Programming Languages

Lecturer: William W.Y. Hsu
Chapter 10 – Functional Languages
What is Functional Programming?

- Functional programming defines the outputs of a program as a mathematical function of the inputs.
  - Functional programming is a declarative programming style (programming paradigm).
  - A program in a functional programming language is basically compositions of functions.
    - The basic building block of such programs is function.
    - Functions produce results (based on arguments), but do not change any memory state -- pure functions do not have side effect.
      - Everything is an expression, not assignments.
  - In an imperative language, a program is in general a sequence of assignments.
    - Assignments produce results by changing the memory state.
Pros and Cons of Functional Programming?

**Pro:** flow of computation is declarative, i.e. more implicit
- The program does not specify the sequence of actions for the computation.
- This gives the language implementer more choices for optimizations.

**Pro:** promotes building more complex functions from other functions that serve as building blocks (component reuse).

**Pro:** behavior of functions defined by the values of input arguments only (no side-effects via global/static variables)
- This makes it easier to reason about the properties of a program.

**Cons:** Not everything can be easily or effectively expressed by stateless (pure) functions.

**Cons:** programmers prefer imperative programming constructs such as statement sequencing, while functional languages emphasize function composition.
Concepts of Functional Programming

• A pure function maps the inputs (arguments) to the outputs with no notion of internal state (no side effects)
  ▫ Pure functional programming languages only allow pure functions in a program
  ▫ A pure function can be counted on to return the same output each time we invoke it with the same input parameter values
    • Can be emulated in traditional languages such as C++ and java: expressions in general behave like pure functions; many routines without using global variables behave like pure functions.
  ▫ No global (statically allocated) variables
  ▫ No explicit (pointer) assignments
    • Dangling pointers and un-initialized variables cannot occur!

• Example pure functional programming languages: Miranda, Haskell, and Sisal
Concepts of Functional Programming

• Non-pure functional programming languages include “imperative features” that cause side effects (e.g. destructive assignments to global variables or assignments/changes to lists and data structures)

• Example: Lisp, Scheme, and ML, F#
Functional Language Constructs

- Building blocks are functions
- No statement composition, only function composition
- No variable assignments, but can use local “variables” to hold a value assigned once
- No loops, only recursion
  - List comprehensions in Miranda and Haskell
  - But: “do-loops” in Scheme
- Conditional flow with if-then-else or argument patterns
  - e.g. (if exp then_exp else_exp)
- Functional languages are statically (Haskell) or dynamically (Lisp) typed

Haskell examples:

```haskell
gcd a b
| a == b = a
| a > b = gcd (a-b) b
| a < b = gcd a (b-a)
```

```haskell
fac 0 = 1
fac n = n * fac (n-1)
```

```haskell
member x []   = false
member x (y:xs)
| x == y = true
| x <> y = member x xs
```
The Theory and Origin of Functional Languages:

- **Church's thesis:**
  - All models of computation are equally powerful.
  - Turing's model of computation: Turing machine.
    - Reading/writing of values on an infinite tape by a finite state machine.
  - Church's model of computation: Lambda Calculus.
  - Functional programming languages implement Lambda Calculus.

- **Computability theory**
  - A program can be viewed as a constructive proof that some mathematical object with a desired property exists.
  - A function is a mapping from inputs to output objects and computes output objects from appropriate inputs
    - For example, the proposition that every pair of nonnegative integers (the inputs) has a greatest common divisor (the output object) has a constructive proof implemented by Euclid's algorithm written as a "function".
Can Programming be Liberated from the von Neumann Style?

- This is the title of a lecture given by John Backus when he received the Turing Award in 1977.
- In this, he pointed out that the program should be abstract description of algorithm rather than sequences of changes in the state of the memory.
  - He called for raising the level of abstraction.
  - A way to realize this goal is functional programming.
- Programs written in modern functional programming languages are a set of mathematical relationships between objects.
  - No explicit memory management takes place.
Impact of Functional Languages on Language Design

- Useful features are found in functional languages that are often missing in procedural languages or have been adopted by modern programming languages:
  - *First-class function values*: the ability of functions to return newly constructed functions
  - *Higher-order functions*: functions that take other functions as input parameters or return functions
  - *Polymorphism*: the ability to write functions that operate on more than one type of data
  - *Aggregate constructs* for constructing structured objects: the ability to specify a structured object in-line such as a complete list or record value
  - *Garbage collection*
Functional Programming Today

- Significant improvements in theory and practice of functional programming have been made in recent years
  - Strongly typed (with type inference)
  - Modular
  - Sugaring: imperative language features that are automatically translated to functional constructs (e.g. loops by recursion)
  - Improved efficiency

- Remaining obstacles to functional programming:
  - Social: most programmers are trained in imperative programming and aren’t used to think in terms of function composition
  - Commercial: not many libraries, not very portable, and no IDEs
Applications

• Many (commercial) applications are built with functional programming languages based on the ability to manipulate symbolic data more easily.

• Examples:
  ▫ Computer algebra (e.g. Reduce system).
  ▫ Natural language processing.
  ▫ Artificial intelligence.
  ▫ Automatic theorem proving.
  ▫ emacs.
  ▫ Google’s map-reduce
Functional Features

- Most functional languages provide:
  - Functions as first-class values.
  - Higher-order functions.
  - List Type (operators on lists).
  - Recursion.
  - Structured function return.
  - Garbage collection.
  - Polymorphism and type inference.
No Hard Dividing Line

• You can write in a “functional” style in imperative languages.
• Most functional languages have imperatives subsets:

```scheme
(define iter-fib (lambda (n)
    (do ((i 0 (+ i 1)) ; init 0
         (a 0 b) ; init 0, set to b in each iter
         (b 1 (+ a b))) ; init 1, set to sub of a & b
         ((= i n) b) ; termination test and final value
        (display b)
    (display " "))))

(iter-fib 10)
```
So Why Functional?

- Teaches truly recursive algorithms.
- Easy Polymorphism.
  - How do you write in Java a method that will operate equally properly on reals as well as strings?
- Natural expressiveness for symbolic and algebraic computations.
  - Algorithms vs system Munging.
History

- Lambda-calculus as semantic model (Church).
- LISP (1958, MIT, McCarthy).

(defun fib (n)
  (if (or (= n 0) (= n 1))
    1
    (+ (fib (- n 1))
       (fib (- n 2)))))
History

- Lisp
  - Dynamic Scoping
- Common Lisp (CL), Scheme
  - Static scoping
- ML, Haskell, Miranda
  - Typing, type inference, pure functional
- FP (Backus): Sort-of functional APL
LISP Properties

• The original functional language and implementation of Lambda Calculus.
• Lisp and dialects (Scheme, common Lisp) are still the most widely used functional languages.
• Homogeneity of programs and data
  ▫ Programs are lists and a program can examine/change itself while running.
• Self-Definition
  ▫ Easy to write a Lisp interpreter in Lisp.
• Interactive: user interaction via \texttt{read-eval-print} loop.
Scheme – First Introduction

• Scheme is a popular Lisp dialect
• Lisp and Scheme adopt a form of prefix notation called *Cambridge Polish* notation
• Scheme is case insensitive.
• A Scheme expression is composed of
  ▫ Atoms, e.g. a literal number, string, or identifier name,
  ▫ Lists, e.g. '(a b c)
  ▫ Function invocations written in list notation: the first list element is the *function* (or operator) followed by the arguments to which it is applied:

\[(function \text{arg}_1 \text{arg}_2 \text{arg}_3 \ldots \text{arg}_n)\]

▪ For example, \(\sin(x^2+1)\) is written as \((\sin (+ (* x x) 1))\)
Cambridge Polish Notation

• How to express $100+200*300-400/20$?

• What is the traditional notion for
  $$\sin (- (+ 10 20) (* 20 40))$$
An Overview of Scheme

- Scheme is a particularly elegant Lisp.
  - Interpreter runs a read-eval-print loop.
  - Things typed into the interpreter are evaluated (recursively) once.
  - Anything in parentheses is a function call (unless quoted)
  - Parentheses are NOT just grouping, as they are in Algo-family languages.
  - Adding a level of parentheses changes meaning.
    \[(+ \ 3 \ 4) \Rightarrow 7\]
    \[((+ \ 3 \ 4)) \Rightarrow \text{error}\]
An Overview of Scheme

- Scheme:
  - Boolean values #\texttt{t} and #\texttt{f}
  - Numbers
  - Lambda Expressions
  - Quoting

\[
(+ 3 4) \Rightarrow 7 \\
(quote (+ 3 4)) \Rightarrow (+ 3 4) \\
\texttt{'(} + 3 4 \texttt{)} \Rightarrow (+ 3 4)
\]
Read-Eval-Print

• The "Read-eval-print" loop provides user interaction in Lisp/Scheme.

• An expression is read, evaluated, and the result printed
  ▫ Input: 9
  ▫ Output: 9
  ▫ Input: (+ 3 4)
  ▫ Output: 7
  ▫ Input: (+ (* 2 3) 1)
  ▫ Output: 7

• Guess how to quit scheme?
Functions

• The lambda call defines functions:
  \( (\text{lambda} \ (x) \ (* \ x \ x)) \)

• Evaluates within scope:
  \( (\text{lambda} \ (x) \ (* \ x \ x) \ 3) \Rightarrow 9 \)
Conditional

- Scheme:
  - Conditional expressions
  - Imperative stuff
  - Assignments
  - Sequencing (begin)
  - Iteration
  - I/O (read, display)

Code examples:

```
(if (< 2 3) 4 5) => 4
(cond
  ((< 3 2) 1)
  ((< 4 3) 2)
  (else 3)) => 3
```
Binding

- You can bind values to names.

(let ((a 3)
       (b 4)
       (square (lambda (x) (* x x)))
       (plus +))
  (sqrt (plus (square a) (square b)))) => 5
Binding

• Scoping

(let ((a 3)
     (let ((a 4)
          (b a))) ; uses the outer value of “a” (a=3)
(+) => 7 ; uses the inner value of “a” (a=4)
Binding

- Scoping ("all at once")

(letrec ((fact
    (lambda (n)
      (if (= n 1) 1
       (* n (fact (- n 1))))))
    ; uses the "inner" fact
 (fact 5))
Lists and Numbers

- The functions \texttt{car}, \texttt{cdr}, and \texttt{cons} allow us to manipulate lists.
  - \texttt{car} has been replaced by \texttt{first} in Common Lisp.
  - \texttt{cdr} has been replaced by \texttt{rest} in Common Lisp.

\begin{verbatim}
(car '(2 3 4)) => 2 ;Returns the head
(cdr '(2 3 4)) => (3 4) ;Returns the tail
(cons 2 '(3 4)) => (2 3 4) ;Amend the List
\end{verbatim}
Overview

• Scheme standard functions:
  ; arithmetic
  ; boolean operators
  ; equivalence
  ; list operators
  (symbol? x)
  (number? x)
  (complex? x)
  (real? x)
  (rational? x)
  (integer? x)
Overview of Scheme

• We’ll invoke the program by calling a function called ‘simulate’, passing it a DFA description and an input string.

• The automaton description is a list of three items:
  ▫ Start state.
  ▫ Transition function.
  ▫ Set of final states.

• The transition function is a list of pairs.
  ▫ The first element of each pair is a pair, whose first element is a state and whose second element in an input symbol.
  ▫ If the current state and next input symbol match the first element of a pair, then the finite automaton enters the state given by the second element of the pair.
(define simulate
  (lambda (dfa input)
    (cons (car dfa)
      (if (nul? input)
        (if (infinal? dfa) '(accept) '(reject))
        (simulate (move dfa (car input)) (cdr input))))))

(define move
  (lambda (dfa symbol)
    (let ((curstate (car dfa)) (trans (cadr dfa)) (finals (caddr dfa))
      (list
        (if (eq? curstate 'error)
          'error
          (let ((pair (assoc (list curstate symbol) trans)))
            (if pair (cadr pair) 'error)))
          trans
          finals))))
Evaluation Order

- Functional programs are evaluated following a reduction (or evaluation or simplification) process.
- There are two common ways of reducing expressions:
  - Application order
    - Impatient evaluation.
    - What you're used to in imperative languages.
    - Usually faster.
  - Normal order
    - Lazy evaluation.
    - Like call-by-name: don't evaluate `arg` until you need it.
    - Sometimes faster.
    - Terminates if anything will (Church-Rosser theorem).
Evaluation Order

• In Scheme:
  ▫ Functions use applicative order defined with lambda.
  ▫ Special forms (aka macros) use normal order defined with syntax-rules.

• A *strict language* requires all arguments to be well-defined, so applicative order can be used.

• A *non-strict language* does not require all arguments to be well-defined; it requires normal-order evaluation.
Applicative Order

- In applicative order, expressions are evaluated following the parsing tree (deeper expressions are evaluated first).

\[
\text{square } (3 + 4) \\
\quad = \{ \text{definition of } + \} \\
\quad \text{square } 7 \\
\quad = \{ \text{definition of square } \} \\
\quad \quad 7 \times 7 \\
\quad = \{ \text{definition of } \ast \} \\
\quad \quad 49
\]
Applicative Order

(define switch (lambda (x a b c)
    (cond ((< x 0) a)
          ((= x 0) b)
          ((> x 0) c))))
Applicative Order

• Evaluating the expression (switch -1 (+ 1 2) (+ 2 3) (+ 3 4)) in applicative order, we have

\[(\text{switch} \ -1 \ (+ \ 1 \ 2) \ (+ \ 2 \ 3) \ (+ \ 3 \ 4))\]
\[\Rightarrow \ (\text{switch} \ -1 \ 3 \ (+ \ 2 \ 3) \ (+ \ 3 \ 4))\]
\[\Rightarrow \ (\text{switch} \ -1 \ 3 \ 5 \ (+ \ 3 \ 4))\]
\[\Rightarrow \ (\text{switch} \ -1 \ 3 \ 5 \ 7)\]
\[\Rightarrow \ (\text{cond} \ ((< \ -1 \ 0) \ 3)\]
\[\Rightarrow \ (\text{cond} \ ((= \ -1 \ 0) \ 5)\]
\[\Rightarrow \ (\text{cond} \ ((> \ -1 \ 0) \ 7))\]
\[\Rightarrow \ 3\]
Normal Order

• In normal order, expressions are evaluated only as their value is needed.

\[
\text{square } (3 + 4) \\
= \{ \text{definition of square } \} \\
(3 + 4) * (3 + 4) \\
= \{ \text{definition of + applied to first term } \} \\
7 * (3 + 4) \\
= \{ \text{definition of + applied to second term } \} \\
7 * 7 \\
= \{ \text{definition of * } \} \\
49
\]
Normal Order

\[(\text{switch} \ -1 \ (+ \ 1 \ 2) \ (+ \ 2 \ 3) \ (+ \ 3 \ 4))\]
\[\Rightarrow \ (\text{cond} \ ((< \ -1 \ 0) \ (+ \ 1 \ 2)) \ (\ (= \ -1 \ 0) \ (+ \ 2 \ 3)) \ (\ (> \ -1 \ 0) \ (+ \ 3 \ 4)))\]
\[\Rightarrow \ (\text{cond} \ (#t \ (+ \ 1 \ 2)) \ (\ (= \ -1 \ 0) \ (+ \ 2 \ 3)) \ (\ (> \ -1 \ 0) \ (+ \ 3 \ 4)))\]
\[\Rightarrow \ (+ \ 1 \ 2)\]
\[\Rightarrow \ 3\]
Evaluation Order and Infinity

- Normal is sometimes more efficient than applicative order (Why?)
- Normal order can handle expressions that never converge to normal forms.

```plaintext
fun looper x = x*looper(x):int;
fun doit (flag, arg, func) =  
    if flag  
        then func(arg): int  
    else 1;
fun do2(flag, arg) = if flag then arg else 1;
doit(true, 5, looper);
doit(flase, 5, looper);
do2(false, looper(5))
```
Haskell Evaluation Order

- Haskell is a lazy functional programming language.
  - Expressions are evaluated in normal order.
  - Identical expressions are evaluated only once.

\[
\text{square (3 + 4)} \\
= \{ \text{definition of square} \} \\
(3 + 4) \times (3 + 4) \\
= \{ \text{definition of } + \text{ applied both terms} \} \\
7 \times 7 \\
= \{ \text{definition of } \times \} \\
49
\]
Functional Programming in Perspective

- Advantages of functional languages:
  - Lack of side effects makes programs easier to understand.
  - Lack of explicit evaluation order (in some languages) offers possibility of parallel evaluation (e.g. MultiLisp).
  - Lack of side effects and explicit evaluation order simplifies some things for a compiler (provided you don't blow it in other ways).
  - Programs are often surprisingly short.
  - Language can be extremely small and yet powerful.
Functional Programming in Perspective

- Problems
  - Difficult (but not impossible!) to implement efficiently on von Neumann machines.
  - Lots of copying of data through parameters.
  - (apparent) need to create a whole new array in order to change one element.
  - Heavy use of pointers (space/time and locality problem).
  - Frequent procedure calls.
  - Heavy space use for recursion.
  - Requires garbage collection.
  - Requires a different mode of thinking by the programmer.
  - Difficult to integrate I/O into purely functional model.
ML History

• ML Stands for “Meta-language”.
• Developed in 1970s by Robert Milner at the University of Edinburgh.

• Characteristics
  ▫ Functional control structures.
  ▫ Strict, formal semantics (provable correctness).
  ▫ Strict polymorphic type system.
    • Coercion not allowed.
  ▫ Still subject of active industry research.
    • Microsoft is working on variant called F# for their .NET framework.
Function Definitions in ML

- Tail-Recursive Functions:

```ml
fun fib(n)=
  let fun fib_helper(f1, f2, i) =
      if i = n then f2
      else fib_helper(f2, f1+f2, i+1)
  in
  fib_helper(0,1,0)
end;

fib(7);
```

- Equivalent in speed (and machine code?) to iterative version!
- What is the **inferred** type of this function?
Types in ML

- Built-in Types:
  - Integer
  - Real
  - String
  - Char
  - Boolean

- From these we can construct
  - **Tuples**: Heterogeneous element types with finite fixed length
    - (#“a”, 5, 3.0, “hello”, true): char *int *real*string*bool
  - **Lists**:
    - [5.0, 3.2, 6.7] : real list
    - [(# “a”, 7), (# “b”, 8)]: (char *int)list
  - Functions
  - Records
Types inference in ML

• Everything is inferred; ML complains if anything is ambiguous.

```
fun circum(r) = r * 2.0 * 3.14159;
circum(7.0);
```

• What is the inferred type of \( r \)? Why?

  ▫ **r** must be of type **real**.
  • Can be explicit by defining fun `circum(r:real)`...

• Type of function `circum`:
  • real->real
Polymorphism in ML

- Consider the following function in ML:

```ml
fun compare(x,p,q) = 
  if x = p then 
    if x = q then "both"
    else "first"
  else 
    if x=q then "second"
    either "neither"
```

- What is the type of compare? x? p? q?
Polymorphism in ML

• `a*`a*`a->string

• All of these are valid:
  ▫ \texttt{compare(1,2,3)};
  ▫ \texttt{compare(1,1,1)};
  ▫ \texttt{let val t = ("larry", "moe", "curly") in compare(t) end;}

Type Checking

- ML verifies type consistency.
- Set of constraints
- All occurrences of same identifier (in same scope) have the same type.
  - In an `if...then..else...` construct, if condition must have type `bool`, and `then` and `else` must have same type.
  - Programmer defines functions have type `a->b` where `a` is type of function parameters and `b` is type of function return.
  - When function is called, the arguments passed and value returned must have same type as definition.
- Process of checking if two types are the same is called unification.
Lists in ML

- Heterogeneous & Homogeneous Lists operator:
- Appending (joining) two lists:

  \[ [1,4]@[3,5] \Rightarrow [1,4,3,5] \]
  \[ ("hi", 3.0)@(4, "bye") \Rightarrow ("hi", 3.0, 4, "bye") \]

- Prefixing a list with an item:

  \[ 1::[2,7,9] \Rightarrow [1,2,7,9]; \]
  \textbf{NOTE:} \[ [2,7,9]::1 \] \text{ is illegal (use } [2,7,9]@[1] \text{ instead)}
Lists in ML

- Other useful List functions:
  
  \[ \text{hd} = \text{head} \]
  \[ \text{tl} = \text{tail} \]
  \[ \text{nth} = \text{list element selector} \]
  \[ \text{rev} = \text{reverse a list} \]
  \[ \text{length} = \text{number of elements} \]
Conditional Expressions & Pattern Matching in ML

• Case Statement:

```ml
fun append(l1,l2) = 
    case l1 of 
        nil => l2 
    | h :: t => h::append(t,l2);
```

• If $L1 = \text{nil}$ then return $L2$.
• If $L1$ has form “$h::t$” then return “$h::\text{append}(t,L2)$”.
Function Pattern Matching in ML

- Function definition as series of alternatives:

```ml
fun appends(l1, l2) = 
  if l1 = nil then l2
  else hd(l1) :: append(tl(l1), l2);
```

- Becomes

```ml
fun append(nil, l2) = l2
| append(h::t, l2) = h :: append(t, l2);
```
Function Pattern Matching in ML

- More complex example:

```ml
fun split(nil) = (nil, nil)
  | split([a]) = ([a], nil)
  | split(a::b::cs) = 
    let val (M,N) = split(cs)
    in
    (a::M, b::N)
  end;
```
Higher-Order Functions

• Higher-order functions are functions that take functions as arguments or return a function as a result.

```haskell
fun map(F, nil) = nil
    | map(F, x::cs) = F(x)::map(F,xs);

fun add5(x) = x+5;
map(add5, [3,24,7,9]); => [8,29,12,14]
map(fn x=> x+5, [3,24,7,9]); => [8,29,12,14]
```
Higher-order Functions

- Every function has an *order*:
  - A function that does not take any functions as parameters, and does not return a function value, has order 1
  - A function that takes a function as a parameter or returns a function value has *order* $n+1$, where $n$ is the order of its highest-order parameter or returned value
- The *quicksort* we just saw is a second-order function
“Currying” in ML

- **Currying** is a method in which a multiple argument function is replaced by a single argument function that returns a function with the remaining arguments.

```ml
fun add(x,y) = x + y : int;
>> val add = fn int * int -> int

fun add x = fn y=> x+y;
>> val add = fn int -> int ->int

fun add x y = x+y;
>> val add = fn int -> int ->int
```
“Currying” in ML

- Functions that take their arguments one at a time are said to be *curried*.
- Functions that take their arguments all at once as a tuple are said to be *uncurried*.
  - These terms are named for the mathematician Haskell B. Curry, for whom the functional programming language Haskell is also named.
  - It is easy to convert a curried function to its uncurried equivalent, and visa-versa.
Currying

- We've seen how to get two parameters into a function by passing a 2-tuple:
  \[ \text{fun } f \ (a,b) = a + b; \]
- Another way is to write a function that takes the first argument, and returns another function that takes the second argument:
  \[ \text{fun } g \ a = \text{fn } b \Rightarrow a+b; \]
- The general name for this is *currying*
Curried Addition

- `fun f (a,b) = a+b;`
  val f = fn : int * int -> int
- `fun g a = fn b => a+b;`
  val g = fn : int -> int -> int
- `f(2,3);`
  val it = 5 : int
- `g 2 3;`
  val it = 5 : int

• Remember that function application is left-associative
• So `g 2 3` means `((g 2) 3)`
Advantages

- No tuples: we get to write $g(2, 3)$ instead of $f(2, 3)$
- But the real advantage: we get to specialize functions for particular initial parameters

- `val add2 = g 2;
val add2 = fn : int -> int
val add2 3;
val it = 5 : int
val add2 10;
val it = 12 : int`
Advantages: Example

- Like the previous \texttt{quicksort}
- But now, the comparison function is a first, curried parameter

\begin{verbatim}
- quicksort (op <) [1,4,3,2,5];
  val it = [1,2,3,4,5] : int list
- val sortBackward = quicksort (op >);
  val sortBackward = fn : int list -> int list
- sortBackward [1,4,3,2,5];
  val it = [5,4,3,2,1] : int list
\end{verbatim}
Multiple Curried Parameters

- Currying generalizes to any number of parameters

```plaintext
- fun f (a,b,c) = a+b+c;
val f = fn : int * int * int -> int
- fun g a = fn b => fn c => a+b+c;
val g = fn : int -> int -> int -> int
- f (1,2,3);
val it = 6 : int
- g 1 2 3;
val it = 6 : int
```
Notation For Currying

• There is a much simpler notation for currying (on the next slide)

\[
\text{fun } g \ a = \text{fn } b \Rightarrow \text{fn } c \Rightarrow a+b+c;
\]

• The long notation we have used so far makes the little intermediate anonymous functions explicit

But as long as you understand how it works, the simpler notation is much easier to read and write
Easier Notation for Currying

- Instead of writing:
  
  ```
  fun f a = fn b => a+b;
  ```

- We can just write:
  
  ```
  fun f a b = a+b;
  ```

- This generalizes for any number of curried arguments

  ```
  fun f a b c d = a+b+c+d;
  ```

  ```
  val f = fn : int -> int -> int -> int -> int
  ```
Standard ML of New Jersey

- Download and Install from
  - http://www.smlnj.org/smlnk.html
- To run (after installation): Type “sml”
- Once typing in a definition, use Ctrl-C to escape to interpreter prompt.
- If you want to exit type:
  - OS.Process.exit(OS.Process.success);
Scope in ML is Lexical

- Top level environment has all **pre-defined bindings**.
- Every “val x = false;” binding adds another row to the symbol table when compiling/interpreting.
- Each row hides earlier bindings of the same name (does not destroy them).
- Local bindings can be made in functions definitions.
- Locals are removed from the symbol stack when the function definition is complete.
Scope in ML is Lexical

<table>
<thead>
<tr>
<th>name</th>
<th>cat</th>
<th>value</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>var</td>
<td>[1,2,3]</td>
<td>int list</td>
</tr>
<tr>
<td>sq</td>
<td>fn</td>
<td>( z : z \times 26.3 )</td>
<td>real -&gt; real</td>
</tr>
<tr>
<td>k2</td>
<td>var</td>
<td>26.3</td>
<td>real</td>
</tr>
<tr>
<td>x</td>
<td>var</td>
<td>false</td>
<td>bool</td>
</tr>
<tr>
<td>r1</td>
<td>var</td>
<td>(7,3.14)</td>
<td>int * real</td>
</tr>
</tbody>
</table>

val r1 = (7,3.14);
val x = false;
val k2 = 26.3;
fun sq z:real = z*k2;
val x = [1,2,3]
Binding Stack

- **When a name is referenced**, you search the stack from the top looking for the first occurrence of the name.
- This corresponds to the **most recently created binding**... in ML this will either by the global binding, or a `let` binding.
- **Evaluating expressions that give value to be bound use the current binding in the stack.**
  - See the `defn` of “sq,” uses the binding to “k2” and puts the value “26.3” in the body of the function.
Binding Stack

- When a function is defined:
  - Name is looked up in the stack to get the definition.
  - Local (“let”) bindings are added to the binding stack.
  - Local binding are popped off the stack at end of the definition.
ML Example: Static Scope

val x = 5;
fun f1 z = z*x;
fun f2 z =
  let val x = 2
  in (f1 z) * x
  end;

f1 4; (*evals to 20 *)
f2 4; (*evals to 40 with static scope *)
(*dynamic scope would cause eval to 16*)
Name/Value Bindings

• May have to search top to bottom (deep) to find a binding $O(n)$ complexity.
• This style table of name/value pairs comes from old Lisp.
Hash table of stacks

- curLev 1
- stack 1: sq, fn, \textbackslash z\textbackslash z26.3, real->real
- stack 2: x, var, [1,2,3], int list
- stack 3: x, var, false, bool
- stack 4: r1, var, (7,3.14), int * real
- stack 5: k2, var, 26.3, real
## Dynamic Scope

### Deep Access A-List

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>var</td>
<td>[1,2,3] int list</td>
</tr>
<tr>
<td>sq</td>
<td>fn</td>
<td>(z:z*26.3) real -&gt; real</td>
</tr>
<tr>
<td>k2</td>
<td>var</td>
<td>26.3 real</td>
</tr>
<tr>
<td>x</td>
<td>var</td>
<td>false bool</td>
</tr>
<tr>
<td>r1</td>
<td>var</td>
<td>(7,3.14) int * real</td>
</tr>
</tbody>
</table>

### Shallow Access CET

- cur

```
  ...
  ...
  ...
  ...
  ...
  ...
  ...
  ...
  ...
  ...
```

2016/12/13
Dynamic Scope

• Deep Access A-List
  ▫ O(n) search time.

• Shallow Access CET
  ▫ O(1) search time but maintenance at each function call/return.

• Some times deep access and shallow access are called “deep binding” and “shallow binding” however these are different concepts.
Binding of Referencing Environments

• Scope rules are used to determine the reference environment.
  ▫ Static and dynamic scoping

• Some languages allow references to subroutines.
  ▫ Are the scope rules applies when the reference is created or when the subroutine is called?

• In **shallow (late) binding**, the referencing environment is bound when the subroutine is called.

• In **deep (early) binding**, the referencing environment is bound when the reference is created.
Binding of Referencing Environments

x: integer := 1
y: integer := 2

procedure add
  x := x + y

procedure second(P:procedure)
  x:integer := 2
  P()

procedure first
  y:integer := 3
  second(add)

----main starts here---
first()
write_integer(x)
Binding of Referencing Environments

- Deep binding binds the environment at the time the procedure is passed as an argument.
- Shallow binding binds the environment at the time the procedure is actually called.
- For dynamic scoping with deep binding when add is passed into second the environment is \( x = 1, y = 3 \) and the \( x \) is the global \( x \) so it writes 4 into the global \( x \), which is the one picked up by the `write_integer`.
- Shallow binding just traverses up until it finds the nearest variable that corresponds to the name so the answer would be 1.


Binding of Referencing Environments

\[
\text{thres: integer} \\
\text{function older(p:person): boolean} \\
\text{~~~~~~~~~~return p.age>thres} \\
\text{procedure show(p:person, c:function)} \\
\text{~~~~~~~~~~thres:integer} \\
\text{~~~~~~~~~~thres:=20} \\
\text{if c(p)} \\
\text{~~~~~~~~~~write(p)} \\
\text{procedure main(p)} \\
\text{~~~~~~~~~~thres:=35} \\
\text{show(p, older)}
\]
Binding of Referencing Environments

- **Deep binding**: reference environment of `older` is established with the first reference to `older`, which is when it is passed as an argument to `show`.

```plaintext
main(p)
thres:=35 <----------------+
show(p, older) |
  thres: integer |
  thres:=20 |
older(p) |
  return p.age>thres +
  if <return value is true>
    write(p)
```

- Program prints person `p` if older than 35.
Binding of Referencing Environments

- **Shallow binding**: reference environment of `older` is established with the call to `older` in `show`.

```plaintext
main(p)
  thres:=35
  show(p, older)
    thres:=integer
    thres:=20 <->
    older(p) |
      return p.age>thres ->
    if <return value is true>
      write(p)
```

- Program prints person `p` if older than 20.
program BindingExample(input, output);
procedure A (I: integer; procedure P);
procedure B;
begin
    writeln(I);
end;

begin (* A *)
    if I > 1 then
        P
    else
        A(2,B);
end;

procedure C; begin end;

being (* main *)
    A(1,C);
end.
Deep and Shallow Binding

- **Deep Binding, Shallow Binding** are both concepts related to giving a function/subroutine a referencing environment in which to run.

- This is important when a subprogram is passed in or out as a parameter to another (i.e., a “funarg”).

- Some questions:
  - When a funarg that is passed in is run, does it use the environment it has when run or the one when defined?
  - Also, when a funarg is passed out and run the environment it created in is gone; how do we deal with threat?
Example

- Pascal uses static scoping.
- Prints 2 if shallow binding is used.
- Prints 1 if deep binding is used.

```pascal
program BindingExample(input, output);

procedure A (I: integer; procedure P);
procedure B;
begin
    writeln(I);
end;

begin (* A *)
    if I > 1 then
        P
    else
        A(2,B);
end;

procedure C; begin end;

being (* main *)
    A(1,C);
end.
```
Closures

• Deep binding is implemented using **closures**

• A closure is the **combination of a reference to a subroutine and an explicit representation of its referencing environment**.

• Typically implemented with
  ▫ A pointer to the subroutines code.
  ▫ If the scoping is dynamic, we need a way to temporarily unroll all the changes since the reference was created.
FunArgs: Problem

• The failure of traditional stack-based implementations of procedure calls in the presence of “first-class” functions (functions that can be passed as procedure parameters and returned as procedure results).

• **Upwards funarg problem**: The problem of returning a function as a procedure result; requires (i) allocating stack frame on the heap and (ii) returning a closure containing a pointer to code and a pointer of the enclosed stack frame.

• **Downwards funarg problem**: the problem of passing a function as a procedure parameter; requires a tree structure for stack frames.
Referential Transparency

- Functional programing languages try to enforce referential transparency.
- A binding is immutable...
- Any time you see a name, you may substitute in the value bound to that name and NOT alter the semantics of the expression.
- “no side effects.”
Referential Transparency

“equals can be substituted for equals”

If two expressions are defined to have equal values, then one can be substituted for the other in any expression without affecting the result of the computation.

For example, in

\[
s = \sqrt{2}; \quad z = f(s, s); \quad \text{we can write}
\]
\[
z = f(\sqrt{2}, \sqrt{2});
\]
A function is called **referentially transparent** if given the same parameter(s), it always **returns the same result**.

In **mathematics**, all functions are referentially transparent,

In **programming**, this is not always the case, with use of imperative features in languages.

- The subroutine/function called could affect some global variable that will cause a second invocation to return a different value.
- Input from keyboard.
Why is referential transparency important?

• Because it allows the programmer to reason about program behavior, which can help in proving correctness, finding bugs that couldn’t be found by testing, simplifying the algorithm, assist in modify the code without breaking it, or even finding ways of optimizing it.

```c
s = sqrt(9);
x = s*s + 17 *k / (s-1);
// can replace x with:
// sqrt(9)*sqrt(9) + 17 *k/(sqrt(9)-1) = 9+17*k/2;
```