PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

CSE341: Programming Languages

Lecture 1
ML Variable Bindings
Semantics
Expressions

Brett Wortzman
Spring 2020

A strange environment

- Next 4-5 weeks will use
  - ML language
  - Emacs editor
  - Read-eval-print-loop (REPL) for evaluating programs
- Need to get things installed and configured
  - Either in the department labs or your own machine
  - We've written thorough instructions (questions welcome)
- Only then can you focus on the content of Homework 1
- Working in strange environments is a CSE life skill

Mindset

- "Let go" of all programming languages you already know
- For now, treat ML as a "totally new thing"
  - Time later to compare/contrast to what you know
  - Lots of subtle, non-obvious differences that pop up at unexpected times
  - You *might* be able to get away with "oh that seems kind of like this thing in [Java]" for a while
  - But this will eventually confuse you, slow you down, and cause you to learn less
- Start with a blank file…

Syntax and semantics

- Syntax is how you write something
- Semantics is what that something means
  - Type-checking (before program runs)
  - Evaluation (as program runs)
- We will define all ML constructs in terms of these properties
- Side note: I claim *semantics* are what primarily define a PL and its pros/cons
  - But lots of programmers focus on *syntax*

A variable binding

```
val z = (x + y) + (y + 2); (* comment *)
```

More generally:

```
val x = e;
```

- *Syntax*:
  - *Keyword* `val` and *punctuation* `=` and `;`
  - *Variable* `x`
  - *Expression* `e`
    - Many forms of these, most containing *subexpressions*

A variable binding

```
val z = (x + y) + (y + 2); (* comment *)
```

More generally:

```
val x = e;
```

- *Semantics*:
  - *Type-checking: if expression type checks, extend static environment*
  - *Evaluation: evaluate* `e` *and extend dynamic environment*

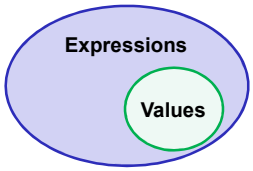
ML, carefully, so far

- A program is a sequence of *bindings*
- *Type-check* each binding in order using the *static environment* produced by the previous bindings
- *Evaluate* each binding in order using the *dynamic environment* produced by the previous bindings
  - Dynamic environment holds *values*, the results of evaluating expressions
- So far, the only kind of binding is a *variable binding*
  - More soon

Expressions

- We have seen many kinds of expressions:
  `34` `true` `false` `x` *`e1+e2`* *`e1<e2`*
  `if` *`e1`* `then` *`e2`* `else` *`e3`*
- Can get arbitrarily large since any subexpression can contain subsubexpressions, etc.
- Every kind of expression has
  1. Syntax
  2. Type-checking rules
     - Produces a type or fails (with a bad error message ☹)
     - Types so far: `int` `bool` `unit`
  3. Evaluation rules (used only on things that type-check)
     - Produces a value (or exception or infinite-loop)

Values



- All values are expressions
- Not all expressions are values
- A value "evaluates to itself" in "zero steps"
- Examples:
  - `34`, `17`, `42` have type `int`
  - `true`, `false` have type `bool`
  - `()` has type `unit`

Variables

- Syntax:
  sequence of letters, digits, _, not starting with digit
- Type-checking:
  Look up type in current static environment
  – If not there fail
- Evaluation:
  Look up value in current dynamic environment

Addition

- Syntax:
  `e1` `+` *`e2`* where *`e1`* and *`e2`* are expressions
- 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`

Slightly tougher ones

What are the syntax, typing rules, and evaluation rules for less-than expressions?

What are the syntax, typing rules, and evaluation rules for conditional expressions?

The foundation we need

We have many more types, expression forms, and binding forms to learn before we can write "anything interesting"

Syntax, typing rules, evaluation rules will guide us the whole way!

For Homework 1: functions, pairs, conditionals, lists, options, and local bindings

- Earlier problems require less

Will not add (or need):

- Mutation (a.k.a. assignment): use new bindings instead
- Statements: everything is an expression
- Loops: use recursion instead