Lecture 3

Building on the core language

Scopes, bindings, syntactic sugar and famous train wrecks

Ras Bodik
Shaon Barman
Thibaud Hottelier

Hack Your Language!

CS164: Introduction to Programming Languages and Compilers, Spring 2012

UC Berkeley

Reflection on HW1

In HW1, most of you wrote your first web mash-up.

Congratulations!!!

Lessons from HW1:

- modern programs use multiple languages: HTML, CSS, JS, regex, d3
- can accomplish a lot with a few lines of code
- but languages and tools not so easy to learn
- in CS164, we'll learn skills to improve tools and languages,
- and learn how to learn languages faster

Our plan

Today's lecture:

- mostly a CS61A refresher, on interpreters and scoping
- get hands-on practice with these concepts in PA1

PA1 assigned today (due Monday):

- we'll give you an interpreter with dynamic scoping
- you extend it with lexical scoping, calls, and little more

PA logistics

- work in teams of two or three (solo not allowed)
- team collaboration via Mercurial (emailing files no ok)

Next lecture:

- developing program constructs with coroutines
- lazy iterators, regular expressions, etc

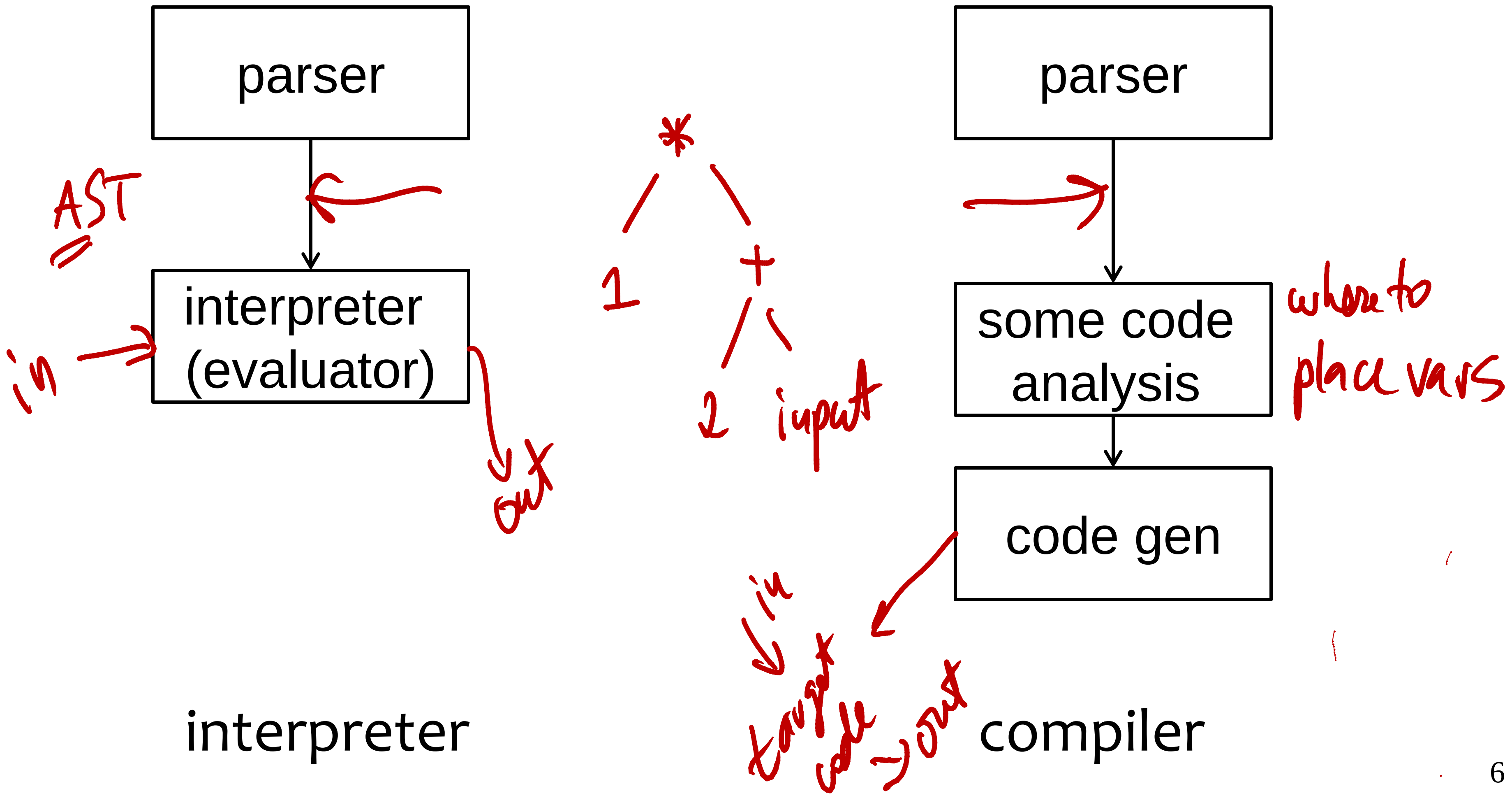
Outline

- The architecture of a compiler and interpreter
- Surface syntax, the core language, and desugaring
- Interpreter for arithmetic and units
- Delaying the evaluation: **Compilation**
- Adding functions and local variables
- We want to reuse code: **high-order functions**
- Nested scopes: **dynamic scoping**
- Fixing language for reusability: **static scoping**

The architecture and program representation

Basic Interpreter and Compiler Architecture

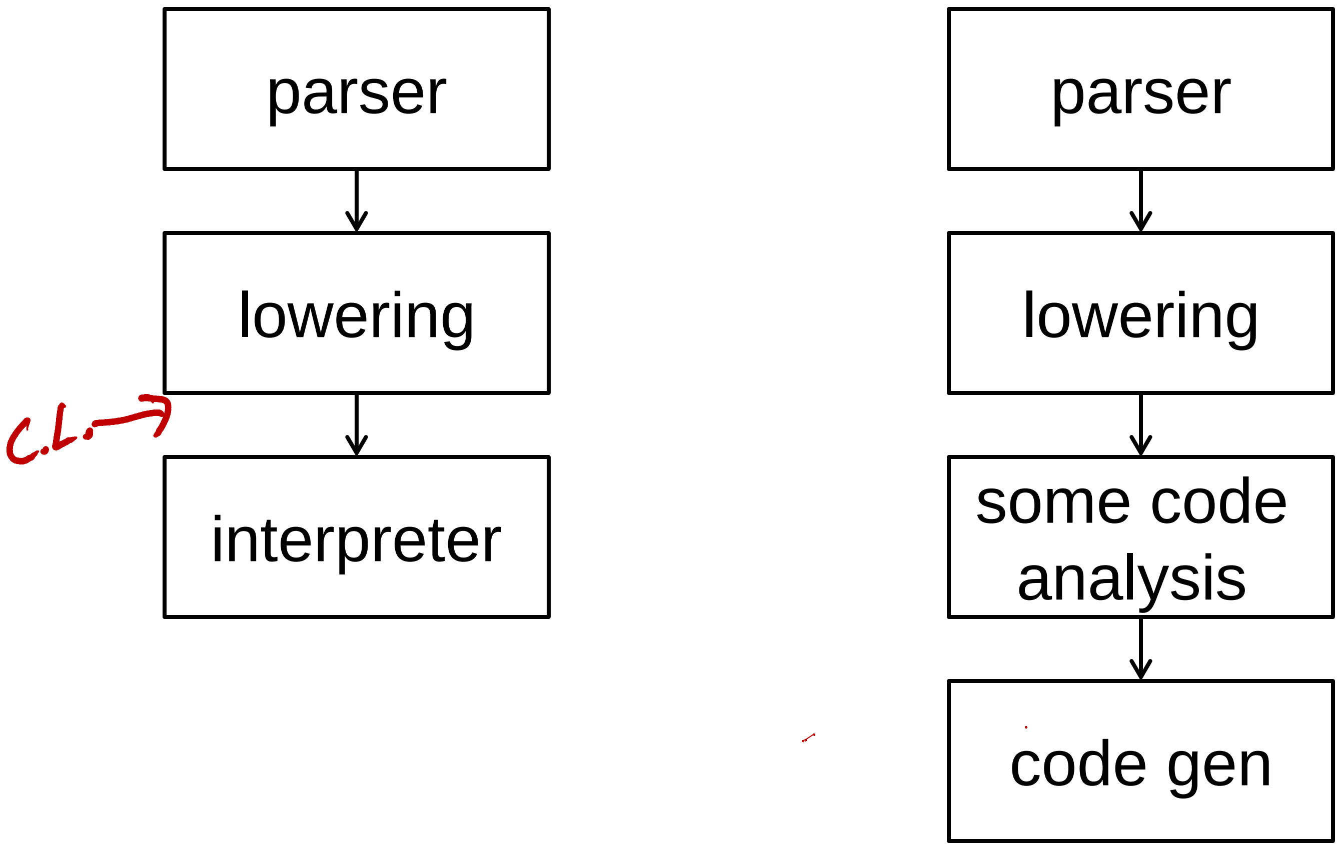
"1 * (2 + input())"



Basic Interpreter and Compiler Architecture

- why AST representation?
  - encodes order of evaluation
  - allows divide and conquer recursive evaluation
- parser turns flat text into a tree
  - so that programmers need no enter trees
- source language vs target language
  - for interpreters, we usually talk about host languages
- AST is a kind of a program intermediate form (IR).
  - an idealized assembly is another. More on this in PA2.

More typical Interpreter and Compiler



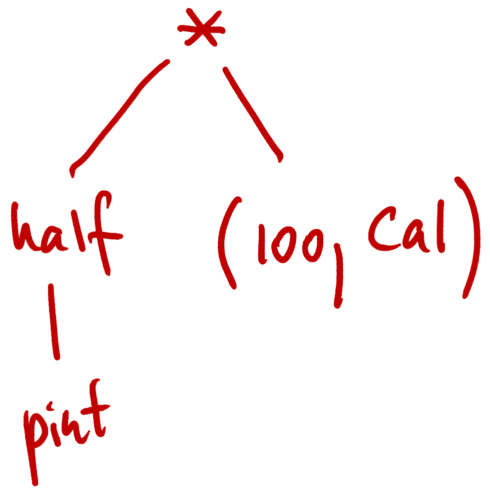
Lowering motivated by desire for simple core lang

Example of code before and after lowering

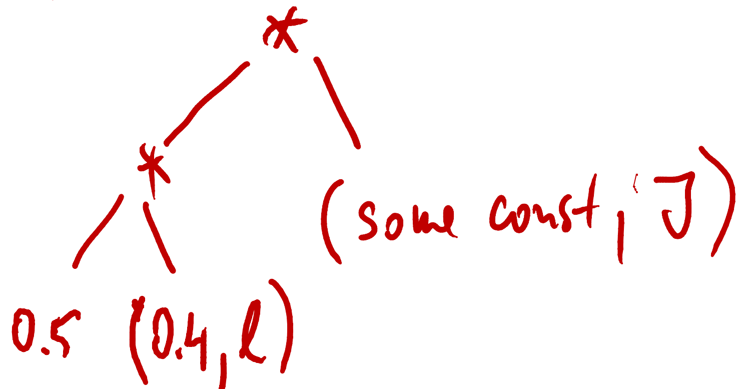
Input to google unit calculator

half a pint * 100 Cal

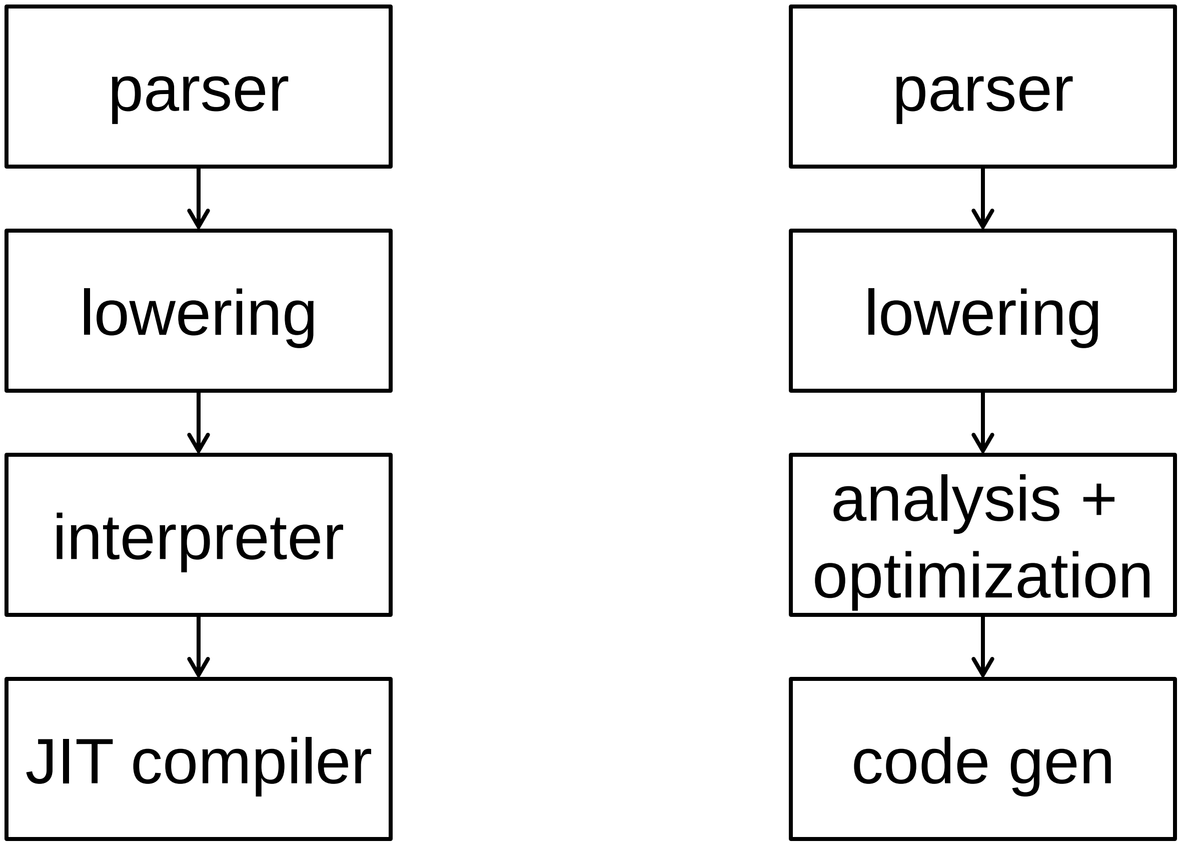
AST after parsing:



AST after lowering:



Advanced Interpreter and Compiler



Example systems: V8 JS engine; gcc C compiler

More on lowering

Effectively, we have two languages

- surface language
- core language

Example:

- for (i = 0,n) { … } is surface syntax
- can be desugared to while (…)

These rewrites are local

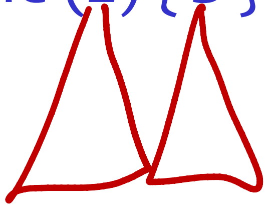
- expand a node (eg for) without looking at the rest of AST

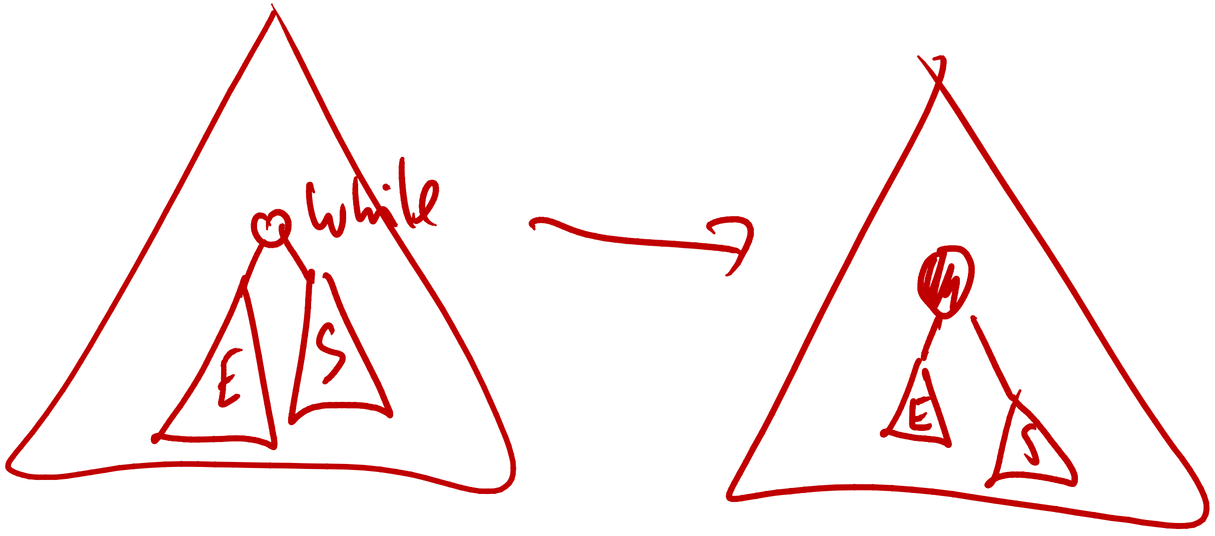
First way to lower the AST: tree rewrite

AST rewriting (sits between parsing and interpreter)

while (***E***) { ***S*** } → parser → AST with *While* node

→ rewriter → AST w/out *While* node





Second way to desugar: in the parser

during "syntax-directed translation" (more later)

while (***E***) { ***S*** } → parser → AST w/out *While* node

```
S ::= 'while' '(' E ')' '{' S_list '}'

   %{ return ('exp', ('call',
                      ('var', 'While'),
                      [('lambda',[], [('exp',n3.val)]),
                       ('lambda',[], n6.val)]))
   %}
```

Summary of key concepts

- Interpreter works on ASTs, programmers on text
- Parser turns text into AST
- AST encodes evaluation order
- Want a simple interpreter ==> a small core language
- Clunky core ==> design friendly surface syntax
- Surface syntax expanded (lowered, desugared)
- Desugaring can happen already in the parser
- Lowering means we have 2 langs: surface and core
- Interpreter evaluates the AST, compiler translates it
- Result of compilation is a program in target lang

Test yourself (what you need to know)

- Design an AST for this code fragment: `x.f[i,j]`.
- Why are parens in (x+y)*z not needed in the AST?
- Rewrite a small JS program into AST form.

The AST interpreter

The core language

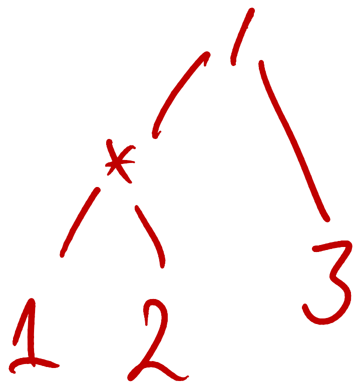
1

First we need to decide what language to interpret

Lets assume expressions over int constants and + - / *.

We say this concisely with a grammar:

E := n | E+E | E-E | E/E | E*E



This (recursive) grammar defines structure of AST:

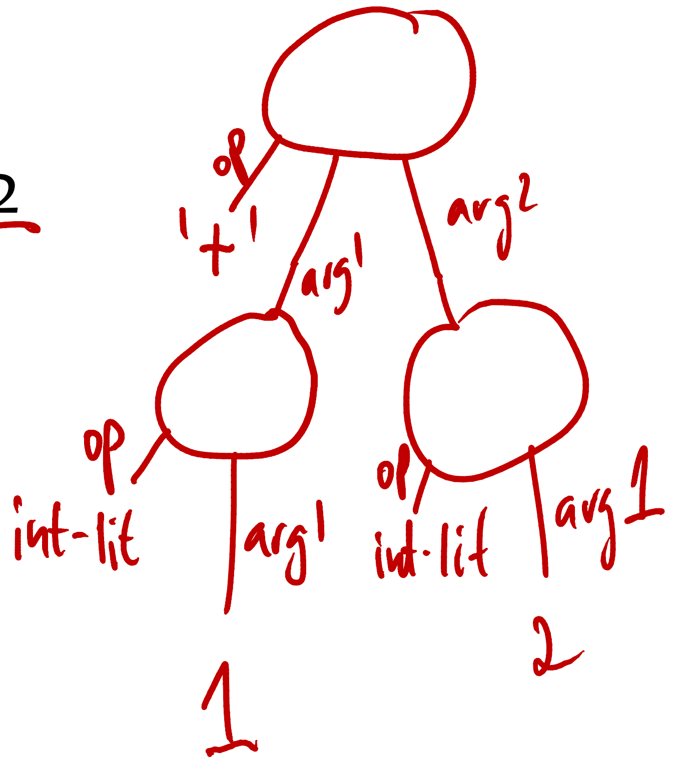
- leaves are integer constants (denoted *n*)
- internal nodes are operators (+ - / *)

A recursive interpreter (part 1)

First, we need to define the AST node

```
class N(op, arg1, arg2) {
        // operator, left and right children
}
```

Example AST, for input 1 + 2



A recursive interpreter (part 2)

Now we can evaluate the AST:

```
// method eval is in the AST node class N
function eval() {
        switch (this.op) {
        case 'int-lit': return this.arg1
        case ’+’:       return this.arg1.eval() + this.arg2.eval();
        case ’*’:       return this.arg1.eval() * this.arg2.eval();
        …
        }
}
```

Intermediate values

Intermediate values = values of sub-expressions

Q: Consider evaluation of `(1+(2+(3+4)))`.
Where are 1 and 2 stored while 3+4 is computed?

A: the stack of the recursive interpreter.

Dynamic type checking

Dynamic type checking: performed at evaluation time

Static is performed at compile time (before inputs known)

Example: 1 m + 2 sec is a type error

note that we converted 1 m + 1 ft into 1 m + 0.3 ft, so no err

Example: object + string should also raise an error

– some languages may coerce object to string and concat
– or do some other strange type conversion

Sample interpreter code:

```
case '/':     var v1 = this.arg1.eval()
              var v2 = this.arg2.eval()
              if v2=0: error ; return v1/v2
```

Runtime type information

Note: values must be tagged with run-time type

or this type info must be obvious from the runtime value

Interpreter for the unit calculator

Same as before except values are pairs (value, unit)

E := (n, U) | E+E | E-E | E/E | E*E
U := m | ft | s | min | …

Operations now must add/mul values as well as types

see posted code of this unit interpreter

Evaluating abstract values

The AST can be "evaluated" in non-standard ways

Example: compute the type of the AST

- more precisely, compute the type of the value that the AST will evaluate to once we know the values in input vars

This type computation is possible even without knowing the input values, provided the programmer annotated the types of variables

Summary of key concepts

- AST interpreter recursively evaluates the AST
- Typically, values flow bottom-up
- Intermediate values stored on interpreter’s stack
- Interpreter performs dynamic type checking
- Values can also be abstract, eg static types

Test yourself

- Extend google calculator to handle *in* conversion, eg `2 ft / s in yard / minute`

Compilation

Compiler: an interpreter that produces code

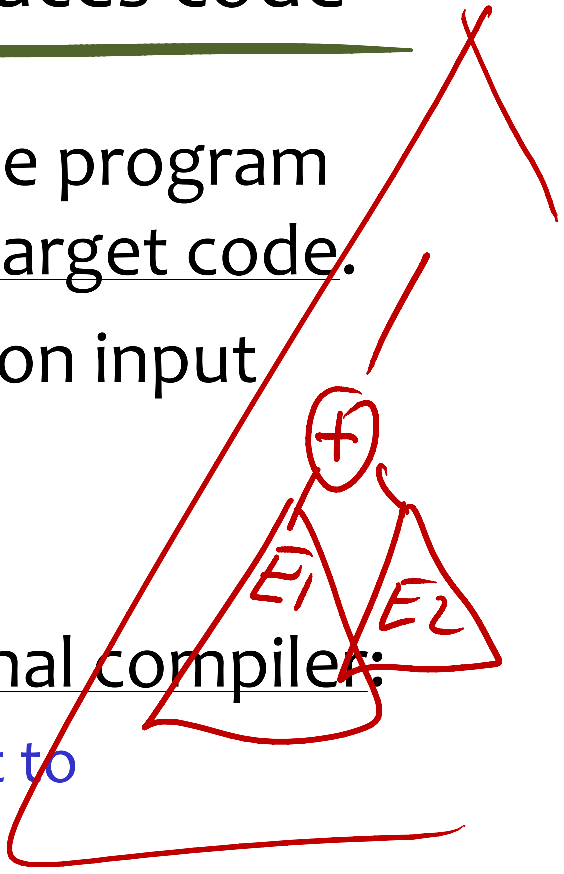
Compilation is a special evaluation of the program (w/out knowing inputs) that produces target code.

This target code, when itself evaluated on input values, evaluates the input program.

The challenge is to define a compositional compiler:

given the AST E of the form *E1+E2*, we want to

i) compile E1 into target code T1
ii) compile E2 into target code T2
iii) create target code T for E by combining T1 and T2 (ideally without changing T1 and T2).



Intermediate values, again

Must agree on where T1 will leave the values of E1.

same contract will hold for target code of all sub-ASTs

A simple strategy: push the value to the stack

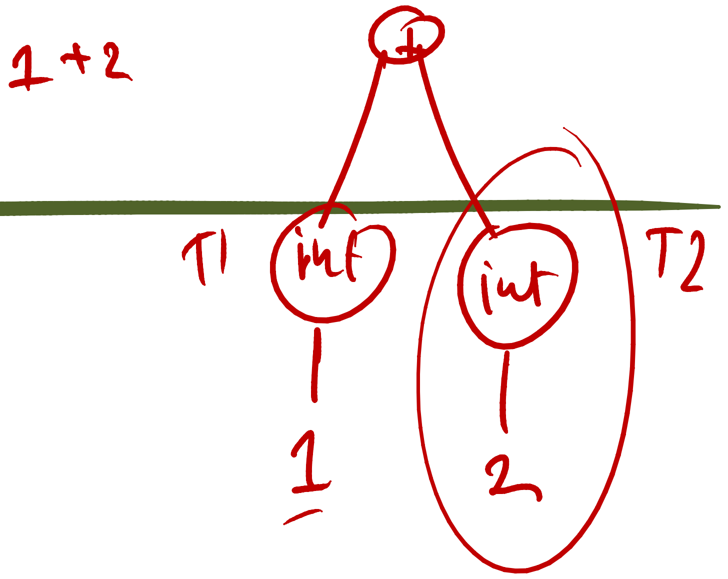
we will call it data stack, called *ds*

The target code *T* for AST *E* composes T1 and T2:

```
<T1 code>
<T2 code>
```

$$\begin{tikzcd}\texttt{ds.push(} & \underbrace{\texttt{ds.pop()}}_{E_2} \arrow[rr, bend left, leftrightarrow] & \texttt{+} & \underbrace{\texttt{ds.pop()}}_{E_1} & \texttt{)}\end{tikzcd}$$

Sample compiler



```
// compile: AST -> String

function compile(n) {
    switch (n.op) {
    case "int": return "ds.push(" + n.arg1 + ")\n"
    case "+":   return compile(n.arg1) \
                     + compile(n.arg2) \
                     + "ds.push( ds.pop() + ds.pop() )\n"
    ...
    }
}
```

ds.push(1) ⟨T1⟩
ds.push(2) ⟨T2⟩
ds.push(ds.pop + ds.pop)

Summary of key concepts

- Compiler: a non-standard interpreter that evaluates the input AST into a target code
- Target code, when run, evaluates the input AST
- Compilation defers evaluation; strips some interpreter overhead; and translates program from source language to target language
- Target code may keep live values in a data stack. This is slow. We'll use "registers" in PA2
- Target code itself can do non-standard evaluation of the input AST, eg draw a picture of this AST

Test yourself

- Write an AST pass that emits d3 code that draws the original AST

Adding functions and local variables

We now add functions

Functions will have parameters and local variables

$\boxed{x()()}\underline{(1+2)}$

Our core language:

```
E := n  |  E+E  |  E-E  |  E/E  |  E*E
    |  function(id*) { E }     // the (anonym) function value
    |  E(E*)                   // function application (call)
    |  id                      // an identifier (var name)
    |  let id = E in E         // let binding (var declaration)
```

init val

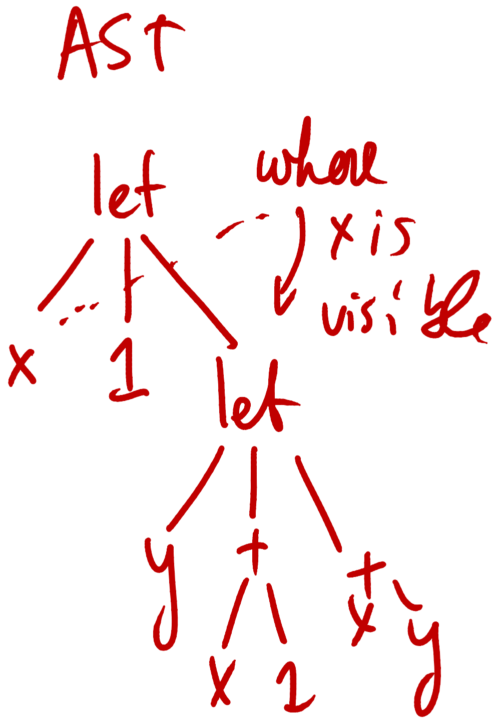
scope of id

visible

Now we desugar a JS-like surface language

Variable definitions

```
var x = 1                          let x = 1 in
var y = x + 1          --->        let y = x + 1 in
x+y                                x + y
```



Named function definitions

```
function f(x) { body }     --->     var f = function(x) { body }
```

How to look up values of variables?

Environment: maps symbols (var names) to values

- think of it as a list of (symbol, value) pairs
- env is passed to interpreter so that it can look up values
- env.lookup(sym) finds returns value of first sym in env

```
function eval(n, env) {
       switch (n.op) {
       case “int”: return n.arg1
       case “id”:  return env.lookup(n.arg1)
       case “+”:   return eval(n.arg1, env) + eval(n.arg2, env)
       case “function”:  // function (ID:arg1) { B:arg2 }
                         return new FunctionObj(ID, B)
       …
}}
```

How to look up values of variables

```
function eval(n, env) {
        switch (n.op) {
        ...
        case “let”:    var init_v  = eval(n.arg2, env)
                       van new_env = prepend(env, (n.arg1, init_v))
                       return eval(n.arg3, new_env)
              call
        case “apply”:  var fun = eval(n.arg1, env)
                       var arg = eval(n.arg2, env)
                       var new_env = prepend(env, (fun.p.name, arg))
                       return eval(fun.body, new_env)
}}
```

$E(E)$

Example

```
let x = 1 in

let f = function (a) { a+1 }  in

f(x+1)
```

Summary of key concepts (1)

- environment: maps symbols to values
- … can be thought of as (ordered) list of pairs

- … *can* be implemented as stack of frames, as in C
- Locals and parameters are stored in a frame
- Frames are pushed onto the call stack at call time
- … popped when the function returns
- … also pushed when entering body of let (a block scope in C)

Summary of key concepts (2)

- Passing by value: evaluate args, then apply function

- Function is an “ordinary” value bound to a symbol
- Such value can be “passed round”, they are first-class values
- It can also be passed around (more on this later)

Test yourself

- In C, returning the address of a local from a function may lead to what bugs problems?
- In our current core language, what constructs are not first-class value? A: symbols (names of vars). We don’t have a way to say “let x = name(y)”.

Implementing If and While via desugaring

Defining control structures

They change the flow of the program

- if (E) S else S
- while (E) S
- while (E_1) S finally E_2 // E_2 executes even when we break

There are many more control structures

- exceptions
- coroutines
- continuations
- event handlers

Assume we are given a built-in conditional

Meaning of ite(E_1, E_2, E_3)

evaluate all three expressions, denote their values v1, v2, v3

if v1 == true then evaluate to v2,

else evaluate to v3

Why is this factorial program incorrect?

```
def fact(n) {
   ite(n<1, 1, n*fact(n-1))
}
```

Ifelse

Can we implement ifelse with just functions?

```
def fact(n) {
  def true_branch() { 1 }
  def false_branch() { n * fact(n-1) }
  ifelse (n<2, true_branch, false_branch)
}

def ifelse (e, th, el) {
  x = ite(e, th, el)
  x()
}
```

Same with anonymous functions

```
def fact(n) {
    if (n<2, function() { 1 }
           , function() { n*fact(n-1) } )
}
```

If

How to define if on top of if-else?

```
def if(e,th) {
    ~~cond~~(e, th, lambda(){} )()
     ite
}
```

While

Can we develop **while** using first-class functions?

Let's desugar While (E) { E } to functions

```
var count = 5
var fact = 1
While (count > 0) {
  count = count - 1
  fact := fact * count
}
```

While

```
var count = 5
var fact = 1
while( lambda() { count > 0 },
       lambda() {
              count = count - 1
              fact := fact * count }
)
while (e, body) {
    x = e()
    if (x, body)
    if (x, while(e, body))
}
```

Tail Call Optimiz

If, while

Desugaring while to

```
while(lambda() { x < 10 } ,
        lambda() {
              loopBody
        })
```

may seem ugly, but jQuery, etc, pass functions, too

```
$(".-123").hover(
              function(){ $(".-123").css("color", "red"); },
              function(){ $(".-123").css("color", "black"); }
          );
```

Smalltalk actually use this model

Control structures not part of the core language

Made visually acceptable by special syntax for *blocks*

which are (almost) like anonymous functions

Smalltalk:

```
| count factorial |
count := 5.
factorial := 1.
// pass two blocks to function whileTrue
[ count > 0 ] whileTrue:
   [ factorial := factorial * (count := count - 1) ]
Transcript show: factorial
```

Almost the same in Ruby

```
count = 5
fact = 1
while count > 0 do
    count = count - 1
    fact = fact * 1
end
```

not a lambda

"block" ≈ lambda

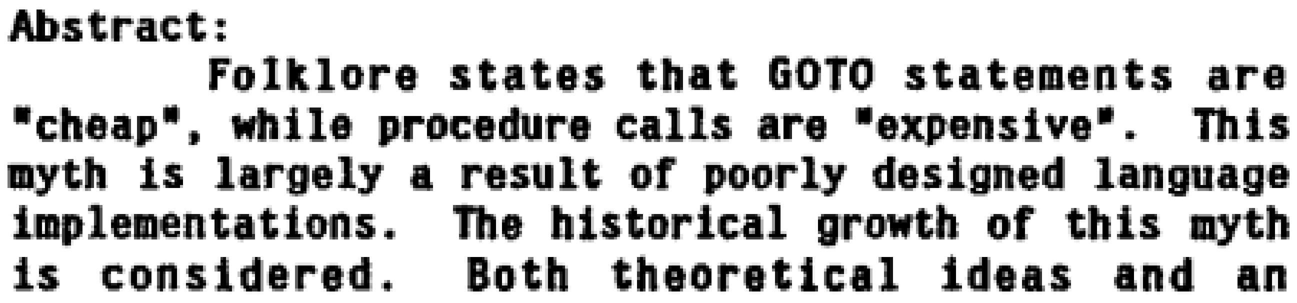
For efficiency, the conditional is not a lambda.

But the body remains a block, as in Smalltalk.

Also see

If this is still strange, observe that calls are just goto’s.

Lambda: The Ultimate GOTO, Guy Lewis Steele, Jr.


Abstract:
Folklore states that GOTO statements are "cheap", while procedure calls are "expensive". This myth is largely a result of poorly designed language implementations. The historical growth of this myth is considered. Both theoretical ideas and an

Syntactic sugar

We provide a more readable syntax

```
While (E) { S }
```

and desugar this ‘surface’ construct to

```
while(lambda() { E } , lambda() { S })
```

Summary of key concepts

- <u>Control construct</u>: decides which part of the program is evaluated next (“where the PC moves”).
- If, while and other control constructs can be desugared to the core constructs, namely function definition and function application
- Parts of if/while are wrapped in lambdas so that we can control when to execute them
- It is necessary that the wrapping function can access data not passed via its parameters, ie data from outer scopes

Test yourself

- Desugar for to while
- Once we have lists in our language, desugar list comprehension to while

Code reuse with high-order functions

Looping constructs in DSLs

d3’s selection.each(f) is also a high-order function

so are the classical list functions such as map and fold

more on these in later lectures

Summary of key concepts

- High-order function accepts a function as argument
- Map, fold, etc allow reuse by accepting a work fun
- Passed functions may be also packed in an object
- This allows syntactically pleasing call chaining

Test yourself

- Write HO functions `lfold` and `rfold` over lists
- Design and implement call chaining for some common task (TBD)

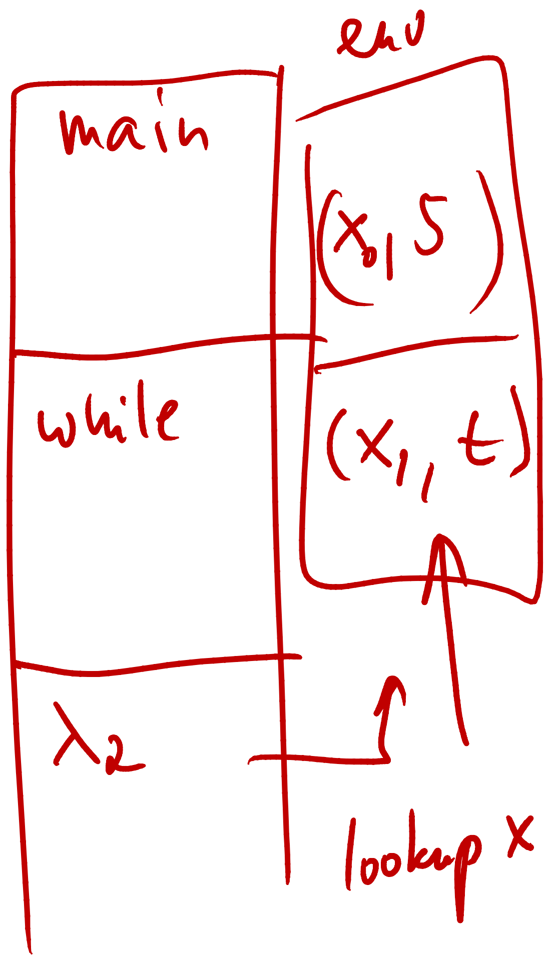
Dynamic scoping

Now let's put our language to a test

```
var x0 = 5          // rename count to x
var fact = 1
while( lambda() { x > 0 },
       lambda2() {
         x = x - 1
         fact := fact * x }
)
while (e, body) {
    x1 = e()
    if (x, body)
    if (x, while(e, body))
}
```

want

we get



Dynamic scoping

The nesting of scopes (and thus symbols are looked up) is determined by the call stack.

It’s called dynamic because the nesting is determined by how functions call each other at run-time.

that is, it is input dependent

we want scoping that is more predictable, that is known when program is written

that is, one that depends on the structure of the program

Summary of key concepts

- Our model of environment, which grows with new symbols at each call and let-in, gives rise to dynamic scoping.
- Dynamic scoping = lookup variables following the calls stack.
- Problem: implementations (eg of while) that should be invisible to work functions may interfere
- That is, dynamic scoping is non-modular

Test yourself

- Find a language with dynamic scoping
- Study its tutorial and learn what dynamic scoping is useful for
- Efficiency of name lookup in dynamic scoping: can you think of a constant-time algorithm for finding variable x in the environment?

Static (aka lexical) Scoping, Closures

Static scoping

Same as in dynamic scoping:

```
function eval(n, env) {
       switch (n.op) {
       case “int”: return n.arg1
       case “id”:  return env.lookup(n.arg1)
       case “+”:   return eval(n.arg1, env) + eval(n.arg2, env)
       case “let”: // same
       …
}}
```

How to look up values of variables

```
function eval(n, env) {
       switch (n.op) {
       ...
       case “function”:  // function (ID:arg1) { B:arg2 }
                     var f = new Function(ID, B)
                     return (f, env) // closure (fun + its env)

       case “apply”:  var fun = eval(n.arg1, env)
                     var arg = eval(n.arg2, env)
                     var new_env = prepend((fun.par, arg),fun.env)
                     return eval(fun.body, new_env)
}}
```

Closures

Closure: a pair (function, environment)

this is our final representation of "function value"

function:

- it's first-class function, ie a value, ie we can pass it around
- representation keeps know the code of body & params
- may have free variables, these are bound using the env

environment:

- it's the environment from when the function was created

Application of closures

From the Lua book

```
names = { "Peter", "Paul", "Mary" }
grades = { Mary: 10, Paul: 7, Paul: 8 }
sort(names, function(n1,n2) {
      grades[n1] > grades[n2]
}
```

Sorts the list names based on grades.

- grades not passed to sort via parameters but via closure

A cool closure

```
c = derivative(sin, 0.001)
print(cos(10), c(10))
   --> -0.83907, -0.83907

def derivative(f,delta)
    function(x) {
        (f(x+delta) - f(x))/delta
    }
}
```

Summary of key concepts

- Idea: allow nested functions + allow access only to nonlocals in *parent* (ie statically outer) functions
- The environment: frames on the parent chain
- Name resolution for x: first x from on parent chain
- Solves modularity problems of dynamic scoping
- Functions are now represented as closures, a pair of (function code, function environment)
- Frames created for a function's locals survive after the function returns
- This allows creating data on the heap, accessed via functions (eg our counting closure)

Test yourself

Fun Facts

The curse of hasty decisions

A language design decision: introducing a var

Choice 1: explicit introduction (Algol, …, JavaScript)

```
def f(x) {
  # Define 'a'. This is binding instance of a.
  var a = x+1
  return a*a
}
```

Choice 2: implicit introduction (BASIC, …, Python)

```
def f(x) {
  a = x+1       # the assignment a=… effectively
  return a*a    # inserts definition var a into f's scope
}
```

Rewrite this code from JS to Python 2.7

```
function getCounter() {
    var a = 0
    function counter() {
        a = a + 1
        return a
    }
    return counter  // returns a counting closure
}
var c1 = getCounter()
var c2 = getCounter()
console.log(c2())            --> 1
console.log(c2())            --> 2
console.log(c1())            --> 1
```

In Python

An incorrect attempt in Python:

```
def getCounter():
    a = 1
    def counter():
        a = a + 1 <-- Error: local variable 'a'
        return a   referenced before assignment
    return bar
c = getCounter()
print(c())
print(c())
```

Why are we getting this error?

```
def foo():
    a = 1
    def bar():
        a = a + 1
        return a
    return bar
```

Python rule: “If a name binding operation occurs anywhere within a code block, all uses of the name within the block are treated as references to the current block['s binding].”

Fix in Python 3, a new version of language

```
def foo():
    a = 1
    def bar():
        nonlocal a
        a = a + 1
        return a
    return bar
f = foo()
```

It works but it’s still a mess: where do we bind a?

in two places, rather than one (a=1 and nonlocal a)

Summary of key concepts

- Decision on how to introduce a symbol seems innocuous but has ramifications
- Often simplifications like implicit binding work when the language is simple, eg has no nested functions
- But once the language grows, surprising interactions between language features (implicit binding and nested functions) may happen

Test yourself

Summary of lecture

- Functions and variables suffice to build a big language
- Desugaring bridges what’s convenient to the programmer with what’s convenient to the interpreter/compiler

Reading

Required:

none

Recommended:

none

Fun:

lambda the ultimate GOTO

Review:

lexical scoping, environments, and frames: Ch 3.2 in CS61A