CSci 4223
Principles of Programming Languages

Lecture 2
Prof. Clarkson
Spring 2013

Review
• Building up SML one construct at a time via precise definitions
  – Constructs have syntax, type-checking rules, evaluation rules
  – Syntax is how to write it construct
  – Type checking is how compiler determines whether program is legal
  – Evaluation is how run-time converts an expression to a value

• So far:
  – Variable bindings
  – Several expression forms: addition, conditionals, ...
  – Several types: int, bool, unit

• Today:
  – Brief discussion on aspects of learning a PL
  – Functions, pairs, and lists (almost enough for all of HW1)

Five different things
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., REPL, debugger, GUI editor, ...)

These are 5 separate issues
− In practice, all are essential for good programmers
− Many people confuse them, but shouldn’t

Our Focus
This course 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 “GW was founded in 1821”
  − People obsess over subjective preferences (yawn)

Review
• Less-than expressions
  − Syntax: a1 < a2
  − Type-checking: if a1 has type int and a2 has type int then a1 < a2 has type bool
  − Evaluation: if a1 evaluates to v1, and a2 to v2, then a1 < a2 evaluates to true if v1 is a smaller integer than v2, otherwise a1 < a2 evaluates to false

• 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 e2 evaluates to false, then if e1 then e2 else e3 evaluates to whatever e3 evaluates to.
Multiple bindings of same variable

Multiple variable bindings of the same variable is usually bad
idiom
(And darn confusing)

But let’s look at it anyway
– Helps explain how the environment "works"
– Helps explain how a variable binding "works"

(Emphasize this now to lay the foundation for first-class functions)

Two reasons (either one sufficient)

1. Expressions in variable bindings are evaluated "eagerly"
   – Before the variable binding "finishes"
   – Afterwards, the expression producing the value is irrelevant
2. There is no way to "assign to" a variable in ML
   – Can only shadow it in a later environment

Function definitions

Functions: the most important building block in the whole course
– Like Java methods, have arguments and result
– But no classes, this, return, etc.

Example function binding:

```plaintext
val a = 1
val b = a (* b is bound to 1 *)
val a = 2
```

Note: The body includes a (recursive) function call: `pow(x, y - 1)`

Function bindings: 3 questions

- Syntax: `fun x0 (x1 : t1, .. , xn : tn) = e`
  (Will generalize in later lecture)
- Evaluation: A function is a value! (No evaluation to do, yet)
  – Adds `x0` to environment so later expressions can call it
  – (Function-call semantics will also allow recursion)
- Type-checking:
  – Adds binding `x0 : (t1 * .. * tn) -> t` if:
    – Can type-check body `e` to have type `t` in the static environment containing:
      • Enclosing static environment (earlier bindings)
      • `x1 : t1, .. , xn : tn` (arguments with their types)
      • `x0 : (t1 * .. * tn) -> t` (for recursion)

More on type-checking

- New kind of type: `(t1 * .. * tn) -> t`
  – Result type on right
  – The overall type-checking result is to give `x0` this type in rest of program (unlike Java, not for earlier bindings)
  – Arguments can be used only in `e` (unsurprising)
- Because evaluation of a call to `x0` will return result of evaluating `e`, the return type of `x0` is the type of `e`
- The type-checker "magically" figures out `t` if such a `t` exists
  – Later lecture: Requires some cleverness due to recursion
  – More magic after HW1: Later can omit argument types too
Function Calls

A new kind of expression: 3 questions

Syntax:
- (Will generalize later)
- Parentheses optional if there is exactly one argument

Type-checking:
If:
- \( e_0 \) has some type \((t_1 \times \ldots \times t_n) \rightarrow t\)
- \( e_1 \) has type \( t_1 \)
- \( e_n \) has type \( t_n \)
Then:
- \( e_0(e_1, \ldots, e_n) \) has type \( t \)

Example: \( \text{pow}(x, y-1) \) in previous example has type \( \text{int} \)

Function-calls continued

Evaluation:
1. (Under current dynamic environment,) evaluate \( e_0 \) to a function \( \text{fun} \ x_0 \ (x_1: t_1, \ldots, x_n: t_n) = e \)
   - Since call type-checked, result is guaranteed to be a function
2. (Under current dynamic environment,) evaluate arguments to values \( v_1, \ldots, v_n \)
3. Result is evaluation of \( e \) in an environment extended to map \( x_1 \) to \( v_1 \), \( \ldots \), \( x_n \) to \( v_n \)
   - (“An environment” is actually the environment where the function was defined, and includes \( x_0 \) for recursion. We’ll give a careful definition later in the course.)

Example, extended

```ml
fun pow (x : int, y : int) = 
  if y=0 
  then 1 
  else x * pow(x, y-1)
fun cube (x : int) = 
  pow (x, 3)
val sixtyfour = cube 4
val fortytwo = pow(2, 4) + pow(4, 2) + cube(2) + 2
```

Some gotchas

- Bad error messages if you mess up function-argument syntax
- \( \times \) use of \( \times \) in type syntax is not multiplication
  - Example: \( \text{int} \times \text{int} \rightarrow \text{int} \)
  - In expressions, \( \times \) is multiplication \( x \times \text{pow}(x, y-1) \)

Tuples and lists

So far: integers, booleans, conditionals, variables, functions
- Now ways to build up data with multiple parts
- This is essential (Java examples: classes with fields, arrays)

Rest of lecture:
- Tuples: fixed “number of pieces” that may have different types
  - Lists: any “number of pieces” that all have the same type

Later: Other more general ways to create compound data

Recursion

- If you’re not yet comfortable with recursion, you will be soon 😊
  - Will use for most functions taking or returning lists
  - Makes sense: calls to same function solve simpler problems
- In functional languages, recursion is the preferred idiom — not loops
  - We won’t use a single loop in ML
  - Loops often (not always) obscure simple, elegant solutions
**Pairs (2-tuples)**

We need a way to build pairs and a way to access the pieces

**Build**

- **Syntax**
  
  \[(e_1, e_2)\]

- **Evaluation**
  
  Evaluate \(e_1\) to \(v_1\) and \(e_2\) to \(v_2\); result is \((v_1, v_2)\)
  
  - A pair of values is a value

- **Type-checking**
  
  If \(e_1\) has type \(t_1\) and \(e_2\) has type \(t_2\), then the pair expression has type \(t_1 \times t_2\)
  
  - A new kind of type, the pair type

**Access**

- **Syntax**
  
  - and

- **Evaluation**
  
  Evaluate \(e\) to a pair of values and return first or second piece
  
  - Example: If \(e\) is a variable \(x\), then look up \(x\) in environment

- **Type-checking**
  
  If \(e\) has type \(t_1 \times t_2\), then \(\#_1 e\) has type \(t_1\) and \(\#_2 e\) has type \(t_2\)

**Examples**

Functions can take and return pairs

```haskell
fun swap (pr : int*bool) = ($2 pr, $1 pr)

fun sum_two_pairs (pr1 : int*int, pr2 : int*int) = ($1 pr1) + ($2 pr1) + ($1 pr2) + ($2 pr2)

fun div_mod (x : int, y : int) = (x div y, x mod y)
```

**Tuples**

Actually, you can have tuples with more than two parts

- A new feature: a generalization of pairs

\[(e_1, e_2, \ldots, e_n)\]

- \(t_1 \times t_2 \times \ldots \times t_n\)

- \(\#_1 e, \#_2 e, \ldots\)

Homework 1 uses a lot of triples of type \(\text{int}\times\text{int}\times\text{int}\)

**Nesting**

Pairs and tuples can be nested however you want

- Not a new feature: implied by the syntax and semantics

```haskell
val x1 = (7, (true, 9)) (* int * (bool*int) *)
val x2 = #1 (#2 x1) (* bool *)
val x3 = (#2 x1) (* bool*int *)
val x4 = ((3, 5), ((4, 8), (0, 0))) (* (int*int)*((int*int)*(int*int)) *)
```