CSE 341: Programming Languages

Explore several **other** programming **paradigms**

ML, Scheme, ...:
functional programming, lists, recursion, pattern-matching, polymorphic typing, type inference

Smalltalk, Java, ...:
object-oriented programming

Prolog, ...:
logic programming

Also explore advanced programming environments
- interactive interpreters (e.g. ML, Scheme)
- graphical environments (Smalltalk)

Course outline

1 lecture: Concepts

4 weeks: ML
1 week: Scheme

2 weeks: Smalltalk (really, Squeak)
1 week: Java, GJ

1 week: Prolog, CLP(R)

1 lecture: Wrap-up

Homework exercises and a free-form project for functional and OO sections
- 2-person **teams** allowed

Midterm and final exam

Why study programming languages?

"But if **thought corrupts language**, **language can also corrupt thought**. A bad usage can spread by tradition and imitation even among people who should and do know better."
George Orwell, *Politics and the English Language,* 1946

"If you cannot be the master of your language, you must be its slave."
Richard Mitchell

"A different language is a different vision of life."
Federico Fellini

"The language we use ... determines the way in which we view and think about the world around us."
The Sapir-Whorf hypothesis

Course motivation and objectives

Hypothesis:
programming language shapes programming thought

Goal: learn several new, interesting, mind-expanding languages
- can conceive of and design better programs if exposed to alternative, more expressive programming languages & constructs
- can apply abstraction better with practice

Ability to select right language for task

Easier to learn new languages

Ability to design "little languages"

Appreciation for programming environment support facilities

Appreciation for rich libraries

Language design goals

Some end goals:
- rapid development
- ease of maintenance
- reliability, safety
- ease of learning
- portability
- efficiency

Some means to these goals:
- readability
- writability
- simplicity
- **orthogonality**
- **expressiveness**

Many goals in conflict
⇒ language design is an engineering & artistic activity
⇒ need to consider target audience's needs

Language design target audiences

Scientific, numerical computing
- Fortran, APL, ZPL

Symbolic computing
- Lisp, Scheme, ML, Prolog, Smalltalk

Systems programming
- C, C++, Modula-3

Applications programming
- C, C++, Modula-3, Java, Lisp, Scheme, ML, Smalltalk, ...

Scripting, macro languages
- csh, Perl, Tcl, Excel macros, ...

Specialized languages
- SQL, LaTeX, PostScript, Unix regular expressions, ...

Main programming language concepts

Separation of syntax, semantics, and pragmatics
- EBNF to specify syntax precisely
- semantics is more important than syntax
- pragmatics: programming style, intended uses, etc.

Control structures
- iteration, conditionals; exceptions
- procedures, functions; recursion
- first-class functions; message passing
- backtracking in logic languages
- parallelism

Data structures and types
- atomic types: numbers, chars, bools
- type constructors: records, tuples, lists, arrays, functions, ...
- user-defined abstract data types (ADTs); classes
- polymorphic/parameterized types, ADTs, classes

Static vs. dynamic typing; type inference
Lexical vs. dynamic scoping
Eager vs. lazy evaluation

Some good language design principles

Strive for a simple, regular model
- for evaluation
- for data reference
- for memory management

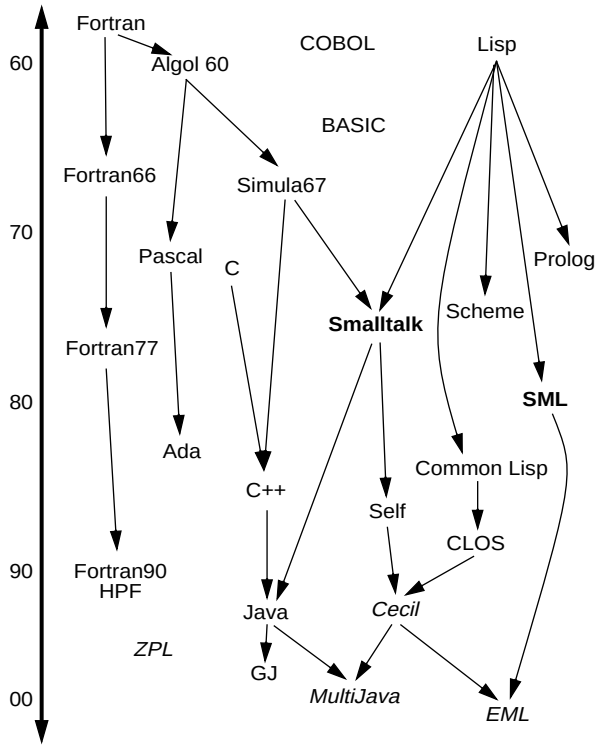
Be expression-oriented
Use heap-allocated data with automatic garbage collection

Include sophisticated abstraction mechanisms
- for control
- for data

Include polymorphic static type checking

Have a complete & precise language specification
- full run-time error checking for cases not detected statically

Partial history of programming languages



ML

Main features:

- expression-oriented
- list-oriented, garbage-collected heap-based
- functional
  - functions are first-class values
  - largely side-effect free
- strongly, statically typed
  - polymorphic type system
  - automatic type inference
- pattern matching
- exceptions
- modules
- **highly regular and expressive**

Designed as a **M**eta **L**anguage for automatic theorem proving system in mid 70's by Milner et al.

Standard ML: 1986

SML'97: 1997

Exemplifies Caml, Haskell, Miranda

Interpreter interface

Read-eval-print loop

- **read** input expression
  - reading ends with semi-colon
  - = prompt indicates continuing expression on next line
- **eval**uate expression
- **print** result
- repeat

```
- 3 + 4;
val it = 7 : int
- it + 5;
val it = 12 : int
- it + 5;
val it = 17 : int
```

`it` (re)bound to last evaluated value,
in case you want to use it again

REPLs: A Big Idea™

Basic ML data types and operations

`int`

- `~`, `+`, `-`, `*`, `div`, `mod`; `=`, `<>`, `<`, `>`, `<=`, `>=`; `real`, `chr`

`real`

- `~`, `+`, `-`, `*`, `/`; `<`, `>`, `<=`, `>=` (no equality); `floor`, `ceil`, `trunc`, `round`

`bool`: different from `int`

- `true`, `false`
- `=`, `<>`; `orelse`, `andalso`

`string`

- e.g. `"I said \"hi\"\tin dir C:\\stuff\\dir\n"`
- `=`, `<>`, `^`

`char`

- e.g. `#"a"`, `#"\n"`
- `=`, `<>`; `ord`, `str`

Variables and binding

Variables declared and initialized with a `val` binding:

```
- val x:int = 6;
val x = 6 : int
- val y:int = x * x;
val y = 36 : int
```

Variable bindings cannot be changed!

- unlike assignment in C
- like equality in math

Variables can be bound again,
but this **shadows** the previous definition

- e.g. `it`

Variable types can be omitted

- they will be **inferred** by ML based on the type of the r.h.s.

```
- val z = x * y + 5;
val z = 221 : int
```

Binding, not Assignment: A Big Idea

Type Inference: A Big Idea

Strong, static typing

ML is **statically typed**: it will check for type errors statically
(i.e., when programs are entered, not when they're run)

ML is **strongly typed**: it catches all type errors

- unlike C's cast, `union`, `void*`, ...

Type errors can look weird, given ML's fancy type system

E.g.:

```
- asd;
Error: unbound variable or constructor: asd
- 3 + 4.5;
Error: operator and operand don't agree
  operator domain: int * int
  operand:         int * real
  in expression:
    3 + 4.5
- 3 / 4;
Error: overloaded variable not defined at type
  symbol: /
  type: int
```

Strong, Static Typing: A Big Idea

Records

ML records are like C structs

- allow heterogeneous element types, but fixed # of elements

A record type:

```
{name:string, age:int}
```

- field order doesn't matter

A record value:

```
{name="Bob Smith", age=20}
```

Can construct record values from expressions for field values

```
{name = "Bob " ^ "Smith",
 age  = 18+num_years_in_college}
```

As with any other value, can bind record values to variables

```
- val bob = {name="Bob " ^ "Smith",
=            age=18+num_years_in_college};
val bob = {age=20,name="Bob Smith"}
        : {age:int,name:string}
```


More on records

Can extract record fields using `#`*`fieldname`* function

```
- val bob' = {name= #name(bob),
=             age= #age(bob)+1};
val bob' = {age=21,name="Bob Smith"} : {...}
```

Cannot assign to a record's fields

- an immutable data structure

Tuples

Like records, but fields ordered by position, not label
Useful for pairs, triples, etc.

A tuple type:

```
string * int
```

- order **does** matter

A tuple value:

```
("Joe Stevens", 45)
```

As with any other value, can bind tuple values to variables

```
- val joe = ("Joe "^"Stevens", 25+num_jobs*10);
val joe = ("Joe Stevens",45) : string * int
```

Can extract record fields using *#pos* function

```
- val joe' = (#1(joe), #2(joe)+1);
val joe' = ("Joe Stevens",46) : string * int
```

Cannot assign to a tuple's components

- another immutable data structure

Lists

ML lists are like C linked lists, but built-in

- require homogeneous element types, but variable # of elements

A list type:

```
int list
```

- in general: *T* `list`, for any type *T*

A list value:

```
[3, 4, 5]
```

Empty list: `[]` or `nil`

Basic operations on lists

Add to front of list: `::`

```
- val l1 = 3::(4::(5::nil));
val l1 = [3,4,5] : int list
- val l2 = 2::l1;
val l2 = [2,3,4,5] : int list
```

Look up the first ("head") element: `hd`

```
- hd(l1) + hd(l2);
val it = 5 : int
```

Extract the rest ("tail") of the list: `tl`

```
- val l3 = tl(l1);
val l3 = [4,5] : int list
- val l4 = tl(tl(l3));
val l4 = [] : int list
- tl(l4);  (* or hd(l4) *)
uncaught exception Empty
```

Cannot assign to a list's elements

- another immutable data structure

First-class values

All of ML's data values are **first-class**

- there are no restrictions on how they can be created, used, passed around, bound to names, stored in other data structures, ...

One consequence:
can nest records, tuples, lists arbitrarily

A legal value, and its type:

```
{foo=(3, 5.6, "seattle"),
 bar=[[3,4], [5,6,7,8], [], [1,2]]}
: {bar:int list list, foo:int*real*string}
```

All values are first-class: A Big Idea

Reference data model

A variable **refers to** a value (of whatever type), uniformly

A record, tuple, or list **refers to** its element values, uniformly

- all values are implicitly referred to by pointer

A variable expression evaluates to
a reference to the value that the variable was bound to

A variable binding makes the l.h.s. variable
refer to its r.h.s. value

No implicit copying upon binding, parameter passing, storing in a data structure

- like Java, Scheme, Smalltalk, ...: all high-level languages
- unlike C, where non-pointer values are copied
  - C arrays?

Reference-oriented values are heap-allocated (logically)

- optimized for scalar values like ints, reals, chars, bools, nil

Reference Data Model: A Big Idea