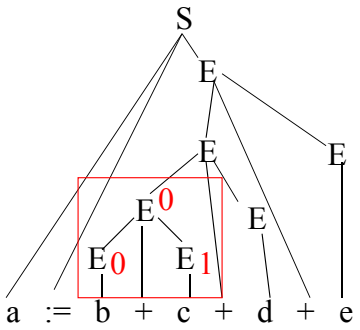



```
Generate:
  lw $t0,b
  lw $t1,c
```

Each number corresponds to a temporary variable.

Expressions

Grammar:
S → id := E
E → E + E
E → id

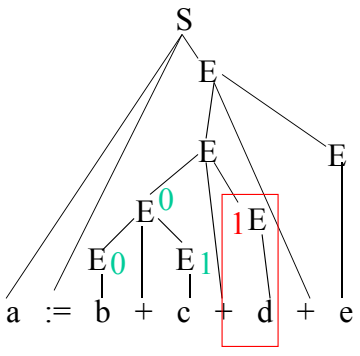



```
Generate:
  lw $t0,b
  lw $t1,c
  add $t0,$t0,$t1
```

Each number corresponds to a temporary variable.

Expressions

Grammar:
S → id := E
E → E + E
E → id

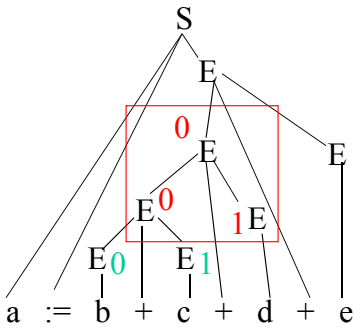



```
Generate:
  lw $t0,b
  lw $t1,c
  add $t0,$t0,$t1
  lw $t1,d
```

Each number corresponds to a temporary variable.

Expressions

Grammar:
S → id := E
E → E + E
E → id

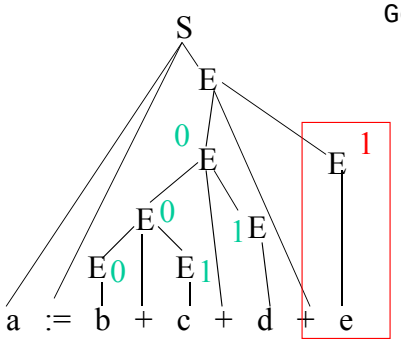



```
Generate:
  lw t0,b
  lw t1,c
  add $t0,$t0,$t1
  lw t1,d
  add $t0,$t0,$t1
```

Each number corresponds to a temporary variable.

Expressions

Grammar:
S → id := E
E → E + E
E → id

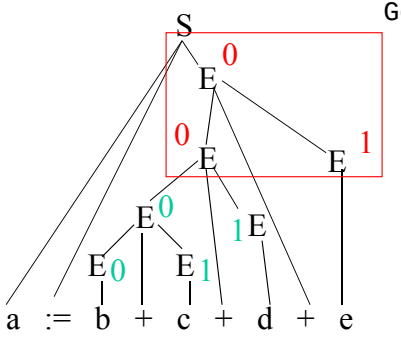



```
Generate:
  lw $t0,b
  lw $t1,c
  add $t0,$t0,$t1
  lw $t1,d
  add $t0,$t0,$t1
  lw $t1,e
```

Each number corresponds to a temporary variable.

Expressions

Grammar:
S → id := E
E → E + E
E → id

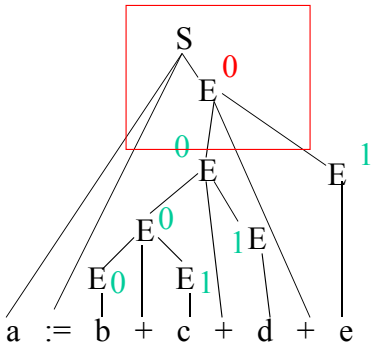



```
Generate:
  lw $t0,b
  lw $t1,c
  add $t0,$t0,$t1
  lw $t1,d
  add $t0,$t0,$t1
  lw $t1,e
  add $t0,$t0,$t1
```

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: MIPS

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 MIPS

- 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 MIPS

- 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

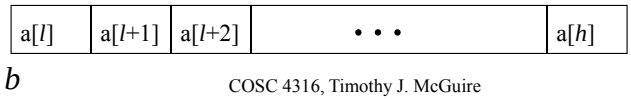
  - Version 2: similar to scalars

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 $l <= i <= h$:

  $b + (i - l) * s$



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

100 104

1-D arrays in MIPS

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]

a[4,x]
a[5,x]
a[6,x]
a[7,x]

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

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

- Allocation

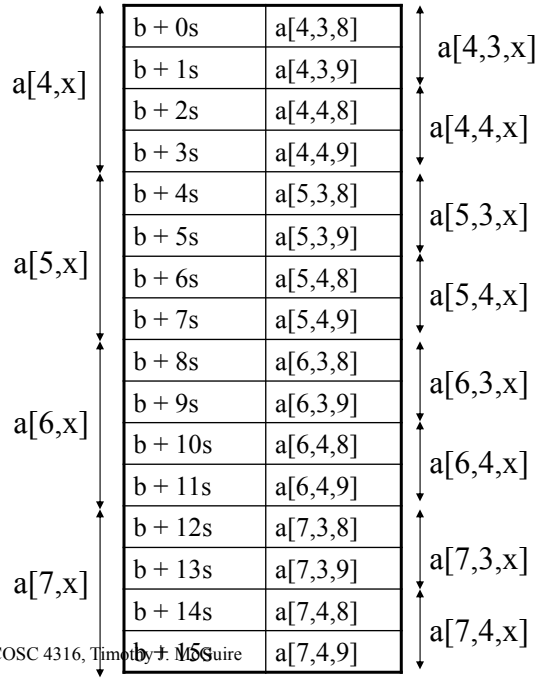
```
    .data
a:  .space 60    # 15 word-size elements * 4
```

- 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



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

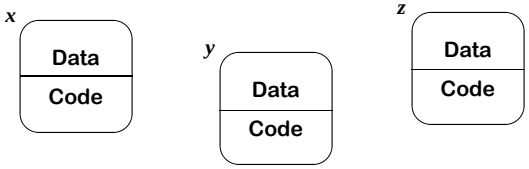
  $d_i = e_{i+1} * d_{i+1}$

  $d_n = s$
- 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:

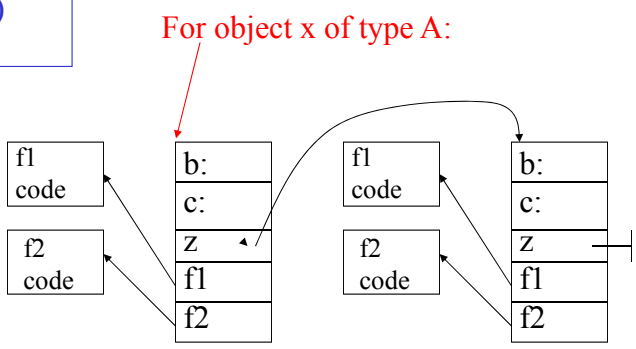
- Each **object** needs local storage for its attributes
  - Attributes are static (*lifetime of object*)
  - Access is through methods
- Some methods are public, others are private
- Object's internal state leads to complex behavior

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

Simplistic Object Representation

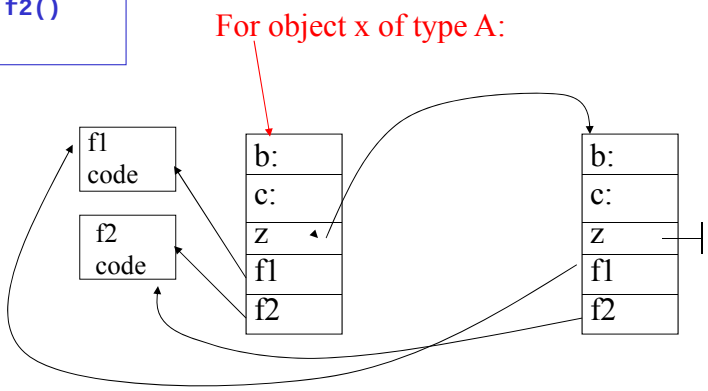
```
Class A {
  int b,c;
  A z;
  f1()
  f2()
}
```



Each object gets copies of all attributes and methods

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

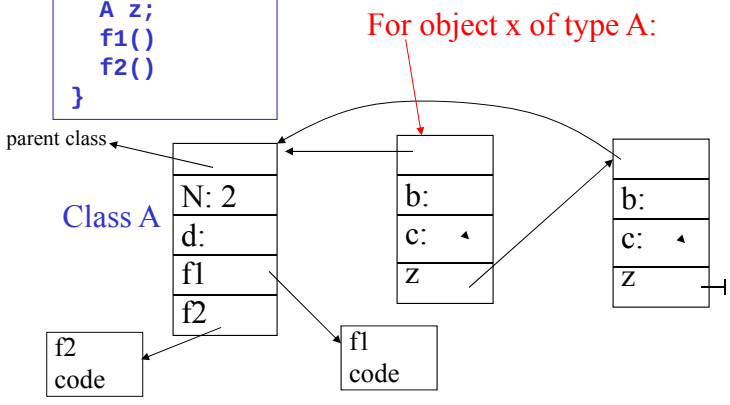
Better Representation



Objects share methods


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

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

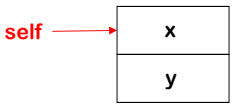
Object Variables

- 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

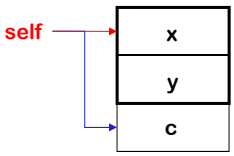
Dealing with Single Inheritance

- Use prefixing of storage for objects

```
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

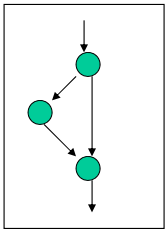
Conditional Examples

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

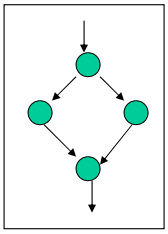
Conditional Examples

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

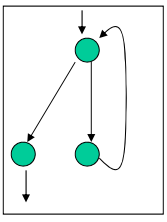
Looping constructs

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.