CSCI 334:
Principles of Programming Languages

Lecture 11: Midterm Exam Review

Instructor: Dan Barowy

Williams

Announcements

- **Midterm exam**, in class, Thursday, Oct 19.
- **Field trip to WCMA**, Thursday, Nov 2.
- Colloquium: **What I Did Last Summer (Research Edition)**, 2:35pm in Wege Auditorium.

Announcements

- **TA Applications** due Friday, Oct 27.
- **TA Evaluation** forms due Friday, Oct 27.
-

Your to-dos

1. Study for **Thursday's exam**.

What is a language?

In this class, we concern ourselves with a specific formulation of “language,” called a **formal language**.

A **formal language** is the set of words whose letters are taken from some **alphabet** and whose construction follows some **rules**.

Example:

```
L = {a, aa, b, bb, ab, ba}

Σ = {a, b}

<expr> ::= <letter> | <letter><letter>
<letter> ::= a | b
```

What is a programming language?

A **programming language** is defined by two machines:

1. A **syntax machine** that determines the set of strings that are in the language.
2. A **semantics machine** that determines what gets done (i.e., what computational work) with an accepted string.

We spend a lot of time in PL thinking about these machines, which we call **language models**.

ML

- Robin Milner
- How to program tactics?
- A “meta language” is needed
- ML is born (1973)
- First impression upon encountering a computer: "Programming was not a very beautiful thing. I resolved I would never go near a computer in my life."

`unit` datatype

```
$ dotnet fsi

Microsoft (R) F# Interactive version 10.2.3 for F# 4.5
Copyright (c) Microsoft Corporation. All Rights Reserved.

For help type #help;;

> unit;;

  unit;;
  ^^^^

stdin(1,1): error FS0039: The value or constructor 'unit' is
not defined.

> ();;
val it : unit = ()

>
```

How does one obtain a value of `unit`? `()`

You can also `ignore`…

```
> let foo() = 2;;
val foo : unit -> int

> foo();;
val it : int = 2

> ignore (foo());;
val it : unit = ()

> foo() |> ignore;;
val it : unit = ()

>
```

"forward pipe" operator

```
<expr> |> <expr>

foo() |> ignore
```

Pattern matching

```
let rec product nums =
  if (nums = []) then
    1
  else
    (List.head nums)
    * product (List.tail nums)
```

Using **patterns**…

```
let rec product nums =
  match nums with
  | []     -> 1
  | x::xs -> x * product xs
```

Activity: Pattern matching on integers

Write a function `listOfInts` that returns a list of integers from **zero** to `n`.

```
let rec listOfInts n =
  match n with
  | 0 -> [0]
  | i -> i :: listOfInts (i - 1)
```

Oops! This returns the list backward.

Let's flip it around.

Revisiting local declarations

Let's fix our code the lazy way…

```
let listOfInts n =
  let rec li n =
    match n with
    | 0 -> [0]
    | i -> i :: listOfInts (i - 1)
  li n |> List.rev
```

… by defining a function inside our function.

Algebraic Data Type

An **algebraic data type** is a composite data type, made by combining other types in one of two different ways:

- by **product**, or
- by **sum**.

You've already seen **product types**: tuples and records.

So-called b/c the set of all possible values of such a type is the cartesian product of its component types.

We'll focus on **sum types**.

A "move" function in a game (F#)

Discriminated Union (sum type)

```
type Direction =
     North | South | East | West;
let move coords dir =
  match coords,dir with
  |(x,y),North -> (x,y - 1)
  |(x,y),South -> (x,y + 1)
```

- Above is an "incomplete pattern"
- ML will warn you when you've missed a case!
  - "proof by exhaustion"

ADTs can be recursive and generic

```
type MyList<'a> =
      | Empty
      | NonEmpty of head: 'a * tail: MyList<'a>
```

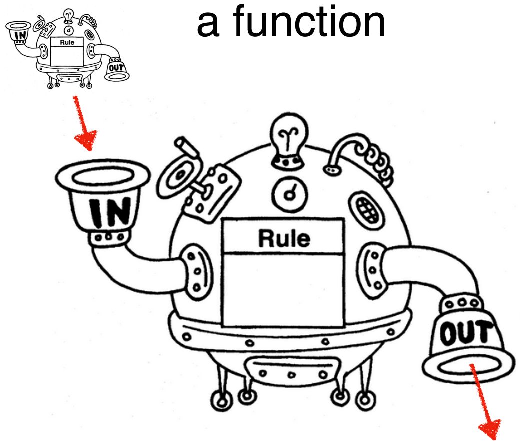
```
> NonEmpty(2, Empty);;
    val it : MyList<int> = NonEmpty (2,Empty)
```

Avoiding errors with patterns

- Another example: handling errors.
- SML has exceptions (like Java)
- But an alternative, easy way to handle many errors is to use the option type:

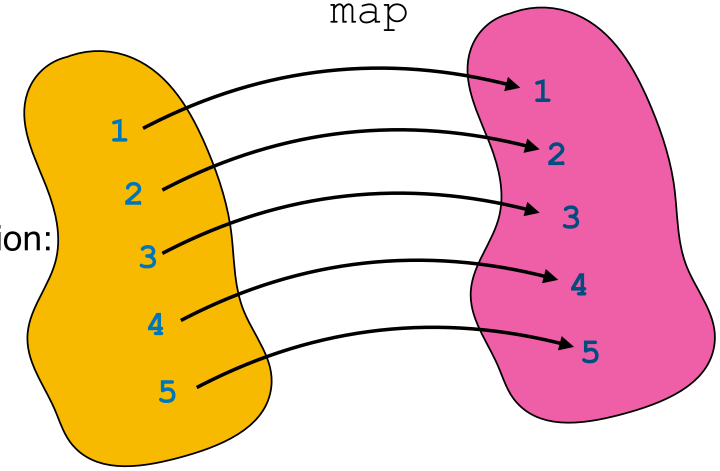
```
type option<'a> =
| None
| Some of 'a
```

a function



map

Intuition:

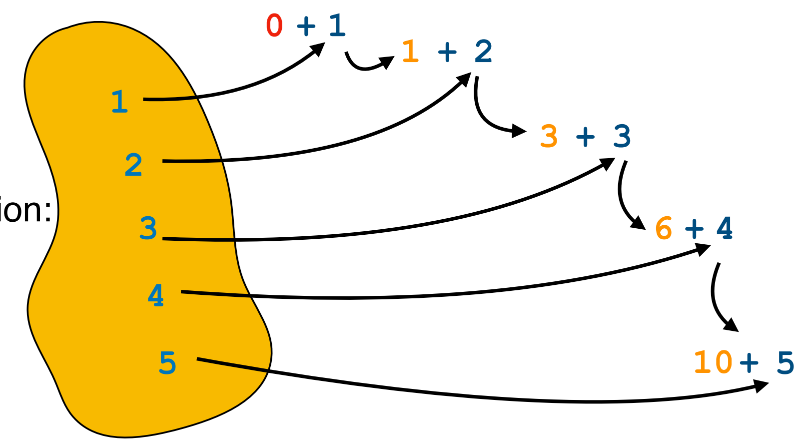


Like a `for` loop, but without mutable variables

```
List.map (fun x -> x + 1) [1;2;3;4;5]
```

fold

Intuition:




```
List.fold (fun acc x -> acc+x) 0 [1;2;3;4;5]
```

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`."

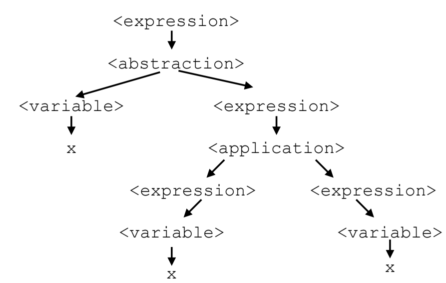
Lambda Calculus Grammar

```
<expr>    ::= <value>
            |  <abs>
            |  <app>
            |  <parens>
<var>     ::= α ∈ { a ... z }
<abs>     ::= λ<var>.<expr>
<app>     ::= <expr><expr>
<parens>  ::= (<expr>)
<value>   ::= v ∈ ℕ
            |  <var>
```

Derivation Tree

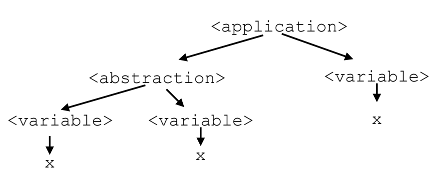
We can create a “derivation tree” by following the rules of a grammar as we interpret a sentence of a language.

`λx.xx`



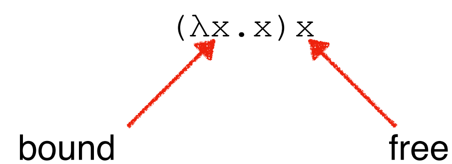
Abstract Syntax Tree

`(λx.x)x`



Obtained by removing all nonterminals not associated with an operation.

Free vs bound variables



Evaluation: Lambda calculus is like algebra

(λx.x)x

Evaluation consists of simplifying an expression using text substitution.

Only two simplification rules:

α-reduction

β-reduction

α-Reduction

(λx.x)x

This expression has two **different** x variables

Which should we rename?

Rule:

⟦λx.<expr>⟧ $=_\alpha$ ⟦λy.[y/x]<expr>⟧

[y/x]<expr> means "substitute y for x in <expr>"

β-Reduction

(λx.x)y

How we "call" or **apply** a function to an argument

Rule:

⟦(λx.<expr>)y⟧ $=_\beta$ ⟦[y/x]<expr>⟧

(λx.λy.yx)xy	given
(λa.λy.ya)xy	α-reduce a for x
(λa.λb.ba)xy	α-reduce b for y
(λb.bx)y	β-reduce x for a
(yx)	β-reduce y for b
yx	remove parens

How far do we go?

We keep going until there is **nothing left to simplify**.

`x`	←	done
`xx`	←	done
`λx.y`	←	done
`(λx.xy)z`	←	not done

That “most simplified” expression is called a **normal form**.

An expression that can be simplified is a called a **redex**.

Watch out!

`λx.xy`	given
`λy.[y/x]xy`	α-reduce `y` for `x`
`λy.yy`	inner α-reduction
	this is incorrect!

The lambda has “captured” the free `y`.
Substitution must be **capture-avoiding**.

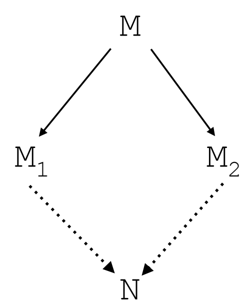
Watch out!

`(λx.λx.x)x`	given
`([x/x]λx.x)`	β-reduce `x` for `x`
`(λx.x)`	β-reduce inner expr
	done

The inner lambda term **redefines** `x` and therefore “blocks” substitution of `x`.

Sometimes multiple reductions available

Order (mostly) does not matter

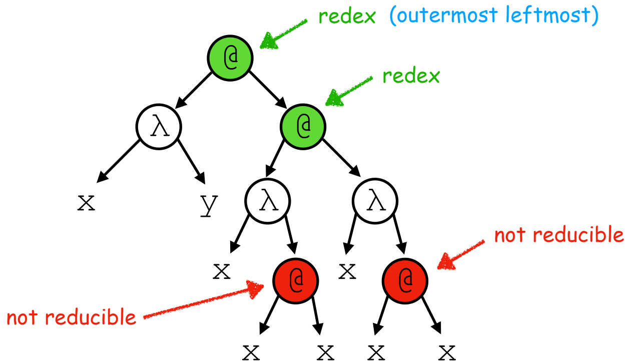


If $M \rightarrow M_1$ and $M \rightarrow M_2$
then $M_1 \rightarrow^* N$ and $M_2 \rightarrow^* N$
for some N

“confluence”

Normal order

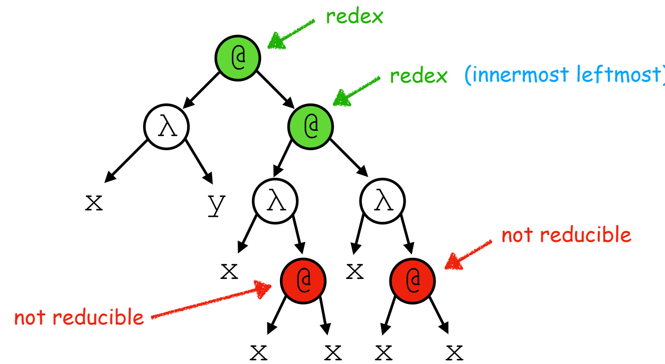
`(λx.y)((λx.xx)(λx.xx))`



Redex: application with abstraction as left child.

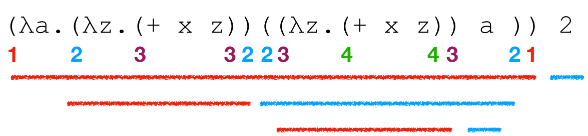
Applicative order

`(λx.y)((λx.xx)(λx.xx))`

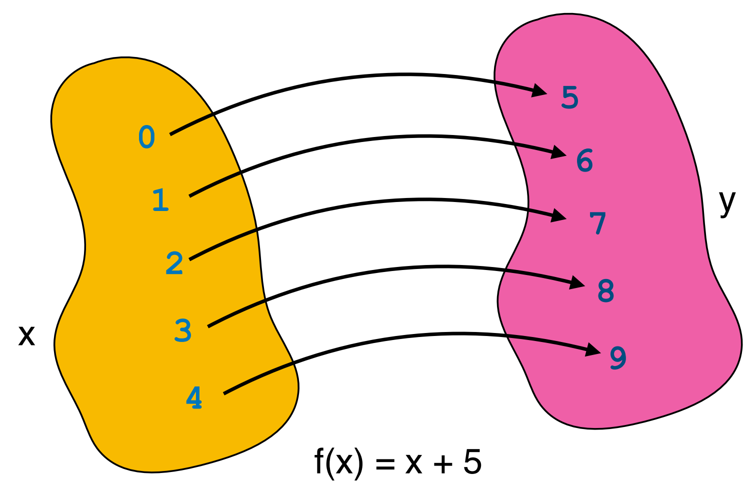


Redex: application with abstraction as left child.

Trouble matching parens? Try this.



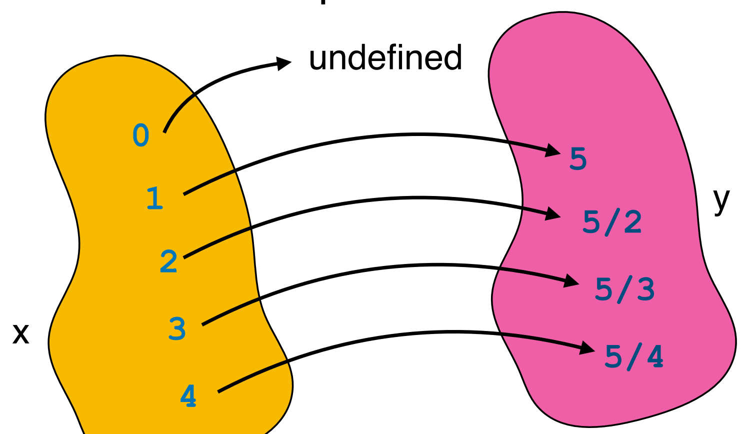
Intuition: total function



f(x) = x + 5

For every element in x, there is a corresponding element in y. x maps to at most one element in y.

Intuition: partial function



$f(x) = 5/x$

x still maps to at most one element in y, however, there is not a y for every x.

The **graph** of a function

$$f(x) = x + 5$$

$$\{\langle x, x+5\rangle \mid x \in \mathbb{Z}\}$$

$$\{\langle x, x+5\rangle \mid x \text{ is an integer}\}$$

The graph is **not a picture**!

Decidability Problems

A **decidability problem** is a question with a **yes** or **no** answer about a **particular input**.

"Is x prime?"

In CS, we care about whether there is an **algorithm** for solving decidability problems.

If there is **no algorithm**, then the problem is **undecidable**.

The Halting Problem

Decide whether program **P** halts on input **x**.

Given program P and input x,

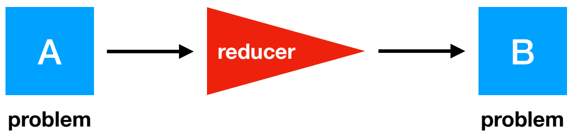
$$\texttt{Halt(P,x)} = \begin{cases} \text{returns } \texttt{true} \text{ if } \texttt{P(x)} \text{ halts} \\ \text{returns } \texttt{false} \text{ otherwise} \end{cases}$$

How might this work?

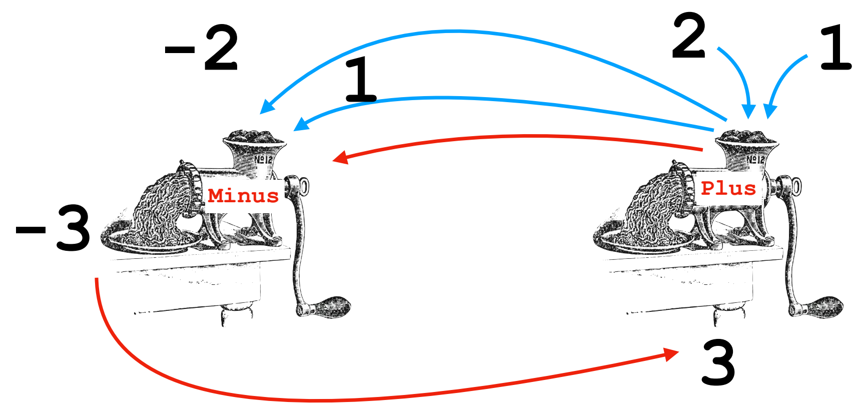
Fact: it is provably impossible to write `Halt`

Reductions

A **reduction** is an **algorithm** that transforms an instance of one problem into an instance of another. Reductions are often **employed to prove something** about a problem given a similar problem.

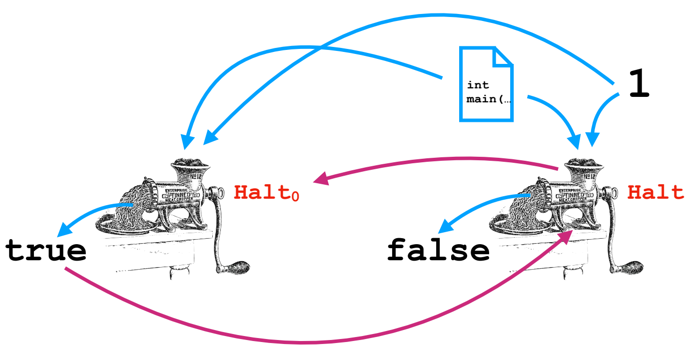


Reductions




```
let reducer(x: int)(y: int) = 0-(0-x-y)
```

Reductions



If we can build this new machine, what does that mean for `Halt`$_0$?

`Halt`$_0$ is **not computable**.

Reductions

We can use the Halting Problem to show that other problems cannot be solved **by reduction** to the Halting Problem.

We cannot tell, in general…

… if a program will **run forever**.

… if a program will **eventually produce an error**.

… if a program **is done using a variable**.

… if a program **is a virus**!

Q&A

Recap & Next Class

Today:

Midterm Exam Review

Next class:

Midterm Exam