Recall Architecture of Compilers, Interpreters

- Front end: syntax, (possibly) typechecking, other checks
- Back end: semantics (i.e. execution)
Specifying Syntax, Semantics

- We have seen how the syntax of a programming language may be specified precisely
  - Regular expressions
  - Context-free grammars

- What about formal methods for defining the semantics of a programming language?
  - I.e., what does a program mean / do?

Formal Semantics of a Prog. Lang.

- Mathematical description of all possible computations performed by programs written in that language

- Three main approaches to formal semantics
  - Denotational
  - Operational
  - Axiomatic
Formal Semantics (cont.)

- Denotational semantics: translate programs into math!
  - Usually: convert programs into functions mapping inputs to outputs
  - Analogous to compilation

- Operational semantics: define how programs execute
  - Often on an abstract machine (mathematical model of computer)
  - Analogous to interpretation

- Axiomatic semantics
  - Describe programs as predicate transformers, i.e. for converting initial assumptions into guaranteed properties after execution
    - Preconditions: assumed properties of initial states
    - Postcondition: guaranteed properties of final states
  - Logical rules describe how to systematically build up these transformers from programs

This Course: Operational Semantics

- We will show how an operational semantics may be defined using a subset of OCaml

- Approach: use rules to define a relation
  
  \[ E \Rightarrow \mathbf{v} \]

  - \( E \): expression in OCaml subset
  - \( \mathbf{v} \): value that results from evaluating \( E \)

- To begin with, need formal definitions of:
  - Set \( \text{Exp} \) of expressions
  - Set \( \text{Val} \) of values
Defining Exp

Recall: operational semantics defines what happens in backend

- Front end parses code into abstract syntax trees (ASTs)
- So inputs to backend are ASTs

How to define ASTs?

- Standard approach: using grammars!
- Idea: grammar defines abstract syntax (no parentheses, grouping constructs, etc.; grouping is implicit)

OCaml Abstract Syntax

\[
E ::= x \mid n \mid \text{true} \mid \text{false} \mid [] \\
\mid E \ op \ E \ (op \in \{+, -, '/', '*', '=' ,'<', '>', '::', etc\}) \\
\mid \text{l\_op} \ E \ (\text{l\_op} \in \{\text{hd}, \text{tl}\}) \\
\mid \text{if} \ E \ \text{then} \ E \ \text{else} \ E \\
\mid \text{fun} \ x \ \rightarrow \ E \ | \ E \ E \ | \text{let} \ x = E \ \text{in} \ E
\]

- \(x\) may be any identifier
- \(n\) is any numeral (digit sequence, with optional -).
- \text{true} and \text{false} stand for the two boolean constants
- \([]\) is the empty list

\(Exp = \) set of (type-correct) ASTs defined by grammar

- Note that the grammar is ambiguous
  - OK because not using grammar for parsing
  - But for defining the set of all syntactically legal terms
Values

- What can results be?
  - Integers
  - Booleans
  - Lists
  - Functions
- We will deal with first three initially

Formal Definition of Val

- Let
  - \( Z = \{\ldots, -1, 0, -1, \ldots\} \) be the (math) set of integers
  - \( B = \{\text{ff, tt}\} \) be the (math) set of booleans
  - nil be a distinguished value (empty list)
- Then Val is the smallest set such that
  - \( Z, B \subseteq \text{Val} \) and nil \( \in \text{Val} \)
  - If \( v_1, v_2 \in \text{Val} \) then \( \langle v_1, v_2 \rangle \in \text{Val} \)
- “Smallest set”? 
  - Every integer and boolean is a value, as is nil
  - Any pair of values is also a value
Operations on Val

- Basic operations will be assumed
  - \(+\), \(-\), \(*\), \(/\), \(=\), \(<\), \(\leq\), etc.
- Not all operations are applicable to all values!
  - \(tt + ff\) is undefined
  - So is \(1 + nil\)
- A key purpose of type checking is to prevent these undefined operations from occurring during execution

Implementing Exp, Val in OCaml

\[
E ::= x \mid n \mid true \mid false \mid [] \mid if E then E else E \\
| \quad fun x = E \mid E E \mid let x = E in E \ldots
\]

\[
\text{type} \quad \text{ast} = \\
\text{Id of string} \\
| \text{Num of int} \\
| \text{Bool of bool} \\
| \text{Nil} \\
| \text{If of ast * ast * ast} \\
| \text{Fun of string * ast} \\
| \text{App of ast * ast} \\
| \text{Let of string * ast * ast} \\
| \ldots
\]

\[
\text{type} \quad \text{value} = \\
\text{Val_Num of int} \\
| \text{Val_Bool of bool} \\
| \text{Val_Nil} \\
| \text{Val_Pair of value *} \\
| \text{value} \\
| \ldots
\]
Defining Evaluation (⇒)

- Approach is inductive and uses rules:
  - Idea: if the conditions \( H_1 \ldots H_n \) (“hypotheses”) are true, the rule says the condition \( C \) (“conclusion”) below the line follows
  - Hypotheses, conclusion are statements about ⇒; hypotheses involve subexpressions of conclusions
  - If \( n=0 \) (no hypotheses) then the conclusion is automatically true and is called an axiom
    - A “-” may be written in place of the hypothesis list in this case
    - Terminology: statements one is trying to prove are called judgments

- This method is often called “Structural Operational Semantics (SOS)” or “Natural Semantics”

---

SOS Rules: Basic Values

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Each basic entity evaluates to its corresponding value
- Note: axioms!
SOS Rules: Built-in Functions

- How about built-in functions (+, -, etc.)?
  - In OCaml, type-checking done in front end
  - Thus, ASTs coming to back end are type-correct
  - So we assume Exp contains type-correct ASTs
- We will use relevant operations on value side

SOS Rules: Built-in Functions

- For arithmetic, comparison operations, etc.

  \[
  \begin{array}{c}
  E_1 \Rightarrow v_1 \quad E_2 \Rightarrow v_2 \\
  E_1 \text{ op } E_2 \Rightarrow v_1 \text{ op } v_2 \\
  \end{array}
  \]

- For ::

  \[
  \begin{array}{c}
  E_1 \Rightarrow v_1 \quad E_2 \Rightarrow v_2 \\
  E_1 \text{ :: } E_2 \Rightarrow \langle v_1, v_2 \rangle \\
  \end{array}
  \]

- Rules are recursive
- :: is implemented using pairing
  - Type system guarantees result is list
Trees of Semantic Rules

- When we apply rules to an expression, we actually get a tree
  - Corresponds to the recursive evaluation procedure
    - For example: \((3 + 4) + 5\)

\[
\begin{align*}
3 & \Rightarrow 3 \\
4 & \Rightarrow 4 \\
(3 + 4) & \Rightarrow 7 \\
5 & \Rightarrow 5 \\
(3 + 4) + 5 & \Rightarrow 12
\end{align*}
\]

Rules for \(\text{hd}, \text{tl}\)

- \(\text{hd } E \Rightarrow v_1\)
- \(\text{tl } E \Rightarrow v_2\)

- Note that the rules only apply if \(E\) evaluates to a pair of values
- Nothing in this rule requires the pair to correspond to a list
- The OCaml type system ensures this
Error Cases

<table>
<thead>
<tr>
<th>E₁ ⇒ v₁</th>
<th>E₂ ⇒ v₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>E₁ + E₂ ⇒ v₁ + v₂</td>
<td></td>
</tr>
</tbody>
</table>

- What if \( v₁, v₂ \) aren’t integers?
  - E.g., what if we write \( \text{false + true} \)?
  - It can be parsed in OCaml, but type checker will disallow it from being passed to back end
- In a language with dynamic strong typing (e.g. Ruby), rules include explicit type checks

<table>
<thead>
<tr>
<th>E₁ ⇒ v₁, v₁ ∈ ( \mathbb{Z} )</th>
<th>E₂ ⇒ v₂, v₂ ∈ ( \mathbb{Z} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>E₁ + E₂ ⇒ v₁ + v₂</td>
<td></td>
</tr>
</tbody>
</table>

- Convention: if no rules are applicable to an expression, its result is an error

Rules for If

<table>
<thead>
<tr>
<th>E₁ ⇒ tt</th>
<th>E₂ ⇒ v₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>if E₁ then E₂ else E₃ ⇒ v₂</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>E₁ ⇒ ff</th>
<th>E₃ ⇒ v₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>if E₁ then E₂ else E₃ ⇒ v₃</td>
<td></td>
</tr>
</tbody>
</table>

- Notice that only one branch is evaluated
- E.g.
  - if true then 3 else 4 ⇒ 3
  - if false then 3 else 4 ⇒ 4
Using Rules to Define Evaluation

- $E \Rightarrow v$ holds if and only if a proof can be built
  - Proofs start with axioms, involve applications of rules whose hypotheses have been proved
  - No proof means $E \not\Rightarrow v$
- Proofs can be constructed in a goal-directed fashion
- Thus, function $\text{eval (E)} = \{v \mid E \Rightarrow v\}$
  - Determinism of semantics implies at most one element for any $E$

Rules for Identifiers

- The previous rules handle expressions that involve constants (e.g. $1$, $\text{true}$) and operations
- What about variables?
  - These are allowed as expressions
  - How do we evaluate them?
- In a program, variables must be declared
  - The values that are part of the declaration are used to evaluate later occurrences of the variables
  - We will use environments to “hold” these declarations in our semantics
Environments

- Mathematically, an environment is a partial function from identifiers to values
  - If $A$ is an environment, and $id$ is an identifier, then $A(id)$ can either be …
  - … a value (intuition: the variable has been declared)
  - … or undefined (intuition: variable has not been declared)
- An environment can also be thought of as a table
  - If $A$ is
    \[
    \begin{array}{c|c}
    \text{Id} & \text{Val} \\
    \hline
    x & 0 \\
    y & \text{ff}
    \end{array}
    \]
  - then $A(x)$ is $0$, $A(y)$ is $\text{ff}$, and $A(z)$ is undefined

Notation, Operations on Environments

- is the empty environment (undefined for all ids)
- $x:v$ is the environment that maps $x$ to $v$ and is undefined for all other ids
- If $A$ and $A'$ are environments then $A$, $A'$ is the environment defined as follows
  \[
  (A, A')(id) = \begin{cases} 
  A'(id) & \text{if } A'(id) \text{ defined} \\
  A(id) & \text{if } A'(id) \text{ undefined but } A(id) \text{ defined} \\
  \text{undefined} & \text{otherwise}
  \end{cases}
  \]
- Idea: $A'$ “overwrites” definitions in $A$
- For brevity, can write $\cdot$, $A$ as just $A$
Semantics with Environments

To give a semantics for identifiers, we will extend judgments from

\[ E \Rightarrow v \]

to

\[ A; E \Rightarrow v \]

where \( A \) is an environment

- Idea: \( A \) is used to give values to the identifiers in \( E \)
- \( A \) can be thought of as containing all the declarations made up to \( E \)

Existing rules can be modified by inserting \( A \) everywhere in the judgments

Existing Rules Have To Be Modified

- E.g.

  \[
  \begin{array}{c}
  E_1 \Rightarrow v_1 \\
  E_2 \Rightarrow v_2 \\
  E_1 + E_2 \Rightarrow v_1 + v_2
  \end{array}
  \]

- becomes

  \[
  \begin{array}{c}
  A; E_1 \Rightarrow v_1 \\
  A; E_2 \Rightarrow v_2 \\
  A; E_1 + E_2 \Rightarrow v_1 + v_2
  \end{array}
  \]

- These modifications can be done systematically
Rule for Identifiers

- $x$ is an identifier
- To determine its value $v$ “look it up” in $A$!

\[
\begin{array}{c}
A(x) = v \\
A; x \Rightarrow v
\end{array}
\]

Rule for Let binding

- We evaluate the first expression, and then evaluate the second expression in an environment extended to include a binding for $x$

\[
\begin{array}{c}
A; E_1 \Rightarrow v_1 \\
A, x : v_1; E_2 \Rightarrow v_2
\end{array}
\]

\[
A; \text{let } x = E_1 \text{ in } E_2 \Rightarrow v_2
\]
Function Values

- So far our semantics handles:
  - Constants
  - Built-in operations
  - Identifiers

- What about function definitions?
  - Recall form: \texttt{fun x \rightarrow E}
  - To evaluate these expressions we need to add closures to our set of values

Closures

- ... are what OCaml function expressions evaluate to
- A closure consists of:
  - Parameter (id)
  - Body (expression)
  - Environment (used to evaluate free variables in body)

- Formal extension to Val:
  - if \( x \) is an id, \( E \) is an expression, and \( A \) is an environment
  - ... then \((A, \lambda x. E) \in \text{Val}\)
Rule for Function Definitions

\[
\begin{array}{c|c}
\text{A; fun} & \text{x } \rightarrow \text{E} \\
\hline
\text{(A, } \lambda x . \text{E)}
\end{array}
\]

- The expression evaluates to a closure
  - The id and body in the closure come from the expression
  - The environment is the one in effect when the expression is evaluated
- This will be used to implement static scope

Evaluating Function Application

- How do we evaluate a function application expression of the form \( E_1 E_2 \)?
  - Static scope
  - Call by value
- Strategy
  - Evaluate \( E_1 \), producing \( v_1 \)
  - If \( v_1 \) is indeed a function (i.e. closure) then
    - Evaluate \( E_2 \), producing \( v_2 \)
    - Set the parameter of closure \( v_1 \) equal to \( v_2 \)
    - Evaluate the body under this binding of the parameter
    - Remember that non-parameter ids in the body must be interpreted using the closure!
Rule for Function Application

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A; E_1 \Rightarrow (A'; \lambda x.E) )</td>
<td>1st hypothesis: ( E_1 ) evaluates to a closure</td>
</tr>
<tr>
<td>( A; E_2 \Rightarrow v_2 )</td>
<td>2nd hypothesis: ( E_2 ) produces a value (call by value!)</td>
</tr>
<tr>
<td>( A', x:v_2; E \Rightarrow v )</td>
<td>3rd hypothesis: Body ( E ) in modified closure environment produces a value</td>
</tr>
<tr>
<td>( A; E_1 E_2 \Rightarrow v )</td>
<td>This last value is the result of the application</td>
</tr>
</tbody>
</table>

Example: \((\text{fun } x \rightarrow x + 3) \ 4\)

\[
\begin{align*}
\ast; x:4; x & \Rightarrow 4 \quad \ast; x:4; 3 \Rightarrow 3 \\
\ast; \text{fun } x \rightarrow x + 3 & \Rightarrow (\ast, \lambda x.x + 3) \\
\ast; 4 & \Rightarrow 4 \\
\ast; x:4; x + 3 & \Rightarrow 7 \\
\ast; (\text{fun } x \rightarrow x + 3) 4 & \Rightarrow 7
\end{align*}
\]
Example: \((\text{fun } x \rightarrow (\text{fun } y \rightarrow x + y)) \ 3 \ 4\)

\[
\begin{align*}
\text{•}; \ (\text{fun } x \rightarrow (\text{fun } y \rightarrow x + y)) & \Rightarrow (\text{•}, \lambda x.(\text{fun } y \rightarrow x + y)) \\
\text{•}; \ 3 & \Rightarrow 3 \\
x:3; \ (\text{fun } y \rightarrow x + y) & \Rightarrow (x:3, \lambda y.(x + y)) \\
\text{•}; \ (\text{fun } x \rightarrow (\text{fun } y \rightarrow x + y)) \ 3 & \Rightarrow (x:3, \lambda y.(x + y))
\end{align*}
\]

Let <previous> = \((\text{fun } x \rightarrow (\text{fun } y \rightarrow x + y)) \ 3\)

Example (cont.)

\[
\begin{align*}
\text{•}; x:3, y:4; \ x & \Rightarrow 3 \quad \text{•}; x:3, y:4; \ y & \Rightarrow 4 \\
\text{•}; \ <\text{previous}> & \Rightarrow (x:3, \lambda y.(x + y)) \\
\text{•}; \ 4 & \Rightarrow 4 \\
x:3, y:4; \ (x + y) & \Rightarrow 7 \\
\text{•}; \ (<\text{previous}> \ 4) & \Rightarrow 7
\end{align*}
\]
Dynamic scoping

- The previous rule for functions implements static scoping, since it implements closures
- We could easily implement dynamic scoping

\[
\begin{align*}
A; E_1 &\Rightarrow (A', \lambda x. E) \\
A; E_2 &\Rightarrow v_2 \\
A, x: v_2; &E \Rightarrow v \\
A; E_1 E_2 &\Rightarrow v
\end{align*}
\]

- The only difference is to use the current environment \( A \), not the environment \( A' \)
  - Easy to see the origins of the dynamic scoping bug!

Practice Examples

- Give a derivation that proves the following (where \( \bullet \) is the empty environment)
  - \( \bullet; \text{let } x = 5 \text{ in } let \ y = 7 \text{ in } x+y \Rightarrow 12 \)
  - \( \bullet; \text{let } x = \text{let } x = 5 \text{ in } x+2 \text{ in } x+2 \Rightarrow 9 \)
  - \( \bullet; \text{let } f = \text{fun } x \rightarrow x+5 \text{ in } f \ 7 \Rightarrow 12 \)
  - \( \bullet; \text{let } y = 5 \text{ in let } f = \text{fun } x \rightarrow x+y \text{ in let } y = 6 \text{ in } f \ 7 \Rightarrow 12 \)
- Using the dynamic scoping version of the function application rule, prove
  - \( \bullet; \text{let } y = 5 \text{ in let } f = \text{fun } x \rightarrow x+y \text{ in let } y = 6 \text{ in } f \ 7 \Rightarrow 13 \)