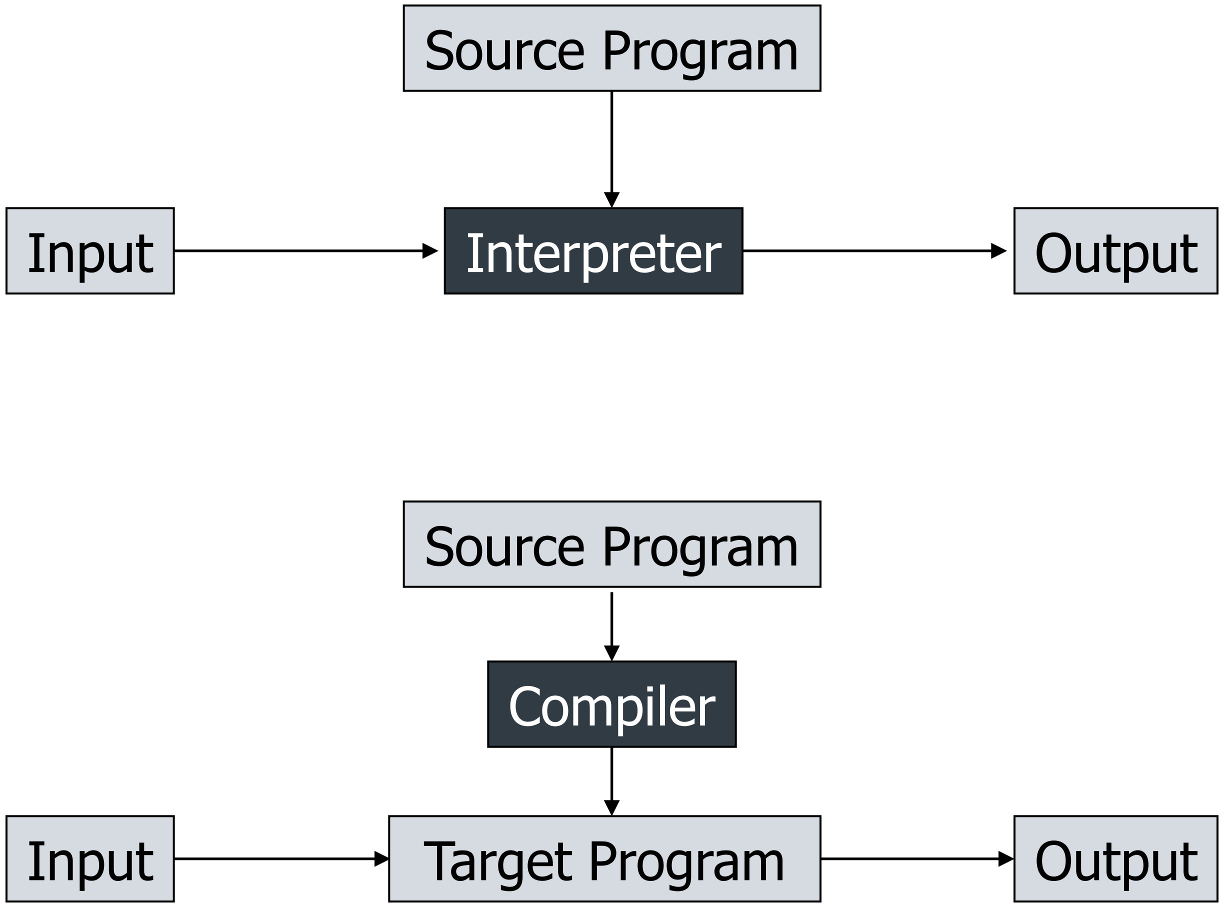
Fundamentals

Reading: See last slide

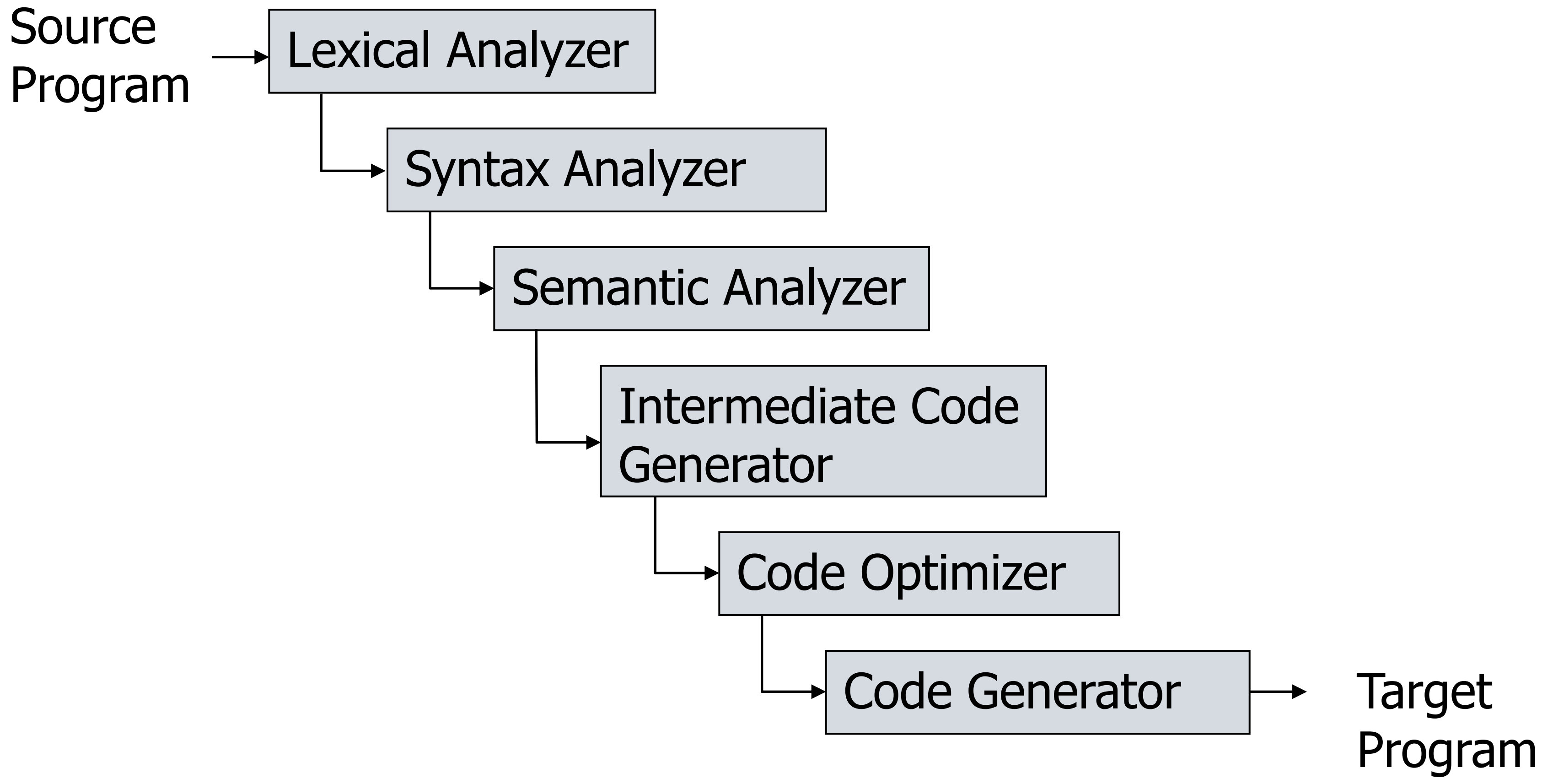
Syntax and Semantics of Programs

- Syntax
  - The symbols used to write a program
- Semantics
  - The actions that occur when a program is executed
- Programming language implementation
  - Syntax $\rightarrow$ Semantics
  - Transform program syntax into machine instructions that can be executed to cause the correct sequence of actions to occur

Interpreter vs Compiler



Typical Compiler



See summary in course text, compiler books

Brief look at syntax

- Grammar

```
e ::= n | e+e | e–e
n ::= d | nd
d ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

- Expressions in language

e → e–e → e–e+e → n–n+n → nd–d+d → dd–d+d

→ … → 27 – 4 + 3

Grammar defines a language
Expressions in language derived by sequence of productions

Many of you are familiar with this to some degree

Theoretical Foundations

- Many foundational systems
  - Computability Theory
  - Program Logics
  - Lambda Calculus
  - Denotational Semantics
  - Operational Semantics
  - Type Theory
- Consider some of these methods
  - Computability theory (halting problem)
  - Lambda calculus (syntax, operational semantics)
  - Operational semantics (not in book)

Lambda Calculus

- Formal system with three parts
  - Notation for function expressions
  - Proof system for equations
  - Calculation rules called *reduction*
- Additional topics in lambda calculus (not covered)
  - Mathematical semantics (=model theory)
  - Type systems

We will look at syntax, equations and reduction

There is more detail in the book than we will cover in class

History

- Original intention
  - Formal theory of substitution (for FOL, etc.)
- More successful for computable functions
  - Substitution --> symbolic computation
  - Church/Turing thesis
- Influenced Lisp, Haskell, other languages
  - See Boost Lambda Library for C++ function objects
    - http://www.boost.org/doc/libs/1_51_0/doc/html/lambda.html
- Important part of CS history and foundations

Why study this now?

- Basic syntactic notions
  - Free and bound variables
  - Functions
  - Declarations
- Calculation rule
  - Symbolic evaluation useful for discussing programs
  - Used in optimization (in-lining), macro expansion
    - Correct macro processing requires variable renaming
  - Illustrates some ideas about scope and binding
    - Lisp originally departed from standard lambda calculus, returned to the fold through Scheme, Common Lisp
    - Haskell, JavaScript reflect traditional lambda calculus

Expressions and Functions

- Expressions

  $x + y \qquad x + 2*y + z$

- Functions

  $\lambda x.\ (x+y) \qquad \lambda z.\ (x + 2*y + z)$

- Application

  $(\lambda x.\ (x+y))\ 3 = 3 + y$

  $(\lambda z.\ (x + 2*y + z))\ 5 = x + 2*y + 5$

Parsing: $\lambda x.\ f\ (f\ x) = \lambda x.(\ f\ (f\ (x))\ )$

Higher-Order Functions

- Given function f, return function f ∘ f

  $\lambda f.\ \lambda x.\ f\ (f\ x)$

- How does this work?

$$(\lambda f.\ \lambda x.\ f\ (f\ x))\ \boxed{(\lambda y.\ y+1)}$$

$$=\ \lambda x.\ (\lambda y.\ y+1)\ ((\lambda y.\ y+1)\ \boxed{x})$$

$$=\ \lambda x.\ (\lambda y.\ y+1)\ \boxed{(x+1)}$$

$$=\ \lambda x.\ (x+1)+1$$

In pure lambda calculus, same result if step 2 is altered.

Same procedure, Lisp syntax

- Given function f, return function f ∘ f

```
(lambda (f) (lambda (x) (f (f x))))
```

- How does this work?

```
((lambda (f) (lambda (x) (f (f x))))  (lambda (y) (+ y 1))

= (lambda (x) ((lambda (y) (+ y 1))
                  ((lambda (y) (+ y 1)) x))))

=  (lambda (x) ((lambda (y) (+ y 1)) (+ x 1))))

= (lambda (x) (+ (+ x 1) 1))
```

Same procedure, JavaScript syntax

- Given function f, return function f ∘ f

```
function (f) { return function (x) { return f(f(x)); }; }
```

- How does this work?

```
(function (f) { return function (x) { return f(f(x)); };})
                  (function (y) { return y +1; })

 function (x) { return (function (y) { return y +1; })
                          ((function (y) { return y + 1; }) (x)); }


function (x) { return (function (y) { return y +1; }) (x + 1); }


function (x) { return ((x + 1) + 1); }
```

Declarations as "Syntactic Sugar"

```
function f(x) {
    return x+2;
}
f(5);
```

$$\underbrace{(\lambda f.\ \ f(5))}_{\text{block body}}\ \ \underbrace{(\lambda x.\ x+2)}_{\text{declared function}}$$

Declaration form used in ML, Haskell:

$$\text{let } x = e_1 \text{ in } e_2 \quad = \quad (\lambda x.\ \ e_2)\ \ e_1$$

Free and Bound Variables

- Bound variable is "placeholder"
  - Variable x is bound in $\lambda x.\ (x+y)$
  - Function $\lambda x.\ (x+y)$ is same function as $\lambda z.\ (z+y)$
- Compare

  $\int x+y\ dx\ =\ \int z+y\ dz \qquad \forall x\ \ P(x) = \forall z\ \ P(z)$
- Name of free (=unbound) variable does matter
  - Variable y is free in $\lambda x.\ (x+y)$
  - Function $\lambda x.\ (x+y)$ is *not* same as $\lambda x.\ (x+z)$
- Occurrences
  - y is free and bound in $\lambda x.\ ((\lambda y.\ y\underbrace{+2)\ x)})+ y$

Reduction

- Basic computation rule is β-reduction

$$(\lambda x.\ e_1)\ e_2 \quad \rightarrow \quad [e_2/x]e_1$$

  where substitution involves renaming as needed

  (next slide)

- Reduction:
  - Apply basic computation rule to any subexpression
  - Repeat
- Confluence:
  - Final result (if there is one) is uniquely determined

Rename Bound Variables

- Function application

$$\underbrace{(\lambda f.\ \lambda x.\ f\ (f\ x))}_{\text{apply twice}}\ \ \underbrace{(\lambda y.\ y+x)}_{\text{add x to argument}}$$

◆Substitute "blindly"

$$\lambda x.\ [(\lambda y.\ y+x)\ ((\lambda y.\ y+x)\ x)]\ \ =\ \ \lambda x.\ x+x+x$$

◆Rename bound variables

$$(\lambda f.\ \lambda z.\ f\ (f\ z))\ \ (\lambda y.\ y+x)$$

$$=\ \ \lambda z.\ [(\lambda y.\ y+x)\ ((\lambda y.\ y+x)\ z))]\ \ =\ \ \lambda z.\ z+x+x$$

Easy rule: always rename variables to be distinct

Main Points about Lambda Calculus

- λ captures "essence" of variable binding
  - Function parameters
  - Declarations
  - Bound variables can be renamed
- Succinct function expressions
- Simple symbolic evaluator via substitution
- Can be extended with
  - Types
  - Various functions
  - Stores and side-effects

  (But we didn't cover these)

Announcements

- Homework due Wed 5PM
  - Most problems on paper
  - Upload Haskell programs using CourseWare
    - Instructions will be posted as CourseWare Announcement
- Homework grading Thurs 5:30 – 8:30 PM
  - Send email to cs242fall2012@cs.stanford.edu

Operational Semantics

- Abstract definition of program execution
  - Sequence of actions, formulated as transitions of an abstract machine
- States corresponds to
  - Expression/statement being evaluated/executed
  - Abstract description of memory and other data structures involved in computation

Structural Operational Semantics

- Systematic definition of operational semantics
  - Specify the transitions in a syntax oriented manner using the inductive nature of program syntax
- Example
  - The state transition for e1 + e2 is described using the transitions for e1 and the transition for e2
- Plan
  - SOS of a simple subset of JavaScript
  - Summarize scope, prototype lookup in JavaScript

Simplified subset of JavaScript

- Three syntactic categories
  - Arith expressions : a ::= n | X | a + a | a * a
  - Bool expressions : b ::= a<=a | not b | b and b
  - Statements : s ::= skip | x = a | s; s | if b then s else s | while b do s
- States
  - Pair S = $\langle$ t , σ $\rangle$
  - t : syntax being evaluated/executed
  - σ : abstract description of memory, in this subset a function from variable names to values, i.e., σ : Var $\rightarrow$ Values

Sample operational rules

A rule for Arithmetic Expressions

$$\frac{\langle a_1, \sigma\rangle \longrightarrow \langle a_1', \sigma\rangle}{\langle a_1 + a_2, \sigma\rangle \longrightarrow \langle a_1' + a_2, \sigma\rangle}[\mathtt{A_{3a}}] \qquad \frac{\langle a_2, \sigma\rangle \longrightarrow \langle a_2', \sigma\rangle}{\langle n + a_2, \sigma\rangle \longrightarrow \langle n + a_2', \sigma\rangle}[\mathtt{A_{3b}}]$$

How to interpret this rule ?

- If the term a_1 partially evaluates to a_1' then $a_1 + a_2$ partially evaluates to $a_1' + a_2$.
- Once the expression a_1 reduces to a value n, then start evaluating a_2

Example :

$$\langle (10 + 12) + (13 + 20), \sigma\rangle \xrightarrow{A_{3a}} \langle 22 + (13 + 20), \sigma\rangle \xrightarrow{A_{3b}} \langle 22 + 33, \sigma\rangle$$

Sample rules

A rule for Statements

$$\frac{\langle a, \sigma\rangle \rightarrow \langle a', \sigma'\rangle}{\langle \mathtt{x} := \mathtt{a}, \sigma\rangle \rightarrow \langle x = a', \sigma'\rangle}[\mathrm{C}_3] \quad \frac{\sigma' = Put(\sigma, x, n)}{\langle \mathtt{x} := \mathtt{n}, \sigma\rangle \rightarrow \langle \sigma'\rangle}[\mathrm{C}_2]$$

How to interpret this rule ?

- If the arithmetic exp a partially evaluates to a' then the statement $x = a$ partially evaluates to $x = a'$.
- Rule C_2 applies when a reduces to a value n.
- $Put(\sigma, x, n)$ updates the value of x to n.

Example : $\langle(x := 10 + 12, \sigma\rangle \xrightarrow{C_3} \langle x := 22, \sigma\rangle \xrightarrow{C_2} \langle\sigma'\rangle$

Form of SOS

General form of transition rule:

$$\frac{P_1, \ldots, P_n}{\langle t, \sigma \rangle \to \langle t', \sigma' \rangle} \qquad \frac{P_1, \ldots, P_n}{\langle t, \sigma \rangle \to \sigma'} \tag{1}$$

$P_1, \ldots, P_n$ are the conditions that must hold for the transition to go through. Also called the premise for the rule. These could be

- Other transitions corresponding to the sub-terms.
- Predicates that must be true.
- Calls to meta functions like :
  - $get(\sigma, x) = v$: Fetch the value of x.
  - $put(\sigma, x, n) = \sigma'$: Update value of x to n and return new store.

Conditional and loops

If Then Else

$$\langle \texttt{if tt then s}_1 \texttt{ else s}_2, \sigma\rangle \longrightarrow \langle \texttt{s}_1, \sigma\rangle [\texttt{C}_{5a}]$$

$$\langle \texttt{if ff then s}_1 \texttt{ else s}_2, \sigma\rangle \longrightarrow \langle \texttt{s}_2, \sigma\rangle [\texttt{C}_{5b}]$$

$$\frac{\langle \texttt{b}, \sigma\rangle \longrightarrow \langle \texttt{b}', \sigma\rangle}{\langle \texttt{if b then s}_1 \texttt{ else s}_2, \sigma\rangle \longrightarrow \langle \texttt{ifb}' \texttt{ then s}_1 \texttt{ else s}_2, \sigma\rangle} [\texttt{C}_{5c}]$$

While

$$\langle \texttt{while b do s}, \sigma\rangle \longrightarrow$$
$$\langle \texttt{if b then s; while b s else skip end}, \sigma\rangle [\texttt{C}_6]$$

Context Sensitive Rules

Similar rules

$$\frac{\langle a_1, \sigma\rangle \longrightarrow \langle a_1', \sigma\rangle}{\langle a_1 + a_2, \sigma\rangle \longrightarrow \langle a_1' + a_2, \sigma\rangle}[\mathtt{A}_{3\mathtt{a}}] \qquad \frac{\langle a_2, \sigma\rangle \longrightarrow \langle a_2', \sigma\rangle}{\langle n + a_2, \sigma\rangle \longrightarrow \langle n + a_2', \sigma\rangle}[\mathtt{A}_{3\mathtt{b}}]$$

$$\frac{\langle a_1, \sigma\rangle \longrightarrow \langle a_1', \sigma\rangle}{\langle a_1 * a_2, \sigma\rangle \longrightarrow \langle a_1' * a_2, \sigma\rangle}[\mathtt{A}_{4\mathtt{a}}] \qquad \frac{\langle a_2, \sigma\rangle \longrightarrow \langle a_2', \sigma\rangle}{\langle n * a_2, \sigma\rangle \longrightarrow \langle n * a_2', \sigma\rangle}[\mathtt{A}_{4\mathtt{b}}]$$

- The above rules have a similar premise :
- Combine them into a single rule of the following form :

$$\frac{\langle a, \sigma\rangle \longrightarrow \langle a', \sigma\rangle}{AC(a) \longrightarrow AC(a')}$$

where AC :: $\_ | \_ + \mathtt{a} | \mathtt{n} + \_ | \_ * \mathtt{a} | \mathtt{n} * \_$

Moving to full JavaScript

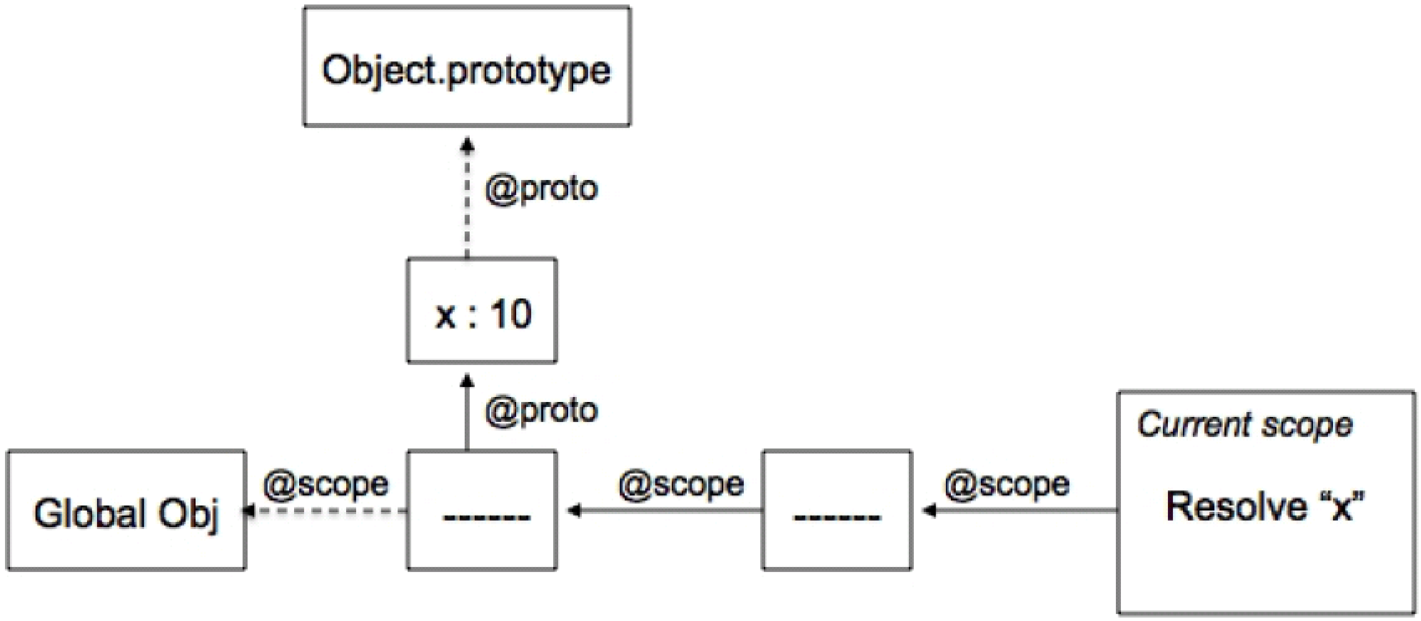
- Program state is represented by a triple ⟨H, l, t⟩
  - H : program heap, mapping locations (L) to objects
  - l : Location of the current scope object ("activation record")
  - t : expression, statement, or program being evaluated
- Note
  - All definable values (including functions) are either objects or primitive values
  - Activation records are normal JavaScript objects and variable declarations define properties of these objects
  - Instead of a stack of activation records, there a chain of scope objects, called the scope chain

This will be a summary to show how operational semantics can be used for realistic programming language. We will not cover this semantics in detail.

Heap operations

- Each Heap object o
  - $\{p_1 : ov_1, \ldots, p_n : ov_n\}$ where p_i are property names and ov_i are primitive values or heap addresses
- Operations on heap objects
  - Dot(H, l, p) = l_1
    - property p of object at location l
  - Put(H, l, p, l_v) = H’
    - Update property p of object at H(l) and return the new Heap
  - H’, l = alloc(H, o)
    - Allocate object o at new location l

Scope and prototype lookup



– Every scope chain has the global object at its base
– Every prototype chain has Object.prototype at the top, which is a native object containing predefined functions such as toString, hasOwnProperty, etc

Some notation (based on ECMA Std)

- o *hasProperty* p
  - p is a property of object o or one of the ancestral prototypes of o
- o *hasOwnProperty* p
  - p is a property of object o itself
- A *JavaScript reference type*
  - pair written l*p where l is a heap address, also called the base type of the reference, and p is a property name

Semantics of scope lookup

ECMA 2.62 :

1. Get the next object (l) in the scope chain. If there isn't one, goto 4.
2. If l "HasProperty" x, return a reference type l*"x".
3. Else, goto 1
4. Return null*x.

$$\frac{\mathtt{Scope}(\mathtt{H}, \mathtt{l}, \text{"}\mathtt{x}\text{"}) = \mathtt{ln}}{\langle \mathtt{H}, \mathtt{l}, \mathtt{x} \rangle \longrightarrow \langle \mathtt{H}, \mathtt{l}, \mathtt{ln} * \text{"}\mathtt{x}\text{"} \rangle}$$

$$\frac{\mathtt{HasProperty}(\mathtt{H}, \mathtt{l}, \mathtt{m})}{\mathtt{Scope}(\mathtt{H}, \mathtt{l}, \mathtt{m}) = \mathtt{l}}$$

$$\frac{\begin{array}{c}\neg(\mathtt{HasProperty}(\mathtt{H}, \mathtt{l}, \mathtt{m})) \\ \mathtt{H}(\mathtt{l}).\mathtt{@Scope} = \mathtt{ln}\end{array}}{\mathtt{Scope}(\mathtt{H}, \mathtt{l}, \mathtt{m}) = \mathtt{Scope}(\mathtt{H}, \mathtt{ln}, \mathtt{m})}$$

$$\mathtt{Scope}(\mathtt{H}, \mathtt{null}, \mathtt{m}) = \mathtt{null}$$

Semantics of prototype lookup

ECMA 2.62 :

1. If base type is null, throw a ReferenceError exception.
2. Else, Call the Get method , passing prop name(x) and base type l as arguments.
3. Return result(2).

$$\frac{\begin{array}{c}\mathtt{H}_2, \mathtt{l}_{\mathtt{excp}} = \mathtt{alloc}(\mathtt{H}, \mathtt{o}) \\ \mathtt{o} = \mathtt{newNativeErr}("", \#\mathtt{RefErrProt})\end{array}}{\langle \mathtt{H}, \mathtt{l}, (\mathtt{null} * \mathtt{m})\rangle \longrightarrow \langle \mathtt{H}_2, \mathtt{l}, \langle \mathtt{l}_{\mathtt{excp}}\rangle\rangle}$$

$$\frac{\mathtt{Get}(\mathtt{H}, \mathtt{l}, \mathtt{m}) = \mathtt{va}}{\langle \mathtt{H}, \mathtt{l}, \mathtt{ln} * \mathtt{m}\rangle \longrightarrow \langle \mathtt{H}, \mathtt{l}, \mathtt{va}\rangle}$$

$$\frac{\begin{array}{c}\mathtt{HasOwnProperty}(\mathtt{H}, \mathtt{l}, \mathtt{m}) \\ \mathtt{Dot}(\mathtt{H}, \mathtt{l}, \mathtt{m}) = \mathtt{va}\end{array}}{\mathtt{Get}(\mathtt{H}, \mathtt{l}, \mathtt{m}) = \mathtt{va}}$$

$$\frac{\begin{array}{c}\neg(\mathtt{HasOwnProperty}(\mathtt{H}, \mathtt{l}, \mathtt{m})) \\ \mathtt{H}(\mathtt{l}).\mathtt{@prototype} = \mathtt{lp}\end{array}}{\mathtt{Get}(\mathtt{H}, \mathtt{l}, \mathtt{m}) = \mathtt{Get}(\mathtt{H}, \mathtt{lp}, \mathtt{m})}$$

Summary of Operational Semantics

- Abstract definition program execution
  - Uses some characterization of program state that reflects the power and expressiveness of language
- JavaScript operational semantics
  - Based on ECMA Standard
  - Lengthy: 70 pages of rules (ascii)
  - Precise definition of program execution, in detail
  - Can prove properties of JavaScript programs
    - Progress: Evaluation only halts with expected set of values
    - Reachability: precise definition of "garbage" for JS programs
    - Basis for proofs of security mechanisms, variable renaming, …

Imperative vs Functional Programs

- Denotational semantics
  - The meaning of an imperative program is a function from states to states.
  - We can write this as a pure functional program that operates on data structures that represent states
- Operational semantics
  - Evaluation $\rightarrow^{v}$ and execution $\rightarrow^{s}$ relations are functions from states to states
  - We could define these functions in Haskell

In principle, every imperative program can be written as a pure functional program (in another language)

What is a *functional* language ?

- “No side effects”
- OK, we have side effects, but we also have higher-order functions…

We will use ***pure functional language*** to mean
“a language with functions, but without side effects
or other imperative features.”

No-side-effects language test

Within the scope of specific declarations of $x_1, x_2, \ldots, x_n$, all occurrences of an expression e containing only variables $x_1, x_2, \ldots, x_n$, must have the same value.

- Example

begin

    integer x=3; integer y=4;

    $5*\underbrace{(x+y)}_{||\,?}-3$

    … // no new declaration of x or y //

    $4*\overbrace{(x+y)}+1$

end

Example languages

- Haskell
- Pure JavaScript

  function (){…}, f(e), ==, [x,y,…], first […], rest […], …
- Impure JavaScript

  x=1; … ; x=2; …
- Common procedural languages are not functional
  - Pascal, C, Ada, C++, Java, Modula, …

Backus’ Turing Award

http://www.cs.cmu.edu/~crary/819-f09/Backus78.pdf

- John Backus was designer of Fortran, BNF, etc.
- Turing Award in 1977
- Turing Award Lecture
  - Functional prog better than imperative programming
  - Easier to reason about functional programs
  - More efficient due to parallelism
  - Algebraic laws

    Reason about programs

    Optimizing compilers

Reasoning about programs

- To prove a program correct,
  - must consider everything a program depends on
- In functional programs,
  - dependence on any data structure is *explicit*
- Therefore,
  - easier to reason about functional programs
- Do you believe this?
  - This thesis must be tested in practice
  - Many who prove properties of programs believe this
  - Not many people really prove their code correct

Haskell Quicksort

- Very succinct program

```
qsort [] = []
qsort (x:xs) = qsort elts_lt_x ++ [x]
                              ++ qsort elts_greq_x
    where elts_lt_x = [y | y <- xs, y < x]
          elts_greq_x = [y | y <- xs, y >= x]
```

- This is the whole thing
  - No assignment – just write expression for sorted list
  - No array indices, no pointers, no memory management, …
  - Disclaimer: does not sort in place

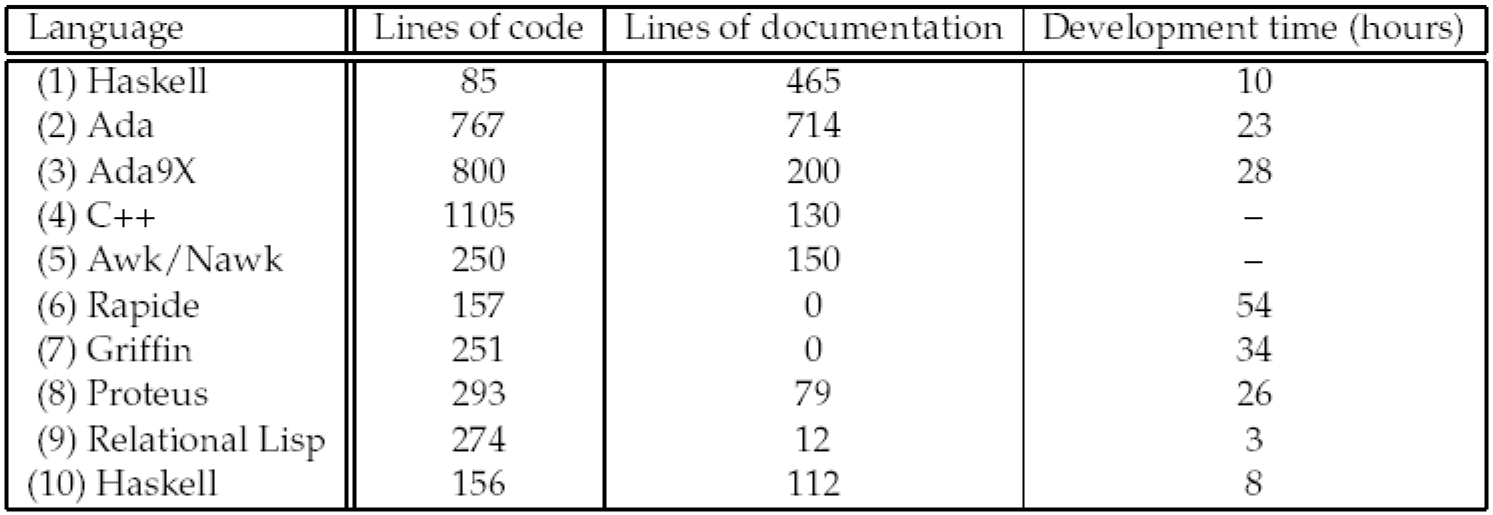
Compare: C quicksort

```
qsort( a, lo, hi ) int a[], hi, lo;
{ int h, l, p, t;
   if (lo < hi) {
         l = lo; h = hi; p = a[hi];
         do {
              while ((l < h) && (a[l] <= p)) l = l+1;
              while ((h > l) && (a[h] >= p)) h = h-1;
              if (l < h) { t = a[l]; a[l] = a[h]; a[h] = t; }
          } while (l < h);
          t = a[l]; a[l] = a[hi]; a[hi] = t;
          qsort( a, lo, l-1 );
          qsort( a, l+1, hi );
    }
}
```

Hudak and Jones, Haskell vs Ada vs …,
Yale University Tech Report, 1994

Interesting case study

- Naval Center programming experiment
  - Separate teams worked on separate languages
  - Surprising differences



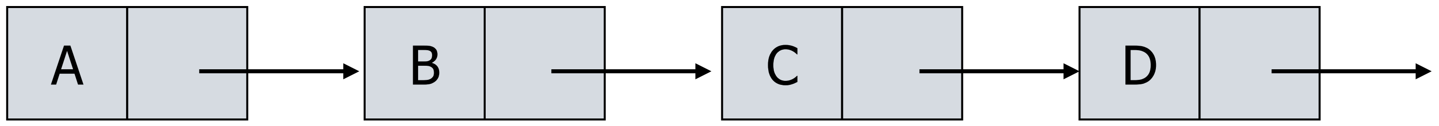
Language	Lines of code	Lines of documentation	Development time (hours)
(1) Haskell	85	465	10
(2) Ada	767	714	23
(3) Ada9X	800	200	28
(4) C++	1105	130	–
(5) Awk/Nawk	250	150	–
(6) Rapide	157	0	54
(7) Griffin	251	0	34
(8) Proteus	293	79	26
(9) Relational Lisp	274	12	3
(10) Haskell	156	112	8

Some programs were incomplete or did not run

  - Many evaluators didn't understand, when shown the code, that the Haskell program was complete. They thought it was a high level partial specification.

Disadvantages of Functional Prog

Functional programs often less efficient. Why?



Change 3rd element of list x to y

- (cons (car x) (cons (cadr x) (cons y (cdddr x))))
  - Build new cells for first three elements of list
- (rplaca (cddr x) y)
  - Change contents of third cell of list directly

However, many optimizations are possible

Von Neumann bottleneck

- Von Neumann
  - Mathematician responsible for idea of stored program
- Von Neumann Bottleneck
  - Backus’ term for limitation in CPU-memory transfer
- Related to sequentiality of imperative languages
  - Code must be executed in specific order

```
function f(x) { if (x<y) then y = x; else x = y; }
g( f(i), f(j) );
```

Eliminating VN Bottleneck

- No side effects
  - Evaluate subexpressions independently
  - Example
    - function f(x) { return x<y ? 1 : 2; }
    - g(f(i), f(j), f(k), …);
- Does this work in practice? Good idea but ...
  - Too much parallelism
  - Little help in allocation of processors to processes
  - ...
  - David Shaw promised to build the non-Von ...
- Effective, easy concurrency is a *hard* problem

Summary

- Parsing
  - The “real” program is the disambiguated parse tree
- Lambda Calculus
  - Notation for functions, free and bound variables
  - Calculate using substitution, rename to avoid capture
- Operational semantics
- Pure functional program
  - May be easier to reason about
  - Parallelism: easy to find, too much of a good thing

Reading

- Textbook
  - Section 4.1.1, Structure of a simple compiler
  - Section 4.2, Lambda calculus, *except*
    - Skip "Reduction and Fixed Points" – too much detail
  - Section 4.4, Functional and imperative languages
- Additional paper (link on web site)
  - "An Operational Semantics for JavaScript"
    - More detail than need, but provided for reference
    - Try to read up through section 2.3 for the main ideas
    - Do not worry about details beyond lecture or homework
  - JavaScript Standard: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf