Lecture 2

Unit Conversion Calculator

Expressions, values, types. Their representation and interpretation.

Ras Bodik
Shaon Barman
Thibaud Hottelier

Hack Your Language!

CS164: Introduction to Programming Languages and Compilers, Spring 2012

UC Berkeley

Administrativia

HW1 was assigned Tuesday. Due on Sunday 11:59pm.

- a programming HW, done individually
- you will implement a web mashup with GreaseMonkey

Did you pick up your account forms?

- you can pick them after the lecture from Shaon

Today is back-to-basics Thursday. No laptops.

Exams

Midterm 1: March 6

Midterm 2: April 26 (during last lecture)

The final exam is Posters and Demos: May 11

Course grading

Projects (PA1-9)	45%
Homeworks (HW1-3)	15%
Midterms	20%
Final project	15%
Class participation	5%

Class participation: many alternatives

- ask and answer questions during lectures or recitations,
- discuss topics on the newsgroup,
- post comments to lectures notes (to be added soon)

Summary of last lecture

What's a programming abstraction?

- data types
- operations on them and
- constructs for composing abstractions into bigger ones

Example of small languages that you may design

- all built on abstractions tailored to a domain

What's a language?

- a set of abstractions composable into bigger ones

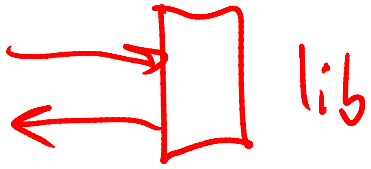
Why small languages?

- see next slide and lecture notes

MapReduce story

Evolution of sw reuse and design

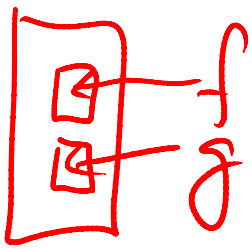
- library



MPI

- framework

parameterizable library

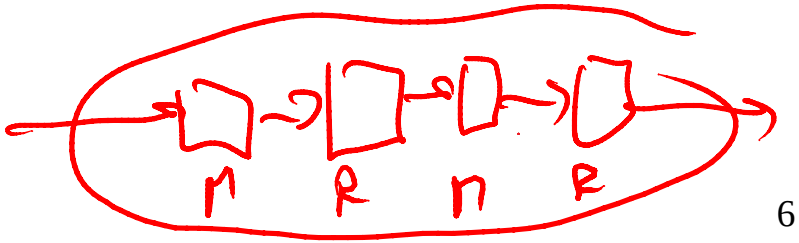


MapReduce

- Small language

composable abstractions

FlumeJava



What's a "true" language

Composable abstractions

not composable:

- networking **socket**: an abstraction but can't build a "bigger" socket from a an existing socket

composable:

- regexes: `foo|bar*` composes regexes `foo` and `bar*`

Today

Programs (expressions), values and types

- their representation in the interpreter
- their evaluation

Finding errors in incorrect programs

- where do we catch the error?

Using unit calculator as our running study

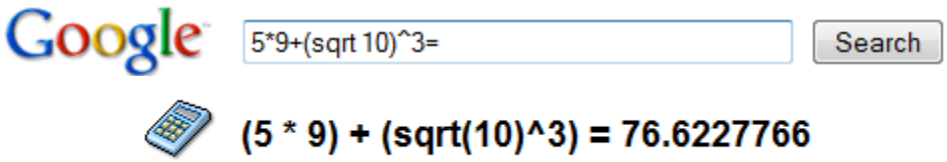
- it's an interpreter of expressions with fun extensions

In Lec3, we'll make the calc language user-extensible

- users can without extending the interpreter

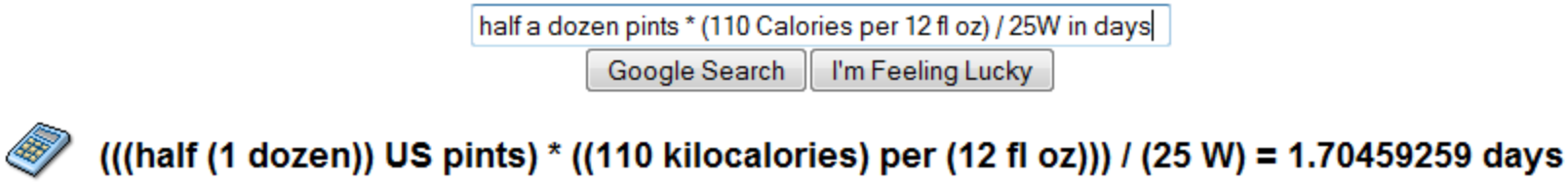
Recall Lecture 1

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



Then you remember cs164 and easily add unit conversion.

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



Programs from our calculator language

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

Constructs of the Calculator Language

- numbers
  - ints
  - floats
- units → modeled as "types"
- operators
  - +, *, /, - "per"-alias for /
  - in : conversion op
  - ( )

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

What additional features may we want

what features we may want to add?

- think usage scenarios beyond those we saw
- talk to your neighbor
- we'll add some of these in the next lecture

can we view these as user extending the language?

- new unit types
- time, real time
- RPN - new syntax
- infinite prec. arith.
- eq solving (vars)
- complex, base-2, matrices
- integrals, ...
- pow, and arbitr. op, int div

Additional features we will implement in Lec3

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

We'll grow the language a feature at a time

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

Sublanguage of arithmetic expressions

A programming language is defined as

Syntax: set of valid program strings

2 + 3 legal

+ 2 3 illegal

Semantics: how the program evaluates

$e_1 + e_2$ performs an addition of the values of expressions e_1 and e_2

Syntax

The set of syntactically valid programs is 🕓 large.

So we define it recursively:

a recursive definition of the language

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 painfully learn soon.

How to represent a program?

concrete syntax

(input program)

"1+2"

"(3+4)*2"

a flat string

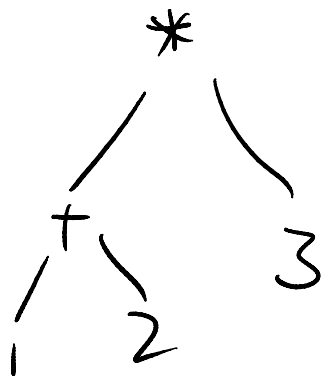
abstract syntax

(internal program representation)

('+', 1, 2)

('*', ('+', 3, 4), 2)

tree built from Python tuples



AST

abstract syntax tree

The interpreter

Recursive descent over the abstract syntax tree

```
ast = ('*', ('+', 3, 4), 5)
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

Add values that are physical units (SI only)

Example:

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

Concrete syntax:

E ::= n | **U** | E op E | (E)

U ::= m | s | kg

op ::= + | - | * | ε | / | ^

1m is same as 1*m

"" empty string

Abstract syntax: represent SI units as string constants

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

parser translates the missing * into an explicit *

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 a map from an SI unit to its exponent:

evaluates to

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

Python dictionary

'm' is key
2 is value

The interpreter

Ex: m/m → (1, {}) empty dict.

```
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))
```

$m^2 * kg \rightarrow m^2 * kg$

$m^2 * m^3 \rightarrow m^5$

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)
3. Non-SI units
4. Explicit unit conversion

You are expected to read the code

It will prepare you for PA1

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

The code

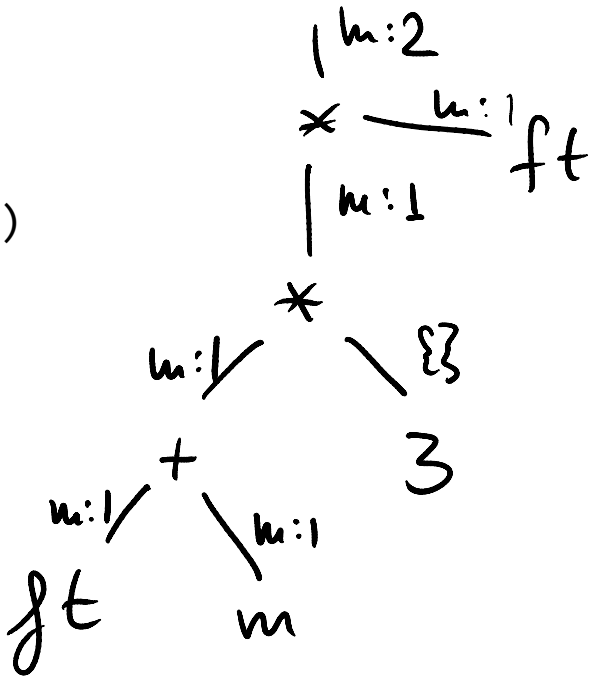
```
def eval(e):
    if type(e) == type(1):    return (e,{})
    if type(e) == type(1.1): return (e,{})
    if type(e) == type('m'): return lookupUnit(e)

def lookupUnit(u):
    return {
        'm'      : (1,       {'m':1}),
        'ft'     : (0.3048, {'m':1}),
        'in'     : (0.0254, {'m':1}),
        's'      : (1,         {'s':1}),
        'year'   : (31556926, {'s':1}),
        'kg'     : (1,           {'kg':1}),
        'lb'     : (0.45359237, {'kg':1})
        }[u];
```

unit

string

dict.



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

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”

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.

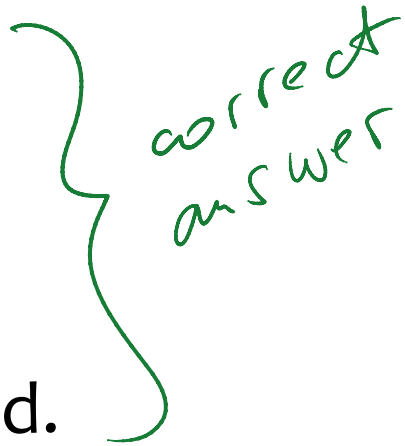
For that, E in C must yield a value. Is that what we want?

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

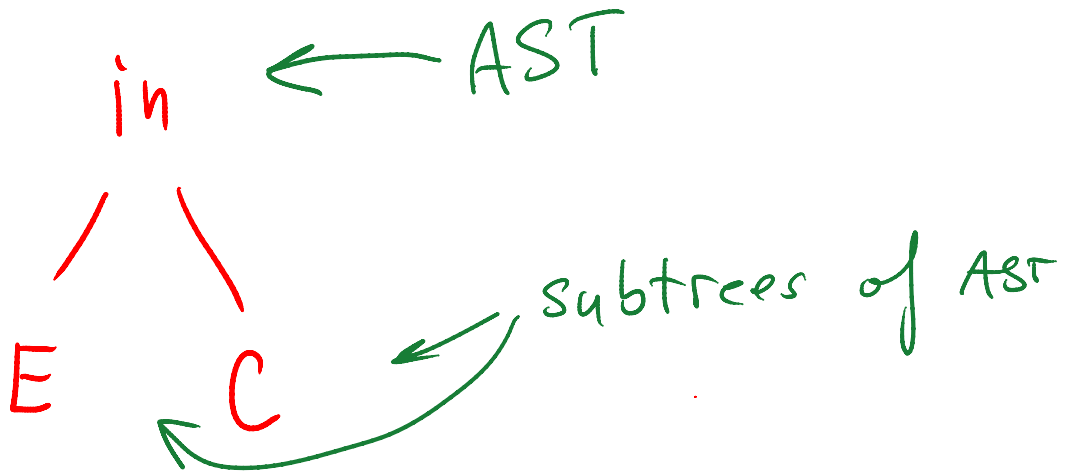
Examples of valid programs:

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 as 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

   2 ft/year in m/s

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

2. a representation of C, for printing

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

let me know if you find simple design

we could make other sensible choices here

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 include 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