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
Typical Compiler

Source Program →Lexical Analyzer → Syntax Analyzer → Semantic Analyzer → Intermediate Code Generator → Code Optimizer → Code Generator → Target Program

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

• 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
  \[ x + y \quad \text{and} \quad x + 2y + z \]

• Functions
  \[ \lambda x. (x+y) \quad \lambda z. (x + 2y + z) \]

• Application
  \[ (\lambda x. (x+y)) \ 3 \quad = \quad 3 + y \]
  \[ (\lambda z. (x + 2y + z)) \ 5 \quad = \quad x + 2y + 5 \]

Parsing: \( \lambda x. \ f \ (f \ x) = \lambda x. (\ f \ (f \ (x)) \) \)
Higher-Order Functions

• Given function f, return function f \circ f
  \[ \lambda f. \lambda x. f (f x) \]

• How does this work?

  \[
  (\lambda f. \lambda x. f (f x))(\lambda y. y+1)
  \]

  \[
  = \lambda x. (\lambda y. y+1)((\lambda y. y+1)x)
  \]

  \[
  = \lambda x. (\lambda y. y+1)(x+1)
  \]

  \[
  = \lambda x. (x+1)+1
  \]

In pure lambda calculus, same result if step 2 is altered.
Declarations as “Syntactic Sugar”

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

f(5);
```

\[
(\lambda f. \ f(5)) \ (\lambda x. \ x + 2)
\]

Declaration form used in ML, Haskell:

\[
\text{let } x = e_1 \text{ in } e_2 = (\lambda x. \ e_2) \ e_1
\]
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
  $\int x+y \, dx = \int z+y \, dz \quad \forall x \ P(x) = \forall z \ P(z)$

• 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 $\lambda x. ((\lambda y. y+2) \ x) + y$
Reduction

• Basic computation rule is $\beta$-reduction

$$\lambda x. e_1 \ e_2 \rightarrow [e_2/x]e_1$$

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

\[
(\lambda f. \lambda x. f (f x)) (\lambda y. y+x)
\]

apply twice 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

\[
(\lambda f. \lambda z. f (f z)) (\lambda y. y+x)
\]

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

= \[
\lambda z. z+x+x+x
\]

Easy rule: always rename variables to be distinct
Main Points about Lambda Calculus

• $\lambda$ 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 )
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 :   \( a ::= n \mid X \mid a + a \mid a \ast a \)
  – Bool expressions :   \( b ::= a\leq a \mid \text{not } b \mid b \text{ and } b \)
  – Statements :   \( s ::= \text{skip} \mid x = a \mid s; s \mid \)
                     \( \text{if } b \text{ then } s \text{ else } s \mid \text{while } b \text{ do } s \)

• States
  – Pair \( S = \langle t, \sigma \rangle \)
  – \( t \): syntax being evaluated/executed
  – \( \sigma \): abstract description of memory, in this subset a function from variable names to values, i.e.,
    \( \sigma : \text{Var} \rightarrow \text{Values} \)
Sample operational rules

A rule for Arithmetic Expressions

\[
\begin{align*}
\langle a_1, \sigma \rangle &\rightarrow \langle a'_1, \sigma \rangle \quad [A_{3a}] \\
\langle a_1 + a_2, \sigma \rangle &\rightarrow \langle a'_1 + a_2, \sigma \rangle \\
\langle a_2, \sigma \rangle &\rightarrow \langle a'_2, \sigma \rangle \\
\langle n + a_2, \sigma \rangle &\rightarrow \langle n + a'_2, \sigma \rangle \quad [A_{3b}]
\end{align*}
\]

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

\[
\begin{align*}
\langle a, \sigma \rangle & \rightarrow \langle a', \sigma' \rangle \quad \text{[C₃]} \\
\langle x := a, \sigma \rangle & \rightarrow \langle x = a', \sigma' \rangle \quad \text{[C₃]} \\
\langle x := n, \sigma \rangle & \rightarrow \langle \sigma' \rangle \quad \text{[C₂]}
\end{align*}
\]

How to interpret this rule?

- If the arithmetic expression \( a \) partially evaluates to \( a' \) then the statement \( x = a \) partially evaluates to \( x = a' \).
- Rule \( C₂ \) applies when \( a \) reduces to a value \( n \).
- \( \text{Put}(\sigma, x, n) \) updates the value of \( x \) to \( n \).

Example: \( \langle (x := 10 + 12, \sigma) \rangle \xrightarrow{C₃} \langle x := 22, \sigma \rangle \xrightarrow{C₂} \langle \sigma' \rangle \)
Form of SOS

General form of transition rule:

\[
\frac{P_1, \ldots, P_n}{\langle t, \sigma \rangle \rightarrow \langle t', \sigma' \rangle} \quad \frac{P_1, \ldots, P_n}{\langle t, \sigma \rangle \rightarrow \sigma'}
\]

(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

\[ \langle \text{if } tt \text{ then } s_1 \text{ else } s_2, \sigma \rangle \rightarrow \langle s_1, \sigma \rangle[C_{5a}] \]
\[ \langle \text{if } ff \text{ then } s_1 \text{ else } s_2, \sigma \rangle \rightarrow \langle s_2, \sigma \rangle[C_{5b}] \]
\[ \langle b, \sigma \rangle \rightarrow \langle b', \sigma \rangle \]
\[ \langle \text{if } b \text{ then } s_1 \text{ else } s_2, \sigma \rangle \rightarrow \langle \text{if } b' \text{ then } s_1 \text{ else } s_2, \sigma \rangle[C_{5c}] \]

While

\[ \langle \text{while } b \text{ do } s, \sigma \rangle \rightarrow \]
\[ \langle \text{if } b \text{ then } s; \text{ while } b \text{ s } \text{ else } \text{ skip end}, \sigma \rangle[C_{6}] \]
Context Sensitive Rules

The above rules have a similar premise:

Combine them into a single rule of the following form:

\[ \langle a, \sigma \rangle \rightarrow \langle a', \sigma \rangle \]

\[ AC(a) \rightarrow AC(a') \]

where \( AC :: \_\_ + a | n + \_\_ * a | n * \_ \)
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{align*}
\text{begin} \\
\quad \text{integer } x=3; \text{ integer } y=4; \\
\quad 5*(x+y)-3 \\
\quad \ldots \quad \text{\( // \) no new declaration of \( x \) or \( y \) \( // \) } \\
\quad 4*(x+y)+1 \\
\text{end}
\end{align*}
\]
Example languages

- Haskell
- Pure JavaScript
  
  ```javascript
  function (){...}, f(e), ==, [x,y,...], first [...], rest [...], ...
  ```

- Impure JavaScript
  
  ```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
```
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 );
  }
}
```
Interesting case study

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

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

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