The Lambda Calculus

- Lambda calculus terms
  - variables, abstractions, & applications
- Variable scope
  - Free vs bound variables
- Evaluation
  - Alpha renaming
  - Beta reduction
  - Normal form
- Church encodings
  - numbers, booleans, etc
- Recursion
  - Fixed-point combinator

Haskell

- A typed, lazy, purely functional programming language
  - Haskell \( = \) \( \lambda \)-calculus +
    - Better syntax
    - Types
    - Built-in features
      - Booleans, numbers, characters
      - Records (tuples)
      - Lists
      - Recursion
      - ...
Haskell topics

- Haskell's type system
  - Recognizing / understanding relationship between Haskell expressions and their types
- Algebraic data types
  - Records
  - Sum types
  - Recursive ADTs
- Pattern matching
  - Overlapped / missing patterns
- Writing algorithms on (recursive) ADTs
  - Base cases + inductive cases

Higher Order Functions

Iteration patterns over collections:
- Filter values in a collection given a predicate
- Map (iterate) a given transformation over a collection
- Fold (reduce) a collection into a value, given a binary operation to combine results

Useful helper HOFs:
- Flip the order of function’s (first two) arguments
- Compose two functions
Evaluating Nano1

Back to our expressions... now with environments!

data Expr = Num Int -- number |
|   Var Id    -- variable |
|   Bin Binop Expr Expr -- binary expression |
|   Let Id Expr Expr -- let expression |

Static vs Dynamic Scoping

Dynamic scoping:

- each occurrence of a variable refers to the most recent binding during program execution
- can’t tell where a variable is defined just by looking at the function body
- nightmare for readability and debugging:

```haskell
let cTimes = \x -> c * x in
let c = 5 in
let res1 = cTimes 2 in -- ==> 10
let c = 10 in
let res2 = cTimes 2 in -- ==> 20!!
res2 - res1
```

Static vs Dynamic Scoping

What we want:

```haskell
let c = 42 in
let cTimes = \x -> c * x in
let c = 5 in
cTimes 2
=> 84
```

Lexical (or static) scoping:

- each occurrence of a variable refers to the most recent binding in the program text
- definition of each variable is unique and known statically
- good for readability and debugging: don’t have to figure out where a variable got “assigned”
Static vs Dynamic Scoping

What we don’t want:

```plaintext
let c = 42 in
let cTimes = \x -> c * x in
let c = 5 in
cTimes 2
=> 10
```

Dynamic scoping:
- each occurrence of a variable refers to the most recent binding during program execution
- can’t tell where a variable is defined just by looking at the function body
- nightmare for readability and debugging:

```plaintext
let cTimes = \x -> c * x in
let c = 5 in
let res1 = cTimes 2 in -- ==> 10
let c = 10 in
let res2 = cTimes 2 in -- ==> 20!!!
res2 - res1
```

Closures

To implement lexical scoping, we will represent function values as closures

```
Closure = lambda abstraction (formal + body) + environment at function definition

data Value = VNum Int
            | VClos Env Id Expr -- env + formal + body
```
Grammars

A grammar is a recursive definition of a set of trees
- each tree is a parse tree for some string
- parse a string $s$ = find a parse tree for $s$ that belongs to the grammar

A grammar is made of:
- Terminals: the leaves of the tree (tokens!)
- Nonterminals: the internal nodes of the tree
- Production Rules that describe how to “produce” a non-terminal from terminals and other non-terminals
  - i.e. what children each nonterminal can have:
  
  \[
  \text{Aexpr} : \quad \text{NT Aexpr can have as children:}
  \]
  \[
  | \text{Aexpr} \ '+' \ Aexpr \ \{ \ldots \} \quad \text{NT Aexpr, T '+'}, \text{and NT Aexpr, or}
  |
  | \text{Aexpr} \ '-' \ AExpr \ \{ \ldots \} \quad \text{NT Aexpr, T '-'}, \text{and NT Aexpr, or}
  |
  | \ldots
  \]

Type system for Nano2

A type system defines what types an expression can have

To define a type system we need to define:
- the syntax of types: what do types look like?
- the static semantics of our language (i.e. the typing rules): assign types to expressions

\[
G |- e :: T
\]

An expression $e$ has type $T$ in $G$ if we can derive $G |- e :: T$ using these rules

An expression $e$ is well-typed in $G$ if we can derive $G |- e :: T$ for some type $T$
- and ill-typed otherwise

Double identity

let id = \x -> x in
  let y = id 5 in
      id (\z -> z + y)

Intuitively this program looks okay, but our type system rejects it:
- in the first application, id needs to have type $\text{Int} \rightarrow \text{Int}$
- in the second application, id needs to have type $(\text{Int} \rightarrow \text{Int}) \rightarrow (\text{Int} \rightarrow \text{Int})$
- the type system forces us to pick just one type for each variable, such as $\text{id} : \text{Int}$

What can we do?
Inference with polymorphic types

With polymorphic types, we can derive \( e :: \text{Int} \rightarrow \text{Int} \) where \( e \) is

\[
\begin{align*}
\text{let } \text{id} & = \lambda x \rightarrow x \text{ in} \\
\text{let } y & = \text{id} \ 5 \text{ in} \\
\text{id} \ (\lambda z \rightarrow z + y) 
\end{align*}
\]

At a high level, inference works as follows:

1. When we have to pick a type \( T \) for \( x \), we pick a fresh type variable \( a \)
2. So the type of \( \lambda x \rightarrow x \) comes out as \( a \rightarrow a \)
3. We can generalize this type to \( \forall a \cdot a \rightarrow a \)
4. When we apply \( \text{id} \) the first time, we instantiate this polymorphic type with \( \text{Int} \)
5. When we apply \( \text{id} \) the second time, we instantiate this polymorphic type with \( \text{Int} \rightarrow \text{Int} \)

Let's formalize this intuition as a type system!

Typing rules

We need to change the typing rules so that:

1. Variables (and their definitions) can have polymorphic types

\[
\begin{align*}
[T-\text{Var}] & \quad \text{G} |- \ x :: S \quad \text{if } x : S \text{ in } \text{G} \\
\text{G} | e1 :: S \quad \text{G, x:S} | e2 :: T \\
[T-\text{Let}] & \quad \text{-----------------------------------} \\
\text{G} | \text{let } x = e1 \text{ in } e2 :: T 
\end{align*}
\]

2. We can instantiate a type scheme into a type

\[
\begin{align*}
[T-\text{Inst}] & \quad \text{G} |- e :: \forall a \cdot S \\
\text{G} |- e :: [a / T] S 
\end{align*}
\]

3. We can generalize a type with free type variables into a type scheme

\[
\begin{align*}
[T-\text{Gen}] & \quad \text{----------------------------------- if not } (a \text{ in FTV} (\text{G})) \\
\text{G} |- e :: \forall a \cdot S
\end{align*}
\]
Typing rules

The rest of the rules are the same:

[T-Num] \[ G |- n :: Int \]

\[ G |- e_1 :: Int \quad G |- e_2 :: Int \]

[T-Add] \[ G |- e_1 + e_2 :: Int \]

G, x:T1 |- e :: T2

[T-Abs] \[ G |- \lambda x : T1 \to e :: T2 \]

G |- e_1 :: T1 -> T2  G |- e_2 :: T1

[T-App] \[ G |- e_1 e_2 :: T2 \]

Formalizing Nano

Goal: we want to guarantee properties about programs, such as:

- evaluation is deterministic
- all programs terminate
- certain programs never fail at run time
- etc.

To prove theorems about programs we first need to define formally

- their syntax (what programs look like)
- their semantics (what it means to run a program)

Nano1: Operational Semantics

We define the step relation inductively through a set of rules:

[Add-L] \[ e_1 \Rightarrow e_1' \quad \text{-- premise} \]

\[ e_1 + e_2 \Rightarrow e_1' + e_2 \quad \text{-- conclusion} \]

[Add-R] \[ e_2 \Rightarrow e_2' \]

\[ n_1 + e_2 \Rightarrow n_1 + e_2' \]

[Add] \[ n_1 + n_2 \Rightarrow n \quad \text{where } n = n_1 + n_2 \]

\[ e_1 \Rightarrow e_1' \]

[Let-Def] \[ \text{let } x = e_1 \text{ in } e_2 \Rightarrow \text{let } x = e_1' \text{ in } e_2 \]

[Let] \[ \text{let } x = v \text{ in } e_2 \Rightarrow e_2[x := v] \]
Operational semantics

We need to extend our reduction relation with rules for abstraction and application:

\[ e_1 \Rightarrow e_1' \]

\[ [\text{App-L}] \]
\[ e_1 \; e_2 \Rightarrow e_1' \; e_2 \]

\[ e \Rightarrow e' \]

\[ [\text{App-R}] \]
\[ v \; e \Rightarrow v \; e' \]

\[ [\text{App}] \; (\lambda x \rightarrow e) \; v \Rightarrow e[x := v] \]

Now what?

Did you like what you learned here? Want to learn more?

- **CSE 114 (not 116) Functional Programming**
  - Winter 2020, Cormac Flanagan
- **CSE 110A Fundamentals of Compiler Design**
  - Fall 2019, Spring 2020, Wesley Mackey
  - Winter 2020, me
- **CSE 210A: Programming languages**
  - Winter 2020, TBD
- **CSE 210B: Adv. Programming languages**
  - Winter 2020, Cormac Flanagan
  - Spring 2020, me

Thanks and good luck!