CMSC330

PDAs, turing machines, types and bindings

Last lecture

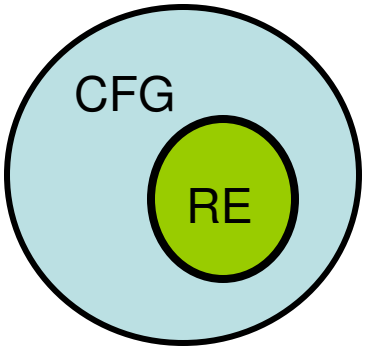
- Parsing
  - Recursive descent
  - FIRST sets
- Rewriting grammars
  - Left factoring
  - Eliminating left recursion
- Question
  - Where do regular expressions and context free grammars stand in the world of complexity?

This lecture part 1

- Regular grammars
- Pushdown automata
- Turing machines
- Decidability
- Turing tests

Regular Expressions and CFGs

- CFGs are strictly more powerful than REs
  - Since CFGs are superset of regular grammar
- This is good, since programming languages are not regular
  - No regular expression for $(^n)^n$
- But programming languages are usually (almost) context-free
  - With a few language constructs being exceptions



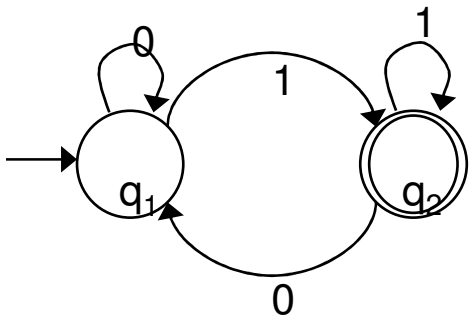
Regular Expressions and CFGs

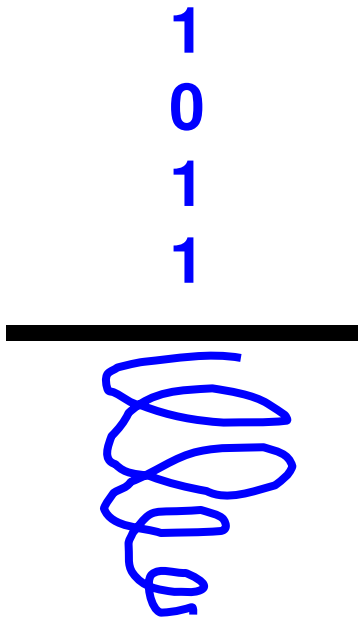
	Description	Machine
regular languages	regular expressions	DFAs, NFAs
context-free languages	context-free grammars	pushdown automata (PDAs)

- We can implement REs with DFAs
- Question
  - How can we implement CFGs?

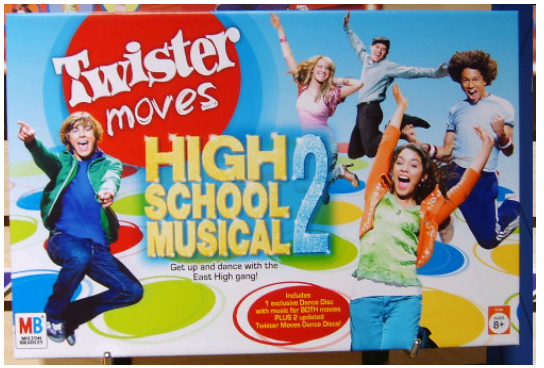
Push Down Automata

- A PDA is an abstract machine similar to a DFA
  - Has a finite set of states
  - But also has a stack





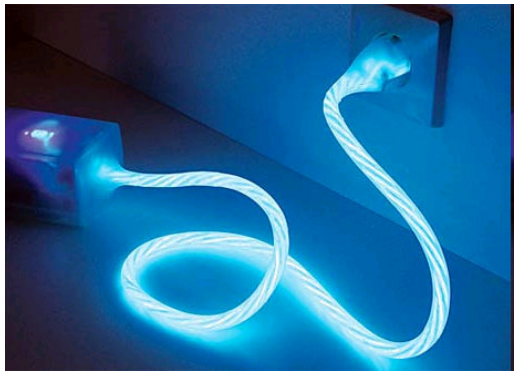
Moves of a PDA



- Reads
  - An input symbol is read
  - Top symbol on the stack
- Based on both inputs, the machine
  - Enters a new state, and
  - Writes zero or more symbols onto the pushdown stack
  - Or pops zero or more symbols from the stack
- String accepted if
  - The stack is empty, AND
  - The string has ended

Power of PDAs

- PDAs are more powerful than DFAs
  - $a^n b^n$, which cannot be recognized by a DFA, can easily be recognized by the PDA
    - Stack all a symbols and, for each b, pop an a off the stack.
    - If the end of input is reached at the same time that the stack becomes empty, the string is accepted
- PDAs can recognize some L(CFG)
  - Table-driven predictive parser
    - Table = Finite automaton
    - Stack = Stack
  - Limited to LR(1) grammars in general

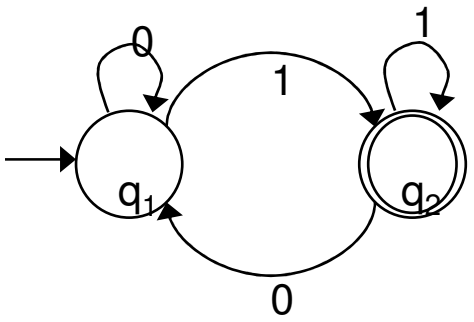


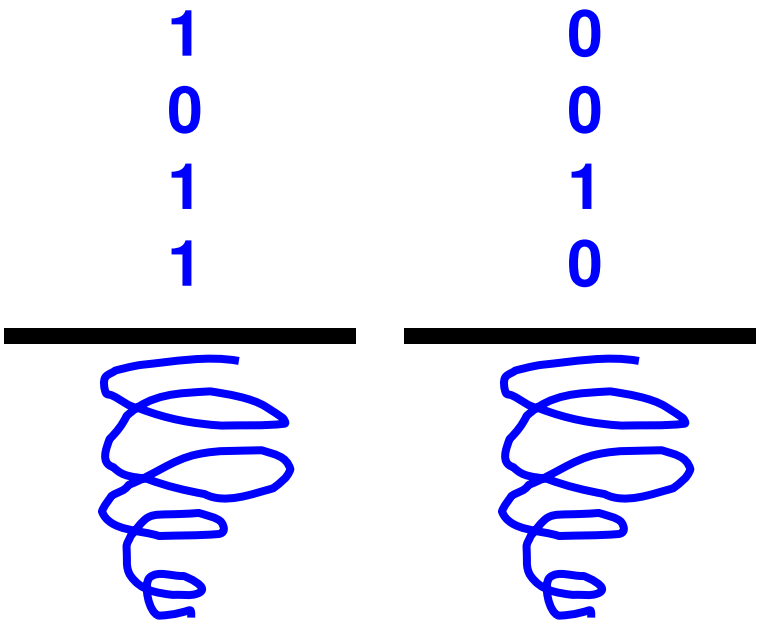
Power of PDAs and NDPDA

- There are also nondeterministic PDA (NDPDA)
  - NDPDA are more powerful than DPDA
  - NDPDA can recognize even length palindromes over {0,1}*, but a DPDA cannot. Why?
    - Hint: Consider palindromes over {0,1}*2{0,1}* vs {0,1}* {0,1}*
- NDPDAs can recognize all L(CFG)
  - Equivalent in power to context-free languages

Build a More Powerful PDA?

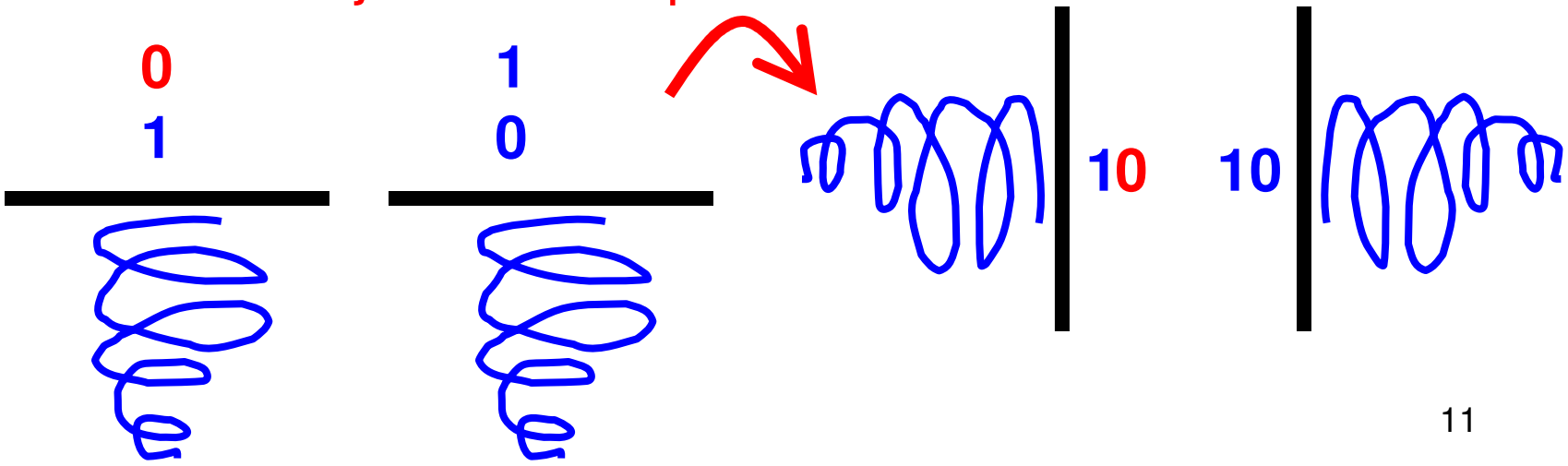
- What if we add another stack to the PDA?
- A 2-stack PDA
  - Has a finite set of states
  - Plus 2 stacks
- How powerful is this?





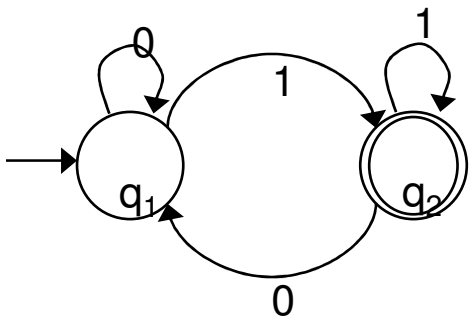
Build a More Powerful PDA?

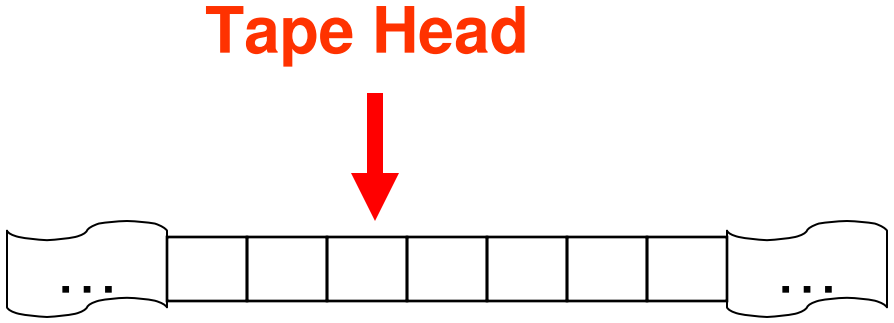
- Answer – very powerful!
- Trick
  - Flip two stacks so they face each other
  - Pop a symbol off stack 1, push it right back onto stack 2
  - 2 stacks is just like a tape!



Turing Machine

- Defined by Alan Turing in 1936
- Finite state machine + tape
- Tape
  - Infinite storage
  - Read / write one symbol at tape head
  - Move tape head one space left / right

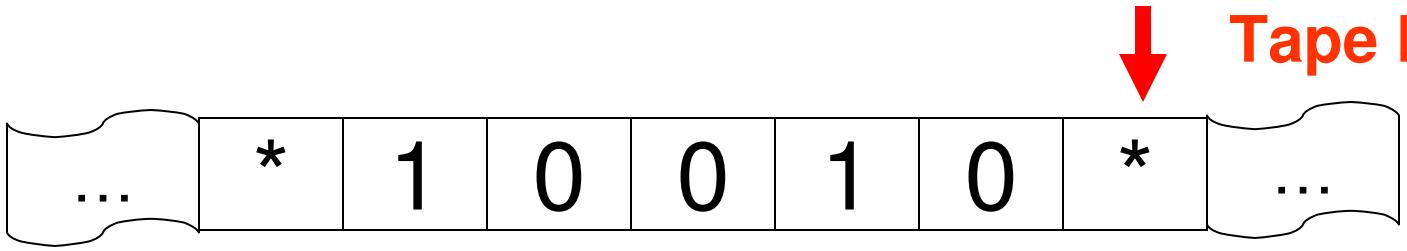




Turing Machine

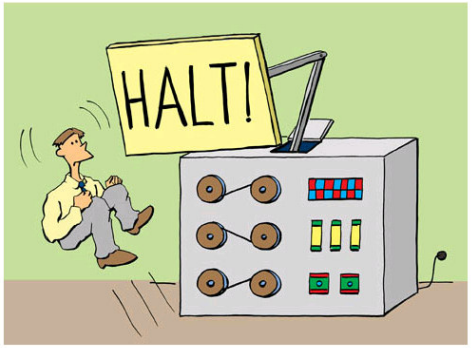
- Allowable actions
  - Read symbol from current square
  - Write symbol to current square
  - Move tape head left
  - Move tape head right
  - Go to next state

Turing Machine



Current State	Current Content	Value to Write	Direction to Move	New state to enter
START	*	*	Left	MOVING
MOVING	1	0	Left	MOVING
MOVING	0	1	Left	MOVING
MOVING	*	*	No move	HALT

Turing Machine



Alan designed the perfect computer

- Operations
  - Read symbol on current square
  - Select action based on symbol & current state
  - Accept string if in accept state
  - Reject string if halts in non-accepting state
  - Reject string if computation does not terminate

- Halting problem
  - It is undecidable in general whether long-running computations will eventually accept

“Enhanced” Turing Machines

- Attempts to increase power of Turing machine
  - Additional tapes
  - Additional tape heads
  - 2-dimensional tape
  - BlueGene/L @ Livermore Lab
    - World’s fastest computer in 2007
    - 106K IBM PowerPC nodes (596 x10^{12} Flops)

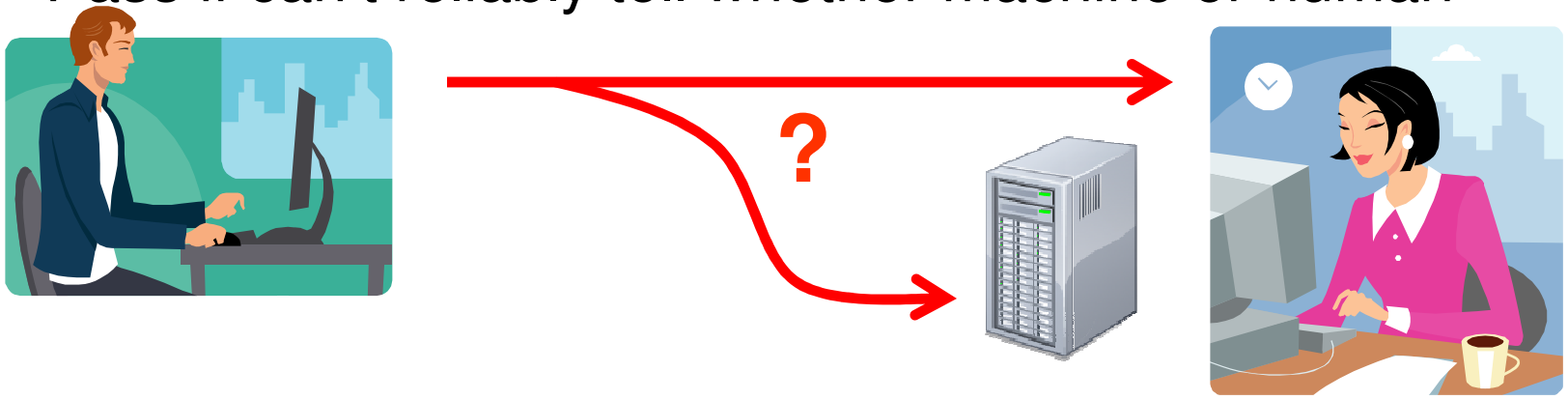
- Turns out any computation on enhanced machine
  - Can be computed on standard Turing machine
  - Though may require more time

Computability

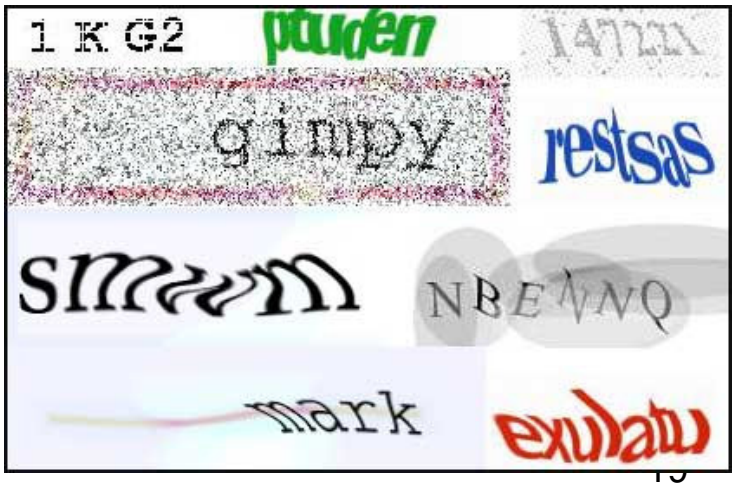
- Computability
  - A language is computable if it can be recognized by some algorithm with finite number of steps
- Church-Turing thesis
  - Turing machine can recognize any language computable on any machine
- Intuition
  - Turing machine captures essence of computing
    - Both in a formal sense, and in an informal practical sense

Turing Test

- Turing Test
  - Test machine's capability to demonstrate intelligence
  - Proposed by Alan Turing in 1950
  - Practical test for "Can machines think?"
- Test procedure
  - Judge converses via text (e.g., instant messaging)
  - Pass if can't reliably tell whether machine or human



- Reverse Turing test
  - Computer determines whether human or machine
- CAPTCHA
  - Completely Automated Public Turing test to tell Computers and Humans Apart
  - Challenge-response asking for word identification
    - Useful for foiling scripts



This lecture part 2

- Types
- Bindings

Language Features Covered Thus Far

- Ruby
  - Implicit declarations — { x = 1 }
  - Dynamic typing — { x = 1 ; x = “foo” }
- OCaml
  - Functional programming — add 1 (add 2 3)
  - Type inference — let x = x+1 (x : int)
  - Higher-order functions — let rec x = fun y -> x y
  - Static (lexical) scoping — let x = let x = …
  - Parametric polymorphism — let x y = y (‘a -> ‘a)
  - Modules — module foo struct … end

Type Programming Language Features

- Type system
  - Typed vs. untyped
  - Static vs. dynamic
  - Weak vs. strong (type safe)

Programming Language Features (cont.)

- Names & binding
  - Namespaces
  - Static (lexical) scopes
  - Dynamic scopes

Type vs. Untyped Languages

- Typed language
  - Operations are only valid for specified types
    - `2 * 3 = 6`
    - `"foo" * "bar"` = undefined
  - Helps catch program errors
    - Either at compile or run time
- Untyped language
  - All operations are valid for all values
  - Treat all values as sequences of 0's and 1's
  - Example
    - Assembly languages, FORTH

Static vs. Dynamic Types

- Static types
  - Before program is run
    - Type of all expressions are determined
      - Usually by compiler
    - Disallowed operations cause compile-time error

- Static types may be manifest or inferred
  - Manifest – specified in text (at variable declaration)
    - C, C++, Java, C#
  - Inferred – compiler determines type based on usage
    - ML, OCaml

Static vs. Dynamic Types (cont.)

- Dynamic types
  - While program is running
    - Type of all expressions determined
      - Values maintain tag indicating type
    - Disallowed operations cause run-time exception

- Dynamic types are not manifest (obviously)
  - Examples
    - Ruby, Python, Javascript, Lisp

Weak vs. Strong Typing

- Weak typing
  - Allows one type to be treated as another
  - …or provides (many) implicit casts
    - C

```
int i = 1 ;
if (i) // checks for 0
    printf("%d", i);
```

    - Ruby

```
i = 1
if i  // checks for nil
    puts i
end;
```

  - Examples
    - C, C++, Ruby, Perl, Javascript

Weak vs. Strong Typing (cont.)

- Strong typing
  - Prevents one type from being treated as another
  - Also known as type-safe
    - Java

```
int i = 1 ;
if (i) // error, not bool
    System.out.println(i);
```

    - OCaml

```
let i = 1 in
    if i then // error, not bool
        print_int i
```

  - Examples
    - Java, OCaml

Weak/Strong vs. Static/Dynamic Types

- How do these properties interact?
  - Weak/strong & static/dynamic are orthogonal
  - Some literature confuse strong & static type
- Strong / static types
  - More work for programmer
  - Catches more errors at compile time
- Weak / dynamic types
  - Less work for programmer
  - More errors occur at run time

Names & Binding Overview

- Order of bindings
- Namespaces
- Static (lexical) scopes
- Dynamic scopes
- Funargs

Names and binding

- Programs use names to refer to things
  - E.g., in `x = x + 1`, x refers to a variable

- A binding is an association between a name and what it refers to
  - `int x;  /* x is bound to a stack location containing an int */`
  - `int f (int){...} /* f is bound to a function */`
  - `class C {...}  /* C is bound to a class */`
  - `let x = e1 in e2 (* x is bound to e1 *)`

Name Restrictions

- Languages often have various restrictions on names to make scanning and parsing easier
  - Names cannot be the same as keywords in the language
  - OCaml function names must be lowercase
  - OCaml type constructor and module names must be uppercase
  - Names cannot include special characters like ; , : etc
    - Usually names are upper- and lowercase letters, digits, and _ (where the first character can't be a digit)
    - Some languages also allow more symbols like ! or -

Names and Scopes

- Good names are a precious commodity
  - They help document your code
  - They make it easy to remember what names correspond to what entities
- We want to be able to reuse names in different, non-overlapping regions of the code

Names and Scopes (cont.)

- A scope is the region of a program where a binding is active
  - The same name in a different scope can refer to a different binding (refer to a different program object)

- A name is in scope if it's bound to something within the particular scope we're referring to

Example

```
void w(int i) {
  ...
}

void x(float j) {
  ...
}

void y(float i) {
  ...
}

void z(void) {
   int j;
   char *i;
   ...
}
```

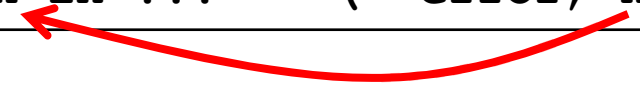
- i is in scope
  - in the body of w, the body of y, and after the declaration of j in z
  - but all those i's are different
- j is in scope
  - in the body of x and z

Order of Bindings – OCaml

- let x = e1 in e2 – x is bound to e1 in scope of e2
- let rec x = e1 in e2 – x is bound in e1 and in e2

```
let x = 3 in
  let y = x + 3 in...    (* x is in scope here *)
```

```
let x = 3 + x in ...     (* error, x not in scope *)
```



```
let rec length = function
    [] -> 0
  | (h::t) -> 1 + (length t)   (* ok, length in scope *)
in ...
```

Order of Bindings – C

- All declarations are in scope from the declaration onward

```
int i;
int j = i;  /* ok, i is in scope */
i = 3;      /* also ok */
```

```
void f(...) { ... }

int i;
int j = j + 3;   /* error */
f(...);          /* ok, f declared */
```

```
f(...);  /* may be error; need prototype (or oldstyle C) */

void f(...) { ... }
```

Order of Bindings – Java

- Declarations are in scope from the declaration onward, except for methods and fields, which are in scope throughout the class

```
class C {
  void f(){
    ...g()...   // OK
  }

  void g(){
    ...
  }
}
```

Shadowing Names

- Shadowing is rebinding a name in an inner scope to have a different meaning
  - May or may not be allowed by the language

C

```
int i;

void f(float i) {

  {
    char *i = NULL;
    ...
  }

}
```

OCaml

```
let g = 3;;
let g x = x + 3;;
```

Java

```
void h(int i) {
  {
    float i; // not allowed
    ...
  }
}
```

Namespaces

- Languages have a “top-level” or outermost scope
  - Many things go in this scope; hard to control collisions
- Common solution seems to be to add a hierarchy
  - OCaml: Modules
    - List.hd, String.length, etc.
    - open to add names into current scope
  - Java: Packages
    - java.lang.String, java.awt.Point, etc.
    - import to add names into current scope
  - C++: Namespaces
    - namespace f { class g { ... } }, f::g b, etc.
    - using namespace to add names to current scope

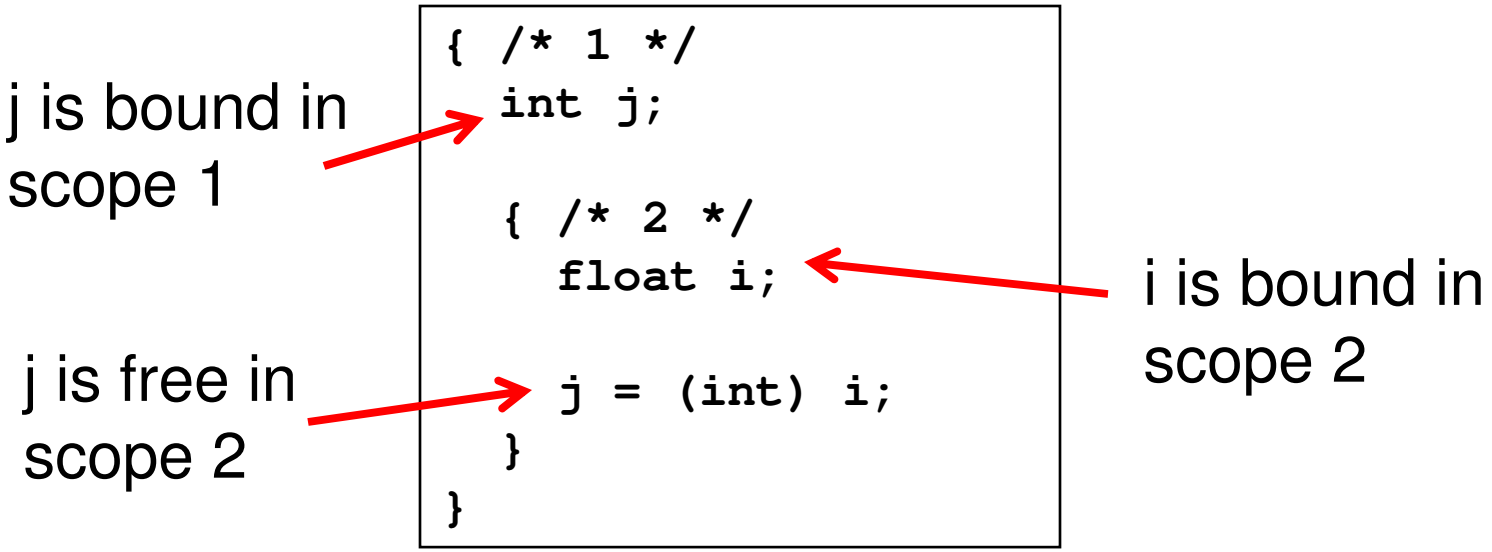
Static Scope Recall

- In static scoping, a name refers to its closest binding, going from inner to outer scope in the program text
  - Languages like C, C++, Java, Ruby, and OCaml are statically scoped

```
int i;

{
  int j;

  {
    float i;

    j = (int) i;
  }
}
```

Free and Bound Variables

- The bound variables of a scope are those names that are declared in it
- If a variable is not bound in a scope, it is free
  - The bindings of variables which are free in a scope are inherited from declarations of those variables in outer scopes in static scoping

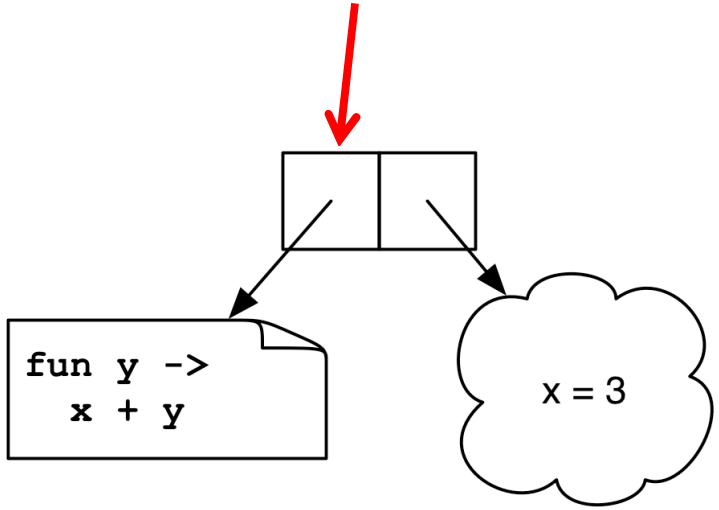


Static Scoping and Nested Functions

- To allow arbitrary nested functions with higher-order functions and static scoping, we needed closures

```
let add x = (fun y -> x + y)
```

```
(add 3) 4    → <closure> 4    → 3 + 4    → 7
```



Functional Arguments (Funargs)

- Funarg problem
  - Difficult to implement functions as first-class objects in stack-based programming languages
- Two solutions:
  - Forbid such references
  - Closures
- Types:
  - Upwards: returning a function
  - Downwards: passing a function as an argument

Dynamic Scope

- In a language with dynamic scoping, a name refers to its closest binding at runtime
  - LISP was the common example

Nested Dynamic Scopes

- Full dynamic scopes can be nested
  - Static scope relates to the program text
  - Dynamic scope relates to program execution trace

Static vs. Dynamic Scope

Static scoping

- Local understanding of function behavior
- Know at compile-time what each name refers to
- A little more work to implement (keep a link to the lexical nesting scope in stack frame)

Dynamic scoping

- Can be hard to understand behavior of functions
- Requires finding name bindings at runtime
- Easier to implement (keep a global table of stacks of variable/value bindings)