CSCI 334:
Principles of Programming Languages

Lecture 6: The Dream of Computation

Instructor: Dan Barowy

Williams

Topics

A little more folding

Syntax in Backus-Naur Form

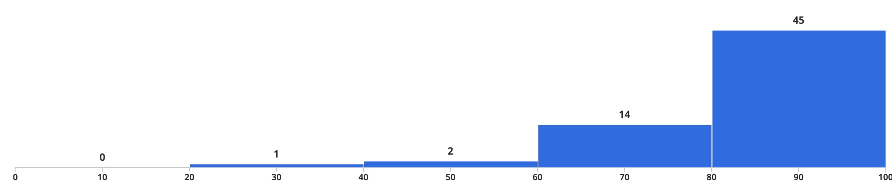
What can computers really do?

Lambda calculus

Your to-dos

1. Lab 3, **due Wednesday 2/26** (partner lab).
2. Read *Higher Order Functions.*
3. Prepare for Quiz 3.

Quiz 2



Median	Maximum	Mean	Std Dev
95.0	100.0	87.16	16.04

If you got >80%, nice work!

If you got <80%, let's find a time to meet.

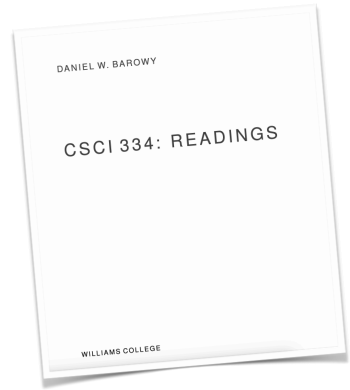
Announcements

- CS Colloquium this **Friday, Feb 28 @ 2:35pm in Wege Auditorium (TCL 123)**

Senior Thesis Proposals, Part 2

Announcements

- Course packet posted to website (see **`Handouts`**)



fold right

```
List.foldBack
  (fun x acc -> acc+x) [1;2;3;4] 0
```

```
[1;2;3;4], acc = 0
[1;2;3], acc = 0+4
[1;2], acc = 4+3
[1] acc = 7+2
[], acc = 9+1
returns acc = 10
```

what does this return?

```
List.foldBack
  (fun x acc -> acc + string x)
  (Seq.toList "williams")
  ""
```

Cartesian product

Write a **function** that computes the **Cartesian product** of two sets, represented by lists:

$$A \times B = \{ (a,b) \mid a \in A \text{ and } b \in B \}$$

Hint: We just learned about maps.

```
let cartesianProduct xs ys =
  …
```


Cartesian product

Write a **function** that computes the **Cartesian product** of two sets, represented by lists:

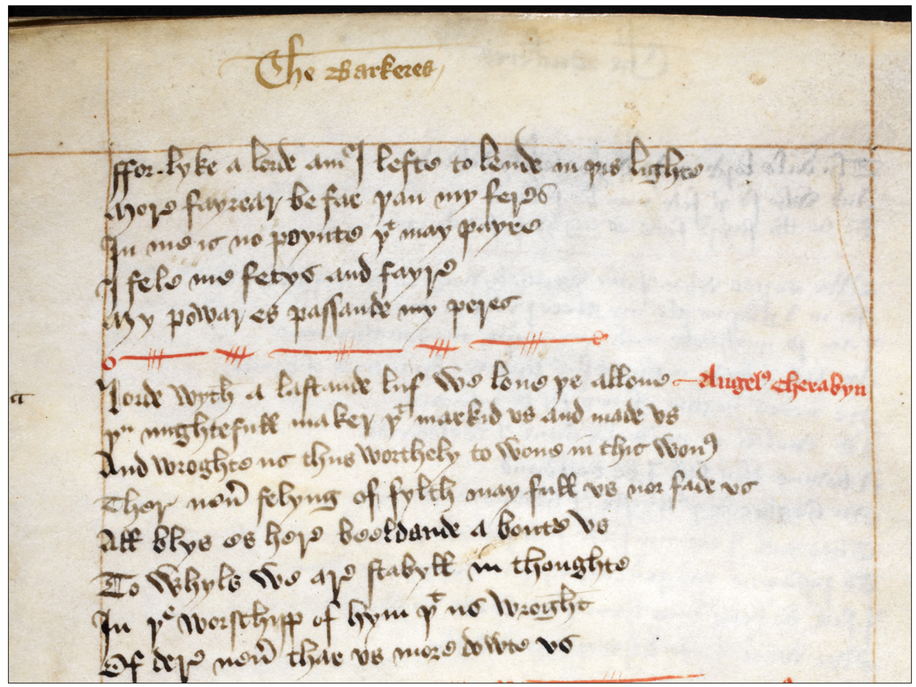
$$A \times B = \{ (a,b) \mid a \in A \text{ and } b \in B \}$$

Hint: We just learned about maps.


```
let rec flatten (xss: ('a*'b) list list) : ('a * 'b) list =
    match xss with
    | [] -> []
    | xs::xss' -> xs @ (flatten xss')

let CartesianProduct (xs: 'a list) (ys: 'b list) =
    List.map (fun x ->
        List.map (fun y ->
            (x,y)
        ) ys
    ) xs
    |> flatten
```

Language of languages



Why couldn't you understand the script?

It's written in English, after all!

We don't know the "ground rules" for the document as it is written:

- Appearance: **syntax**
  - What is the set of valid **symbols**?
  - What **arrangements** of symbols are permissible?
- Meaning: **semantics**
  - What does **a given arrangement** of symbols correspond **mean**?

Formal language

A **formal language** is the set of permissible **sentences** whose **symbols** are taken from an **alphabet** and whose word **order** is determined by a specific set of **rules**.

Intuition: a language that can be defined mathematically, using a **grammar**.

English **is not** a formal language.

Java **is** a formal language.

More formally

$L(G)$ is the set of all sentences (a "language") defined by the grammar, G.

$G = (N, \Sigma, P, S)$ where

N is a set of nonterminal symbols.

Σ is a set of terminal symbols.

P is a set of production rules of the form
$N ::= (\Sigma \cup N)^*$
where * means "zero or more" (Kleene star) and
where $\cup$ means set union

$S \in N$ denotes the "start symbol."

Backus-Naur Form (BNF)

More concretely, for programming languages, we conventionally write **G** in a form called **BNF**.

John Backus

Peter Naur

Invented in 1959 to describe the ALGOL 60 programming language.

Tower of Hanoi (ALGOL 60)

```
begin
  procedure movedisk(n, f, t);
  integer n, f, t;
  begin
    outstring (1, "Move disk from");
    outinteger(1, f);
    outstring (1, "to");
    outinteger(1, t);
    outstring (1, "\n");
  end;

  procedure dohanoi(n, f, t, u);
  integer n, f, t, u;
  begin
    if n < 2 then
      movedisk(1, f, t)
    else
      begin
        dohanoi(n - 1, f, u, t);
        movedisk(1, f, t);
        dohanoi(n - 1, u, t, f);
      end;
  end;

  dohanoi(4, 1, 2, 3);
  outstring(1,"Towers of Hanoi puzzle completed!")
end
```

Backus-Naur Form (BNF)

Nonterminals, **N**, are in brackets: `<expression>`

Terminals, **Σ**, are “bare”: `x`

A production rule, **P**, consists of the `::=` operator, a nonterminal on the left hand side, and
a sequence of one or more symbols from **N** and **Σ** on the right hand side.

```
<variable> ::= x
```

The `|` symbol means “alternatively”: `<num> ::= 1 | 2`

We use ε to denote the empty string.

Backus-Naur Form (BNF)

You should read the following BNF expression:

```
<num> ::= <digit>
        | <num><digit>
```

as

“`num` is defined as a `digit` or as a `num` followed by a `digit`.”

Backus-Naur Form (BNF)

The following definition might look familiar:

```
<expr>  ::= <num>
          | <expr> + <expr>
          | <expr> - <expr>
<num>   ::= <digit>
          | <num><digit>
<digit> ::= 0|1|2|3|4|5|6|7|8|9
```

`<expr>` is the start symbol.

Conventionally, we ignore whitespace, but if it matters, use the ␣ symbol. E.g.,

```
<expr>␣+␣<expr>
```

Parsing and Parse Trees

Parsing is the process of analyzing a string of symbols, conforming to the rules of a formal grammar, to understand:

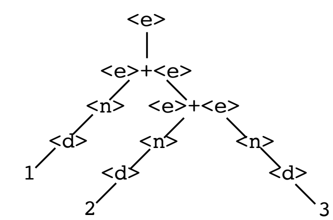
1) whether that sentence is valid (**s** $\in$ L(**G**)), or
2) the structure (e.g., "parts of speech") of that sentence (a **parse tree**).

Derivation Tree

Shows **every step** of how a sentence is **parsed**.

```
<e> ::= <n> | <e>+<e> | <e>-<e>
<n> ::= <d> | <n><d>
<d> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

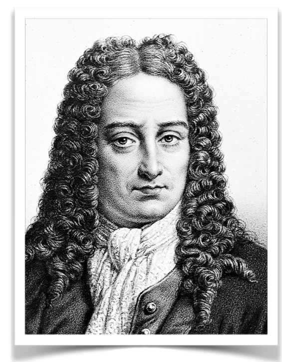
1+2+3



What can computers really do?

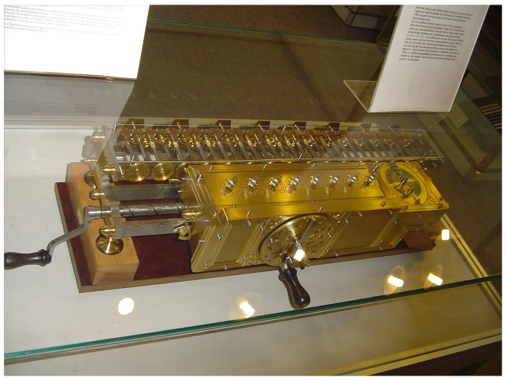
The Dream

"I thought again about my early plan of a new language or writing-system of reason, which could serve as a communication tool for all different nations... If we had such an universal tool, we could discuss the problems of the metaphysical or the questions of ethics in the same way as the problems and questions of mathematics or geometry. That was my aim: Every misunderstanding should be nothing more than a miscalculation (...), easily corrected by the grammatical laws of that new language. Thus, in the case of a controversial discussion, two philosophers could sit down at a table and just calculating, like two mathematicians, they could say, 'Let us check it up …'"

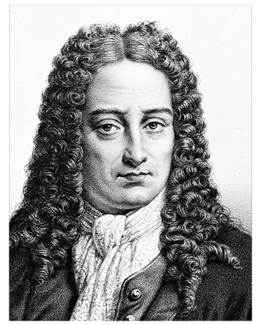


Wilhelm Gottfried Leibniz

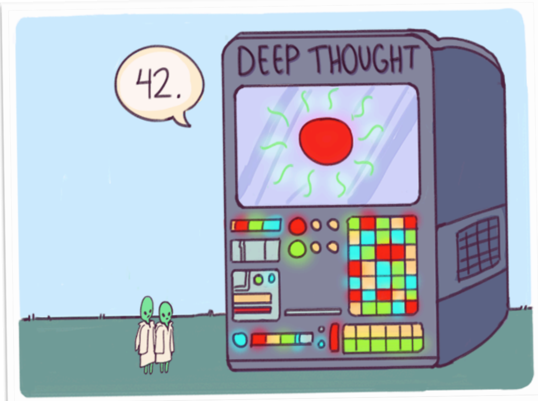
The Dream



"stepped reckoner"



Wilhelm Gottfried Leibniz



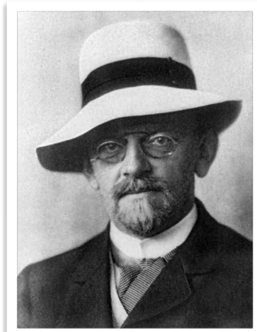
"What is the answer to the ultimate question of life, the universe, and everything?"

What is computable?

- Hilbert: Is there an **algorithm** that can decide whether **any logical statement is valid**?
- "Entscheidungsproblem" (literally "decision problem")
- Leibniz thought so!

What is computable?

- Why do we care?
- f(x) = x + 1
- We can clearly do this with pencil and paper.
- *∫6x dx*
- Also computable, in a different manner.
- We care because the computable functions can be done on a "**computer**."



Lambda calculus

- Invented by Alonzo Church in order to solve the Entscheidungsproblem.
- Short answer to Hilbert's question: no.
- Proof: **No algorithm can decide equivalence** of two arbitrary λ-calculus expressions.
- By implication: no algorithm can determine whether an arbitrary logical statement is valid.

What is the meaning of `x` in **algebra**?

Pro tip

Don't try to "**understand**" the lambda calculus.

Aside from "**variable**," "**function definition**," and "**application**," it has no inherent meaning.

We **ascribe meaning** to it, just as we do with algebra.

The lambda calculus is simply a **system** for reasoning by using **the logic of functions**.

Lambda calculus grammar

```
<expr>  ::= <var>
          |  <abs>
          |  <app>
<var>   ::= x
<abs>   ::= λ<var>.<expr>
<app>   ::= <expr><expr>
```

`<expr>` is the start symbol.

What is a variable?

```
<var>    ::= x
```

It's just a value.

What is an abstraction?

```
<abs>    ::= λ<var>.<expr>
```

It's a function definition

```
def foo(x):
  <expr>
```

What is an application?

```
<app>    ::= <expr><expr>
```

It's a "function call"

```
foo(2)
```

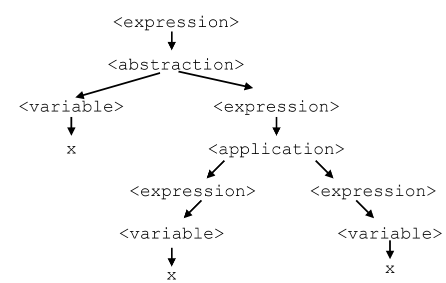
```
<expr><expr>
```

function argument

Parsing Lambda Expressions

Let's try parsing this expression

```
λx.xx
```



Recap & Next Class

Today:

BNF

Lambda calculus / computation

Next class:

More on lambda calculus