CMSC 330: Organization of Programming Languages

More on Scope
Operational Semantics

Tail Calls

- A *tail call* is a function call that is the last thing a function does before it returns

```
let add x y = x + y
let f z = add z z (* tail call *)
```

```
let rec length = function
   [] -> 0
 | (_::t) -> 1 + (length t) (* not a tail call *)
```

```
let rec length a = function
   [] -> a
 | (_::t) -> length (a + 1) t (* tail call *)
```

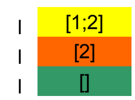

Tail Recursion

- Recall that in OCaml, all looping is via recursion
  - Seems very inefficient
  - Needs one stack frame for recursive call
- A function is *tail recursive* if it is recursive and the recursive call is a tail call

Tail Recursion (cont'd)

```
let rec length l = match l with
     [] -> 0
   | (_::t) -> 1 + (length t)

length [1; 2]
```



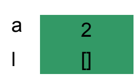
eax: 2

- However, if the program is tail recursive...
  - Can instead reuse stack frame for each recursive call

Tail Recursion (cont'd)

```
let rec length a l = match l with
     [] -> a
   | (_::t) -> (length (a + 1) t)

length 0 [1; 2]
```



eax: 2

- The same stack frame is reused for the next call, since we'd just pop it off and return anyway

Names and Binding

- Programs use *names* to refer to things
  - E.g., in x = x + 1, x refers to a variable
- A *binding* is an association between a name and what it refers to
  - `int x;                /* x is bound to a stack location containing an int */`
  - `int f (int) { ... }  /* f is bound to a function */`
  - `class C { ... }       /* C is bound to a class */`
  - `let x = e1 in e2      (* x is bound to e1 *)`

Name Restrictions

- Languages often have various restrictions on names to make lexing and parsing easier
  - Names cannot be the same as keywords in the language
  - OCaml function names must be lowercase
  - OCaml type constructor and module names must be uppercase
  - Names cannot include special characters like ; , : etc
    - Usually names are upper- and lowercase letters, digits, and _ (where the first character can't be a digit)
    - Some languages also allow more symbols like ! or -

Names and Scopes

- Good names are a precious commodity
  - They help document your code
  - They make it easy to remember what names correspond to what entities
- We want to be able to reuse names in different, non-overlapping regions of the code

Names and Scopes (cont'd)

- A *scope* is the region of a program where a binding is active
  - The same name in a different scope can refer to a different binding (refer to a different program object)
- A name is *in scope* if it's bound to something within the particular scope we're referring to

Example

```
void w(int i) {
 ...
}

void x(float j) {
 ...
}

void y(float i) {
 ...
}

void z(void) {
  int j;
  char *i;
  ...
}
```

- i is in scope
  - in the body of w, the body of y, and after the declaration of j in z
  - but all those i's are different
- j is in scope
  - in the body of x and z

Ordering of Bindings

- Languages make various choices for when declarations of things are in scope

Order of Bindings – OCaml

- let x = e1 in e2 – x is bound to e1 in scope of e2
- let rec x = e1 in e2 – x is bound in e1 and in e2

```
let x = 3 in
  let y = x + 3 in...    (* x is in scope here *)
```

```
let x = 3 + x in ...    (* error, x not in scope *)
```

```
let rec length = function
    [] -> 0
  | (h::t) -> 1 + (length t)   (* ok, length in scope *)
in ...
```

Order of Bindings – C

- All declarations are in scope from the declaration onward

```
int i;
int j = i;  /* ok, i is in scope */
i = 3;      /* also ok */
```

```
void f(...) { ... }

int i;
int j = j + 3;   /* error */
f(...);          /* ok, f declared */
```

```
f(...);  /* may be error; need prototype (or oldstyle C) */

void f(...) { ... }
```


Order of Bindings – Java

- Declarations are in scope from the declaration onward, except for methods and fields, which are in scope throughout the class

```
class C {
  void f(){
    ...g()...    // OK
  }

  void g(){
    ...
  }
}
```


Shadowing Names

- *Shadowing* is rebinding a name in an inner scope to have a different meaning
  - May or may not be allowed by the language

C

```
int i;

void f(float i) {

  {
    char *i = NULL;
    ...
  }

}
```

OCaml

```
let g = 3;;
let g x = x + 3;;
```

Java

```
void h(int i) {
  {
    float i; // not allowed
    ...
  }
}
```


Namespaces

- Languages have a "top-level" or outermost scope
  - Many things go in this scope; hard to control collisions
- Common solution seems to be to add a hierarchy
  - OCaml: Modules
    - List.hd, String.length, etc.
    - open to add names into current scope
  - Java: Packages
    - java.lang.String, java.awt.Point, etc.
    - import to add names into current scope
  - C++: Namespaces
    - namespace f { class g { ... } }, f::g b, etc.
    - using namespace to add names to current scope

Mangled Names

- What happens when these names need to be seen by other languages?
  - What if a C program wants to call a C++ method?
    - C doesn't know about C++'s naming conventions
- For multilingual communication, names are often mangled into some flat form
  - E.g., `class C { int f(int *x, int y) { ... } }` becomes symbol `__ZN1C3fEPii` in g++
  - E.g., native valueOf(int) in java.lang.String corresponds to the C function `Java_java_lang_String_valueOf__I`

Static Scope Recall

- In *static scoping*, a name refers to its closest binding, going from inner to outer scope in the program text
  - Languages like C, C++, Java, Ruby, and OCaml are statically scoped

```
int i;

{
  int j;

  {
    float i;

    j = (int) i;
  }
}
```

Free and Bound Variables

- The *bound variables* of a scope are those names that are declared in it
- If a variable is not bound in a scope, it is *free*
  - The bindings of variables which are free in a scope are "inherited" from declarations of those variables in outer scopes in static scoping

```
{ /* 1 */
  int j;

  { /* 2 */
    float i;

    j = (int) i;
  }
}
```

j is bound in scope 1

i is bound in scope 2

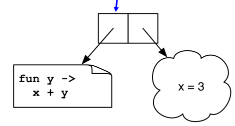
j is free in scope 2

Static Scoping and Nested Functions

- To allow arbitrary nested functions with higher-order functions and static scoping, we needed closures

```
let add x = (fun y -> x + y)
```

```
(add 3) 4  → <closure> 4  → 3 + 4  → 7
```

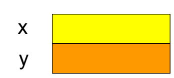


Nested Functions (cont'd)

- We need closures for *upward funargs*
  - Functions that are returned by other functions
- If we only have *downward funargs*, then we don't need full closures
  - These are functions that are only passed inward
  - So when they're called, any nonlocal variables they access from outer scopes are still around

Example

```
let f x =
  let g y = x + y in
    g 3
```

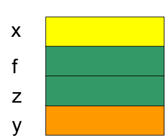


- When g is called, x is still on the stack

Example

```
let app f z = f z

let f x =
  let g y = x + y in
    app g 3
```



- When g is called, x is still on the stack

Downward Funargs

- It turns out that if we only pass functions downward, there are cheaper implementation strategies for static scoping than closures
- They're called *static links* and *displays*, and they're used by
  - Pascal and Algol-family languages
  - gcc nested functions
- We won't go into details, though (CMSC 430 covers these in exciting detail.)

Dynamic Scope

- In a language with *dynamic scoping*, a name refers to its closest binding *at runtime*
  - LISP was the common example

```
Scheme (top-level scope only is dynamic)

(define f (lambda () a))
; defines a no-argument function which returns a

(define a 3)    ; bind a to 3
(f)             ; calls f and returns 3
(define a 4)
(f)             ; calls f and returns 4
```


Nested Dynamic Scopes

- Full dynamic scopes can be nested
  - Static scope relates to the program text
  - Dynamic scope relates to program execution trace

```
Perl (the keyword local introduces dynamic scope)

$l = "global";

sub A {
  local $l = "local";
  B();
}
                                    global
sub B { print "$l\n"; }             local
                                    global
B(); A(); B();
```


Static vs. Dynamic Scope

Static scoping	Dynamic scoping
– Local understanding of function behavior	– Can be hard to understand behavior of functions
– Know at compile-time what each name refers to	– Requires finding name bindings at runtime
– A bit trickier to implement	– Easier to implement (just keep a global table of stacks of variable/value bindings)

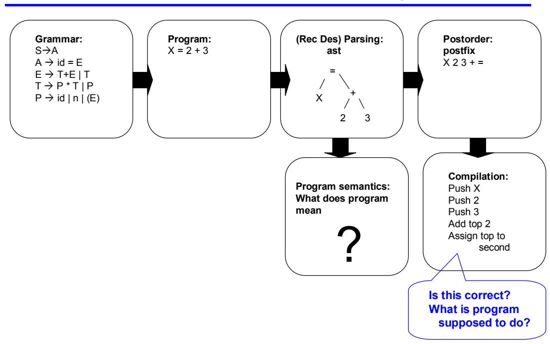
CMSC 330: Organization of Programming Languages

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"?

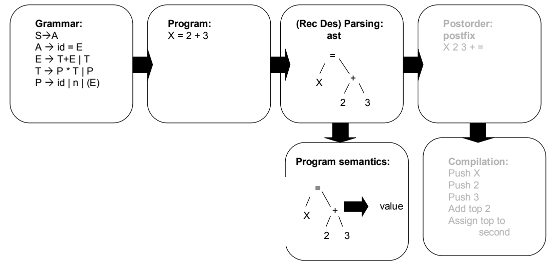
Roadmap: Compilation of program



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
- 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

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

```
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
```

```
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

$\text{n} \rightarrow n$

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

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

$[] \rightarrow []$

- Each basic entity evaluates to the corresponding value

Operational Semantics Rules (cont'd)

- How about built-in functions?

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

- 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

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

  - 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

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

- 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 + 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

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

Rules for If

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

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

- Examples
  - if false then 3 else 4 → *4*
  - if true then 3 else 4 → *3*
- Notice that only one branch is evaluated

Rule for ::

$$\frac{E_1 \to v_1 \qquad E_2 \to v_2}{::\ E_1\ E_2 \to v_1{::}v_2}$$

- 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

$$\frac{}{x \to ???}$$

- 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 → *v*
  - Means in environment A, the program text E evaluates to *v*
- Notation:
  - We write • 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

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

no hypothesis means "in all cases"

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

- 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 = ?

$$\frac{\bullet;\ 4 \to 4 \qquad \dfrac{\bullet, x{:}4;\ x \to 4 \qquad \bullet, x{:}4;\ 3 \to 3}{\bullet, x{:}4;\ +\ x\ 3 \to 7}}{\bullet;\ (\text{fun } x = +\ x\ 3)\ \ 4 \to 7}$$

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, λx.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 | true | false | [] | v::v*
  | (A, λx.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

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

- 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

Example

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

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

Example (cont'd)

$$\frac{\bullet;\ \langle\text{previous}\rangle \rightarrow (x{:}3,\ \lambda y.(+\ x\ y)) \qquad \bullet;\ 4 \rightarrow 4 \qquad x{:}3,\ y{:}4;\ (+\ x\ y) \rightarrow 7}{\bullet;\ (\ \langle\text{previous}\rangle\ 4\ ) \rightarrow 7}$$

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