

CMSC 330: Organization of Programming Languages

Operational Semantics

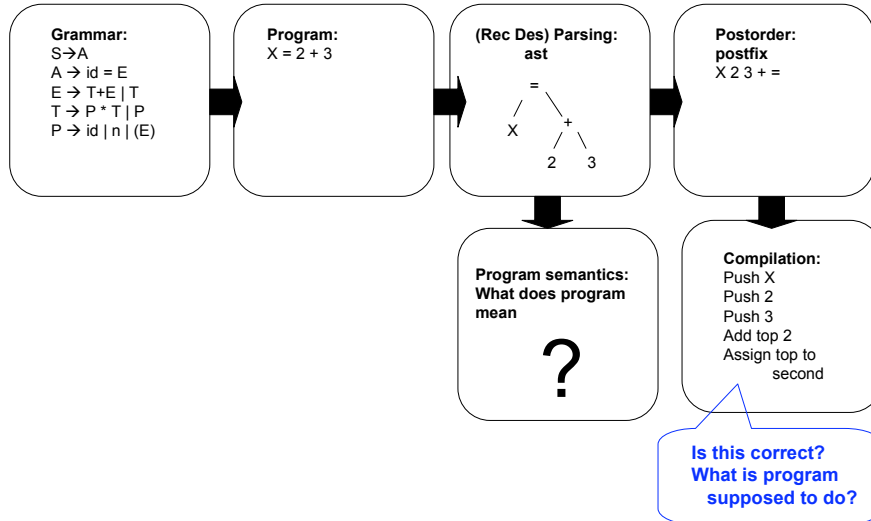
Introduction

- So far we've looked at regular expressions, automata, and context-free grammars
 - These are ways of defining sets of strings
 - We can use these to describe what programs you can write down in a language
 - (Almost...)
 - I.e., these describe the *syntax* of a language
- What about the *semantics* of a language?
 - What does a program “mean”?

CMSC 330

2

Roadmap: Compilation of program



CMSC 330

3

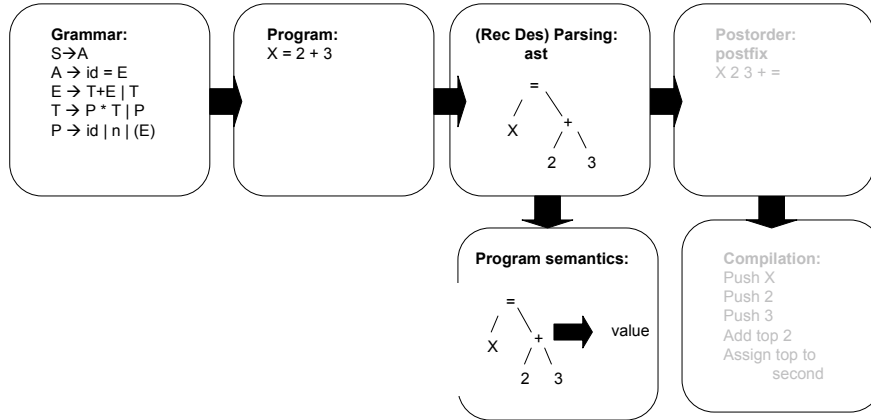
Operational Semantics

- There are several different kinds of semantics
 - *Denotational*: A program is a mathematical function
 - *Axiomatic*: Develop a logical proof of a program
 - Give predicates that hold when a program (or part) is executed
- We will briefly look at *operational semantics*
 - A program is defined by how you execute it on a mathematical model of a machine
 - Operational semantics are easy to understand
- We will look at a subset of OCaml as an example

CMSC 330

4

Roadmap: Semantics of a program



Evaluation

- We're going to define a relation $E \rightarrow v$
 - 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

- $E ::= x \mid n \mid true \mid false \mid [] \mid \text{if } E \text{ then } E \text{ else } E$
 $\mid \text{fun } x = E \mid E E$
 - 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 \rightarrow to avoid some confusion later

Values

- $v ::= n \mid true \mid false \mid [] \mid v::v$
 - 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
 - This is usually not done, and it's up to the reader 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
```

```
type ast =
  | Id of string
  | Num of int
  | Bool of bool
  | Nil
  | If of ast * ast * ast
  | Fun of string * ast
  | App of ast * ast
```

Given a program, we saw last time how to convert it to an ast (e.g., recursive descent parsing)

```
type value =
  | Val_Num of int
  | Val_Bool of bool
  | Val_Nil
  | Val_Pair of value * value
```

Goal: For any ast, we want an operational rule to obtain a value that represents the execution of ast

Operational Semantics Rules

$n \rightarrow n$

$\text{true} \rightarrow \text{true}$

$\text{false} \rightarrow \text{false}$

$[] \rightarrow []$

- Each basic entity evaluates to the corresponding value

Operational Semantics Rules (cont'd)

- 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, and pretend we have multi-argument functions

- 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

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 $n+m$

Error Cases

$$\frac{E_1 \rightarrow n \quad E_2 \rightarrow m}{+ E_1 E_2 \rightarrow n + m}$$

- Because we wrote n, m in the hypothesis, we mean that they must be integers
- But what if E_1 and E_2 aren't integers?
 - E.g., what if we write $+ \text{false true}$?
 - It can be parsed, but we can't execute it
- We will have no rule that covers such a case
 - Convention: If there is not rule to cover a case, then the expression is erroneous
 - A program that evaluates to a stuck expression produces a run time error in practice

Trees of Semantic Rules

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

$$\frac{3 \rightarrow 3 \quad 4 \rightarrow 4}{(+ 3 4) \rightarrow 7} \quad 5 \rightarrow 5$$
$$\frac{(+ 3 4) \rightarrow 7 \quad 5 \rightarrow 5}{+ (+ 3 4) 5 \rightarrow 12}$$

Rules for If

$$\frac{E_1 \rightarrow \text{true} \quad E_2 \rightarrow v}{\text{if } E_1 \text{ then } E_2 \text{ else } E_3 \rightarrow v}$$

$$\frac{E_1 \rightarrow \text{false} \quad E_3 \rightarrow v}{\text{if } E_1 \text{ then } E_2 \text{ else } E_3 \rightarrow v}$$

- Examples
 - $\text{if false then } 3 \text{ else } 4 \rightarrow 4$
 - $\text{if true then } 3 \text{ else } 4 \rightarrow 3$
- 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 Hd and Tl

$$\frac{E \rightarrow v_1 :: v_2}{\text{hd } E \rightarrow v_1}$$

$$\frac{E \rightarrow v_1 :: v_2}{\text{tl } E \rightarrow v_2}$$

Rules for Identifiers

$$\frac{}{x \rightarrow ???}$$

- Let's assume for now that the only identifiers are parameter names
 - Ex. (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 , the program text E evaluates to v
- Notation:
 - We write \bullet for the empty environment
 - 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
 - The empty environment can be omitted when things are clear, and in adding other bindings to an empty environment we can write just those bindings if things are clear

Rules for Identifiers and Application

$$\frac{}{A; x \rightarrow A(x)}$$

no hypothesis means
"in all cases"

$$\frac{A; E_2 \rightarrow v \quad A, x:v; E_1 \rightarrow v'}{A; (\text{fun } x = E_1) E_2 \rightarrow v'}$$

- 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: (fun x = + x 3) 4 = ?

$$\begin{array}{l} \bullet; 4 \rightarrow 4 \qquad \bullet, x:4; x \rightarrow 4 \quad \bullet, x:4; 3 \rightarrow 3 \\ \hline \bullet, x:4; + x 3 \rightarrow 7 \\ \hline \bullet; (\text{fun } x = + x 3) 4 \rightarrow 7 \end{array}$$

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
- λx will be discussed next time. Means 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

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

- 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

$$\frac{A; E_1 \rightarrow (A', \lambda x.E) \quad A; E_2 \rightarrow v \quad A, A', x:v; E \rightarrow v'}{A; (E_1 E_2) \rightarrow v'}$$

- 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

CMSC 330

25

Example

$$\begin{aligned} & \bullet; (\text{fun } x = (\text{fun } y = + x y)) \rightarrow (\bullet, \lambda x. (\text{fun } y = + x y)) \\ & \qquad \bullet; 3 \rightarrow 3 \\ & \qquad x:3; (\text{fun } y = + x y) \rightarrow (x:3, \lambda y. (+ x y)) \\ & \bullet; (\text{fun } x = (\text{fun } y = + x y)) 3 \rightarrow (x:3, \lambda y. (+ x y)) \end{aligned}$$

Let <previous> = (fun x = (fun y = + x y)) 3

CMSC 330

26

Example (cont'd)

$$\begin{aligned} & \bullet; \text{<previous>} \rightarrow (x:3, \lambda y. (+ x y)) \\ & \qquad \bullet; 4 \rightarrow 4 \\ & \qquad x:3, y:4; (+ x y) \rightarrow 7 \\ & \bullet; (\text{<previous>} 4) \rightarrow 7 \end{aligned}$$

CMSC 330

27

Why Did We Do This? (cont'd)

- 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

CMSC 330

28