Make Your Language!

CSE401 Winter 2016
Introduction to Compiler Construction

Ras Bodik
Alvin Cheung
Maaz Ahmad
Talia Ringer
Ben Tebbs

Lecture 2: Interpreters

Unit calculator
Dynamic Scoping
Desugaring

Ground rules

No laptops and phones in the classroom.

Exception: taking notes.

- You must email the notes to us after class
- Please sit in the back of the classroom

If lecture pace is slow:

- ask us for challenge problems

More administrative stuff

- HW1 and PA1 posted on website
  - Use your CSE ID to access documents
  - HW1 will be due this Sunday
- Late day policy:
  - **PA:** up to three late days, 15% penalty for each day
  - **HW:** no late days
- Sections will start tomorrow!
  - Please bring your laptop
- Office hours have started this week
  - See website for details

Compilers for 21st century

CSE 401 Course description

Fundamentals of compilers and interpreters; symbol tables; lexical analysis, syntax analysis, semantic analysis, code generation, and optimizations for general-purpose ^ programming languages.

and domain-specific

Also adding:

Design of programming abstractions for modern programming challenges (internet, parallelism)

★

Unit calculator

an interpreter with interesting “types”

The calculator section includes slides not covered in the lecture. Read them to learn about the design process of such a language. Slides covered in class have a star.

Compilers for 21st century

CSE 401 Course description

Fundamentals of compilers and interpreters; symbol tables; lexical analysis, syntax analysis, semantic analysis, code generation, and optimizations for general-purpose ^ programming languages.

and domain-specific

Also adding:

Design of programming abstractions for modern programming challenges (internet, parallelism)

Today

Programs, values and types

- In today's lecture, programs are expressions
- Expressions compute values
- Types are properties of values

How to build an interpreter

- representation of values and types in the interpreter

Finding errors in incorrect programs

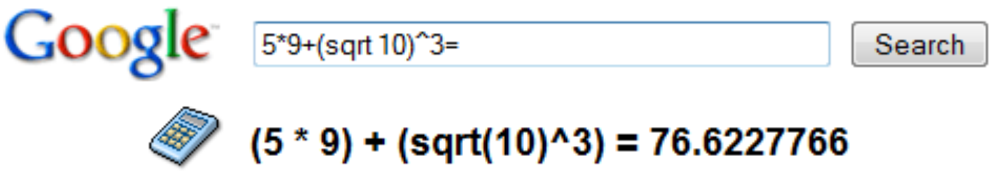
- when do we catch the error? Parsing or execution?

Two languages:

- a unit calculator and a simple language

Recall Lecture 1

Your boss asks: *"Could our search box answers some semantic questions?"* You build a calculator:

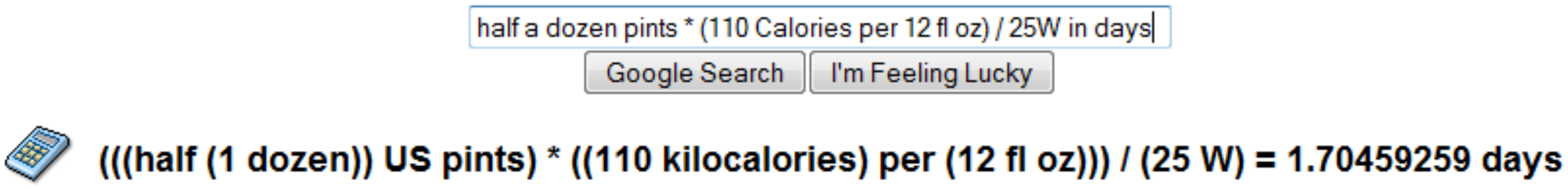


5*9+(sqrt 10)^3=

(5 * 9) + (sqrt(10)^3) = 76.6227766

Then you remember cse401 and easily add unit conversion.

How long a brain could function on 6 beers --- if alcohol energy was not converted to fat.



half a dozen pints * (110 Calories per 12 fl oz) / 25W in days

(((half (1 dozen)) US pints) * ((110 kilocalories) per (12 fl oz))) / (25 W) = 1.70459259 days

Interpreter for our calculator language

speed of a high-speed ferry

`37 knots in mph`

--> 42.5788 mph

time to power your brain on 6 beers

half a dozen pints * (110 Calories per 12 fl oz) / 25 W in days

--> 1.704 days

Let’s pretend we are designing this language

What do we want from the language?

- evaluate arithmetic expressions
- … including those with physical units
- check if operations are legal (area + volume is not)
- convert units

Constructs of the Calculator Language

Numbers:

- ints
- Floats

Units (Modeled as types):

- Some are canonical (SI units)
- Some are not (imperial)

Operators:

- + - * / per
- ( )

Addn’l features we will actually implement

- allow users to extend the language with their units
- … with new measures (eg Ampere)
- bind names to values

We’ll grow the language one feature at a time

1. Arithmetic expressions
2. Physical units for (SI only)
3. Non-SI units
4. Explicit unit conversion

Let’s start with the sublanguage of arithmetic

A programming language is defined with

Syntax: structure of valid programs

2 + 3 legal

\+ 2 3 illegal

given by a grammar (see next slide)

Semantics: to what values the program evaluates

E_1+E_2 evaluates to the sum of the evaluation of E_1 and the evaluation of E_2.

Can be written as $\llbracket E_1 + E_2 \rrbracket = \llbracket E_1 \rrbracket +_{32} \llbracket E_2 \rrbracket$

We’ll define it by writing an interpreter.

Syntax

The set of syntactically valid programs is infin. large. So we define it recursively:

E ::= n | E op E | (E)
op ::= + | - | * | / | ^

E is set of all expressions expressible in the language.
n is a number (integer or a float constant)

Examples: 1, 2, 3, … , 1+1, 1+2, 1+3, … , (1+3)*2, …

Semantics (Meaning)

Syntax defines what our programs look like:

- 1, 0.01, 0.12131, 2, 3, 1+2, 1+3, (1+3)*2, …

But what do they mean? Let's try to define $e_1 + e_2$

- Given the values e_1 and e_2,
- the value of $e_1 + e_2$ is the sum of the two values.

We need to state more. What is the range of ints?

- Is it $0..2^{32}-1$?
- Our calculator borrows Python's unlimited-range integers

How about if e_1 or e_2 is a float?

- Then the result is a float.

There are more subtleties, as we'll discover shortly.

How to represent a program?

concrete syntax (input program, flat)	**abstract syntax** (internal program representation, tree)
`1+2`	`(‘+’, 1, 2)`
`(3+4)*2`	`(‘*’, (‘+’, 3, 4), 2)`

Conversion done by parser guided by a grammar

(writing a correct grammar can be tricky, so we’ll skip it today)

Tricky parsing examples:

2 / m / s	is this (2/m)/s or 2/(m/s) ?
in in in	means 1 inch in inches
in in in in	means 1 inch^4

The interpreter

Recursive descent over the abstract syntax tree

```
ast = ('*', ('+', 3, 4), 5.1)
print(eval(ast))

def eval(e):
    if type(e) == type(1): return e
    if type(e) == type(1.1): return e
    if type(e) == type(()):
        if e[0] == '+': return eval(e[1]) + eval(e[2])
        if e[0] == '-': return eval(e[1]) - eval(e[2])
        if e[0] == '*': return eval(e[1]) * eval(e[2])
        if e[0] == '/': return eval(e[1]) / eval(e[2])
        if e[0] == '^': return eval(e[1]) ** eval(e[2])
```

How we’ll grow the language

1. Arithmetic expressions ✓
2. Physical units for (SI only)
3. Non-SI units
4. Explicit unit conversion

Let’s grow the langauge with SI units

Example:

(2 m) ^ 2 --> 4 m^2

Concrete syntax:

E ::= n | **U** | E op E | (E)
U ::= m | s | kg
op ::= + | - | * | 🎁 | / | ^

Abstract syntax: represent SI units as string constants

`3 m^2` `('*', 3, ('^', 'm', 2))`

A question: catching illegal programs

Our language now allows us to write illegal programs.

Examples: 1 + m, 2ft – 3kg.

Question: Where should we catch such errors?

a) in the parser (as we create the AST)

b) during the evaluation of the AST

c) parser and evaluator will cooperate to catch this bug

d) these bugs cannot generally (ie, all) be caught

Answer:

b: parser has only a local (ie, node and its children) view of the AST, hence cannot tell if ((m))+(kg) is legal or not.

Representing values of units

How to represent the value of `('^', 'm', 2)` ?

A pair (numeric value, Unit)

Unit is a map from an SI unit to its exponent:

```
('^', 'm', 2)                → (1, {'m':2})
('*', 3, ('^', 'm', 2))  → (3, {'m':2})
```

The interpreter

```
def eval(e):
   if type(e) == type(1):   return (e,{})
   if type(e) == type('m'): return (1,{e:1})
   if type(e) == type(()):
        if e[0] == '+': return add(eval(e[1]), eval(e[2]))
        …
def sub((n1,u1), (n2,u2)):
   if u1 != u2: raise Exception(“Subtracting incompatible units")
   return (n1-n2,u1)
def mul((n1,u1), (n2,u2)):
   return (n1*n2,mulUnits(u1,u2))
```

Read rest of code at:

http://bitbucket.org/bodik/cs164fa09/src/9d975a5e8743/L3-ConversionCalculator/Prep-for-lecture/ConversionCalculator.py

How we'll grow the language

1. Arithmetic expressions
2. Physical units for (SI only) ✓ [code](link) (link)
3. Non-SI units
4. Explicit unit conversion

You are expected to read the code

It will prepare you for PA2

Step 3: add non-SI units

Trivial extension to the syntax

- E ::= n | U | E op E | (E)
- U ::= m | s | kg | **ft** | **year** | …

But how do we extend the interpreter?

- We will evaluate ft to 0.3048 m.
- This effectively converts ft to m at the leaves of the AST.

We are canonicalizing non-SI values to their SI unit

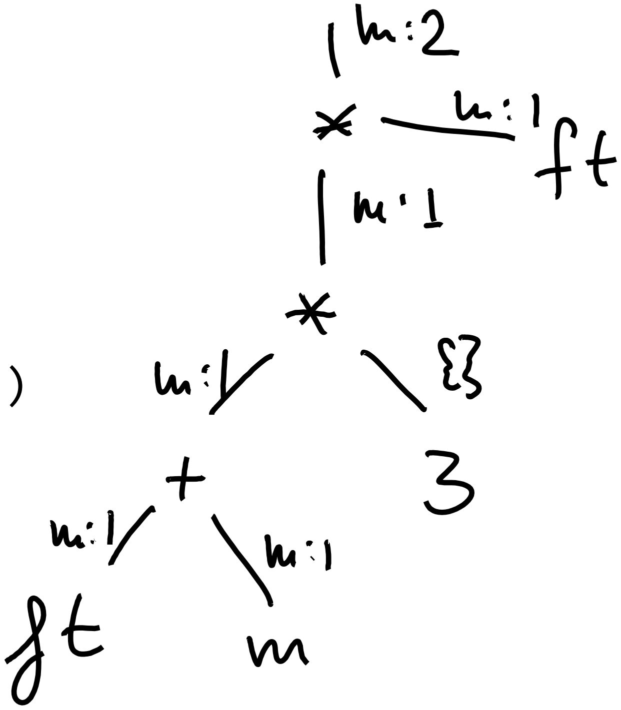
- SI units are the "normalized type" of our values

Adding non-SI units

Now we want to evaluate **`(ft +m)*3*ft`**

```
def eval(e):
    if type(e) == type(1):   return (e,{})
    if type(e) == type(1.1): return (e,{})
    if type(e) == type('m'): return lookupUnit(e)

def lookupUnit(u):
    return {
        'm'      : (1,      {'m':1}),
        'ft'     : (0.3048, {'m':1}),
        's'      : (1,        {'s':1}),
        'year'   : (31556926, {'s':1}),
        'kg'     : (1,          {'kg':1}),
        'lb'     : (0.45359237, {'kg':1})
        }[u];
```



how values are propagated up the tree

Rest of code at : http://bitbucket.org/bodik/cs164fa09/src/c73c51cfce36/L3-ConversionCalculator/Prep-for-lecture/ConversionCalculator.py

How we’ll grow the language

1. Arithmetic expressions
2. Physical units for (SI only) code (link) 44LOC
3. Add non-SI units code (link) 56LOC
   3.5 Revisit integer semantics (a coersion bug)
4. Explicit unit conversion

Coercion revisited

4/2

To what should "1 m / year" evaluate?

- our interpreter outputs 0 m / s
- problem: value 1 / 31556926 * m / s was rounded to zero

Because we naively adopted Python coercion rules

- They are not suitable for our calculator.

We need to define and implement our own.

- Keep a value in integer type whenever possible. Convert to float only when precision would otherwise be lost.

Read the code: explains when int/int is an int vs a float

http://bitbucket.org/bodik/cs164fa09/src/204441df23c1/L3-ConversionCalculator/Prep-for-lecture/ConversionCalculator.py

How we’ll grow the language

1. Arithmetic expressions
2. Physical units for (SI only) code (link) 44LOC
3. Add non-SI units code (link) 56LOC
   3.5 Revisit integer semantics (a coersion bug) code (link) 64LOC
4. Explicit unit conversion

Explicit conversion

Example:

`3 ft/s` **`in`** `m/year` --> *28 855 653.1 m / year*

The language of the previous step:

```
E  ::= n | U | E op E | (E)
U  ::= m | s | kg | J | ft | in | …
op ::= + | - | * | ε | / | ^
```

Let's extend this language with “E in C”

Unit conversion

Where in the program can "E in C" appear?

Attempt 1:

```
E ::= n | U | E op E | (E)  | E in C
```

That is, is the construct "E in C" a kind of expression?

If yes, we must allow it wherever expressions appear.

For example in (2 m in ft) + 3 km.

For that, E in C must yield a value. Is that what we want?

Attempt 2:

```
P ::= E | E in C
E ::= n | U | E op E | (E)
```

"E in C" is a top-level construct.

It decides how the value of E is printed.

Next, what are the valid forms of C?

Attempt 1:

```
C  ::= U op U
U  ::= m | s | kg | ft | J | …
op ::= + | - | * | ε | / | ^
```

2 ft in m-mm ?
NO

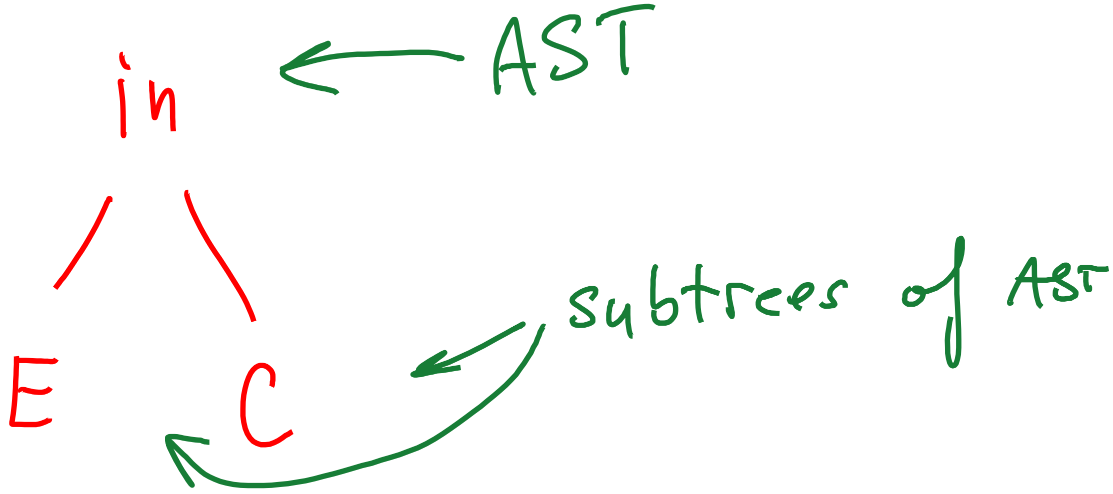
Examples of valid programs:

Attempt 2:

```
C ::= C * C  |  C C  |  C / C  |  C ^ n  |  U
U ::= m | s | kg | ft | J | …
```

How to evaluate C?

Our ideas:



what's the "value" of C?

how is it represented?

we would like to evaluate C with the same function a E. But this seems impossible

How to evaluate C?

What value(s) do we need to obtain from sub-AST C?

1. conversion ratio between the unit C and its SI unit

   ex: (ft/year)/(m/s) = **$9.65873546 \times 10^{-9}$**

2. a representation of C, for printing

   ex: ft * m * ft --> {ft:2, m:1}

How we'll grow the language

1. Arithmetic expressions
2. Physical units for (SI only) code 44LOC
3. Add non-SI units code 56LOC

   3.5 Revisit integer semantics (a coersion bug)

   code 64LOC
4. Explicit unit conversion code 78LOC

   this step also includes a simple parser: code 120LOC

You are asked to understand the code.

you will understand the parser code in later chapters

Where are we?

The grammar:

```
P  ::= E | E in C
E  ::= n | E op E | ( E ) | U
op ::= + | - | * | ε | / | ^
U  ::= m | s | kg  | ft | cup | acre | l | …
C  ::= U | C * C | C C | C/C | C^n
```

After adding a few more units, we have google calc:

`34 knots in mph` *--> 39.126 mph*

What you need to know

- Understand the code of the calculator
- Able to read grammars (descriptors of languages)

Key concepts

programs, expressions

are parsed into abstract syntax trees (ASTs)

values

are the results of evaluating the program,

in our case by traversing the AST bottom up

types

are auxiliary info (optionally) propagated with values during evaluation; we modeled physical units as types

Part 2

Grow the calculator language some more.

Allow the user to

- add own units
- reuse expressions

Review of progress so far

Example:

`34 knots in mph` # speed of S.F. ferry boat

--> 39.126 mph

Example: # volume * (energy / volume) / power = time

half a dozen pints * (110 Calories per 12 fl oz) / 25 W in days

--> 1.704 days

Now we will change the language to be extensible

How we’ll grow the language

1. Arithmetic expressions
2. Physical units for (SI only) code 44LOC
3. Add non-SI units code 56LOC
4. Explicit unit conversion code 78LOC

   this step also includes a simple parser: code 120LOC

5. Allowing users to add custom non-SI units

Growing language w/out interpreter changes

We want to design the language to be extensible

- Without changes to the base language
- And thus without changes to the interpreter

For calc, we want the user to add new units

- Assume the language knows about meters (feet, …)
- Users may wan to add, say, Angstrom and light year

How do we make the language extensible?

Our ideas

```
minute = 60 s
yard = 36 inch
```

existing units

units just introduced

Bind a value to an identifier

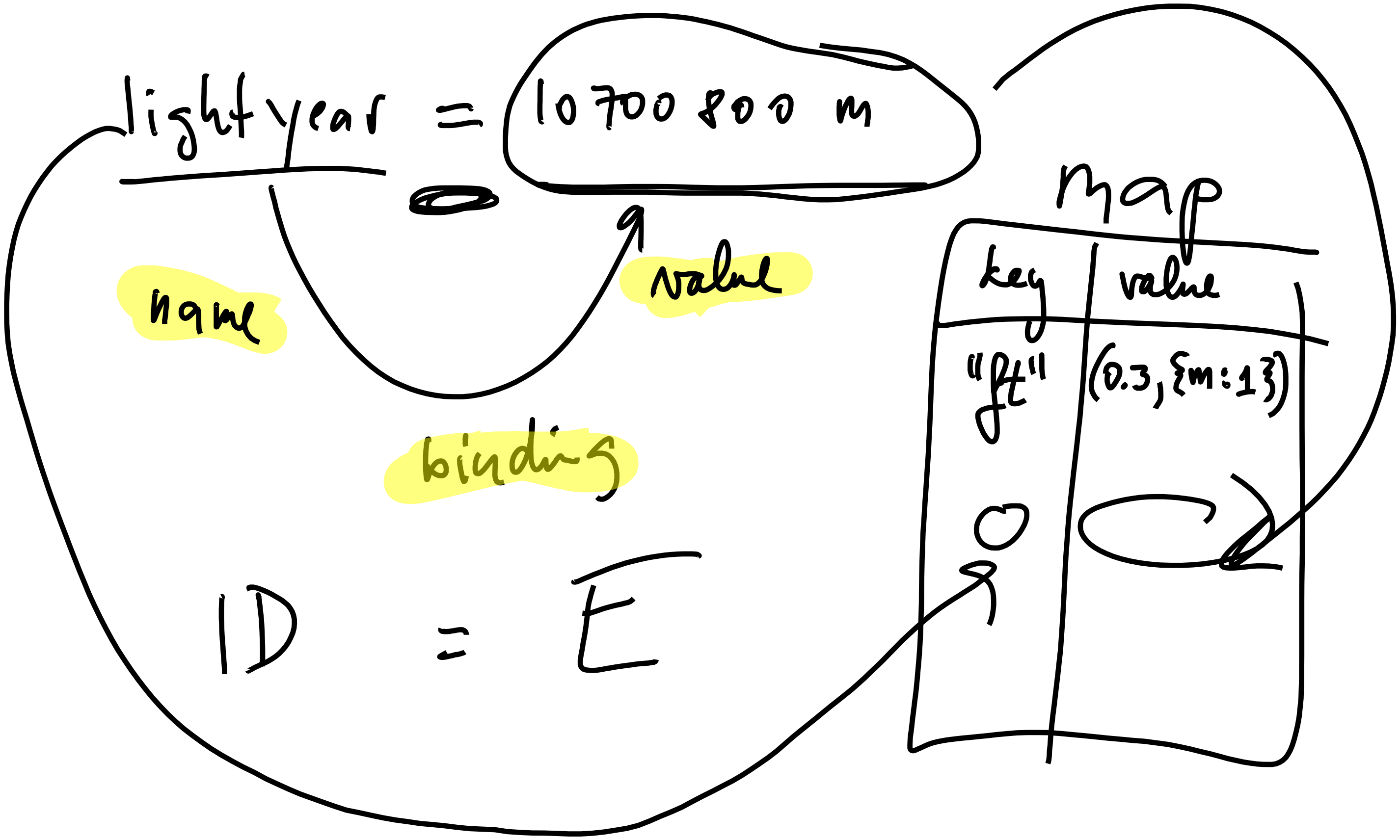
```
minute = 60 s
hour = 60 minute
day = 24 hour
month = 30.5 day   // maybe not define month?
year = 365 day
km = 1000 m
inch = 0.0254 m
yard = 36 inch
acre = 4840 yard^2
hectare = (100 m)^2
```

`2 acres in hectare` → ***0.809371284 hectare***

Implementing user units

Assume units extends existing measures.

- We want the user to add **ft** when **m** or **yard** is known



How we’ll grow the language

1. Arithmetic expressions
2. Physical units for (SI only) code 44LOC
3. Add non-SI units code 56LOC
4. Explicit unit conversion code 78LOC

   this step also includes a simple parser: code 120LOC

5. Allowing users to add custom non-SI units ✓
6. Allowing users to add custom measures

How do we add new measures?

No problem for Joule, as long you have kg, m, s:

J = kg m^2 / s^2

But other units must be defined from first principles:

Electric current:

- Ampere

Currency:

- USD, EUR, YEN, with BigMac as the SI unit

Coolness:

- DanGarcias, with Fonzie as the SI unit

Our ideas

Attempt 1:

when we evaluate `a = 10 b` and b is not known, add it as a new SI unit.

This may lead to spuriously SI units introduced due to typos.

Attempt 2:

ask the user to explicitly declare the new SI unit:

```
SI Ampere
```

Introduce your own SI units

Add into language a construct introducing an SI unit

```
SI A                         // Ampere
mA = 0.0001 A
SI BigMac
USD = BigMac / 3.57          // BigMac = $3.57
GBP = BigMac / 2.29          // BigMac = £2.29
```

With “SI <id>”, language needs no built-in SI units

```
SI m
km = 1000 m
inch = 0.0254 m
yard = 36 inch
```

Implementing `SI id`

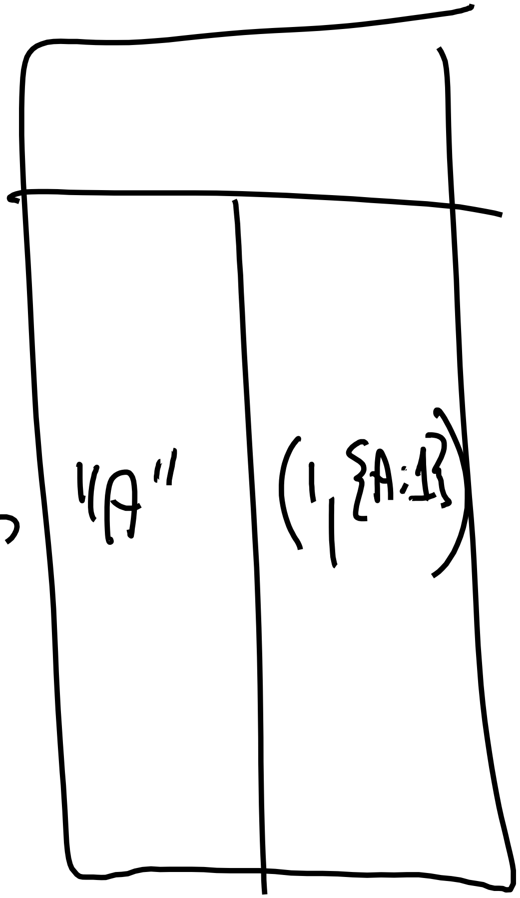
Problem

$mA = \boxed{A} / 1000$

Solve

declaration: SI␣A

$mA = 0.001\ A$



How we'll grow the language

1. Arithmetic expressions
2. Physical units for (SI only) code 44LOC
3. Add non-SI units code 56LOC
4. Explicit unit conversion code 78LOC

   this step also includes a simple parser: code 120LOC

5. Allowing users to add custom non-SI units
6. Allowing users to add custom measures code ✓
7. Reuse of values

Closing example

Compute # of PowerBars burnt on a 0.5 hour-long run

```
SI m, kg, s
lb = 0.454 kg;  N = kg m / s^2
J = N m;  cal = 4.184 J
powerbar = 250 cal

0.5hr * 170lb * (0.00379 m^2/s^3) in powerbar
  --> 0.50291 powerbar
```

Want to retype the formula after each morning run?

0.5 hr * 170 lb * (0.00379 m^2/s^3)

Reuse of values

To avoid typing

```
170 lb * (0.00379 m^2/s^3)
```

… we’ll use same solution as for introducing units:

Just name the value with an identifier.

```
c = 170 lb * (0.00379 m^2/s^3)
28 min * c
# … next morning
1.1 hour * c
```

Should time given be in min or hours?

Either. Check this out! Calculator converts automatically!

How we'll grow the language

1. Arithmetic expressions
2. Physical units for (SI only) code 44LOC
3. Add non-SI units code 56LOC
4. Explicit unit conversion code 78LOC

    this step also includes a simple parser: code 120LOC

5. Allowing users to add custom non-SI units
6. Allowing users to add custom measures code
7. Reuse of values (no new code needed) ✓

How we’ll grow the language

1. Arithmetic expressions
2. Physical units for (SI only) code 44LOC
3. Add non-SI units code 56LOC
4. Explicit unit conversion code 78LOC

   this step also includes a simple parser: code 120LOC

5. Allowing users to add custom non-SI units
6. Allowing users to add custom measures code
7. Reuse of values (no new code needed)

Summary: Calculator is an extensible language

Very little knowledge hardcoded in the interpreter

- Introduce all base units with ‘SI name’
- Arithmetic then checks values for compatible types

The user can add her own non-SI units, too

```
cal = 4.184 J
```

Reuse of values by naming the values.

```
myConstant = 170 lb * (0.00379 m^2/s^3)
0.5 hr * myConstant in powerbar
```

-> uses the same mechanism as for non-SI units!

No need to remember units! Both hrs & minutes OK:

```
0.5 hr * myConstant in powerbar
30 minutes * myConstant in powerbar
```

Limitations of calculator

No relational definitions

- We may want to define ft with ‘12 in = ft’
- We could do those with Prolog
  - recall the three colored stamps example in Lecture 1

Limited parser

- Google parses 1/2/m/s/2 as ((1/2) / (m/s)) / 2
- There are two kinds of / operators
- Their parsing gives the / operators intuitive precedence

What you were supposed to learn

Binding names to values

- and how we use this to let the user grow the calculator

Introducing new SI units required a declaration

- the alternative could lead to hard-to-diagnose errors

names can bind to expressions, not only to values

- these expressions are evaluated lazily

A simple programming language

A simple language

Key constructs in the language:

- *first-class* functions (i.e., they can be passed as values)
- definitions of local variables (variable binding)
- objects aside, these are sufficient to build a DSL like d3

The grammar

```
E := n
  |  id                           // reference to a variable
  |  E+E | E-E | E/E | E*E
  |  function (id,…,id) { E }  // (anon.) function value
  |  E(E,…,E)                     // application (call)
  |  var id=E                     // var introduction
```

There are no named functions in the grammar!

We can obtain them by rewriting “named” definitions

```
function foo(x) { body }

    →

var foo = function(x) { body }
```

Let’s simplify this language further

To demonstrate scoping issues, we don’t need

- var id=E : our only vars will be function parameters
- we also don’t need multiple parameters

So, we will develop interpreters for this simple language:

```
E := n
   |  id
   |  E+E | E-E | E/E | E*E
   |  function (id) { E }
   |  E(E)
```

Currying (optional material)

We can create multi-param functions from single-param function by means of currying

```
function f(x,y) { x+y }
f(1,2)

→

function f_y(x) { function (y) { x + y } }
f_y(1)(2)
```

evaluates to function f(1,y)
partially applied

The AST structure

AST nodes

Each grammar rule will have its own kind of AST node

- each node is a struct
- the field op determines the types of node
- other fields link to children ASTs

  such as the two expressions in E(E), which we call `fun` and `arg`
- other fields give attributes

  such as the name of the `id` node or the value of the int literal n

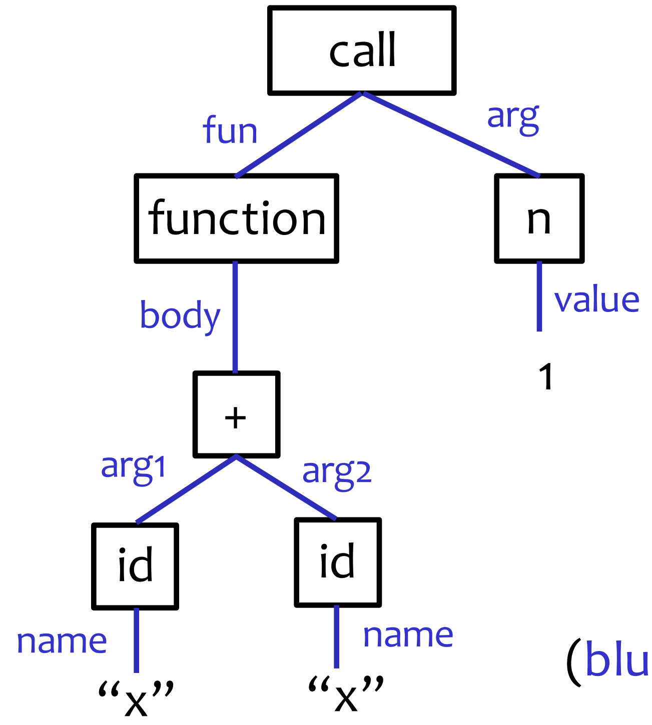
Examples:

E := n	{ op="int", value }
E := id	{ op="id", name }
E := E(E)	{ op="call", fun, arg }
E := E + E	{ op="add", arg1, arg2}

Example AST

Draw the AST for this program:

```
(function(x){ x + x } ) (1)
```


call
fun
arg
function
n
body
value
1
+
arg1
arg2
id
id
name
name
"x"
"x"

(blue indicates node attributes)

The environment

Variables

Our language contains variables

- so we need a method for storing and looking up their value

This will be done by the *environment* (env):

- env is a <u>map</u> from symbols (var names) to var's value.

A variable is added to env when it is introduced

- at this point, we also bind the variable to its initial value

Note: in our language, there is no assignment (id=E)

- Hence the initial value of a var does not change
  - so variables are not assignable, and hence the term "variable" is not quite suitable, but we'll use the term anyway

How to look up values of variables?

Environment: maps symbols (var names) to values

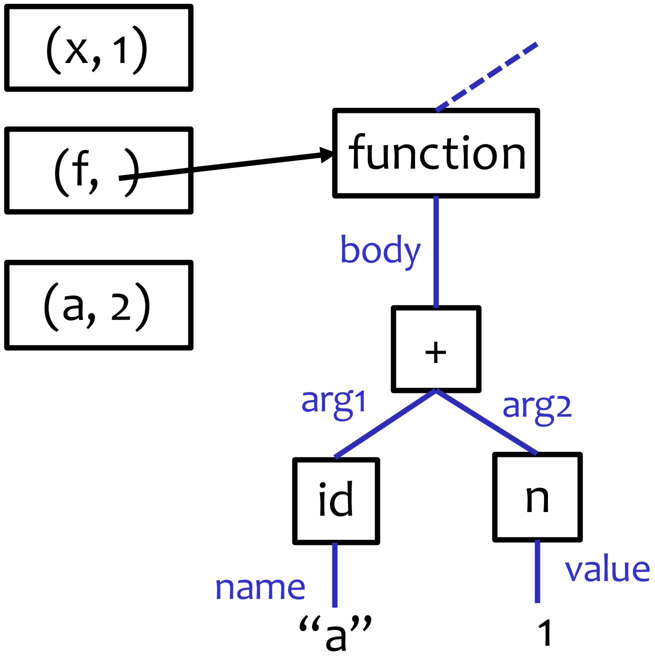
- think of it as a list of (symbol, value) pairs
- env.lookup(sym) returns the value of the *first* sym in env
- the first variable shadows the pairs with the same name

- how env is implemented and what "first" means matters!

Example

Draw the environment (as a list of pairs) for this program when it reaches ♦.

```
var x = 1
var f = function (a) { ♦ a+1 }
f(x+1)
```



(x, 1)

(f,)

(a, 2)

function

body

+

arg1

arg2

id

n

name

“a”

value

1

Desugaring

how to accomplish more with less

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

By the way, there are many more control structures

- exceptions
- coroutines
- continuations
- event handlers

Assume we are given a built-in conditional

Meaning of E=ite(E_1, E_2, E_3)

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

if v1 == true then E evaluates to v2

otherwise E evaluates to v3

Why is this factorial program incorrect?

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

Abstract into a library function

Can we use functions rather than values?

Code wrapped in a function is called a thunk

```
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()
}
```

`_ifelse_` is now a library function provided by the interpreter (notated using "_")

Same but with anonymous functions

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

This is an example of *desugaring*

Rewriting expressions

if (E_1) E_2 E_3

into

ifelse(E_1, function(){E_2},
function(){E_3})

is an example of a rewrite rule

Defining If

How to desugar if into _ifelse_?

```
def if(e,thunk) {
    _ifelse_(e, thunk, function(){} )()
}
```

Let’s abstract this into another library function as well: `_if_(E,thunk)`

What about while?

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

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

Let’s desugar while (E) { E } to function calls

while

```
var count = 5
var fact = 1
_while_( function() { count > 0 },
         function() {
              count = count - 1
              fact = fact * count }
)
def _while_ (e, body) {
    var x = e()
    _if_ (x, body)
    _if_ (x, function () {_while_(e, body)})
}
```

where `_while_(E,thunk)` is yet another library function in the interpreter

Notice that there are multiple ways to desugar while

What if we rename count to x?

```
var x = 5      // rename count to x
var fact = 1
_while_( function() { x > 0 },
         function() {
             x = x - 1
             fact := fact * x }
)
def _while_ (e, body) {
    var x = e()
    _if_ (x, body)
    _if_ (x, function () {_while_(e, body)})
}
```

Scoping

dynamic vs. static

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.lookup(sym) returns the value of the *first* sym in env
- the first variable shadows the pairs with the same name

- how env is implemented and what “first” means matters!

Frames

Implementation:

The (sym,value) pairs created in the same function are usually collapsed into a single *frame*, which is a dictionary mapping syms to values.

A frame has a parent pointer to the frame where the lookup should continue.

Different ways that frames are organized correspond to different scoping rules

Scope

We must define where a variable is visible (its scope)

Dynamic scoping:

Variable is visible globally until the end of its lifetime.

The environment is a **stack**. New bindings are pushed.

The lookup will proceed from the top of stack.

Static scoping:

A function carries its own env (fun+env is called *closure*).

vars defined by different functions are kept separate.

Env is a **tree**; lookup proceeds from a leaf towards the root.

Test yourself

- In C, returning the address of a local from a function may lead to what bugs problems?

Interpreter for dynamic scoping

Scoping and lifetimes

We need to make one more semantic decision:

Will a function parameter disappear when the function returns? Or is it live till the end of the evaluation?

```
(function(x){x})(1)
x+x                 <-- is x still live here?
```

Let's decide to end the parameter's lifetime when the function returns.

so, x+x fails above because x no longer exist at that point

Recall the AST interpreter for calculator

- AST interpreter recursively evaluates the AST
- Typically, values flow bottom-up
- Intermediate values stored on interpreter's stack
- Interpreter performs dynamic type checking

The dynamic-scoping interpreter

```
var  env  =  […]                // env is global; initially an empty stack
function eval(n) {
   switch (n.op) {
   case “int”:        return n.val
   case “id”:         return lookup(env,n.name)
   case “+”:          return eval(n.arg1) + eval(n.arg2)
   ...
   // function (id) { E }
   case “function”:  return { “ast_node”:  n }  // this dict is our fun value
   // E(E)
   case “call”:   var f = eval(n.fun) var a = eval(n.arg)
                 check if f is a function value. If not, exit with error
                 env.push(f.ast_node.param.name, a)
                 var ret = eval(f.ast_node.body)
                 env.pop()    // end the life time of the parameter
                 return ret    }}
```

Problems with dynamic scoping

Dynamic scoping

In dynamic scoping, `env.lookup("x")` returns

the last x added to env

that is still live.

Problem with dynamic scoping:

```
var x=1
hof( function(){ x } )
function hof(callback) {
    var x=2
    callback()
}
```

← 2 is returned!! Why?

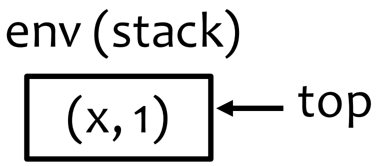
Note: hof is a high-order function:

It accepts other functions as arguments

Dynamic scoping illustration

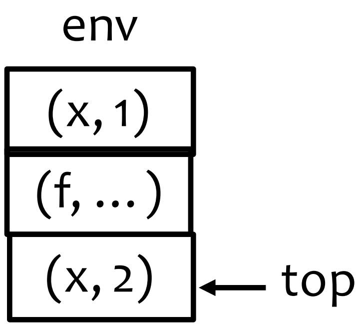
→ current program counter

```
→ var x=1
  hof( function(){ x } )
  function hof(callback) {
          var x=2
          callback()
  }
```


env (stack)

(x, 1) ← top

```
  var x=1
  hof( function(){ x } )
  function hof(callback) {
→         var x=2
          callback()
  }
```

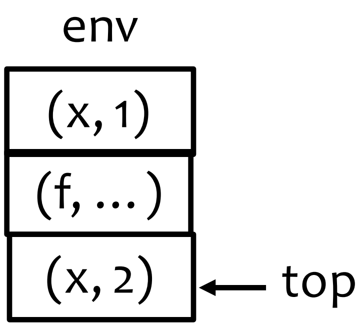

env

(x, 1)

(f, …)

(x, 2) ← top

```
  var x=1
→ hof( function(){ x } )
  function hof(callback) {
          var x=2
          callback()
  }
```


env

(x, 1)

(f, …)

(x, 2) ← top

What value of x is returned by hof?

Learn more

- Find a language with dynamic scoping
- Study its tutorial and find useful applications of dynamic scoping
- Efficiency of name lookup in dynamic scoping:

  Our lookup must traverse the entire stack. Can you think of a constant-time algorithm for finding a variable in env.

Static scoping with closures

Closures

Closure: a pair (function, environment)

this is our final representation of "function value"

the function:

- it's first-class function
  - a value that can be passed around
- Keeps parameter names and the code of the body
- may have free variables
  - these are resolved (looked up) using the env

the environment:

- the environment in which the function was created
- where the function finds vars from its enclosing scope

Application of closures

From the book *Programming in Lua*

```
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 a closure that increments its counter)

The interpreter for static scoping

Interpreter for lexical scoping

Grammar are the same as for dynamic scoping

only the scoping rules change, after all.

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

Static-scoping interpreter

This part is the same as in dynamic scoping

except that env is passed into recursive calls to eval,

which is cleaner than updating the global 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)
       …
}}
```

The lexical-scoping interpreter

```
eval(program, { “parent”: null })  // env with an empty frame

function eval(n, env) {
 switch (n.op) {
 ...
 case “id”:        return env.lookup(n.name)
 case “function”:  // construct and return the closure
                   return { “ast_node”: n, “env”: env }

 case “call”: var f = eval(n.fun, env)
             var a = eval(n.arg, env)
             check if f is a function value. If not, exit with error!
             var new_env = f.env.prepend(f.ast_node.param.name, a)
             return eval(f.ast_node.body, new_env)
             ~~env.pop()~~   // the life time of param does not end here as in dyn.sc.
}}
```