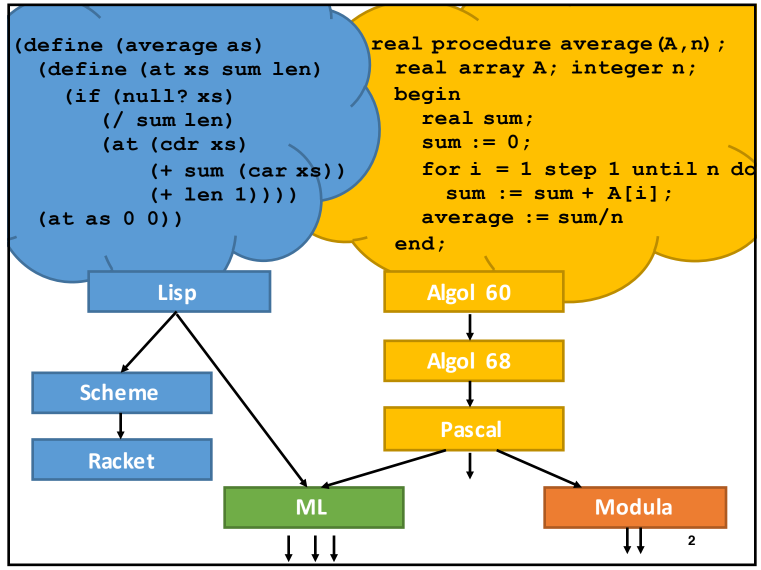
The ML Language

(We will use Standard ML.)

Warning to concurrent CS235 students:

Ocaml and SML are very similar semantically and syntactically, but there are just enough differences to make things annoying. Watch out!

1



ML: Meta-Language for Theorem-Proving

- Dana Scott, 1969
  - **Logic of Computable Functions (LCF):** for stating theorems about programs
- Robin Milner, 1972
  - **Logic for Computable Functions (LCF):** automated theorem proving for LCF
- Theorem proving is a hard search problem.
  - Needs its own language...
  - **ML: M**eta-**L**anguage for writing programs (tactics) to find proofs of theorems (about other programs)
- ***Proof Tactic:*** *Partial* function from formula to proof.
  - Guides proof search
  - Behavior is one of:
    - find and return proof
    - never terminate
    - report an error

3

Language Support for Tactics

- Static type system
  - guarantee correctness of generated proof
- Exception handling
  - deal with tactics that fail (Turing Award)
  - make failure explicit, force programmer to deal with it
- First-class/higher-order functions
  - compose tactics
  - `fun compose(tactic1, tactic2) =`
    `fn formula => tactic2 (tactic1 (formula))`

4

The ML language:
statically-typed, expression-oriented

Several important ideas beyond what we studied in Racket
- Static typing
- Type inference
- Algebraic data types
- Pattern matching
- Exceptions
- Modules

We will also consider...
- Limited mutation
- Lazy evaluation
- Implementation issues for
  - exceptions
  - closures and lexical scope
- ... And other things along the way...

Slides mix material from Ben, Steve Freund, Dan Grossman

Wipe your syntax slate clean.

Much (but not all!) of ML's semantics
will seem familiar from Racket.

An ML program is a sequence of bindings.

```
(* My first ML program *)

val x = 34;

val y = 17;

val z = (x + y) + (y + 2);

val q = z + 1;

val abs_of_z = if z < 0 then 0 – z else z;

val abs_of_z_simpler = abs z

(* comment: ML has (* nested comments! *) *)
```


Variable binding

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

More generally: `val x = e;` — Semicolon optional; may improve debugging.

3 Questions:

Syntax:
- *Keyword* `val` *and punctuation* `=`
- *Variable* `x`
- *Expression* `e`

Type-checking:
- *Type-check* `e : t` *in the current static environment, for some type* `t`.
- *Extend the current static environment with the typing* `x : t`

Evaluation *(only for things that type-check):*
- *Evaluate* `e` *to a value* `v` *using the current dynamic environment.*
- *Extend the current dynamic environment with the binding* `x` → `e`.

Bindings, types, and environments

- A program is a sequence of *bindings.*
- Bindings build ***two*** environments:
  - ***static*** *environment* maps variable to type ***before evaluation***
  - ***dynamic*** *environment* maps variable to value ***during evaluation***
- ***Type-check*** each binding in order:
  - using ***static*** *environment* produced by previous bindings
  - and extending it with a binding from variable to type
- ***Evaluate*** each binding in order:
  - using ***dynamic*** *environment* produced by previous bindings
  - and extending it with a binding from variable to value

Expressions and types

- `e : t` means "expression `e` has type `t`"
- Variables:
  - Syntax: sequence of letters, digits, _, not starting with digit
  - **Type-check:** Lookup in current static environment, fail if not found.
  - Evaluation: Look up value in current dynamic environment
- Addition
  - Syntax: ***e1* + *e2*** where ***e1*** and ***e2*** are expressions
  - **Type-check:**
    - If ***e1* : int** and ***e2* : int**, then ***e1* + *e2* : int**
  - Evaluation:
    - If ***e1*** evaluates to **v1** and ***e2*** evaluates to **v2**, then ***e1* + *e2*** evaluates to sum of **v1** and **v2**

Type-checking expressions

`34 : int` `~1 : int (* negative one *)`

`3.14159 : real` `true : bool` `false : bool`

`x : t`
- if `t` = *lookup* `x`'s *type in current static environment*

***e1* + *e2* : int**
- if **e1 : int** and **e2 : int** in current static environment

***e1* < *e2* : bool**
- if **e1 : int** and **e2 : int** in current static environment

if *e1* then *e2* else *e3* : t
- if **e1 : bool** and **e2 : t** and **e3 : t** in current static environment
- (**e2** and **e3** must have the same type)

e1* = *e2* : *bool ***e1* <> *e2* : bool (* not equal *)**
- if **e1 : t** and **e2 : t** in current static environment
- (**e2** and **e3** must have the same type, one more restriction later)

Function binding examples

```
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 (2,2+2) + pow (4,2) + cube (2) + 2
```

Watch out

Odd error messages for function-argument syntax errors

`*` in type syntax is not arithmetic

- Example: `int * int -> int`
- In expressions, `*` is multiplication: `x * pow(x,y-1)`

Cannot refer to later function bindings

- Helper functions must come before their uses
- Special construct for *mutual recursion* (later)

13

Function bindings

- Syntax: `fun x0 (x1 : t1, … , xn : tn) = e`
  - `x0 ... xn` are variable names
  - `t1 ... tn` are types
  - `e` is an expression
  - (Will generalize later)
- **Type-check:**
  - Adds binding `x0 : (t1 * … * tn) -> t` to current static environment if:
  - Can type-check body `e` to have type `t` in the current static environment, extended with:
    - `x1 : t1, …, xn : tn` (arguments with their types)
    - `x0 : (t1 * … * tn) -> t` (for recursion)
- Evaluation:
  - Produce a function closure `c` capturing the function code and the current dynamic environment extended with `x0` → `c`
  - Extend the current dynamic environment with `x0` → `c`

14

Function types

`fun x0 (x1 : t1, … , xn : tn) = e`

- Function types: `(t1 * … * tn) -> t`
  - Result type on right
  - Overall type-checking result: *give* `x0` *this type in rest of program*
- Calling `x0` returns result of evaluating `e`, thus return type of `x0` is type of `e`.
- Type-checker infers `t` if such a `t` exists. Later:
  - Requires some cleverness due to recursion
  - Can omit argument types too

15

Function call

A new kind of expression: 3 questions

Syntax: `e0 (e1,…,en)`

- `e0 ... en` are expressions
- (Will generalize later. Parentheses optional if exactly one argument.)

Type-check:

- If:
  - `e0` has some type `(t1 * … * tn) -> t`
  - `e1` has type `t1`, …, `en` has type `tn`
- Then:
  - `e0(e1,…,en)` has type `t`

  Example: `pow(x,y-1)` in previous example has type `int`

16

Function-call evaluation

Evaluation: `e0 (e1, …, en)`

1. Under current dynamic environment, evaluate **e0** to a function closure
   - Since call type-checked, result *will be* a function taking parameters ***x1,…,xn*** of types matching those of ***e1,…,en***
2. Under current dynamic environment, evaluate arguments to values **v1, …, vn**
3. Result is evaluation of ***e*** in an environment extended to map **x1** to **v1**, …, **xn** to **vn**

17

Let expressions

- Syntax: `let b1 b2 … bn in e end`
  - Each ***bi*** is any *binding* and ***e*** is any *expression*
- **Type-check:**
  - Type-check each ***bi*** and ***e*** in a static environment that includes the previous bindings.
  - Type of whole let-expression is the type of ***e***.
- Evaluation:
  - Evaluate each ***bi*** and ***e*** in a dynamic environment that includes the previous bindings.
  - Result of whole let-expression is result of evaluating ***e***.

Like Racket's let*

18

Anonymous functions

3 questions:

- Syntax: `fn (x1 : t1, ..., xn : tn) => e`
- **Type-check:**
  - Type-check **e** in the current static environment, extended with ***x1 : t1, ..., xn : tn***.
  - If e has type t, the function has type **(t1 * … * tn) -> t**
- Evaluation: A function (closure) is a value.
  Recall the function's body is not evaluated until a function call.
- Difference with **fun**: no recursion.

19