NCS-455 FUNCTIONAL AND LOGIC PROGRAMMING LAB

LAB MANUAL

Diwakar Yagyasen

Department of CS&E, BBDNITM

Contents

Introduction to ML....4

Why ML? 5

History of ML....5

The interactive ML interpreter 6

Expressions, values, and bindings....7

`val` bindings....9

Aside: What about assignment? 11

A few words on type inference 12

Built-in compound data types 13

Records 14

Tuples 15

Lists....16

Uniform reference data model 18

All values are first-class citizens....18

Let-expressions and nested environments 19

Again: all values are first-class.... 20

: Functions and patterns 21

Function basics 21

Function syntax and types 21

Naming functions 22

Function application 22

Functions with no meaningful return value or arguments 24

Control: Branching, sequencing, and patterns....25

`if` expressions....25

Sequencing 26

Pattern-matching and `case` 27

Patterns, patterns, everywhere 30

Recursive functions 32

Recursion vs. iteration 33

Tail recursion 34

: Functions and patterns, supplementary notes 37

What is functional programming? 37

Language Construct X in a Nutshell 37

`if/then/else` in a Nutshell 37

Functions in a Nutshell 38

Sample exercises 39

: Functions and patterns, solutions to supplementary exercises 41

More on scoping of names 48

Lexical scoping 48

Lexical vs. dynamic scoping 50

Nested scopes: an extended example 50

Let-expressions and function application 51

NCS-455: FUNCTIONAL AND LOGIC PROGRAMMING LAB

Program in SML- NJ or CAML for following:

1. To implement Linear Search.

2. To implement Binary Search.

3. To implement Bubble Sorting.

4. To implement Selection Sorting.

5. To implement Insertion Sorting.

Implement using LISP

6. Write a function that compute the factorial of a number.(factorial of 0 is 1, and factorial of n is n*(n-1)*...1.Factorial is defined only for integers greater than or equal to 0.)

7. Write a function that evaluate a fully parenthesized infix arithmetic expression . For examples, (infix (1+(2*3))) should return 7.

8. Write a function that performs a depth first traversal of binary tree. The function should return a list containing the tree nodes in the order they were visited.

9. Write a LISP program for water jug problem.

10. Write a LISP program that determines whether an integer is prime.

Implement using PROLOG

11. Write a **PROLOG** program that answers questions about family members and relationships includes predicates and rules which define sister, brother, father, mother, grandchild, grandfather and uncle. The program should be able to answer queries such as the following:

1. father(x, Amit)

2. grandson(x, y)

3. uncle(sumit, puneet)

4. mother(anita, x)

References:

1. ML for the Working Programmer (Book Support Page)
2. LISP programming

3. PROLOG programming

1 Introduction to ML

1.1 Why ML?

ML is clean and powerful, and has many traits that language designers consider hallmarks of a good high-level language:

- Uniform reference model: all objects allocated in heap and accessed by reference
- Garbage collected
- Strongly, statically typed, with an excellent type system
- Strongly typed = programs cannot corrupt data by using it as incorrect type.
- Statically typed = types known and checked at compile time.
- "Milner-Damas" type system hits a "sweet spot" combining flexibility (polymorphism) with ease of use (type inference). (Note: sometimes also called "Hindley-Milner" type system.)
- Highly orthogonal design: most features are independent and (where sensible) arbitrarily combinable.
- Exceptions: well-defined error handling.
- Expression-oriented, not statement-oriented
- Sophisticated module system: powerful mechanisms for organizing larger units of code. However, module system's complexity is controversial, and rarely fully exploited in practice.
- Clean syntax
- ML is the "exemplary" statically typed, strict **functional programming** language.

1.2 History of ML

Fig. 1 gives an abbreviated family DAG of the ML family, and a few related languages. Dotted lines indicate significant omitted nodes. The rounded box indicates those variants of the ML family that most people would call ML. A brief history of ML:

1. **1973**: ML invented as part of the University of Edinburgh's LCF project, led by Robin Milner et al., who were conducting research in constructing automated theorem provers. Eventually observed that the "Meta Language" they used for proving theorems was more generally useful as a programming language.
2. **Late 1970's**: polymorphic type inference system completed by Milner and Damas.
3. **Mid-80's**: Standard ML; parameterized module system added by Dave MacQueen; Caml fork at INRIA (France).

4. **1996**: Objective Caml, by Xavier Leroy et al.; adds inheritance to Caml module system.
5. **1997**: ML97 revision of Standard ML.

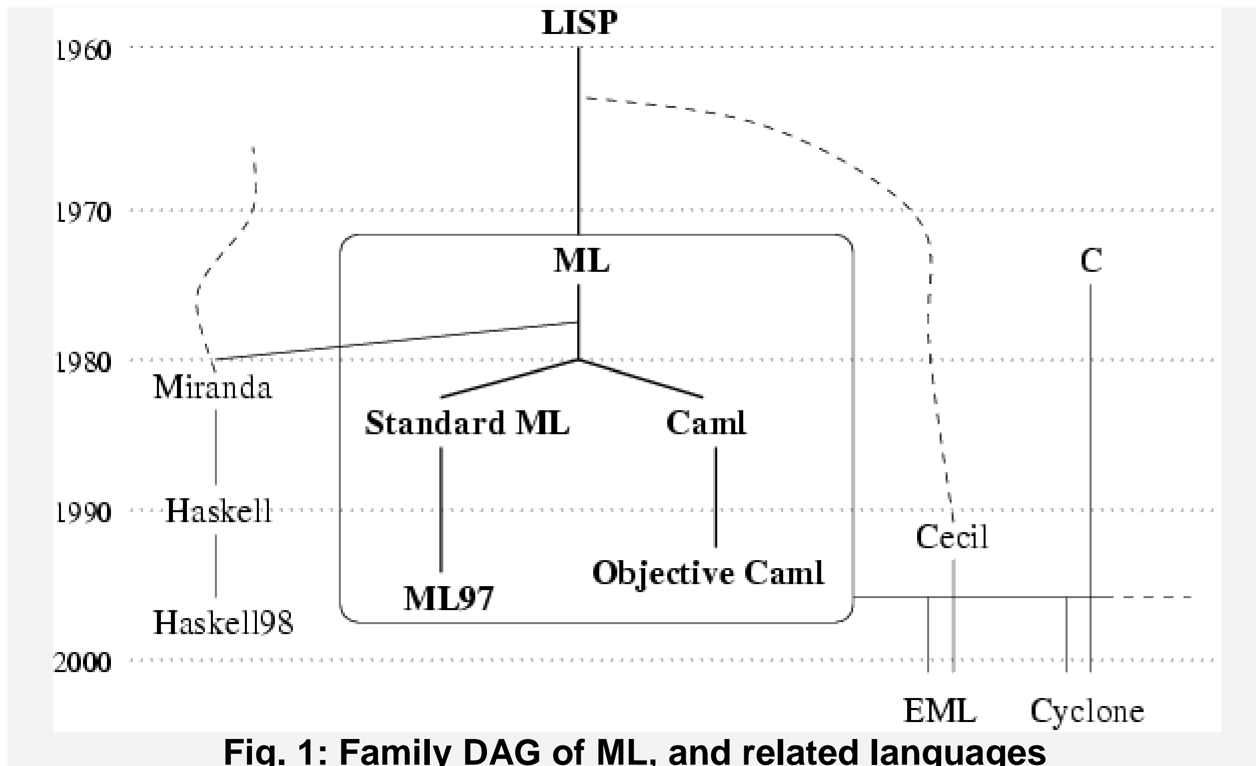


Fig. 1: Family DAG of ML, and related languages

Miranda and Haskell are statically typed **lazy** (as opposed to strict) functional languages, with many similarities to ML (including ML-like polymorphic type systems).

EML and Cyclone are research languages devised by people who are currently at UW, or have been in the past. (We will discuss these languages towards the end of the quarter.) They are marked as descending from the entire ML family because the distinctions between, e.g., SML and O'Caml are not important w.r.t. the way ML influenced these languages.

The general ideas of ML have been highly influential in the research community; if we enumerated all the ML dialects or relatives that researchers have devised over the years, the dotted line to the lower right of the figure would probably have hundreds of descendants.

The interactive ML interpreter

For this class, we'll use the SML/NJ implementation of ML97. Like most ML implementations, SML/NJ provides a **read-eval-print loop** ("repl"), so named because the interpreter repeatedly performs the following:

1. **read**s an expression or declaration from standard input,
2. **eval**uates the expression/declaration, and
3. **print**s the value of expressions, or perhaps the type and initial value of declarations.

The primary advantage of programming in a repl is **immediate feedback**. The read-eval-print cycle is *much* faster than the edit-compile-run cycle in a typical compiled programming environment. You can quickly and easily experiment with different snippets of code. If a function doesn't work, you can try out a different version in a second or two, and re-run your program. This makes interactive repls ideal for "exploratory" programming.

(Often in the course of my teaching, a student has presented me with a code snippet and asked: "What happens if I write X? Or XY? Or, how about XYZ?" Of course, the best way to find out is simply to write X, Y, and Z, and then run the various combinations. But in a compiled environment, you have to create a new file, and repeatedly compile each different version of the program. In a repl, it's easy to quickly experiment interactively with all these variations.)

Of course, typing long chunks of code repeatedly can be tedious, so the repl allows you to load source from a file with the `use` function, which takes a `string`filename and loads the contents of the named file as though it were typed into the interpreter directly. (You can also use Unix pipes, or programming environments like Emacs sml-mode, to send code into the interpreter.)

1.3 Expressions, values, and bindings

All programming languages allow users to manipulate data, and all useful languages provide two kinds of data:

1. **Atomic data:** simple, "indivisible" types, including booleans, integers, characters, floating point numbers, and (in some languages, ML included) strings.
2. **Compound data:** ways of "building up" larger data structures from simpler ones.

We'll start with atomic data. Here's the result of entering some expressions that evaluate to atomic data into the SML/NJ read-eval-print loop:

```
$ sml
Standard ML of New Jersey, Version 110.0.7 ...
- 3;
val it = 3 : int
- 3.0;
val it = 3.0 : real
```

```
- #"3";
val it = #"3" : char
- "3";
val it = "3" : string
- true;
val it = true : bool
```

The dash is the SML/NJ prompt indicating that it's waiting for you to type in an expression or a declaration. When you type in an expression followed by a semicolon, SML/NJ **parses** the expression, then **evaluates** it to a **value**. Then it prints that value, along with its type. The above values are of type `bool`, `int`,`real`, `char`, and `string` respectively.

There are many operators defined over atomic types, including most of the ones you'd expect. See Ullman sections 2.1-2.2 and ch. 9.1 for information about these. Minor surprises:

1. `~` is the arithmetic negation operator, and it is distinct from the subtraction operator.

There is no equality defined directly over reals.

2. The short-circuiting boolean "and" and "or" operators are named `andalso` and `orelse` respectively.

Technical note: Values are expressions that are "done evaluating". Therefore, `3` is a value, whereas `3 + 4` is not a value, because this expression can evaluate one more step, to `7`.

Technical note 2: ML also has an odd atomic type called `unit`. `unit` has only one value, which is written `()` (empty parens):

```
- ();
val it = () : unit
```

`unit` plays a role similar to (but not identical to) that of `void` in other languages --- for example, functions that don't have a meaningful return value will have return type `unit`. The difference is that `()` is a real value --- one that can be bound to names, passed to functions, etc., just like any other value. We'll discuss the relative merits of `unit` vs. `void` more when we discuss functions.

1.3.1 `val` bindings

But what is this `val it = 3` business? In order to explain this, we must first examine **bindings**, which resemble what other languages call "variables". Bindings are **declarations**; the `val` declaration **binds** a value to a name. A bound name can then be used later to refer to the value that was bound to it:

```
- val x = 3;
val x = 3 : int
- x;
val it = 3 : int
- x + 3;
val it = 7 : int
```

Aha, now we can guess what `it` is...

```
- it;
val it = 3 : int
```

When you do not bind an expression to a name at the top-level interpreter prompt, it gets bound to the name `it` by default. This is *not* a feature of ML per se; it's just a helpful feature of the SML/NJ repl. If you want to prevent this, you can bind the value to the **wildcard**, _ (single underscore):

```
- val _ - 4;
- it;
val it = 3 : int
```

Notice that the interpreter does not print the `val it = ...` after the wildcard binding, and that `it` is unchanged afterwards. The wildcard _ is *not* a variable name; it's a placeholder that means, "evaluate this as if you were binding it to a name, but instead throw it away". We'll revisit wildcards in more depth when we discuss pattern matching.

Name bindings resemble variable declarations in a language like C or Java, with several important differences:

1. Bindings must *always* be bound to a value in the declaration.
2. Bindings are *always* `by reference`; that is, the declaration `val x = 3;` *refers* to a value 3 in the heap. Another way of saying this is that all values are (conceptually) always accessed by a pointer.

3. Bindings are **immutable**: you cannot alter what a binding points to --- which is one reason why they must always be bound initially. (Bindings are therefore somewhat like `final` variables in Java.)

But wait --- the last bullet may appear to be a lie, because look:

```
- val y = 5;
val y = 5 : int;
- val y = 6;
val y = 6 : int;
```

What's going on? Is the `y` binding getting modified? Well, actually, no --- the second declaration is **shadowing** the earlier declaration.

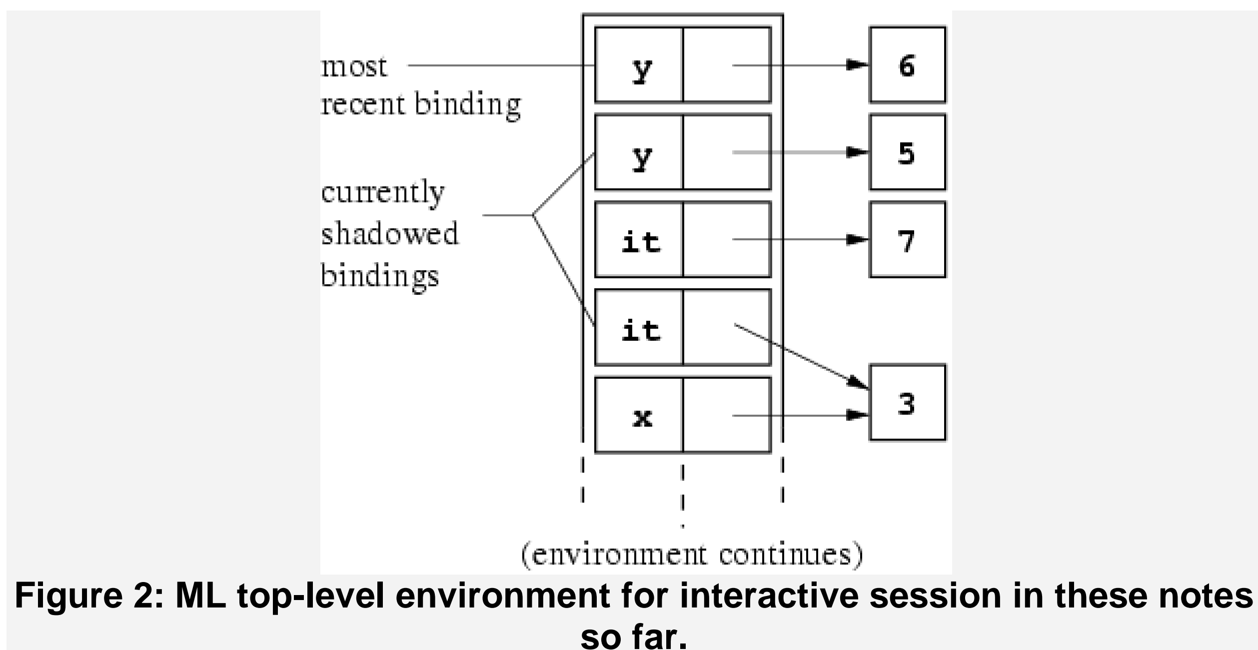


Figure 2: ML top-level environment for interactive session in these notes so far.

Bindings in ML live in **environments**, and the "top-level" environment can be visualized conceptually as an ever-growing stack of bindings. Fig. 2 shows a diagram of the top-level environment resulting from the interactive ML session so far. There are several interesting things to note about this picture.

First, the second `y` and the second `it` binding are shadowed by later bindings: names in a given scope always refer to the *most recent* binding with a matching name; this binding hides any earlier bindings with the same name.

This may seem like it doesn't matter, but only because we've so far only been dealing with the top-level environment. The top-level environment corresponds, roughly, to the "global" scope in

C-like languages. Bindings at top-level are available anywhere that they are not shadowed by some other binding. We'll discuss other environments shortly.

Second, `x` and the shadowed `it` share a pointer to the same `3` value. When a binding is assigned a value, conceptually the *pointer* to that value is copied to the new binding. All values in ML are implicitly by-reference.

Third, this picture only shows the logical picture of data in memory. The implementation may optimize how it represents values in various ways, provided the behavior is indistinguishable from the behavior in this picture. For example, it can discard unused or shadowed bindings, if it can prove that those bindings can never be accessed again. It may also have a special, more efficient representation for pointers-to-integers --- such as the integers themselves. (It is a useful thought exercise to consider why this representation optimization is safe. Remember that most ML values, including integers, are immutable.)

1.4 Aside: What about assignment?

OK, making a new `val` doesn't modify bindings; what about assignment? Suppose a Java programmer forgets for a moment that this is ML, and tries to assign a different value to `y` using `=`:

```
- y = 10;
val it = false : bool
- y;
val it = 3 : int
```

What's going on? Well, for one thing, `=` does *not* mean assignment in ML. Actually, you cannot perform assignment on ML bindings at all --- as previously noted, they are immutable. The *expression* `y = 10` is a comparison, which evalutes to the boolean value `false`. This is why SML/NJ prints `val it = false`, and why `y` is unchanged.

Computation in ML, as in all functional languages, proceeds primarily by *evaluating expressions*. Assignment and with other "side effects" of evaluation play a much smaller role in functional languages than in imperative languages. Code without side effects is said to be **purely functional**, or simply **pure**.

Most of the code we write in this class will be pure. One of the important lessons of functional programming is that *side effects are rarely necessary*. In fact, some languages, such as Haskell, are completely pure (side-effect free). Functional programming advocates claim that code that extensively employs side effects tends to be confusing and harder to reason about (both

automatically and manually) than pure code. When you see a function call `f(x)`, and you know that `f` is a pure function, then you don't have to worry about "hidden" consequences --- the only thing the call does is produce its return value. If `f` has side effects, then you must remember what those side effects are, and what order they happen relative to other side effects, etc.

You can apply this lesson even in non-functional languages: for example, in Java, make as many fields and variables `final` as you can.

1.5 A few words on type inference

If you're used to languages like Java, ML's `val` declarations should look slightly odd to you. In Java, you might write:

```
int a = 5;
float b = 5.0;
char c = '5';
String d = "5";
```

Notice that the syntax of declarations requires that the programmer always explicitly specify the type. ML's syntax doesn't require this, because ML has a **type inference** system. Generally, ML will determine the types of names and values based on how you use them. You only need to declare the types of names explicitly in certain cases when the type inference algorithm doesn't have enough information to do it automatically. To write down a value's type explicitly is to**ascribe** the type to the value; in ML, the syntax for ascription is *`expr`*`:`*`type`* or *`name`*`:`*`type`*, e.g.:

```
- 5:int;
val it = 5 : int
- val x:int = 5;
val x = 5 : int
- val x = 5:int;
val x = 5 : int
```

Notice that you may ascribe the type after either the name or the initializing expression. Actually, type ascriptions can syntactically appear after (nearly) *any*value or declared name. ML's type inference algorithm "propagates" the ascribed type to other positions in the code that must have the same type.

For simple values like the ones we've seen so far, ascription is never necessary, but we will eventually see examples where types must be explicitly `ascribed`(i.e., written down).

(Side note: In some cases, ML programmers ascribe types even where it's not necessary --- either for documentation, or to give a value a "more specific" type than the inference algorithm will infer by itself.)

1.6 Incorrect type ascriptions

What if the programmer ascribes an incorrect type?

```
- val z:char = 5;
stdIn:1.1-40.4 Error: pattern and expression
    in val dec don't agree [literal]
  pattern:    char
  expression:    int
  in declaration:
    z : char = 5
```

Short answer: if the ascriptions cause the inference algorithm to assign an invalid type to an expression, then a type error results. We'll discuss this in more detail when we cover type inference and polymorphism.

Built-in compound data types

ML has several families of built-in data types; these include:

Records: fixed-size, heterogeneous collections indexed by name

Tuples: fixed-length, heterogeneous sequences indexed by position

Lists (singly linked): variable-length, homogeneous sequences with O(1) time to prepend a value and O(n) time to access a value in the list.

Vectors: variable-length, homogeneous sequences with O(1) access time for elements (i.e., like arrays)

You should be familiar with these fundamental types from Java, but in ML all these built-in types are immutable. If you want to "alter" one of these compound values, you must create a *new* value that copies all the components except the field or position you want to change; that field/position should contain the updated value.

ML has special syntactic support for constructing and manipulating its built-in types. This is one of the reasons ML code is much more compact than C or Java code. Each family of built-in types has a **constructor** syntax that constructs a value of appropriate type from that family. (In ML, a constructor

for a type `t` is a function that takes zero or more arguments and constructs a fresh value of `t`.)

1.7 Records

Records resemble structs in C, or method-less objects in Java; they are constructed by writing a list of one or more field assignments `name = value` in between two curly braces `{}`. Here are some examples:

```
- val foo = {x = 3};
val foo = {x=3} : {x:int}
- val bar = {x = 3, y = true};
val bar = {x=3,y=true} : {x:int, y:bool}
- val baz = {x = "hi", y = foo};
val baz = {x="hi",y={x=3}} : {x:string, y:{x:int}}
- val boo = {foo = #"h", bar = "i", baz = 123.0};
val boo = {bar="i",baz=123.0,foo=#"h"} : {bar:string, baz:real,
foo:char}
```

As you can see, a record *type* (e.g., `{x:int}`) is written a comma-separted list of one or more field declarations `name:type` in between curly braces. In general, the syntax of types in ML closely mirrors the syntax for constructing values of those types.

Record types are equivalent if they have *exactly* the same field names and types. A record of one type cannot be assigned to a record of a different type:

```
- val aPoint:{x:int, y:int} = {x = 1.0, y = 2.2};
stdIn:1.1-50.20 Error: pattern and expression
    in val dec don't agree [tycon mismatch]
  pattern:    {x:int, y:int}
  expression:    {x:real, y:real}
  in declaration:
    aPoint : {x:int, y:int} = {x=1.0,y=2.2}

- val simpleRecord:{x:int} = {x = 1, y = 2};
stdIn:55.1-55.42 Error: pattern and expression
    in val dec don't agree [tycon mismatch]
  pattern:    {x:int}
  expression:    {x:int, y:int}
  in declaration:
    simpleRecord : {x:int} = {x=1,y=2}
```

Notice that, unlike objects in a language like Java, a record value cannot be "implicitly promoted" to a record with fewer fields. In other words, ML does not have**subtype polymorphism**.

Fields of a record value are accessed using the special function *#fieldName* applied to *recordValue*:

```
- val r = {x=1, y=2};
val r = {x=1,y=2} : {x:int, y:int}
- #x(r);
val it = 1 : int
```

Side note: What happens if you put zero fields in a record?

```
- {};
val it = () : unit
```

Oops. That doesn't look like a record type --- that's `unit`. In my opinion, this is a bug in ML. However, see below on the empty tuple.

1.8 Tuples

Tuples work a lot like records, except that the fields have an explicit *order*; and instead of using field names, you use *positions* to access the members.

Tuples are constructed simply by enclosing a comma-separated list of two or more values in round parentheses `()`:

```
- (1, 2);
val it = (1,2) : int * int
- ("foo", 25, #"b", false);
val it = ("foo",25,#"b",false) : string * int * char * bool
```

As you can see, tuple types are written as a `*`-separated sequence of types: *type1* * *type2* * ... * *typeN*.

The Kth element of a N-tuple can be accessed by the special accessor function `#K`, as follows:

```
- val x = (54, "hello");
val x = (54,"hello") : int * string
- val firstX = #1(x);
val firstX = 54 : int
- val secondX = #2(x);
val secondX = "hello" : string
```

Side note: What happens if you put one element in parens? Zero?

```
- (1);
val it = 1 : int
- ();
val it = () : unit
```

In my opinion, unlike the empty record case, these make sense. As in other languages, parentheses group terms that should be evaluated before other terms. Rather than constructing a 1-tuple, which is useless, `(expr)` evaluates *expr* before any surrounding expressions and returns it. Also, viewing `unit` as a "zero-tuple" makes more sense to me than viewing empty records as `unit`, though I can't justify this opinion with anything other than my arbitrary taste.

1.9 Lists

Linked lists are the bread and butter of functional programming. (Perhaps recursive, higher-order functions are the knife and fingers.) ML lists are `homogeneous`; that is, all elements must have the same type. The type of a list of elements of type `t` is written "**`t list`**", e.g. `int list` or `string list`. For any type `t`, a `t list` has two constructors:

1. `nil`, the empty list (also written `[]`)
2. `::` (pronounced "cons", terminology borrowed from Lisp), which is an **infix** operator that constructs a single list cell from its left and right arguments. The left argument must be of some type `t`, and the right argument must be of some type `t list`. Intuitively, this should be familiar; in a Java-like language, a node in a singly linked list whose elements have type `T` would usually be defined as follows:

```
3.  class TListNode {
4.      T value;
5.      TListNode next;
6.  }
```

Lists may also be constructed from a comma-separated list of values inside square brackets `[]`. This is **syntactic sugar** for a sequence of conses; and, in fact, when you type a list of conses at the repl, SML/NJ will answer using this sugared syntax.

```
- val x = 1::nil;
val x = [1] : int list
- val y = 1::2::3::nil;
val y = [1,2,3] : int list
- val z = 4::x;
val z = [4,1] : int list
```

A picture of the data structres in memory that result from the above three declarations is shown in Fig. 3.

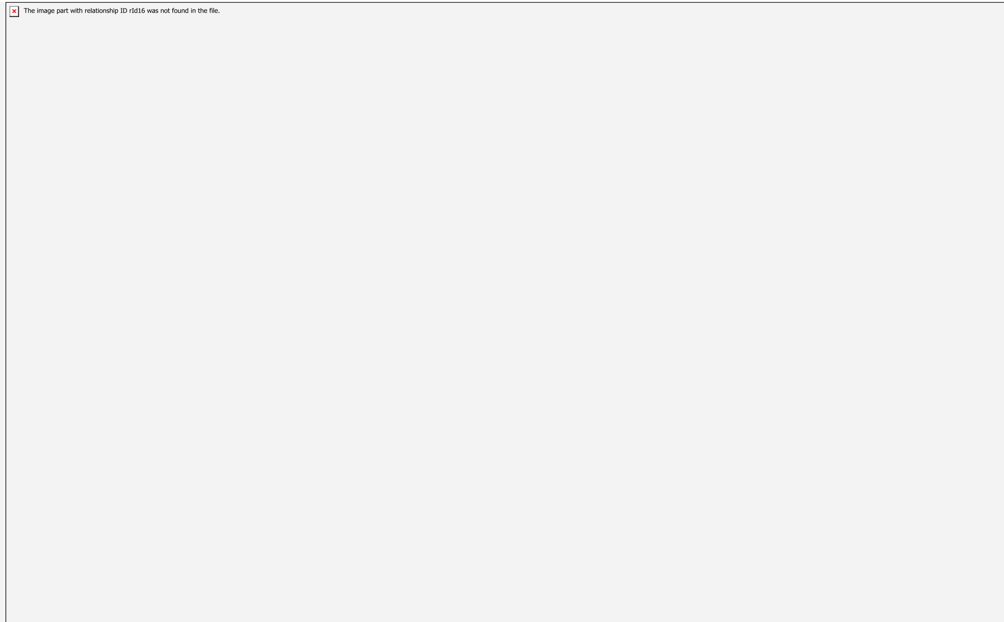


Figure 3: ML top-level environment and data structures in heap resulting from list construction.

Note the following:

1. Lists have finite length, so the last element must always be `nil`.
2. The value bound to `z` is well-typed because the `4` is an `int`, and `x` is an `int list`.
3. The list constructed for `z` uses the list value bound to `x` directly as its "tail". This is safe because lists are immutable.

The first element of a list can be obtained using the function `hd` ("head"), and the rest of a list can be obtained using `tl` ("tail"). Note that, in functional programming terminology, the tail is the *entire* rest of the list after the head, not the last element (think tadpoles, not dogs). Calling `hd` or `tl` on an empty list results in a runtime error (exception).

```
- hd([1,2,3]);
val it = 1 : int
- hd(tl([1,2,3]));
val it = 2 : int
- hd(tl(1::nil));

uncaught exception Empty
  raised at: boot/list.sml:36.38-36.43
```

Q: What is the type of a bare `nil`?

```
- nil;
val it = [] : 'a list
```

What is this `'a` business? In ML, a type whose name begins with a single quote character is a **type variable** which means, roughly, "any type can be substituted here". Types with type variables are called **polymorphic types**. `nil` is actually a **polymorphic value**, i.e. it has polymorphic type; this must be so, because lists of all types share `nil` as the terminating value.

The polymorphism in ML's type system is actually one of its best features. We will describe this in more detail as the quarter goes on; for now, we'll work mostly with lists with some concrete element type.

Uniform reference data model

As depicted in the figures in the previous section, *all* ML values are accessed by *reference*, a.k.a. by pointer. When a value is bound to a name or stored in another data structure, the *pointer* to that value is copied to the appropriate location, not the value itself.

Uniformly accessing variables by reference greatly simplifies program understanding. In languages where values can be "inline" rather than by-reference, there are complex and confusing rules for how and when values are implicitly copied, and what happens when these implicit copies occur.

(If you're familiar with C++, consider the uses of copy constructors, or what happens when you copy a value of type T to a stack-allocated value belonging to one of T's superclasses.)

All values are first-class citizens

All ML's data values are **first-class** citizens, meaning that all values have "equal rights": they can all be passed to functions, returned from functions, bound to names, stored as elements of other values, etc.

One consequence is that in ML, as in most reasonable languages, compound types can be nested arbitrarily. You can have lists of tuples, tuples of lists, or records of lists of tuples of records of tuples, etc., because a compound type can be used anywhere an atomic type can be used. This is an example of ML's high degree of orthogonality:

```
- val a = [{x=1,y=2},{x=3,y=4}];
val val = [{x=1,y=2},{x=3,y=4}] : {x:int, y:int} list
- val b = ("hello", [#"w", #"o", #"r", #"l", #"d"], #"!");
val b = ("hello",[#"w",#"o",#"r",#"l",#"d"],#"!")
    : string * char list * char
- val c = {name=("Keunwoo", "Lee"),
=           classes=["341","590dg","590l"],
=           age=26};
val c =
{age=26,classes=["341","590dg","590l"],name=("Keunwoo","Lee")}
     : {age:int, classes:string list, name:string * string}
```

Exercise: try writing code in Java, or your favorite other programming language, that constructs objects that are roughly equivalent to the above three values. How many lines does it take?

Let-expressions and nested environments

In the above, we alluded to the fact that the top-level environment was not the only environment. **Let expressions** are one way to introduce *local* environments, which produce names that are visible only in a local scope.

Let expressions have the form `let` *`decls`* `in` *`expr`* `end`, where *`decls`* is a semicolon-separated sequence of declarations and *expr* is some expression that may optionally use the names bound in *decls*. Names bound in a let-expression are only visible to later bindings in the same let-expression, and inside the body expression. Outside the scope of the let-expression, the bindings are no longer visible. For example:

```
- let val x = 5 in x + x end;
val it = 10 : int
- let
=   val localA = "hello";
=   val localB = "+++++++";
=   val localB = ", ";
=   val localC = localB ^ "world"
= in
=   localA ^ localC                          (* XXX *)
= end;
val it = "hello, world" : string
- localA;
stdIn:88.1-88.9 Error: unbound variable or constructor: localA
- let
=   val earlierBinding = laterBinding + 1;
=   val laterBinding = 5
= in
=   earlierBinding + laterBinding
= end;
stdIn:120.24-120.36 Error: unbound variable or constructor:
laterBinding
```

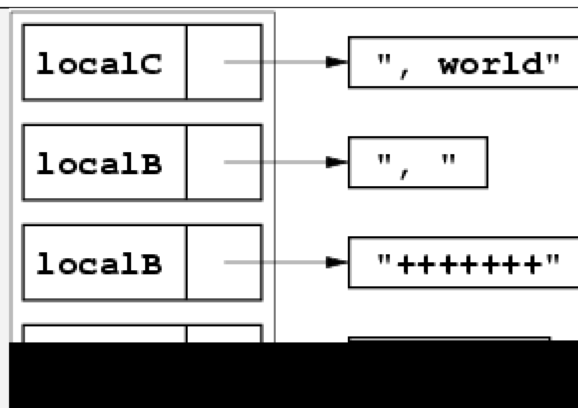


Figure 4: Contents of local let-environment at pointxxx.

Order of bindings matters:

1. Later bindings are not visible to earlier ones
2. Later bindings shadow earlier bindings with the same name.

These are really the same rules that apply in the top-level environment. All environments in ML work the same way. This is an example of ML's high degree of *regularity*: there are no special rules for top-level versus local environments.

A diagram of the local environment at the point marked `XXX` is given in Fig. 4.

Again: all values are first-class

All expressions are first-class, and let expressions are expressions. Therefore, let expressions can be nested, and more generally may appear anywhere other expressions may appear:

```
- val longLetExpr =
=   let
=     val aString = let val x = "hi, "; val y = "there" in x ^ y end;
=     val anInt = 17
=   in
=     (anInt, let val period = "." in aString ^ period end)
=   end;
val longLetExpr = (17,"hi, there.") : int * string
```

The image part with relationship ID rId19 was not found in the file.

This work is licensed under a Creative Commons License. Rights are held by the University of Washington, Department of Computer Science and Engineering (see RDF in XML source).

: Functions and patterns

Function basics

Function syntax and types

A function in ML is written as follows:

```
fn arg => returnValue
```

For example, the following function returns an integer that is one greater than its argument:

```
- fn x => x + 1;
val it = fn : int -> int
```

1. SML/NJ gives the above function the type `int -> int`. In general, the type of a function is written *`argType -> returnType`*, which is reminiscent of the way mathematicians describe the domain and codomain of functions in math. In the academic programming languages literature, function types are sometimes be called "arrow types".
2. Unlike atomic or simple compound types, function *values* are not echoed by SML/NJ; instead, SML/NJ simply writes `fn` as a placeholder for the value.

The return value is the entire body of the function. Since ML proceeds by evaluaton of expressions, this is a natural way to define functions: the body is an expression that gets evaluated. This design differs from imperative languages, where a function body is usually a block of code to be executed.

Ascribed argument or return types

Function arguments (like all names in binding positions) and function bodies (like all expressions), can optionally be ascribed types:

```
- fn x:int => x + 1;          (* ascribing to the argument *)
val it = fn : int -> int
- fn x => (x + 1):int;        (* ascribing to the body *)
val it = fn : int -> int
```

This will sometimes be necessary when the body does not provide enough information to determine the exact type of an argument or return value. For example:

```
- fn stringPair => #1(stringPair) ^ "!";
stdIn:32.1-32.38 Error: unresolved flex record
   (can't tell what fields there are besides #1)
```

We know that this function's argument is a tuple --- in fact, the programmer probably intends a pair, . However, we can't tell how many elements the tuple has. ML needs to know this in order to assign a type to the function, so we must ascribe a type to the argument:

```
- fn stringPair:(string * string) => #1(stringPair) ^ "!";
val it = fn : string * string -> string
```

In order to construct examples where ascribing to the return value is necessary, we must wait until we see more of the ML type system.

Naming functions

Recall that all values in ML are *first-class*. Functions are values. All values can be bound to names. Therefore, functions can be bound to names, which evaluate to their bound value exactly the same way that any other name evaluates:

```
- val addOne = fn x => x + 1;
val addOne = fn : int -> int
- addOne;
val it = fn : int -> int
```

Since it is so common to bind function values to names, ML has syntactic sugar for function declarations:

```
- fun addOne x = x + 1;
val addOne = fn : int -> int
```

Notice that SML/NJ echoes the desugared form of the `val` declaration. The two *syntactic* forms are *semantically* equivalent in every way.

ML's treatment of functions and naming contrasts strongly with languages like C (where functions may only occur at top level, and must always be named) or Java (where methods may not be defined independently of classes, and methods only occur as "values" in the sense that object values can have methods).

Function application

Functions are applied to arguments by writing the argument after the function expression, and parenthesis around the argument are strictly optional. All of the following apply the function value bound to `addOne` to the integer 3:

```
- addOne 3;
val it = 4 : int
- addOne(3);
val it = 4 : int
- (addOne 3);
val it = 4 : int
- (addOne)3;
val it = 4 : int
```

In ML programming, we usually include the parenthesis only where needed to enforce order of evaluation.

Unlike some other languages, functions do not need to be bound to a name before they are applied; you may use the `fn` expression (an `anonymous function`) directly:

```
(fn x => x + 1) 3;
val it = 4 : int
```

This is yet another instance of ML's regularity. Functions are simply values. Evaluating a function application simply proceeds by three steps:

Evaluate the function value.

Evaluating the argument value.

Apply the function to the argument.

It doesn't matter whether step 1 is a variable expression (for looking up a function value bound to a name) or an anonymous function expression. Both expressions evaluate to function values. More generally, it doesn't matter *where* the function expression comes from --- it may be obtained from the return value of a function, or by accessing a component of a data structure, or any of the other ways that a value may obtained.

Typechecking function applications

Function calls are typechecked in the obvious way: the actual argument must match the formal argument type. When it does not, you get an error:

```
- addOne "hello";
```

```
stdIn:22.1-22.15 Error: operator and operand
    don't agree [tycon mismatch]
  operator domain: int
  operand:         string
  in expression:
    addOne "hello"
```

Precedence of function application

Function application has quite high precedence, which can sometimes be confusing. Consider the folowing code fragment:

```
fun italic s = "<i>" ^ s ^ "</i>";
- val italic = fn : string -> string
fun italicGreeting name = italic "Hello, " ^ name;
- val italicGreeting = fn : string -> string
italicGreeting "Keunwoo";
val it = "<i>Hello, </i>Keunwoo" : string
```

The `italic` function surrounds the input string in the HTML markup for italic text. You might think that the string concatenation expression `"Hello, " ^ name` gets evaluated, and the result passed to `italic`, but function application has higher precedence than string concatenation (or, indeed, most other operators).

Thought question: Suppose you're typing a list in the square-bracket syntax and you accidentally omit a comma:

```
[1, 2 3, 4];
```

What happens? Why?

Functions with no meaningful return value or arguments

Sometimes side effects are unavoidable. For now, we will acknowledge one limited use for side effects: input and output. The standard library function `print`must have a side effect: printing to standard output changes the world. But what should a function like this return? It might return a status code, but often such functions have no natural return value.

Languages like Pascal solve this problem by dividing the universe of control abstractions into two kinds: functions, which return values, and procedures, which do not. Languages like C solve this problem by having `void` functions ---

functions that return nothing. ML uses an approach similar, but not identical to, the latter: it uses the `unit` type, which has one value, written `()`:

```
- print;
val it = fn : string -> unit
- print "hi\n";
hi
- val it = () : unit
```

Functions that naturally take no parameters can accept `unit`:

```
- val printHi = fn () => print "hi\n";
val printHi = fn : unit -> unit
- printHi()
hi
val it = () : unit
```

Because `unit` is written `()`, this is a sort of "visual pun" on zero-argument function calls in other languages.

Control: Branching, sequencing, and patterns

Imperative languages express branching through conditional *statements*; functional languages like ML, being expression-oriented, express branching primarily through conditional *expressions*.

`if` expressions

`if` conditional expressions in ML have the following syntax:

```
if booleanExpr then expr1 else expr2
```

These have the "obvious" semantics (similar to the `:?` operator in C):

1. First, *`booleanExpr`* is evaluated.

If the test expression is true, then the first expression is evaluated and is returned.

2. If the test expression is false, then the *`expr2`* is evaluated, and is returned.

Here's a simple conditional expression:

```
- if 1 > 2 then
```

Like all expressions, `if` expressions are first-class. The result of an `if` expression can be used anywhere any other expression can be used. For example:

```
[1, 2, if x = 4 then 5 else 6 ];
if x = 4 then
  (if x > 10 then y else z,
   if x > 20 then a else b)
else
  (17, 18)
```

Be careful --- the first branch in the outermost `if` is a tuple (comma-separated value in parens), not a sequence of two expressions.

Note that conditional expressions do not evaluate the un-taken branch --- this is why `if` cannot be naively implemented as an ordinary function call, which evaluates all its arguments prior to invoking the function.

(Actually, we can implement a proper `if` function using function parameters, but as we shall see this would be rather more verbose to use given ML's anonymous function syntax.)

Typechecking conditional expressions

A conditional expression may return either of its branches. What should be the type of the following expression?

```
if p then 27 else "hello"
```

In the ML type system, this expression has no sensible type --- depending on the value of `p`, either branch may be returned, so neither `int` nor `string` describes the result value adequately.

In ML, branches of a conditional expression must have exactly the same type.

Sequencing

Expression sequences in ML are written as one or more *semicolon*-separated sequence of expressions in round parenthesis. Sequences

Expression sequences have the following semantics:

Evaluate each expression in left-to-right order.

1. Return the *last* expression evaluated as the value of the whole expression

All results besides the last expression are discarded. Expression sequences are primarily useful for side-effecting expressions like `print` calls (in this class, you will primarily use them for inserting debugging statements):

```
- val x = (print "hi\n"; 3)
hi
val x = 3 : int
```

Thought question: What should the type checking rules for expression sequences be, if any? Need there be any relationship among the types of expressions in the sequence, as there are with `if`? Why or why not?

Pattern-matching and `case`

The `if` expression essentially provides a way to **match** a boolean value against true or false. Another way to write this in ML is as follows:

```
case booleanExpr of
    true => expr1
  | false => expr2
```

The `case` construct takes a value and attempts to match it against one or more `patterns` --- in this case, the two boolean **constant patterns**, `true` and `false`. If a pattern matches, then its corresponding expression is evaluated and returned as the value of the entire case expression. Matching is **first-match**: the pattens are tried in left-to-right order, and the *first* matching pattern's expression is evaluated and returned.

As with `if` expressions, the body expressions of all branches of a `case` statement must have the same type. The reason for this restriction is the same as with `if`.

`case` would be overkill if we only had boolean values; but `case` can be used with any type, not just boolean. Let's try integers:

```
- val x = 3;
val x = 3 : int
- case x of
=     1 => "one"
=   | 2 => "two"
=   | 3 => "three";
stdIn:40.1-43.15 Warning: match nonexhaustive
          1 => ...
          2 => ...
          3 => ...

val it = "three" : string
```

We got the answer we expected, but why the warning? The answer is that the cases are not **exhaustive**, which means that the cases we gave do not cover the entire possible range of the data type being tested --- in this case, `int`. We have not enumerated all the possible integer values.

ML *does* have a well-defined behavior in the case we apply the case to a bad value --- it raises a match failure exception:

```
- case 25 of
    1 => "one"
  | 2 => "two";
stdIn:17.1-19.15 Warning: match nonexhaustive
          1 => ...
          2 => ...

uncaught exception nonexhaustive match failure
  raised at: stdIn:19.10
```

But ML raises a warning because it's generally good programming style to cover all the cases. If you're a Java programmer, you might conclude that we need a way to provide a default case. Indeed, that is correct, but ML actually contains a better, more generally useful mechanism that solves this problem: it simply allows more general patterns, some of which can match more than one value.

The first of these is **wildcard** patterns, which match *any* value:

```
- case x of
=   1 => "one"
= | _ => "anything else";
```

```
val it = "anything else" : string
```

What if we reversed the order of cases?

```
- case x of
=    _ => "anything else"
= | 1 => "one";
stdIn:53.1-55.13 Error: match redundant
          _ => ...
    -->   1 => ...
```

Oops. What's going on? Recall that ML is first-match --- the second case can never be reached, because the wildcard pattern will always match. More generally, ML will raise an error if you try to define any pattern case after some other case which subsumes it.

The second interesting type of non-constant pattern is **variable patterns**, which not only match any value but bind that value to a variable name for later use:

```
- case x of
=      1 => "one"
=    | y => "x is: " ^ Int.toString y;
val it = "x is: 3" : string
```

This may seem a bit silly --- aren't we just naming a value that we've either constructed, or already have a name for? --- but variable patterns really come a live when we add the third kind of pattern, **constructor patterns**.

Constructor patterns

When we discussed ML's built-in data types, we talked about **constructors**, which were functions that produced values of a given type. ML allows constructors to appear in patterns. Wherever subexpressions would go in a constructor expression, sub*patterns* appear in the constructor *pattern*. For example:

```
- val aPair = (1, 2);
val aPair = (1,2) : int * int
- case aPair of
    (0, 0)   => "origin"
  | (1, _)   => "first is one"
  | (2, snd) => "first is two; second is " ^ Int.toString snd
  | (a, b)   =>
      "other value: (" ^ Int.toString a ^ "," ^ Int.toString b;
val it = "first is one" : string
```

The value is a pair (2-tuple) of `int`s, so all pattern cases must match pairs of `int`s. The first case has two constant patterns for the two tuple members, and therefore matches only the value `(0, 0)`. The second case has a wildcard as its second value, and therefore matches any pair with `1` as its first element. The third pattern matches any pair with `2` as its first element, but then saves and uses second element in the expression body. The last pattern matches any 2-tuple, binding both elements to names, and uses them in the expression body.

Any of the constructors we have seen may appear in a pattern. Here are some case expressions that use various constructors we've seen:

```
case foo of
    {x=0, y=0} => "origin"
  | {x=_, y=y} => "non-origin at y-coord " ^ Int.toString y;

case bar of () => "unit has only one value."

case aStringList of
    nil    => "empty"
  | hd::tl => "first list element is: " ^ hd;
```

The last of these --- matching against the `nil` case of a list and then against the cons case --- will shortly become quite familiar to you, because essentially all functions that operate over lists do this.

Patterns, patterns, everywhere

Patterns are not restricted to use in `case` statements. They may appear wherever any name binding may appear, including `val` declarations and function arguments. In fact, *all name binding* in the ML core language is really pattern matching. Here is a function that concatentates the elements of a string pair:

```
- fn (x, y) => x ^ y;
val it = fn : string * string -> string
```

Note the use of a tuple pattern in the argument. This looks almost like a function definition in C or Java, where the parameters are separated by commas, but it's completely different. For example, the argument patterns can be a record rather than a tuple, or it can contain nested subpatterns with structure rather than simply names:

```
- fn {first=firstName, last=lastName} =>
```

```
    firstName ^ " " ^ lastName;
val it = fn : {first:string, last:string} -> string

- fn {x=_:int, y=(a:int, b:int), z=z:string} =>
    Int.toString a ^ z ^ Int.toString b
val it = fn : {x:'a, y:int * int, z:string} -> string
```

For the last of the above, note the use of type ascriptions inside the pattern, and the nested tuple subpattern.

Functions use `case` at top-level so often that ML also has a special syntactic sugar which allows you to define a function in multiple cases. The following two functions are exactly equivalent:

```
- fun emptyTest aList =
    case aList of
      nil     => "empty!"
    | (x::xs) = "not empty; first elem: " ^ x;
val emptyTest = fn : string list -> string

- fun emptyTest nil     = "empty!"
    | emptyTest (x::xs) = "not empty; first elem: " ^ x;
val emptyTest = fn : string list -> string
```

Here is how we use a `val` binding to take apart the elements of a record:

```
- aPoint = {x=1, y=2};
val aPoint = {x=1,y=2} : {x:int, y:int}
- val {x=x, y=y} = aPoint;
val x = 1 : int
val y = 2 : int
```

Notice that you can bind more than one name at a time. For records, it is so common to bind field names to variables of the same name that ML provides a syntactic sugar which allows you to write each field name once, omitting the `=`*`name`*:

```
- val {x,y} = aPoint;
val x = 1 : int
val y = 2 : int
```

Val bindings do not provide a way to handle multiple cases in a pattern, so they fail if there is no match.

How patterns match

This is the complete algorithm, in ML-like pseudocode, for determining whether a value matches a pattern:

```
fun match(value, pattern) =
  case pattern of
    constant => if value equals the constant then true else false
  | wildcard => true
  | variable => bind value to variable name; true
  | constructor =>
      if value has same constructor then
         match subpatterns of pattern with corresponding
            parts of value
         if all parts match then true else false
      else false
```

Notice that this definition is recursive. Speaking of which...

Recursive functions

Functions in ML may be recursive, and must be bound to a name (Thought exercise: why can't ML anonymous functions be recursive?):

```
- fun length nil = 0
=   | length (x::xs) = 1 + length xs;
val length = fn : 'a list -> int
```

Recursive functions, as this example shows, are ideal for handling recursive data structures like lists, trees, etc. Inductive recursive definitions, whether for data or for functions, are defined in cases:

At least one base case, where the recursion "bottoms out"

At least one inductive case, where the recursion continues.

For lists, the base *data* case is `nil`, and the inductive *data* case is cons. The length *function* likewise has two cases, one for the base case and one for the inductive case.

More generally, to write almost any function over a recursive data type, you generally follow a simple formula:

Look at the cases of the data type.

For each data case, write one or more function cases:

For a base case, (usually) compute the appropriate result directly.

For an inductive case,

Compute a partial result for the parts directly available (e.g., the head of the list);

Call recursively on the recursive portion(s) of the data structure (e.g., the tail of the list).

If necessary, combine the result of the direct and recursive computations.

This recursive formula will occur again and again in your functional programming. Learn it well, and it will help you organize your thinking about recursive data structures even in non-functional languages.

(Aside: what's this `'a list` type that ML infers for the `length` function's argument? Well, if you examine the body of `length`, there's actually no indication as to the*element* type of this list. The list could be any type --- and this makes perfect sense, since a function that takes the length of a list never needs to know the type of that list's elements. ML's type system allows this function to be **polymorphic** over different types of lists --- i.e., the same function can be applied to different types. `'a` is a **type variable** --- it stands for "any type". When the function is applied to an argument, the type variable will be instantiated with the type of its argument's element type. We'll discuss type variables and polymorphism in much more detail next week.)

Recursion vs. iteration

In Java, you wouldn't write a recursive length function. You would use a loop:

```
class Node { Object o; Node next; }
...
int length = 0;
for (Node i = List.firstNode; i != null; i = i.next) {
    length++;
}
```

Observe, however, that a loop of this kind requires mutation: the `i = i.next` changes what `i` points to, and the length field must be incremented. In functional programming, you typically use recursion instead of iteration. Functional programming advocates claim recursion is typically clearer and less error-prone:

It is "intuitively obvious" that the ML length function is correct (indeed, it is hard to imagine how to get it wrong), whereas iteration presents many opportunities for error because of the numerous assignments.

The ML length function is "natural" because it parallels directly the inductive definition of the list data type.

On the other hand, naively implemented recursion often has greater overhead than naively implemented iteration:

Time and space overhead for procedure call, stack allocation, and return.

The "natural" inductive definitions of some algorithms are less efficient efficient than iterative definitions.

Tail recursion

The `length` function, as defined above, has one important drawback. It must keep an activation record on the procedure call stack for every recursive call.

But this is not true of all recursive functions; or, of all functions that call another function. In particular, consider the case where a function returns directly the value of another function --- this is called a **tail call**. A very simple example:

```
fun f aList = length aList;
```

In this case, it is clear that once `f` passes control to `length`, then the compiler need not keep the activation record for `length` around (including, e.g., the space for the `aList` parameter), because `f` does nothing after `length` returns. The compiler can *reuse* that space on the call stack for the activation record of the `length` call.

This space-saving optimization is called **tail call elimination**, because the call is at the "tail" of the function. This optimization plays a crucial role in functional language implementation, because of the heavy use of recursion; indeed, most functional languages specify that implementations *must* perform tail call elimination. Here are a couple of tail-recursive functions:

```
fun last nil        = raise Empty
  | last (x::nil)  = x
  | last (_::rest) = last rest;

fun includes (aValue,  nil)     = false
  | includes (aValue, (x::xs)) =
    if aValue = x then
        true
    else
```

```
        includes (aValue, xs)
```

Every case of these functions either "bottoms out" or directly returns the result of a recursive call. Therefore, they are tail recursive.

So what prevents a function from being tail recursive? And is there any way to *make* a function tail recursive when it isn't to begin with? It is instructive to examine ordinary tail calls first. Here is a function that resembles `f`, but is *not* tail call:

```
fun g aList = 1 + length aList;
```

This function's body does not tail call, because the result of the call is not returned directly --- `g` must do more work (namely, adding one to the result) before returning. The compiler must keep the activation record for `g` around while it is waiting for `length` to return.

Well, what if we could "push down" that work into the callee, so that `g` didn't have work remaining? That would be great, but in general the caller has no way to modify what the callee will do. On the other hand, in a recursive function, the callee is the caller...

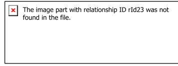
```
fun helper (nil,   lengthSoFar) = lengthSoFar
  | helper (x::xs, lengthSoFar) = helper (xs, lengthSoFar + 1);

fun length aList = helper (aList, 0);
```

How these functions work:

1. `helper` is tail-recursive, and has an extra parameter that keeps track of the "length so far".
2. For `nil`, `helper` returns the length computed so far, because an empty list cannot add more length to the list.
3. For cons, `helper` adds one to `lengthSoFar` and calls itsself on the tail of the list.
4. To complete the function, we add a "driver", `length`, that invokes the helper on its argument with the whole list and a length so far of zero.

This sort of conversion can be performed on any singly recursive function. Simply add a helper function that keeps the "results computed so far" as a

parameter, and invoke it with a suitable initial value. See Ullman 3.5.3 (background in 3.2, 3.3.1) for a discussion of `reverse` using this trick.



This work is licensed under a Creative Commons License. Rights are held by the University of Washington, Department of Computer Science and Engineering (see RDF in XML source).

Functions and patterns, supplementary notes

What is functional programming?

Functional programming is a style that emphasizes:

Expression-oriented programming

Heavy use of functions as values: higher-order and anonymous functions

1. Side-effect-free code; **referential transparency**

Recursion instead of iteration

Secondary characteristics (usually, but not always part of FP) include:

Strong typing

Garbage collection

Emphasis on lists as a data type

Language Construct X in a Nutshell

All language constructs (in a statically typed language) have three parts:

1. **1: Syntax**: What does it look like?

Semantics: What does it mean?; in two parts:

   1. **2: Dynamic semantics**: What does it do at runtime?
   2. **3: Static semantics**: How is it typechecked? This consists of at least two parts:

What subexpressions are legal and illegal in the type system? Alternatively phrased: What constraints does the type system place on well-typed instances of construct X?

For well-typed constructs, what is the resulting type of the construct?

Stating these three properties gives you the essence of the language construct. When language designers design a language, they go through these three steps for each construct --- either consciously or unconsciously.

In these notes, we will do two examples informally, to show this formula in action.

`if/then/else` in a Nutshell

Syntax:: If/then/else expressions have the syntactic form:

```
if expr1 then expr2 else expr3
```

Dynamic semantics: First, `expr1` is evaluated to a value `v`. If `v` has the boolean value `true`, then *expr2* is evaluated and returned. Otherwise, `v` has the boolean value `false`, and *expr3* is evaluated and returned.

Static semantics:

1. *Constraints:* `expr1` must have the type `bool`. It must be possible to unify `expr2` and `expr3` to the same type; call this type `T`.
2. *Result type:* The expression has type `T`

Functions in a Nutshell

There are two constructs related to functions: definition (sometimes called **abstraction** in the programming languages literature) and application (a.k.a. function call).

Function definition

Syntax: Function definitions have the syntactic form

```
fn pattern => returnValue
```

Dynamic semantics: A function definition constructs a **closure** that contains two parts:

The code of the compiled function.

An environment pointer that "captures" variables in the surrounding environment.

Static semantics:

1. *Constraints:* A function must have a well-typed body, assuming the names bound in its argument pattern have the inferred types (and any references to enclosing names have the proper types, etc.).
2. *Result type:* The type of the function is `argType -> returnType`, where *argType* and *returnType* are inferred from the argument pattern and body.

Function application

Syntax: Function applications have the syntactic form

```
expr1 expr2
```

Dynamic semantics: First, *expr1* is evaluated to a value *value1*. Second, *expr2* is evaluated to a value *value2*. Then, *value1*'s function body is evaluated in the environment produced by matching *value2* against the argument pattern.

Static semantics:

1. *Constraints:* A function application is well-typed if *expr2*'s type can be unified with *expr1*'s argument type.
2. *Result type:* The type of the function application is the result type of `expr1`*

* With polymorphic type variables appropriately instantiated. We'll learn about polymorphic types in the next couple of lectures.

Sample exercises

The answers to the following exercises are available here.

Which of the following pattern-matches fail? Which succeed? For successful matches, draw a diagram of the bindings that result, and annotate each name binding with its type. For unsuccessful matches, describe briefly the reason for the failure.

1. `val (a, _) = (("hi", "bye"), fn x => x + 1);`
2. `val (_, b) = (("hi", "bye"), fn x => x + 1);`
3. `val {a=a, b=b} = ({a="hi", b="bye"}, fn x => x + 1)`
4. `val (x:char)::xs = ["a","b","c"];`
5. `val x::y::z = ["a","b","c"];`
6. `val x::y::z = ["a","b"];`
7. `val x::y::z = ["a"];`
8. `val x::y::(z:string list)::zz = ["a", "b", "c"];`
9. `val (a:int->int, b) = (fn x => x + 1, fn x => x ^ "1");`

10. `val (a, b) = (fn x => x + 1, {foo=fn x => x ^ "1", bar=fn x => x * x});`

For each of the following recursive functions, state briefly why it isn't properly tail-recursive, and then write a tail-recursive version.

```
11.   fun sumN 0 = 0
12.     | sumN n = n + sumN (n-1);
```

13.

```
14.   fun factorial 0 = 1
15.     | factorial n = n * factorial (n-1);
```

16.

```
17.   fun joinStrings nil      = ""
18.     | joinStrings (x::xs) = x ^ joinStrings xs;
```

19.

```
20.   fun countDown 0 = [0]
21.     | countDown n = n::(countDown (n-1));
```

22.

```
23.   fun countUp 0 = [0]
24.     | countUp n = countUp (n-1) @ [n];
```

25.

3. Try writing the syntax, dynamic semantics, and static semantics for the following language constructs.*

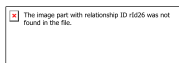
Integer addition. (Actually, this is a function application --- using infix operator syntax --- but pretend it's a primitive.)

List cell cons.

1. `val` binding. (You do not have to describe pattern matching --- assume this has been defined.)
2. `let` expressions. This can be defined two ways --- as a "primitive" construct, from the ground up, or as a syntactic sugar for a series of function applications.** Define it both ways.

* For most of these constructs, the static semantics for full ML uses polymorphic types. For now pretend ML only has monomorphic types like `int`, `string`, or `(int * int * int)`.

** Actually, there are slightly different typechecking requirements between function application and `let`, but the distinctions are beyond the scope of your current knowledge.



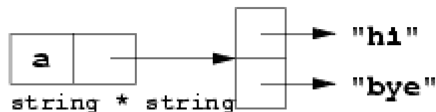
This work is licensed under a Creative Commons License. Rights are held by the University of Washington, Department of Computer Science and Engineering (see RDF in XML source).

: Functions and patterns, solutions to supplementary exercises

Which of the following pattern-matches fail? Which succeed? For successful matches, draw a diagram of the bindings that result, and annotate each name binding with its type. For unsuccessful matches, describe briefly the reason for the failure.

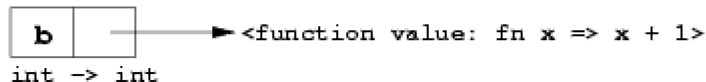
1.
```
val (a, _) = (("hi", "bye"), fn x => x + 1);
```

   Succeeds.



2.
```
val (_, b) = (("hi", "bye"), fn x => x + 1);
```

   Succeeds.



3.
```
val {a=a, b=b} = ({a="hi", b="bye"}, fn x => x + 1)
```

   Fails with a static type error. The pattern is a record with `{a, b}` fields; the value is a tuple whose *first element* is a record with `{a, b}` fields.

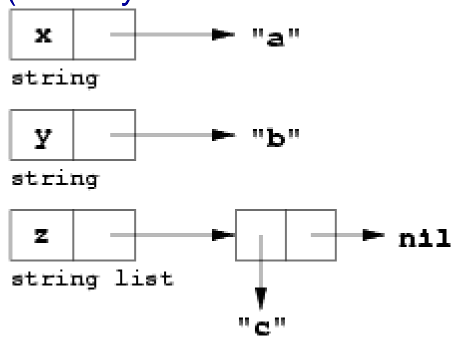
4.
```
val (x:char)::xs = ["a","b","c"];
```

   Fails with a static type error. The pattern is a cons whose head has the ascribed type `char`; therefore, the type of the entire cons pattern is `char list`. The values is a cons of type `string list`.

5.
```
val x::y::z = ["a","b","c"];
```

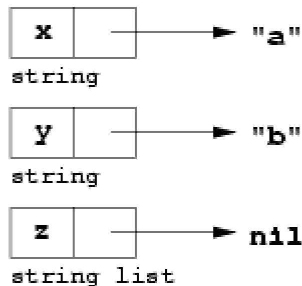
   Succeeds. The pattern is a two-level cons; this can be matched against the list expression, which has at least two cons cells

(actually it has three:`"a"::"b"::"c"::nil`).



6. `val x::y::z = ["a","b"];`

   Succeeds. The pattern has two conses, and the value has two conses.



7. `val x::y::z = ["a"];`

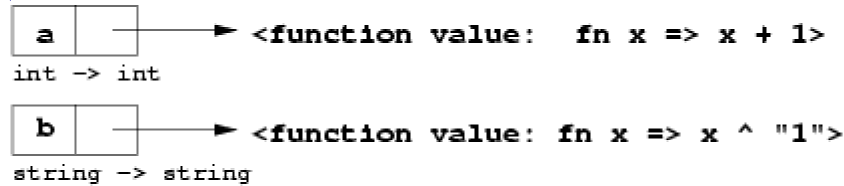
   Fails with a dynamic match error. The left-hand (top-level) cons pattern can match, but the right-hand (inner) cons pattern fails against `nil`. Note that, unlike the previous two failures, this pattern match is statically well-typed --- the failure only occurs when value `nil` is dynamically matched against the cons pattern.

8. `val x::y::(z:string list)::zz = ["a", "b", "c"];`

   Fails with a static type error. Cons associates right-to-left, so `z` appears on the left-hand-side of a cons operator. Therefore, the ascribed type of `z`(`string list` must be the *element type* of the whole list pattern. The entire list pattern must therefore have type `string list list` (list of string lists). The value on the right-hand side is only `string list`. Therefore, the declaration does not type check, and is statically rejected.

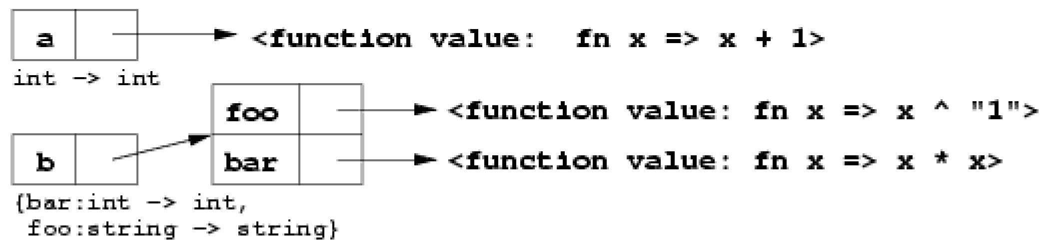
9. `val (a:int->int, b) = (fn x => x + 1, fn x => x ^ "1");`

Succeeds



10. `val (a, b) = (fn x => x + 1, {foo=fn x => x ^ "1", bar=fn x => x * x});`

Succeeds.



For each of the following recursive functions, state briefly why it isn't properly tail-recursive, and then write a tail-recursive version.

```
11.   fun sumN 0 = 0
12.     | sumN n = n + sumN (n-1);
```

```
13.   fun sumN n =
14.       let
15.           fun helper (sum, 0) = sum
16.             | helper (sum, n) = helper (sum+n, n-1)
17.       in
18.           helper (0, n)
19.       end;
```

20.

```
21.   fun factorial 0 = 1
22.     | factorial n = n * factorial (n-1);
```

```
23.   fun factorial n =
24.       let
25.           fun helper (m, 0) = m
26.             | helper (m, n) = helper (m*n, n-1)
27.       in
28.           helper (1, n)
29.       end;
```

30.

```
31.   fun joinStrings nil      = ""
32.     | joinStrings (x::xs) = x ^ joinStrings xs;
```

```
33.   fun joinStrings aList =
34.       let
```

```
35.         fun helper (joined, nil) = joined
36.           | helper (joined, x::xs) = helper (joined ^ x,
      xs)
37.     in
38.         helper ("", aList)
39.     end;
```

40.

```
41.   fun countDown 0 = [0]
42.     | countDown n = n::(countDown (n-1));
```

```
43.   fun countDown n =
44.       let
45.           fun helper (counted, 0) = 0::counted
46.             | helper (counted, n) = helper (n::counted, n-1)
47.
48.           fun reverse (reversed, nil) = reversed
49.             | reverse (reversed, x::xs) = reverse
      (x::reversed, xs);
50.       in
51.           reverse ([], helper ([], n))
52.       end;
```

53.

```
54.   fun countUp 0 = [0]
55.     | countUp n = countUp (n-1) @ [n];
```

```
56.   fun countUp n =
57.       let
58.           fun helper (counted, 0) = 0::counted
59.             | helper (counted, n) = helper (n::counted, n-1)
60.       in
61.           helper ([], n)
62.       end;
```

63.

4. Try writing the syntax, dynamic semantics, and static semantics for the following language constructs.*

Integer addition. (Actually, this is a function application --- using infix operator syntax --- but pretend it's a primitive.)

Syntax: *expr1 + expr2*.

Dynamic semantics: Evaluate *expr1* to a value *v1*. Evaluate *expr2* to a value *v2*. The result of the expression is the integer addition of *v1* and *v2*, unless the result overflows, in which case an exception is raised.

Static semantics:

1. *Constraints:* Both operands must have type *int*.
2. *Result type:* The result is type `int`.

List cell cons.

Syntax: `expr1 :: expr2`.

Dynamic semantics: Evaluate `expr1` to a value `v1`. Evaluate `expr2` to a value `v2`. The result of the expression is a fresh cons cell whose head points to `v1` and whose tail points to `v2`.

Static semantics:

3. *Constraints:* Suppose the left-hand operand has type *T*. In this case, the right-hand operand must have type *T* `list`.
4. *Result type:* The result has type *T* `list`.

---

2. `val` binding. (You do not have to describe pattern matching --- assume this has been defined.)

Syntax: `val pattern = expr`.

Dynamic semantics: Evaluate `expr` to a value `v`. Then, attempt to match `v` against `pattern`, yielding zero or more bindings to variables *{ x1, x2, ... xN }*. If the match succeeds, add the bindings to the current environment stack. If the match fails, add no bindings, but raise an exception.

Static semantics:

1. *Constraints:* It must be possible to assign the same type *T* to both the expression and the pattern.

2. *Result type:* This is not an expression; it is a declaration, so it has no result type. However, the bindings do have types: they will have the type corresponding to their position in the pattern.

(You might complain that some of this definition amounts to hand-waving; and you would be right. Defining the precise semantics of name binding in a formal fashion requires some moderate complexity. For this class, an informal description such as this one will suffice.)

---

3. `let` expressions. This can be defined two ways --- as a "primitive" construct, from the ground up, or as a syntactic sugar for a series of function applications.** Define it both ways.

"Primitive" definition

Syntax: `let` *`declList`* `in` *`expr`* `end`.

Dynamic semantics: Evaluate each declaration in *`declList`* in sequence, producing zero or more bindings to be added to the current environment. Then, evaluate *`expr`* in the current environment.

Static semantics:

1. *Constraints:* Each declaration must be well-typed. The body expression must be well-typed based in the environment produced by all the declarations.
2. *Result type:* The result type of the entire expression is the result type of the body expression.

Syntactic sugar definition

Syntax: let *declList* in *expr* end.

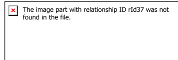
Dynamic semantics: First, rewrite the term as a series of nested let expressions, with only *one* declaration per let, as follows: let *decl* in (let*decl* in ... in *expr* end ... end) end). Then, rewrite each let-expression let
val *pattern* = *expr1* in *expr2* end as a function

call ((fn *pattern* =>*expr2*) *expr1*). Evaluate the resulting expression.

Static semantics: Perform the rewriting specified in the dynamic semantics, except for the evaluation step. Typecheck the resulting rewritten expression.

* For most of these constructs, the static semantics for full ML uses polymorphic types. For now pretend ML only has monomorphic types like `int`, `string`, or `(int * int * int)`.

** Actually, there are slightly different typechecking requirements between function application and `let`, but the distinctions are beyond the scope of your current knowledge.



This work is licensed under a Creative Commons License. Rights are held by the University of Washington, Department of Computer Science and Engineering (see RDF in XML source).

More on scoping of names

Lexical scoping

We have so far discussed the notion of **scope** (i.e., where a name binding is visible) rather informally. I would like to make your intuition more precise by describing ML's scoping rules in more detail.

Bindings in ML live in **environments**. Conceptually, each environment (except the top-level environment) consists of

a list of name-value pairs (the bindings in this environment); and

1. a pointer to the "parent" environment. This pointer refers to the textually enclosing environment *at the place where the environment is first defined*. (The top-level environment differs only in that it does not have a parent pointer.)

Consider the following code:

```
- val x = 5;
val x = 5 : int
- fun f y = x + y;                (* 1 *)
val f = fn : int -> int
- val x = 7;                      (* 2 *)
val x = 7 : int
- f 10;                           (* 3 *)
val it = 15 : int
```

In this code, the reference to `x` inside `f` refers to the binding `x = 5`. Regardless of whatever other bindings are later added to the top-level environment, the body of `f` will always be evaluated in that environment.

Figs. 1-3 show how this is implemented, conceptually. In Fig. 1, we have evaluated the declaration of `f` (at the line marked `(* 1 *)` above), which includes evaluating the function value. Unlike previous diagrams of function values in memory, we have included a picture of the value, sometimes called a **closure**, which contains two parts:

The code pointer, which points to the actual code to be evaluated.

1. The environment pointer, which points to the parent environment at the point of function definition. We say that the definition of `f` "captures" the environment in a "closure". Notice that this points to a part of the

environment that *includes* `f` --- this is necessary for recursive function definitions.

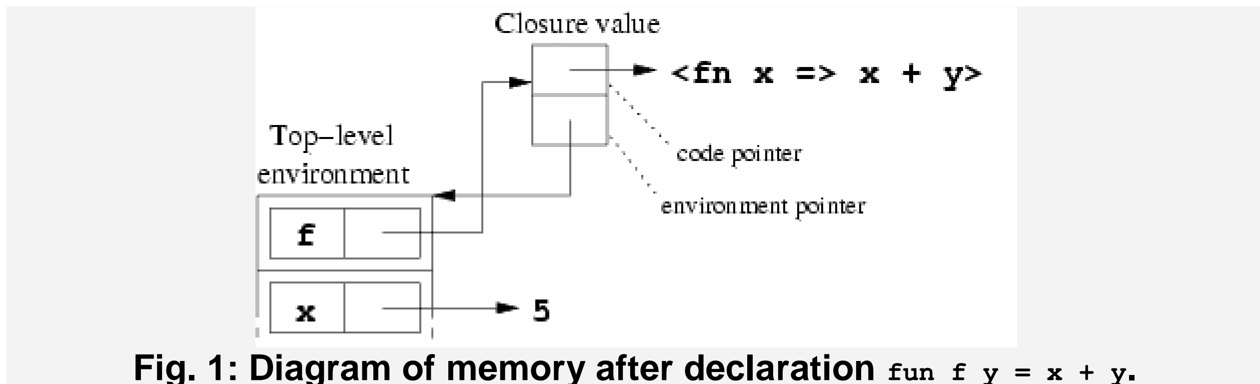


Fig. 1: Diagram of memory after declaration `fun f y = x + y`.

Fig. 2 shows what happens when we evaluate `val x = 7` (at the line marked `(* 2 *)`). Another binding is added that shadows the previous binding --- but only for later declarations. The closure continues to point to the older environment.

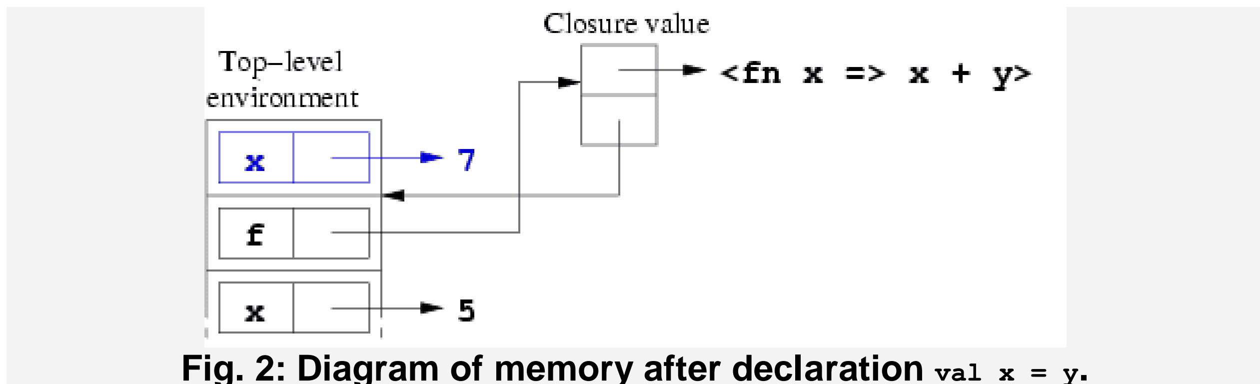


Fig. 2: Diagram of memory after declaration `val x = y`.

Finally, Fig. 3 shows what happens when we evaluate the function bound to `f`. First, a function *activation record* is created for this function. This activation has a pointer for the parent environment. Second, a the pointer from the closure is copied into the parent environment slot. Third, the actual argument value is matched against the function's argument pattern --- in this case, simply binding 10 to `y`. Finally, the function is executed in the environment of the activation --- lookup of `y` yields 10, and lookup of `x` yields 5. The expression `x + y` evaluates to 15, and this value is returned.

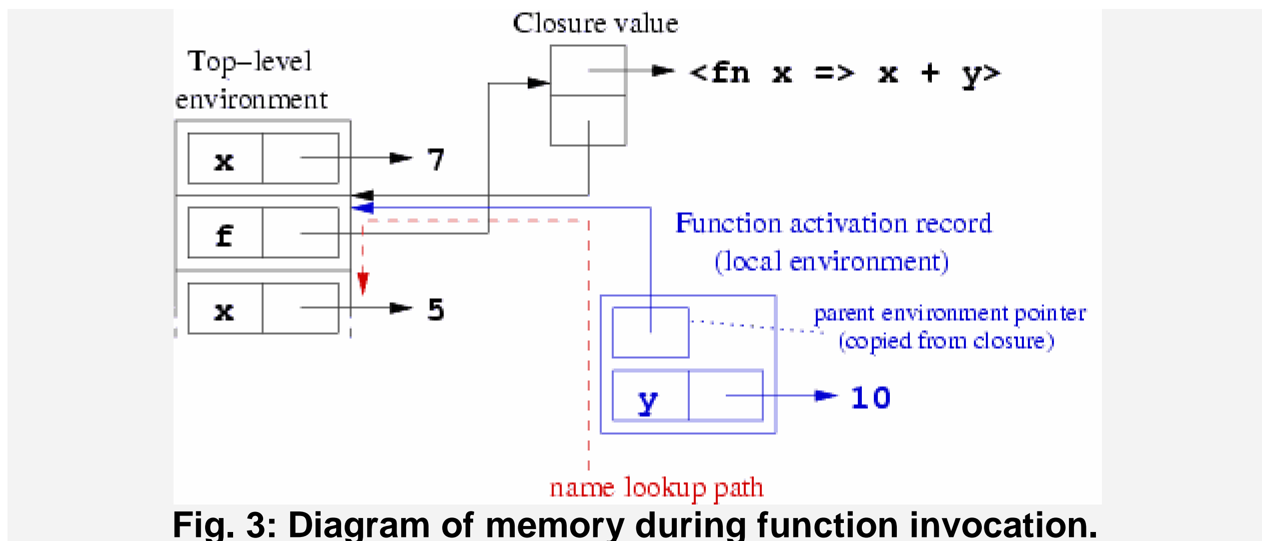


Fig. 3: Diagram of memory during function invocation.

Note that this is only the *conceptual* picture of what's going on. An optimized implementation might allocate activation records on a stack (perhaps the same stack as the top-level environment), and it might *copy* the captured bindings into the closure instead of keeping a pointer to the original environment.

Lexical vs. dynamic scoping

The scoping rule used in ML is called **lexical scoping**, because names refer to their nearest preceding lexical (textual) definition.

The opposite scheme is called **dynamic scoping** --- under dynamic scoping, names from outer scopes are re-evaluated using the most recently executed definition, not the one present when the code was originally written. Under dynamic scoping, the above transcript would return 17 for the value of `f 10`.

All sensible languages use lexical scoping. Dynamic scoping is of mostly historical interest --- early implementations of Lisp used dynamic scoping, which Joseph McCarthy (the inventor of Lisp) simply considered a bug. In languages that use dynamic scoping, functions are difficult to use and do not serve as well-behaved abstractions --- it is possible to make a function misbehave in ways that the writer of the function never anticipated, simply by accidentally redefining some name that may be used in the function.

Nested scopes: an extended example

So far, we have examined four contexts where name bindings may take place:

The top-level environment: bindings are visible until they are shadowed by later bindings.

The declaration parts of let-expressions: bindings are visible until the end of the let-expression body, unless they are shadowed by later bindings in the same let-expression.

Function arguments: bindings are visible in the function body, unless they are shadowed by nested binding constructs.

1. Rules of `case` statements: bindings are visible only inside the body of the rule, and may be shadowed by nested binding constructs inside the body.

These all follow rules similar to function arguments. For example, when one rule of a case expression is evaluated, a fresh environment is created with a parent pointer to the textually enclosing scope.

You may find it useful to figure out the scope of various names in the following code.

```
val a = 1;
val b = 3;
val f = fn x => x + a + b;
val a = 2;
fun g (foo, bar) =
    let
        val x = f a;
        val y = fn foo =>
            foo + x + let val n = 4 in n * n end;
        val x = y bar;
    in
        case foo of
            nil   => 0 - x
          | x::_ => x
    end;
```

Let-expressions and function application

Let-expressions are actually roughly equivalent to function applications --- witness the following equivalence:

```
- let val x = 5 in x + x end;
val it = 10 : int
- (fn x => x + x) 5
val it = 10 : int
```

In both cases, we bind a value to a name, then evaluate an expression in the environment produced by that name binding.

Let-expressions with more than one binding can be rewritten as a sequence of let-expressions, which in turn can be rewritten as a sequence of function applications:

```
- let val x = 5
```

```
    val y = 7 in
    x + y
end;
val it = 12 : int
- let val x = 5 in
    let val y = 7 in
        x + y
    end
end;
val it = 12 : int
- (fn x =>
    (fn y =>
       x + y
    ) 7) 5;
val it = 12 : int
```

Thought exercise: Why can't a let-expression with more than one binding simply be translated into a function taking a tuple of values? Hint: consider the let-expression

```
let
  val x = 5;
  val y = x + 1
in
  x + y
end;
```

Linear Search

Ver1:

```
use "/home/sak/sml/programs/random.sml";


local
   (* INV: 0<=i<= l = len(L) /\ forall j: 0<=j<i: a =/= L[j] *)
   fun linsearch (a, [], i) = i
     | linsearch (a, h::T, i) = if a=h then i else linsearch (a, T, i+1)


in fun search (a, L) =
      let val l = length L
             val i = linsearch (a, L, 0)
      in  if i >= l then print ("not found\n")
             else print ("found at index "^Int.toString(i)^"\n")
      end
end


val A = mkIntRandlist 1000;
val a = randomInt ();
search (a, A);
val b = List.nth (A, 786);
search (b, A)
```

Ver2:Iterative

```
use "/home/sak/sml/programs/random.sml";
val A = Array.fromList (mkIntRandlist 1000);
```

```
local

  fun linsearch (a, A, i, l) =
  (* INV: 0<=!refi<= l = len(L) /\ forall j: 0 <= j < !refi: a <> A[j] *)
        let val refi = ref i;
        in (while (!refi < l) andalso (a <> sub (A, !refi)) do
            refi := !refi+1
          );
          !refi
        end

in fun search (a, A) =
     let val l = Array.length A
           val i = linsearch (a, A, 0, l)
     in  if i >= l then print ("not found\n")
           else print ("found at index "^Int.toString(i)^"\n")
     end
end

val a = randomInt ();
search (a, A);
val b = sub (A, 786);
search (b, A)
```

Binary Search

(* Given a sorted array to implement binary search *)

```
open Array;

(* A[!lo..!hi] is the slice of the array A that needs to be searched
   !mid is the value returned -- !mid = n if x is not in the array.

   Assume R is an irreflexive total order on the domain of A and
   A is sorted according to R
*)
fun binsearch (A, x) =
    let val n = length A;
            val lo = ref 0 and hi = ref n;
            val mid = ref ((!lo + !hi) div 2);

    in  (* INV: ordered A /\ (forall i: 0<= i < lo: x <> A[i]) /\
                (forall j: hi < j < n: x <> A[j]) /\
            0<=!lo <= !mid <= !hi <= n /\
            2*!mid <= lo+hi <= 2*!mid +1

          Notice that mid = lo if hi = lo or hi = lo+1  and
            !lo <  !mid <  !hi only if !hi - !lo > 1
        *)
            while ((!hi - !lo > 1) andalso (x <> sub (A, !mid))) do
            (
             if x < sub (A, !mid) then hi := !mid - 1
             else (* sub (A, !mid) < x *) lo := !mid + 1;
             mid := (!lo + !hi) div 2
```

```
         );
         (* INV /\ ((!hi - !lo <= 1) \/ (x = sub (A, !mid)) *)
      if x = sub (A, !mid) then SOME (!mid)
         else NONE
   end;
(* try it *)

open Array;
val A = fromList [~24, ~24, ~12, ~12, 0, 0, 1, 20, 45, 123];
binsearch (A, 0);
binsearch (A, ~24);
binsearch (A, 123);
binsearch (A, 100);
binsearch (A, ~25);
binsearch (A, 124);
```

selection-sort-fun.sml

```
local
  exception emptyList;
  (* findMin finds the minimum element in the list and removes it *)
  fun findMin  [] = raise emptyList
    | findMin  [h] = (h, [])
    | findMin  (h::t) =
     let
        val (m, tt) = findMin  t;

     in  if m < h
           then (m, h::tt)
           else (h, t)
     end;

in fun selSortFun  [] = []
    | selSortFun  [h] = [h]
    | selSortFun  (L as h::t) =
     let
        val (m, LL) = findMin  L
     in m::(selSortFun  LL)
     end;
end
```

```
val sf = selSortFun;

sf [~12, ~24, ~12, 0, 123, 45, 1, 20, 0, ~24];
```

selection-sort-imp.sml

```
(* IMPERATIVE *)

open Array;

fun printarray A =
   let val n = length A;
          val i = ref 0;
   in  ( print ("        [");
        while (!i < n) do
        (  print (Int.toString (sub (A, !i)));
              if (!i < n-1) then print (",") else ();
          i := !i +1
        );
        print ("]\n")
          )
   end;

(*
## Assume A0 is the initial value of the array A[0..n-1]
## ordered (A, p, q) = forall j: 0 <= p <= j < q < n -> A[j] <= A[j+1]
## leftmin (A, p ) = forall i: 0 <= i < p < n: forall j: p <= j< n: A[i] <= A[j]
*)
```

```
fun swap (A, i, j) =
   let val temp = sub (A, i)
   in (update (A, i, sub (A, j));
      update (A, j, temp)
     )
   end;

local
    fun findIndexOfMin (A, i, n) =
           let val m = ref i and k = ref i
              (*
                INV2: 0<= i <= !m <= !k /\
                 forall j: 0<= i<=j < k <= n: A[m] <= A[j]
              *)
           in while !k < n do
              (
               if sub (A, !k) < sub (A, !m)
                       then m := !k
               else ();
                       k := !k + 1
                     );
              !m
           end

in fun selSortImp A =
  let
```

```
   val n = length A; (* A[0..n-1] *)
   val p = ref 0;
   val q = ref 0;

  (*
    ordered (A, i, k) = forall j: 0 <= i <= j < k < n-1: A[j] <= A[j+1]
    leftmin (A, p) = forall i, j: 0 <= i < p <= j < n: A[i] <= A[j]

    INV1: perm (A, A0) & 0<=p<=n & ordered (A, 0, p) & leftmin (A, p)

    BF : n-p
  *)
  in (
      while !p < n do
      (
        q := findIndexOfMin (A, !p, n);
        swap (A, !q, !p);
            p := !p + 1
      ); (* the algo terminates once !p = n *)
      printarray A
    )
 end
end;

val si = selSortImp;
si (fromList [~12, ~24, ~12, 0, 123, 45, 1, 20, 0, ~24]);
```

Insertion sort[edit]

Insertion sort for lists of integers (ascending) is expressed concisely as follows:

```
fun ins (n, []) = [n]
  | ins (n, ns as h::t) = if (n<h) then n::ns else h::(ins (n, t))
val insertionSort = List.foldr ins []
```

This can be made polymorphic by abstracting over the ordering operator. Here we use the symbolic name `<<` for that operator.

```
fun ins' << (num, nums) = let
   fun i (n, []) = [n]
     | i (n, ns as h::t) = if <<(n,h) then n::ns else h::i(n,t)
  in
    i (num, nums)
  end
fun insertionSort' << = List.foldr (ins' <<) []
```

The type of `insertionSort'` is `('a * 'a -> bool) -> ('a list) -> ('a list)`.

§Mergesort[edit]

Main article: Merge sort

Here, the classic mergesort algorithm is implemented in three functions: split, merge and mergesort.

The function `split` is implemented with a local function named `loop`, which has two additional parameters. The local function `loop` is written in a tail-recursivestyle; as such it can be compiled efficiently. This function makes use of SML's pattern matching syntax to differentiate between non-empty list (`x::xs`) and empty list (`[]`) cases. For stability, the input list `ns` is reversed before being passed to `loop`.

```
(* Split list into two near-halves, returned as a pair.
 * The “halves” will either be the same size,
 * or the first will have one more element than the second.
 * Runs in O(n) time, where n = |xs|. *)
```

```
local
  fun loop (x::y::zs, xs, ys) = loop (zs, x::xs, y::ys)
    | loop (x::[], xs, ys) = (x::xs, ys)
    | loop ([], xs, ys) = (xs, ys)
in
  fun split ns = loop (List.rev ns, [], [])
end
```

The local-in-end syntax could be replaced with a let-in-end syntax, yielding the equivalent definition:

```
fun split ns = let
  fun loop (x::y::zs, xs, ys) = loop (zs, x::xs, y::ys)
    | loop (x::[], xs, ys) = (x::xs, ys)
    | loop ([], xs, ys) = (xs, ys)
  in
    loop (List.rev ns, [], [])
  end
```

As with split, merge also uses a local function loop for efficiency. The inner `loop` is defined in terms of cases: when two non-empty lists are passed, when one non-empty list is passed, and when two empty lists are passed. Note the use of the underscore (`_`) as a wildcard pattern.

This function merges two "ascending" lists into one ascending list. Note how the accumulator `out` is built "backwards", then reversed with `List.rev` before being returned. This is a common technique—build a list backwards, then reverse it before returning it. In SML, lists are represented as imbalanced binary trees, and thus it is efficient to prepend an element to a list, but inefficient to append an element to a list. The extra pass over the list is a linear time operation, so while this technique requires more wall clock time, the asymptotics are not any worse.

```
(* Merge two ordered lists using the order lt.
 * Pre: the given lists xs and ys must already be ordered per lt.
 * Runs in O(n) time, where n = |xs| + |ys|. *)
```

```
fun merge lt (xs, ys) = let
  fun loop (out, left as x::xs, right as y::ys) =
        if lt (x, y) then loop (x::out, xs, right)
        else loop (y::out, left, ys)
    | loop (out, x::xs, []) = loop (x::out, xs, [])
    | loop (out, [], y::ys) = loop (y::out, [], ys)
    | loop (out, [], []) = List.rev out
  in
   loop ([], xs, ys)
  end
```

The main function.

```
(* Sort a list in according to the given ordering operation lt.
 * Runs in O(n log n) time, where n = |xs|.
 *)
fun mergesort lt xs = let
  val merge' = merge lt
  fun ms [] = []
    | ms [x] = [x]
    | ms xs = let
      val (left, right) = split xs
      in
        merge' (ms left, ms right)
      end
  in
```

```
  ms xs
 end
```

Also note that the code makes no mention of variable types, with the exception of the :: and [] syntax which signify lists. This code will sort lists of any type, so long as a consistent ordering function lt can be defined. Using Hindley–Milner type inference, the compiler is capable of inferring the types of all variables, even complicated types such as that of the lt function.

§Quicksort[edit]

Quicksort can be expressed as follows. This generic quicksort consumes an order operator `<<`.

```
fun quicksort << xs = let
  fun qs [] = []
   | qs [x] = [x]
   | qs (p::xs) = let
      val (less, more) = List.partition (fn x => << (x, p)) xs
      in
        qs less @ p :: qs more
      end
  in
   qs xs
  end
```

§Expression language[edit]

Note the relative ease with which a small expression language is defined and processed.

```
exception Err


datatype ty
  = IntTy
```

```
  | BoolTy

datatype exp
  = True
  | False
  | Int of int
  | Not of exp
  | Add of exp * exp
  | If of exp * exp * exp

fun typeOf (True) = BoolTy
  | typeOf (False) = BoolTy
   | typeOf (Int _) = IntTy
   | typeOf (Not e) = if typeOf e = BoolTy then BoolTy else raise Err
  | typeOf (Add (e1, e2)) =
     if (typeOf e1 = IntTy) andalso (typeOf e2 = IntTy) then IntTy else raise Err
  | typeOf (If (e1, e2, e3)) =
     if typeOf e1 <> BoolTy then raise Err
     else if typeOf e2 <> typeOf e3 then raise Err
     else typeOf e2

fun eval (True) = True
  | eval (False) = False
   | eval (Int n) = Int n
   | eval (Not e) =
```

```
    (case eval e
      of True => False
       | False => True
        | _ => raise Fail "type-checking is broken")
  | eval (Add (e1, e2)) = let
      val (Int n1) = eval e1
     val (Int n2) = eval e2
     in
      Int (n1 + n2)
     end
  | eval (If (e1, e2, e3)) =
     if eval e1 = True then eval e2 else eval e3

 fun chkEval e = (ignore (typeOf e); eval e) (* will raise Err on type error *)
```

§Arbitrary-precision factorial function (libraries)[edit]

In SML, the IntInf module provides arbitrary-precision integer arithmetic. Moreover, integer literals may be used as arbitrary-precision integers without the programmer having to do anything.

The following program "fact.sml" implements an arbitrary-precision factorial function and prints the factorial of 120:

```
fun fact n  : IntInf.int =
    if n=0 then 1 else n * fact(n - 1)
val () =
    print (IntInf.toString (fact 120) ^ "\n")
```

and can be compiled and run with:

```
$ mlton fact.sml
```

```
$ ./fact

66895029134491270575881180540903725867527463331380298102956713523016335

57244962989366874165271984981308157637893214090552534408589408121859898

4811143896500059649605212569600000000000000000000000000000
```

§Numerical derivative (higher-order functions)[edit]

Since SML is a functional programming language, it is easy to create and pass around functions in SML programs. This capability has an enormous number of applications. Calculating the numerical derivative of a function is one such application. The following SML function "d" computes the numerical derivative of a given function "f" at a given point "x":

```
- fun d delta f x =

    (f (x + delta) - f (x - delta)) / (2.0 * delta);

 val d = fn  : real -> (real -> real) -> real -> real
```

This function requires a small value "delta". A good choice for delta when using this algorithm is the cube root of the machine epsilon.[citation needed]

The type of the function "d" indicates that it maps a "float" onto another function with the type "(real -> real) -> real -> real". This allows us to partially apply arguments. This functional style is known as currying. In this case, it is useful to partially apply the first argument "delta" to "d", to obtain a more specialised function:

```
- val d = d 1E~8;

 val d = fn  : (real -> real) -> real -> real
```

Note that the inferred type indicates that the replacement "d" is expecting a function with the type "real -> real" as its first argument. We can compute a numerical approximation to the derivative of $f(x) = x^3 - x - 1$ at $x = 3$ with:

```
- d (fn x => x * x * x - x - 1.0) 3.0;

 val it = 25.9999996644  : real
```

The correct answer is $f'(x) = 3x^2 - 1$; $f'(3) = 27 - 1 = 26$.

The function "d" is called a "higher-order function" because it accepts another function ("f") as an argument.

Curried and higher-order functions can be used to eliminate redundant code. For example, a library may require functions of type `a -> b`, but it is more convenient to write functions of type `a * c -> b` where there is a fixed relationship between the objects of type `a` and `c`. A higher order function of type (a * c -> b) -> (a -> b) can factor out this commonality. This is an example of the adapter pattern.

§Discrete wavelet transform (pattern matching)[edit]

The 1D Haar wavelet transform of an integer-power-of-two-length list of numbers can be implemented very succinctly in SML and is an excellent example of the use of pattern matching over lists, taking pairs of elements ("h1" and "h2") off the front and storing their sums and differences on the lists "s" and "d", respectively:

```
- fun haar l = let
    fun aux [s] [] d = s  :: d
      | aux [] s d = aux s [] d
      | aux (h1::h2::t) s d = aux t (h1+h2  :: s) (h1-h2  :: d)
      | aux _ _ _ = raise Empty
    in
      aux l [] []
    end;
  val haar = fn  : int list -> int list
```

For example:

```
- haar [1, 2, 3, 4, ~4, ~3, ~2, ~1];
  val it = [0,20,4,4,~1,~1,~1,~1]  : int list
```

Pattern matching is a useful construct that allows complicated transformations to be represented clearly and succinctly. Moreover, SML compilers turn pattern matches into efficient code, resulting in programs that are not only shorter but also faster.