Code Generation

Martin Kellogg

Course Announcements

- We recently fixed a bug in the reference compiler's x86-64 module. Only use Cool **version 1.39** for compiling to x86.
  - There will be an extra credit question on the midterm asking why I had to do this, if you want to do a comparison.
  - I will also award extra credit if you can find another bug in the reference compiler

Course Announcements

- We recently fixed a bug in the reference compiler's x86-64 module. Only use Cool **version 1.39** for compiling to x86.
  - There will be an extra credit question on the midterm asking why I had to do this, if you want to do a comparison.
  - I will also award extra credit if you can find another bug in the reference compiler
- Don't forget there is a **midterm** in this class on Wednesday!
  - Review session: tonight at 5pm (virtually)
  - Extra office half-hours tomorrow at 10am, 4:30pm

Course Announcements

- We recently fixed a bug in the reference compiler's x86-64 module. Only use Cool **version 1.39** for compiling to x86.
  - There will be an extra credit question on the midterm asking why I had to do this, if you want to do a comparison.
  - I will also award extra credit if you can find another bug in the reference compiler
- Don't forget there is a **midterm** in this class on Wednesday!
  - Review session: tonight at 5pm (virtually)
  - Extra office half-hours tomorrow at 10am, 4:30pm
- Hopefully you started PA3c3 over break
  - its due date is **one week from today**

Agenda

- Last time, all the way before the break:
  - Stack machine basics
    - accumulator, stack pointer
  - Stack discipline, calling convention for our stack machine
    - with a bit of optimization thrown in to give you a taste of the idea
- Today:
  - Quick review
  - Finish basics of stack machine codegen (i.e., variables, temps)
  - Object layout and its interactions with subtyping
  - Dispatch tables/vtables

Agenda

- Last time, all the way before the break:
  - Stack machine basics
    - accumulator, stack pointer
  - Stack discipline, calling convention for our stack machine
    - with a bit of optimization thrown in to give you a taste of the idea
- Today:
  - **Quick review**
  - Finish basics of stack machine codegen (i.e., variables, temps)
  - Object layout and its interactions with subtyping
  - Dispatch tables/vtables

Review: Stack Machine Basics

Review: Stack Machine Basics

- A ***stack machine*** maintains a stack of values for intermediate results

Review: Stack Machine Basics

- A ***stack machine*** maintains a stack of values for intermediate results
  - all operations read their inputs off of the stack

Review: Stack Machine Basics

- A ***stack machine*** maintains a stack of values for intermediate results
  - all operations read their inputs off of the stack
  - to reduce memory usage, keep the top of the stack in a special ***accumulator register***

Review: Stack Machine Basics

- A ***stack machine*** maintains a stack of values for intermediate results
  - all operations read their inputs off of the stack
  - to reduce memory usage, keep the top of the stack in a special ***accumulator register***
- It is critical that the **stack is preserved** across the evaluation of subexpressions

Review: Stack Machine Basics

- A ***stack machine*** maintains a stack of values for intermediate results
  - all operations read their inputs off of the stack
  - to reduce memory usage, keep the top of the stack in a special ***accumulator register***
- It is critical that the **stack is preserved** across the evaluation of subexpressions
  - this lets us write a **recursive descent** code generator

Review: Stack Machine Basics

- A ***stack machine*** maintains a stack of values for intermediate results
  - all operations read their inputs off of the stack
  - to reduce memory usage, keep the top of the stack in a special ***accumulator register***
- It is critical that the **stack is preserved** across the evaluation of subexpressions
  - this lets us write a **recursive descent** code generator
- Last time, we saw how to generate Cool-ASM using a stack machine for arithmetic expressions (add/subtract), if, and function calls

Review: Stack Machine Basics

- A ***stack machine*** maintains a stack of values for intermediate results
  - all operations read their inpu
  - to reduce memory usage, kee
    special ***accumulator register***
- It is critical that the **stack is prese** **f** subexpressions
  - this lets us write a **recursive descent** code generator
- Last time, we saw how to generate Cool-ASM using a stack machine for arithmetic expressions (**add**/subtract), if, and function calls

```
cgen(e1 + e2) =
    cgen(e1)
    push r1
    cgen(e2) ;; e2 now in r1
    pop t1
    add r1 <- t1 r1
```

Code Generation: Variables

- **Variable references** are the last construct

Code Generation: Variables

- **Variable references** are the last construct
- The “variables” of a function are just its **parameters**

Code Generation: Variables

- **Variable references** are the last construct
- The “variables” of a function are just its **parameters**
  - They are all in the AR

Code Generation: Variables

- **Variable references** are the last construct
- The “variables” of a function are just its **parameters**
  - They are all in the AR
  - Pushed by the caller

Code Generation: Variables

- **Variable references** are the last construct
- The “variables” of a function are just its **parameters**
  - They are all in the AR
  - Pushed by the caller
- Problem: because the stack grows when intermediate results are saved, the variables are **not at a fixed offset** from sp

Code Generation: Variables

- **Variable references** are the last construct
- The “variables” of a function are just its **parameters**
  - They are all in the AR
  - Pushed by the caller
- Problem: because the stack grows when intermediate results are saved, the variables are **not at a fixed offset** from sp
  - Challenge question: what are they at a fixed offset from?

Code Generation: Variables

- **Variable references** are the last construct
- The “variables” of a function are just its **parameters**
  - They are all in the AR
  - Pushed by the caller
- Problem: because the stack grows when intermediate results are saved, the variables are **not at a fixed offset** from sp
  - Challenge question: what are they at a fixed offset from?
  - Answer: the **frame pointer**

Code Generation: Variables

- **Variable references** are the last construct
- The “variables” of a function are just its **parameters**
  - They are all in the AR
  - Pushed by the caller
- Problem: because the stack grows when intermediate results are saved, the variables are **not at a fixed offset** from sp
  - Challenge question: what are they at a fixed offset from?
  - Answer: the **frame pointer**
    - Always points to the return address on the stack
      - = the value of sp on function entry

Code Generation: Variables

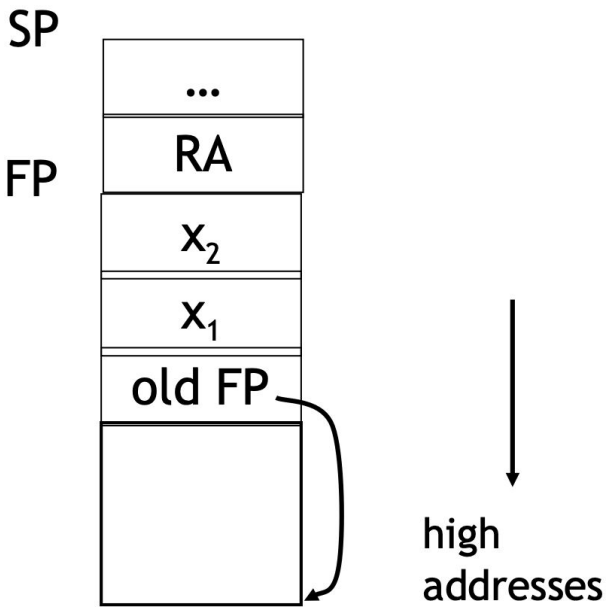
- **Variable references** are the last construct
- The “variables” of a function are just its **parameters**
  - They are all in the AR
  - Pushed by the caller
- Problem: because the stack grows when intermediate results are saved, the variables are **not at a fixed offset** from sp
  - Challenge question: what are they at a fixed offset from?
  - Answer: the **frame pointer**
    - Always points to the return address on the stack
      - = the value of sp on function entry
    - It doesn’t move => args on the stack are at a fixed offset

Code Generation: Variables

- Example: For a function def $f(x_1, x_2) = e$ the activation and frame pointer are set up as follows:

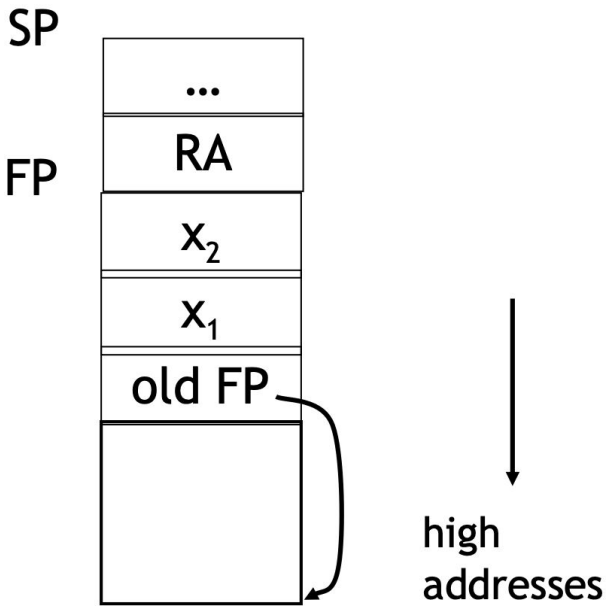
Code Generation: Variables

- Example: For a function def $f(x_1, x_2) = e$ the activation and frame pointer are set up as follows:



Code Generation: Variables

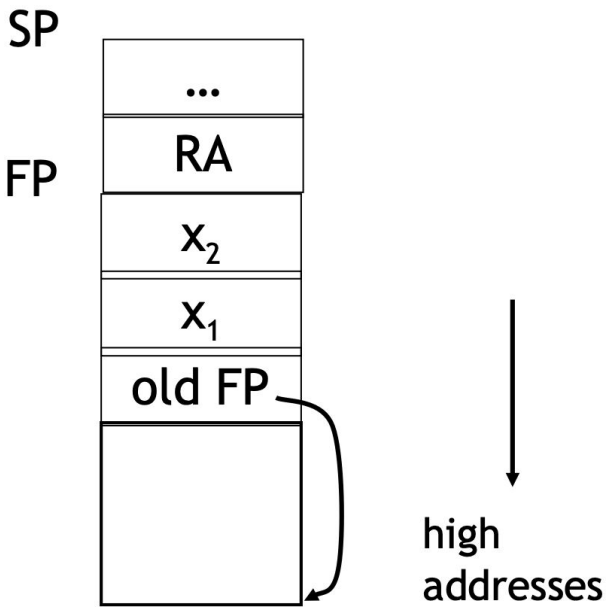
- Example: For a function def f(x_1, x_2) = e the activation and frame pointer are set up as follows:



- x_1 (first parameter) is at **fp + 2**

Code Generation: Variables

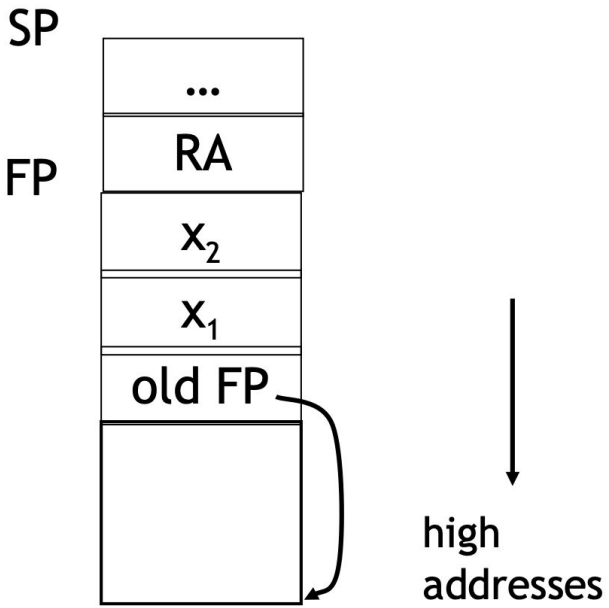
- Example: For a function def f(x_1, x_2) = e the activation and frame pointer are set up as follows:



- x_1 (first parameter) is at **fp + 2**
- x_2 (second parameter) is at **fp + 1**

Code Generation: Variables

- Example: For a function def f(x_1, x_2) = e the activation and frame pointer are set up as follows:

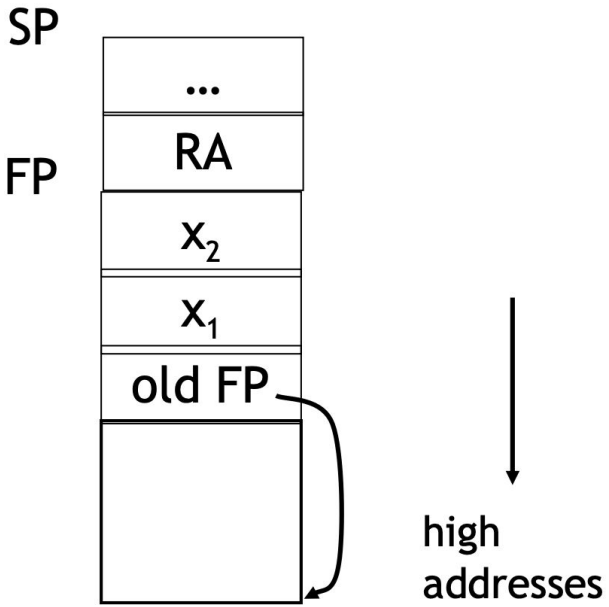


- x_1 (first parameter) is at **fp + 2**
- x_2 (second parameter) is at **fp + 1**
- Thus:

cgen(x_i) =
ld r1 <- fp[z]

Code Generation: Variables

- Example: For a function def f(x_1, x_2) = e the activation and frame pointer are set up as follows:



- x_1 (first parameter) is at **fp + 2**
- x_2 (second parameter) is at **fp + 1**
- Thus:

cgen(x_i) =
ld r1 <- fp[z]

- where $z \approx n+1 - i$

Summary

Summary

- The activation record must be **designed together** with the code generator

Summary

- The activation record must be **designed together** with the code generator
- Code generation can be done by **recursive traversal** of the AST

Summary

- The activation record must be **designed together** with the code generator
- Code generation can be done by **recursive traversal** of the AST
- As you write your compiler, we recommend starting with a **stack machine** (simpler!)

Summary

- The activation record must be **designed together** with the code generator
- Code generation can be done by **recursive traversal** of the AST
- As you write your compiler, we recommend starting with a **stack machine** (simpler!)
  - `./cool --asm` generates Cool-ASM stack machine code for Cool
    - use this to help you with PA3!

Summary

- The activation record must be **designed together** with the code generator
- Code generation can be done by **recursive traversal** of the AST
- As you write your compiler, we recommend starting with a **stack machine** (simpler!)
  - `./cool --asm` generates Cool-ASM stack machine code for Cool
    - use this to help you with PA3!
- Production compilers do different things:

Summary

- The activation record must be **designed together** with the code generator
- Code generation can be done by **recursive traversal** of the AST
- As you write your compiler, we recommend starting with a **stack machine** (simpler!)
  - `./cool --asm` generates Cool-ASM stack machine code for Cool
    - use this to help you with PA3!
- Production compilers do different things:
  - keep as many values as possible in registers, etc

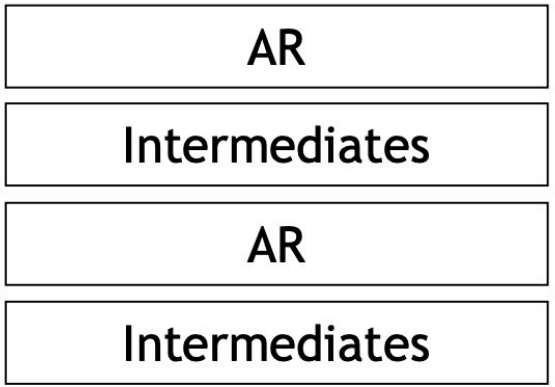
Summary

- The activation record must be **designed together** with the code generator
- Code generation can be done by **recursive traversal** of the AST
- As you write your compiler, we recommend starting with a **stack machine** (simpler!)
  - `./cool --asm` generates Cool-ASM stack machine code for Cool
    - use this to help you with PA3!
- Production compilers do different things:
  - keep as many values as possible in registers, etc
  - save this stuff for PA4

Temporaries

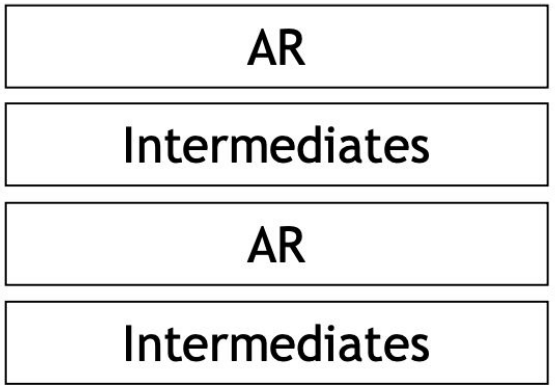
Temporaries

- The stack machine code layout we've described so far has activation records and intermediate results **interleaved** on the stack.



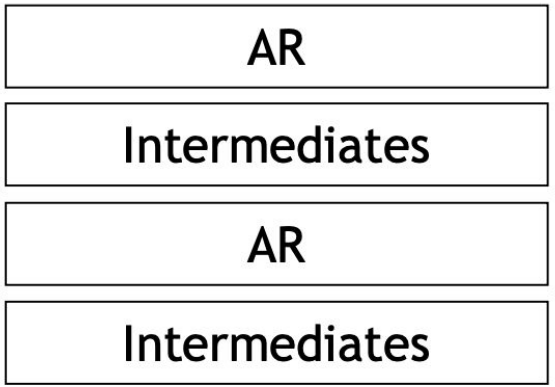
Temporaries

- The stack machine code layout we've described so far has activation records and intermediate results **interleaved** on the stack.
  - Advantage: **Very simple** code generation (great for PA3)



Temporaries

- The stack machine code layout we've described so far has activation records and intermediate results **interleaved** on the stack.
  - Advantage: **Very simple** code generation (great for PA3)
  - Disadvantage: **Very slow** code (bad for PA4)
    - Storing and loading temporaries requires a store/load and sp adjustment



Temporaries: A Better Way

- **Idea**: Keep temporaries **in the AR**

Temporaries: A Better Way

- **Idea**: Keep temporaries **in the AR**
  - Creates work for us: the code generator must **assign space** in the AR for each temporary

Temporaries: A Better Way

- **Idea**: Keep temporaries **in the AR**
  - Creates work for us: the code generator must **assign space** in the AR for each temporary
  - Therefore, we need to **know** how many temporaries there are!

Temporaries: A Better Way

- **Idea**: Keep temporaries **in the AR**
  - Creates work for us: the code generator must **assign space** in the AR for each temporary
  - Therefore, we need to **know** how many temporaries there are!
- In other words, our compiler must determine:

Temporaries: A Better Way

- **Idea**: Keep temporaries **in the AR**
  - Creates work for us: the code generator must **assign space** in the AR for each temporary
  - Therefore, we need to **know** how many temporaries there are!
- In other words, our compiler must determine:
  - **What** intermediate values are placed on the stack?

Temporaries: A Better Way

- **Idea**: Keep temporaries **in the AR**
  - Creates work for us: the code generator must **assign space** in the AR for each temporary
  - Therefore, we need to **know** how many temporaries there are!
- In other words, our compiler must determine:
  - **What** intermediate values are placed on the stack?
  - **How many** slots are needed in the AR to hold these values?

Temporaries: How Many?

- Let ***NT***(e) = number of temps needed to evaluate e

Temporaries: How Many?

- Let ***NT***(e) = number of temps needed to evaluate e
- Example: ***NT***($e_1 + e_2$)

```
cgen(e1 + e2) =
    cgen(e1)
    push r1
    cgen(e2) ;; e2 now in r1
    pop t1
    add r1 <- t1 r1
```

Temporaries: How Many?

- Let ***NT***(e) = number of temps needed to evaluate e
- Example: ***NT***($e_1 + e_2$)
  - Needs at least as many temporaries as ***NT***(e_1)

```
cgen(e1 + e2) =
    cgen(e1)
    push r1
    cgen(e2) ;; e2 now in r1
    pop t1
    add r1 <- t1 r1
```

Temporaries: How Many?

- Let ***NT***(e) = number of temps needed to evaluate e
- Example: ***NT***($e_1 + e_2$)
  - Needs at least as many temporaries as ***NT***(e_1)
  - Needs at least as many temporaries as ***NT***(e_2) + 1

```
cgen(e1 + e2) =
    cgen(e1)
    push r1
    cgen(e2) ;; e2 now in r1
    pop t1
    add r1 <- t1 r1
```

Temporaries: How Many?

- Let ***NT***(e) = number of temps needed to evaluate e
- Example: ***NT***($e_1 + e_2$)
  - Needs at least as many temporaries as ***NT***(e_1)
  - Needs at least as many temporaries as ***NT***(e_2) + 1
- **Insight:** Space used for temporaries in e_1 can be **reused** for temporaries in e_2

```
cgen(e1 + e2) =
    cgen(e1)
    push r1
    cgen(e2) ;; e2 now in r1
    pop t1
    add r1 <- t1 r1
```

Temporaries: The NumTemps Equations

- $NT(e_1 + e_2) = \max(NT(e_1), 1 + NT(e_2))$

Temporaries: The NumTemps Equations

- $NT(e_1 + e_2) = \max(NT(e_1), 1 + NT(e_2))$
- $NT(e_1 - e_2) = \max(NT(e_1), 1 + NT(e_2))$

Temporaries: The NumTemps Equations

- $NT(e_1 + e_2) = \max(NT(e_1), 1 + NT(e_2))$
- $NT(e_1 - e_2) = \max(NT(e_1), 1 + NT(e_2))$
- $NT(\text{if } e_1 = e_2 \text{ then } e_3 \text{ else } e_4) =$
  $\max(NT(e_1), 1 + NT(e_2), NT(e_3), NT(e_4))$

Temporaries: The NumTemps Equations

- $NT(e_1 + e_2) = \max(NT(e_1), 1 + NT(e_2))$
- $NT(e_1 - e_2) = \max(NT(e_1), 1 + NT(e_2))$
- $NT(\text{if } e_1 = e_2 \text{ then } e_3 \text{ else } e_4) =$
  $\max(NT(e_1), 1 + NT(e_2), NT(e_3), NT(e_4))$
- $NT(id(e_1, \ldots, e_n)) = \max(NT(e_1), \ldots, NT(e_n))$

Temporaries: The NumTemps Equations

- $NT(e_1 + e_2) = \max(NT(e_1), 1 + NT(e_2))$
- $NT(e_1 - e_2) = \max(NT(e_1), 1 + NT(e_2))$
- $NT(\text{if } e_1 = e_2 \text{ then } e_3 \text{ else } e_4) =$
  $\max(NT(e_1), 1 + NT(e_2), NT(e_3), NT(e_4))$
- $NT(id(e_1, \ldots, e_n)) = \max(NT(e_1), \ldots, NT(e_n))$
- $NT(int) = 0$

Temporaries: The NumTemps Equations

- $NT(e_1 + e_2) = \max(NT(e_1), 1 + NT(e_2))$
- $NT(e_1 - e_2) = \max(NT(e_1), 1 + NT(e_2))$
- $NT(\text{if } e_1 = e_2 \text{ then } e_3 \text{ else } e_4) =$
  $\max(NT(e_1), 1 + NT(e_2), NT(e_3), NT(e_4))$
- $NT(id(e_1, \ldots, e_n)) = \max(NT(e_1), \ldots, NT(e_n))$
- $NT(int) = 0$
- $NT(id) = 0$

Temporaries: The NumTemps Equations

- ***NT***(e_1 + e_2) = max(***NT***(e_1), 1 + ***NT***(e_2))
- ***NT***(e_1 - e_2) = max(***NT***(e_1), 1 + ***NT***(e_2))
- ***NT***(if e_1 = e_2 then e_3 else e_4) =
  max(***NT***(e_1), 1 + ***NT***(e_2), ***NT***(e_3), ***NT***(e_4))
- ***NT***(*id*(e_1, …, e_n)) = max(***NT***(e_1), …, ***NT***(e_n))
- ***NT***(*int*) = 0
- ***NT***(*id*) = 0

In class exercise: what is ***NT***(def fib(x) =
  if x = 1 then 0 else
    if x = 2 then 1
    else fib(x - 1) + fib(x – 2))?

Revised AR

Revised AR

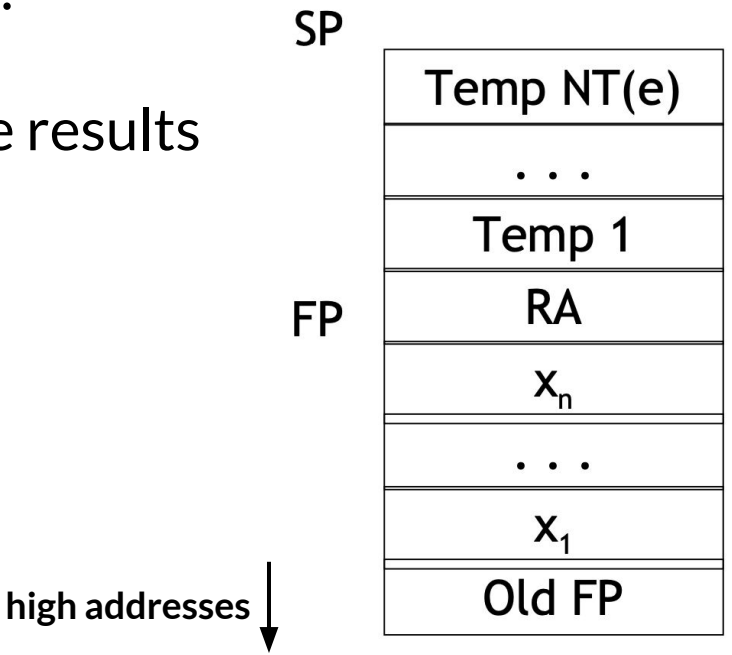
- For a function definition $f(x_1, ..., x_n) = e$ the AR has `n + NT(e) + 2` elements (so far):

Revised AR

- For a function definition $f(x_1, ..., x_n) = e$ the AR has `n + NT(e) + 2` elements (so far):
  - `n` arguments
  - ***NT***(e) locations for intermediate results
  - Return address
  - Frame pointer

Revised AR

- For a function definition $f(x_1, ..., x_n) = e$ the AR has `n + NT(e) + 2` elements (so far):
  - `n` arguments
  - ***NT***(e) locations for intermediate results
  - Return address
  - Frame pointer



Revised Code Generation

- Code generation must know how many temporaries are in use at each point

Revised Code Generation

- Code generation must know how many temporaries are in use at each point
- Add a new argument to code generation: the position of the next available temporary:

cgen(e, n) : generate code for e and use temporaries whose address is (fp - n) or lower

Revised Code Generation: +

Old:

New:

Revised Code Generation: +

Old:

New:

cgen(e_1 + e_2) =
    cgen(e_1)
    push r1
    cgen(e_2)
    pop t1
    add r1 <- t1 r1

Revised Code Generation: +

Old:

```
cgen(e1 + e2) =
    cgen(e1)
    push r1
    cgen(e2)
    pop t1
    add r1 <- t1 r1
```

New:

```
cgen(e1 + e2, nt) =
    cgen(e1, nt)
    st fp[-nt] <- r1
    cgen(e2, nt + 1)
    ld temp <- fp[-nt]
    add r1 <- t1 r1
```

Revised Code Generation: +

Old:

```
cgen(e1 + e2) =
    cgen(e1)
    push r1
    cgen(e2)
    pop t1
    add r1 <- t1 r1
```

New:

```
cgen(e1 + e2, nt) =
    cgen(e1, nt)
    st fp[-nt] <- r1
    cgen(e2, nt + 1)
    ld temp <- fp[-nt]
    add r1 <- t1 r1
```

Where are the savings?
(Hint: “push” is more expensive than it looks…)

Notes: Temporaries

Notes: Temporaries

- The temporary area is used like a **small, fixed-size stack**

Notes: Temporaries

- The temporary area is used like a **small, fixed-size stack**
- Exercise that might help if you are struggling with PA3c3: Write out **cgen** for other constructs

Notes: Temporaries

- The temporary area is used like a **small, fixed-size stack**
- Exercise that might help if you are struggling with PA3c3: Write out **cgen** for other constructs
- Hint: on function entry, you'll have to increment something by ***NT***(e)
  - ... and on function exit, decrement it ...

Trivia Break: ??

Code Generation for Object-Oriented Langs

- The remainder of today will be spent on two primary topics:

Code Generation for Object-Oriented Langs

- The remainder of today will be spent on two primary topics:
  - object layout in object-oriented languages (i.e., subclasses)

Code Generation for Object-Oriented Langs

- The remainder of today will be spent on two primary topics:
  - object layout in object-oriented languages (i.e., subclasses)
  - dynamic dispatch

Code Generation for Object-Oriented Langs

- The remainder of today will be spent on two primary topics:
  - object layout in object-oriented languages (i.e., subclasses)
  - dynamic dispatch
- These are both tricky because of the **Liskov substitution principle**: If B is a subclass of A, then an object of class B can be used wherever an object of class A is expected

Code Generation for Object-Oriented Langs

- The remainder of today will be spent on two primary topics:
  - object layout in object-oriented languages (i.e., subclasses)
  - dynamic dispatch
- These are both tricky because of the **Liskov substitution principle**: If B is a subclass of A, then an object of class B can be used wherever an object of class A is expected
  - This means that code in class A **must work unmodified** on an object of class B

Object Layout

Object Layout

- An object is like a **struct** in C

Object Layout

- An object is like a **struct** in C
  - The reference foo.field is an index into a foo struct at an offset corresponding to field

Object Layout

- An object is like a **struct** in C
  - The reference foo.field is an index into a foo struct at an offset corresponding to field
- Objects in Cool are implemented this way

Object Layout

- An object is like a **struct** in C
  - The reference foo.field is an index into a foo struct at an offset corresponding to field
- Objects in Cool are implemented this way
  - Objects are laid out in **contiguous memory**

Object Layout

- An object is like a **struct** in C
  - The reference foo.field is an index into a foo struct at an offset corresponding to field
- Objects in Cool are implemented this way
  - Objects are laid out in **contiguous memory**
  - Each attribute stored at a **fixed offset** in object

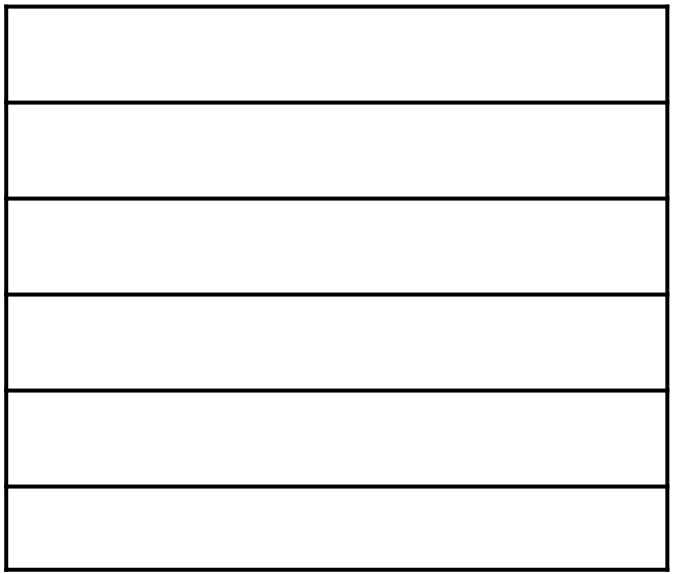
Object Layout

- An object is like a **struct** in C
  - The reference foo.field is an index into a foo struct at an offset corresponding to field
- Objects in Cool are implemented this way
  - Objects are laid out in **contiguous memory**
  - Each attribute stored at a **fixed offset** in object
  - When a method is invoked, the object becomes self and the fields are the object's attributes

Cool Object Layout

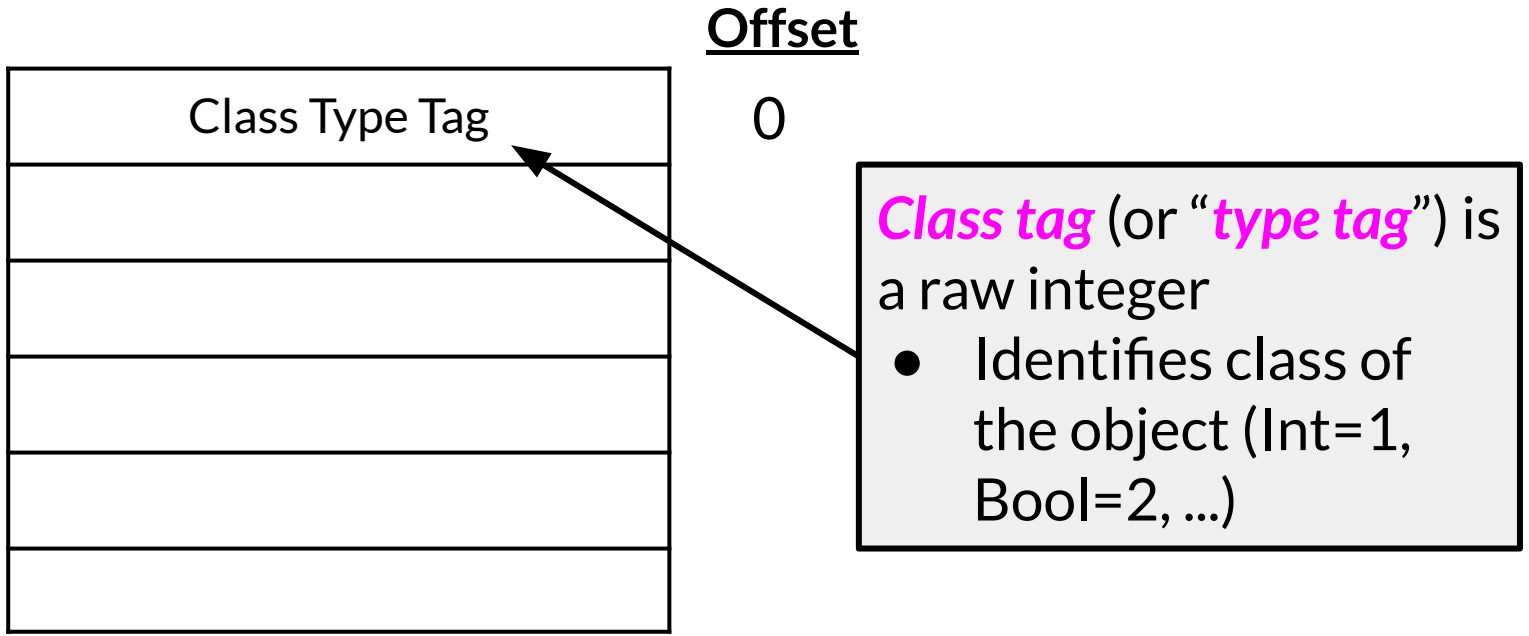
- The first three words of each Cool object contains ***header information***:

Offset



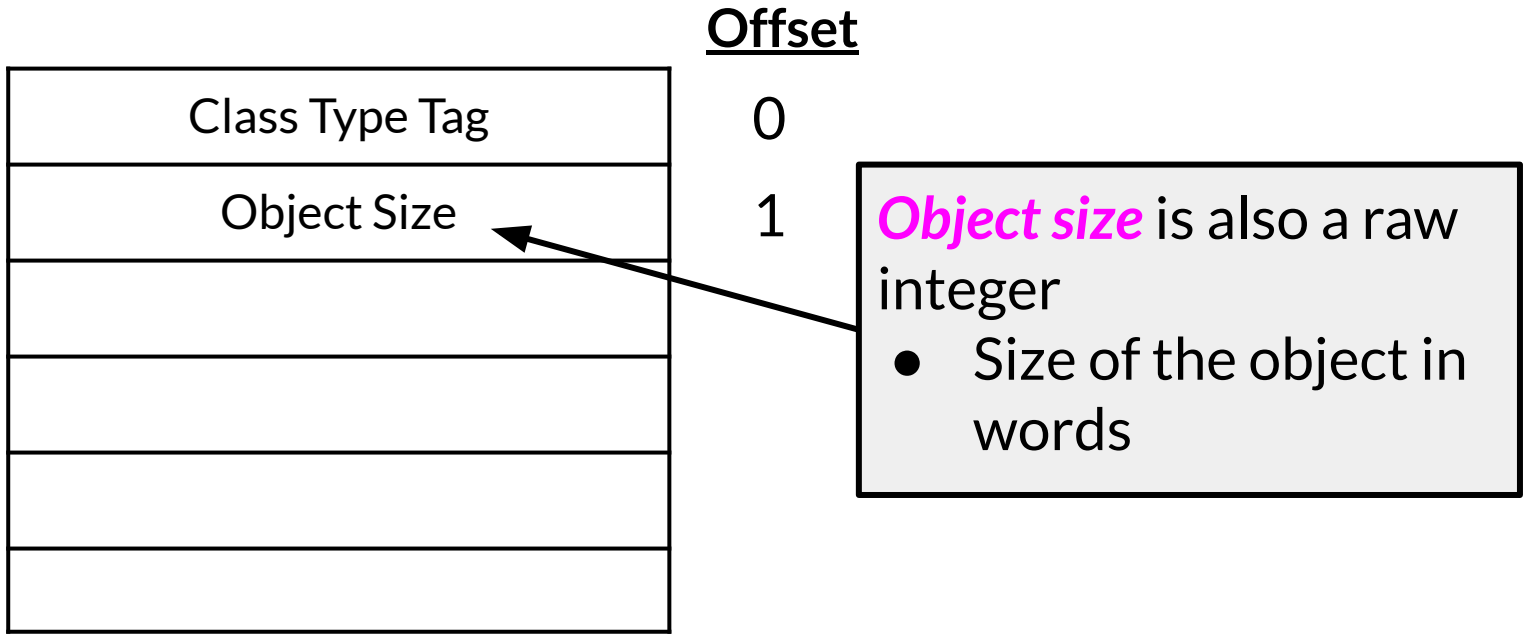
Cool Object Layout

- The first three words of each Cool object contains ***header information***:



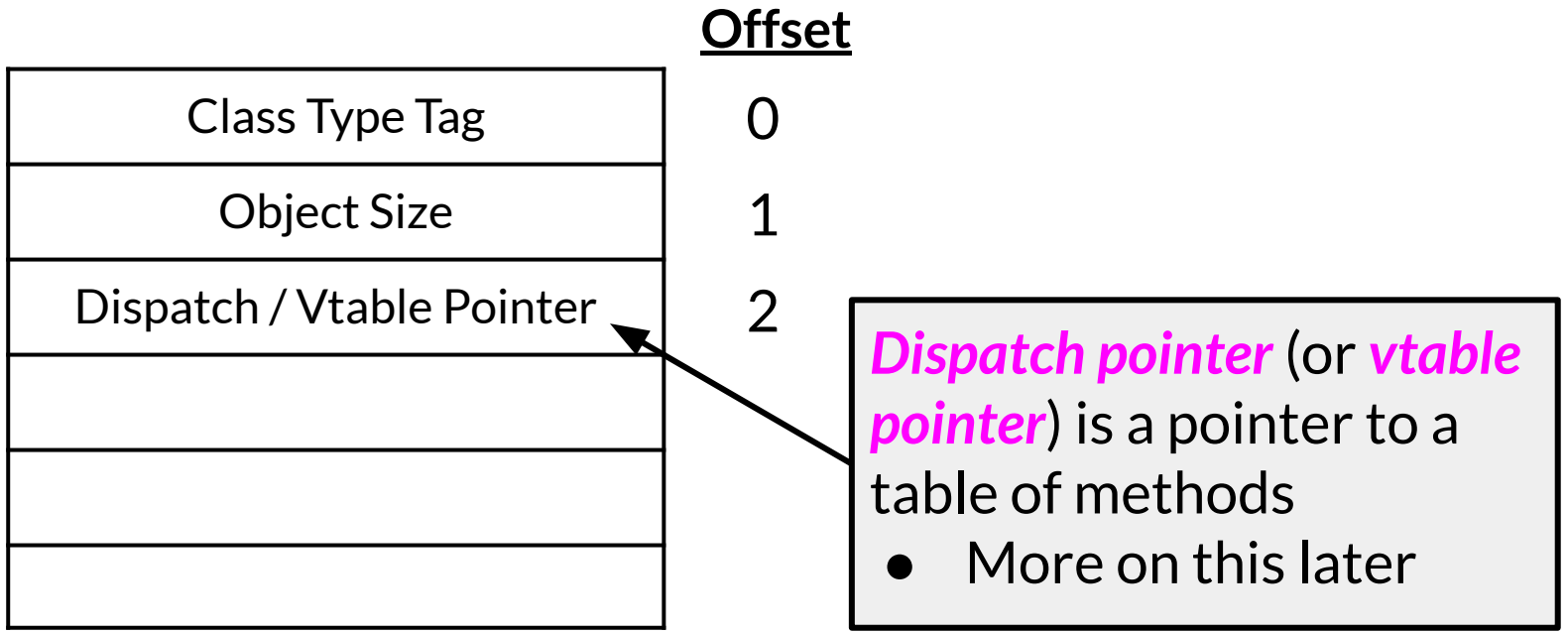
Cool Object Layout

- The first three words of each Cool object contains ***header information***:



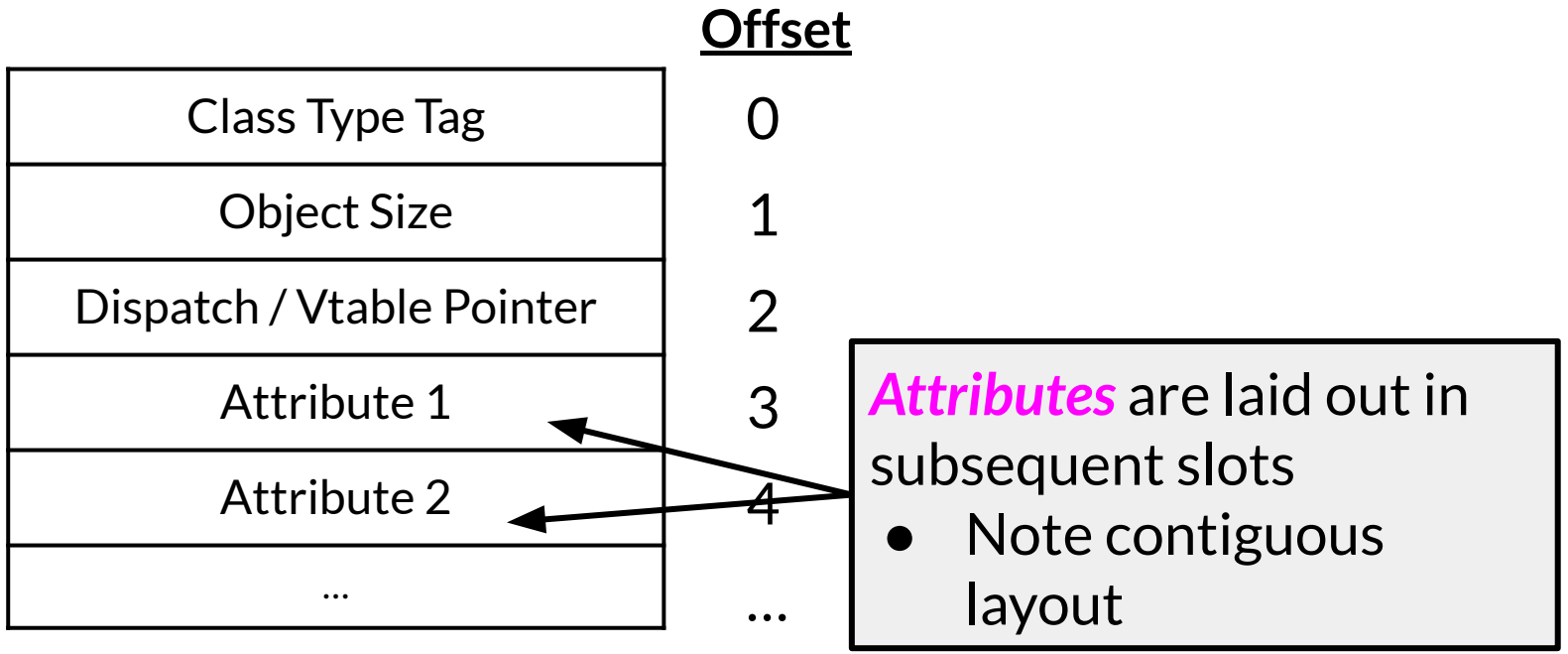
Cool Object Layout

- The first three words of each Cool object contains ***header information***:



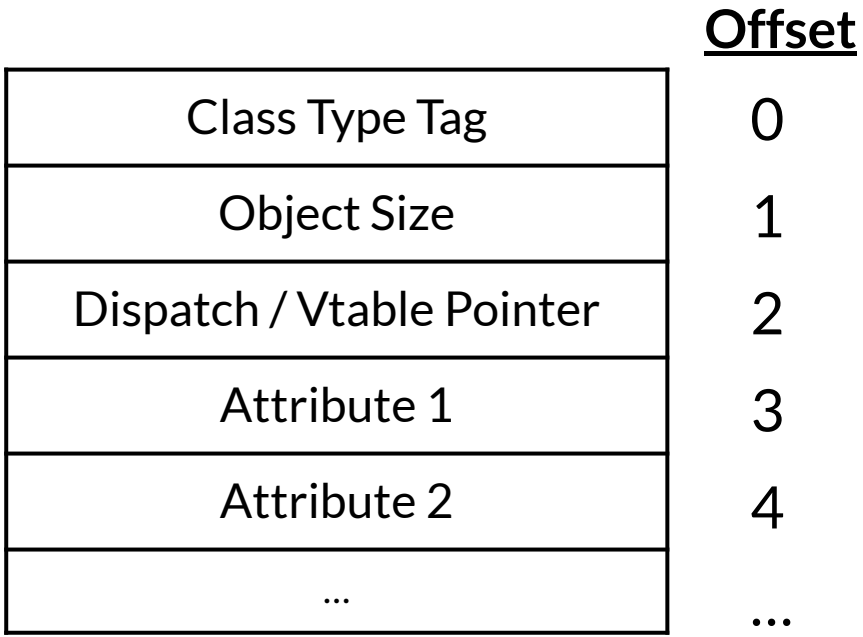
Cool Object Layout

- The first three words of each Cool object contains ***header information***:



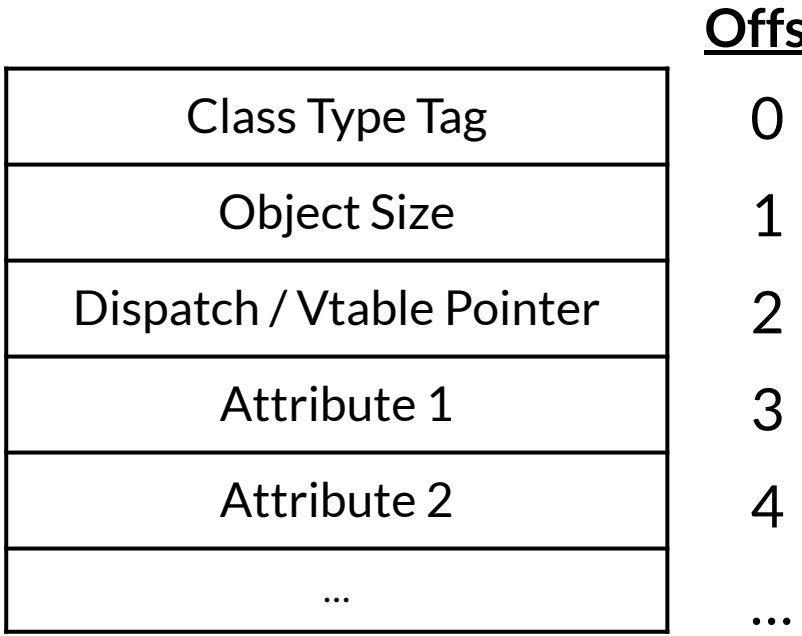
Cool Object Layout

- The first three words of each Cool object contains ***header information***:



Cool Object Layout

- The first three words of each Cool object contains ***header information***:



Note this is a convention that we made up, but it is similar to how Java and C++ lay things out. For example, you could swap #1 and #2 without loss.

Cool Object Layout: Example

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

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 };
};
```

Cool Object Layout: Example

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

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 };
};
```

Things to note:

Cool Object Layout: Example

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

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 };
};
```

Things to note:

- Attributes **a** and **d** are inherited by classes B and C

Cool Object Layout: Example

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

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 };
};
```

Things to note:

- Attributes **a** and **d** are inherited by classes B and C
- All methods in all classes refer to **a**

Cool Object Layout: Example

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

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 };
};
```

Things to note:

- Attributes **a** and **d** are inherited by classes B and C
- All methods in all classes refer to **a**
- For A methods to work correctly in A, B, and C objects, attribute **a** must be in the same “place” in each object

Object Layout: Key Point

Object Layout: Key Point

- **Key Observation**: Given a layout for class A, a layout for subclass B can be defined by ***extending*** the layout of A with additional slots for the additional attributes of B

Object Layout: Key Point

- **Key Observation**: Given a layout for class A, a layout for subclass B can be defined by ***extending*** the layout of A with additional slots for the additional attributes of B
  - (i.e., append new fields at the bottom)

Object Layout: Key Point

- **Key Observation**: Given a layout for class A, a layout for subclass B can be defined by ***extending*** the layout of A with additional slots for the additional attributes of B
  - (i.e., append new fields at the bottom)
  - leaves the layout of A **unchanged** (B is an extension)

Object Layout: Key Point

- **Key Observation**: Given a layout for class A, a layout for subclass B can be defined by ***extending*** the layout of A with additional slots for the additional attributes of B
  - (i.e., append new fields at the bottom)
  - leaves the layout of A **unchanged** (B is an extension)
  - this is where the “extends” keyword in Java etc comes from

Cool Object Layout: Example w/ Picture

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

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 };
};
```

Class / Offset	A	B	C
0 (tag)	Atag	Btag	Ctag
1 (size)	5	6	6
2 (vtable)	*	*	*
3 (attr#1)	a	a	a
4 ...	d	d	d
5		b	c

Object Layout: Subclass Invariant

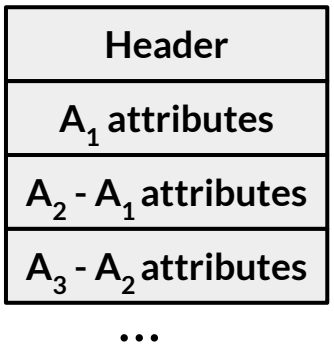
- The **offset for an attribute** is the **same** in a class and all of its subclasses

Object Layout: Subclass Invariant

- The **offset for an attribute** is the **same** in a class and all of its subclasses
  - This choice allows any method defined for an A_1 to be used on a subclass A_2
    - without any change to the implementation!

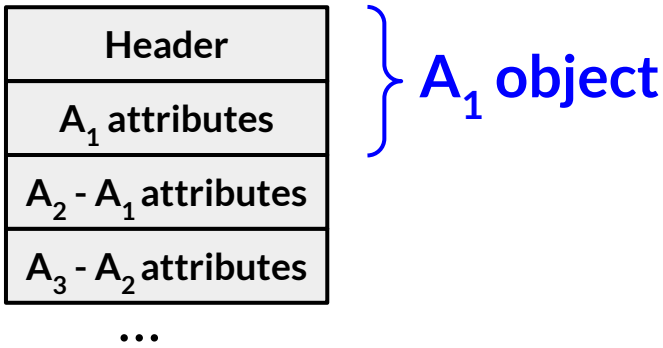
Object Layout: Subclass Invariant

- The **offset for an attribute** is the **same** in a class and all of its subclasses
  - This choice allows any method defined for an A_1 to be used on a subclass A_2
    - without any change to the implementation!
- Consider layout for $A_n \leq \ldots A_3 \leq A_2 \leq A_1$:



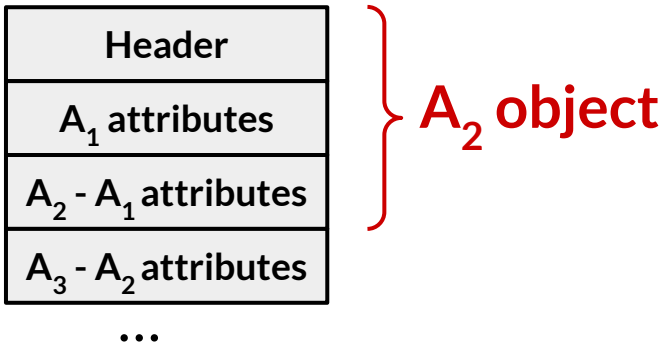
Object Layout: Subclass Invariant

- The **offset for an attribute** is the **same** in a class and all of its subclasses
  - This choice allows any method defined for an A_1 to be used on a subclass A_2
    - without any change to the implementation!
- Consider layout for $A_n \leq \ldots A_3 \leq A_2 \leq A_1$:



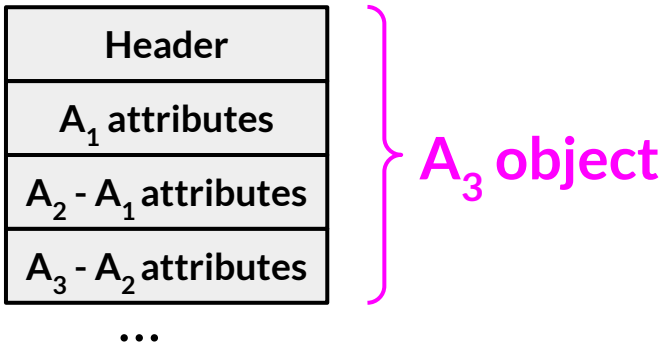
Object Layout: Subclass Invariant

- The **offset for an attribute** is the **same** in a class and all of its subclasses
  - This choice allows any method defined for an A_1 to be used on a subclass A_2
    - without any change to the implementation!
- Consider layout for $A_n \leq \ldots A_3 \leq A_2 \leq A_1$:



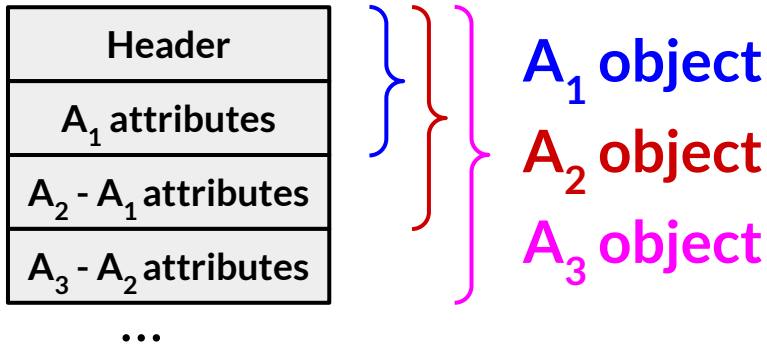
Object Layout: Subclass Invariant

- The **offset for an attribute** is the **same** in a class and all of its subclasses
  - This choice allows any method defined for an A_1 to be used on a subclass A_2
    - without any change to the implementation!
- Consider layout for $A_n \leq \ldots A_3 \leq A_2 \leq A_1$:



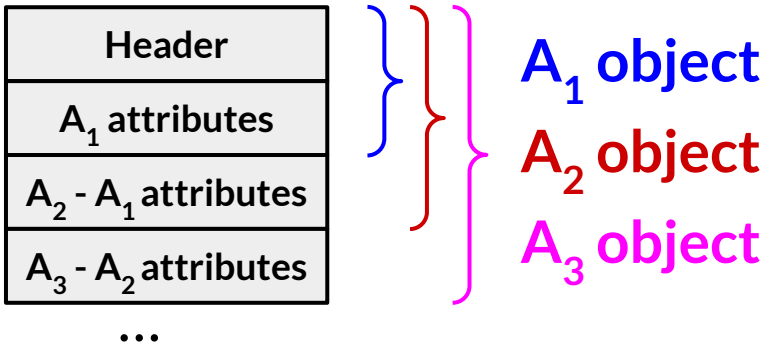
Object Layout: Subclass Invariant

- The **offset for an attribute** is the **same** in a class and all of its subclasses
  - This choice allows any method defined for an A_1 to be used on a subclass A_2
    - without any change to the implementation!
- Consider layout for $A_n \leq \ldots A_3 \leq A_2 \leq A_1$:



Object Layout: Subclass Invariant

- The **offset for an attribute** is the **same** in a class and all of its subclasses
  - This choice allows any method defined for an A_1 to be used on a subclass A_2
    - without any change to the implementation!
- Consider layout for $A_n \leq \ldots A_3 \leq A_2 \leq A_1$:



Challenge question: what about **multiple inheritance**, as in C++?

Cool Object Layout: Dynamic Dispatch

Cool Object Layout: Dynamic Dispatch

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

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 };
};
```

Consider f() and g()

Cool Object Layout: Dynamic Dispatch

- Consider `e.g()`

Cool Object Layout: Dynamic Dispatch

- Consider `e.g()`
  - `g` refers to method in B if `e` is a B

Cool Object Layout: Dynamic Dispatch

- Consider `e.g()`
  - `g` refers to method in B if `e` is a B
- Consider `e.f()`

Cool Object Layout: Dynamic Dispatch

- Consider `e.g()`
  - `g` refers to method in B if `e` is a B
- Consider `e.f()`
  - `f` refers to method in A if `f` is an A or C (inherited in the case of C)

Cool Object Layout: Dynamic Dispatch

- Consider `e.g()`
  - `g` refers to method in B if `e` is a B
- Consider `e.f()`
  - `f` refers to method in A if `f` is an A or C (inherited in the case of C)
  - `f` refers to method in B for a B object

Cool Object Layout: Dynamic Dispatch

- Consider `e.g()`
  - `g` refers to method in B if `e` is a B
- Consider `e.f()`
  - `f` refers to method in A if `f` is an A or C (inherited in the case of C)
  - `f` refers to method in B for a B object
- There is a correspondence here: the implementation of methods and dynamic dispatch **strongly resembles** the implementation of attributes

Cool Object Layout: Dispatch Tables

- Assumption: every class has a **fixed** set of methods (including inherited methods)

Cool Object Layout: Dispatch Tables

- Assumption: every class has a **fixed** set of methods (including inherited methods)
- A ***dispatch table*** (or ***virtual function table*** or ***vtable***) indexes these methods

Cool Object Layout: Dispatch Tables

- Assumption: every class has a **fixed** set of methods (including inherited methods)
- A ***dispatch table*** (or ***virtual function table*** or ***vtable***) indexes these methods
  - A vtable is an array of method entry points
    - Thus, a vtable is an array of function pointers.

Cool Object Layout: Dispatch Tables

- Assumption: every class has a **fixed** set of methods (including inherited methods)
- A ***dispatch table*** (or ***virtual function table*** or ***vtable***) indexes these methods
  - A vtable is an array of method entry points
    - Thus, a vtable is an array of function pointers.
- A method f lives at a **fixed offset** in the dispatch table for a class **and all of its subclasses**
  - this works exactly the same way that attributes do

Cool Object Layout: Dispatch Table Example

Cool Object Layout: Dispatch Table Example

Class / Offset	A	B	C
0	f_A	f_B	f_A
1		g	h

Cool Object Layout: Dispatch Table Example

- The dispatch table for class A has only 1 method

Class / Offset	A	B	C
0	f_A	f_B	f_A
1		g	h

Cool Object Layout: Dispatch Table Example

- The dispatch table for class A has only 1 method
- The tables for B and C extend the table for A with more methods

Class / Offset	A	B	C
0	f_A	f_B	f_A
1		g	h

Cool Object Layout: Dispatch Table Example

- The dispatch table for class A has only 1 method
- The 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
  - (i.e., offset 0 here)

Class / Offset	A	B	C
0	f_A	f_B	f_A
1		g	h

Cool Object Layout: Using Dispatch Tables

Cool Object Layout: Using Dispatch Tables

- The dispatch pointer in an object of class X points to the dispatch table for class X

Cool Object Layout: Using Dispatch Tables

- The dispatch pointer in an object of class X points to the dispatch table for class X
  - i.e., all objects of class X **share one table**

Cool Object Layout: Using Dispatch Tables

- The dispatch pointer in an object of class `X` points to the dispatch table for class `X`
  - i.e., all objects of class `X` **share one table**
- Every method `f` of class `X` is assigned an offset O_f in the dispatch table **at compile time**

Cool Object Layout: Using Dispatch Tables

- The dispatch pointer in an object of class `X` points to the dispatch table for class `X`
  - i.e., all objects of class `X` **share one table**
- Every method `f` of class `X` is assigned an offset O_f in the dispatch table **at compile time**
- To implement a dynamic dispatch `e.f()` we:

Cool Object Layout: Using Dispatch Tables

- The dispatch pointer in an object of class `X` points to the dispatch table for class `X`
  - i.e., all objects of class `X` **share one table**
- Every method `f` of class `X` is assigned an offset O_f in the dispatch table **at compile time**
- To implement a dynamic dispatch `e.f()` we:
  - Evaluate `e`, obtaining an object `x`

Cool Object Layout: Using Dispatch Tables

- The dispatch pointer in an object of class `X` points to the dispatch table for class `X`
  - i.e., all objects of class `X` **share one table**
- Every method `f` of class `X` is assigned an offset O_f in the dispatch table **at compile time**
- To implement a dynamic dispatch `e.f()` we:
  - Evaluate `e`, obtaining an object `x`
  - Find `D` by reading the dispatch-table field of `x`

Cool Object Layout: Using Dispatch Tables

- The dispatch pointer in an object of class `X` points to the dispatch table for class `X`
  - i.e., all objects of class `X` **share one table**
- Every method `f` of class `X` is assigned an offset O_f in the dispatch table **at compile time**
- To implement a dynamic dispatch `e.f()` we:
  - Evaluate `e`, obtaining an object `x`
  - Find `D` by reading the dispatch-table field of `x`
  - Call $D[O_f](x)$

Cool Object Layout: Using Dispatch Tables

- The dispatch pointer in an object of class `X` points to the dispatch table for class `X`
  - i.e., all objects of class `X` **share one table**
- Every method `f` of class `X` is assigned an offset O_f in the dispatch table **at compile time**
- To implement a dynamic dispatch `e.f()` we:
  - Evaluate `e`, obtaining an object `x`
  - Find `D` by reading the dispatch-table field of `x`
  - Call $D[O_f](x)$
    - `D` is the dispatch table for `x`

Cool Object Layout: Using Dispatch Tables

- The dispatch pointer in an object of class `X` points to the dispatch table for class `X`
  - i.e., all objects of class `X` **share one table**
- Every method `f` of class `X` is assigned an offset O_f in the dispatch table **at compile time**
- To implement a dynamic dispatch `e.f()` we:
  - Evaluate `e`, obtaining an object `x`
  - Find `D` by reading the dispatch-table field of `x`
  - Call $D[O_f](x)$
    - `D` is the dispatch table for `x`
    - In the call, `self` is bound to `x` (why?)

Cool Object Layout: Dyn. Dispatch Codegen

- Cgen for `objexp.mname(arg1)`:

Cool Object Layout: Dyn. Dispatch Codegen

- Cgen for `objexp.mname(arg1)`:

push self

Cool Object Layout: Dyn. Dispatch Codegen

- Cgen for `objexp.mname(arg1)`:

```
push self
push fp
```

Cool Object Layout: Dyn. Dispatch Codegen

- Cgen for `objexp.mname(arg1)`:

push self
push fp
cgen(arg1)

Cool Object Layout: Dyn. Dispatch Codegen

- Cgen for `objexp.mname(arg1)`:

```
push self
push fp
cgen(arg1)
push r1                                  ; push arg1
```

Cool Object Layout: Dyn. Dispatch Codegen

- Cgen for `objexp.mname(arg1)`:

```
push self
push fp
cgen(arg1)
push r1                          ; push arg1
cgen(objexp)
```

Cool Object Layout: Dyn. Dispatch Codegen

- Cgen for `objexp.mname(arg1)`:

```
push self
push fp
cgen(arg1)
push r1                                  ; push arg1
cgen(objexp)
bz r1 dispatch_on_void_error
```

Cool Object Layout: Dyn. Dispatch Codegen

- Cgen for `objexp.mname(arg1)`:

```
push self
push fp
cgen(arg1)
push r1                              ; push arg1
cgen(objexp)
bz r1 dispatch_on_void_error
push r1                              ; will be “self” for callee
```

Cool Object Layout: Dyn. Dispatch Codegen

- Cgen for `objexp.mname(arg1)`:

```
push self
push fp
cgen(arg1)
push r1                              ; push arg1
cgen(objexp)
bz r1 dispatch_on_void_error
push r1                              ; will be “self” for callee
ld temp<-r1[2]                       ; temp <- vtable
```

Cool Object Layout: Dyn. Dispatch Codegen

- Cgen for `objexp.mname(arg1)`:

```
push self
push fp
cgen(arg1)
push r1                                  ; push arg1
cgen(objexp)
bz r1 dispatch_on_void_error
push r1                                  ; will be “self” for callee
ld temp<-r1[2]                           ; temp <- vtable
ld temp <- temp[X]                       ; X is offset of mname in vtables
                                         ; for objects of typeof(objexp)
```

Cool Object Layout: Dyn. Dispatch Codegen

- Cgen for `objexp.mname(arg1)`:

```
push self
push fp
cgen(arg1)
push r1                          ; push arg1
cgen(objexp)
bz r1 dispatch_on_void_error
push r1                          ; will be “self” for callee
ld temp<-r1[2]                   ; temp <- vtable
ld temp <- temp[X]               ; X is offset of mname in vtables
                                 ; for objects of typeof(objexp)
call temp
```

Cool Object Layout: Dyn. Dispatch Codegen

- Cgen for `objexp.mname(arg1)`:

```
push self
push fp
cgen(arg1)
push r1                          ; push arg1
cgen(objexp)
bz r1 dispatch_on_void_error
push r1                          ; will be “self” for callee
ld temp<-r1[2]                   ; temp <- vtable
ld temp <- temp[X]               ; X is offset of mname in vtables
                                 ; for objects of typeof(objexp)
call temp
pop fp
```

Course Announcements

- We recently fixed a bug in the reference compiler's x86-64 module. Only use Cool **version 1.39** for compiling to x86.
  - There will be an extra credit question on the midterm asking why I had to do this, if you want to do a comparison.
  - I will also award extra credit if you can find another bug in the reference compiler
- Don't forget there is a **midterm** in this class on Wednesday!
  - Review session: tonight at 5pm (virtually)
  - Extra office half-hours tomorrow at 10am, 4:30pm
- Hopefully you started PA3c3 over break
  - its due date is **one week from today**