Node find (Node prev, Node cur, int key) {
    while (cur.key < key) {
        prev = cur;
        cur = cur.next;
    }
    return cur;
}
Course Staff

- Armando Solar-Lezama
  - Instructor
What this course is about

The top N good ideas in programming languages that you might be embarrassed not to know about. ;)

L01-3
What this course is about

• How we define the meanings of programs and programming languages unambiguously?
• How can we prove theorems about the behavior of individual programs?
• How can we design programming tools to automate that kind of understanding?

Applications:
- finding bugs
- designing languages to prevent bugs
- synthesizing programs
- manipulating programs automatically (refactoring, optimization)
Course outline

Functional Programming
- learn about lambda calculus, Haskell, and OCaml
- learn to make formal arguments about program behavior

Type Theory
- learn how to design and reason about type systems
- use type-based analysis to find synchronization errors, avoid information leaks and manage your memory efficiently

Axiomatic Semantics/Program Logics
- a different view of program semantics
- learn how to make logical arguments about program correctness
Course Outline

Abstract Interpretation
- use abstraction to reason about the behavior of the program under all possible inputs

Model checking
- learn how to reason exhaustively about program states
- learn how abstraction and symbolic reasoning can help you find bugs in device drivers and protocol designs
Big Ideas (recurring throughout the units)

Operational Semantics
(give programs meanings via stylized interpreters)
Program Proofs as Inductive Invariants
(all induction, all the time!)
Abstraction
(model programs with specifications)
Modularity
(break programs into pieces to analyze separately)
Skills

- Haskell
- Coq
- Ocaml
- Spin
Grading

6 homework assignments
- Each is 15-20% of your grade
- start on them early!
6 Homework Assignments

Pset 1 (out now, due in about 2 weeks!)
- Practice functional programming
- Build some Lambda Calculus interpreters

Pset 2
- Practice more functional programming
- Implement a type inference engine
- Practice writing proofs in Coq

Pset 3
- How to make formal arguments about the properties of a type system
- Coq proof of type safety for a simple language

Pset 4
- Learn about SMT solvers
- Implement your own verifier for simple C-like programs
Homework Assignments Cont.

Pset 5
- Implement an analysis to check for memory errors in C-like programs

Pset 6
- Practice LTL and CTL (two specification languages)
- Learn how to use a model checker
Functional Programming: Functions and Types

Armando Solar Lezama
Computer Science and Artificial Intelligence Laboratory
M.I.T.

Adapted from Arvind 2010

September 9, 2015
Function Execution by Substitution

\[ \text{plus } x \ y = x + y \]

1. \[ \text{plus } 2 \ 3 \rightarrow 2 + 3 \rightarrow 5 \]

2. \[ \text{plus } (2*3) \ (\text{plus } 4 \ 5) \]

\[ \rightarrow \text{plus } 6 \ (4+5) \]
\[ \rightarrow \text{plus } 6 \ 9 \]
\[ \rightarrow 6 + 9 \]
\[ \rightarrow 15 \]

\[ \rightarrow (2*3) + (\text{plus } 4 \ 5) \]
\[ \rightarrow 6 + (4+5) \]
\[ \rightarrow 6 + 9 \]
\[ \rightarrow 15 \]

The final answer did not depend upon the order in which reductions were performed.
Confluence

Informally - The order in which reductions are performed in a Functional program does not affect the final outcome.

This is true for all functional programs regardless whether they are right or wrong.

A formal definition will be given later.
Blocks

\[ \text{let} \]
\[ x = a \cdot a \]
\[ y = b \cdot b \]
\[ \text{in} \]
\[ (x - y)/(x + y) \]

- A variable can have at most one definition in a block
- Ordering of bindings does not matter
This convention allows us to omit many delimiters

\[
\begin{align*}
\text{let} & \quad x = a \times a \\
& \quad y = b \times b \\
\text{in} & \quad (x - y)/(x + y)
\end{align*}
\]

is the same as

\[
\begin{align*}
\text{let} & \quad \{ \quad x = a \times a \; ; \\
& \quad \quad y = b \times b \; ; \}
\text{in} & \quad (x - y)/(x + y)
\end{align*}
\]
Lexical Scoping

```
let
  y = 2 * 2
  x = 3 + 4
  z = let
    x = 5 * 5
    w = x + y * x
  in
  w
in
  x + y + z
```

Lexically closest definition of a variable prevails.
Renaming Bound Identifiers
(\(\alpha\)-renaming)

\[
\begin{align*}
let & \\
y & = 2 \ast 2 \\
x & = 3 + 4 \\
z & = \text{let} \\
x & = 5 \ast 5 \\
w & = x + y \ast x \\
\text{in} & \\
w & \\
\text{in} & \\
x + y + z
\end{align*}
\]

\[
\begin{align*}
let & \\
y & = 2 \ast 2 \\
x & = 3 + 4 \\
z & = \text{let} \\
x' & = 5 \ast 5 \\
w & = x' + y \ast x' \\
\text{in} & \\
w & \\
\text{in} & \\
x + y + z
\end{align*}
\]
Lexical Scoping and $\alpha$-renaming

\[
\text{plus } x \ y = x + y
\]
\[
\text{plus'} a \ b = a + b
\]

plus and plus' are the same because plus' can be obtained by systematic renaming of bounded identifiers of plus
Capture of Free Variables

\[ f \ x = \ldots \]
\[ g \ x = \ldots \]
\[ \text{foo} \ f \ x = f \ (g \ x) \]

Suppose we rename the bound identifier \( f \) to \( g \) in the definition of \( \text{foo} \)

\[ \text{foo}' \ g \ x = g \ (g \ x) \]

\[ \text{foo} \ \equiv \ \text{foo}' \]

While renaming, entirely new names should be introduced!

September 9, 2015
Curried functions

\[
\text{plus } x \ y = x + y
\]

\[
\text{let } \\
\hspace{1cm} f = \text{plus } 1 \\
\text{in } \\
\hspace{1cm} f \ 3
\]

\[\rightarrow (\text{plus } 1) \ 3 \rightarrow 1 + 3 \rightarrow 4\]

syntactic conventions:
\[
e_1 \ e_2 \ e_3 \equiv ((e_1 \ e_2) \ e_3)\]
\[
x + y \equiv (+) \ x \ y\]
Local Function Definitions

\[
\text{integrate } dx \ a \ b \ f = \\
\text{let} \\
\quad \text{sum } x \ \text{tot} = \\
\qquad \text{if } x > b \ \text{then} \ \text{tot} \\
\qquad \text{else sum } (x+dx) \ (\text{tot}+(f \ x)) \\
\qquad \text{in} \\
\quad (\text{sum } (a+dx/2) \ 0) \ * \ dx
\]

\text{Integral}(a,b) = (f(a + dx/2) + f(a + 3dx/2) + \ldots) \ \Box \ dx
Local Function Definitions

Free variables of `sum`?

Any function definition can be "closed" and "lifted"
All expressions in Haskell have a type

23 :: Int

"23 belongs to the set of integers"
"The type of 23 is Int"

true :: Bool
"hello" :: String
Type of an expression

(sq 529) :: Int

sq :: Int -> Int

"sq is a function, which when applied to an integer produces an integer"

"Int -> Int is the set of functions, each of which when applied to an integer produces an integer"

"The type of sq is Int -> Int"
Type of a Curried Function

\[ \text{plus } x \ y = x + y \]

\((\text{plus } 1)\) \(:=\) Int

\((\text{plus } 1)\) \(:=\) Int \(\to\) Int

\(\text{plus}\) \(:=\) Int \(\to\) (Int \(\to\) Int)
\textbf{\(\lambda\)-Abstraction}

Lambda notation makes it explicit that a value can be a function. Thus,

\begin{equation*}
\text{(plus 1) can be written as } \lambda y \rightarrow (1 + y)
\end{equation*}

(In Haskell \(\lambda x\) is a syntactic approximation of \(\lambda x\))

\begin{equation*}
\text{\texttt{plus x y = x + y}}
\end{equation*}

can be written as

\begin{equation*}
\text{\texttt{plus = \lambda x \rightarrow \lambda y \rightarrow (x + y)}}
\end{equation*}

or as

\begin{equation*}
\text{\texttt{plus = \lambda x \ y \rightarrow (x + y)}}
\end{equation*}
Parentheses Convention

\[ f \ e_1 \ e_2 \equiv ((f \ e_1) \ e_2) \]

\[ f \ e_1 \ e_2 \ e_3 \equiv (((f \ e_1) \ e_2) \ e_3) \]

application is *left associative*

\[
\text{Int} \to (\text{Int} \to \text{Int}) \equiv \text{Int} \to \text{Int} \to \text{Int}
\]

type constructor “\(\to\)” is *right associative*
Type of a Block

\[(\text{let} \quad \begin{align*}
    x_1 &= e_1 \\
    \ldots \\
    x_n &= e_n \\
    \text{in} \\
    e \end{align*}) \quad ::= \quad t \quad \text{provided} \quad e :: t\]
Type of a Conditional

\[(if \ e \ then \ e_1 \ else \ e_2) :: t\]

provided

\[\begin{align*}
e & :: \ Bool \\
e_1 & :: \ t \\
e_2 & :: \ t
\end{align*}\]

The type of expressions in both branches of conditional must be the same.
Polymorphism

twice f x = f (f x)

1. twice (plus 3) 4
   → (Plus 3) ((plus 3) 4)
   → ((plus 3) 7)
   → 10
   twice :: (Int → Int) → Int → Int

2. twice (append "Zha") "Gabor"
   → "ZhaZhaGabor"
   twice :: (Str → Str) → Str → Str
Deducing Types

```
twice f x = f (f x)
What is the most "general type" for twice?
```

1. Assign types to every subexpression

```
x :: t0           f :: t1
f x :: t2     f (f x) :: t3
⇒ twice :: t1 -> t0 -> t3
```

2. Set up the constraints

```
t1 = t0 -> t2       because of (f x)
t1 = t2 -> t3       because of f (f x)
```

3. Resolve the constraints

```
t0 -> t2 = t2 -> t3
⇒ t0 = t2 and t2 = t3 ⇒ t0 = t2 = t3
⇒ twice :: (t0 -> t0) -> t0 -> t0
```
Another Example: \textit{Compose}

\begin{equation*}
\text{compose } f \ g \ x = f \ (g \ x)
\end{equation*}

What is the type of \texttt{compose}?

1. Assign types to every subexpression
   \begin{align*}
   x & : : t_0 & f & : : t_1 & g & : : t_2 \\
   g \ x & : : t_3 & f \ (g \ x) & : : t_4
   \end{align*}
   \[\Rightarrow \text{compose} : : t_1 \rightarrow t_2 \rightarrow t_0 \rightarrow t_4\]

2. Set up the constraints
   \begin{align*}
   t_1 &= t_3 \rightarrow t_4 & \text{because of } f \ (g \ x) \\
   t_2 &= t_0 \rightarrow t_3 & \text{because of } (g \ x)
   \end{align*}

3. Resolve the constraints
   \[\Rightarrow \text{compose} : : (t_3 \rightarrow t_4) \rightarrow (t_0 \rightarrow t_3) \rightarrow t_0 \rightarrow t_4\]
Now for some fun

twice \(\text{f} \ x = \text{f} \ (\text{f} \ x)\)

\[a = \text{twice}_{1} \ (\text{twice}_{2} \ \text{succ}) \ 4\]
\[b = \text{twice}_{3} \ \text{twice}_{4} \ \text{succ} \ 4\]

1. Is \(a=b\) ?
   \[\text{yes} \quad \text{succ} \ (\text{succ} \ (\text{succ} \ (\text{succ} \ 4)))\]
2. Are the types of all the twice instances the same?
   \[\text{no}\]

\[\text{twice}_{1} :: (I \to I) \to I \to I\]
\[\text{twice}_{2} :: (I \to I) \to I \to I\]
\[\text{twice}_{3} :: ((I \to I) \to I \to I) \to (I \to I) \to I \to I\]
\[\text{twice}_{4} :: (I \to I) \to I \to I\]

\[\text{The first person with the right types gets a prize!}\]
Haskell and most modern functional languages follow the Hindley-Milner type system.

The main source of polymorphism in this system is the *Let block*.

The type of a variable can be instantiated differently within its lexical scope.

*much more on this later ...*