# Lecture 2

# Unit Conversion Calculator

## Expressions, values, types. Their representation and interpretation.

**Ras Bodik**
with Mangpo and Ali

***Hack Your Language!***
***CS164***: Introduction to Programming Languages and Compilers, Spring 2013
UC Berkeley

# Administrativia

These are supplementary slides.

Review the material and the sample code to understand interpreters.

# 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

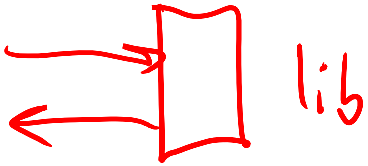
Why small languages?

- see next slide and lecture notes

# MapReduce story

Evolution of sw reuse and design

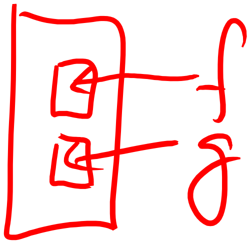
- library



MPI

- framework

parameterizable library

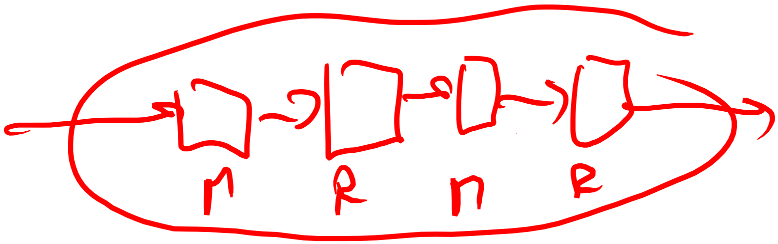


MapReduce

- 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

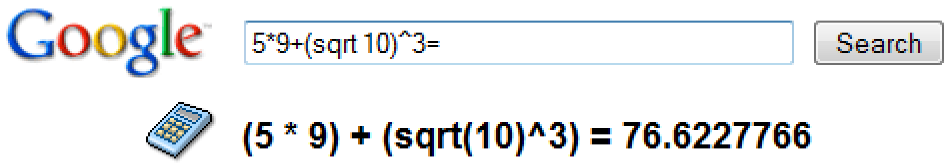
- where do we catch the error?

Using unit calculator as our running study

- it’s an interpreter of expressions with fun extensions

# Recall Lecture 1

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

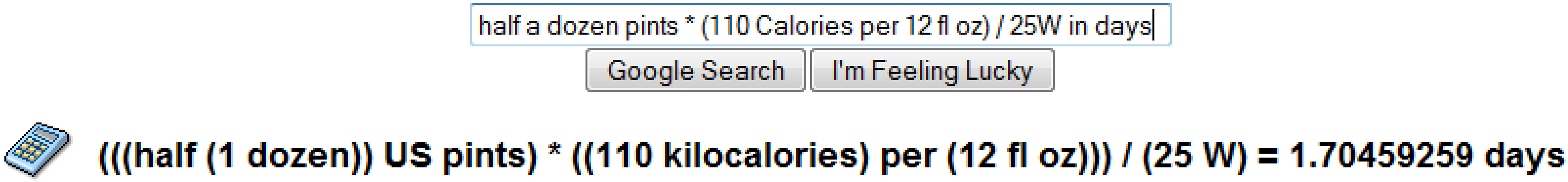


5*9+(sqrt 10)^3=

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

Then you remember cs164 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**

# 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

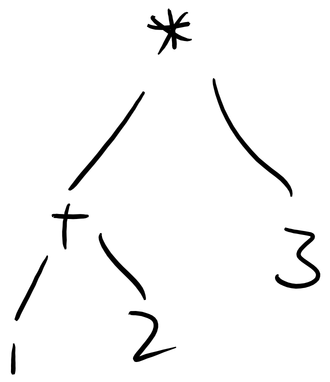
How about if $e_1$ or $e_2$ is a float?

- Then the result is a float.

There are more subtleties, as we painfully learn soon.

# How to represent a program?

| **concrete syntax** | **abstract syntax** |
|---|---|
| (input program) | (internal program representation) |
| "1+2" | `('+', 1, 2)` |
| "(3+4)*2" | `('*', ('+', 3, 4), 2)` |
| a flat string | tree built from Python tuples |



AST
abstract
syntax
tree

# The interpreter

Recursive descent over the abstract syntax tree

(3+4) * 5

```
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])
```

int

float

tuple

# 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 | / | ^

"" empty string

=> 1m is same as 1*m

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: $m/m \rightarrow (1, \{\})$ ← 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

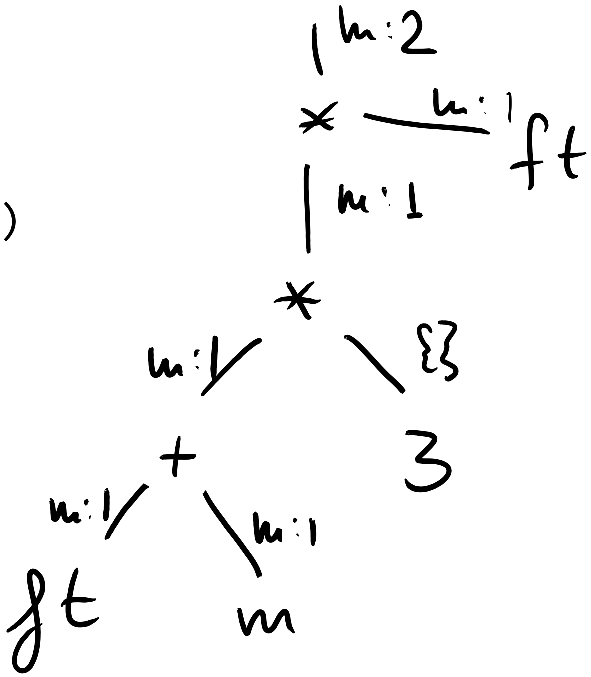
```
def eval(e):
    if type(e) == type(1):   return (e,{})
    if type(e) == type(1.1): return (e,{})
    if type(e) == type('m'): return lookupUnit(e)

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.

correct answer

# Next, what are the valid forms of C?

**Attempt 1:**

2 ft in m-mm ?

NO

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

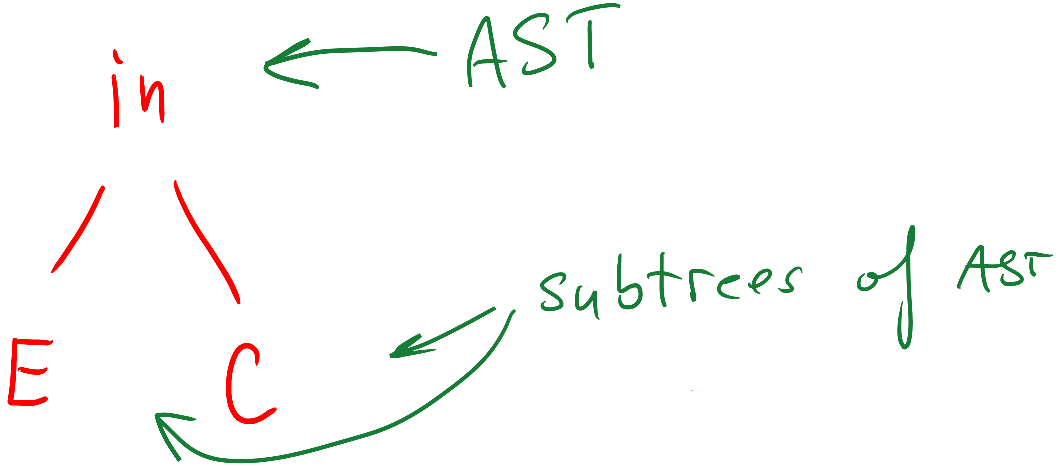
Examples of valid programs:

**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

2ft/year in m/s

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

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 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

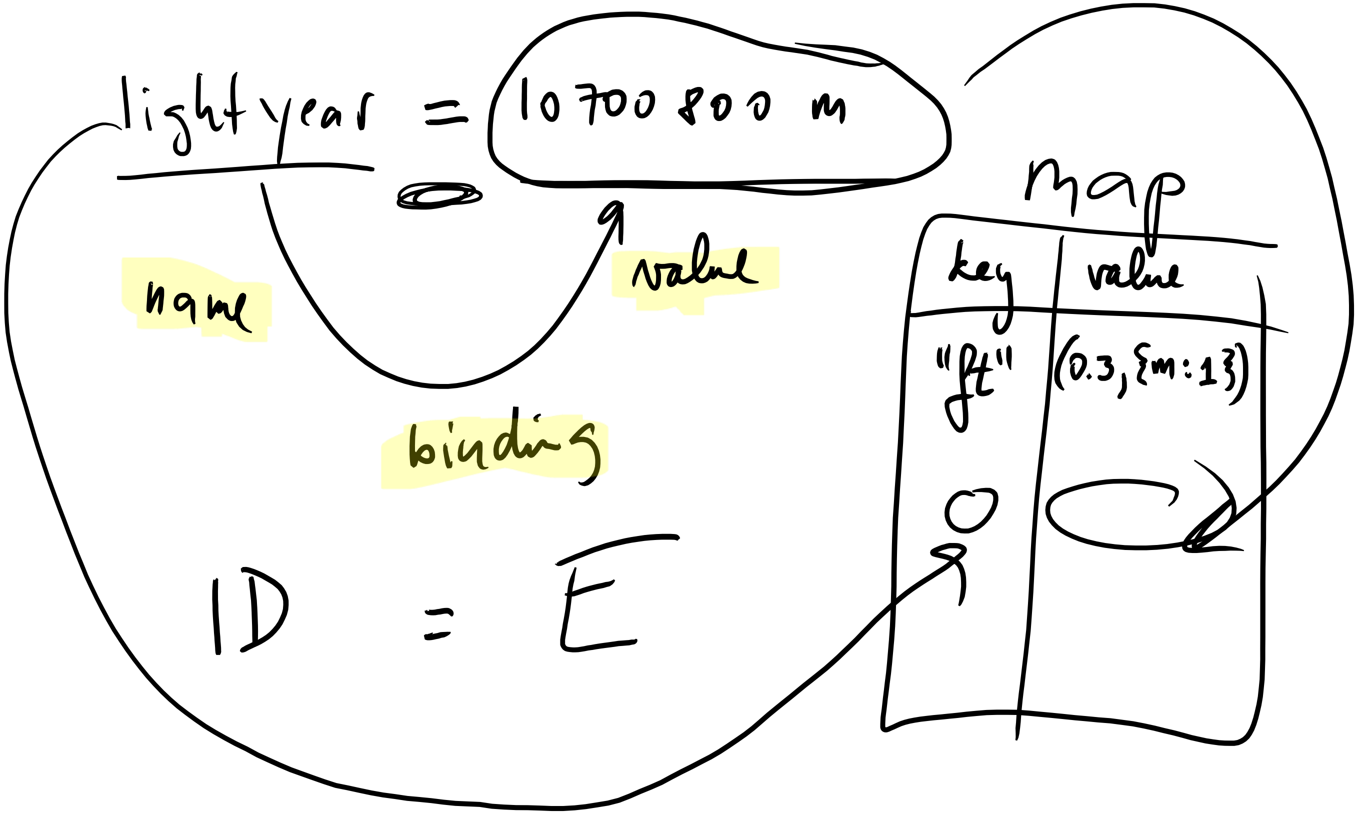
# Bind a value to an identifier

```
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
```

# Our solution

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 = A / 1000

Solve

declaration: SI␣A

mA = 0.001 A

| | |
|---|---|
| "A" | (1, {A: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
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

# Motivating 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
```

we wish to remember it as a constant

```
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) ✓
8. Reuse of expressions (bind names to expressions)

# Another motivating example

You want to print the current time left to deadline

now = 2011 year + 0 month + 18 day + 15 hour + 40 minute

--- pretend that now is always set to current time of day

Let’s try to compute time to deadline

deadline = 2011 year + 1 month + 3 day // 2/3/2012

timeLeft = deadline - now

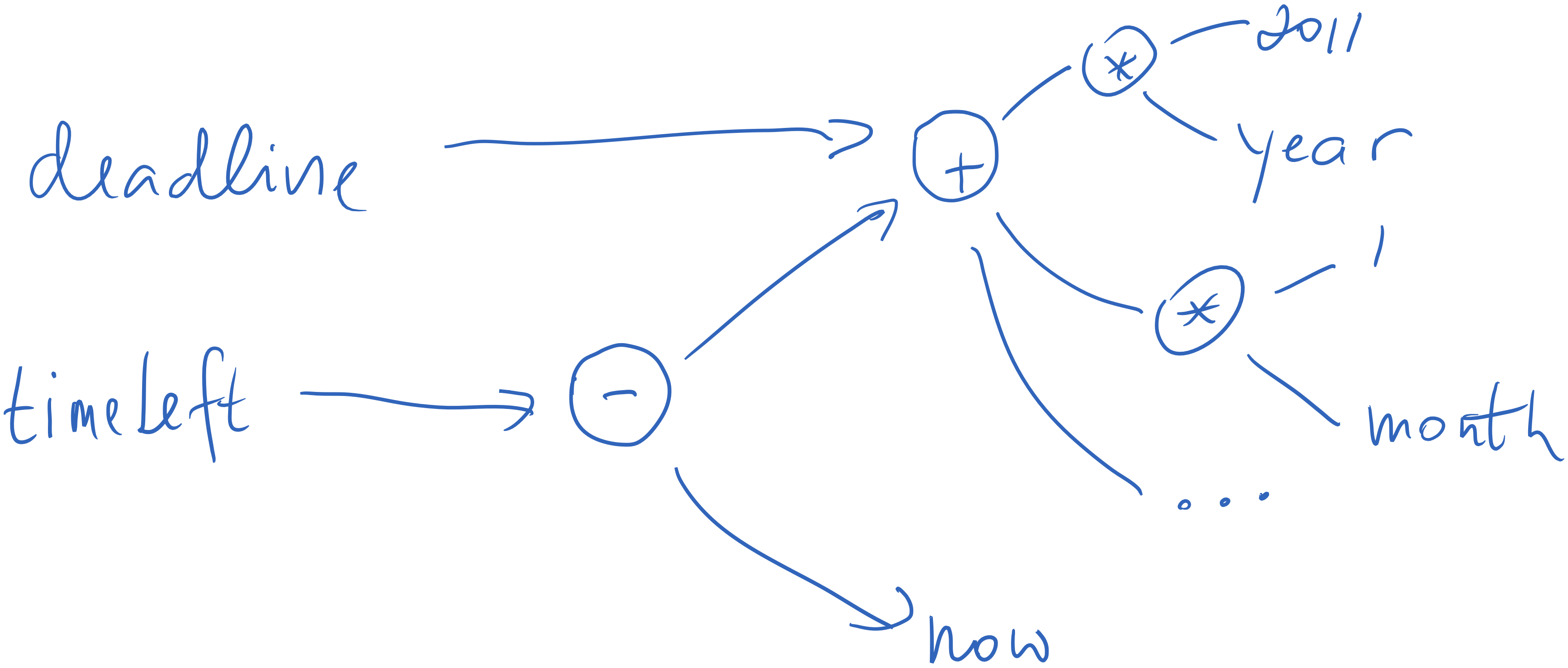
timeLeft in day --> *time left*

Wait for current time to advance. Print time left now. What does the following print?

timeLeft in day --> updated *time left*

How to achieve this behavior?

# timeLeft is bound to an expression



year, month are actually expressions, too

# Naming values vs. naming expressions

“Naming an expression” means that we evaluate it <u>lazily</u> when we need its value

# 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)
8. Reuse of expressions code (not fully lazy)

# Summary: Calculator is an extensible language

Very little built-in knowledge

- Introduce base units with ‘SI name’
- Arithmetic performs general unit types and conversion

No need to define all units in terms of SI units

```
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
```

-> Same mechanism as for introduction of non-SI units!

No need to remember units! Both will work fine!

```
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
- You will implement his parser in PA6

# 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 declaration

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

names can bind to expressions, not only to values

- these expressions are evaluated lazily