Principles of Programming Languages V

Wael Aboelsaadat
wael@cs.toronto.edu

http://www.dgp.toronto.edu/~wael/324.html

Acknowledgement: Some of the material in these lectures are based on material by Diane Horton
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.

```java
public class Shape{
    public void draw( int x, int y){
        // do nothing
    }
}

public class Rectangle extends Shape{
    public void draw( int x, int y){
    }
}

public class Circle extends Shape{
    public void draw( int x, int y){
    }
}

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 allow 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
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)
Typing: why?

- **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?!
    - Can assume a buffer is 1K! (when really it is only 512 bytes?!)  
      - 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.
Typing: why?

- **Abstraction**
  - Types allow programmers to think about programs in higher level, not bothering with low-level implementation.
  
  - E.g.

    // In pseudo-C
    void initqueue(int* head, int* tail, type* queue) {
    {
    ...
    
    int isEmpty (int* head, int* tail, type* queue) {
    ... }
    
    int isFull (int* head, int* tail, type* queue) {
    ...
    
    int enqueue (int* head, int* tail, type* queue) {
    ...
    
    element dequeue (int* head, int* tail, type* queue) {
    ...
    
    class queue{
    // in C++
    int head;
    int tail;
    int nmelements;
    struct element queueentry[maxitem];
    
    void initqueue( ){
    ...
    
    int isEmpty ( ){
    ...
    
    int isFull( ){
    ...
    
    int enqueue ( element ditem){
    ...
    
    element dequeue ( ){
    ...
    
    }
Typing: why?

• **Modularity**
  – Allow programmers to express the interface between two subsystems.
  – This localizes the definitions required for interoperability of the subsystems and prevents inconsistencies when those subsystems communicate.
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.
    ```java
    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. \( \text{if test then} \ldots \)
    - Q: What type is test?
    - A: Must be Boolean
  - E.g. \( A3 := \text{Func} (\text{Test1}); \)
    - 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: 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
    • Standard ML (1991) & ML 2000
    • 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…)

<table>
<thead>
<tr>
<th>Time</th>
<th>Example</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1930’s</td>
<td>Lambda calculus (Church)</td>
<td></td>
</tr>
<tr>
<td>1950’s</td>
<td>Lisp (McCarthy)</td>
<td></td>
</tr>
<tr>
<td>1960’s</td>
<td>semantics, deconstruction</td>
<td></td>
</tr>
<tr>
<td>1970’s</td>
<td>FP (Backus)</td>
<td></td>
</tr>
<tr>
<td>1980’s</td>
<td>Miranda (Turner), ML (Milner)</td>
<td></td>
</tr>
<tr>
<td>1990’s</td>
<td>Haskell</td>
<td></td>
</tr>
</tbody>
</table>
ML: main features

- **Strong, static typing**
  - Quite a fancy type system!

- **Parametric polymorphism**
  - Similar to OOP *(in fact, it influenced OO)*

- **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

\[
gcd(m,n) = \begin{cases} 
  n & m = 0 \\
  gcd(n \mod m, m) & m > 0 
\end{cases}
\]

**Pascal**

```pascal
function gcd(m, n: integer): integer;
var prevm: integer;
begin
  while m<>0 do begin
    prevm := m; m := n mod m; n := prevm
  end;
gcd := n
end;
```

**Scheme**

```scheme
(define gcd
  (lambda (m n)
    (if (zero? n) m
        (gcd n (remainder m n))))))
```

**ML**

```ml
fun gcd(m, n) = 
  if m=0 then n else 
  gcd(n mod m, m);
```
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/type-case e.g. real(2) + 3.0 : real
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
- **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 and also 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…

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>not</td>
<td>Negation</td>
</tr>
<tr>
<td>andalso</td>
<td>Conjunction</td>
</tr>
<tr>
<td>orelse</td>
<td>Disjunction</td>
</tr>
<tr>
<td>if … then … else</td>
<td>conditional selection</td>
</tr>
</tbody>
</table>

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

• Syntax: \[ \text{obj}_1, \text{obj}_2, \ldots \]
  – Objects in a list must be homogenous (*same type*)
    • E.g.
      
      \[
      [1,2,3] : \text{int list} \quad \text{-- a list of integers}
      \]
      
      \[
      [\text{“dog”}, \text{“cat”}, \text{“moose”}] : \text{string list} \quad \text{-- a list of strings}
      \]
      
      \[
      [1.0,2.0,3.0] : \text{real list} \quad \text{-- a list of reals}
      \]
      
      \[
      [[1,2],[3,4],[5,6]] : \text{int list list} \quad \text{-- a list of lists of integers}
      \]
  – The empty list is written \([]\) or \text{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
    • E.g.
      
      \[
      2 :: [3,4] \quad \text{returns} \quad [2,3,4]
      \]
      
      \[
      [1,2] :: [ [3,4], [5,6]] \quad \text{returns} \quad ??
      \]
  – hd returns the first element of a list
    • E.g.
      
      \[
      \text{hd}[1,2,3] \quad \text{returns} \quad 1
      \]
  – tl returns the tail
    • E.g.
      
      \[
      \text{tl}[1,2,3] \quad \text{returns} \quad [2,3]
      \]
ML: constructor types - tuples

- **Syntax**
  - (obj\(_1\), obj\(_2\), ...)
  - 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,"ab"),"cd") : int * (real * string) * string
      - [(1,"a"),(3,"bc"),(7,"efg")]: (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.
      - val x = ~3; // -3
      - (3,"a",true) = (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 foo L = (1 + hd L) :: (tl L);` ??

• Functions have types too. ML interpreter will infer the type.
  – E.g.  
    `fun square x = x * x;`
    • The function square takes an integer as input and returns an integer as output.
    • This is written as `square: int → int` (→ indicates this a function)
ML: constructor types - functions

• **Anonymous functions**  
  \[ \text{fn } \langle \text{func-param} \rangle \Rightarrow \langle \text{func-body} \rangle \]  
  – E.g.  
  \[(\text{fn } n \Rightarrow n \ast 2)(9);\]

• **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.  
    \[ \text{fun divideBy2 } (y : \text{real}) = y / 2.0; \]

• **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: constructor types - functions

- **Recall syntax**
  
  ```ml
  fun <func-name> <input-param> = <expression>;
  ```

- **Functions without parameters**
  - E.g.
    
    ```ml
    - fun message () = "hello world";
    > val message = fn : unit → string
    ```

- **Functions with more than one parameter**
  - E.g.
    
    ```ml
    - fun birthday date = (date = (1, "Jan",1900));
    > val birthday = fn : int * string * int → bool
    ```

- **Functions returning more than 1 result**
  - E.g.
    
    ```ml
    - fun quotrem (x,y) = ( ( x div y), (x mod y));
    > val quotrem = fn : int * int → int * int
    - val (quot, rem) = quotrem(x,y);
    ```
### ML: constructor types - functions

- **Examples:**
  - Factorial \((n!\)\)
    
    ```ml
    fun fact n =  
    if n = 0 then 1  
    else n * fact(n-1);  
    ```

  - List reverse
    
    ```ml
    fun reverse L =  
    if L = nil then nil  
    else reverse(tl L) @ [hd L];  
    ```

    **Evaluation:**
    - \(\text{fun 4} \rightarrow 4 * (\text{fact 3})\)
      - \(4 * 3 * (\text{fact 2})\)
      - \(4 * 3 * 2 * (\text{fact 1})\)
      - \(4 * 3 * 2 * 1 * (\text{fact 0})\)
      - \(4 * 3 * 2 * 1 * 1\)
      - \(24\)

    - \(\text{reverse [1,2,3]} \rightarrow \text{reverse[2,3]} @ 1\)
      - \(\text{reverse[3]} @ 2 @ 1\)
      - \(\text{reverse[]} @ 3 @ 2 @ 1\)
      - \([3,2,1]\)
ML: local environment using let

- **Syntax**
  
  ```ml
  let
  val <variable_1> = <expression_1>;
  ....
  val <variable_n> = <expression_n>;
  in
  <expression>
  end;
  ```

- **Let allows declarations to be used in expressions**

- **Example:**
  
  - Compute hundredth power of a number

    ```ml
    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**
  
  \[
  \text{fun} \ <\text{func}> <\text{pattern}_1> = <\text{expression}_1> \\
  \text{  |} \ <\text{func}> <\text{pattern}_2> = <\text{expression}_2> \\
  \text{  \ldots\ldots} \\
  \text{  |} \ <\text{func}> <\text{pattern}_n> = <\text{expression}_n>
  \]

- **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 \(a_n = a_{n-1} + a_{n-2} : 0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots\)
    
    \[
    \text{fun fib n =} \\
    \text{  if n = 0 then 0} \\
    \text{  else if n = 1 then 1} \\
    \text{  else fib(n-1) + fib(n-2);}
    \]

- **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
    
    ```ml
    fun listsum L = 
      if (null L) then 0
      else (hd L) + listsum(tl L);
    
    fun listsum [] = 0
    | listsum L = (hd L) + listsum(tl L);
    
    fun listsum [] = 0
    | listsum(h::t) = h + listsum(t);
    ```

  - Reversing a list
    
    ```ml
    fun reverse L = 
      if L = nil then nil
      else reverse(tl L) @ [hd L];
    
    fun reverse(nil) = nil
    | reverse(h::t) = reverse(t) @ [h];
    ```

  - Return first n elements of a list
    
    ```ml
    fun take ([ ], I) = [ ]
    | take (h::tl, I) = if I > 0 then h::take (tl, I - 1) else [ ];
    ```
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: match nonexhaustive
ML: pattern matching – cont’d

- How ML matches patterns?

\[
\text{func } (x::y::zs, w) =
\]
\[
\mid (\ldots) =
\]
\[
\mid (\ldots) =
\]

\[
\text{func } ([1,2,3,4], 5)
\]

```
func(
  x::y::zs, w)
  =
  | (\ldots)
  | (\ldots)

func([1,2,3,4],5)
```
ML: Read-Evaluate-Print Cycle

The system response \texttt{val it =} indicates that the built-in name \texttt{it} always holds the result of the last evaluated command.
ML: Read-Evaluate-Print Cycle – cont’d

```
type: int
- 6 div 2; (* integer division *)
val it = 3 : int
- 7 div 3;
val it = 2 : int
- val freezingFahr = 32;
val freezingFahr = 32 : int
- fun fahrToCelsius f = (f - freezingFahr) * 5 div 9;
val fahrToCelsius = fn : int -> int
- fun celsiusToFahr c = c * 9 div 5 + freezingFahr;
val celsiusToFahr = fn : int -> int
- fahrToCelsius 0;
val it = ~18 : int
- fahrToCelsius 32;
val it = 0 : int
- GC #0.0.0.0.1.5: (0 ms)
fahrToCelsius 212;
val it = 100 : int
- celsiusToFahr 0;
val it = 32 : int
- celsiusToFahr 100;
val it = 212 : int
- celsiusToFahr 30;
val it = 86 : int
- `
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> <sequence> 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

b) A = 1

Answer: valid

c) A = B = C = 3

Answer: syntax error

d) do Z if A = 1 B = 2 else C = 3 fi od

Answer: syntax error

e) if A do Z if B C = 3 fi od fi

Answer: valid
Using the following grammar:

\begin{align*}
\text{<assign>} & \rightarrow \text{id} = \text{expr} \\
\text{id} & \rightarrow A \mid B \mid C \\
\text{expr} & \rightarrow \text{id} + \text{expr} \\
& \quad \mid \text{id} \ast \text{expr} \\
& \quad \mid (\text{expr}) \\
& \quad \mid \text{id}
\end{align*}

Show a parse tree and a leftmost derivation for

\[ A = A \ast (B + (C \ast A)) \]
Midterm Review: Syntax – cont’d

• Answer:

\[
\begin{align*}
\langle\text{assign}\rangle & \rightarrow \langle\text{id}\rangle = \langle\text{expr}\rangle \\
\langle\text{id}\rangle & \rightarrow A | B | C \\
\langle\text{expr}\rangle & \rightarrow \langle\text{id}\rangle + \langle\text{expr}\rangle \\
& \quad | \langle\text{id}\rangle \ast \langle\text{expr}\rangle \\
& \quad | (\langle\text{expr}\rangle) \\
& \quad | \langle\text{id}\rangle
\end{align*}
\]

\[
A = A \ast (B + (C \ast A))
\]

<assign> => <id> = <expr>

=> A = <expr>

=> A = <id> \ast <expr>

=> A = A * <expr>

=> A = A * ( <expr> )

=> A = A * ( <id> + <expr> )

=> A = A * ( B + <expr> )

=> A = A * ( B + ( <expr> ) )

=> A = A * ( B + ( <id> * <expr> ) )

=> A = A * ( B + ( C * <expr> ) )

=> A = A * ( B + ( C * <id> ) )

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

\[
\text{Tree diagram}
\]

- <assign>
  - <id> := <expr>
    - <id> * <expr>
      - A ( (expr) )
        - A
      - B
        - B
          - C
          - <id>
            - <id>
              - A
Midterm Review: functional PL

- Characteristics
- Basic constructs
- Lists
- Anonymous functions
- map, reduce, for-each...
- let, let*
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:

```
(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?

```
]=> (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:

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

]=> (within? 'b '(a b (c) d))
#t
]=> (within? 'd '(a b c))
#f

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:

\[
(\text{define within? } (\lambda (\text{item lst})
  (\text{or (equal? item lst)}
    (\text{and (pair? lst )}
      (\text{or (within? item (car lst)) } [1]
        (\text{within? item (cdr lst)))))) \text{) [2]})
\]

\[=> (\text{within? 'c '}(a (b c)))\]