Implementing OCaml in OCaml
Part 1: Representing Program Syntax

Speaker: David Walker
COS 326
Princeton University
To write a program, you have to know how the language works.

**Semantics**: The study of “how a programming language works”
Defining Programming Language Semantics

To write a program, you have to know how the language works.

Semantics: The study of “how a programming language works”

Methods for defining program semantics:

– Operational: show how to rewrite program expressions step-by-step until you end up with a value
  • we’ve done some of this already
– Denotational: how interpret a program in a different language that is well understood
  • we aren’t going to do much of this – see COS 510
– Equational: specify the equal programs
  • we’ll do more of this later & use this semantics to prove things about our programs
– Axiomatic: provide (other kinds of) reasoning rules about programs
In this series of lectures, we’ll focus on operational definitions.

We’ll use the following techniques to communicate:

1. *examples* (good for intuition, but highly incomplete)
   – this doesn’t get at the corner cases
2. *an interpreter program* written in OCaml
3. *mathematical notation*
Defining Program Semantics

In this series of lectures, we’ll focus on operational definitions.

We’ll use the following techniques to communicate:

1. *examples* (good for intuition, but highly incomplete)
   - this doesn’t get at the corner cases

2. *an interpreter program* written in OCaml

3. *mathematical notation*
Implementing an Interpreter

let x = 3 in
x + x

data structure representing program

Let ("x",
Num 3,
Binop(Plus, Var "x", Var "x"))

data structure representing result of evaluation

Num 6

Parsing

Evaluation

Pretty Printing

6

text file/stdout containing formatted output

text file containing program as a sequence of characters
REPRESENTING SYNTAX
Representing Syntax

Program syntax is a complicated tree-like data structure.
Representing Syntax

Program syntax is a complicated tree-like data structure.

let x = 3 in
x + x
Syntax Trees

This is the parse tree.
Useful for some purposes, but for the semantics it’s too much information.
Don’t need all the “punctuation” (key words, white space).

let x = 3 in
x + x
More generally each let expression has 3 parts:
More generally each let expression has 3 parts:

And you can represent a let expression using a tree like this:
More generally each let expression has 3 parts:

And you can represent a let expression using a tree like this:

this part has to contain a variable, like $x$

these parts contain arbitrary subexpressions
Representing Syntax

More generally each let expression has 3 parts:

\[
\text{let } \quad \text{= } \quad \text{in }
\]

And you create complicated programs by nesting let expressions (or any other expression) recursively inside one another:
OCaml for the Win

Functional programming languages have sometimes been called “domain-specific languages for compiler writers”

Datatypes are amazing for representing complicated tree-like structures and that is exactly what a program is.

Use a different constructor for every different sort of expression
- one constructor for variables
- one constructor for let expressions
- one constructor for numbers
- one constructor for binary operators, like add
- ...

Languages like Java, that are based exclusively around heavy-weight class tend to be vastly more verbose when trying to represent syntax trees:

- one whole class for each different kind of syntax
- one class for variables
- one class for let expressions
- one class for numbers ...

In addition, writing traversals over the syntax is annoying, because your code is spread over N different classes (using a visitor pattern) rather than in one place.
Aside: Java for the loss

Languages like Java, that are based exclusively around heavy-weight class tend to be vastly more verbose when trying to represent syntax trees:

- one whole class for each distinct kind of syntax
- one class for variables
- one class for let expressions
- one class for numbers...

In addition, writing traversals over the syntax is annoying, because all code is scattered into different classes (using a visitor pattern), rather being in one place.

SCORE: OCAML 3.8, JAVA 0

(C: who cares?)
A datatype for simple OCaml expressions:

```ocaml
type variable = string

type op = Plus | Minus | Times | ...

type exp =
  | Int of int
  | Op of exp * op * exp
  | Var of variable
  | Let of variable * exp * exp

type value = exp
```
A datatype for simple OCaml expressions:

```ocaml
type variable = string

type op = Plus | Minus | Times | ...

type exp =
  | Int of int
  | Op of exp * op * exp
  | Var of variable
  | Let of variable * exp * exp

type value = exp

let e1 = Int 3
```
A datatype for simple OCaml expressions:

```ocaml
type variable = string
type op = Plus | Minus | Times | ...
type exp =
  | Int of int
  | Op of exp * op * exp
  | Var of variable
  | Let of variable * exp * exp
type value = exp

let e1 = Int 3
let e2 = Int 17
```
A datatype for simple OCaml expressions:

```ocaml
type variable = string

type op = Plus | Minus | Times | ...

type exp =
  | Int of int
  | Op of exp * op * exp
  | Var of variable
  | Let of variable * exp * exp

type value = exp

let e1 = Int 3
let e2 = Int 17
let e3 = Op (e1, Plus, e2)
```

represents “3 + 17”
We can represent the OCaml program:

```
let x = 30 in
let y =
  (let z = 3 in
   z*4)
in
y+y
```

as an exp value:

```
Let("x", Int 30, 
  Let("y", 
    Let("z", Int 3, 
      Let("z", Int 3, 
        Op(Var "z", Times, Int 4)), 
      Op(Var "y", Plus, Var "y"))
  )
)
Let(“x”, Int 30,
    Let(“y”, Let(“z”, Int 3,
        Op(Var “z”, Times, Int 4)),
        Op(Var “y”, Plus, Var “y”))

Visualizing the OCaml expression as a tree
Let("x", Int 30, 
   Let("y", Let("z", Int 3, 
         Op(Var "z", Times, Int 4)), 
         Op(Var "y", Plus, Var "y"))

Now that we have a data structure to represent programs, we can write other programs to analyze them.
Free vs Bound Variables

let x = 30 in
x + y
let x = 30 in
x + y

this use of x is bound here
Free vs Bound Variables

let x = 30 in x + y

this use of y is free

we say: "y is a free variable in the expression (let x = 30 in x + y)"
Other Examples

fun \( z \rightarrow z + y \)

- \( z \) is bound
- \( y \) is a free variable

match \( x \) with
  (\( y, z \)) \rightarrow y + z + w

- \( x \), \( w \) are free variables
- \( y \), \( z \) are bound

let rec \( f \) \( x \) =
    match \( x \) with
    [] \rightarrow y
    | \( \text{hd} : \text{tl} \) \rightarrow \text{hd} :: f \text{ tl}

- \( y \) is a free variable
- \( f \), \( x \), \( \text{hd} \), \( \text{tl} \) are all bound
A Few More Examples

What are the free variables of the following expressions?

\[
\text{if true then } x \text{ else } y
\]

\[x \text{ and } y\]

\[
\text{(fun } x \ y \rightarrow \text{match } x \text{ with } \[] \rightarrow 0 \mid \text{hd}::\text{tl} \rightarrow w + \text{hd})[] z
\]

\[w \text{ and } z\]

The free variables of an expression do not depend upon the flow of control.
Given a variable occurrence, we can find where it is bound by ...

```
let a = 30 in
let a = (let a = 3 in a*4) in
a+a
```
let a = 30 in
let a =
  (let a = 3 in a*4)
in
a+a
crawling up the tree to the nearest enclosing let...

```
let a = 30 in
let a =
  (let a = 3 in a*4)
in
a+a
```
crawling up the tree to the nearest enclosing let...

let a = 30 in
let a =
    (let a = 3 in a*4)
in
a+a
and checking if the “let” binds the same variable – if so, we’ve found the nearest enclosing definition. If not, we keep going up.

```
let a = 30 in
let a =
  (let a = 3 in a*4)
in
a+a
```
We can also systematically rename the variables so that it’s not so confusing. Recall systematic renaming is called *alpha-conversion*. 

```
let a = 30 in
let a =
  (let a = 3 in a*4)
in
a+a
```
Start with a `let`, and pick a fresh variable name, say “x”

```
let a = 30 in
let a =
    (let a = 3 in a*4)
in a+a
```
Rename the binding occurrence from “a” to “x”.

```
let x = 30 in
let a =
  (let a = 3 in a*4)
in
a+a
```
Then rename all of the occurrences of the variables *that this let binds*.

```
let x = 30 in
let a =
    (let a = 3 in a*4)
in
a+a
```
There are none in this case!

```
let x = 30 in
let a =
  (let a = 3 in a*4)
in
a+a
```
There are none in this case!

```
let x = 30 in
let a =
  (let a = 3 in a*4)
in
a+a
```
Let’s do another let, renaming “a” to “\( y \)”. 

```plaintext
let \( x = 30 \) in
let \( a = (let a = 3 in a*4) \) in
a+a
```
Let’s do another let, renaming “a” to “y”.

```
let x = 30 in
let y =
    (let a = 3 in a*4)
in
y+y
```
Implementing Renaming

```ocaml
let rec rename (x:var) (y:var) (e:exp) : exp =
```

```ocaml
type var = string

type op = Plus | Minus

type exp =
   | Int of int
   | Op of exp * op * exp
   | Var of var
   | Let of var * exp * exp
```
type var = string

type op = Plus | Minus

type exp =
  | Int of int
  | Op of exp * op * exp
  | Var of var
  | Let of var * exp * exp

let rec rename (x:var) (y:var) (e:exp) : exp =
  match e with
  | Op (e1, op, e2) ->
  | Var z ->
  | Int i ->
  | Let (z,e1,e2) ->
type var = string

type op = Plus | Minus

type exp =
  | Int of int
  | Op of exp * op * exp
  | Var of var
  | Let of var * exp * exp

let rec rename (x:var) (y:var) (e:exp) : exp =
  match e with
  | Op (e1, op, e2) ->
    Op (rename x y e1, op, rename x y e2)
  | Var z ->
  | Int i ->
  | Let (z,e1,e2) ->
type var = string
type op = Plus | Minus

let rec rename (x:var) (y:var) (e:exp) : exp =
  match e with
  | Op (e1, op, e2) ->
    Op (rename x y e1, op, rename x y e2)
  | Var z ->
    if z = x then Var y else e
  | Int i ->
  | Let (z,e1,e2) ->
Implementing Renaming

```ocaml
let rec rename (x:var) (y:var) (e:exp) : exp =
  match e with
  | Op (e1, op, e2) ->
    Op (rename x y e1, op, rename x y e2)
  | Var z ->
    if z = x then Var y else e
  | Int i ->
    Int i
  | Let (z,e1,e2) ->
```

```ocaml
type var = string

type op = Plus | Minus

type exp =
  | Int of int
  | Op of exp * op * exp
  | Var of var
  | Let of var * exp * exp
```
type var = string

type op = Plus | Minus

type exp =
  | Int of int
  | Op of exp * op * exp
  | Var of var
  | Let of var * exp * exp

let rec rename (x:var) (y:var) (e:exp) : exp =
  match e with
  | Op (e1, op, e2) ->
    Op (rename x y e1, op, rename x y e2)
  | Var z ->
    if z = x then Var y else e
  | Int i ->
    Int i
  | Let (z,e1,e2) ->
    Let (z, rename x y e1,
         if z = x then e2 else rename x y e2)
Exercise

Here’s the syntax of our little language:

```ocaml
type var = string
type op = Plus | Minus
type exp =
  | Int of int
  | Op of exp * op * exp
  | Var of var
  | Let of var * exp * exp
```

Extending the abstract syntax of expressions. Extend the implementation of the renaming function.

- **(Easy)** Booleans true and false, if statements, and operations like and, or, not
- **(Harder)** Pairs and patterns “let (x,y) = e1 in e2”