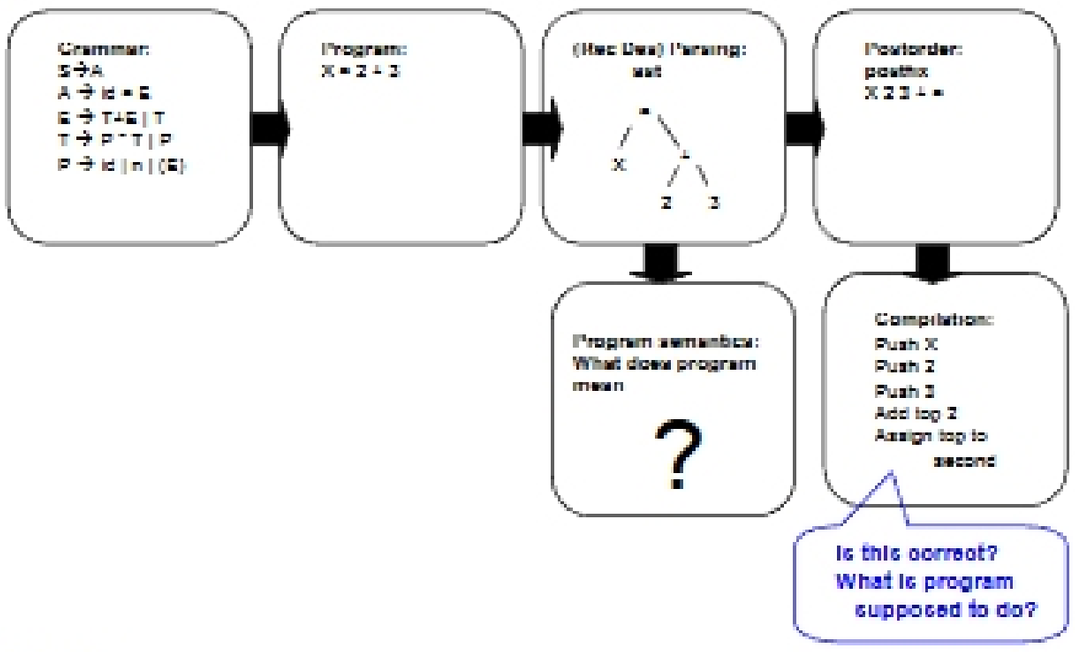
CMSC 330: Organization of Programming Languages

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"?

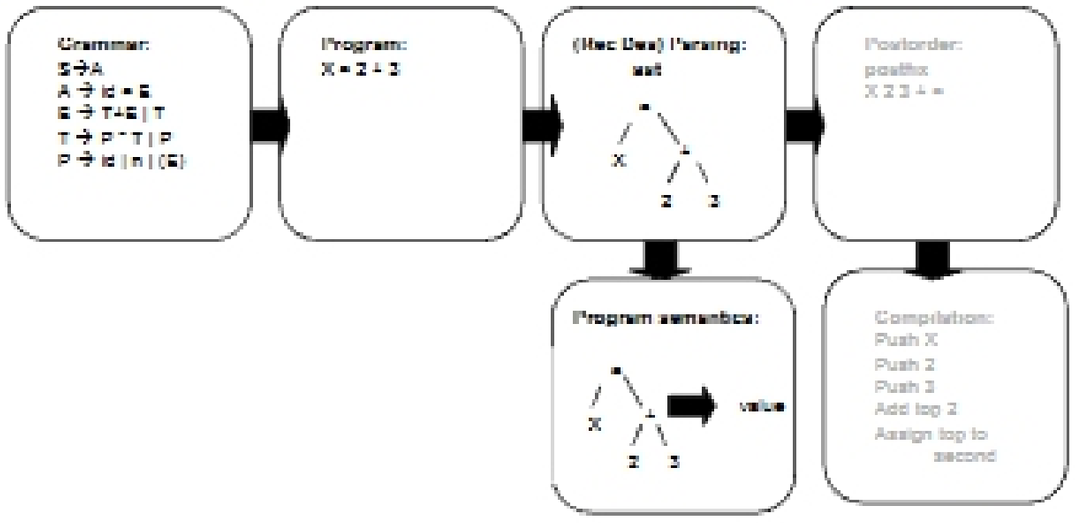
Roadmap: Compilation of program



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

Roadmap: Semantics of a program



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
    - 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

$$n \quad n$$

$$\text{true} \quad \textit{true}$$

$$\text{false} \quad \textit{false}$$

$$[] \quad \textit{[]}$$

- Each basic entity evaluates to the corresponding value

Operational Semantics Rules (cont'd)

- How about built-in functions?

$$(\ +\ )\ n\ m \quad n + m$$

  - 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

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

  - 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*