We've looked at several formal methods for defining the syntax of a programming language
- Regular expressions
- Context-free grammars

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

Formal Semantics
- Formal semantics of a programming language
  - Mathematical model 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 parts of language into another language
    - Usually a mathematical function
    - Equivalent to compilation
- Operational semantics
  - Describe effect of parts of language
    - Usually on (a mathematical model of) an abstract machine
    - For lambda calculus, can use syntactic transformations
    - Equivalent to interpretation
- Axiomatic semantics
  - Describe each part of language through logical axioms

Operational Semantics
- We will briefly look at operational semantics
  - Using a subset of OCaml as an example

Useful for
- Specifying the meaning of a program
- Proving the correctness of an algorithm
  - Through formal verification
    - For cryptographic algorithms, combinatorial circuits, etc...
    - Currently limited to smaller programs
Roadmap: Semantics of a Program

Grammar:
P * T | P
d | n | (E)
T + E | T
Parser:
(recursive descent)

Program: X = 2 + 3
Program Semantics:

Evaluation
- We’re going to define a relation E → v
  • This means “expression E evaluates to v”
- So we need a formal way of defining programs and of defining things they may evaluate to
- We’ll use grammars to describe each of these
  • One to describe abstract syntax trees E
  • One to describe OCaml values v

OCaml Programs
- E ::= x | n | true | false | [] | if E then E else E
  | fun x = E | E E
  • x stands for any identifier
  • n stands for any integer
  • true and false stand for the two boolean values
  • [] is the empty list
  • Using = in fun instead of -→ to avoid some confusion later

Values
- v ::= n | true | false | [] | v::v
  • n is an integer (not a string corresp. to an integer)
  • Same idea for true, false, []
  • v1::v2 is the pair with v1 and v2
  • This will be used to build up lists
  • Notice: nothing yet requires v2 to be a list
  • Important: Be sure to understand the difference between program text S and mathematical objects v
  • E.g., the text 3 evaluates to the mathematical number 3
  • To help, we’ll use different colors and italics
  • If not present, it’s up to you to remember which is which

Grammars for Trees
- We’re just using grammars to describe trees
  E ::= x | n | true | false | [] | if E then E else E
  | fun x = E | E E
  v ::= n | true | false | [] | v::v

Given a program, we know how to convert it to an AST using recursive descent parsing

Operational Semantics Rules

| n → n |
| true → true |
| false → false |
| [] → [] |

Each basic entity evaluates to its corresponding value

Goal: For any AST, we want an operational rule to obtain a value that represents the execution of that AST
Operational Semantics Rules (cont.)

- How about built-in functions?
  - $(+ \ n \ m) \rightarrow n + m$
  - We're applying the $+$ function
    - We put parens around it because it's not in infix notation
    - Will skip this from now on
    - Ignore currying for the moment
  - On the right-hand side, we're computing the mathematical sum; the left-hand side is source code
  - But what about $(+ 3 4) 5$?
    - We need recursion

Error Cases

\[
\frac{E_1 \rightarrow n \quad E_2 \rightarrow m}{+ \ E_1 \ E_2 \rightarrow n + m}
\]

- What if $E1$ and $E2$ aren’t integers?
  - E.g., what if we write $+ \ false \ true$?
  - It can be parsed, but we can’t execute it
- Previous rule does not cover such a case
  - Because we wrote $n, m$ in the hypothesis
    - So they must be integers
- Convention
  - If there is no rule to cover a case
    - Then the expression is erroneous
  - A program that evaluates to an erroneous expression
    - Produces a run-time error in practice

Rules with Hypotheses

- To evaluate $+ \ E_1 \ E_2$, we need to evaluate $E_1$, then evaluate $E_2$, then add the results
  - This is call-by-value
    \[
    \frac{E_1 \rightarrow n \quad E_2 \rightarrow m}{+ \ E_1 \ E_2 \rightarrow n + m}
    \]
  - This is a “natural deduction” style rule
  - It says that if the hypotheses above the line hold, then the conclusion below the line holds
    - i.e., if $E_1$ executes to value $n$ and if $E_2$ executes to value $m$, then $+ \ E_1 \ E_2$ executes to value $n + m$

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 If

\[
\begin{align*}
\color{red}{E_1 \rightarrow true} & \quad \color{red}{E_2 \rightarrow v} \\
\text{if } E_1 \text{ then } E_2 \text{ else } E_3 & \rightarrow v\\
\color{red}{E_1 \rightarrow false} & \quad \color{red}{E_3 \rightarrow v} \\
\text{if } E_1 \text{ then } E_2 \text{ else } E_3 & \rightarrow v
\end{align*}
\]

- Examples
  - if false then 3 else 4 $\rightarrow 4$
  - if true then 3 else 4 $\rightarrow 3$
- Notice that only one branch is evaluated

Rule for ::

\[
\frac{E_1 \rightarrow v_1 \quad E_2 \rightarrow v_2}{:: \ E_1 \ E_2 \rightarrow v_1::v_2}
\]

- So :: allocates a pair in memory
- Are there any conditions on $E_1$ and $E_2$?
  - No! We will allow $E_2$ to be anything
  - OCaml’s type system will disallow non-lists
Rules for Identifiers

- Let’s assume for now that the only identifiers are parameter names
  - Example: \((\text{fun } x = + x 3) \ 4\)
  - When we see \(x\) in the body, we need to look it up
  - So we need to keep some sort of environment
    - This will be a map from identifiers to values

Semantics with Environments

- Extend rules to the form \(A; E \rightarrow v\)
  - Means in environment \(A\), program text \(E\) evaluates to \(v\)

Notation

- We write \(\cdot\) for the empty environment (may be omitted)
- We write \(A(x)\) for the value that \(x\) maps to in \(A\)
- We write \(A, x:v\) for the same environment as \(A\), except \(x\) is now \(v\)
- \(x\) might or might not have mapped to anything in \(A\)
- We write \(A, A'\) for the environment with the bindings of \(A'\) added to and overriding the bindings of \(A\)

Rules for Identifiers and Application

\[
\frac{A; x \rightarrow A(x)}{A; E \rightarrow v \quad A, x:v; E_1 \rightarrow v'}
\]

- To evaluate a user-defined function applied to an argument:
  - Evaluate the argument (call-by-value)
  - Evaluate the function body in an environment in which the formal parameter is bound to the actual argument
  - Return the result

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

\[
\begin{align*}
\cdot; \cdot; 4 & \rightarrow 4 \\
\cdot; \cdot; 4; + x 3 & \rightarrow 7 \\
\cdot; (\text{fun } x = + x 3) \ 4 & \rightarrow 7
\end{align*}
\]

Nested Functions

- This works for cases of nested functions
  - …as long as they are fully applied

- But what about the true higher-order cases?
  - Passing functions as arguments, and returning functions as results
  - We need closures to handle this case
  - …and a closure was just a function and an environment
  - We already have notation around for writing both parts

Closures

- Formally, we add closures \((A, \lambda x . E)\) to values
  - \(A\) is the environment in which the closure was created
  - \(x\) is the parameter name
  - \(E\) is the source code for the body
- \(\lambda x\) is a binding of \(x\) in \(E\)
- \(v ::= n \mid \text{true} \mid \text{false} \mid [] \mid v::v \mid (A, \lambda x . E)\)
Revised Rule for Lambda

\[ A; \text{fun } x = E \rightarrow (A, \lambda x.E) \]

- To evaluate a function definition, create a closure when the function is created
  - Notice that we don’t look inside the function body

Revised Rule for Application

\[ A; E_1 \rightarrow (A', \lambda x.E) \quad A; E_2 \rightarrow v \]
\[ \quad A, A', x: v; E \rightarrow v' \]

- To apply something to an argument:
  - Evaluate it to produce a closure
  - Evaluate the argument (call-by-value)
  - Evaluate the body of the closure, in
    - The current environment, extended with the closure’s environment, extended with the binding for the parameter

Example

\[ \ast; (\text{fun } x = (\text{fun } y = + x y)) \rightarrow (^*; \lambda x.(\text{fun } y = + x y)) \]
\[ \quad \ast; 3 \rightarrow 3 \]
\[ x:3; (\text{fun } y = + x y) \rightarrow (x:3, \lambda y.(+ x y)) \]

\[ \ast; (\text{fun } x = (\text{fun } y = + x y)) \cdot 3 \rightarrow (x:3, \lambda y.(+ x y)) \]

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

Example (cont.)

\[ \ast; \text{<previous>} \rightarrow (x:3, \lambda y.(+ x y)) \]
\[ \ast; 4 \rightarrow 4 \]
\[ x:3, y:4; (+ x y) \rightarrow 7 \]

\[ \ast; (\text{<previous}> 4) \rightarrow 7 \]

Why Did We Do This?

- Operational semantics are useful for
  - Describing languages
    - Not just OCaml! It’s pretty hard to describe a big language like C or Java, but we can at least describe the core components of the language
  - Giving a precise specification of how they work
    - Look in any language standard – they tend to be vague in many places and leave things undefined
  - Reasoning about programs
    - We can actually prove that programs do something or don’t do something, because we have a precise definition of how they work