# Lecture 03: Programming Languages [CMPU-235]
# September 7, 2017

**Sameer S Pradhan**
Vassar College

# Functional Programming

# Pure Functional Programming

- **The result of a function application is independent of the context in which it occurs (referential transparency).**
- **Variables are bound to values only through the linking of actual parameters to formal parameters in function calls (no side effects).**
- **Control flow is governed by function calls and conditional expressions (and recursion).**
- **Storage management is implicit (and requires garbage collection).**
- **Functions are first class citizens!**

# Pure Functional Programming

**"Look, Ma, No Hands!"**

- **No assignment statements!**
- **No iteration!**

**How is it possible to write programs in a language like this?**

**Some Simple Examples:**

- **append**
- **length**
- **flatten**
- **atomcount**

# What is SML?

- Mostly-pure safe strongly-typed functional programming language
- Suitable both for programming in the small and programming in the large

# SML in the Small

Core-SML: a program is a series of declarations and an expression to be evaluated

- The declarations provide definitions of types, and functions — "the environment"
- The expression is evaluated to get the answer
- Comments are written `(* like this *)`

# Expressions

- Every expression in SML has a type.
  - But never have to write it down. SML will *infer* the type for you.
  - If you're get type-errors, sometimes helps to write out types of expressions.
- The most basic expressions are values
- All other expressions are evaluated down to values
  - Some expressions have side-effects (*e.g.*, print), but I won't talk about those today.

# Values: Basic Types

| Type | Values | Comments |
|---|---|---|
| unit | () | A trivial type with a single value |
| bool | `true`,`false` | |
| int | 0, 1, 42, $\sim 1$ | integers |
| real | 0.0, 1.2, $\sim 2.45$ | won't be used much in this course |
| char | `#"a"`, `#newline` | characters |
| string | "foo", "", "bar baz" | *not* the same as a list of characters |

# Expression: Basic Types

| Expression | Types | Result type | Comments |
|---|---|---|---|
| `if` $b$ `then` $e_1$ `else` $e_2$ | $b$ of type `bool`<br>$e_1, e_2$ of the same type | <br>same as $e_i$ | |
| $e_1$ `andalso` $e_2$<br>$e_1$ `orelse` $e_2$ | $e_1, e_2$ : `bool` | `bool` | <br>"&&", "\|\|" |
| $e_1 + e_2$, $e_1 - e_2$, $e_1 * e_2$ | $e_1, e_2$ either both `int`<br>or both `real` | <br>same as $e_i$ | overloaded<br>operators |
| $e_1 < e_2$ | $e_1, e_2$ either both `int`<br>or both `real` | <br>`bool` | <br>also overloaded |
| $e_1 = e_2$ | $e_1, e_2$ : same equality type | `bool` | |
| $e_1 / e_2$ | $e_1, e_2$ : `real` | `real` | real division |
| $e_1$ `div` $e_2$ | $e_1, e_2$ : `int` | `int` | integer division |
| $e_1$ ^ $e_2$ | $e_1, e_2$ : `string` | string | concatenation |

For now, "equality types" are just `unit`, `int`, `char`, `string`, but not `real`.

- A wart in SML. As we'll see, you can avoid = most of the time

# More Types: Tuples, Lists, Option

| Type | Values |
|---|---|
| int * bool | (3,true) |
| int * string * string | (42, "foo", "J.Random Hacker") |
| int list | nil, 2::3::nil |
| (int * bool) list | [], [(3,true), (~ 2,false)] |
| (string * bool) option | NONE, SOME ("foo", 23) |
| int option list | [NONE, NONE, SOME 1, SOME 3, NONE] |

- Tuples are of fixed but arbitrary arity
- Lists are homogeneous
- List and option are polymorphic: more on that later

# Defining your own Type

Two mechanisms:

- Type abbreviations:
  - `type age = int`
  - `type person = string * age`
    - `("J. Random Hacker", 12)`
- New datatypes:
  - 
```
datatype employee =
  Grunt of person
| Manager of person * employee list
```
    - `Grunt ("J. Random Hacker", 12)`
    - 
```
Manager (("PHB", 51),
         [Grunt ("J.Random Hacker", 12)])
```
  - The datatype declares several *constructor* names that must be unique to the datatype
  - May be recursive (*e.g.*, `employee` above)

# Datatypes: Two built in ones

Turns out that `list` and `option` are standard datatypes

- `datatype 'a option = NONE | SOME of 'a`
  - `'a` is a *type variable*: stands for any type
    - `SOME 42 : int option`
    - `SOME ("J. Random Hacker, 12) : person option`
    - `NONE : 'a option`
- `datatype 'a list = nil`
  `| :: of 'a * 'a list`

  plus a *fixity* declaration to make `::` be right-associative

# Analyzing values: Patterns

In SML, analysis of values is done using *pattern matching*:

- Informally speaking, a pattern describes the structure of a value, and binds some variables to the subcomponents of the value.

| Pattern | Matches | Comments |
|---|---|---|
| _ | anything | wildcard |
| `x` | anything | binds `x` to the value |
| 42 | the integer 42 | |
| `false` | the boolean $false$ | |
| $(pat_1, pat_2)$ | a pair $(v_1, v_2)$ if $pat_i$ matches $v_i$ | |
| $(x, \_)$ | matches `(false, 42)`, binds $x$ to `false` | |
| $(pat_1, pat_2, pat_3)$ | a triple $(v_1, v_2, v_3)$ ... | |
| `NONE` | matches `NONE` of any option type | |
| `SOME` $pat$ | matches `SOME` $v$ if $pat$ matches $v$ | |
| $pat_1 :: pat_2$ | | |

# val Declarations

Patterns may be used (*e.g.*, at the SML prompt) to define some variables:

- `val x = 42`
- `val (x,y) = (42, false)`
- 
```
val Manager (phb,lackeys) =
  Manager (("PHB",51),
           [Grunt ("J.Random Hacker", 12)])
```
- `val piApprox = 3.14159`
- `val SOME x = NONE`
  - Compiler comes back with a warning “Non-exhaustive match”, then runs the code anyway and comes back with a runtime error “binding failure” or “match error”

# case Expressions

Analyze a value by cases: like a generalized if-expression.

```
case (42, false) of
  (x, true) => x
| (23, false) => 17
| _ => 0
```

- Tests each pattern in order, executes the branch that matches.
- Exhaustiveness-checks at compile-time: generates a warning.

# fun Declarations

In addition to `val` declarations, can also define functions:

```
fun employeeName (Manager ((name, _), _)) = name
  | employeeName (Grunt (name,_))         = name
```

The compiler:

- infers the type of the function `employee -> string`
- checks that we covered all the cases for `employee`s

# Recursion, Polymorphism

Functions may be recursive, and indeed, polymorphic:

- `fun length nil = 0`
- `  | length (_::l') = 1 + (length l')`
- Since we don't case about the elements of the list, this function has type `'a list -> int`