Hack Your Language!

CSE401 Winter 2016

Introduction to Compiler Construction

Ras Bodik
Alvin Cheung
Maaz Ahmad
Talia Ringer
Ben Tebbs

Lecture 7: Implementing Arrowlets

Implementing event handlers
Continuation-passing style

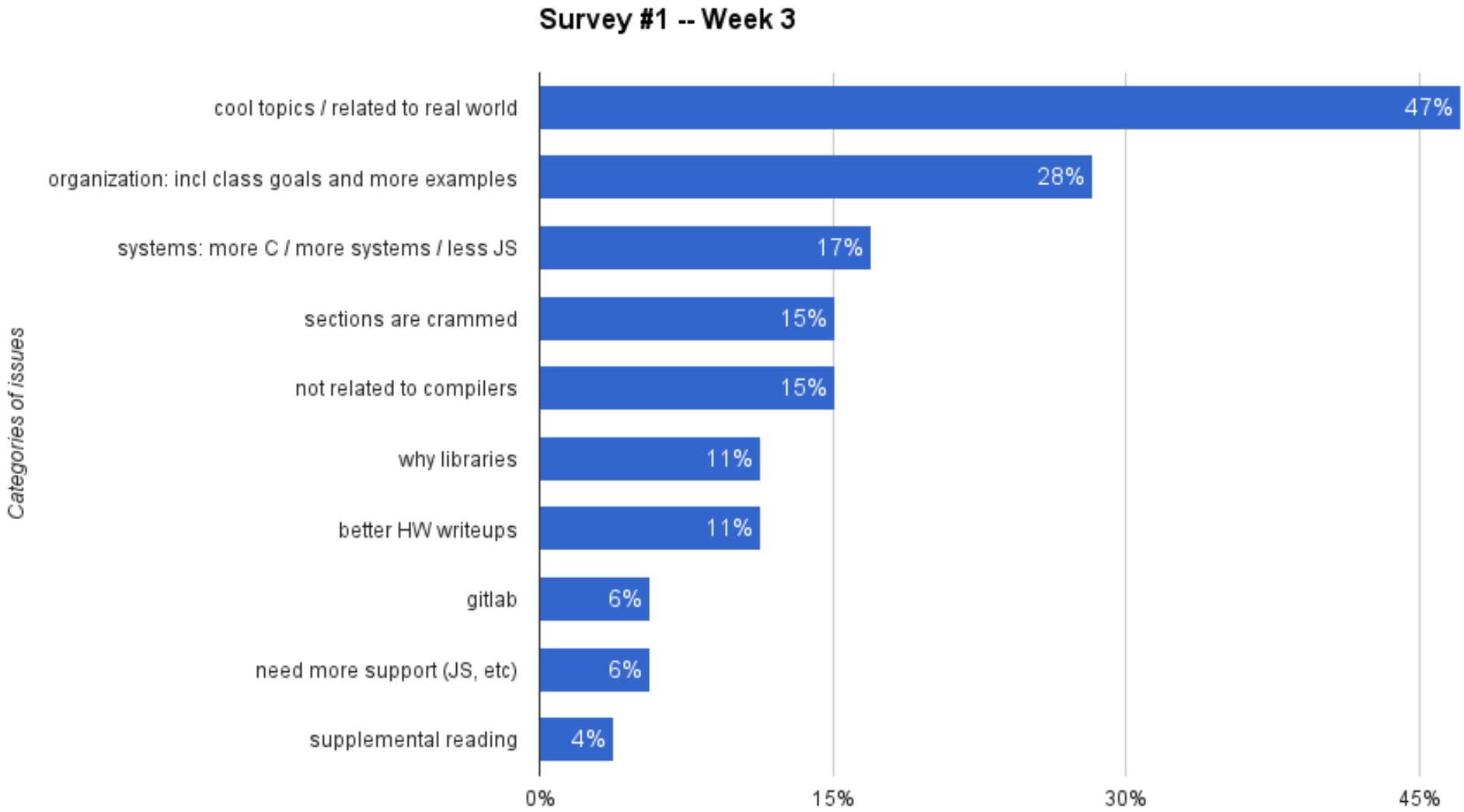
Announcements

- PA2 due this Sunday
  - why work in group of 3
- Regular OHs this week
  - See website for details

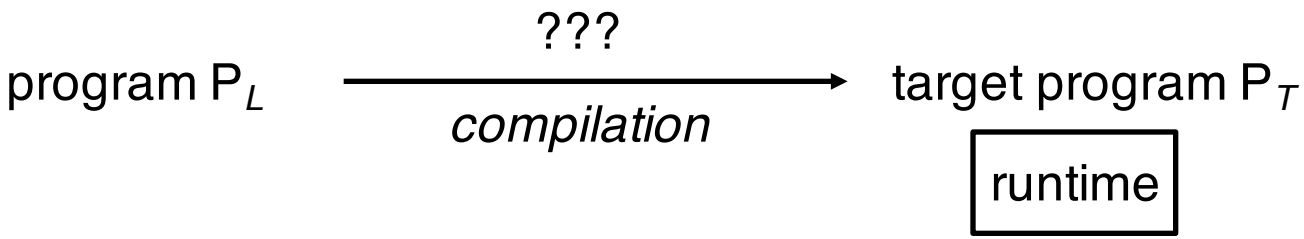
Final project timeline

- now - 2/3: project proposal
- 2/4 - 2/14: milestone 1: write sample programs in your language, and review two other teams' by doing the same
- 2/16 - 2/17: team meeting with course staff
- 2/18 - 3/2: milestone 2: implementation plan
- 3/3 - 3/10: milestone 3: implementation checkpoint
- 3/15: final project presentation

Student survey, top 5 concerns (53 responses)

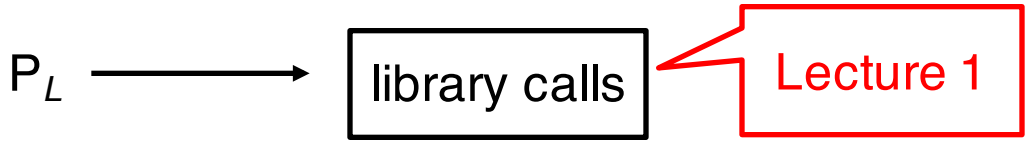


401: Implementing programming languages

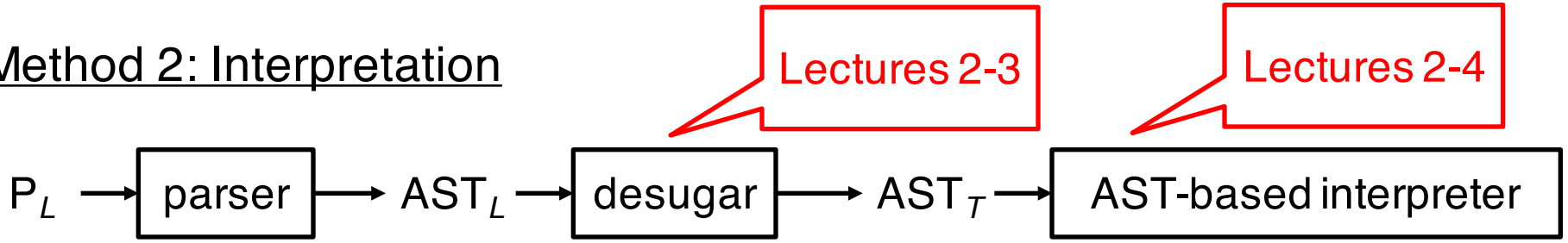


401: Implementing programming languages

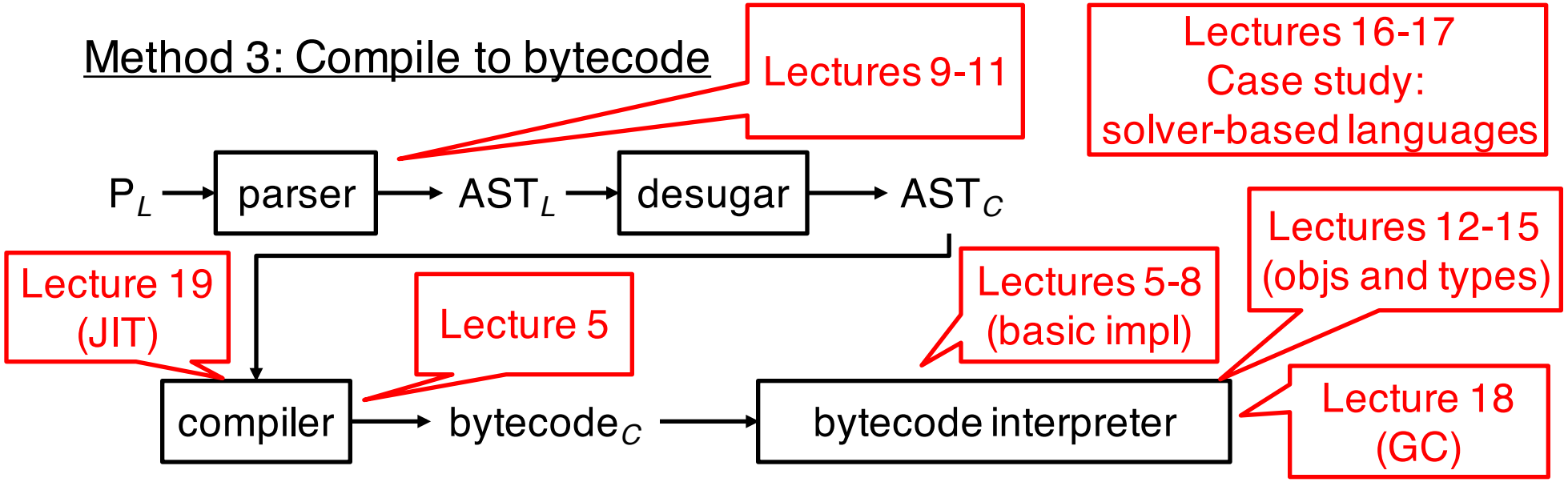
Method 1: Implement as library



Method 2: Interpretation



Method 3: Compile to bytecode



Keep in mind

Desugaring is compiling:

- it lowers code to a lower language

Translating to bytecode is compiling:

- Compiling to x86 is similar

Using real-world languages as examples

- Google unit calculator
- D3
- JavaScript
- Arrowlets
- Rx / Vega
- Lua
- Rosette
- Java (GC, JIT)

- The lessons you learn are equally applicable to other compilers (gcc, javac, gfortran, …) and target languages (x86, CPython, …)

Some job stats (from dice.com)

Javascript Compiler jobs

1 - 30 of 11621 positions

C++ Compiler jobs

1 - 30 of 4307 positions

X86 Compiler jobs

1 - 30 of 201 positions

Minijava Compiler jobs

1 - 30 of 50 positions

Just for fun ☺

Class evaluations

- This is *not* 341 or an advanced (web) prog. class
  - We do not survey different programming paradigms
  - We choose common useful features across languages and show how they are compiled and executed
  - We use examples from the web as you interact with them everyday
- We use *real-world languages* to show how such features are used
  - We expect you to pick up language syntax on your own (you don't need to understand the full syntax)
  - You will be designing and implementing your own for the final project!

Where are we in the course

- Different modes of compilation
  - Library, embedding, bytecode
- Control abstractions
  - Transfer of control
- This is where a lot of changes happen in today's PL
  - Parallelism
  - Distributed computing
  - Backtracking
  - Sampling (MCMC)

Functions as “jumps”

Implementing control flow

Classical compilation vs. thunks

We added this section of the lecture to show you how functions implement transfer of control and how it compares to what the C compiler does.

We will use a while loop as a case study.

Compiling the while loop in C

in a C compiler, `while(E)S` is compiled to

```
        …
L1:     compilation of E          compilation of while(E)S
        …
        branch if zero to L2
        compilation of S
        …
        jump to L1
L2:     nop
        …
```

Only one jump + one fall-through branch / iteration!

The compiler algorithm

Compilation of the AST `while(E)S` to machine code:

Works like compilation to bytecode

- bottom-up code generation on the AST
- compiled code bubbles up and is composed into more code
- the root produces code for the entire AST

One extension wrt compilation to bytecode

- must generate fresh labels L_i to pass down to AST subtrees
- necessary to generate jumps to appropriate labels
- **Example:** on previous slide,
  - `break` inside S compiles to "jump to L2"
  - `continue` inside S compiles to "jump to L1"
- **Note:** if break and continue are in a nested inner loop, that loop will generate its fresh labels for its breaks and continues
- These labels are symbolic assembler labels, not actual addresses

Optimizations

For simplicity, our code generates a no-op at label L2

- Rationale: compiled code does not end with a hanging label that would need to be carefully attached to the adjacent machine code.

We could write a more careful code generator

- It could avoid this no-op as well as jumps that are immediately followed by jumps. But it may be less complex to delegate these optimizations to the linker which will see the entire machine code.

Compiling the while loop in lambda language

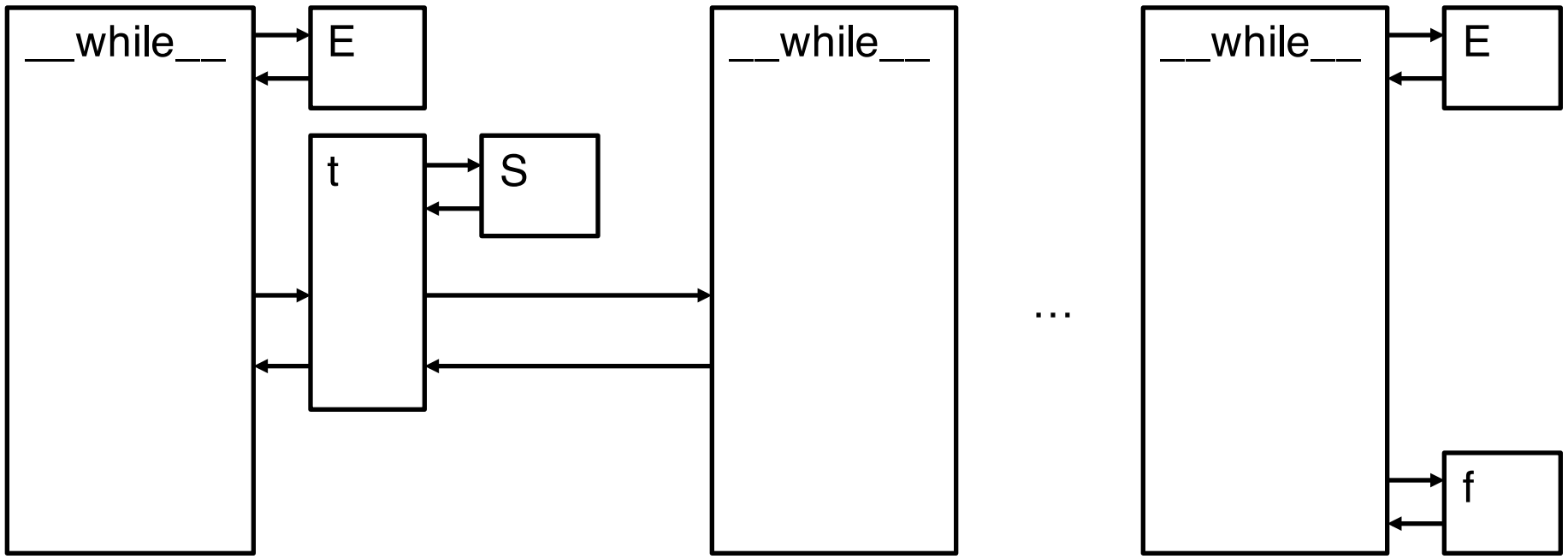
In a functional language, `while(E)S` is desugared to

```
__while__(function(){E}, function(){S})
```

```
def __while__(e, s) {
    def t = function() {s()3; __while__(e,s)4}
    def f = function() {}
    // call t or f
    ite(e()1,t,f)()2
}
```

Appears less efficient: 4 calls and 4 returns / iteration

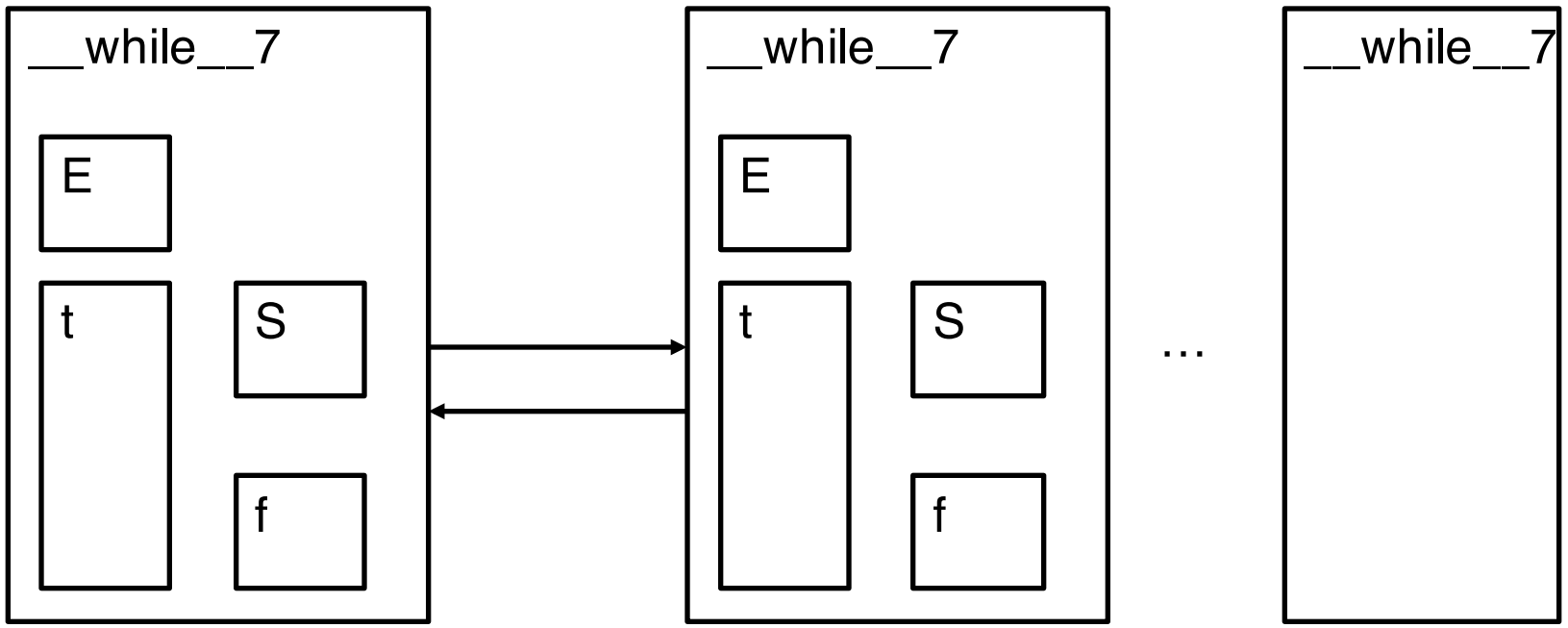
The functional while loop, unrolled



Eight control transfers per loop iteration.

Also, the call stack overflows on long-running loops.

Step 1: after function inlining



inlining: the body of a function f replaces f().

The compiler must be certain that the call instruction always calls the same function.

Optimized desugared while loop in detail

```
// call __while__ specialized to this while loop
__while__7()   // previously was __while__(thunk,thunk)
def __while__7() {// here's the specialized __while__
    $1 = E  // E is the result of inlining e()
    if ($1) {
          S  // s() inlined into t
          __while__7()
    } else { /* the empty f is inlined here */ } }
```

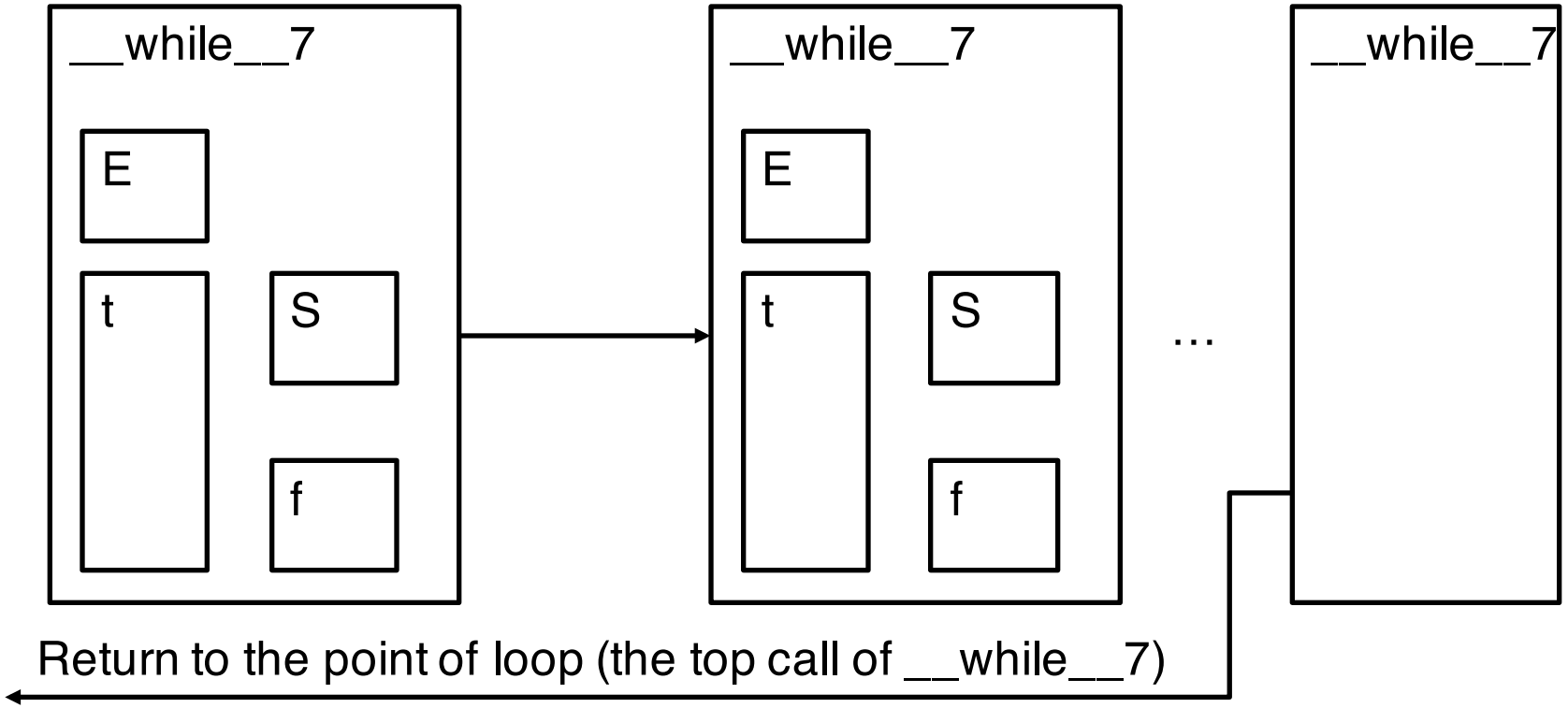
Note 1: Besides inlining, we need to create a copy of __while__ specialized to E and S, it to the given while loop. In the lecture, we said that we can instead inline __while__ to the point of the loop. We could do this, too, but it's trickier to explain.

Note 2: The call $()_2$ in **ite**($1,t,f)$()_2$ does not always call the same function. Hence we need to inline the two calls (t() and f()) separately and chose one of them using a condiitonal. This is called polymorphic inlining.

Note 3: The polymorphic inlining requires that the low-level language into which we are desugaring include the if statement (eg branch if zer to a label). This is typical in compilers: sometimes to express an optimization, you need the target language to have certain low-level instructions.

Step 2: after tail call elimination

One control transfer per iteration, as in C version.



Def: f() in function g() { f() } is a tail call because no work is done in g between the return from f and the return from g. **Observation:** there is no need to return to g. Instead, return to the caller of g. **Benefits** of such tail call optimization (TCO): when calling f(), there's no need to grow the call stack because we'll not return to g.

Discussion

Compilation of control flow constructs like while (E) S by desugaring allows creating new constructs

- e.g., Python does not have a switch statement.
  It would be nice if the programmer could just add it.

Efficiency (nearly) comparable to C via optimizations

- No need to generate optimal code separately for each construct you have. Instead, desugar in a simple way and then optimize all constructs with the same optimizer.

Programming with Arrowlets

Assigned reading: paper on Arrowlets

What you will see today:

- Implementing arrowlets using functions as control transfers

Why reactive / event-driven programming

Common programming paradigm for many tasks:

- User interface (buttons, menus, game controls)
- Databases (disk and packet requests, hw interrupts)
- Robotics (move arm when certain event happens)
- Virtual Reality and Video Games
- Car and airplane controller

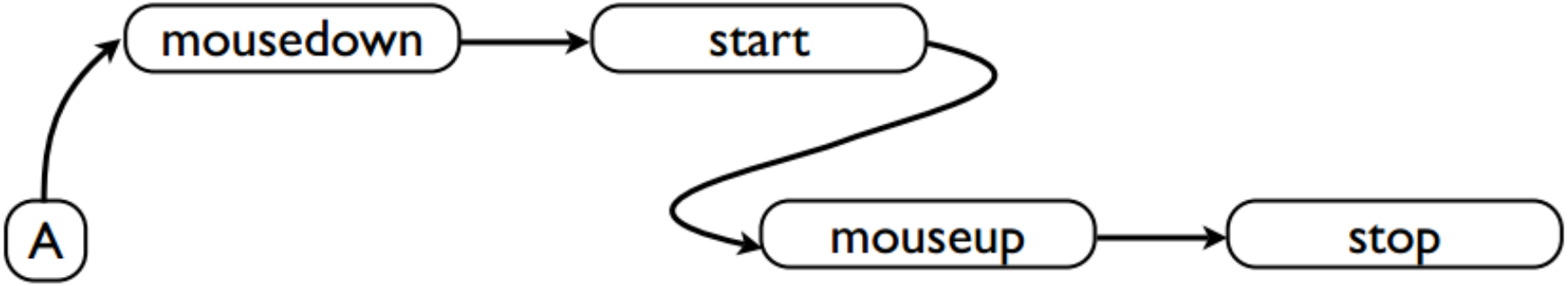
All languages support reactive programming:

- Java (swing), Haskell (),
- Visual Basic, JavaScript (DOM),
- (Your final project)

Refresher from the previous lecture

A simple Arrowlets program:

- The state machine:

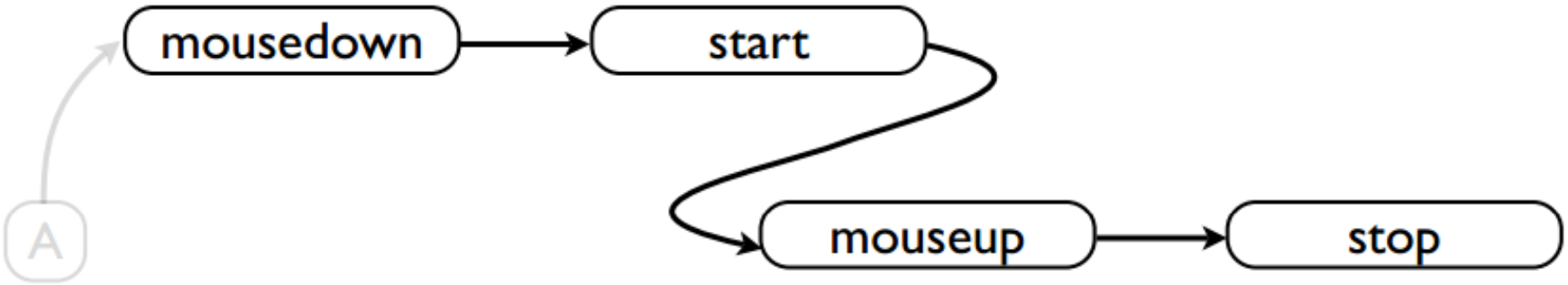


- Compose using Arrowlets:

```
var step1 = EventA("mousedown").bind(start);
var step2 = EventA("mouseup").bind(stop);
var step1and2 = step1.next(step2);
step1and2.run(A);
```

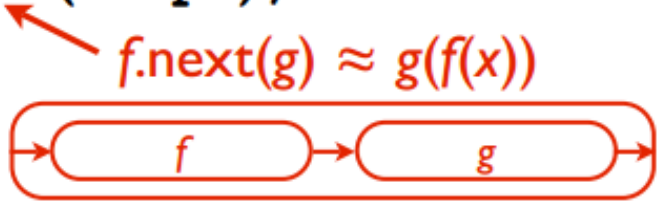
The next combinator

- The state machine:



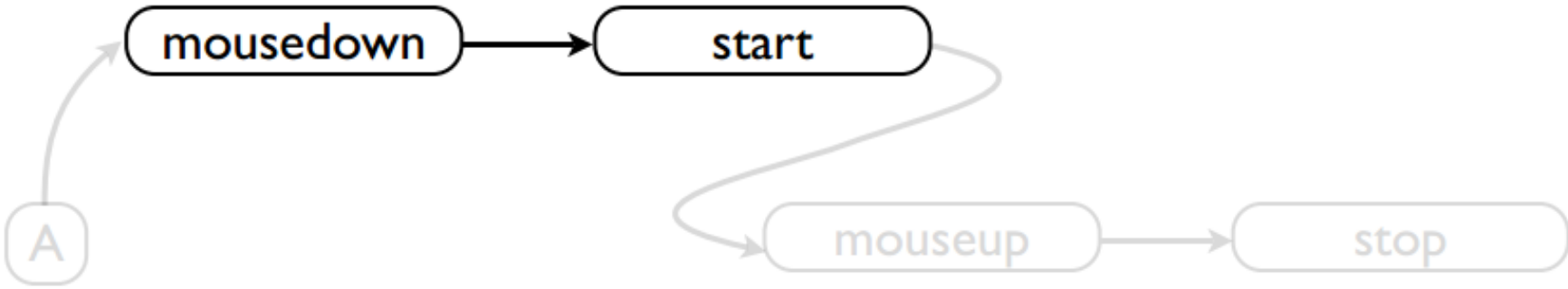
- Compose using Arrowlets:

```
var step1 = EventA("mousedown").bind(start);
var step2 = EventA("mouseup").bind(stop);
var step1and2 = step1.next(step2);
```



The bind combinator

- The state machine:

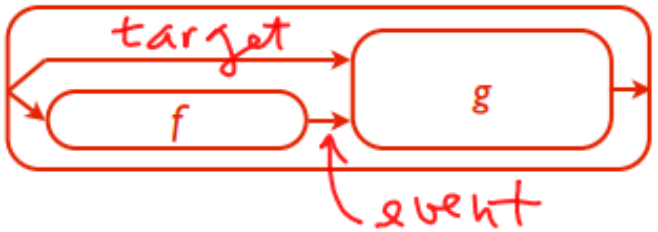


- Compose using Arrowlets:

```
var step1 = EventA("mousedown").bind(start);
```

EventA("event")

f.bind(g) ≈ g(f(x), x)



Summary of Arrowlets Combinators

Combinator	**Use**
h = f.next(g)	h(x) is g(f(x))
h = f.bind(g)	h(x) is g(x, f (x))
h = f.product(g)	h takes a pair (a,b) as input; h((a,b)) is (f (a), g(b))
h = f. repeat()	h calls f with its input; if the output of f is: Repeat(x), then f is called again with x; Done(x), then x is returned
h = f.or(g)	h executes whichever of f and g is triggered first, and cancels the other
h.run(x)	begins execution of h, passing x to h
f.AsyncA()	lift function f into an arrow (called internally by combinators)

Arrows

The design of Arrowlets is based on the concept of arrows popularized by the Haskell language.

Arrows help improve modularity, by separating composition strategies from actual computations. Arrows help isolate program concerns.

Arrows are flexible, because operations can be composed in many different ways.

Arrow

An arrow is a so-called *lifting* class with two methods

A(f)	construct an arrow from fun f
	lift f to the "world" of arrows
next(a1,a2)	compose arrows a1, a2

It is sufficient for us to understand A and next.

Remaining operators are implemented ~ on top of these.

FYI: Haskell arrows

```
class Arrow a where
    arr :: (b → c) → a b c
  (>>>) :: a b c → a c d → a b d


instance Arrow (→) where
   arr f       = f
   (f >>> g) x = g (f x)
```

Our implementation plan

We'll start with a toy implementation, then grow it:

1. Function Arrows (no Events)
2. CPS Function Arrows (still no Events)

3. Simple Async Event Arrows (one Evt chain, no repeat) [next lecture]

Function Arrows

composing functions using direct-style calls

Programs we want to write

This subset of Arrowlets can compose ordinary functions (but not events).

We want to support programs like this:

```
function add1(x) { return x + 1; }
var add2 = add1.next(add1);
var result = add2(1);   /*returns 3 */
```

Truly faithful to Arrowlets operators, we should write:

```
function add1(x) { return x + 1; }
var add2 = (add1.A()).next(add1.A());
var result = add2.run(1);   /*returns 3 */
```

Function arrows in JS

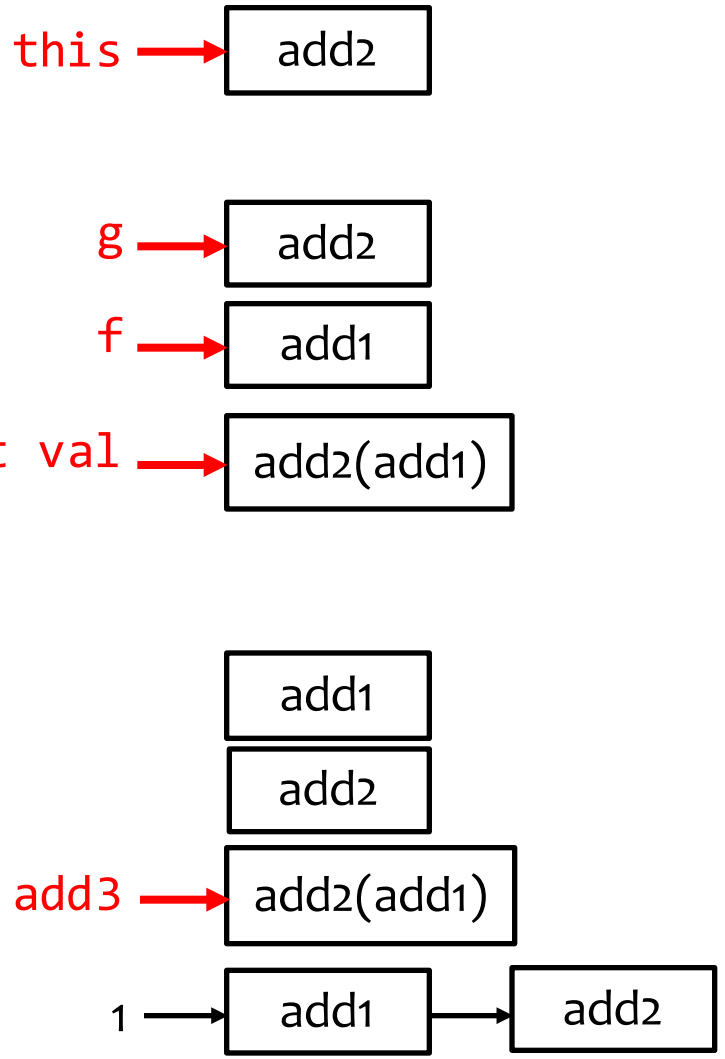
```
// in JS, each function is an object of class Function.  It has callable methods.
// Here we add methods A and next to all functions in the JS program.
Function.prototype.A = function() {
    return this ;
}
Function.prototype.next = function(g) {
   var f = this; // in a call foo.next(bar), this binds to foo
   g = g.A();   // dyn type check: passes if g a fun, fails if it is, say, an int
   return function(x) { return g(f(x)); } // compose f and g
}
// Now we can run our Arrowlets program:
function add1(x) { return x + 1; }
var add2 = add1.next(add1);
var result = add2(1);   /*returns 3 */
```

Another example

```
Function.prototype.A = function()
{ return this ; }

Function.prototype.next =
function(g) {
   var f = this;
   g = g.A();
   return function(x) {
        return g(f(x)); }
}

function add1(x) { return x + 1;}
function add2(x) { return x + 2;}
var add3 = add1.next(add2);
var result = add3(1);
```



Summary

We did nothing more than compose functions

- We implement the next construct as a library function
- next creates a new program from two smaller programs

The only "special" trick: add A and next to Function

- This added more functionality to JS functions

Exercise: implement arrowlet.run(x)

Intermezzo:
JavaScript Handlers and CPS

In this section

Before we move on to CPS Arrows, we need to understand the rationale for CPS

- The rationale is a limitation of our target language (i.e. stuff we cannot do in the JS event model).

CPS = continuation-passing style (not as scary as it may sound)

JavaScript Event Semantics

This is impossible to do with JS events

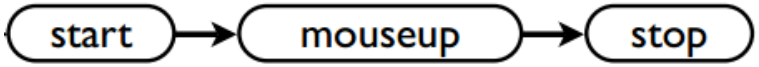
```
function my_handler(event) {
    foo();
    wait_for_event("click", some_target);
    bar();
}
```

It's impossible because JS event handlers are *atomic*:

- an event handler can't suspend it's execution
  - it can only register new handlers, for events and for the timer
- the handler always finishes and returns to the "event loop"
  - Handler execution is single-threaded (i.e., one handler at a time)
- in some ways, this single-threaded no-preemption is good
  - because it prevents data races among threads
- we'll see shortly how CPS overcomes this restriction

Overcoming JavaScript Event Semantics

Atomicity of handler execution explains why we cannot translate an Arrowlet program like this one



into JS code shown on the previous slide.

Instead, we need to generate code like this:

```
function my_handler(event) {
    foo();
    some_target.addEventListener("click", continuation);
    bar();
}
function continuation(event) { bar(); }
```

Continuation-passing-style (CPS) functions

A CPS function f takes

- a normal argument x and
- a continuation k

The *continuation* is a function

- it executes the "rest of the program"
- its single argument receives f's return value

So, after f evaluates its body, instead of returning to the caller, f calls k, passing to k the return value of f.

- instead of a *return*, f performs a *call*
  - Note that both return and call are control transfer constructs
  - CPS just encodes the execution differently. Why?
  - So that we can continue the execution after an event.

CPS example and properties

Direct style:

```
def g(x) { return x/2; }        print g(1)+2
```

CPS style:

```
def g(x,k) { k(x/2); }          g(1, function(v){print v+2})
```

In CPS style, all calls are *tail calls* (recall PA2)

TCO ensures that the call stack does not grow as the execution proceeds.

JS machines do not have tail call optimizations (yet)

Example

```
def g(x,k) { k(x/2); }
g(1, function(v){print v+2})
```

```
def g(x,k) {
  k(x/2)
}

g(1, fn(v){print v+2})
```

```
function(v) {
  print v+2
}
```

Call stack:

function	arguments
g	1, fn(v){ … }

Example

```
def g(x,k) { k(x/2); }
g(1, function(v){print v+2})
```

```
def g(x,k) {
  k(x/2)
}

g(1, fn(v){print v+2})
```

```
function(v) {
  print v+2
}
```

Call stack:

function	arguments
g	1, fn(v){ … }
function(v)	1/2

Summary

We now have a second way to implement arrows using continuations

Composing arrows (i.e., next) is equivalent to composing continuations

We will see next time how to use continuations to implement events