Programming Languages, Part A

Status Completed

Course Motivation

01/14/24

Introduction

- On a theoretical level, all programming languages are equivalent in the sense that any concept can be implemented in any language.
  - Nonetheless, certain concepts can be taught easier in certain programming languages.
  - The universality of a concept can be emphasized by implementing that concept in multiple programming languages.
- A programming language is just an interface for writing software.
  - Therefore there is no substitute for precisely understanding programming language semantics.

Why study general programming language concepts?

- To understand programming languages, one needs to understand semantics and idioms.
  - Correct reasoning about programs, interfaces, and compilers requires knowledge of semantics.
  - Idioms make you a better programmer.
    - Seeing idioms in multiple settings clue you in to common programming patterns that can be implemented in most, if not all, languages.

Are all programming languages the same?

- On a technical level, all programming languages are the same.
  - Any input-output behavior implementable in language X is implementable in language Y (Church-Turing thesis).
- Some fundamentals reappear, such as variables, abstraction, one-of types, recursive definitions, and so on.
- What separates these programming languages are:
  - The fact that the primitive/default condition in one language is awkward in another.
- Learning other programming languages is exciting, and can make you appreciate your preferred programming languages even more.

Why functional languages?

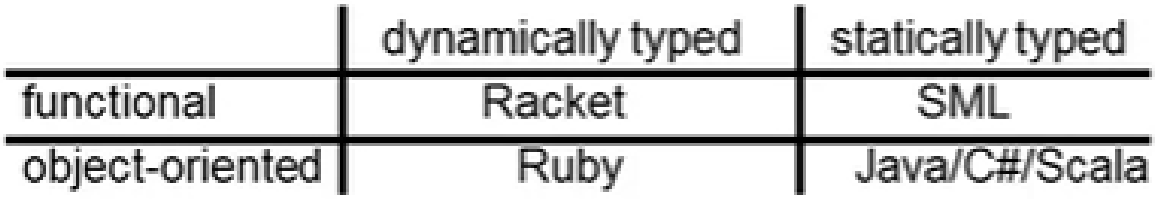
- Functional languages are useful because:
  1. They have language features that are invaluable for correct, elegant, and efficient software, and are a great way to think about computation.
  2. They have always been “ahead of their time”.
     a. Features like garbage collection, higher-order functions, type inference, and recursion were all implemented in functional languages, and dismissed as worthless ideas until they weren’t.

    b. Functional language concepts tend to get assimilated into other programming languages over time.
3. They are more well-suited towards where computing trends are going.

- With the absence of state, every input will have a consistent output.
  - Avoiding mutation is the easiest way to make concurrent and parallel programming easier.

Why SML, Racket, and Ruby?

- These three programming languages provide a very useful combination.



- SML provides polymorphic types, pattern-matching, and abstract types and modules.
- Racket provides dynamic typing, macros, and minimalist syntax.
  - We can define our own language in the Racket system.
- Ruby provides classes but not types, and is very object oriented.
- There are reasonable questions to ask when deciding to use/learn a programming language:
  - What libraries are available?
  - What tools are available?
  - What can land a job?
  - What is the industry standard?
- Not needing to deal with most of these questions for the course affords us the opportunity to use these more niche programming languages to learn the fundamentals.
  - In the long term, we will be able to apply these fundamentals to any programming language we want.

▼ Lecture 1:

01/05/24

ML variable bindings and expressions

- Comments in SML are created by enclosing text with `(* *)`.

```
(* This is a comment *)
```

- Variables are created using the `val` keyword.
  - Semicolons `;` are used to mark the end of expressions.

```
(* val <x> = <expression> *)

(* Assign 34 to x *)
val x = 34;
```

- Files can be run in the SML (read-eval-print-loop) REPL with `use`.

```
(* Run fil.sml *)

$ use "file.sml";
```

- Type-checking means checking what types are allowed for expression, and what type expression return.
  - If a type is invalid, an error will be thrown.
  - SML is a type-checked language.
- The static environment is active before a program is run.
  - The static environment does type-checking among other tasks.
- The dynamic environment is active when a program is run.
  - The dynamic environment evaluates bindings and expressions.
- New environments are "created" as the program is evaluated.
- Conditionals can be created using the keywords `if`, `then`, and `else`.

```
(* Evaluate the absolute value of z *)

val abs_oz_z = if z < 0 then 0 - z else z;
```

- Syntax refers to how something is written, where semantics refers to what something means.
- In this case, semantics are broken down into:
  - Type checking (before the program runs).
  - Evaluation (as the program runs).

Rules for expressions

- For every expression, ask yourself three questions:
  1. What is the syntax?
  2. What are the type-checking rules?
     a. How could the type-checking fail?
  3. What are the evaluation rules?
     a. An expression produces a value.
        - All values are expressions, but not all expressions are values.

The REPL and errors

- We need to ask ourselves some questions:
  - How do we run programs using the REPL?
  - What happens when we make mistakes?
- `use` enters variable bindings from a `.sml` file.
  1. The REPL reads the program.
  2. Next it evaluates the program.
  3. Finally, it prints the results.
  4. Repeat
- Values are printed in the REPL with their name, value, and type.

```
(* Anatomy of a value in the SML REPL *)

val <name> = <value> : <type>

(* Example of "x" being bound to the integer 7 *)

val x = 7 : int
```

- Errors can be raised as a result of improper syntax, type-checking, or during evaluation.
- The `~` character is used in SML to negate numbers.

```
(* negative 5 *)

~5
```

- The keyword `div` is used in SML to divide numbers.

```
(* 1 divided by 2 *)

1 div 2
```

Shadowing

- Shadowing is when a variable is added to an environment more than once.
  - In SML, assignment to a variable in not possible (they are immutable).
  - One can only shadow it in a later environment.

```
(* "a" is being shadowed, as its change in value doesn't affect other
   assignment statements like with "b" *)

(* In this example, the previous "a" is "shadowed" by the most recent "a" *)
```

```
val a = 10

(*  a : int
    a -> 10 *)

val b = a * 2

(* a -> 10
   b -> 20 *)

val a = 5 (* this is not an assignment statement *)

(*  a -> 5, b -> 20 *)
```

- Expressions in variable bindings are evaluated eagerly, meaning it is evaluated before the variable binding finishes.
  - After this, the expression producing the value is irrelevant.

Functions

- Functions are a key building block in SML, just like most other programming languages.

```
(* Example function to raise "x" to the power of "y" *)

(* Anatomy of a function *)
fun <name>(<param> : <type>...) =
    <expression>

(* Call a function *)
<name>(<arg>...)


fun pow(x : int, y : int) =
    if y = 0
    then 1
    else x * pow(x, y - 1)
```

- Functions are printed in the REPL with a special value `fn`, along with their name, input and output types.
  - Each input type is separated by a `*`.

```
(* Anatomy of a function in the REPL *)
val <name> = fn : type_1 * type_2 * ... * type_n -> type

(* Example using the previous "pow" function *)
val pow = fn : int * int -> int
```

- Remember that a function is a value itself.

Pairs and other tuples

- Tuples are a fixed number of values that may have different types.
- Pairs are tuples with two values.
- The type for tuples are their base component values separated by `*`s.
- The tuple components can be accessed individually using the syntax `#n`.

```
(* Example tuple *)
(* Tuple type is: int * bool *)

tup = (100, true)


(* Access the first element, 100 *)
#1 tup

(* Access the second element, true *)
#2 tup
```

Lists

- Unlike tuples, lists can have any number of elements.
- Unlike tuples, all element in a list have to contain the same type.

```
(* Empty list *)
[]

(* Anatomy of a list *)
[<val_1>, <val_2>, ..., <val_n>] (* where each value is of the same type *)
```

- `::` can be used to append a value to the front of a copy of a list.

```
(* Append "e1" onto "e2", creating a new list with "e1" on the front *)
e1::e2
```

- `null` returns a true if a list is empty.
- `hd` gets the first element of a list, or the head.
- `tl` get every element but the first element of a list, or the tail.

```
(* Ask if a list "e" is empty *)
null e

(* Produce "v1" of list "e" *)
hd e
```

```
(* Produce "v2, ..., vn" of list "e" *)
tl e
```

- The type `list` describes a list

```
(* A list "x" that bound to a list of ints, with type "int list" *)
$ val x = [1, 2, 3, 4] : int list
```

- An empty list has type `'a list`, or alpha list.
  - This means that the `a'` "alpha" can be replaced with any value.

Let expressions

- Let expressions allow for local scoping in SML.
- Let expressions are constructed in SML using the `let`, `in`, and `end` keywords.
  - The type returned by the expression is the type returned by the `in` binding enclosed the let expression.

```
(* Anatomy of a let expression *)
let
    <b_1,
     b_2,
   ...,
     b_n>
in
    <expression>
end
```

Nested functions

- Nested helper functions are useful when:
  - The helper function is unlikely to be useful elsewhere.
  - The helper function is likely to be misused elsewhere.
  - The helper function is going to be changed or removed later.
- One should but a function is a wide enough scope to save effort and avoids bugs, but also put it in a narrow enough scope to restrict how it is used.

01/06/24

Let and efficiency

- Avoid repeated recursive calls in a function, as this leads to exponential time complexity.
- To avoid this, `let` expressions can be used to precompute recursive function values.

```
(* "max_tail" is precomputed inside the let statement, and therefore the
   recursive function is only called once per recursion level instead of
   twice *)
```

```
(* Find the maximum value in a list *)
fun max (xs : int list) =
    if null xs
        then 0 (* horrible style, fix later in options section *)
        else
            let
                val max_tail = max(tl xs)
            in
                else if null (tl xs)
                    then hd xs
                else if hd xs > max_tail
                    then hd xs
                else
                    max_tail
    end
```

Options

- Sometimes an function runs into a problem where it needs to return either a zero or one element list.
- The type `option` achieves this goal by giving "options" to how many values it can hold.
- `NONE` builds an empty option.
- `SOME` builds a one element option.
- `isSome` returns true if the option is a `SOME`.
- `valOf` returns the value of a `SOME`.

```
(* The type "int option" is returned, meaning either an "int" or "NONE"
     is returned *)


(* fn : int list -> int option *)
fun max (xs : int list) =
    if null xs
        then NONE
    else
        let
            val tl_ans = max1(tl xs)
        in
            if isSome tl_ans andalso valOf tl_ans > hd xs
                then tl_ans
            else
                SOME (hd xs)
        end
```

Booleans and comparison operations

- Boolean operations in SML are `andalso`, `orelse`, and `not`.
- For evaluating `e1 <boolean operation> e2`:
  - `e2` is not evaluated if `andalso` `e1` is `false`.
  - `e2` is not evaluated if `orelse` `e1` is `true`.
  - This means that `andalso` and `orelse` are keywords, where `not` is just a function.
- This means that program languages don't actually need Boolean keywords.
  - Nonetheless, these Boolean operations are useful as they are more concise and readable.

```
(* Equivalents to "andalso", "orelse", "not" *)


(* e1 andalso e2 *0
if e1
    then e2
else
    false

(* e1 orelse e2 *)
if e1
    then true
else
    e2

(* not e1 *)
if e1
    false
else
    true
```

- In SML, "not equal to" is `<>`.
- `<`, `>`, `<=`, and `>=` can not be used with a combination of `int` and `real`.
  - `real` is the SML type for floats.

Benefits of no mutation

- In SML, there is no mutation for most forms of data.
- A lack of mutation in SML means that no data can be changed in a new dynamic environment.

```
(* In SML, there is no way to track if the returned "pr" is an alias
   of the input "pr" or not, where it would be in a language with
   mutable tuples *)
```

```
fun sort_pair (pr : int * int) =
    if #1 pr < #2 pr
        then pr
    else
        (#2 pr, #1 pr)
```

- Without mutation:
  - No code can ever distinguish aliasing vs. identical copies.
  - Because the original data can never change, there is no need to think about aliasing.
  - We can use aliasing to save space, without danger.
- This leads to SML being more efficient than other programming languages in some cases.
  - For example, `tl` in SML is constant in time complexity since it doesn't need to copy the list every time a recursive function is called.

Pieces of a programming language

1. Syntax: How do you write language constructs?
2. Semantics (Evaluation rules): What do programs mean?
3. Idioms: What are typical patterns for using language features to express your computation?
4. Libraries: What facilities does the language provide "standard"?
5. Tools: What do language implementation provide to make your job easier?
   a. These tools are separate from the language itself.

- In practice, all of these are essential for good programmers.
- It is essential how each of these pieces are separate from one another.
- This course focusses on semantics and idioms.
  - Learning semantics and idioms can make you better at understanding libraries and tools.

▼ Lecture 2:

01/07/24

Building compound types

- Most programming languages have base and compound types.
- There are three important type building blocks that can describe most types:
  - Each-of types, one-of types, and self-reference types.
- Tuples build each-of types.
- Options build one-of types.
- Lists use all three building blocks.
- These types can be nested to create more complex types.

Records

- Records are a type that can have multiple field names, with each field having a unique type.
  - Records are enclosed by `{}` with field names separated by commas.
  - Field names can be accessed with `#`.

```
(* A record "x" with the type {foo=int*int, bar=bool} *)
val x = {foo=(3, 2), bar=true}

(* Bind (3, 2) to y *)
val y = #foo x
```

- Tuples are useful when one wants to refer to elements by position, where records are useful when one wants to refer to elements by field name.\

Tuples as syntactic sugar

- There are no such thing as tuples in SML, as they are actually just syntactic sugar for records with ordered numeric field names.

```
(* This record will be show as a tuple (4, true, 7) with type int * bool * int
   by the REPL *)
val actually_a_record = (1=4, 3=7, 2=true)

(* 4 *)
#1 actaully_a_record
```

- Syntactic sugar simplify the understanding and implementation of a programming language.
- It also makes the core of programming languages more concise, and shortens proofs.
  - It is only “syntactic”, because the semantics can still be described in terms of another construct.
  - It is “sugar” because it makes the language sweeter 🍬.
  - As discussed previously, `andalso` and `orelse` are also forms of syntactic sugar.

01/09/24

Datatype bindings

- A `datatype` is a one-of type in SML.
- `datatype`s add constructors to the environment.
  - One job of a constructor is to be a function that makes values of the new type.

```
(* mytype contains one of int * int, string, or mytype, with each one
     being a constructor function *)
(* Each is a function that takes a value of the specified type, and converts
     it to a "mytype" *)
```

```
(* TwoInts: int * int -> mytype *)
(* Str: string -> mytype *)
(* Pizza: mytype *)

datatype mytype = TwoInts of int * int | Str of string | Pizza
```

- The value contains a tag and its corresponding data.
  - The tag specifies which constructor is being used.
- There are two aspects to accessing a datatype value.
  - Check what variant it is (what constructor made it).
  - Extract the data (if that variant has any).
- Types like lists and options have these components:
  - `null` and `isSome` check variants.
  - `hd`, `tl`, and `valOf` extract data.

Case expressions

- Case expressions are used to extract the data from `datatype` bindings.
  - It does so through accessing the pieces of a `datatype`.
  - It is really a multi-branch conditional to pick a branch based on a variant.
    - Case expressions use the first branch that matches via pattern matching.
    - They extract data and binds that data to local variables in the branch.
  - All branches must have the same type.

```
(* A case expression takes a value "x" with the type "datatype", and
     access the pieces of "mytype". *)

(* mytype -> int *)
fun f (x : mytype) =
    case x of
        Pizza => 3
      | Str s => s ^ "s"
      | TwoInts (i1, i2) => i1 + i2


(* Call "f" with a "mytype", return 3 *)
f (TwoInts (1, 2))
```

- A pattern is SML is a constructor name followed by the right number of variables.
  - Syntactically, most patterns look like expressions, but they are not the same.
  - They are not evaluated, they are just used for pattern matching and accessing `datatype` variables.
  - Patterns are evaluated in order.

Useful datatypes

- `datatype`s have many applications, such as enumerations, or representing real-world objects.

```
(* An enumeration to describe playing card suits *)
datatype suit = Club | Diamond | Heart | Spade


(* Datatype to describe students in a student database *)
datatype id = StudentNum of int
            | Name of string * (string option) * string
```

- `datatype`s are different from records in the sense that they are one-of types instead of each-of types.
- `datatype`s can have recursive constructors.

```
(* The "exp" datatype uses itself in the connstructor of "Add" and "Negate" *)

datatype exp = Constant of int
                        | Negate   of exp
                        | Add      of exp * exp
```

Type synonyms

- `datatype` bindings introduce a new type name that is distinct from all existing types.
- The only way to create values of the new type is a constructor.
- A type synonym is a new kind of binding, which creates another name for an existing type.
  - The type and name are interchangeable in every way.

```
(* The type "card" is a synonym for a tuple of type "suit" and type "int" *)

type card = suit * int


(* Create a new card value representing Ace of spades *)
(* This value is both a "card" and a "suit * int" *)

val ace_of_spades : card = (Ace, 1)
```

- For now, type synonyms are just a convenience for talking about types.

Lists and options are datatypes

- Options and lists are just a predefined datatype binding.
  - For options, `NONE` and `SOME` are constructors, and `isSome` and `valOf` use pattern matching.
  - For lists, `[]` and `::` are constructors.
- Pattern matching is better for options and lists for the same reasons as for all datatypes.

    - There are no missing cases.
    - There are no exceptions for the wrong variant.
- `null`, `tl`, `[]`, etc. are predefined for passing them as arguments to other functions, or just plain convenience.

Polymorphic datatypes

- Polymorphism is the ability for a value to take on multiple forms.
- Datatype bindings like `list` and `option` are polymorphic type constructors, in the sense that `list` is not a type, but type `'a list` is a type.
- SML allows one to define their own polymorphic type constructors.
- `datatype` has the option to take one or more type parameters in constructor definitions.

```
(* Example of a custom option datatype that takes in type 'a *)

datatype 'a option = NONE | SOME of 'a

(* Example of a tree datatype that takes in types 'a and 'b *)

datatypte ('a, 'b) tree = Node of 'a * ('a, 'b) tree * ('a, 'b) tree
                        | Leaf of 'b
```

Each-of pattern matching

- All `val` and `fun` binding uses pattern matching.
- Pattern matching works for records and tuples, as in the pattern `(x1, ..., xn)` matches the tuple `(v1, ..., vn)`, and the pattern `{f1=x1, ..., fn=xn}` matches the pattern `{f1=v1, ..., fn=vn}`.

```
(* Example of a case expression to match a triple datatype and sum
   it three values *)

fun sum_triple triple =
    case triple of
        (x, y, z) + x + y + z
```

- Value bindings are actually `val p = e`, where p can be a pattern, not just a variable.

```
(* Building on the previous example, we now use values to extract the
   values from triple *)

fun sum_triple triple =
    let
        (* The type checker would throw an error if this pattern did not match *)
        val (x, y, z) = triple
    in
```

```
        x + y + z
    end
```

- A function argument can also be a pattern, meaning that the arguments `p` in a function call `f` can be matched with `fun f (p) = e`.

```
(* Building on the previous example, we now see that parameters in
   functions are just syntactic sugar that use pattern matching *)
(* Because a pattern in being used, not explicit types
   are needed for functions *)

fun sum_triple (x, y, z) =
        x + y + z
```

- This removes the need for explicit types to be stated for arguments, as the type checker will catch any patterns that don't match their specified types.
- The important thing to understand is that this is exactly the same syntax as functions describes earlier in the course.
- We have been using pattern matching for functions all along.
    - Every function takes exactly one argument, that argument being a pattern.
    - What we have been calling multi-argument functions are just functions with a single tuple, and then pattern matching is used on that tuple to extract its values.
- This enables us to do useful things in SML that one may not be able to do in other programming languages.

```
(* Simple function to rotate the pieces of a tuple *)

fun rotate_left (x, y, z) = (y, z, x);

(* Because every function only takes one parameter, the return value can be
   immediately passed to another function, as long as the types match *)

rotate_left(rotate_left(3, 4, 5))
```

01/10/24

Type inference

- When case expressions are used for pattern matching, no types need to be explicitly stated in function parameters.

```
(* Using pattern matching is superior because the type checker knows how
     many fields the tuple has *)

fun sum_triple1 (x, y, z) =
    x + y + z
```

```
(* Without the int * int * int value, the type checker has
   no idea how many fields triple has *)

fun sum_triple2 triple =
    #1 triple + #2 triple + #3 triple
```

- If types are not used in a function and is flexible as a type, the type checker will declare it as an `'a` type.

```
(* Because "y" is not used, it does not neccesarily have to be an int type,
   and therefore the type checker declares it as an 'a type *)

fun partial_sum (x, y, z) =
    x + z
```

Polymorphic and equality types

- `'a` is more general type that can be used in place of a more specific type, as long as the alpha is used consistently for only that specific type.
- `'a` in a function signature cannot be replaced with multiple types.
- Variables such as `''a` with two `'` specify equality types.
  - Equality types are types that you can use the `=` operator on.
    - For example, `fn` or `real` types do not count as equality types.
  - Equality types are more specific that polymorphic types.

```
(* The type signature for "same_thing" is ''a * ''a -> string *)
fun same_thing(x, y) =
    if x=y then "yes" else "no"
```

Nested patterns

- Patterns can be put inside of other patterns.
- Nested pattern can lead to elegant, concise code.
  - One should avoid nested case expressions when possible.
  - A common idiom is matching against a tuple of datatypes to compare them.
- `_` is a pattern “wildcard” that matches everything.
  - Wildcards are good style, as they specify that the data from a variable does not need to be used inside that branch.

```
(* Each list is matched with a head and a tail *)
(* The heads are extracted and the tails are recursively fed into the zip
   function until only empty lists are left *)

fun zip list_triple =
    case list_triple of
        ([], [], []) = []
```

```
    | (hd1::tl1, hd2::tl2, hd3::tl3) => (hd1, hd2, hd3)::zip(tl1, tl2, tl3)
(* If the length of the functions don't match, that is, a call has neither
   a fully empty list or a full list, the _ (default case) will be called
   and raise an error to inform that the lengths of the lists mismatch *)
    | _ => raise ListLengthMismatch
```

Function pattern syntactic sugar

- SML allows functions that only contain case expressions to be written more concisely.

```
(* Without syntactic sugar *)
fun f x =
    case x of
        p1 => e1
    | p2 => e2
    ...
    | pn = en

(* With syntactic sugar *)

fun f p1 = e1
    | f p2 = e2
    ...
    | f pn = en
```

Exceptions

- Exceptions are thrown when there are runtime conditions that should be an error.
- The `exception` keyword can be used to bind custom exceptions.
  - All exceptions in SML have the type of `exn`.
  - These exceptions have the option to carry values.

```
(* This exception binding introduces a new kind of exception *)

exception CustomException

(* This exception binding carries two ints *)

exception TwoIntException of int * int
```

- The `raise` keyword is used to throw exceptions in SML.

```
(* Implementation of "hd" *)
(* An error is raised when the list is empty and there is no hd to access *)

fun hd xs =
```

```
    case xs of
        [] => raise List.Empty
    | x::_ => x
```

- `handle` can be used to match an exception name to a pattern, and then execute an expression in that branch.

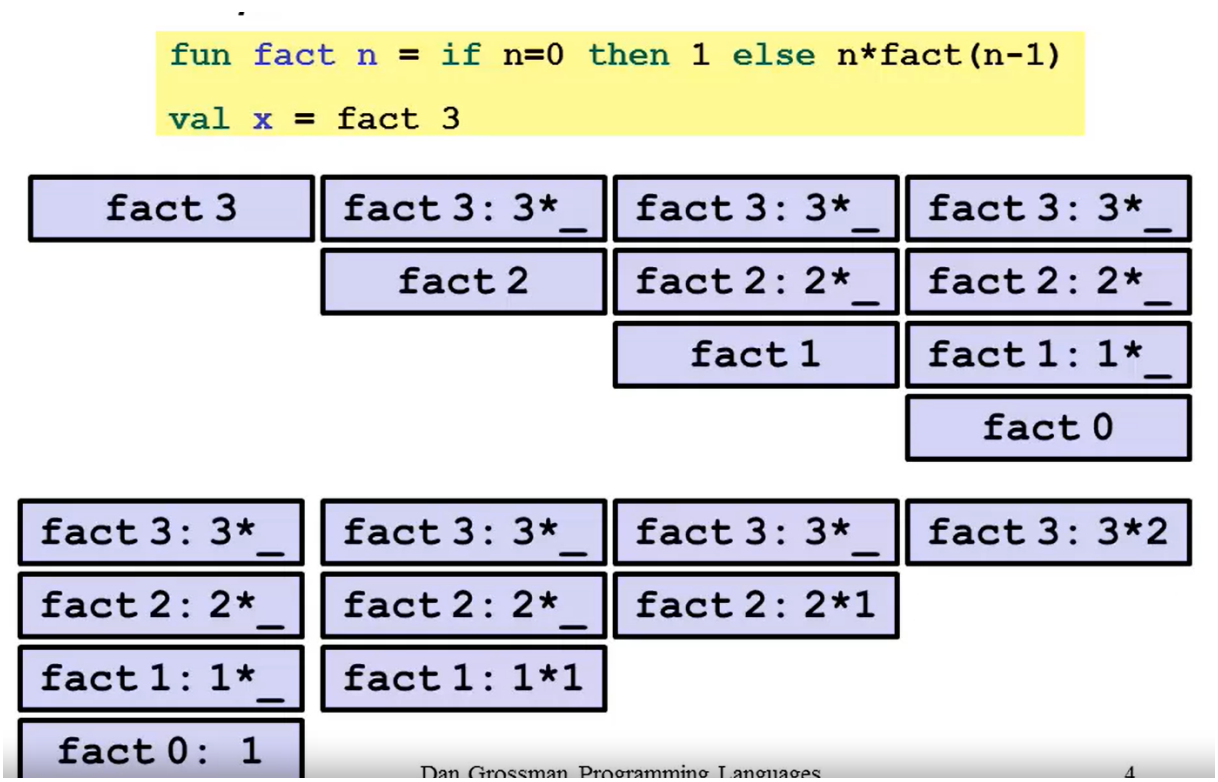
```
(* If the exception is not raised,
     the first expression is evaluated normaly *)
(* If the exception is raised,
   then the second expression is evaluated instead *)

<expression1> handle
    <exception> => <expression2>
```

- Exceptions are very similar to `datatype` constructors.
  - Declaring an exception creates a constructor for type `exn`, which can be passed anywhere.

Tail recursion

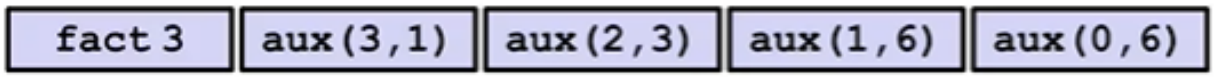
- While a program runs, there is a call stack of functions that have started but not yet returned.
  - Calling a function `f` pushes an instance of `f` onto the stack.
  - When a call to `f` finishes, it is popped from the stack.
- These stack-frames store information like the value of local variable and “what is left to do” in the function.
- Due to recursion, multiple stack-frames may be calls to the same function.



- It is unnecessary to keep around a stack-frame just so it can get a callee's result and return it without any further evaluation.
- In that case, tail recursion can be implemented with the assistance of an accumulator.
  - Such a situation is called a tail call.
  - ML recognizes these tail calls and treats them differently to optimize them.
    - The caller is popped from the stack before the call, allowing the callee to reuse the same stack space.
    - With this, among other optimizations, recursion is as efficient as a loop.

```
fun fact n =
    let fun aux(n,acc) =
            if n=0
            then acc
            else aux(n-1,acc*n)
    in
       aux(n,1)
    end

val x = fact 3
```



- It is reasonable to assume that all functional-language implementations do tail-call optimization.

Accumulators for tail recursion

- There is a methodology that can guide implementing an accumulator.
  1. Create a helper function that takes an accumulator.
  2. The old base case becomes the initial accumulator.
  3. The new base case becomes the final accumulator.
- Tail recursion can be implemented for case expressions.

```
(* The accumulator "acculumates" the sum of the int list *)

fun sum xs =
    let
        fun aux (xs, acc) =
            case xs of
                [] => acc
            | x::xs; => aux(xs', x + acc)
    in
```

```
        aux(xs, 0)
    end
```

Perspective on tail recursion

- There are certain cases where tail-recursive functions cannot be evaluated in a constant amount of space.
  - For example, functions that process trees should utilize the call stack to keep track of where they have been in the tree.
- Beware of premature optimization.
  - One should favor clear, concise code.
  - Do use less space if inputs may be large.
- A tail call is a function call that is in tail position.
  - If an expression is not in tail position, then none of its subexpressions will be in tail position either.
  - The tail position is that last expression in a function that needs evaluating.

▼ Lecture 3:

01/12/24

First-class functions

- Functional programming means:
  1. Avoiding mutation is most/all cases.
  2. Using functions as values.
- This style encourages recursion and recursive data structures, and is closer to mathematical definitions.
- First-class functions can be used wherever we use other values.
  - Functions are values too.

```
(* Define some functions *)
fun double x = 2 * x
fun incr x = x + 1

(* Bind the functions to a tuple *)
(* val a_tuple = (fn, fn) : (int -> int) * (int -> int) *)
val a_tuple = (double, incr)

(* Access double and pass in 9 as an argument, results in 18 *)
val eighteen = (#1 a_tuple)(9)
```

- The most common use in as an argument or result of another function.
  - The function that takes or returns first-class functions is called a higher-order function.

- Function closures are functions that use bindings from outside the function definition, in the scope where the function is defined.

Functions as arguments

- We can pass one function as an argument to another function.
- First class functions allow us to create higher-order functions that generalize specific functions, and avoid repetition.

```
(* Example of an higher-order function that can do the tasks of multiple
   simple recursive functions *)

(* Apply a function to "x", "n" time *)
fun n_times(f, n, x) =
    if n=0
        then x
        else f(n_times(f, n-1, x))


(* Use the higher-order function to create a simple double "n" times function *)
fun double x = x * 2
fun double_n_times(n, x) =
    n_times(double, n, x)

(* Use the higher-order function to create a simple triple "n" times function *)
fun triple x = 3 * x
fun triple_n_times(n, x) =
    n_times(triple, n, x)
```

Polymorphic types and functions as arguments

- Higher-order functions are often so generic and reusable that they have polymorphic types.

```
(* The polymorphism of n_times makes it more useful *)
(* Type:
   ('a -> 'a) * int * 'a -> 'a *)

fun n_times(f, n, x) =
    if n=0
        then x
        else f(n_times(f, n-1, x))
```

- That being said, not all higher-order functions are polymorphic, and not all polymorphic functions are higher-order.

Anonymous functions

- Anonymous functions, commonly known as lambda functions in other programming languages, are functions that can be defined without a binding, but with an expression.
    - These are commonly used when a simple function is only needed in one place and nowhere else.

- Anonymous functions use `fn` instead of `fun`, use `=>` instead of `=`, and have no name.
- Anonymous functions cannot define recursive functions.
  - If our function doesn't have a name, therefore there is no way to invoke the function recursively.

```
(* Anonymous function anatomy:
   fn <args> => <expression>


(* We can define triple in triple_n_times with an anonymous function *)
fun triple_n_times(n, x) =
    n_times(fn x => 3 * x, n, x)
```

Unnecessary function wrapping

- Sometimes we don't need anonymous functions.
  - When all an anonymous function does is pass its argument to another function, just use that other function instead.

```
(* This lambda function is redundant *)
fun nth_tail(n, xs) =
    n_times(fn y => tl y, n, xs)


(* A simper solution is just to use tl, as tl is
   already the expression that is needed *)
fun nth_tail(n, xs) =
    n_times(tl, n, xs)
```

Map and filter

- `map` and `filter` are higher-order functions that are built into most programming languages.
- `map` uses, or "maps" a function to every element in a list.
  - In SML, the function syntax is `List.map`.

```
(* How the built-in map function works *)
(* Type:
    (a' -> b') * 'a list -> 'b list *)
fun map (f, xs) =
    case xs of
        [] => []
      | x:xs' => (f x)::map(f, xs')

(* Use map, produce [5, 9, 13, 17] *)
(* Specific Type (with anonymouse function):
    (int' -> int') * 'int list -> 'int list *)
val x1 = map(fn x => x + 1, [4, 8, 12, 16])
```

- `fitler` filters a list based on whether each element returns true for the argument function.
  - In SML, the function syntax is `List.filter`.

```
(* How the built-in filter function works *)
(* Type:
   ('a -> bool) * 'a list -> 'a list *)
fun filter (f, xs) =
  case xs of
    [] => []
  | x:xs' => if (f x)
               then x::filter(f, xs')
               else filter(f, xs')

(* Specific Type (with anonymouse function):
   ('int -> bool) * 'int list -> 'int list *)
(* Use filter, produce [12, 16]) *)
val x2 = filter(fn x => x > 9, [4, 8, 12, 16])
```

01/14/24

Generalizing prior topics

- First class functions are also useful in the fact that they can return specific function based on input.

```
(* Returning a double or a triple function based on the input "f" *)
fun double_or_triple f =
    if f 7
    then fn x => 2 * x
    else fn x => 3 * x
```

- Higher order functions are not just good for numbers and lists.
  - They work great for common recursive traversals over data structures.
- Functions that processes data into Booleans are sometimes called predicates.

Lexical scope

- Function bodies can use any bindings that were defined in their scope, not the scope where the function is being called.
- The semantics for this is called lexical scope.
  - Understanding lexical scope is crucial for utilizing first-class functions effectively.

```
(* f uses the value of x = 1 that was mapped
   when the function was defined, not x = 2 *)

val x = 1
    (* x maps to 1 *)
fun f y = x + y
```

```
    (* f maps to a function that adds 1 to its argument *)
val x = 2
    (* x shadows to 2 *)
val y = 3
    (* y maps to 3 *)
val z = f (x + y)
    (* Call the function defined on line 2 with f(5) *)
    (* z maps to 6 *)
```

- The language implementation uses closures to hold on to older environments that wouldn't be around otherwise, in order to retrieve the correct function definition.
- A function value has two parts:
    1. The code.
    2. The environment that was current when the function was defined.
- This pieces of this pair cannot be accessed, but the pair can be called.
- This pair is called a function closure.
    - A call evaluates the code using the environment.

```
(* When we define "f", a function closure is created *)

(* Code: take y and have body x + y *)
(* Enviornment: x maps to 1 *)

val x = 1
fun f y = x + y
...

(* Now when "f" is used, it calls the closure where x is mapped to 1 *)

val x = 2
val z = f (x + y)
```

Lexical scope and higher-order functions

- The function closures binds values within a function's lexical scope, similar to how a `let` statement works.

```
(* Written code *)

val x = 1
fun f y = x + y

(* f is implemented with a let statement where x is defined within scope *)

fun f y =
    let
```

```
        x = 1
    in
        x + y
    end
```

Why lexical scope

- Lexical scope is the environment where a function is defined.
- Dynamic scope is the environment where a function is called.
- Lexical scope is used in almost all modern programming language implementation for three precise, technical reasons:
  1. Function meaning does not depend on variable names used.
     a. Variable names can be changed within the scope, and it won't effect any of the function calls.
     b. This allows for modularity in software, as variable names won't override one another if they are in separate lexical scopes.
  2. Functions can be type-checked where they are defined.
     a. This can lead to operating on combinations of types that are not allowed, resulting in undefined types.
  3. Closures can easily store the data they need.
     a. Variable names are "locked in" so-to-say.

```
(* A function that takes x, and returns a function that asks
   if y is greater than x *)
fun greater_than_x x = fn y => y > x

(* The value of x is locked in as -1 *)
(* fn y => y > -1 *)
greaterThan ~1
```

Closures and recomputation

- Closures can be used to avoid repeating computations that do not depend on function arguments.
  - If we did not use closures to store these computations, we would have to compute them every time the function is called instead.

```
(* Filter strings that are equal to or longer than string s *)
fun all_shorter_than_1 (xs, s) =
    filter (fn x => String.size x < String.size s, xs)

(* Avoid recomputation by computing the size of s once *)
fun all_shorter_than_2 (xs, s) =
    let
        val len_s = String.size s
    in
```

```
        filter (fn x => String.size x < len_s, xs)
    end
```

Fold and more closures

- `fold` is an iterator over recursive functions.
  - It accumulates an answer by repeatedly applying `f` to an answer so far.

```
(* How the built-in fold function works *)
(* Type:
    ('a * 'b -> 'a) * 'a * 'b list -> 'a *)
fun map (f, acc, xs) =
    case xs of
        [] => acc
      | x:xs => fold(f, f(acc, x), xs)


(* Use fold to find the sum of a list, produce 10 *)
(* Specific Type:
    ('int * 'int -> 'int) * 'int * 'int list -> 'int *)
fold(fn (x, y) => x + y, 0, [1, 2, 3, 4])
```

- This pattern separates recursive traversal from data processing.
  - One can use the same traversal for different data processing.
  - We can reuse the same data processing for different data structures.
  - In both cases, using a common vocabulary concisely communicates intent.
- Functions like `map`, `filter`, and `fold` are much more powerful thanks to closures and lexical scope.
  - Functions passed to iterators can use any “private” data in its environment.
  - The iterator does not need to know what data is there or what type the data has.

Composing functions

- Function closures can be used to compose two or more functions.

```
(* Use closures to compose two functions *)
(* Type:
   ('b -> 'c) * ('a -> 'b) -> ('a -> 'c) *)
fun compose (f, g) = fn x => f(g x)


(* SML implementation, where the "o" composes f and g *)
f o g
```

Currying

- Functions take “multiple” arguments encoded in one argument via a tuple.

- Currying is a closure idiom that takes one argument, and returns a function that takes the next argument, and so on.
  - Currying allows one to call functions using multiple arguments with spaces instead of a tuple expression.
  - SML also has built in syntax support for currying to make implementation less verbose.
    - Function parameters are listed separately with spaces instead of with commas, and aren't enclosed with parentheses.

```
(* Old way to get the effect of multiple arguments *)
fun sorted3_tuple (x, y, z) =
    z >= y andalso y >= x1

val t1 = sorted3_tuple (7, 9, 11)


(* New way using currying *)
val sorted3_curry =
    fn x => fn y => fn z => z >= y andalso y >= x

val t2 = (((sorted3_curry 7) 9) 11)


(* Currying cleaned up with syntactic sugar *)
val sorted3 x y z =
    z >= y andalso y >= x

val t3 = sorted3 7 9 11
```

- It is important to remember that despite the similar syntax to tuples, the currying is semantically returning functions nested within other functions.
  - The syntax support is just syntactic sugar.

Partial application

- If the caller provides too few arguments to a curried function, a function closure is returned that is "waiting for the remaining arguments".
- This process is called a partial application, and can be used to "save" a function for later midway through its evaluation.

```
(* List.filter usually takes two arguemnts, but by partial application
   one can create a more spcific filter function that can be applied
   multiple times *)

val remove_zeros = List.filter (fn x => x <> 0)

(* Produces [1, 3, 7] *)
val list_without_zeroes = remove_zeroes [1, 0, 3, 0, 0, 7]
```

- One may run into the value restriction when using partial application on curried polymorphic functions.

- This is a limitation of SML programming language itself.
- This only happens when the resulting function of a partial application is polymorphic.

Currying wrap-up

- Higher-order wrapper functions can be used to:
  - Modify the order of arguments in curried functions.

```
(* Modify the order of arguments in a two argument curried function *)
fun other_curry x y = y x
```

  - Covert tupled functions to curried functions.

```
(* Convert tupled function to curried function *)
(* Tupled function *)
range (i, j) =
    ...

(* Curry wrapper function *)
fun curry f x y = f (x, y)

(* Pass "range" into "curry" to create a range function that uses
   currying syntax *)
val range_curry = curry range
```

  - Convert curried functions to tupled functions.

```
(* Convert curried function to tupled function *)
(* Curried function *)
range i, j =
    ...

(* Un-curry wrapper function *)
fun uncurry f (x, y) = f x y

(* Pass "range" into "uncurry" to create a range function that uses
   tupled syntax *)
val range_tuple = uncurry range
```

- Currying and tupling multiple arguments are both constant-time operations, so it generally doesn’t matter which method is used.
- One should prioritize programming in a way that is elegant, short, concise, and correct.

Mutable references

- Mutable data structures are useful in certain situations.
  - For example, when the state model needs to be updated.

- SML does this with a separate language construct called a `t ref`, where `t` is a type.

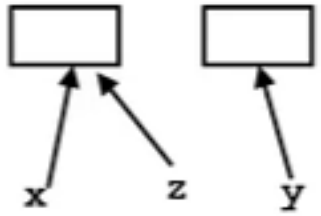
```
(* Create a reference with initial contents e *)
ref e

(* Update reference contents *)
e1 := e2

(* Retrieve reference contents *)
!e
```

- A variable bound to a reference is still immutable, as in it will always refer to the same reference.
  - The difference is that the contents of the reference may change via `:=`.
- Aliases to the reference are allowed.

```
val x = ref 42
val y = ref 42
val z = x
val _ = x := 43
val w = (!y) + (!z) (* 85 *)
(* x + 1 does not type-check *)
```



- In the example above, the `val _ =` is necessary because the `x := 43` is an assignment operation. The value name is `_` because that value is not being used anywhere.

Callbacks

- Callbacks are when a library takes a function to apply later, when an event like a keypress, mouse movement, or data arrival occurs.
  - Different callbacks may need different private date with different types.
  - Fortunately, a function's type does not include the types of bindings in its environment.
- Because of these properties, closures are needed for this language idiom.
- Mutable state is appropriate here, because we want the library to internally maintain a list of registered callbacks that can be updated.

Standard-library documentation

- Many programming languages have a standard library. These libraries exist:
  - For things you cannot implement on your own using the programming language, like openings files or setting timers.
  - For things so common, a standard definition is appropriate, like string concatenation or mapping functions.

- The documentation for Standard ML is:

> https://smlfamily.github.io/Basis/manpages.html

- REPLs can be used for reminding oneself of how a function works.
  - It gives information about function types, arguments, and sometimes structure.
  - Some programming languages have special support for printing documentation.

▼ Lecture 4:

01/16/24

What is type inference

- Static type checking can reject a program before it runs to prevent errors.
  - Statically typed programming languages have static type checking.
  - SML is one such statically typed language, where every binding has one type that is determined the program is compiled.
    - SML is implicitly typed, meaning that one rarely needs to write down types.
- Dynamically typed programming languages do little such checking.
  - Dynamically typed languages determine types during runtime.
- Type inference is the giving every binding/expression in a program a type such that type-checking succeeds.

ML type inference

- The key steps for type checking are as follows:
  1. Determine types of bindings in order.
     a. One cannot use bindings from environments later in the program that the type checker has not gone over yet.
  2. For each `val` or `fun` binding:
     a. Analyze the definition for all necessary facts.
        i. For example, `x` in `x > 0` must have type int.
     b. A `TypeError` is raised if there is no way for all facts to hold.
        i. This would mean that the expression is over-constrained.
  3. Afterward, use polymorphic type variables like `'a` for any unconstrained types.
  4. Finally, enforce the value restriction.

```
val x = 42 (* val x : int *)

fun f (y, z, w) =
    if y (* y must be bool *)
    then z + x (* z must be int *)
    else 0 (* both branches have same type *)
(* f must return an int
   f must take a bool * int * ANYTHING
   so val f : bool * int * 'a -> int
 *)
```

- A central feature of SML type inference is that it infers polymorphic type variables when it can.
- However, type inference and type variables are two separate concepts.
  - Languages can have type inference without type variables, and vice versa.

Type inference examples

- SML statically type checks a function using the rules listed above in order.

```
(*
1. Determine types of bindings in order:
    f : T1 -> T2 [must be a function; all functions take 1 argument]
    x : T1

    y: T3
    z: T4

2. For each val and fun binding analyze the definition for all facts:
    T1 = T3 * T4 [else pattern match does not type-check]
    T1 = int [abs has type int -> int]
    T4 = int [because we added z to an int]
    so: T1 = int * int
    so: (abs y + z) : int, so let-expression : int, so body : int,
    so: T2 = int

    f : int * int -> int

3. n/a
4. n/a
 *)
fun f x =
    let
        val (y, z) = x
    in
```

```
        (abs y) + z
    end
```

01/17/24

Polymorphic type inference examples

- The type checker infers the existence of polymorphic type variables when a function is under-constrained.
  - All the `T` variables are replaced consistently with type variables like `'a`, `'b`, etc.

```
(*
1. Determine types of bindings in order:
    length : T1 -> T2
    xs : T1

    x : T3
    xs' : T3 list

2. For each binding analyze the definitions for each fact:
    T2 = int
    that's all...

    T3 list -> int

3. Use polymorphic types for unconstrained types:
    T3 = 'a

    'a list -> int
 *)
fun length xs =
    case xs of
        [] => 0
      | x:xxs' => 1 + (length xs')
```

The value restriction and other type-inference challenges

- SML type-inference is too lenient, and therefore the value restriction limits where polymorphic types can occur.
- In the example case below, the type checker would allow `1` to be added to a `string`.
  - This is because the type of `r` is declared to have a polymorphic type.
  - Because `r` is a `ref` and can be mutated, the type can be made more specific in an environment further down, breaking the type checking rules.

```
val r = ref NONE (* val r : 'a option ref *)
val _ = r := SOME "hi"
val i = 1 + valOf (!r)
```

- To restore correctness, there needs to be a stricter type system.
- The value restriction solves with by making sure that the only expressions that can have polymorphic types are variables or values.
  - These unconstrained types are filled with dummy types that are essentially unusable.
- Type inference can be done without polymorphism, but this can lead to more restrictive functions.
  - For example, a function that determines the length of a list would only be able to take a list of one specific type.

Mutual recursion

- Mutual recursion is when two or more functions can call each other.
  - For example, function `f` can call function `g`, and `g` can call `f`.
- This is useful for multiple common programming idioms and patterns.
- In SML, the keyword `and` can be used to define mutually recursive functions.
  - Every function in the expression are type-checked together and can therefore refer to each other.

```
(* Functions f1, f2, and f3 are allowed to call eachother *)
fun f1 p1 = e1
and f2 p2 = e2
and f3 p3 = e3
```

- This can also be used for defining mutually recursive datatypes.
- Mutually recursive functions can be used for a state-machine idiom.
  - A state machine processes an arbitrary length list.
  - As the state machine processes the list, it is always in one of a known number of states.
  - Each next input tells the state machine what “state” to go to next.
  - This can be implemented with mutually recursive function by having one function for each “state”.

Modules for namespace management

- For larger programs, one top-level sequence of bindings is poor style.
- SML provides the keyword `structure` to define modules that can separate bindings from other modules.
  - Inside a module, one can use earlier bindings as usual.
  - Outside a module, one must refer to earlier modules’ bindings using `ModuleName.bindingName`.
  - It is convention to capitalize module names.

- SML has built-in modules like `String` and `List`.
- Structure names are not variables, one can only use the bindings inside of them.

```
(* Create a module "MyModule" *)
structure <Name> = struct <bindings> end

(* Refer to module binding *)
val x = <Name>.<binding>
```

- This module system allows for namespace management.
  - It gives a hierarchy to names, and therefore avoids shadowing.
  - It allows for different modules to reuse names.

Signatures and hiding things

- Signatures are simply types for modules.
  - They describe what bindings a module has and what those bindings types are.
- One can define a signature with `signature` and ascribe it to a module with the `:>` operator.
  - The module will not type-check unless it matches the signature, meaning it has all the bindings in the signature with all the correct types.

```
(* Define a signature *)
signature MY_MATH_LIB
sig
    val fact : int -> int
    val half_pi : int
end


(* Define a structure *)
(* Type:
    structure MyMathLib:
        sig
            val fact : int -> int
            val half_pi : int
        end
*)
strucutre MyMathLib :> MY_MATH_LIB
struct
    fun fact x = ...
    val half_pi = Math.pi / 2
end
```

- Signatures are useful for hiding bindings and type definitions.
- It is useful to have a “private” top-level function so that two or more functions could easily share a helper function.

- SML does this with signatures that omit bindings.
- One defines private bindings that can only be used in the module by omitting that binding from the module's assigned signature.

```
(* Omit half_pi from the signature, essentially making it a private binding *)

(* Define a signature *)
signature MY_MATH_LIB
sig
    val fact : int -> int
end

(* Define a structure *)
(* Type:
    structure MyMathLib:
        sig
            val fact : int -> int
            val half_pi : int
        end
*)
strucutre MyMathLib :> MY_MATH_LIB
struct
    fun fact x = ...
    val half_pi = Math.pi / 2
end


(* This will not work, as half_pi cannot be accessed outside of MyMathLib *)
val x = MyMathLib.half_pi
```

Abstract data types using signatures

- Modules should hide concrete type definitions so clients cannot create invariant-violating values of the type directly.
  - This is achieved with abstract data types (ADTs), which are custom types that exist but have inaccessible definitions.
  - One can define ADTs in signatures with the keyword `type`.

```
(* Define a type "natural" which can be type checked *)
(* This type "natural" cannot be accessed outside the structure in which
     the signature MY_MATH_LIB is used *)

signature MY_MATH_LIB
sig
    type natural
    (* fact takes an ADT natural, and returns int *)
    val fact : natural -> int
end
```

- As a result, there are two ways to use signatures to hide data from an external user of a module.
    1. Omit bindings from the signature so they cannot be used outside a module.
    2. Create abstract data types so that values cannot be created from them outside the module.

Signature matching

- Signature matching is what the type checker does to check if a `structure` can have a `signature`.
- The type checker checks if every non-abstract datatype, abstract datatype, `val` binding, and exception in the signature is provided in the structure/module.
    - The module can be more general than the signature.
    - The signature must be more specific than the module.

Equivalent implementations

- A key purpose of abstraction is to allow different implementations to be equivalent.
    - In essence, a function should be a “black-box”, where the end user does not need to know how it works.
    - It allows the implementation of a function to be improved or changed as long as the functionality doesn’t change.
    - This is easier to do if one starts with more abstract signatures that reveal less to an end user.

Different modules define different types

- Although multiple signatures can define abstract datatypes with the same name, each implementation introduces a new type.
- This means that when passing a module’s unique type into another module containing a type of the same name, it does not type check.

```
(* Although the types may seem to be equivalent, natural in MathLib1
   actually has type MathLib1.natural, and natural in MathLib2 actually
   has type MathLib2.natural


signature MLIB_1
sig
    type natural
end
structure MathLib1 :> MLIB_1
struct
    ...
end


signature MLIB_2
sig
    type natural
end
structure MathLib2 :> MLIB_2
```

```
struct
    ...
end
```

Equivalent functions

- Understanding when two pieces of code are equivalent is a fundamental software engineering idea.
  - Equivalence is made easier with more abstraction and fewer side effects.
- Code maintenance, backward compatibility, optimization, and abstraction rely on Equivalence.
- This raises the question of how one can define that two functions are equivalent.
- Two functions are equivalent if they have the same “observable behavior” no matter how they are used anywhere in a program.
  - They have the same input and the same output.
  - They produce equivalent results.
  - They raise the same exceptions.
  - They have the same termination and non-termination behavior.
  - They mutate non-local memory in the same way.
- Pure functions are functions without side effects.
  - Pure functions are therefore preferred more maintaining function equivalence.

Standard equivalences

- Syntactic sugar is, by definition, always equivalent with the semantics that it is syntactic sugar for.
  - If an implementation changes the evaluation order of a program, it cannot be syntactic sugar.
- Consistently renamed bound variables maintains function equivalence.
- Using a helper function maintains function equivalence.

Equivalence versus performance

- Even though functions are equivalent, that doesn’t mean that they are just as efficient.
- Program language equivalence refers to having the same inputs, outputs, and side effects.
- Asymptotic equivalence refers to focusing on time efficiency for an algorithm for a large input.
- Systems equivalence is like asymptotic equivalence, but accounts for differences between time efficiencies with different coefficients.
  - This equivalence does not study the general algorithm like asymptotic equivalence does.
- All three of these equivalences are important tools that are used to help software engineers write efficient programs.