CMSC 330: Organization of Programming Languages

Operational Semantics

Introduction

- We looked at methods for describing syntax formally- regular expressions, automata, and context-free grammars
  – These are ways of defining sets of strings
  – We can use these to describe (almost) what programs you can write in a language
- What about the *semantics* of a language?
  – What does a program "mean"?

Operational Semantics

- There are several different ways of describing semantics:
  – *Denotational*: Views a program as a mathematical function
  – *Axiomatic*: Gives predicates that hold when a program (or part) is executed (describing program parts using logical axioms)
- We will briefly look at *operational semantics*
  – In operational semantics, a program's result is defined by how you execute it on a mathematical model of a machine
  – Operational semantics are easy to understand

Evaluation

- We're going to define a relation $S \rightarrow v$
  – This means "program code S evaluates to value 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 S
  – One to describe Scheme values v

Scheme Programs

- S ::= n | #t | #f | "str" | nil | id | (T)
  T ::= T S | S
  - n stands for an integer
  - "str" stands for any string
  - id stands for any identifier
    - Including user-defined functions
      - (define next (lambda (x) (+ x 1)))
    - And primitives
      - (+ 3 4)   ; + is an identifier

Values

- *v ::= n | true | false | "str" | nil | (v, v)*
  - *n* is an integer (*not* a string corresponding to an integer)
    - Same idea for *true*, *false*, *"str"*, *nil*
  - *(v, v)* is a pair of values (called a "cons" cell)
  - **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
  - S ::= n | #t | #f | "str" | nil | id | (T)
    T ::= T S | S
  - *v ::= n | true | false | "str" | nil | (v, v)*
- If we wanted to write an OCaml program to manipulate Scheme (such as an interpreter), we could use these types to store code and values:

```
type ast =
    Num of int
  | Bool of bool
  | String of string
  | Id of string
  | List of ast list
```

```
type value =
    Val_Num of int
  | Val_Bool of bool
  | Val_String of string
  | Val_Nil
  | Val_Cons of value * value
```


Operational Semantics Rules

n → *n*

#t → *true*

#f → *false*

"str" → *"str"*

nil → *nil*

- Each basic entity evaluates to the corresponding value

Operational Semantics Rules (cont'd)

- How about built-in functions?

$$(+\ \mathrm{n}\ \mathrm{m}) \rightarrow n + m$$

  - On the right-hand side, we're computing the mathematical sum; the left-hand side is Scheme source code
  - But what about (+ (+ 3 4) 5) ?
    - We need recursion

Rules with Hypotheses

- To evaluate $(+\ S_1\ S_2)$, we need to evaluate S_1, then evaluate S_2, then add the results
  - Scheme is call-by-value

$$\frac{S_1 \rightarrow n \qquad S_2 \rightarrow m}{(+\ S_1\ S_2) \rightarrow n + m}$$

  - This is a "natural deduction" style rule
  - It says that if the *hypotheses* above the line hold, then the *conclusion* below the line holds

Error Cases

$$\frac{S_1 \rightarrow n \qquad S_2 \rightarrow m}{(+\ S_1\ S_2) \rightarrow n + m}$$

- Because we wrote *n*, *m* in the hypothesis, we mean that they must be integers
- But what if S_1 and S_2 aren't integers?
  - E.g., what if we write (+ #f #t)?
  - 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 a 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, which corresponds to the recursive evaluation procedure

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

Rules for If

$$\frac{S_1 \rightarrow \textit{true} \qquad S_2 \rightarrow v}{(\text{if } S_1\ S_2\ S_3) \rightarrow v}$$

$$\frac{S_1 \rightarrow \textit{false} \qquad S_3 \rightarrow v}{(\text{if } S_1\ S_2\ S_3) \rightarrow v}$$

- Examples
  - (if #f 3 4) → *4*
  - (if #t 3 4) → *3*
- Notice that only one branch is evaluated

Why Did We Do This?

- Operational semantics are useful for
  - Describing languages
    - Not just Scheme! 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
  - Note that we could extend this to give semantics to the remaining parts of Scheme