Syntax of All Sorts

COS 441
Princeton University
Fall 2004

Acknowledgements

Many slides from this lecture have been adapted from John Mitchell’s CS-242 course at Stanford
– Good source of supplemental information
  http://theory.stanford.edu/people/jcm/books/cpl-teaching.html
– Differs in foundational approach when compared to Harper’s approach will discuss this difference in later lectures

Interpreter vs Compiler

| Source Program | Input | Interpreter | Output |
| Source Program | Compiler | Input | Target Program | Output |

• The difference is actually a bit more fuzzy
• Some “interpreters” compile to native code
  – SML/NJ runs native machine code!
  – Does fancy optimizations too
• Some “compilers” compile to byte-code which is interpreted
  – javac and a JVM which may also compile

Interactive vs Batch

• SML/NJ is a native code compiler with an interactive interface
• javac is a batch compiler for bytecode which may be interpreted or compiled to native code by a JVM
• Python compiles to bytecode then interprets the bytecode it has both batch and interactive interfaces
• Terms are historical and misleading
  – Best to be precise and verbose

Typical Compiler

Source Program
  Lexical Analyzer
  Syntax Analyzer
  Semantic Analyzer
  Intermediate Code Generator
  Code Optimizer
  Code Generator
  Target Program
Brief Look at Syntax

- Grammar
  \[ e ::= n | e + e | e \times e \]
  \[ n ::= d | nd \]
  \[ d ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 \]

- Expressions in language
  \[ e \rightarrow e \times e \rightarrow e \times e + e \rightarrow n \times n + n \rightarrow nd \times d + d \rightarrow ... \rightarrow 27 \times 4 + 3 \]

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

Dealing with Parsing Ambiguity

- Ambiguity
  - Expression \( 27 \times 4 + 3 \) can be parsed two ways
  - Problem: \( 27 \times (4 + 3) \neq (27 \times 4) + 3 \)

- Ways to resolve ambiguity
  - Precedence
    - Group \( \times \) before +
    - Parse \( 3 \times 4 + 2 \) as \((3 \times 4) + 2 \)
  - Associativity
    - Parenthesize operators of equal precedence to left (or right)
    - Parse \( 3 + 4 \times 5 \) as \((3 + 4) \times 5\)

Rewriting the Grammar

- Harper describe a way to rewrite the grammar to avoid ambiguity
- Eliminating ambiguity in a parsing grammar is a dark art
  - Not always even possible
  - Depends on the parsing technique you use
  - Learn all you want to know and more in a compiler book or course
- Ambiguity is bad when trying to think formally about programming languages

Abstract Syntax Trees

- We want to reason about expressions inductively
  - So we need an inductive description of expressions
- We could use the parse tree
  - But it has all this silly term and factor stuff to parsing unambiguous
- Introduce nicer tree after the parsing phase

Syntax Comparisons

<table>
<thead>
<tr>
<th>Ambiguous Concrete</th>
<th>Abstract</th>
</tr>
</thead>
<tbody>
<tr>
<td>Digits ( d ::= 0 \ldots 9 )</td>
<td>Numbers ( n \in \mathbb{N} )</td>
</tr>
<tr>
<td>Numbers ( n ::= d</td>
<td>nd )</td>
</tr>
<tr>
<td>Expressions ( e ::= n )</td>
<td></td>
</tr>
<tr>
<td>| ( e ::= e + e )</td>
<td></td>
</tr>
<tr>
<td>Terms ( t ::= f \times t )</td>
<td></td>
</tr>
<tr>
<td>Factors ( f ::= n</td>
<td>(f) )</td>
</tr>
</tbody>
</table>
Induction Over Syntax

- We can think of the CFG notation as defining a predicate \( \text{exp} \).
- Thinking about things this way lets us use rule induction on abstract syntax.

\[
\frac{N \in \mathbb{N}}{\text{num}(N) \text{ exp}} \quad \frac{E_1 \text{ exp} \quad E_2 \text{ exp}}{\text{plus}(E_1, E_2) \text{ exp}} \quad \frac{E_1 \text{ exp} \quad E_2 \text{ exp}}{\text{times}(E_1, E_2) \text{ exp}} \quad \frac{}{\text{num-e}}
\]

A More General Pattern

- One might ask what is the general pattern of predicates defined by CFG that represent abstract syntax.
  - \( \Omega \) represents a signature that assigns arities to operators such as \( \text{num}(N), \text{plus}, \text{times} \).
  - Note the \( T_1 \ldots T_n \) are all unique.

\[
\frac{T_1 \text{ term}_{\Omega} \ldots T_n \text{ term}_{\Omega} \quad \Omega(OP) = n}{\text{OP}(T_1,\ldots,T_n) \text{ term}_{\Omega}}
\]

An Example

\[
\frac{}{\text{zero} \text{ nat}} \quad \frac{X \text{ nat}}{\text{succ}(X) \text{ nat}}
\]

Another Example

\[
\frac{}{\text{zero} \text{ even}} \quad \frac{X \text{ even}}{\text{succ}(X) \text{ odd}} \quad \frac{X \text{ odd}}{\text{succ}(X) \text{ even}}
\]

<table>
<thead>
<tr>
<th>Operator</th>
<th>Arity</th>
</tr>
</thead>
<tbody>
<tr>
<td>zero</td>
<td>0</td>
</tr>
<tr>
<td>succ</td>
<td>1</td>
</tr>
</tbody>
</table>

\( n \ := \text{zero} \quad | \quad \text{succ}(n) \)
Another Example

\[
\begin{array}{c|c|c}
\text{Operator} & \text{Arity} & \text{even} := \text{zero} \\
\text{zero} & 0 & \mid \text{succ(odd)} \\
\text{succ} & 1 & \\
\end{array}
\]

Another Example

\[
\begin{array}{c|c|c}
\text{Operator} & \text{Arity} & \text{even} := \text{zero} \\
\text{zero} & 0 & \mid \text{succ(odd)} \\
\text{succ} & 1 & \\
\end{array}
\]

NOT Abstract Syntax

\[
\begin{array}{c|c|c}
\text{Operator} & \text{Arity} & \text{even} := \text{zero} \\
\text{zero} & 0 & \mid \text{succ(odd)} \\
\text{succ} & 1 & \\
\end{array}
\]

Abstract Syntax To ML

\[
\begin{array}{c|c|c}
\text{Operator} & \text{Arity} & \text{odd} := \text{succ(even)} \\
\text{zero} & 0 & \mid \text{succ(odd)} \\
\text{succ} & 1 & \\
\end{array}
\]

Abstract Syntax To ML

\[
\begin{array}{c|c|c}
\text{Operator} & \text{Arity} & \text{even} := \text{zero} \\
\text{zero} & 0 & \mid \text{succ(odd)} \\
\text{succ} & 1 & \\
\end{array}
\]

Abstract Syntax To ML

\[
\begin{array}{c|c|c}
\text{Operator} & \text{Arity} & \text{odd} := \text{succ(even)} \\
\text{zero} & 0 & \mid \text{succ(odd)} \\
\text{succ} & 1 & \\
\end{array}
\]

Abstract Syntax To ML

\[
\begin{array}{c|c|c}
\text{Operator} & \text{Arity} & \text{odd} := \text{succ(even)} \\
\text{zero} & 0 & \mid \text{succ(odd)} \\
\text{succ} & 1 & \\
\end{array}
\]

Abstract Syntax To ML

\[
\begin{array}{c|c|c}
\text{Operator} & \text{Arity} & \text{odd} := \text{succ(even)} \\
\text{zero} & 0 & \mid \text{succ(odd)} \\
\text{succ} & 1 & \\
\end{array}
\]

Abstract Syntax To ML

\[
\begin{array}{c|c|c}
\text{Operator} & \text{Arity} & \text{odd} := \text{succ(even)} \\
\text{zero} & 0 & \mid \text{succ(odd)} \\
\text{succ} & 1 & \\
\end{array}
\]

Abstract Syntax To ML

\[
\begin{array}{c|c|c}
\text{Operator} & \text{Arity} & \text{odd} := \text{succ(even)} \\
\text{zero} & 0 & \mid \text{succ(odd)} \\
\text{succ} & 1 & \\
\end{array}
\]

Abstract Syntax To ML

\[
\begin{array}{c|c|c}
\text{Operator} & \text{Arity} & \text{odd} := \text{succ(even)} \\
\text{zero} & 0 & \mid \text{succ(odd)} \\
\text{succ} & 1 & \\
\end{array}
\]

Abstract Syntax To ML

\[
\begin{array}{c|c|c}
\text{Operator} & \text{Arity} & \text{odd} := \text{succ(even)} \\
\text{zero} & 0 & \mid \text{succ(odd)} \\
\text{succ} & 1 & \\
\end{array}
\]

Abstract Syntax To ML

\[
\begin{array}{c|c|c}
\text{Operator} & \text{Arity} & \text{odd} := \text{succ(even)} \\
\text{zero} & 0 & \mid \text{succ(odd)} \\
\text{succ} & 1 & \\
\end{array}
\]

Abstract Syntax To ML

\[
\begin{array}{c|c|c}
\text{Operator} & \text{Arity} & \text{odd} := \text{succ(even)} \\
\text{zero} & 0 & \mid \text{succ(odd)} \\
\text{succ} & 1 & \\
\end{array}
\]

Abstract Syntax To ML

\[
\begin{array}{c|c|c}
\text{Operator} & \text{Arity} & \text{odd} := \text{succ(even)} \\
\text{zero} & 0 & \mid \text{succ(odd)} \\
\text{succ} & 1 & \\
\end{array}
\]

Abstract Syntax To ML

\[
\begin{array}{c|c|c}
\text{Operator} & \text{Arity} & \text{odd} := \text{succ(even)} \\
\text{zero} & 0 & \mid \text{succ(odd)} \\
\text{succ} & 1 & \\
\end{array}
\]

Abstract Syntax To ML

\[
\begin{array}{c|c|c}
\text{Operator} & \text{Arity} & \text{odd} := \text{succ(even)} \\
\text{zero} & 0 & \mid \text{succ(odd)} \\
\text{succ} & 1 & \\
\end{array}
\]

Abstract Syntax To ML

\[
\begin{array}{c|c|c}
\text{Operator} & \text{Arity} & \text{odd} := \text{succ(even)} \\
\text{zero} & 0 & \mid \text{succ(odd)} \\
\text{succ} & 1 & \\
\end{array}
\]

Abstract Syntax To ML

\[
\begin{array}{c|c|c}
\text{Operator} & \text{Arity} & \text{odd} := \text{succ(even)} \\
\text{zero} & 0 & \mid \text{succ(odd)} \\
\text{succ} & 1 & \\
\end{array}
\]

Abstract Syntax To ML

\[
\begin{array}{c|c|c}
\text{Operator} & \text{Arity} & \text{odd} := \text{succ(even)} \\
\text{zero} & 0 & \mid \text{succ(odd)} \\
\text{succ} & 1 & \\
\end{array}
\]

Abstract Syntax To ML

\[
\begin{array}{c|c|c}
\text{Operator} & \text{Arity} & \text{odd} := \text{succ(even)} \\
\text{zero} & 0 & \mid \text{succ(odd)} \\
\text{succ} & 1 & \\
\end{array}
\]

Abstract Syntax To ML

\[
\begin{array}{c|c|c}
\text{Operator} & \text{Arity} & \text{odd} := \text{succ(even)} \\
\text{zero} & 0 & \mid \text{succ(odd)} \\
\text{succ} & 1 & \\
\end{array}
\]

Abstract Syntax To ML

\[
\begin{array}{c|c|c}
\text{Operator} & \text{Arity} & \text{odd} := \text{succ(even)} \\
\text{zero} & 0 & \mid \text{succ(odd)} \\
\text{succ} & 1 & \\
\end{array}
\]
Abstract Syntax To ML

\[
\begin{align*}
n &::= \text{zero} \\
& \mid \text{succ}(n) \\
\text{nat} &::= \text{Zero} \\
& \mid \text{Succ of nat} \\
\text{exp} &::= \text{num}[n] \\
& \mid \text{plus}(e_1, e_2) \\
& \mid \text{times}(e_1, e_2) \\
\text{datatype} &\text{exp} = \text{Num of int} \\
& \mid \text{Plus of } (\text{exp} \ast \text{exp}) \\
& \mid \text{Times of } (\text{exp} \ast \text{exp})
\end{align*}
\]

Capitalize by convention A small cheat for performance

Abstract Syntax To ML (cont.)

\[
\begin{align*}
n &::= \text{zero} \\
& \mid \text{succ}(n) \\
\text{nat} &::= \text{Zero} \\
& \mid \text{Succ of nat} \\
\text{even} &::= \text{zero} \\
& \mid \text{succ}(\text{odd}) \\
\text{odd} &::= \text{succ}(\text{even}) \\
\text{even} &::= \text{EvenZero} \\
& \mid \text{EvenSucc of } (\text{odd}) \\
\text{odd} &::= \text{OddSucc of } (\text{even})
\end{align*}
\]

Syntax Trees Are Not Enough

- Programming languages include binding constructs
  - variables function arguments, variable function declarations, type declarations
- When reasoning about binding constructs we want to ignore irrelevant details
  - e.g. the actual “spelling” of the variable
  - we only that variables are the same or different
- Still we must be clear about the scope of a variable definition

\[
\begin{align*}
\text{fun}
\end{align*}
\]
Abstract Binding Trees

• Harper uses abstract binding trees (Chapter 5)
  – This solution is based on very recent work

• The solution is new but the problems are old!

• We will talk about the problem with binding today and deal with the solutions in the next lecture

Lambda Calculus

• Formal system with three parts
  – Notation for function expressions
  – Proof system for equations
  – Calculation rules called reduction

• Basic syntactic notions
  – Free and bound variables
  – Illustrates some ideas about scope of binding

• Symbolic evaluation useful for discussing programs

History

• Original intention
  – Formal theory of substitution

• More successful for computable functions
  – Substitution → symbolic computation
  – Church/Turing thesis

• Influenced design of Lisp, ML, other languages

• Important part of CS history and theory

Expressions and Functions

• Expressions
  \[ x + y \quad x + 2 \times y + z \]

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

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

Parsing: \[ \lambda x. f (f \ x) = \lambda x. (f (f \ x)) \]

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
  \[ (\lambda x. e_1) \ e_2 \rightarrow [x \leftarrow e_2]e_1 \]
  where substitution involves renaming as needed

• Example:
  \[ (\lambda f. \lambda x. f (f \ x)) (\lambda y. y+x) \]
Rename Bound Variables

• Rename bound variables to avoid conflicts
  \((\lambda f. \lambda z. f \, (f \, z)) \, (\lambda y. y+x)\) →
  ... →
  \(\lambda z. (z+x)+x\)

• Substitute “blindly”
  \((\lambda f. \lambda x. f \, (f \, x)) \, (\lambda y. y+x)\) →
  ... →
  \(\lambda x. (x+x)+x\)

Reduction

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

Renamed bound “x” to “z” to avoid conflict with free “x”
<table>
<thead>
<tr>
<th>Reduction</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\lambda f. \lambda z. f (f z)) (\lambda y. y+x)) → (\lambda z. (\lambda y. y+x) ((\lambda y. y+x) z)) → (\lambda z. (\lambda y. y+x) (z+x)) → (\lambda z. (z+x+x))</td>
<td>((\lambda f. \lambda z. f (f z)) (\lambda y. y+x)) → (\lambda z. (\lambda y. y+x) ((\lambda y. y+x) z)) → (\lambda z. (\lambda y. y+x) (z+x)) → (\lambda z. (y+ (z+x)) (y+x))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reduction</th>
<th>Incorrect Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\lambda f. \lambda z. f (f z)) (\lambda y. y+x)) → (\lambda z. (\lambda y. y+x) ((\lambda y. y+x) z)) → (\lambda z. (\lambda y. y+x) (z+x)) → (\lambda z. ((z+x)+x))</td>
<td>((\lambda f. \lambda x. f (f x)) (\lambda y. y+x)) → (\lambda x. ((\lambda y. y+x) x))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Incorrect Reduction</th>
<th>Incorrect Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\lambda f. \lambda x. f (f x)) (\lambda y. y+x)) → (\lambda x. ((\lambda y. y+x) (\lambda x. f (f x))))</td>
<td>((\lambda f. \lambda x. f (f x)) (\lambda y. y+x)) → (\lambda x. ((\lambda y. y+x) x))</td>
</tr>
</tbody>
</table>
Incorrect Reduction

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

Incorrect Reduction

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

Incorrect Reduction

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

Incorrect Reduction

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

Main Points About \(\lambda\)-calculus

- \(\lambda\) captures “essence” of variable binding
  - Function parameters
  - Bound variables can be renamed
- Succinct function expressions
- Simple symbolic evaluator via substitution
- Easy rule: always rename bound variables to be distinct