The Basics of SML

Lars-Henrik Eriksson

Functional Programming 1

Presentation originally by Tjark Weber based on notes by Pierre Flener,
Jean-Noël Monette, Sven-Olof Nyström
Today: The Basics of SML

1. Types and type inference
2. Literals and built-in operators
3. Value declarations
4. Tuples
5. Functions
6. Specifying functions
7. Pattern matching
8. Local declarations
9. Infix operators
10. Side effects
11. Exceptions
12. Modules
Table of Contents

1. Types and type inference
2. Literals and built-in operators
3. Value declarations
4. Tuples
5. Functions
6. Specifying functions
7. Pattern matching
8. Local declarations
9. Infix operators
10. Side effects
11. Exceptions
12. Modules
SML is a typed language. Every expression has a type.

- **int**: integers
- **real**: real numbers
- **char**: characters
- **string**: character sequences
- **bool**: truth values (Booleans), i.e., true and false
- **unit**: only one possible value: ()
Compound Types

Certain expressions have a compound type. Compound types are built from basic types using type constructors (such as ->, *, list).

- Functions: e.g. int -> int
- Tuples: e.g. int * int
- Lists: e.g. int list

\[
> ([\text{abs}, \sim], ("cool", 3.5));
\]
\[
\textbf{val} \ \text{it} = ([\text{fn}, \text{fn}], ("cool", 3.5)):
\]
\[
\quad (\text{int} \to \text{int}) \ \text{list} \ast (\text{string} \ast \text{real})
\]

We will see later in the course that you can even declare your own data types.
Type Inference

- SML is *strongly typed*, meaning that all expressions have a well-defined type that can be determined statically (i.e., without running the program).
- It is—except in special situations—not necessary to declare the type of an expression.
- The compiler automatically infers the type of each expression.
- Functions can only be applied to type-correct arguments.

```sml
fun double x = 2 * x; (* infers type int -> int *)
double 3.0; (* Error—Type error in function application *)
2 * 3.0; (* Error—Type error in function application *)
```
Table of Contents

1 Types and type inference
2 Literals and built-in operators
3 Value declarations
4 Tuples
5 Functions
6 Specifying functions
7 Pattern matching
8 Local declarations
9 Infix operators
10 Side effects
11 Exceptions
12 Modules
Integer and Real Literals

Integers:
- An optional `~` for negative literals.
- In base 10: a sequence of (one or more) digits (0-9).
- In base 16: `0x`, followed by a sequence of digits (0-9, A-F).

Examples: 0, ~1, 42, ~0x2A.

Reals:
1. An optional `~` for negative literals.
2. A sequence of digits.
3. One or both of:
   - A period (.) followed by a sequence of digits.
   - E, optional `~`, followed by a sequence of digits.

Examples: 0.0, ~15.5E3, 15E~2
Built-in Operators

- SML has several built-in operators that work on the basic types.
- Several of them are overloaded for convenience, e.g.,
  
  \[
  2 + 3; \\
  2.0 + 3.0;
  \]

- There is no implicit conversion between types.
- Operators are simply a special case of functions.
# Operators on Integers

<table>
<thead>
<tr>
<th>op</th>
<th>type</th>
<th>form</th>
<th>precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>int × int → int</td>
<td>infix</td>
<td>6</td>
</tr>
<tr>
<td>−</td>
<td>int × int → int</td>
<td>infix</td>
<td>6</td>
</tr>
<tr>
<td>*</td>
<td>int × int → int</td>
<td>infix</td>
<td>7</td>
</tr>
<tr>
<td>div</td>
<td>int × int → int</td>
<td>infix</td>
<td>7</td>
</tr>
<tr>
<td>mod</td>
<td>int × int → int</td>
<td>infix</td>
<td>7</td>
</tr>
<tr>
<td>=</td>
<td>int × int → bool *</td>
<td>infix</td>
<td>4</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>int × int → bool *</td>
<td>infix</td>
<td>4</td>
</tr>
<tr>
<td>&lt;</td>
<td>int × int → bool</td>
<td>infix</td>
<td>4</td>
</tr>
<tr>
<td>&lt;=</td>
<td>int × int → bool</td>
<td>infix</td>
<td>4</td>
</tr>
<tr>
<td>&gt;</td>
<td>int × int → bool</td>
<td>infix</td>
<td>4</td>
</tr>
<tr>
<td>&gt;=</td>
<td>int × int → bool</td>
<td>infix</td>
<td>4</td>
</tr>
<tr>
<td>~</td>
<td>int → int</td>
<td>prefix</td>
<td></td>
</tr>
<tr>
<td>abs</td>
<td>int → int</td>
<td>prefix</td>
<td></td>
</tr>
</tbody>
</table>

Infix operators associate to the left. (* the exact type will be given later)
### Operators on Reals

<table>
<thead>
<tr>
<th>op</th>
<th>type</th>
<th>form</th>
<th>precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>real × real → real</td>
<td>infix</td>
<td>6</td>
</tr>
<tr>
<td>−</td>
<td>real × real → real</td>
<td>infix</td>
<td>6</td>
</tr>
<tr>
<td>×</td>
<td>real × real → real</td>
<td>infix</td>
<td>7</td>
</tr>
<tr>
<td>/</td>
<td>real × real → real</td>
<td>infix</td>
<td>7</td>
</tr>
<tr>
<td>&lt;, &lt;=</td>
<td>real × real → bool</td>
<td>infix</td>
<td>4</td>
</tr>
<tr>
<td>&gt;, &gt;=</td>
<td>real × real → bool</td>
<td>infix</td>
<td>4</td>
</tr>
<tr>
<td>~</td>
<td>real → real</td>
<td>prefix</td>
<td></td>
</tr>
<tr>
<td>abs</td>
<td>real → real</td>
<td>prefix</td>
<td></td>
</tr>
<tr>
<td>Math.sqrt</td>
<td>real → real</td>
<td>prefix</td>
<td></td>
</tr>
<tr>
<td>Math.ln</td>
<td>real → real</td>
<td>prefix</td>
<td></td>
</tr>
</tbody>
</table>

Infix operators associate to the left.

Note the absence of (in-)equality: there is no = or <> for reals!
Characters and Strings

- A *character* literal is written as the symbol `#` followed by the character enclosed in double-quotes `"`.
  
  Examples: `"a"`, `" "`, `"4"

- A *string* is a character sequence enclosed in double-quotes `"`.
  Example: "Hello! Goodbye!"

- Control characters can be included:
  - new-line: `\n`
  - double-quote: `\"`
  - backslash: `\\`
Operators on Characters and Strings

Let ‘strchar × strchar’ be ‘char × char’ or ‘string × string’.

\[
\begin{array}{|c|c|c|c|}
\hline
\text{op} & \text{type} & \text{form} & \text{precedence} \\
\hline
= & \text{strchar} \times \text{strchar} \rightarrow \text{bool} & \text{infix} & 4 \\
<> & \text{strchar} \times \text{strchar} \rightarrow \text{bool} & \text{infix} & 4 \\
< & \text{strchar} \times \text{strchar} \rightarrow \text{bool} & \text{infix} & 4 \\
\leq & \text{strchar} \times \text{strchar} \rightarrow \text{bool} & \text{infix} & 4 \\
> & \text{strchar} \times \text{strchar} \rightarrow \text{bool} & \text{infix} & 4 \\
\geq & \text{strchar} \times \text{strchar} \rightarrow \text{bool} & \text{infix} & 4 \\
^ & \text{string} \times \text{string} \rightarrow \text{string} & \text{infix} & 6 \\
\text{size} & \text{string} \rightarrow \text{int} & \text{prefix} & \\
\hline
\end{array}
\]

Infix operators associate to the left. (* the exact type will be given later)

Comparison of strings uses the \textit{lexicographic order}, according to the ASCII code of characters.
## Type Conversions

<table>
<thead>
<tr>
<th>op</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>real</td>
<td>int → real</td>
</tr>
<tr>
<td>ceil</td>
<td>real → int</td>
</tr>
<tr>
<td>floor</td>
<td>real → int</td>
</tr>
<tr>
<td>round</td>
<td>real → int</td>
</tr>
<tr>
<td>trunc</td>
<td>real → int</td>
</tr>
<tr>
<td>chr</td>
<td>int → char</td>
</tr>
<tr>
<td>ord</td>
<td>char → int</td>
</tr>
<tr>
<td>str</td>
<td>char → string</td>
</tr>
<tr>
<td>Int.toString</td>
<td>int → string</td>
</tr>
</tbody>
</table>

Conversions `chr` and `ord` use the ASCII code of characters.
Operators on Booleans

<table>
<thead>
<tr>
<th>op</th>
<th>type</th>
<th>form</th>
<th>precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>andalso</td>
<td>bool × bool → bool</td>
<td>infix</td>
<td>3</td>
</tr>
<tr>
<td>orelse</td>
<td>bool × bool → bool</td>
<td>infix</td>
<td>2</td>
</tr>
<tr>
<td>not</td>
<td>bool → bool</td>
<td>prefix</td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>bool × bool → bool</td>
<td>*</td>
<td>4</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>bool × bool → bool</td>
<td>*</td>
<td>4</td>
</tr>
</tbody>
</table>

Infix operators associate to the left. (* the exact type will be given later)

`andalso` and `orelse` are evaluated **lazily**, i.e., their second argument is evaluated only when the value of the first argument does not already determine the result.

Note that `and` and `or` are **not** Boolean operators!
Lazy Evaluation: Examples

```sml
> ( 34 < 649 ) orelse ( Math.Ln 12.4 * 3.4 > 12.0 ) ;  
val it = true : bool
```

The second operand is *not* evaluated because the first operand evaluates to true.

```sml
> ( 34 < 649 ) orelse ( 1 div 0 > 99 ) ;  
val it = true : bool
```

The second operand (*1 div 0 > 99*) is *not* evaluated, even though by itself it would lead to a runtime error:

```sml
> 1 div 0 > 99;  
! Uncaught exception: Div
```
if ... then ... else ...

if B then E1 else E2

- This is an expression in SML, not a control structure. Like any expression, it evaluates to a value.
- Typing rules:
  - B must be a Boolean expression.
  - The type of E1 and E2 must be the same.
  - The type of if B then E1 else E2 is the same as the type of E1.
- Evaluation rules:
  - B is evaluated first.
  - If B evaluates to true, E1 is evaluated.
  - If B evaluates to false, E2 is evaluated.
- There is no "if B then E" expression. What should be the value of the expression when B is false?!
- if-then-else has lower precedence than all other operators.
Exercises

1. Express the following expressions as if-then-else expressions:
   1. E orelse F
   2. E andalso F

2. Use step-by-step evaluation to evaluate the following expression:
   \[ \text{if } 1 + 2 < 4 \text{ then size ("hel" ^ "lo!") else 4 div 2} \]
Value declarations

Table of Contents

1. Types and type inference
2. Literals and built-in operators
3. Value declarations
4. Tuples
5. Functions
6. Specifying functions
7. Pattern matching
8. Local declarations
9. Infix operators
10. Side effects
11. Exceptions
12. Modules
**Value Declarations**

Value declarations associate a value to an identifier.

```sml
val a = 1;
val pi = 3.14159;
val two_pi = 2.0 * pi;
val &a!+< = 42;
```

two_pi;

```sml
a + a;
&a!+< div 7;
```

Note: value declarations are *not* expressions!
Identifiers: Syntax

Note that $3+\sim 2$ is different from $3 + \sim 2$! SML thinks that $+\sim$ is an (undeclared) identifier.
The execution of a declaration, say `val x = expr`, creates a binding: the identifier `x` is *bound* to the value of the expression `expr`.

A collection of bindings is called an environment.

```plaintext
> val sum = 24 ;
val sum = 24 : int
> val sum = 3.51 ;
val sum = 3.51 : real
```

Later declarations of the same identifier change the environment.

The identifier `it` is always bound to the value of the last unidentified expression that was evaluated.
Execution of Declarations

Declarations are executed from left to right:

```sml
> val a = 1 ;
val a = 1 : int
> val b = 2 ;
val b = 2 : int
> val a = a+b val b = a+b ;
val a = 3 : int
val b = 5 : int
```

Simultaneous execution is achieved with and:

```sml
> val a = 1 val b = 2 ;
val a = 1 : int
val b = 2 : int
> val a = a+b and b = a+b ;
val a = 3 : int
val b = 3 : int
```
Identifiers Are *Not* Variables

```
> val x = 10;
val x = 10 : int
> fun addX y = x+y;
val addX = fn : int -> int
> addX 5;
val it = 15 : int
> x = 100;
val it = false : bool
> val x = 100;
val x = 100 : int
> addX 5;
val it = 15 : int
```
# Table of Contents

1. Types and type inference
2. Literals and built-in operators
3. Value declarations
4. **Tuples**
5. Functions
6. Specifying functions
7. Pattern matching
8. Local declarations
9. Infix operators
10. Side effects
11. Exceptions
12. Modules
Tuples

Tuples group $n(\neq 1)$ values (of possibly different types) into $n$-tuples.

Syntax: e.g., (22>5, "abc", 123)

- Particular cases of $n$-tuples: pairs, triples, ...
- Careful: There are no 1-tuples in ML! (1) is just 1 in parentheses.
- Selector $#i$ returns the $i^{th}$ component of a tuple.
- It is possible to have tuples of tuples.
- The value () is the only 0-tuple. It has type unit.
Tuples: Examples

\[
\begin{align*}
\texttt{val it = (2.3, 5) : real } \ast \texttt{int} \\
\text{Here, } \ast \text{ is a type constructor. It denotes the Cartesian product of types.}
\end{align*}
\]

\[
\begin{align*}
\texttt{val bigTuple = ((2.3, 5), "two", (8, true))} ; \\
\texttt{val bigTuple = ((2.3, 5), "two", (8, true)) : (real \ast int) \ast string \ast (int \ast bool)} \\
\texttt{val it = (8, true) : int } \ast \texttt{bool} \\
\texttt{val it = 13 : int}
\end{align*}
\]
Functions

Function declaration:

```sml
fun double x = 2 * x;
fun even x = x mod 2 = 0;
val even = fn x => x mod 2 = 0; (* anonymous function *)
fun odd x = not (even x);
```

Function application:

```sml
> double 3;
val it = 6 : int
> even 17;
val it = false : bool
> odd 17 orelse even 17;
val it = true : bool
```
Function Application: Remarks

- Function application has the highest precedence.
- Parentheses are redundant: \( f(x) \) and \( f\ (x) \) and \( f\ x \) are the same. (The latter version is most idiomatic in SML.)
- Function application is left-associative: \( f\ x\ y \) is the same as \( (f\ x)\ y \)
Functions: Remarks

- Functions are values, just as integers, tuples, etc.
- They have a type (that can be inferred by the system).
- Any identifier can be bound to them.
- They can be arguments or return values of other functions.
Functions (cont.)

> **fun** even \( x \) = \( x \mod 2 = 0 \);

**val** even = **fn**: int \( \rightarrow \) bool

> **val** plop = even;

**val** plop = **fn**: int \( \rightarrow \) bool

> plop 3;

**val** it = false : bool

> (**fn** \( x \) \( \rightarrow \) \( x \mod 2 = 1 \)) 3;

**val** it = true : bool

> even 3 + 4;

**Error**: Type error in function application.

...
A function application $\text{exp1} \; \text{exp2}$ is evaluated from left to right:

1. $\text{exp1}$ is evaluated until it becomes (the name of a built-in function or) a value of the form $\text{fn} \; x \; => \; \text{body}$.
2. Then $\text{exp2}$ is evaluated to, say, $v$.
3. Now, $(\text{fn} \; x \; => \; \text{body}) \; v$ is reduced to $\text{body}'$, where $\text{body}'$ is $\text{body}$ with every free occurrence of $x$ replaced by $v$.
4. Evaluation continues until a value is obtained.
val even = fn x => x mod 2 = 0;
fun odd x = not (even x); (* val odd = fn x => not (even x) *)

> (if false then even else odd) (15 + 2);
  → odd (15 + 2);
  → (fn x => not (even x)) (15 + 2);
  → (fn x => not (even x)) 17;
  → not (even 17);
  → not ((fn x => x mod 2 = 0) 17);
  → not (17 mod 2 = 0);
  → not (1 = 0);
  → not false;
  → true;
Technically, in SML functions only have one parameter (and one result).

How can we implement, e.g., the mathematical function max(a, b)?

Two ways:

1. The parameter is a tuple (here: a pair).
   
   ```sml
   fun max (a,b) = if a > b then a else b;
   ```

2. Use curried functions.
   
   ```sml
   fun max a b = if a > b then a else b;
   ```

Does it look the same? Does that small difference matter?
Functions of Several Parameters

- Technically, in SML functions only have one parameter (and one result).
- How can we implement, e.g., the mathematical function max(a, b)?
- Two ways:
  - The parameter is a tuple (here: a pair).
    
    > fun max (a,b) = if a > b then a else b;
  - Use curried functions.
    
    > fun max a b = if a > b then a else b;
- Does it look the same? Does that small difference matter?
Functions of Several Parameters (cont.)

> fun max1 (a,b) = if a > b then a else b;
val max1 = fn: int * int -> int
> val max1 = fn (a,b) => if a > b then a else b;
val max1 = fn: int * int -> int
> fun max2 a b = if a > b then a else b;
val max2 = fn: int -> int -> int
> val max2 = fn a => fn b => if a > b then a else b;
val max2 = fn: int -> int -> int
> val posOrZero = max2 0;
val posOrZero = fn: int -> int
> posOrZero 3;
val it = 3: int
> posOrZero ~3;
val it = 0: int
Currying

There is a correspondence of the types of the following functions:

\[ f : A \times B \rightarrow C \]

\[ g : A \rightarrow (B \rightarrow C) \]

H.B. Curry (1958): \( f (a, b) = g a b \)

Currying = passing from the first form to the second form

Let \( a \) be an object of type \( A \), and \( b \) an object of type \( B \)

- \( f (a, b) \) is an object of type \( C \), the application of the function \( f \) to the pair \( (a, b) \)
- \( g a \) is an object of type \( B \rightarrow C \): \( g a \) is thus a function, the result of a function can thus also be a function!
- \( (g a) b \) is an object of type \( C \), the application of the function \( g a \) to \( b \)
- Attention: \( f (a, b) \) is different from \( f a b \)
Currying (cont.)

Every function on a Cartesian product can be curried:

\[ g : A_1 \times A_2 \times \cdots \times A_n \rightarrow C \]

\[ \downarrow \]

\[ g : A_1 \rightarrow (A_2 \rightarrow \cdots \rightarrow (A_n \rightarrow C)) \]
\[ g : A_1 \rightarrow A_2 \rightarrow \cdots \rightarrow A_n \rightarrow C \]

The symbol \( \rightarrow \) associates to the right.

Usefulness of currying:

- The rice tastes better . . .
- Partial application of a function for getting other functions
- Easier design and usage of higher-order functions (functions with functional arguments)
Currying: Examples

```sml
> fun greet word name = word ^ "", " " ^ name ^ "!";
val greet = fn : string -> string -> string
> val greetEng = greet "Hello";
val greetEng = fn : string -> string
> val greetSwe = greet "Hej";
val greetSwe = fn : string -> string
> greetEng "Tjark";
val it = "Hello, Tjark!": string
> greetSwe "Kjell";
val it = "Hej, Kjell!": string
```
Functions can return tuples when several results are needed.
Functions can take or return the unit argument: `fun bof () = ();`
The type of functions can be *polymorphic*:

```
> fun id x = x;
val id = fn : 'a -> 'a
```

The type variable `’a` can be instantiated to any type:

```
> id 5;
val it = 5: int
> id 3.5;
val it = 3.5: real
```
Polymorphism Limitations

- When an arithmetic operator is encountered, if the type is not determined by another mean, the operator refers to integers.

```sml
> fun sqr x = x * x;
val sqr = fn : int -> int
```

- Expressions can be typed explicitly.

```sml
> fun sqr x = (x: real) * x;
val sqr = fn : real -> real;
> fun sqr (x: real) = x * x;
val sqr = fn : real -> real;
> fun sqr x: real = x * x; (* (sqr x): real *)
val sqr = fn : real -> real;
```

- * is both type constructor and arithmetic op. Check the precedence...

```sml
> fun sqr x = x: real * x;
Error - Type constructor (x) has not been declared
    Found near x : real * x
```
There is a complication with type variables. Fortunately the compiler will warn you about it.

```sml
> fun id x = x;
val id = fn: 'a -> 'a
> val iidd = id id;
```

Warning—The type of (iidd) contains a free type variable.

- Setting it to a unique monotype.
```sml
val iidd = fn: _a -> _a
> iidd 1;
```

Error—Type error in function application.

- Function: iidd : _a -> _a
- Argument: 1 : int
- Reason:
  - Can’t unify int (**In Basis**) with
    - _a (**Constructed from a free type variable.**) (Different type constructors)
```

Lars-Henrik Eriksson (UU)
The Basics of SML
Specifying Functions

- **Function name** and **argument**
- **Type** of the function: types of the argument and result (can be inferred by the compiler)
- **Pre-condition** on the argument:
  - If the pre-condition does not hold, then the function *may return any* result!
  - If the pre-condition does hold, then the function *must* return a result satisfying the post-condition!
- **Post-condition** on the result: its description and meaning
- **Side effects** (if any): printing of the result, . . .
- **Examples** and **counter-examples** (if useful)
Specification: Example

(* triangle n
  TYPE: int -> int
  PRE: n >= 0
  POST: \sum_{0 \leq i \leq n}(i)
*)

fun triangle n = ...

Beware: The post-condition and side effects should usually involve all the components of the argument.
Role of well-chosen examples and counter-examples

In theory, they are redundant with the pre/post-conditions.
In practice:

- They often provide an intuitive understanding that no assertion or definition could achieve
- They often help eliminate risks of ambiguity in the pre/post-conditions by illustrating delicate issues
- If they contradict the pre/post-conditions, then we know that something is wrong somewhere!

\[
(*) \quad \text{floor } x
\]

\begin{align*}
\text{TYPE: } & \text{real } \rightarrow \text{int} \\
\text{PRE: } & \text{true} \\
\text{POST: } & \text{the largest integer } m \text{ such that } m \leq x \\
\text{EXAMPLES: } & \text{floor } 23.65 = 23, \text{ floor } \sim 23.65 = \sim 24 \\
\text{COUNTER-EXAMPLE: } & \text{floor } \sim 23.65 \not= \sim 23
\end{align*}

\[
(*)
\]

\text{fun floor x = ...}
# Table of Contents

1. Types and type inference
2. Literals and built-in operators
3. Value declarations
4. Tuples
5. Functions
6. Specifying functions
7. **Pattern matching**
8. Local declarations
9. Infix operators
10. Side effects
11. Exceptions
12. Modules
Pattern Matching

> val x = (18, true);
val x = (18, true): int * bool
> val (n, b) = x;
val n = 18: int
val b = true: bool

- The left-hand side of a value declaration is called a pattern.
- The value on the right must respect that pattern.
- An identifier can match anything.
- _ matches anything and has no name.

> val (n, _ ) = x;
val n = 18: int
> val (18, b) = x;
val b = true;
> val (17, b) = x;
Exception—Bind raised
Pattern Matching: as

- `as` introduces a pattern alias for identifiers.

```sml
val t = ((3.5, true), 4);
val t = ((3.5, true), 4): (real * bool) * int
val (d as (a,b), c) = t;
val a = 3.5: real
val b = true: bool
val c = 4: int
val d = (3.5, true): real * bool
val (d,c) = t;
val ((a,b),c) = t;
val s as (d,c) = t;
val s as u as v = t;
val (t,d) = t; (* t is bound to a different value after this *)
```
Motivation: Conditional Computation

Often, functions need to perform different computations, based on the values of their arguments.

We have already seen one way of doing this in SML: conditional expressions.

```sml
fun sign x = 
  if x = 0 then 0 else if x < 0 then ~1 else 1
```

When there are many different cases to consider, we need many nested if-then-else expressions. These can be hard to read.
Function Declarations with Clauses

A case distinction that tests whether the function argument is equal to a constant can be written more elegantly:

```
fun sign 0 = 0
   | sign x = if x < 0 then ~1 else 1
```

Here, 0 and \( x \) (on the left) are called patterns. Each line of the function declaration is called a clause.
fun sign 0 = 0
  | sign x = if x < 0 then ~1 else 1

When `sign` is called with an argument, SML evaluates only the first clause where the pattern matches the actual argument.

A constant matches only itself. An identifier matches any value.
Clause Order is Important

The order of clauses is important! Consider

```ml
fun sign x = if x < 0 then ~1 else 1
    | sign 0 = 0
```

Now `sign 0` → ...?!
The underscore _ can be used as a pattern when we do not care about the value of the argument. Like an identifier pattern, it matches any value. But the underscore creates no binding.

Example:

```sml
fun is_zero 0 = true
  | is_zero x = false
```
The underscore _ can be used as a pattern when we do not care about the value of the argument. Like an identifier pattern, it matches any value. But the underscore creates no binding.

Example:

```sml
fun is_zero 0 = true
| is_zero _ = false
```
Beware: Redundant Patterns and Typos

Some datatypes have values that look just like identifiers: e.g, true and false are (the only) values of type bool.

If we match against such values, a small typo can result in a syntactically correct program with very different semantics!

```
fun bool_string true = "true" |
    bool_string false = "false"

fun bool_string' truw = "true" |
    bool_string' false = "false"
```

```
bool_string true → "true"
bool_string false → "false"

bool_string' true → "true"
bool_string' false → "true"
```

Watch out for “Pattern is redundant” warnings from Poly/ML!
(Non-)Exhaustive Matching

It is (possible, but) usually a bad idea to provide clauses only for some of the possible argument values.

```ml
> fun is_zero 0 = true;
Warning-Matches are not exhaustive. ...
val is_zero = fn: int -> bool
> is_zero 0;
val it = true: bool
> is_zero 1;
Exception- Match raised
```

Poly/ML will generate a warning for the declaration, and a runtime error when the function is called with an argument that does not match any given pattern.
“Catch-All” Clauses

It is easy to avoid non-exhaustive matches. One can simply specify a final clause that matches any value.

Example:

```sml
fun is_zero 0 = true
| is_zero _ = false
```
General Form of a Function Declaration

\[
\text{fun name pattern}_1 = \text{expression}_1 \\
\quad \mid \text{name pattern}_2 = \text{expression}_2 \\
\quad \ldots \\
\quad \mid \text{name pattern}_N = \text{expression}_N
\]

Every **pattern** is a constant, identifier, underscore (\_\_) or a “skeleton” for a datatype (e.g., tuples) where the skeleton components are patterns. Note that real constants and function applications are not patterns.

All patterns in a function declaration must have the same type (namely the argument type of the function).

All expressions on the right-hand side of a function declaration must have the same type (namely the return type of the function).
Patterns in SML must be **linear**: each identifier can occur at most once.

**Linear:**

```ml
> fun equal (x, y, z) = x = y andalso y = z;
```

**Not linear:**

```ml
> fun equal (x, x, x) = true
   | equal _ = false;
Error-x has already been bound in this clause. ...
```
Case Analysis

The case expression allows to match an expression against several patterns.

```plaintext
case Expr of
    Pat1 => Expr1
    Pat2 => Expr2
    ...
    PatN => ExprN
```

- case ... of ... is an expression.
- Expr1, ..., ExprN must be of the same type.
- Expr, Pat1, ..., PatN must be of the same type.
- Patterns are tested in order. If Pati is matched, then only Expri is evaluated (lazy evaluation).
Case Analysis (cont.)

Our earlier remarks about redundant and non-exhaustive patterns in function clauses equally apply to case expressions.

\[
\text{\texttt{case \ 17 \ mod \ 2 \ of}} \\
\quad 0 \ \Rightarrow \ "\text{even}" \\
\quad | \ 1 \ \Rightarrow \ "\text{odd}";
\]

Warning—Matches are not exhaustive.

\texttt{val \ it = "odd": string}

\[
\text{\texttt{case \ 17 \ mod \ 2 \ of}} \\
\quad 0 \ \Rightarrow \ "\text{even}" \\
\quad | \ _ \ \Rightarrow \ "\text{odd}";
\]

\texttt{val \ it = "odd": string}
Pattern Matching and Shadowing

When a pattern matches a value (and only then), identifiers in the pattern are bound to corresponding parts of the matching value. These bindings may cause shadowing.

Example:

```sml
fun multiply (x, y) =
  case x * y of
    1 => "one"
  | 2 => "two"
  | x => Int.toString x
```

What is the value of `multiply (3, 4)`?
Exercises

- How to express if-then-else as a case-of expression?

- Write a function of two integer arguments that returns the number of arguments that are equal to zero.
# Local declarations

## Table of Contents

1. Types and type inference
2. Literals and built-in operators
3. Value declarations
4. Tuples
5. Functions
6. Specifying functions
7. Pattern matching
8. Local declarations
9. Infix operators
10. Side effects
11. Exceptions
12. Modules
Local Declarations

It is possible to declare values locally with an expression of the form

\[
\text{let } declarations \text{ in expression end}
\]

The values declared between \texttt{let} and \texttt{in} are bound only in the following expression (up to \texttt{end}).

\[
> \text{let val } x = 1 \text{ in } x + 10 \text{ end;}
\]

\[
\text{val it = 11: int}
\]

Error − Value or constructor (x) has not been declared ...
Local Declarations

It is possible to declare values locally with an expression of the form

```
let declarations in expression end
```

The values declared between `let` and `in` are bound only in the following expression (up to `end`).

```
> let
  val x = 1
  in
  x + 10
end;
val it = 11: int
> x;
Error−Value or constructor (x) has not been declared ...
```
Local declarations **shadow** any other declaration of the same name, i.e., they render it inaccessible (within the scope of the local declaration):

```
> val x = 0;
> val y =
   let
     val x = 1
   in
     x + 10
   end;
> (x, y);
```

val it = (0, 11): int * int
Local declarations **shadow** any other declaration of the same name, i.e., they render it inaccessible (within the scope of the local declaration):

```sml
> val x = 0;
> val y =
  let
    val x = 1
  in
    x + 10
  end;
> (x, y);
val it = (0, 11): int * int;
```
Evaluation of Local Declarations

As with other declarations, the expression to which the identifier is bound is evaluated (exactly) once, no matter how often the identifier is then used.

Example:

```sml
fun discount unit_price quantity = 
  let
    val price = unit_price * real quantity
  in
    if price < 100.0 then price else 0.95 * price
  end
```

When the function `discount` is applied to arguments, `unit_price * real quantity` is computed once (rather than three times).
Local Function Declarations

Functions may be declared locally.

Example:

```sml
fun is_leap_year year = let
  fun is_divisible b = year mod b = 0
  in
    is_divisible 4 andalso
    (not ( is_divisible 100) orelse is_divisible 400)
  end
```
# Table of Contents

1. Types and type inference
2. Literals and built-in operators
3. Value declarations
4. Tuples
5. Functions
6. Specifying functions
7. Pattern matching
8. Local declarations
9. **Infix operators**
10. Side effects
11. Exceptions
12. Modules
New Infix Operators

You may declare your own infix operators in SML.

Suppose \( f \) is a function (whose argument must be a pair).\(^1\)

- `infix n f` makes \( f \) a left-assoc. infix operator with precedence \( n \).
- `rinfix n f` does the same but with right association.
- `nonfix f` returns to ordinary prefix notation for \( f \).

```ml
> fun x (a,b) = a*b;
> infix 5 x;
> 2 * 4;
val it = 8: int
> nonfix x;
> x (2,4);
val it = 8: int
```

\(^1\)There is no curried form of infix operators in SML.
Using Infix Operators as Values

To refer to an infix operator (without necessarily applying it to some arguments), prefix the name of the operator with the keyword `op`.

```ml
> 2 + 4;
val it = 6 : int

> op +;
val it = fn : int * int -> int

> (op +) (2,4);
val it = 6 : int
```
Table of Contents

1. Types and type inference
2. Literals and built-in operators
3. Value declarations
4. Tuples
5. Functions
6. Specifying functions
7. Pattern matching
8. Local declarations
9. Infix operators
10. Side effects
11. Exceptions
12. Modules
Side Effects: The print Function

Like most functional languages, SML has *some* functions with side effects, e.g., for input/output.

```ml
(* print s
   TYPE: string -> unit
   PRE: true
   POST: ()
   SIDE-EFFECTS: s is printed to stdout
   *)
```
The print Function: Example

> fun welcome name = print ("Hello, " ^ name ^ "!

val welcome = fn : string -> unit
> welcome "world";
Hello, world!
val it = () : unit
Sequential composition is an expression of the form

\[( \text{Expr}_1 ; \text{Expr}_2 ; \ldots ; \text{Expr}_n ) \]

To evaluate this expression, first \( \text{Expr}_1 \) is evaluated; then \( \text{Expr}_2 \) is evaluated; \ldots The value of the expression is the value of \( \text{Expr}_n \).

```ml
> (1; 2.0; "three");
val it = "three": string
```

Sequential composition is useful with expressions that have side effects.
# Table of Contents

1. Types and type inference
2. Literals and built-in operators
3. Value declarations
4. Tuples
5. Functions
6. Specifying functions
7. Pattern matching
8. Local declarations
9. Infix operators
10. Side effects
11. Exceptions
12. Modules
Exceptions

Execution may interrupted immediately upon a runtime error.

> 1 div 0;
Exception— Div raised

Exceptions can be handled:

> 1 div 0 handle Div => 42;
val it = 42: int

You can declare and raise your own exceptions:

exception errorDiv ;
fun safeDiv a b =
  if b = 0 then raise errorDiv
  else a div b;
# Table of Contents

1. Types and type inference  
2. Literals and built-in operators  
3. Value declarations  
4. Tuples  
5. Functions  
6. Specifying functions  
7. Pattern matching  
8. Local declarations  
9. Infix operators  
10. Side effects  
11. Exceptions  
12. Modules
Modules

- Modules group related functionalities together.
- SML defines a Basis Library with standard modules: see http://sml-family.org/Basis/
- A function declared in a module can be accessed by typing the name of the module, a dot (.), and the name of the function.
- Some functions are also available at the top-level.

  Int.toString
  Int.+
  Int.abs
  Real.Math.sqrt
  ...