Today

- Introduction to Typing
- Introduction to ML

Typing: introduction

- A name for a set of values and some operations which can be performed on that set of values.

- Give some semantic meaning to what is ultimately just mere bits

Typing: categories

- Primitive types
  - Simplest kind of type e.g. integer and floating-point number

- Composite Types
  - Consisting of basic types
  - E.g. struct Person {
    int age;
    char *name;
  };

- Object types
  - A datatype that is used in object-oriented programming to wrap a non-object type to make it look like an object.
  - E.g. class Integer in Java

- Class types
  - Classes describe the rules by which objects behave, those objects described by a particular class are known as “instances” of said class

- Subtypes
  - If given type A is compatible with type B, then A is a subtype of B
  - Hence, one datatype can be more than one subtype
  - Polymorphism
    - E.g.
      public class Shape{
        public void draw(int x, int y){
          // do nothing
        }
      }
      public class Rectangle extends Shape{
        public void draw(int x, int y){
          // do nothing
        }
      }
      public class Circle extends Shape{
        public void draw(int x, int y){
          // do nothing
        }
      }
      Shape myShape;
      myShape = new Rectangle();
      myShape.draw();
      myShape = myCircle;
      myShape.draw();

- Interfaces/protocols
  - A definition of methods and values which the objects agree upon in order to cooperate.
  - A specification of those properties of a software component that other components may rely upon
    - E.g.
      public interface Shape{
        public abstract void draw(int x, int y);
      }
Typing: categories

- Function types
  - A type that allows an object to be invoked or called as if it were an ordinary function
  - E.g.

```c
// Declaration of C sorting function
void sort (int* itemlist, int numitems, int (*cmpfunc)(int*,int*));
...
// Callback function
int compare_function( item* A, item* B) {…… }
...
sort( itemlist, numitems, compare_function);
```

- Declaration of C++ sorting function.
  - void sort(int* itemlist, int numitems, functor_class&cmpfunctor);
  - C++ class type functor
    ```c
    class functor_class {
    int operator()( item* A, item* B ) { ... }
    };
    ```
    - functor_class X;
    - sort( itemlist, numitems, X );

Typing: why?

- Efficiency & Optimization
  - Compiler can generate better code if it knows exactly what is in each variable
  - Historically that’s where typing comes from
  - In Fortran
    - INTEGER I
    - REAL R
    - Subroutines have parameters of specified type
      - Compiler can generate efficient code for exactly the type of data present without any kind of checks.
    - Types match machine data types
      - For example: float(32bit-IEEE754), double(64bit-IEEE754)

- Error Checking & Prevention
  - In a freely typed system (e.g. assembler)
    - Can add a float to an integer
    - Can compare a string to a float
    - Can use a float as a pointer
    - Typing prevents errors of this kind
  - Stronger typing capabilities mean that more errors can be detected.

- Documentation & Readability
  - Using types in languages also improves documentation of code.

Type-checking: introduction

- The process of verifying and enforcing the constraints of types is called type checking.

- Type checking may either occur at compile-time (static check) or at run-time (dynamic check)
Type Checking: static vs. dynamic

- **Static type checking**
  - Part of semantic analysis carried by compiler
  - E.g. Code contains explicit declarations
  - E.g. A : Integer;
  - Compiler knows the type
  - Compiler checks the type at compile time

- **Dynamic type checking**
  - Type is not determined till run-time
  - Variables can be differently typed according to execution path
  - E.g.
    ```
    if( !bFlag ){
        String strClassFilePath;
        strClassFilePath= readfromConfigurationFile( );
        myShape= (Shape)class.forName( strClassFilePath);
    }
    myShape.draw( );
    ```

Type Checking: type inference

- **Compiler/interpreter infer type from code!**
  - E.g. A3 := B4 + 1;
    - Q: What type is A3 and B4 ?
      - A: Must be integer
    - E.g. if test then …
      - Q: What type is test ?
        - A: Must be Boolean
    - E.g.           A3 := Func (Test1);
      - Q: What is Func?
        - A: Must be array/function Boolean->Integer

A language with a sound type system is one in which all types can always be inferred in any valid program.

Example of language designed this way is Standard ML.

Note that we still have strong static typing and many of the benefits that come from that.

ML: main features

- **Strong, static typing**
  - Quite a fancy type system!
- **Parametric polymorphism**
  - Similar to OOP (in fact, it influenced OOP)
- **Pattern matching**
  - Function as a template
- **Exception handling**
  - Allow you to handle errors/exception
- **Elaborate module system**
  - Most highly developed of any language
- **Type inference**
  - Recursive data type

ML: types & expressions

- **Primitive types**
  - bool, int, real, string
- **Constructors**
  - list, tuple, array, record, function
- **Each ML expression has a type associated with it.**
  - Interpreter builds the type expression for each input
  - Cannot mix types in expressions 2+3.0 : real error!
  - Must explicitly coerce/convert e.g. real(2) + 3.0 : real

ML: how far have PL advanced?

- Writing a gcd implementation
  ```
  gcd(n,m) =
  if m = 0 then
    n
  else
    gcd(n mod m, m);
  ```

ML: introduction

- Developed at Edinburgh (early '80s) as Meta-Language for a program verification system
  - Now a general purpose language
  - There are two basic dialects of ML
    - Caml (including Objective Caml, or OCaml)
- A pure functional language
  - Based on typed lambda calculus
  - Grew out of frustration with Lisp!
  - Serious programs can be written without using variables
- Widely accepted
  - reasonable performance (claimed)
  - can be compiled
  - syntax not as arcane as LISP (nor as simple …)

ML: main features

- **Strong, static typing**
  - Quite a fancy type system!
- **Parametric polymorphism**
  - Similar to OOP (in fact, it influenced OOP)
- **Pattern matching**
  - Function as a template
- **Exception handling**
  - Allow you to handle errors/exception
- **Elaborate module system**
  - Most highly developed of any language
- **Type inference**
  - Recursive data type

ML: how far have PL advanced?

- Writing a gcd implementation
  ```
  gcd(n,m) =
  if m = 0 then
    n
  else
    gcd(n mod m, m);
  ```

ML: introduction

- Developed at Edinburgh (early '80s) as Meta-Language for a program verification system
  - Now a general purpose language
  - There are two basic dialects of ML
    - Caml (including Objective Caml, or OCaml)
- A pure functional language
  - Based on typed lambda calculus
  - Grew out of frustration with Lisp!
  - Serious programs can be written without using variables
- Widely accepted
  - reasonable performance (claimed)
  - can be compiled
  - syntax not as arcane as LISP (nor as simple …)
ML: Primitive Types

- **int**
  - e.g. x: int;
  - Negative sign uses ~
  - Operators: + - * div mod

- **real**
  - e.g. x: real;
  - - 3.45 or using e notation (3E7)
  - Operators: + - * /
  - Conversion functions: real(integer), floor(real), abs(x)

- **string**
  - e.g. s: string;
  - Delimited by double quotes
  - Caret ^ is concatenation e.g. "house" ^ "cat"
  - Function size returns length of string
  - Special characters: \n \t " \" \n \t " \"

- **bool**
  - e.g. b: bool;
  - true (and false)

ML: operators

- All operators are infix

- **Numeric operators**
  - The usual <, >, <=, >= and <> are available
  - For reals, = and <> are not available (a <= b andalso a >= b)
  - For strings, these can be used for lexicographic ordering

- **Logical operators**
  - Short-circuit evaluation
  - if ... then ... else is an expression, not a control structure...

- **Operator overloading**
  - Same symbol could be used for operations that are internally disimilar
  - *, +, -, <, <=, >, >= are all overloaded
  - When an overloaded operator is used, the leftmost argument is inspected first to decide on type

ML: constructor types - lists

- **Syntax**
  - [ obj1, obj2, ... ]
  - Objects in a list must be homogenous (same type)
  - E.g. [1,2,3] : int list — a list of integers
          ["dog","cat","moose"] : string list — a list of strings
          [1.0,2.0,3.0] : real list — a list of reals
  - The empty list is written [] or nil

- **Operations**
  - @ operator is used to concatenate two lists of the same type
  - :: operator returns a new list with the first argument append to the front
  - [1,2] :: [3,4,5,6] returns ??
  - hd returns the first element of a list
    - E.g. hd([1,2,3]) returns 1
  - tl returns the tail
    - E.g. tl([1,2,3]) returns [2,3]

ML: constructor types - tuples

- **Syntax**
  - ( obj1, obj2, ... )
  - Objects in a tuple can be heterogeneous (different types)
  - E.g. (2,"abc") : int* string
          (2,3.0,"abc") : int* real* string
          (2,3.0,4.0,"abc") : int* real* string* string
          (1,2,3) : int* string list
  - The empty tuple is written () and often called unit
  - Composite format of a tuple can be used on left-hand side of val
    - E.g. val (day, month, year) = (13,"March",1066);

- **Operations**
  - # operator to extract the ith field of a tuple
    - E.g. #2(6,7, " abc") returns 7
    - #3(6,7, " abc") returns abc
  - = and <> operators for equality/in-equality
    - E.g. (3,"a") = (abs x, "a", (3 > 2));

ML: constructor types - functions

- **Syntax**
  - fun <func-name> <input-param> = <expression>;
  - Keyword fun starts the function declaration
  - Function arguments don’t always need parentheses, doesn’t hurt to use them
  - All ML functions are unary, i.e. have one argument.
    - However, this argument may be a complex data type...

- **Examples**
  - fun fahrToCelsius f = (f - freezingFahr) * 5 div 9;
  - fun celsiusToFahr c = c * 9 div 5 + freezingFahr;
  - fun square x = x * x;
    - Function square takes an integer as input and returns an integer as output.
    - This is written as square: int -> int (indicates the function)

ML: assignment

- **Use val to assign value to variables**
  - val <constant-name> = <expression>;

- **Examples**
  - val seconds = 60;
  - val seconds = 60 : int
  - val minutes = 60;
  - val minutes = 60 : int
  - val tm = seconds * minutes;
    - val tm = 86400 : int
  - val shout = "aaa" ^ "rgh" ^ "!!!!";
    - val shout = "aaargh!!!" : string
ML: constructor types - functions

- Anonymous functions
  - `fn <func-param> => <func-body>`
  - E.g.
    ```
    (fn n => n*2)(9);
    ```
- The following declarations are identical:
  - `fun f(n) = 2*n;`
  - `val f = fn n => 2*n;`
- ML figures out the input and/or output types for simple expressions, constant declarations, and function declarations
  - Type checking requires that type expression of functions and their arguments match, and that type expression of context match result of function
  - If the default isn't what you want, you can specify the input and output types
  - E.g.
    ```
    fun divideBy2 (y : real) = y / 2.0;
    ```
- What is this doing?
  - `fun foo (m, n) = if m > n then [] else m :: foo(m+1, n);`
  - `> foo(1,6); // [1,2,3,4,5,6]`

ML: local environment using `let`

- Syntax
  ```
  let
  val <variable> = <expression>;
  …..
  val <variable> = <expression>;
  in
  <expression>
  end;
  ```
- Let allows declarations to be used in expressions
- Example:
  - Compute hundredth power of a number
    ```
    fun hundredthPower (x : real) =
      let
        val four = x * x * x * x;
        val twenty = four * four * four * four * four
      in
        twenty* twenty* twenty* twenty* twenty
      end;
    ```

ML: pattern matching

- Syntax
  ```
  fun <func> <pattern> = <expression>;
  | <func> <pattern> = <expression>
  …..
  | <func> <pattern> = <expression>
  ```
- Define a function by a series of equations, LHS is a pattern.
  - Always put the most specific pattern first
  - ML interpreter will use the first equation whose LHS matches
- Example:
  - Fibonacci function (`a₀ = a₁ = 0, a₂ = 1, 1, 2, 3, 5, 8, 13, 21,...)`
    ```
    fun fib n =
      if n = 0 then 0
      else if n = 1 then 1
      else fib(n-1) + fib(n-2);
    ```
- Pattern matching is powerful:
  - Allows the programmer to see the arguments
  - No more `hd`'s and `tl`'s sprinkled all over the place

ML: pattern matching – cont’d

- Examples:
  - Sum all the elements in a list of integers
    ```
    fun listsumL =
      if (null L) then 0
      else (hd L) + listsum(tl L);
    ```
  - Reversing a list
    ```
    fun reverse L =
      if L = nil then nil
      else reverse(tl L) @ [hd L];
    ```
  - Return first n elements of a list
    ```
    fun take (L, n) =
      if n = 0 then []
      else if n > 0 then (hd L :: take(tl L, n-1)) else []
    ```

fun 4 è 4 * (fact 3)
è 4 * 3 * (fact 2)
è 4 * 3 * 2 * (fact 1)
è 4 * 3 * 2 * 1 * (fact 0)
è 4 * 3 * 2 * 1 * 1
è 24

fun find(1,6) ; // [1,2,3,4,5,6]
ML: pattern matching – cont’d

- Patterns may consist of constants (integers, true, false...), tuples and variables. Arithmetic or logical expressions are invalid.
  - E.g. fun wrong(x,y) = "..."

- No duplicates in patterns
  - E.g. fun wrong_equal (x,y) = true
  | wrong_equal (x,y) = false.

- Pattern matching with wild cards
  - E.g. fun first (x,_) = x.
  - Matches anything like a variable. Binds nothing.
  - Avoid need to name every pattern

- ML does extensive pattern checking
  - E.g. - fun reverse (h::t) = reverse(t) @[h];
   > Warning: matchnonexhaustive

ML: Read-Evaluate-Print Cycle – cont’d

ML: pattern matching – cont’d

- How ML matches patterns?

ML: Read-Evaluate-Print Cycle

- The system response val it = indicates that the built-in name it always holds the result of the last evaluated command.

Midterm Review: introduction

- Abstraction levels of programming languages
  - Machine language
  - Assembly language
  - High-level

- Language translation:
  - Compilation
  - Interpretation

- Language Paradigms:
  - Imperative
  - Object-oriented
  - Functional
  - Logic-based

Midterm Review: language spec.

- Language Specification:
  - Syntax vs Semantics
  - Semantics-informal descriptions

- Context-free grammars:
  - How are CFG’s descriptions of languages?
  - Derivations
  - Parse trees
  - Ambiguity
  - Fixing an ambiguous grammar:
    - Introducing delimiters
    - Imposing associativity/precedence
    - Inherently ambiguous grammar
Consider the following BNF grammar for a new language called Cork:

```plaintext
<sequence> ::= <statement> | <sequence> <statement>
<statement> ::= <assignment>  | <alternation> | <iteration>
<assignment> ::= <variable> = <constant>
<alternation> ::= if <variable> <sequence> fi | if <variable> else <sequence> fi
<iteration> ::= do <variable> <sequence> od
<variable> ::= A | B | C | ... | Z
<constant> ::= 0 | 1 | 2 | ... | 9
```

For each of the following Cork code fragments, indicate whether a Cork parser would produce a syntax error or not. Explain how the parser will determine that.

- **a)** if A = 1 fi
  - **Answer:** Syntax error
  - **Explanation:** The parser will error on the equality assignment if there is no variable on the left side.

- **b)** A = 1
  - **Answer:** Valid
  - **Explanation:** This is a valid assignment statement.

- **c)** A = B = C = 3
  - **Answer:** Syntax error
  - **Explanation:** The parser will error on the equality assignment if there is no variable on the left side.

- **d)** do Z if A = 1 B = 2 else C = 3 fi od
  - **Answer:** Syntax error
  - **Explanation:** The `do` statement requires an `od` to close it, but it is missing.

- **e)** if A do Z if B C = 3 fi od fi
  - **Answer:** Valid
  - **Explanation:** This is a valid code fragment as it follows the grammar rules.

Using the following grammar:

```plaintext
<assign>  ::=  <id> = <expr>
<id>     ::= A | B | C
<expr>   ::= <id> + <expr>
           | <id> * <expr>
           | ( <expr> )
           | <id>
```

Show a parse tree and a leftmost derivation for

```
A = A * (B + (C * A))
```

**Answer:**

```
A = A * (B + (C * A))
```

**Midterm Review: Scheme questions**

- **What is a high-order function? Give an example.**

- **What does it mean for a language to be referentially transparent?**

- **What is the manifest interface principle?**

**Suppose we have the following definitions:**

```scheme
(define w 42)
(define x 17)
(define y (lambda (n) (+ n w)))
(define z (lambda (x) (+ x 2)))
```

What is the result of the following expressions?

```scheme
(map (lambda(x) (z x)) '(1 2 3 4))
(3 4 5 6)
(let ( (w (map y '(1 2 3 4)))) w)
(43 44 45 46)
```
Midterm Review: Scheme questions

• The following function finds if a particular item is found anywhere in an expression:

\[
\text{within?} \quad (\lambda (\text{item} \ \text{lst}) \quad (\text{or} \quad \text{equal?} \ \text{item} \ \text{lst}) \quad \\
(\text{and} \quad \text{pair?} \ \text{lst}) \quad (\text{or} \quad \text{within?} \ \text{item} \ \text{car} \ \text{lst}) \quad \\
(\text{within?} \ \text{item} \ \text{cdr} \ \text{lst})))
\]

\[
\Rightarrow \quad \text{(within? 'c '(a b c d))}
\]

\[
\Rightarrow \quad \text{(within? 'd '(a b c))}
\]

Show a trace for (within? 'c '(a (b c))) Your trace should explain the sequence of calls in the recursion tree.

Midterm Review: Scheme questions

• The following function finds if a particular item is found anywhere in an expression:

\[
\text{within?} \quad (\lambda (\text{item} \ \text{lst}) \quad (\text{or} \quad \text{equal?} \ \text{item} \ \text{lst}) \quad \\
(\text{and} \quad \text{pair?} \ \text{lst}) \quad (\text{or} \quad \text{within?} \ \text{item} \ \text{car} \ \text{lst}))) \quad \text{[1]}
\]

\[
\Rightarrow \quad \text{(within? 'c '(a (b c))) [1]}
\]

\[
\Rightarrow \quad \text{(within? 'c 'a (b c)) [2]}
\]

Show a trace for (within? 'c '(a (b c))) Your trace should explain the sequence of calls in the recursion tree.