```sml
(* Comments in Standard ML begin with (* and end with *).  Comments can be
   nested which means that all (* tags must end with a *) tag.  This comment,
   for example, contains two nested comments. *)

(* A Standard ML program consists of declarations, e.g. value declarations: *)
val rent = 1200
val phone_no = 5551337
val pi = 3.14159
val negative_number = ~15  (* Yeah, unary minus uses the 'tilde' symbol *)

(* And just as importantly, functions: *)
fun is_large(x : int) = if x > 37 then true else false

(* Floating-point numbers are called "reals". *)
val tau = 2.0 * pi         (* You can multiply two reals *)
val twice_rent = 2 * rent  (* You can multiply two ints *)
(* val meh = 1.25 * 10 *)  (* But you can't multiply an int and a real *)

(* +, - and * are overloaded so they work for both int and real. *)
(* The same cannot be said for division which has separate operators: *)
val real_division = 14.0 / 4.0  (* gives 3.5 *)
val int_division  = 14 div 4    (* gives 3, rounding down *)
val int_remainder = 14 mod 4    (* gives 2, since 3*4 = 12 *)

(* ~ is actually sometimes a function (e.g. when put in front of variables) *)
val negative_rent = ~(rent)  (* Would also have worked if rent were a "real" *)

(* There are also booleans and boolean operators *)
val got_milk = true
val got_bread = false
val has_breakfast = got_milk andalso got_bread  (* 'andalso' is the operator *)
val has_something = got_milk orelse got_bread   (* 'orelse' is the operator *)
val is_sad = not(has_something)                 (* not is a function *)

(* Many values can be compared using equality operators: = and <> *)
val pays_same_rent = (rent = 1300)  (* false *)
val is_wrong_phone_no = (phone_no <> 5551337)  (* false *)

(* The operator <> is what most other languages call !=. *)
(* 'andalso' and 'orelse' are called && and || in many other languages. *)

(* Actually, most of the parentheses above are unnecessary.  Here are some
   different ways to say some of the things mentioned above: *)
fun is_large x = x > 37  (* The parens above were necessary because of ': int' *)
val is_sad = not has_something
val pays_same_rent = rent = 1300  (* Looks confusing, but works *)
val is_wrong_phone_no = phone_no <> 5551337
val negative_rent = ~rent  (* ~ rent (notice the space) would also work *)

(* Parentheses are mostly necessary when grouping things: *)
val some_answer = is_large (5 + 5)      (* Without parens, this would break! *)
(* val some_answer = is_large 5 + 5 *)  (* Read as: (is_large 5) + 5. Bad! *)


(* Besides booleans, ints and reals, Standard ML also has chars and strings: *)
val foo = "Hello, World!\n"  (* The \n is the escape sequence for linebreaks *)
val one_letter = #"a"        (* That funky syntax is just one character, a *)

val combined = "Hello " ^ "there, " ^ "fellow!\n"  (* Concatenate strings *)

val _ = print foo       (* You can print things. We are not interested in the *)
val _ = print combined  (* result of this computation, so we throw it away. *)
(* val _ = print one_letter *)  (* Only strings can be printed this way *)


val bar = [ #"H", #"e", #"l", #"l", #"o" ]  (* SML also has lists! *)
(* val _ = print bar *)  (* Lists are unfortunately not the same as strings *)

(* Fortunately they can be converted.  String is a library and implode and size
```

```
   are functions available in that library that take strings as argument. *)
val bob = String.implode bar          (* gives "Hello" *)
val bob_char_count = String.size bob  (* gives 5 *)
val _ = print (bob ^ "\n")            (* For good measure, add a linebreak *)

(* You can have lists of any kind *)
val numbers = [1, 3, 3, 7, 229, 230, 248]  (* : int list *)
val names = [ "Fred", "Jane", "Alice" ]    (* : string list *)

(* Even lists of lists of things *)
val groups = [ [ "Alice", "Bob" ],
               [ "Huey", "Dewey", "Louie" ],
               [ "Bonnie", "Clyde" ] ]     (* : string list list *)

val number_count = List.length numbers     (* gives 7 *)

(* You can put single values in front of lists of the same kind using
   the :: operator, called "the cons operator" (known from Lisp). *)
val more_numbers = 13 :: numbers  (* gives [13, 1, 3, 3, 7, ...] *)
val more_groups  = ["Batman","Superman"] :: groups

(* Lists of the same kind can be appended using the @ ("append") operator *)
val guest_list = [ "Mom", "Dad" ] @ [ "Aunt", "Uncle" ]

(* This could have been done with the "cons" operator.  It is tricky because the
   left-hand-side must be an element whereas the right-hand-side must be a list
   of those elements. *)
val guest_list = "Mom" :: "Dad" :: [ "Aunt", "Uncle" ]
val guest_list = "Mom" :: ("Dad" :: ("Aunt" :: ("Uncle" :: [])))

(* If you have many lists of the same kind, you can concatenate them all *)
val everyone = List.concat groups  (* [ "Alice", "Bob", "Huey", ... ] *)

(* A list can contain any (finite) number of values *)
val lots = [ 5, 5, 5, 6, 4, 5, 6, 5, 4, 5, 7, 3 ]  (* still just an int list *)

(* Lists can only contain one kind of thing... *)
(* val bad_list = [ 1, "Hello", 3.14159 ] : ??? list *)


(* Tuples, on the other hand, can contain a fixed number of different things *)
val person1 = ("Simon", 28, 3.14159)  (* : string * int * real *)

(* You can even have tuples inside lists and lists inside tuples *)
val likes = [ ("Alice", "ice cream"),
              ("Bob",   "hot dogs"),
              ("Bob",   "Alice") ]     (* : (string * string) list *)

val mixup = [ ("Alice", 39),
              ("Bob",   37),
              ("Eve",   41) ]  (* : (string * int) list *)

val good_bad_stuff =
  (["ice cream", "hot dogs", "chocolate"],
   ["liver", "paying the rent" ])           (* : string list * string list *)


(* Records are tuples with named slots *)

val rgb = { r=0.23, g=0.56, b=0.91 } (* : {b:real, g:real, r:real} *)

(* You don't need to declare their slots ahead of time. Records with
   different slot names are considered different types, even if their
   slot value types match up. For instance... *)

val Hsl = { H=310.3, s=0.51, l=0.23 } (* : {H:real, l:real, s:real} *)
val Hsv = { H=310.3, s=0.51, v=0.23 } (* : {H:real, s:real, v:real} *)

(* ...trying to evaluate `Hsv = Hsl` or `rgb = Hsl` would give a type
   error. While they're all three-slot records composed only of `real`s,
```

```
   they each have different names for at least some slots. *)

(* You can use hash notation to get values out of tuples. *)

val H = #H Hsv (* : real *)
val s = #s Hsl (* : real *)

(* Functions! *)
fun add_them (a, b) = a + b    (* A simple function that adds two numbers *)
val test_it = add_them (3, 4)  (* gives 7 *)

(* Larger functions are usually broken into several lines for readability *)
fun thermometer temp =
    if temp < 37
    then "Cold"
    else if temp > 37
         then "Warm"
         else "Normal"

val test_thermo = thermometer 40  (* gives "Warm" *)

(* if-sentences are actually expressions and not statements/declarations.
   A function body can only contain one expression.  There are some tricks
   for making a function do more than just one thing, though. *)

(* A function can call itself as part of its result (recursion!) *)
fun fibonacci n =
    if n = 0 then 0 else                   (* Base case *)
    if n = 1 then 1 else                   (* Base case *)
    fibonacci (n - 1) + fibonacci (n - 2)  (* Recursive case *)

(* Sometimes recursion is best understood by evaluating a function by hand:

 fibonacci 4
   ~> fibonacci (4 - 1) + fibonacci (4 - 2)
   ~> fibonacci 3 + fibonacci 2
   ~> (fibonacci (3 - 1) + fibonacci (3 - 2)) + fibonacci 2
   ~> (fibonacci 2 + fibonacci 1) + fibonacci 2
   ~> ((fibonacci (2 - 1) + fibonacci (2 - 2)) + fibonacci 1) + fibonacci 2
   ~> ((fibonacci 1 + fibonacci 0) + fibonacci 1) + fibonacci 2
   ~> ((1 + fibonacci 0) + fibonacci 1) + fibonacci 2
   ~> ((1 + 0) + fibonacci 1) + fibonacci 2
   ~> (1 + fibonacci 1) + fibonacci 2
   ~> (1 + 1) + fibonacci 2
   ~> 2 + fibonacci 2
   ~> 2 + (fibonacci (2 - 1) + fibonacci (2 - 2))
   ~> 2 + (fibonacci (2 - 1) + fibonacci (2 - 2))
   ~> 2 + (fibonacci 1 + fibonacci 0)
   ~> 2 + (1 + fibonacci 0)
   ~> 2 + (1 + 0)
   ~> 2 + 1
   ~> 3  which is the 4th Fibonacci number, according to this definition

 *)

(* A function cannot change the variables it can refer to.  It can only
   temporarily shadow them with new variables that have the same names.  In this
   sense, variables are really constants and only behave like variables when
   dealing with recursion.  For this reason, variables are also called value
   bindings. An example of this: *)

val x = 42
fun answer(question) =
    if question = "What is the meaning of life, the universe and everything?"
    then x
    else raise Fail "I'm an exception. Also, I don't know what the answer is."
val x = 43
val hmm = answer "What is the meaning of life, the universe and everything?"
(* Now, hmm has the value 42.  This is because the function answer refers to
   the copy of x that was visible before its own function definition. *)
```

```
(* Functions can take several arguments by taking one tuples as argument: *)
fun solve2 (a : real, b : real, c : real) =
    ( (~b + Math.sqrt(b * b - 4.0*a*c)) / (2.0 * a),
      (~b - Math.sqrt(b * b - 4.0*a*c)) / (2.0 * a) )

(* Sometimes, the same computation is carried out several times. It makes sense
   to save and re-use the result the first time. We can use "let-bindings": *)
fun solve2 (a : real, b : real, c : real) =
    let val discr  = b * b - 4.0*a*c
        val sqr = Math.sqrt discr
        val denom = 2.0 * a
    in ((~b + sqr) / denom,
        (~b - sqr) / denom) end


(* Pattern matching is a funky part of functional programming.  It is an
   alternative to if-sentences.  The fibonacci function can be rewritten: *)
fun fibonacci 0 = 0  (* Base case *)
  | fibonacci 1 = 1  (* Base case *)
  | fibonacci n = fibonacci (n - 1) + fibonacci (n - 2)  (* Recursive case *)

(* Pattern matching is also possible on composite types like tuples, lists and
   records. Writing "fun solve2 (a, b, c) = ..." is in fact a pattern match on
   the one three-tuple solve2 takes as argument. Similarly, but less intuitively,
   you can match on a list consisting of elements in it (from the beginning of
   the list only). *)
fun first_elem (x::xs) = x
fun second_elem (x::y::xs) = y
fun evenly_positioned_elems (odd::even::xs) = even::evenly_positioned_elems xs
  | evenly_positioned_elems [odd] = []  (* Base case: throw away *)
  | evenly_positioned_elems []    = []  (* Base case *)

(* When matching on records, you must use their slot names, and you must bind
   every slot in a record. The order of the slots doesn't matter though. *)

fun rgbToTup {r, g, b} = (r, g, b)    (* fn : {b:'a, g:'b, r:'c} -> 'c * 'b * 'a *)
fun mixRgbToTup {g, b, r} = (r, g, b) (* fn : {b:'a, g:'b, r:'c} -> 'c * 'b * 'a *)

(* If called with {r=0.1, g=0.2, b=0.3}, either of the above functions
   would return (0.1, 0.2, 0.3). But it would be a type error to call them
   with {r=0.1, g=0.2, b=0.3, a=0.4} *)

(* Higher order functions: Functions can take other functions as arguments.
   Functions are just other kinds of values, and functions don't need names
   to exist.  Functions without names are called "anonymous functions" or
   lambda expressions or closures (since they also have a lexical scope). *)
val is_large = (fn x => x > 37)
val add_them = fn (a,b) => a + b
val thermometer =
    fn temp => if temp < 37
               then "Cold"
               else if temp > 37
                    then "Warm"
                    else "Normal"

(* The following uses an anonymous function directly and gives "ColdWarm" *)
val some_result = (fn x => thermometer (x - 5) ^ thermometer (x + 5)) 37

(* Here is a higher-order function that works on lists (a list combinator) *)
val readings = [ 34, 39, 37, 38, 35, 36, 37, 37, 37 ]  (* first an int list *)
val opinions = List.map thermometer readings (* gives [ "Cold", "Warm", ... ] *)

(* And here is another one for filtering lists *)
val warm_readings = List.filter is_large readings  (* gives [39, 38] *)

(* You can create your own higher-order functions, too.  Functions can also take
   several arguments by "currying" them. Syntax-wise this means adding spaces
   between function arguments instead of commas and surrounding parentheses. *)
```

```
fun map f [] = []
  | map f (x::xs) = f(x) :: map f xs

(* map has type ('a -> 'b) -> 'a list -> 'b list and is called polymorphic. *)
(* 'a is called a type variable. *)


(* We can declare functions as infix *)
val plus = add_them   (* plus is now equal to the same function as add_them *)
infix plus            (* plus is now an infix operator *)
val seven = 2 plus 5  (* seven is now bound to 7 *)

(* Functions can also be made infix before they are declared *)
infix minus
fun x minus y = x - y (* It becomes a little hard to see what's the argument *)
val four = 8 minus 4  (* four is now bound to 4 *)

(* An infix function/operator can be made prefix with 'op' *)
val n = op + (5, 5)   (* n is now 10 *)

(* 'op' is useful when combined with high order functions because they expect
   functions and not operators as arguments. Most operators are really just
   infix functions. *)
val sum_of_numbers = foldl op+ 0 [1,2,3,4,5]


(* Datatypes are useful for creating both simple and complex structures *)
datatype color = Red | Green | Blue

(* Here is a function that takes one of these as argument *)
fun say(col) =
    if col = Red then "You are red!" else
    if col = Green then "You are green!" else
    if col = Blue then "You are blue!" else
    raise Fail "Unknown color"

val _ = print (say(Red) ^ "\n")

(* Datatypes are very often used in combination with pattern matching *)
fun say Red   = "You are red!"
  | say Green = "You are green!"
  | say Blue  = "You are blue!"
  | say _     = raise Fail "Unknown color"


(* Here is a binary tree datatype *)
datatype 'a btree = Leaf of 'a
                  | Node of 'a btree * 'a * 'a btree (* three-arg constructor *)

(* Here is a binary tree *)
val myTree = Node (Leaf 9, 8, Node (Leaf 3, 5, Leaf 7))

(* Drawing it, it might look something like...

           8
          / \
 leaf -> 9   5
            / \
   leaf -> 3   7 <- leaf
 *)

(* This function counts the sum of all the elements in a tree *)
fun count (Leaf n) = n
  | count (Node (leftTree, n, rightTree)) = count leftTree + n + count rightTree

val myTreeCount = count myTree  (* myTreeCount is now bound to 32 *)


(* Exceptions! *)
(* Exceptions can be raised/thrown using the reserved word 'raise' *)
```

```
fun calculate_interest(n) = if n < 0.0
                            then raise Domain
                            else n * 1.04

(* Exceptions can be caught using "handle" *)
val balance = calculate_interest ~180.0
              handle Domain => ~180.0    (* x now has the value ~180.0 *)

(* Some exceptions carry extra information with them *)
(* Here are some examples of built-in exceptions *)
fun failing_function []    = raise Empty  (* used for empty lists *)
  | failing_function [x]   = raise Fail "This list is too short!"
  | failing_function [x,y] = raise Overflow  (* used for arithmetic *)
  | failing_function xs    = raise Fail "This list is too long!"

(* We can pattern match in 'handle' to make sure
   a specfic exception was raised, or grab the message *)
val err_msg = failing_function [1,2] handle Fail _ => "Fail was raised"
                                          | Domain => "Domain was raised"
                                          | Empty  => "Empty was raised"
                                          | _      => "Unknown exception"

(* err_msg now has the value "Unknown exception" because Overflow isn't
   listed as one of the patterns -- thus, the catch-all pattern _ is used. *)

(* We can define our own exceptions like this *)
exception MyException
exception MyExceptionWithMessage of string
exception SyntaxError of string * (int * int)

(* File I/O! *)
(* Write a nice poem to a file *)
fun writePoem(filename) =
    let val file = TextIO.openOut(filename)
        val _ = TextIO.output(file, "Roses are red,\nViolets are blue.\n")
        val _ = TextIO.output(file, "I have a gun.\nGet in the van.\n")
    in TextIO.closeOut(file) end

(* Read a nice poem from a file into a list of strings *)
fun readPoem(filename) =
    let val file = TextIO.openIn filename
        val poem = TextIO.inputAll file
        val _ = TextIO.closeIn file
    in String.tokens (fn c => c = #"\n") poem
    end

val _ = writePoem "roses.txt"
val test_poem = readPoem "roses.txt"  (* gives [ "Roses are red,",
                                                 "Violets are blue.",
                                                 "I have a gun.",
                                                 "Get in the van." ] *)

(* We can create references to data which can be updated *)
val counter = ref 0 (* Produce a reference with the ref function *)

(* Assign to a reference with the assignment operator *)
fun set_five reference = reference := 5

(* Read a reference with the dereference operator *)
fun equals_five reference = !reference = 5

(* We can use while loops for when recursion is messy *)
fun decrement_to_zero r = if !r < 0
                          then r := 0
                          else while !r >= 0 do r := !r - 1
(* This returns the unit value (in practical terms, nothing, a 0-tuple) *)

(* To allow returning a value, we can use the semicolon to sequence evaluations *)
fun decrement_ret x y = (x := !x - 1; y)
```