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

Armando Solar-Lezama  Adam Chlipala

Co-instructors
What this course is about

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

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)
A revolution in the making

Building a better bug-trap

People who write it are human first and programmers only second—indeed, better software?

June 19th 2003 j from the print edition

Our civilization runs on software. Bjørne Strøm, a programming guru, once observed. Software is everywhere, not just in computers but in household appliances, cars, aeroplanes, lifts, telephones, toys and countless other objects whose purpose is to make our lives more comfortable. It becomes increasingly software-dependent as our lives become more complex.

The Economist

SOFTWARE

Technology Quarterly: Q2 2002

A Matter of Integrity: Tools That Deliver Software Assurance Go Mainstream

By: Paula Biren (News - Alert)

The failure of the levees in New Orleans and the collapse of the I-35W bridge in Minneapolis gave many in the software industry a greater appreciation for the importance of ensuring vital infrastructure is sound. Businesses and organizations would do well to apply these lessons to the area of software development. And many already have.

Software that hasn't been thoroughly vetted can result in leakage in safety and security, customer-serving performance issues and lost revenue—some of the most catastrophic problems a business can have.

Case in point: A major telecommunications company recently was working with a supplier to implement a new CRM software revision for its FTTH network and, as one Internet Telephony Officer - Alert source who asked not to be named put it, 'All hell broke loose.' The CRM system was tied to the telco's core entity system, and the new software release resulted in lost orders. To service providers had to resort to the prior software revision. This little fly in the ointment cost the service provider a two- to three-day in production and a whopping $60 million in revenue.

Firefox code gets vetted

By: Jonas Evers

AUGUST 10, 2006 5:12 PM PDT

The company has licensed Coventry's Prevent to scan the source code of the browser and help detect flaws in the software before its release, Ben Cheif, chief technology officer at Coventry said Thursday. Coventry and Mozilla plan to jointly announce the arrangement on Monday, he said.

Even though the announcement isn't coming until Monday, Mozilla actually licensed the Coventry tool about a year and a half ago, Cheif said. The companies held off on the announcement until Mozilla felt comfortable with the product and it actually yielded some results, he said.
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
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)
An Experiment This Term

We'll try using the Coq proof assistant for some lectures/homeworks to get **machine-checking** of our proofs.

(Turn math into something more like programming, and catch proof bugs in the process.)
Grading

6 homework assignments
- Each is 15-20% of your grade (details TBD)
- 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
- Practice writing proofs in Coq
- Implement a type inference engine

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

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

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

Pset 6
- Practice LTL and CTL (two specification languages)
- Learn how to use a model checker
- Fun with synthesis
And now the fun part...
Functional Programming:
Functions and Types

Adam Chlipala
MIT

Adapted from Solar-Lezama's 2011 slides,
which were adapted from Arvind 2010

September 4, 2013
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.

In other words: execution is “just algebra” (contrast with, e.g., Java!).

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

A formal definition will be given later.
Blocks

```
let
  x = a * a
  y = b * b
in
  (x - y) / (x + y)
```

- a variable can have at most one definition in a block
- ordering of bindings does not matter
Layout Convention in Haskell

This convention allows us to omit many delimiters.

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

is the same as

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

```
let
  y = 2 * 2
  x = 3 + 4
let
  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)

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

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

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

\textit{plus} and \textit{plus'} are the same because \textit{plus'} can be obtained by \textit{systematic renaming of bound identifiers} of \textit{plus}. 
Capture of Free Variables

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

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

\[
\text{foo} \equiv \text{foo}' \quad ? \quad \text{No}
\]

While renaming, entirely new names should be introduced!
Curried Functions

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

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

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

syntactic conventions:

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

\[
x + y \equiv (+) \ x \ y
\]
Local Function Definitions

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

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

Any function definition can be “closed” and “lifted.”
Types

All expressions in Haskell have types.

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

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

\[(\text{plus } 1) \ 3 \quad :: \ \text{Int} \]

\[(\text{plus } 1) \quad :: \ \text{Int} \rightarrow \text{Int} \]

\[\text{plus} \quad :: \ \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}) \quad ?\]
Lambda notation makes it explicit that a value can be a function. Thus,

(\textit{plus 1}) can be written as $\lambda y \rightarrow (1 + y)$

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

\texttt{plus x y = x + y}

can be written as

\texttt{plus} = $\lambda x \rightarrow \lambda y \rightarrow (x + y)$

or as

\texttt{plus} = $\lambda x \ y \rightarrow (x + y)$
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 \textit{left associative}

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

type constructor “\rightarrow” is \textit{right associative}
Type of a Block

\[
\begin{align*}
(let \\
\quad x_1 & = e_1 \\
\quad \cdot & \\
\quad \cdot & \\
\quad x_n & = e_n \\
\quad in & \\
\quad e & ) :: t
\end{align*}
\]

provided

\[
\begin{align*}
\quad e_i & :: t_i
\end{align*}
\]

and

assuming

\[
\begin{align*}
\quad x_i & :: t_i
\end{align*}
\]
Type of a Conditional

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

provided

\[e :: \text{Bool} \]
\[e_1 :: t \]
\[e_2 :: t \]

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

1. Assign a type to every subexpression
   \[ x :: \text{t}_0 \quad \text{f} :: \text{t}_1 \]
   \[ \text{f x} :: \text{t}_2 \quad \text{f (f x)} :: \text{t}_3 \]
   \[ \Rightarrow \text{twice} :: \text{t}_1 \rightarrow \text{t}_0 \rightarrow \text{t}_3 \]

2. Set up the constraints
   \[ \text{t}_1 = \text{t}_0 \rightarrow \text{t}_2 \quad \text{because of (f x)} \]
   \[ \text{t}_1 = \text{t}_2 \rightarrow \text{t}_3 \quad \text{because of f (f x)} \]

3. Resolve the constraints
   \[ \text{t}_0 \rightarrow \text{t}_2 = \text{t}_2 \rightarrow \text{t}_3 \]
   \[ \Rightarrow \text{t}_0 = \text{t}_2 \quad \text{and} \quad \text{t}_2 = \text{t}_3 \Rightarrow \text{t}_0 = \text{t}_2 = \text{t}_3 \]
   \[ \Rightarrow \text{twice} :: (\text{t}_0 \rightarrow \text{t}_0) \rightarrow \text{t}_0 \rightarrow \text{t}_0 \]
Another Example: *Compose*

```haskell
compose f g x = f (g x)
```

What is the type of `compose`?

1. Assign a type to every subexpression.
   
   ```
   x :: t0    f :: t1    g :: t2
   g x :: t3    f (g x) :: t4
   \Rightarrow compose :: t1 -> t2 -> t0 -> t4
   ```

2. Set up the constraints
   
   ```
   t1 = t3 --> t4 because of f (g x)
   t2 = t0 --> t3 because of (g x)
   ```

3. Resolve the constraints
   
   ```
   \Rightarrow compose ::
   (t3 --> t4) --> (t0 --> t3) --> t0 --> t4
   ```
Now for some fun

\[
\text{\texttt{twice}} \ f \ x = f \ (f \ x)
\]
\[
a = \text{\texttt{twice}}_1 \ (\text{\texttt{twice}}_2 \ \text{\texttt{succ}}) \ 4
\]
\[
b = \text{\texttt{twice}}_3 \ \text{\texttt{twice}}_4 \ \text{\texttt{succ}} \ 4
\]

1. Is \ a = b ?
   - yes  \ \text{\texttt{succ}} \ (\text{\texttt{succ}} \ (\text{\texttt{succ}} \ (\text{\texttt{succ}} \ 4))

2. Are the types of all the \texttt{twice} instances the same?
   - no

- \text{\texttt{twice}}_1 :: (I \to I) \to I \to I
- \text{\texttt{twice}}_2 :: (I \to I) \to I \to I
- \text{\texttt{twice}}_3 :: ((I \to I) \to I \to I) \to (I \to I) \to I \to I
- \text{\texttt{twice}}_4 :: (I \to I) \to I \to I
Hindley-Milner Type Systems

Haskell and most modern functional languages follow the **Hindley-Milner** type system approach.

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 ...*