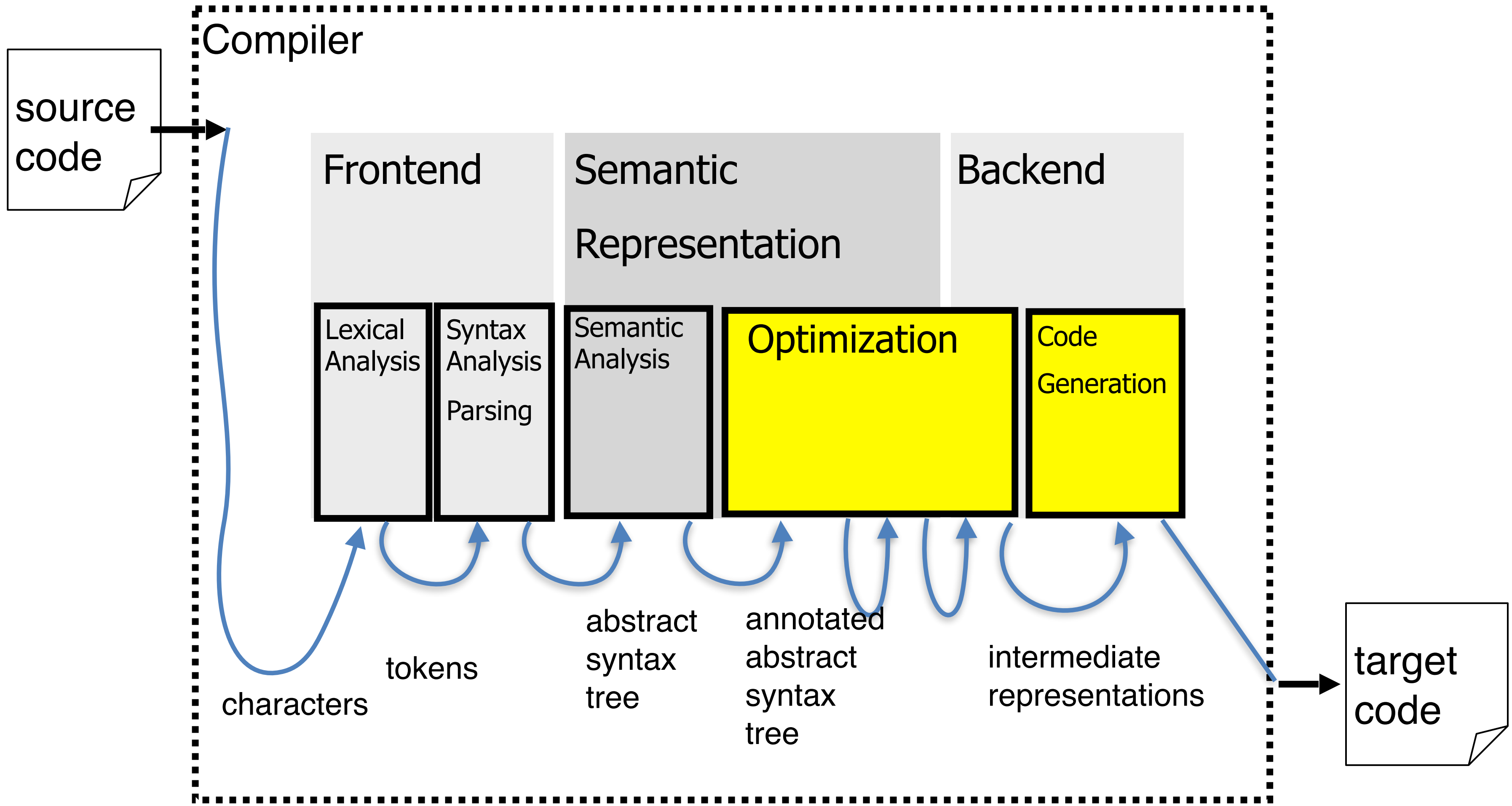
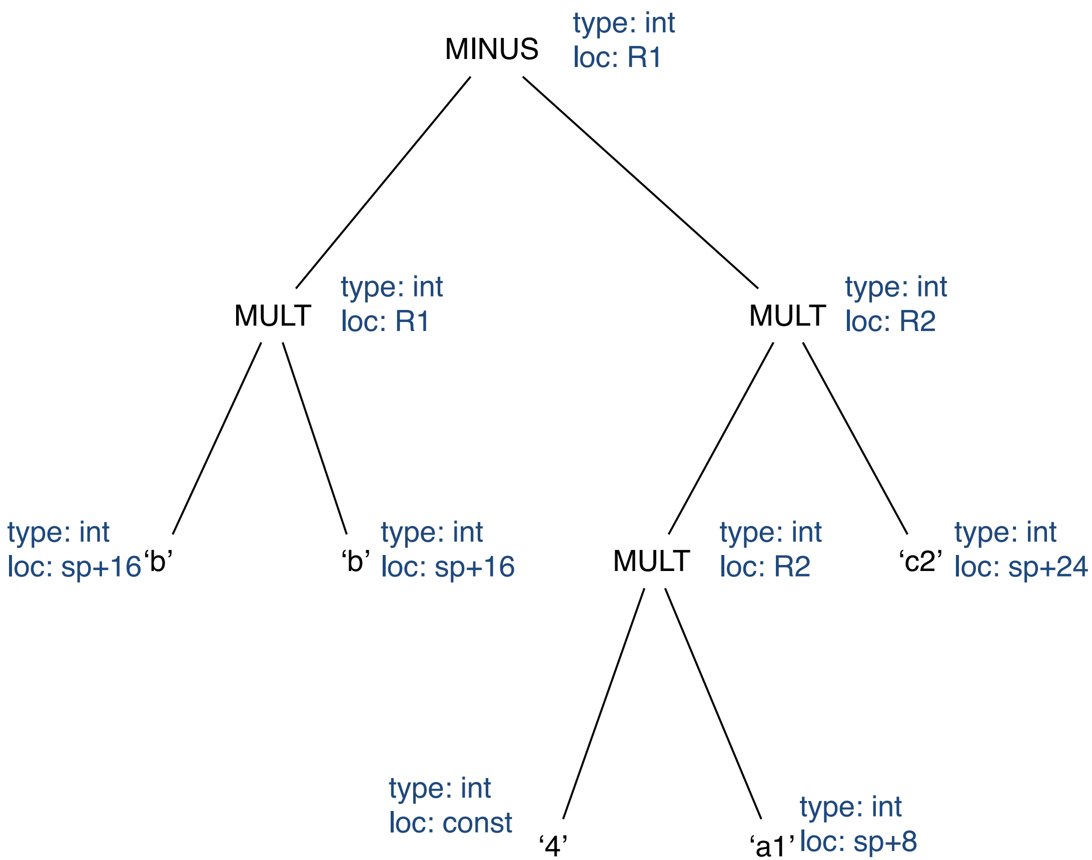
Code generation and optimization

Greta Yorsh



source code
Compiler
Frontend
Semantic Representation
Backend
Lexical Analysis
Syntax Analysis
Parsing
Semantic Analysis
Optimization
Code Generation
characters
tokens
abstract syntax tree
annotated abstract syntax tree
intermediate representations
target code

Code generation

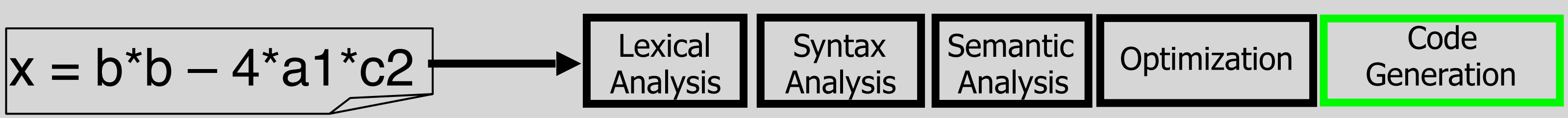


Intermediate Representation

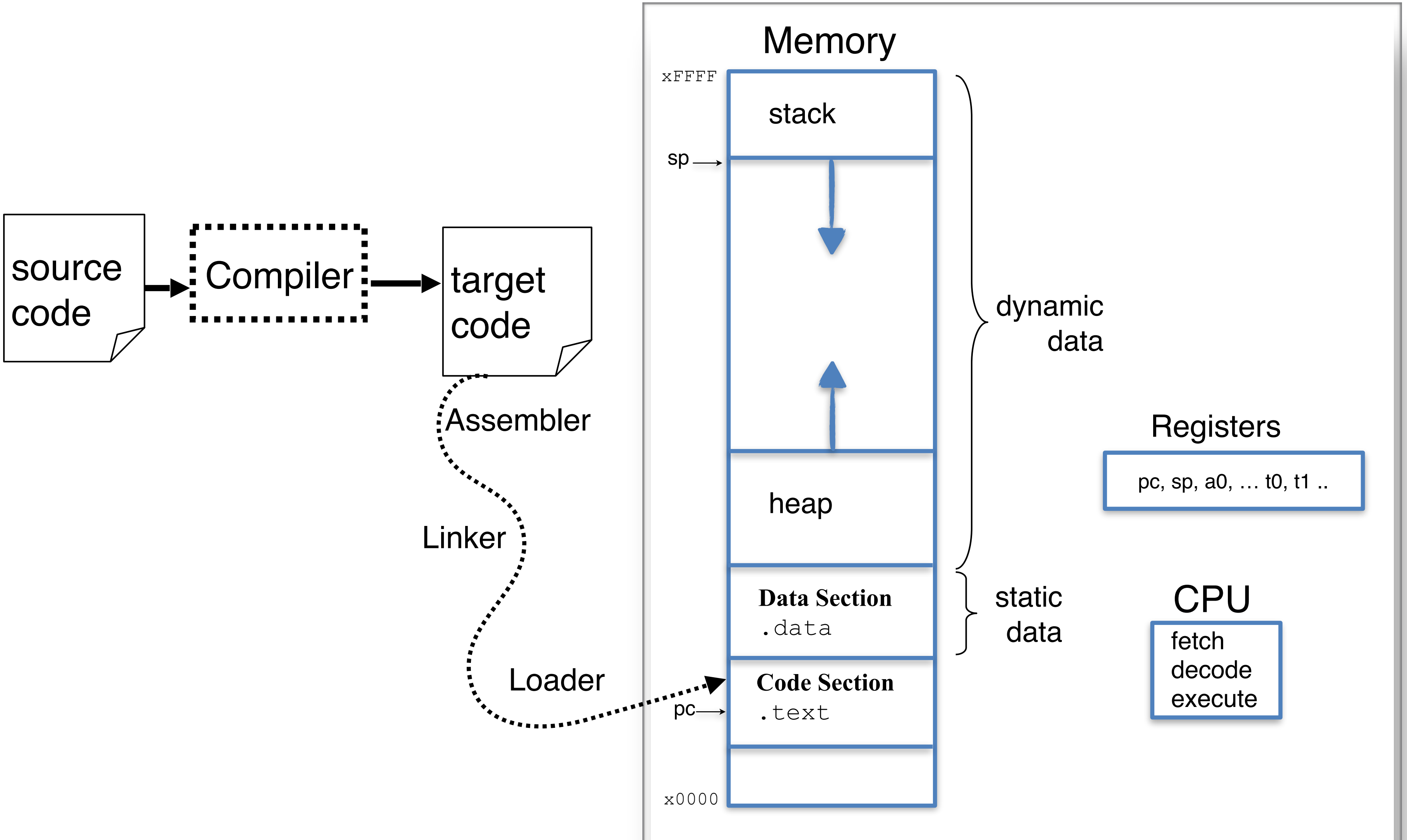
R2 = 4*a1
R1=b*b
R2= R2*c2
R1=R1-R2

Assembly Code

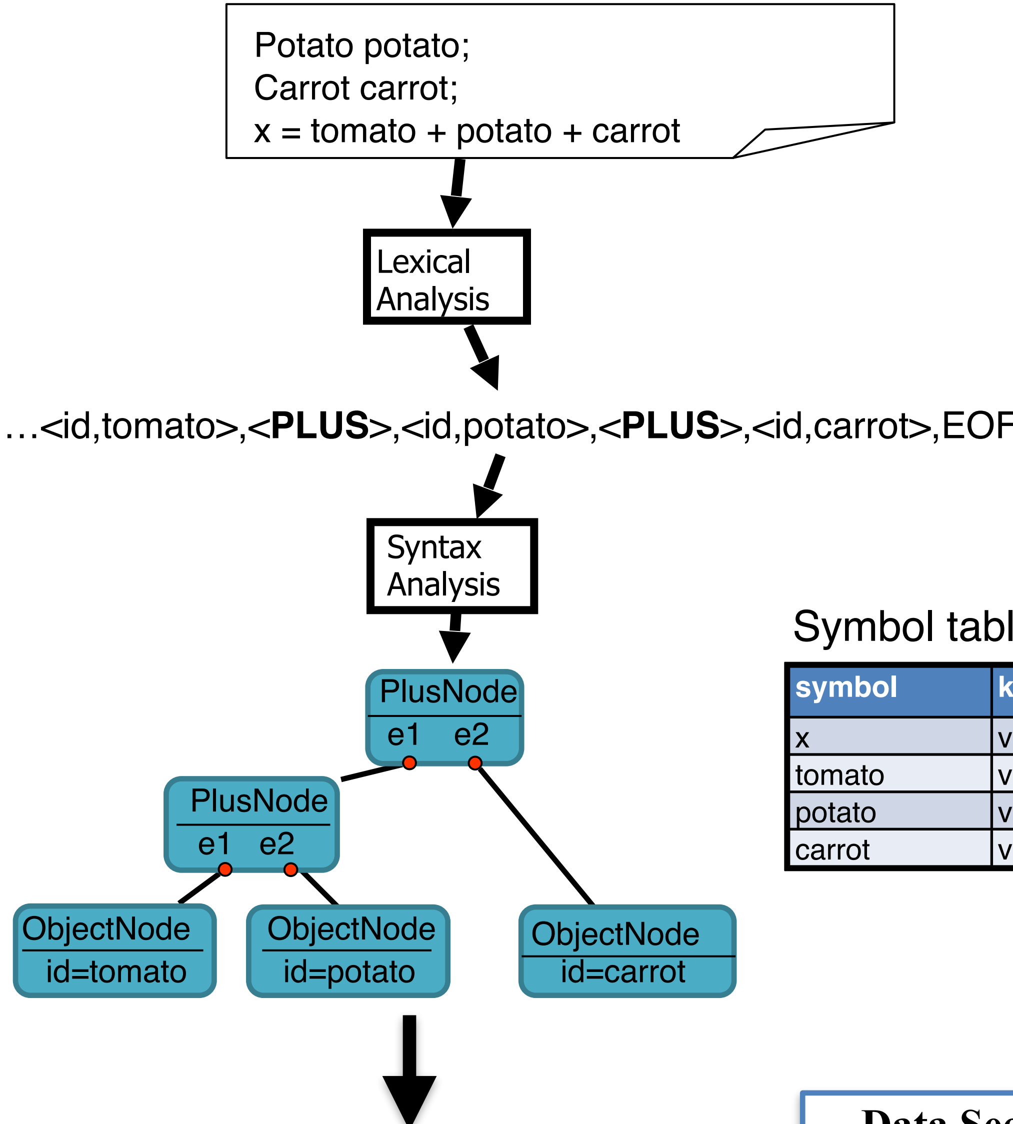
```
lw     $t0,  8($sp)
sll    $t0, $t0, 2
lw     $t1, 16($sp)
mul  $t1, $t1, $t1
lw     $t2, 24($sp)
mul  $t1, $t1, $t2
subu $t0,$t1
```



Hardware



*disclaimer: very abstract view




```
lw  $a0 tomatoes($s0)
lw  $s1 potatoes($s0)
add $s1 $a0 $s1
jal Object.copy
sw  $s1 12($a0)
…
```

MIPS

Symbol tables

symbol	kind	type
x	var	?
tomato	var	?
potato	var	Potato
carrot	var	Carrot

Inheritance Graph

TypeId	GraphNode
Int	*
Bool	*
MyClass	*

Data Section `.data`	Statically allocated data - constants - prototype objects - dispatch tables
Code Section `.text`	Code - code for methods - bootstrapping code - error handlers } trap.handler - garbage collector } trap.handler

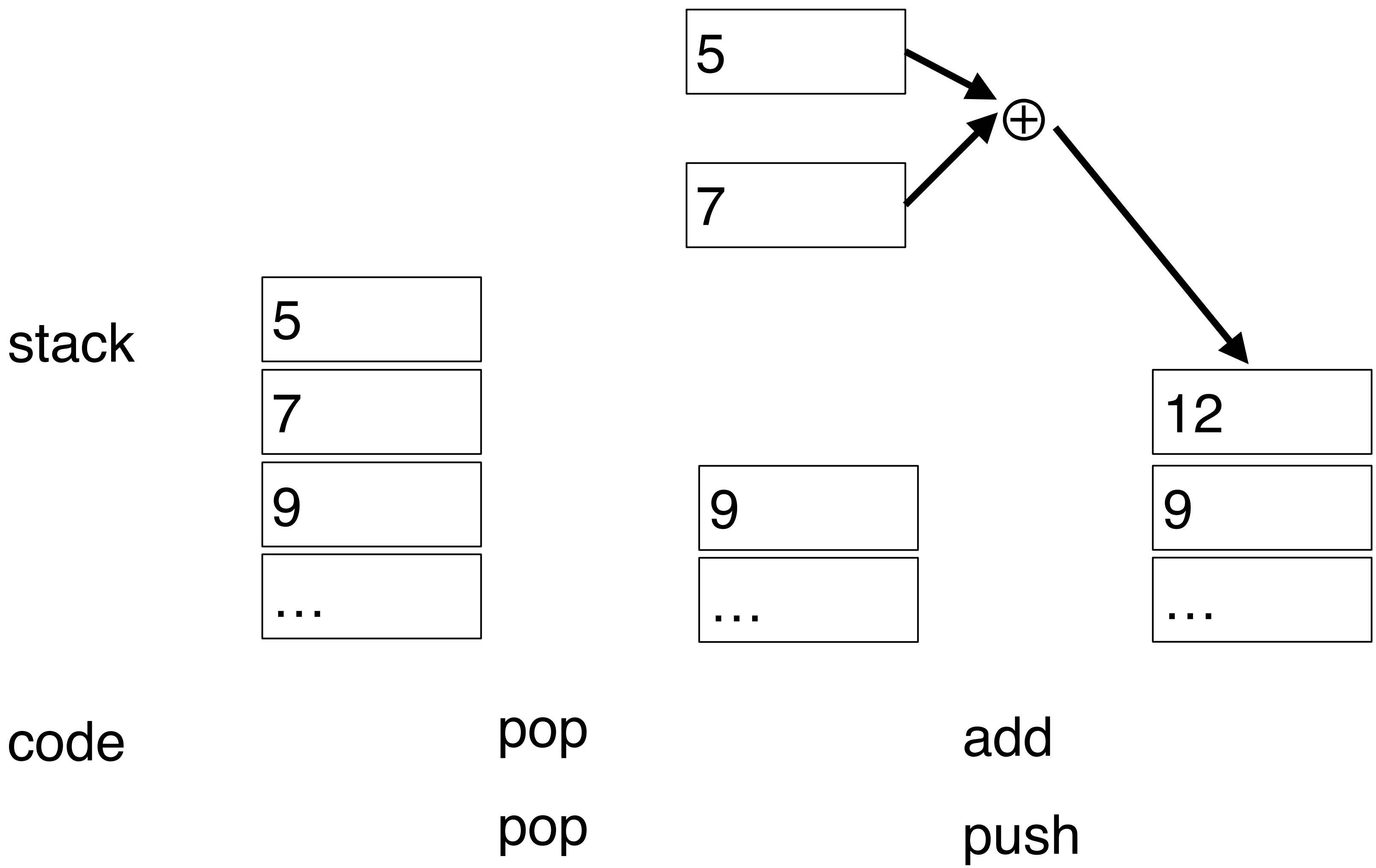
Outline: code generation

- Expressions
  - simple stack machine
  - stack machine with accumulator
  - weighted register allocation
- Objects
- Method calls
- Runtime system

Stack machines

- A simple evaluation model
  - no variables or registers
  - a stack of values for intermediate results
  - stack manipulation: push, pop, top
- Each instruction
  - takes its operands from the top of the stack
  - removes those operands from the stack
  - computes the required operation on them
  - pushes the result on the stack

Stack machine: add



Why use a stack machine?

- Each instruction takes operands from the same place and puts results in the same place
- Location of the operands is implicit: always on the top of the stack
- No need to specify operands explicitly
- No need to specify the location of the result
- Instruction “**add**” as opposed to “**add r1 r2 r3**”
- Advantages
    - uniform compilation scheme
    - simpler compiler
    - compact target code
    - smaller encoding of instructions
- Java Bytecodes use a stack evaluation model

so, what’s wrong?

Optimizing the stack machine

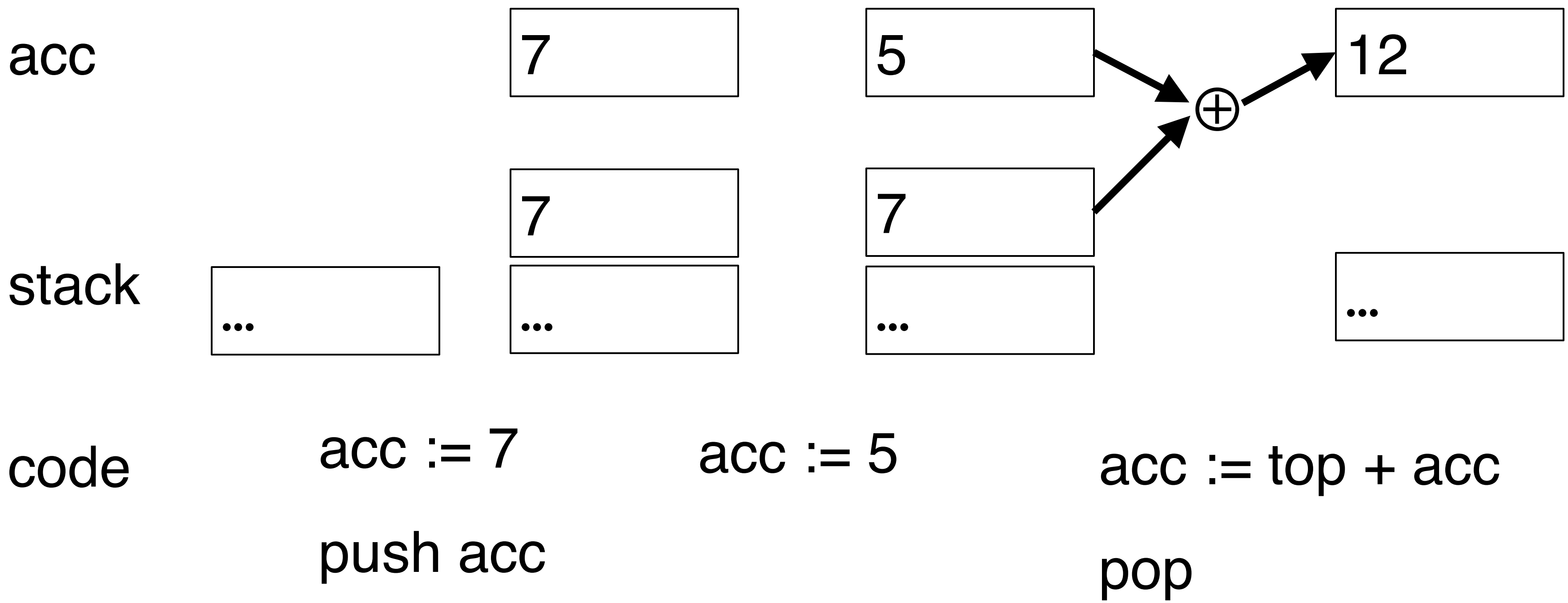
- The add instruction does 3 memory accesses
  - two reads and one write to the stack
  - the top of the stack is frequently accessed
- Idea: keep the top of the stack in a register
  - called accumulator or **acc**
  - register accesses are faster
- The “add” instruction is now
  - acc := acc + top
  - only one memory operation!

Stack machine with accumulator

- The result of computing an expression is always in the accumulator
- For an operation op(e_1,…,e_n)
  - push accumulator on the stack after computing each of e_1,…,e_{n-1}
  - the result of e_n is in the accumulator before op
  - pop n-1 values after the operation
- After computing an expression the stack is as before

Stack machine with accumulator

- Compute 7 + 5 using an accumulator



A bigger example: 3 + (7 + 5)

Code	Accumulator	Stack
`acc := 3`	3	<init>
`push acc`	3	3,<init>
`acc := 7`	7	3,<init>
`push acc`	7	7,3,<init>
`acc := 5`	5	7,3,<init>
`acc := top + acc`	12	7,3,<init>
`pop`	12	3,<init>
`acc := top + acc`	15	3,<init>
`pop`	15	<init>

Invariant

- The stack is preserved across the evaluation of a subexpression
  - Stack **before** the evaluation of 7 + 5 is 3, <init>
  - Stack **after** the evaluation of 7 + 5 is 3, <init>

From stack machines to MIPS

- We can generate code for a stack machine with accumulator
- We want to run the code on MIPS processor (or simulator)

- Simulate stack machine instructions using MIPS instructions

Simulate stack machine in MIPS

- The accumulator is kept in MIPS register $a0
- The stack is kept in memory
- The stack grows towards lower addresses (standard for MIPS)
- The address of the next location on the stack is kept in MIPS register $sp
  - the top of the stack is at address $sp + 4

Recap: MIPS Instructions

- **load** 32-bit **word** from address **reg2+offset** into **reg1**
  - **lw reg1 offset**(**reg2**)
  - lw $a0 4($fp)

- **store** 32-bit **word** from **reg1** at address **reg2+offset**
  - **sw reg1 offset(reg2)**
  - sw $a0 16($sp)

Stack operations

push	`sw $a0 0($sp)` `addiu $sp $sp -4`
pop	`addiu $sp $sp 4`
top	`lw $t1 4($sp)`

Code generation

- For each expression e, **cgen(e)** generates MIPS code that
  - computes the value of e in $a0
  - preserves $sp and the contents of the stack

- Can be implemented as AST traversal

Code generation for **constants**

- The code to evaluate a constant simply copies it into the accumulator:
  - **cgen(i)** = `li $a0 i`

- Preserves the stack, as required

Code generation for **add**

cgen(e1 + e2) =

```
cgen(e1)
push
cgen(e2)
$t1 := top
add $a0 $t1 $a0
pop
```

Code generation for **add** example

cgen(7 + 5) =

cgen(7)

push

cgen(5)

$t1 := top

add $a0 $t1 $a0

pop

Code generation for **add** example

cgen(7 + 5) =

li $a0 7

push

li $a0 5

$t1 := top

add $a0 $t1 $a0

pop

Code generation for **add** example

cgen(7 + 5) =

li \$a0 7

push

li \$a0 5

\$t1 := **top**

add \$a0 \$t1 \$a0

pop

Code generation for **add** example

cgen(7 + 5) =

```
li $a0 7
sw $a0 0($sp)
addiu $sp $sp -4
li $a0 5
lw $t1 0($sp)
add $a0 $t1 $a0
addiu $sp $sp 4
```

Code generation for **add** example

cgen(7 + 5) =

```
li $a0 7
sw $a0 0($sp)
addiu $sp $sp -4
li $a0 5
lw $t1 0($sp)
add $a0 $t1 $a0
addiu $sp $sp 4
```

```
li    $a0 7
move  $t1 $a0
li    $a0 5
add   $a0 $t1 $a0
```

Can we optimize these load and store operations?

Code generation for **add**

cgen(e1 + e2) =

cgen(e1)

push

cgen(e2)

$t1 := top

add $a0 $t1 $a0

pop

Can we put the result of e1 directly in $t1?

cgen(e1 + e2) =

cgen(e1)

move $t1 $a0

cgen(e2)

add $a0 $t1 $a0

INCORRECT!

Try to generate code for : 3 + (7 + 5)

Code generation for add

- cgen(e1 + e2) prints out MIPS assembly code for e1 + e2

- Template with “holes” for code for evaluating e1 and e2
- Recursively calls cgen on e1 and e2 to fill the holes
- Glues together the code code for e1 and e2

Code generation for **sub**

cgen(e1 - e2) =

```
cgen(e1)
push $a0
cgen(e2)
$t1 := top
sub $a0 $t1 $a0
pop
```

Code generation for **if**

cgen(if e1 = e2 then e3 else e4) =

```
        cgen(e1)
        push $a0
        cgen(e2)
        $t1 := top
        pop
        beq $a0 $t1 true_branch

                false_branch:
                        cgen(e4)
                        b end_if
                true_branch:
                        cgen(e3)
                end_if:
```

Use a counter to generate unique labels

When do we determine the address of true_branch? How?

Backpatching

- Determine the address of true_branch
- Only possible after we know the length of the code for e4
- Can we do it in a single pass?

- For every label, maintain a list of instructions that jump to this label
- When the address of the label is known, go over the list and update the corresponding jump instructions

Short circuit evaluation

- Second argument of a boolean operator is only evaluated if the first argument does not already determine the outcome

- (x and y) is equivalent to
  if x then y else false;
- (x or y) is equivalent to
  if x then true else y

Example: short circuit evaluation

a < b or (c < d and e < f)

naive

```
100: if a < b goto 103
101: T1 := 0
102: goto 104
103: T1 := 1
104: if c < d goto 107
105: T2 := 0
106: goto 108
107: T2 := 1
108: if e < f goto 111
109: T3 := 0
110: goto 112
111: T3 := 1
112: T4 := T2 and T3
113: T5 := T1 or T4
```

short circuit evaluation

```
100: if a < b goto 105
101: if !(c < d) goto 103
102: if e < f goto 105
103: T := 0
104: goto 106
105: T := 1
106:
```

Short circuit evaluation: examples

```
int denom = 0;
if (denom && nom/denom) {
        oops_i_just_divided_by_zero();
}
```

```
int x=0;
if (++x>0 && x++) {
        hmmm();
}
```

Code generation for **while**

cgen(while e1 loop e2 pool) =

start: **cgen(e1)**

beqz $a0 end

cgen(e2)

j start

end: li $a0 $0

In the real world….

- Production compilers do different things
- Emphasis is on keeping values in registers
- Intermediate results are laid out on the stack, not pushed and popped from the stack

Outline: code generation

✓ Expressions
  ✓ simple stack machine
  ✓ stack machine with accumulator
  - weighted register allocation
- Objects
- Method calls
- Runtime system

OBJECT LAYOUT

Representing data at runtime

- Source language types
  - int, boolean, string, pointer types, object types
- Target language types
  - single bytes, integers, address representation
- Compiler should map source types to some combination of target types
  - implement source types using target types

Basic types

- Examples: int, boolean, char
- Arithmetic operations: addition, subtraction, multiplication, division, remainder
- Can be mapped directly to target language types and operations

Pointer types

- Represent addresses of source language data structures
- Usually implemented as an unsigned integer
- Pointer dereferencing: retrieves pointed value
- May produce an error
  - null pointer dereference
  - when is this error triggered?
  - how is this error produced?

Object types

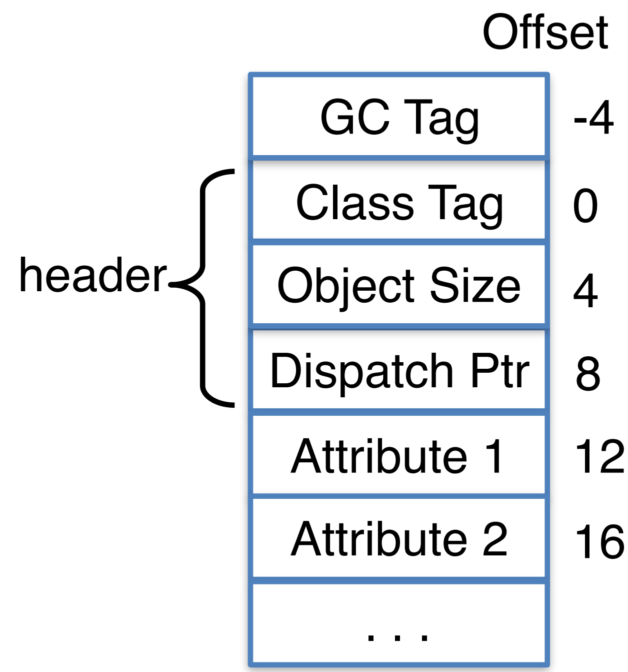
- Basic operations
  - Field selection: computing address of field, dereferencing address
  - Copying: copy block or field-by-field copying
  - Method invocation: identifying method to be called, calling it
- How does it look at runtime?

Object layout

- New objects are allocated on the heap
- Not necessarily adjacent to other objects
- Each object is a contiguous block of memory
- Each attribute stored at a fixed offset in object
  - x.f is a reference into object x at an offset corresponding to field f
- When a method is invoked, the object is self and the fields are the object's attributes

Cool object layout

- Object header (3 words)
  - Class tag is an integer: identifies class of the object
  - Object size is an integer: size of the object in words
  - Dispatch ptr: a pointer to a table of methods
- Attributes in subsequent slots



Cool object layout with inheritance

- Suppose that B is a subclass of A
- An object of dynamic type B can be used wherever an object of class A is expected
- Any method for A can be used on an object of dynamic type B
- Code in class A works unmodified for an object of dynamic type B
- The offset for an attribute is the same in a class and all of its subclasses
- Layout for subclass B defined by extending layout of A with additional slots for the additional attributes of B

x : A ← new A	
x.a	`lw $t1 12($t0)`
x ← new B	
x.a	`lw $t1 12($t0)`

Example: Cool object layout

```
Class A {
   a: Int ← 0;
   d: Int ← 1;
   f(): Int { a <- a + d };
};
```

All methods in all classes refer to a

Attributes a and d are inherited by classes B and C

```
Class  B inherits A {
   b: Int ← 2;
   f(): Int { a }; -- Override
   g(): Int { a ← a - b };
};
```

```
Class  C inherits A {
   c: Int <- 3;
   h(): Int { a ← a * c };
};
```

For A methods to work correctly in A, B, and C objects, attribute a must be in the same "place" in each object

Example: Cool object layout

Offset	Layout	new A()	new B()	new C()
0	Class Tag	tagA	tagB	tagC
4	Object Size	5	6	6
8	Dispatch Ptr	*	*	*
12	Attribute 1	a	a	a
16	Attribute 2	d	d	d
20	. . .		b	c

Prototype objects

- Allocation can be done only by Object.copy
- Prepare an object of every class with correct
  - garbage collection tag
  - class tag
  - object size
  - dispatch pointer
- Who sets attributes to defaults ?
  - basic classes – prototype objects
  - user-defined classes – prototype object or class_init
- Initialization <class>_init

Dispatch tables

- Every class has a fixed set of methods
  - including inherited methods
- A dispatch table indexes these methods
  - an array of method entry points
  - method **f** lives at a **fixed offset** in the dispatch table for a class and **all of its subclasses**
- Every method must know what object is “self”
  - “self” is passed as the first argument to all methods

Example: dispatch tables

- The dispatch table for class A has only 1 method
- Tables for B and C extend the table for A with more methods
- Because methods can be overridden, the method for f is not the same in every class, but is always at the same offset

Offset	new A()	new B()	new C()
0	fA	fB	fA
4		g	h

Class A { a: Int; d: Int; **f**(): Int {…}; };
Class B inherits A { b: Int; **f**(): Int {…}; **g**(): Int {…}; };
Class C inherits A { c: Int; **h**(): Int {…}; };

Example: dynamic dispatch

e.g()

- g refers to method in B if e is a B

e.f()

- f refers to method in A if e is an A or C
- f refers to method in B for a B object

Offset	new A()	new B()	new C()
0	fA	fB	fA
4		g	h

Class A { a: Int; d: Int; **f**(): Int {…}; };
Class B inherits A { b: Int; **f**(): Int {…}; **g**(): Int {…}; };
Class C inherits A { c: Int; **h**(): Int {…}; };

Using dispatch tables

- The implementation of methods and dynamic dispatch strongly resembles the implementation of attributes
- The dispatch pointer in an object of class X points to the dispatch table for class X
- Every method f of class X is assigned an offset in the dispatch table at compile time

Example: dispatch e.f(a1,a2)

- Arguments **a1** and **a2** are evaluated and results pushed on the stack
- Dispatch expression **e** is evaluated and the result put in **$a0**
- Dispatch expression **$a0** is tested for **void**
- If void, computation is aborted
  - **jal _dispatch_abort**
- Dispatch table address of the dispatch value is loaded
  - **lw $t2 8($a0)**
- Dispatch table is indexed with the method offset (12 in this example)
  - **lw $t2 12($t2)**
- Jump to the method (jal)
  - **jal $t2**
- In the callee, self is bound to e
  - **move $s0 $a0**

Cool prototype objects and dispatch tables

```
        .word   -1
A_protObj:
        .word   5
        .word   5
        .word   A_dispTab
        .word   int_const0
        .word   int_const0
        .word   -1
B_protObj:
        .word   6
        .word   6
        .word   B_dispTab
        .word   int_const0
        .word   int_const0
        .word   int_const0
        .word   -1
C_protObj:
        .word   7
        .word   6
        .word   C_dispTab
        .word   int_const0
        .word   int_const0
        .word   int_const0
```

```
A_dispTab:
        .word   Object.abort
        .word   Object.type_name
        .word   Object.copy
        .word   A.f
B_dispTab:
        .word   Object.abort
        .word   Object.type_name
        .word   Object.copy
        .word   B.f
        .word   B.g
C_dispTab:
        .word   Object.abort
        .word   Object.type_name
        .word   Object.copy
        .word   A.f
        .word   C.h
```

Class A { a: Int ←0; d: Int ← 1; **f**(): Int { a ← a + d}; };
Class B inherits A { b: Int ←2; **f**(): Int {a}; **g**(): Int {a ← a - c}; };
Class C inherits A { c: Int ←3; **h**(): Int {a ← a * c}; };

Designing the layout

- There is only one dispatch table per class. Why?
- The table is the same for all objects of a given class and does not change at run time
- Saves memory: all objects of a given class share a dispatch table
- Pays in execution time: extra memory access on every call

- Could we put the class tags and object size in the dispatch table?
- Yes! The class tag and object size are the same for all objects of a given class and do not change at run time
- The trade-offs are the same as for dispatch tables

MULTIPLE INHERITANCE

Multiple inheritance

- Independent
  - **no undirected cycles** in the inheritance graph (except root)
  - any pair of classes have at most one directed path between them (except root)
- Dependent
  - **no directed cycles** in the inheritance graph
  - a class can be superclass of another via more than one path

Example: independent multiple inheritance

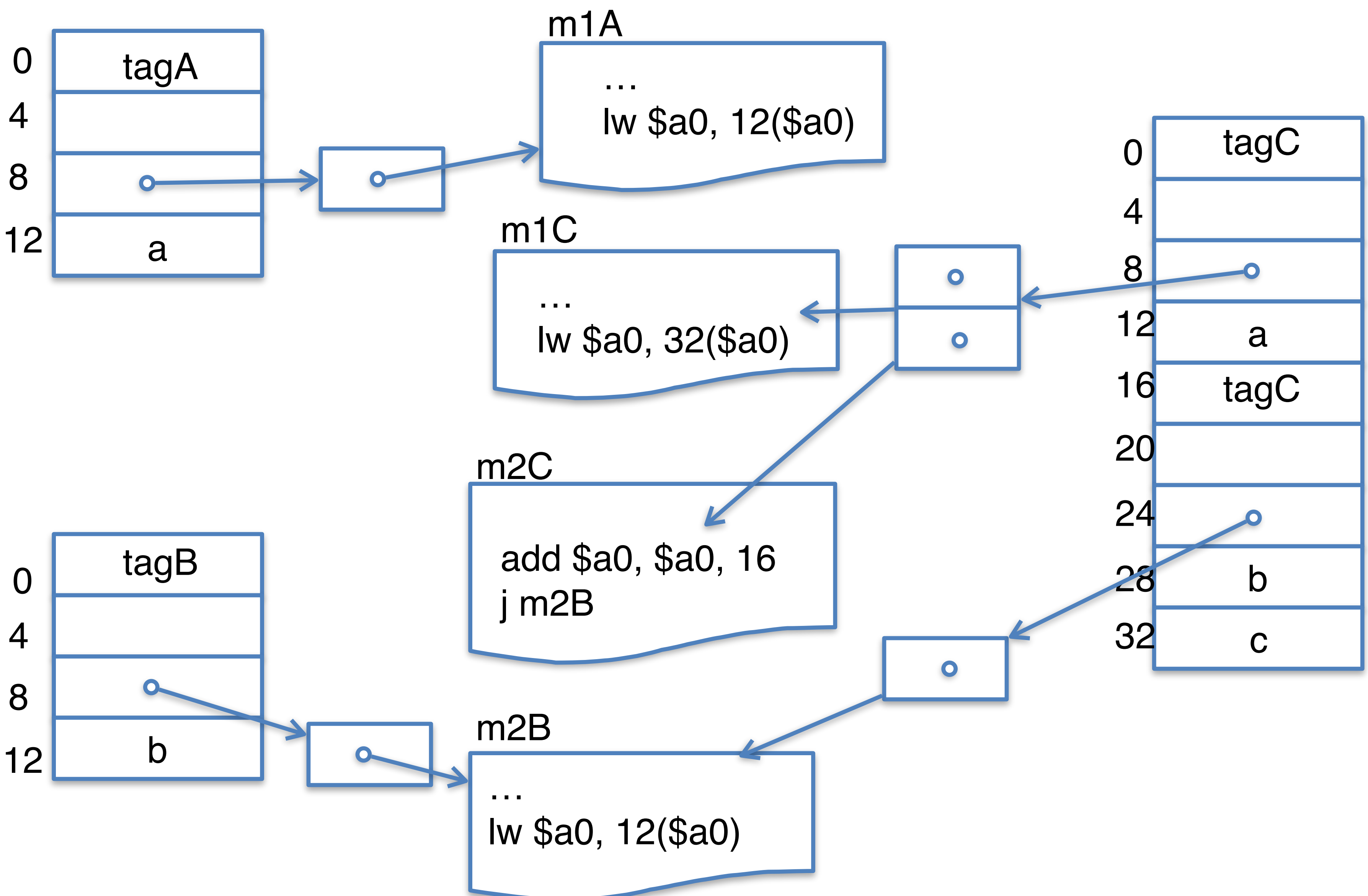
- How to extend Cool with multiple inheritance?
- Merge dispatch tables of superclasses
- Generate code for upcasts and downcasts

```
Class A { a: Int; m1() : Int { a }; };

Class B { b: Int; m2(): Int { b }; } ;

Class C inherit A, B { c: Int; m1(): Int { c }; };
```

Example: independent multiple inheritance



Example: independent multiple inheritance

- Fields and methods with the same name inherited from different superclasses
- How to handle it?
  - overriding
  - explicitly qualify every occurrence to disambiguate
  - order of declaration to disambiguate

Class A { **a**: Int; **m1**() : Int { a }; };

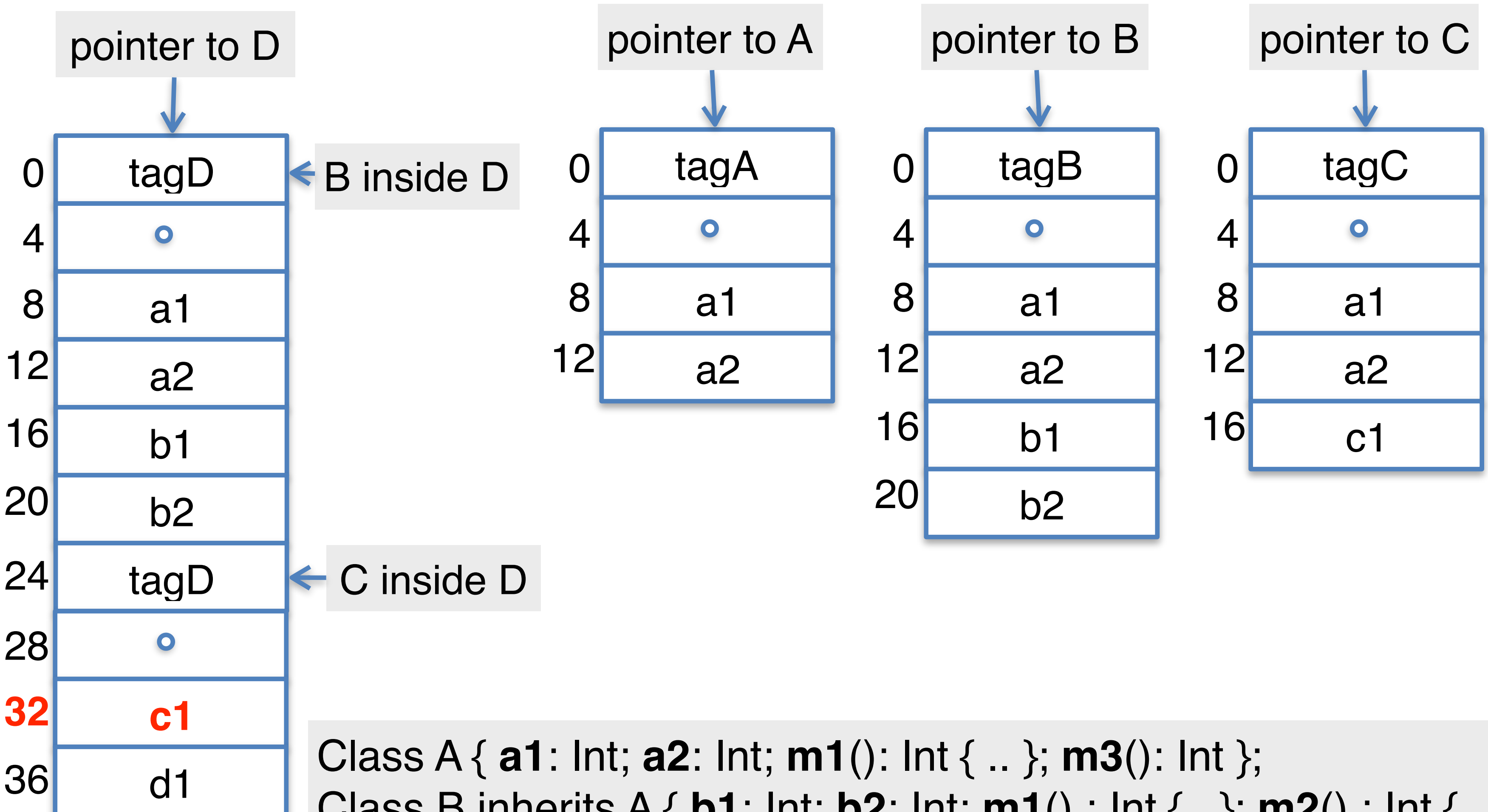
Class B { **a**: Int; **m1**(): Int { a }; } ;

Class C inherit A, B { **c**: Int; **m2**(): Int { a }; };

x : C ← new C()
x.m1();

Does x.m1 refer to m1 defined in A or B?

Example: dependent multiple inheritance



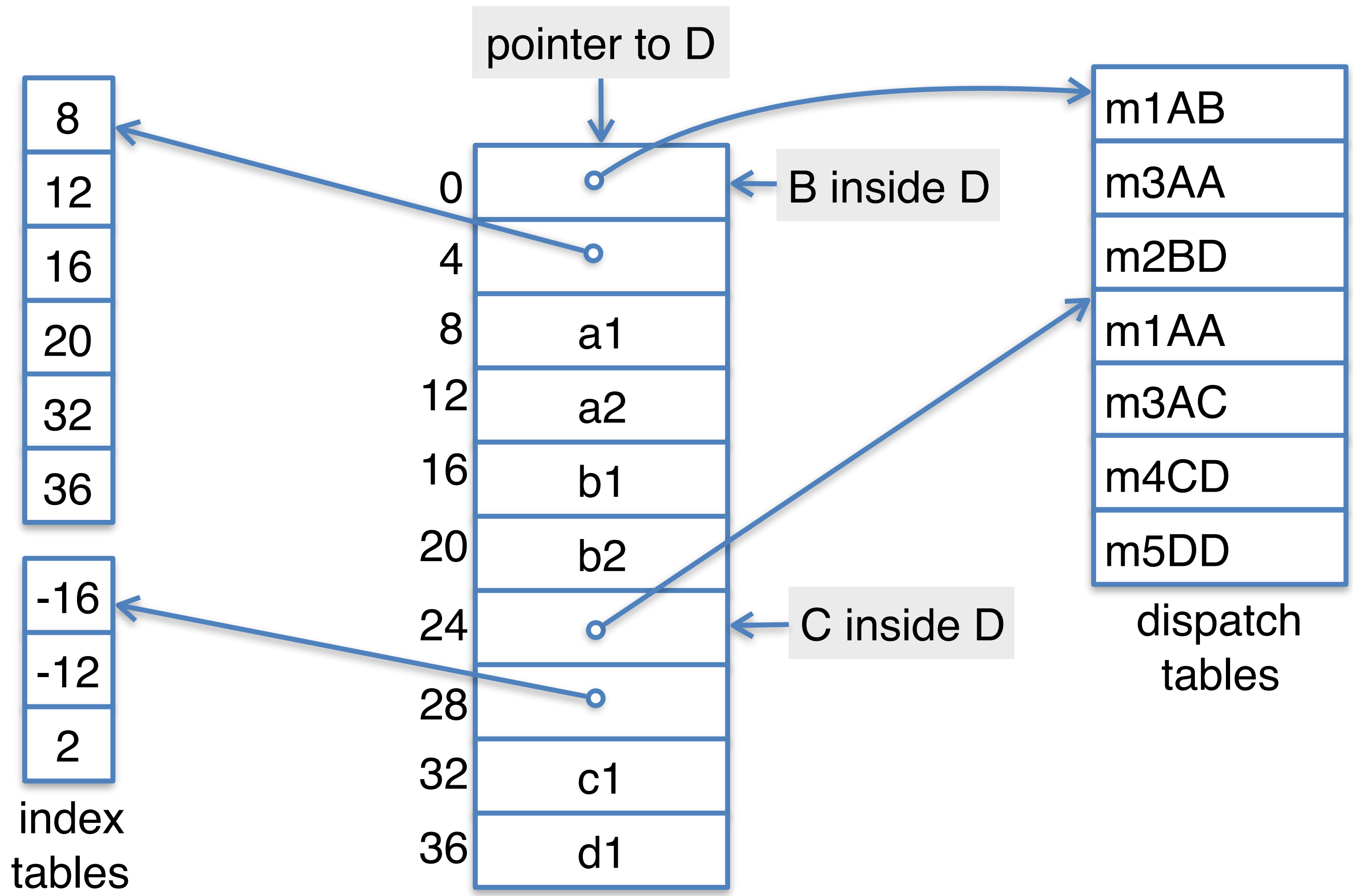
Class A { **a1**: Int; **a2**: Int; **m1**(): Int { .. }; **m3**(): Int };
Class B inherits A { **b1**: Int; **b2**: Int; **m1**() : Int { ..}; **m2**() : Int { ..};} ;
Class C inherits A { **c1**: Int; **m3**():Int {..}; **m4**():Int { ..}; } ;
Class D inherit B, C { **d1**: Int; **m2**():Int; **m4**():Int {..}; **m5**():Int {..}; };

Dependent multiple inheritance

- The simple solution does not work
- The positions of nested fields do not agree
- How many copies of common superclass?

- Use an **index table**
- Access offsets indirectly
- Some compilers avoid index table: use register allocation techniques to globally assign offsets

Example: dependent multiple inheritance



Class A { **a1**: Int; **a2**: Int; **m1**(): Int { .. }; **m3**(): Int };
Class B inherits A { **b1**: Int; **b2**: Int; **m1**() : Int { ..}; **m2**() : Int { ..};} ;
Class C inherits A { **c1**: Int; **m3**():Int {..}; **m4**():Int { ..}; } ;
Class D inherit B, C { **d1**: Int; **m2**():Int; **m4**():Int {..}; **m5**():Int {..}; };