CS 3110

Lecture 2: Introduction to OCaml Semantics

Prof. Clarkson

Fall 2014

Today's music: Prelude and Fugue in G minor, BWV 885, by J.S. Bach (1685-1750)
Michael Clarkson, live in concert, 1999

Review

- Recitation 1: Introduction to OCaml syntax
- OCaml Tutorial (once more tonight, 7:30 pm, Upson B7)
- PS0 is out; PS1 will come out next Thursday

Today:

- Brief discussion on aspects of learning a PL
- Evaluation and type checking of OCaml

Five aspects of learning a PL

1. **Syntax**: How do you write language constructs?
2. **Semantics**: What do programs mean? (Type checking, evaluation rules)
3. **Idioms**: What are typical patterns for using language features to express your computation?
4. **Libraries**: What facilities does the language (or a well-known project) provide "standard"? (E.g., file access, data structures)
5. **Tools**: What do language implementations provide to make your job easier? (E.g., top-level, debugger, GUI editor, …)

   – All are essential for good programmers to understand
   – Breaking a new PL down into these pieces makes it easier to learn

Our Focus

3110 focuses on **semantics** and **idioms**

- **Libraries** and **tools** are crucial, but throughout your career you'll learn new ones on the job every year

- **Semantics** is like a meta-tool: it will help you learn languages
- **Idioms** will make you a better programmer in those languages

- **Syntax** is almost always boring
  - A fact to learn, like "Cornell was founded in 1865"
  - People obsess over subjective preferences {yawn}
  - Class rule: We don't complain about syntax

Review of syntax

Syntactic class	Meta-variable	Examples
identifiers	*x*, *f*	**`a,x,y,x_y,foo1000`**
qualified identifiers		**`String.length,Char.uppercase`** (first part is *module* name)

Review of syntax

Syntactic class	Meta-variable	Examples
identifiers	*x*, *f*	**`a,x,y,x_y,foo1000`**
qualified identifiers		**`String.length,Char.uppercase`** (first part is *module* name)
constants	*c*	**`-2,-1,0,1,2`** (integers) **`1.0,-0.001,3.141`** (floats) **`true,false`** (booleans) **`"hello","!"`** (strings) **`'A',' ','\n'`** (characters)

Review of syntax

Syntactic class	Meta-variable	Examples
identifiers	*x*, *f*	**`a,x,y,x_y,foo1000`**
qualified identifiers		**`String.length,Char.uppercase`** (first part is *module* name)
constants	*c*	**`-2,-1,0,1,2`** (integers) **`1.0,-0.001,3.141`** (floats) **`true,false`** (booleans) **`"hello","!"`** (strings) **`'A',' ','\n'`** (characters)
unary operator	*u*	**`-,not`**

Review of syntax

Syntactic class	Meta-variable	Examples
identifiers	*x*, *f*	**a,x,y,x_y,foo1000**
qualified identifiers		**String.length,Char.uppercase** (first part is *module* name)
constants	*c*	**-2,-1,0,1,2** (integers) **1.0,-0.001,3.141** (floats) **true,false** (booleans) **“hello”,“!”** (strings) **‘A’,‘’,‘\n’** (characters)
unary operator	*u*	**-,not**
binary operators	*b*	**+,+.,*,-,>,<,>=,<=,^,!=**

Review of syntax

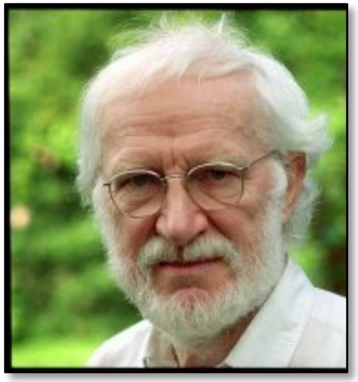
Expressions (aka *terms*):

- primary unit of OCaml programs
- akin to *statements* or *commands* in imperative languages
- described here in *Backus-Naur Form (BNF):*

```
e ::=  x | c | u e | e1 b e2
     | if e then e else e
     | let d1 and … and dn in e
     | e (e1,…,en)
d ::=  x = e
     | f ((x1:t),…,(xn:t)) : t = e
```

Backus and Naur

John Backus (1924-2007)
ACM Turing Award Winner 1977
"For profound, influential, and lasting contributions to the design of practical high-level programming systems"



Peter Naur (b. 1928)
ACM Turing Award Winner 2005
"For fundamental contributions to programming language design"

Review of syntax

Types:

```
t ::= int | float | bool
    | string | char
    | t1 * … * tn -> t
    | t1 -> t2 -> t      (built-in binary operators)
```

Type annotations are

- mostly optional from OCaml's perspective; can be *inferred*
- hugely helpful from programmer's perspective in reading and debugging code

Expressions

- Can get arbitrarily large since any subexpression can contain subsubexpressions, etc.

- Every kind of expression has:
  - **Syntax**
  - **Semantics:**
    - **Type-checking rules:** produce a type or fail with an error message
    - **Evaluation rules:** produce a *value*
      - (or exception or infinite loop)
      - Used only on expressions that type-check

Values

- All values are expressions
- Not all expressions are values
- A value is an expression that does not need any further evaluation
- Examples:
  - **`34`**, **`17`**, **`42`** are values of type **`int`**
  - **`true`**, **`false`** are values type **`bool`**

Question 1

What is `42`?

A. A value
B. An expression
C. Both a value and an expression
D. Neither a value nor an expression
E. (I'm lost)

Question 1

What is `42`?

A. A value

B. An expression

C. Both a value and an expression

D. Neither a value nor an expression

E. (I'm lost)

Question 2

What is `int`?

A. A value
B. An expression
C. Both a value and an expression
D. Neither a value nor an expression
E. (I'm lost)

Question 2

What is `int`?

A. A value

B. An expression

C. Both a value and an expression

D. Neither a value nor an expression

E. (I'm lost)

Question 3

What is `"cs"^"3110"`?

A. A value
B. An expression
C. Both a value and an expression
D. Neither a value nor an expression
E. (I'm lost)

Question 3

What is `"cs"^"3110"`?

A. A value

B. An expression

C. Both a value and an expression

D. Neither a value nor an expression

E. (I'm lost)

Addition expressions

- **Syntax:**
  `e1 + e2`

- **Type-checking:**
  If `e1` and `e2` have type `int`,
  then `e1` + `e2` has type `int`

- **Evaluation:**
  If `e1` evaluates to `v1` and `e2` evaluates to `v2`,
  then `e1 + e2` evaluates to sum of `v1` and `v2`

Other expressions

Less-than expressions

- **Syntax:** `e1 < e2`
- **Type-checking:** if `e1` has type `int` and `e2` has type `int` then `e1<e2` has type `bool`
- **Evaluation:** if `e1` evaluates to `v1`, and `e2` to `v2`, then `e1<e2` evaluates to `true` if `v1` is a smaller integer than `v2`, otherwise `e1<e2` evaluates to `false`

Other expressions

Conditional expressions

- **Syntax:** `if e1 then e2 else e3`
- **Type-checking:** if `e1` has type `bool` and, for some type `t`, both `e2` and `e3` have type `t`, then `if e1 then e2 else e3` has type t
- **Evaluation:**
  - if `e1` evaluates to `true`, then `if e1 then e2 else e3` evaluates to whatever `e2` evaluates to.
  - If `e1` evaluates to `false`, then `if e1 then e2 else e3` evaluates to whatever `e3` evaluates to.

Some shorthand notation

- Instead of "has type", we'll write a colon
  - That's what OCaml does anyway
  - "if `e1 : int` and `e2 : int` then `e1<e2:bool`"
- Instead of "evaluates to", we'll write long right arrow
  - No notion of this in OCaml syntax
  - "if `e1-->v1`, and `e2-->v2`, then `e1<e2-->true` if `v1` is a smaller integer than `v2`, otherwise `e1<e2--> false`"

Evaluation

Execution of an OCaml program is evaluation:

- Each step of execution involves *rewriting* (aka *reducing*) an expression into a simpler expression
- Until reaches a value
- That value is the result of the execution

E.g.

- `(1+2)*3 --> 3*3 --> 9`
- **`if true then`** ***`e1`*** **`else`** ***`e2`*** **`-->`** ***`e1`*** **`--> ?`**
- **`if false then`** ***`e1`*** **`else`** ***`e2`*** **`-->`** ***`e2`*** **`--> ?`**

Let expressions

- **Simplified syntax:**

  `let x = e1 in e2`

- **Type-checking:**

  If `e1 : t1`, and if `e2 : t2` under the assumption that `x : t1`, then `let x = e1 in e2 : t2`

- **Evaluation: ???**

Let expressions

```
let x = 1+4 in x*3
    --> let x = 5 in x*3
    --> 5*3
    --> 15
```

Let expressions

- **Simplified syntax:**

  `let x = e1 in e2`

- **Type-checking:**

  If `e1 : t1`, and if `e2 : t2` under the assumption that `x : t1`, then `let x = e1 in e2 : t2`

- **Evaluation:**
  - Evaluate `e1-->v1`
  - Substitute `v1` for `x` in `e2` (tricky!). Name that expression `e2'`.
  - Evaluate `e2'` to `v`
  - Result of evaluation is `v`

Let expressions

Multiple variable bindings of the same name is usually bad **idiom** (and darn confusing)

```
let x = 5
in ((let x = 6 in x) + x)
```

- By the end of week 3, we'll be able to explain exactly how this evaluates
- Temptation to think of rebinding as "assignment in Java." It's not the same. Avoid that trap!

Let expressions in REPL

Syntax:

```
let x = e
```

– Implicitly, “**`in`** *rest of what you type*”

E.g., you type:

```
let a="zar"
let b="doz"
let c=a^b
```

OCaml understands as

```
let a="zar" in
  let b="doz" in
    let c=a^b in…
```

Registration

- The course is full. Yay!
- **I can't add anybody now**. Boo.
- If you (still) want in:
  - Keep attending and doing problem sets
  - Don't stop trying to add the course
  - Email Course Administrator with your full name and NetID
  - You will be placed in "Waiting Set". NO PROMISES.

Upcoming events

- **PS 0 is out now**
- No recitations on Monday or Tuesday next week
- Office hours and consulting start next week; times and places TBA

Syntax is boring. This isn't.

THIS IS 3110