First Look at ML

San Skulrattanakulchai

Feb 22, 2018

ML Features

- The first ML compiler was built in 1974.

ML Features

- The first ML compiler was built in 1974.
- The dialect of ML we will be studying is called Standard ML (SML), defined in 1997.

ML Features

- The first ML compiler was built in 1974.
- The dialect of ML we will be studying is called Standard ML (SML), defined in 1997.
- SML is a high-level language. It has "automatic garbage collection." It supports **functional programming**. It provides mutable variables and arrays for fast execution. It also provides **modules** for structuring large systems.

ML Features

- The first ML compiler was built in 1974.
- The dialect of ML we will be studying is called Standard ML (SML), defined in 1997.
- SML is a high-level language. It has "automatic garbage collection." It supports **functional programming**. It provides mutable variables and arrays for fast execution. It also provides **modules** for structuring large systems.
- ML protects the programmers from their own errors. The compiler checks every program to make sure it's **type safe**. An ML program cannot crash! It may quit and report errors but it cannot crash.

Imperative vs Functional

- An imperative language like Fortran or C centers around "commands" while a functional programming languge like ML centers around "expressions."

Imperative vs Functional

- An imperative language like Fortran or C centers around "commands" while a functional programming languge like ML centers around "expressions."
- A command has side effects. In fact, an imperative language relies heavily on the side effect of changing the values of memory locations.

Imperative vs Functional

- An imperative language like Fortran or C centers around "commands" while a functional programming languge like ML centers around "expressions."
- A command has side effects. In fact, an imperative language relies heavily on the side effect of changing the values of memory locations.
- On the other hand, a functional language works with "expressions" that can be reasoned with using mathematics.

Imperative vs Functional

- An imperative language like Fortran or C centers around "commands" while a functional programming languge like ML centers around "expressions."
- A command has side effects. In fact, an imperative language relies heavily on the side effect of changing the values of memory locations.
- On the other hand, a functional language works with "expressions" that can be reasoned with using mathematics.
- It has the property of "referential transparency," meaning that equals can be substituted for equals without changing the meaning of the expression.

Working from the command line

- We'll be using Standard ML of New Jersey (SMLNJ).

Working from the command line

- We'll be using Standard ML of New Jersey (SMLNJ).
- Add line

  `export PATH=/opt/local/bin/:$PATH`

  in your `~/.bash_profile` file.

Working from the command line

- We'll be using Standard ML of New Jersey (SMLNJ).
- Add line

  `export PATH=/opt/local/bin/:$PATH`

  in your `~/.bash_profile` file.
- Logout, then relogin to the shell. Type

  `rlwrap sml`

Working from the command line

- We'll be using Standard ML of New Jersey (SMLNJ).
- Add line

  `export PATH=/opt/local/bin/:$PATH`

  in your `~/.bash_profile` file.
- Logout, then relogin to the shell. Type

  `rlwrap sml`
- Explain about prompt. Type in something like `3 * 4` without `;` Explain secondary prompt. Explain the output, `it`, and "type annotation."

Constants (Literals)

- int: 3, 4, ~4

Constants (Literals)

- int: `3`, `4`, `~4`
- real: `4.57`, `1.5e-9`

Constants (Literals)

- int: `3`, `4`, `~4`
- real: `4.57`, `1.5e-9`
- bool: `true`, `false`
  (names in ML are case-sentitive, `True`, `False` not the same as `true`, `false`)

Constants (Literals)

- int: `3`, `4`, `~4`
- real: `4.57`, `1.5e-9`
- bool: `true`, `false`
  (names in ML are case-sentitive, `True`, `False` not the same as `true`, `false`)
- string:

Constants (Literals)

- int: `3`, `4`, `~4`
- real: `4.57`, `1.5e-9`
- bool: `true`, `false`
  (names in ML are case-sentitive, `True`, `False` not the same as `true`, `false`)
- string:
  - `"metalanguage"`,

Constants (Literals)

- int: `3`, `4`, `~4`
- real: `4.57`, `1.5e-9`
- bool: `true`, `false`
  (names in ML are case-sentitive, `True`, `False` not the same as `true`, `false`)
- string:
  - `"metalanguage"`,
  - `"hello\nworld"`,

Constants (Literals)

- int: `3`, `4`, `~4`
- real: `4.57`, `1.5e-9`
- bool: `true`, `false`
  (names in ML are case-sentitive, `True`, `False` not the same as `true`, `false`)
- string:
  - `"metalanguage"`,
  - `"hello\nworld"`,
  - `"\t1\t2\t3"`,

Constants (Literals)

- int: `3`, `4`, `~4`
- real: `4.57`, `1.5e-9`
- bool: `true`, `false`
  (names in ML are case-sentitive, `True`, `False` not the same as `true`, `false`)
- string:
  - `"metalanguage"`,
  - `"hello\nworld"`,
  - `"\t1\t2\t3"`,
  - `"double \"quote\" ok"`

Constants (Literals)

- int: `3`, `4`, `~4`
- real: `4.57`, `1.5e-9`
- bool: `true`, `false`
  (names in ML are case-sentitive, `True`, `False` not the same as `true`, `false`)
- string:
  - `"metalanguage"`,
  - `"hello\nworld"`,
  - `"\t1\t2\t3"`,
  - `"double \"quote\" ok"`
- char: `#"a"`

Operators

- int operators: `~ + - * div mod`

Operators

- int operators: `~ + - * div mod`
- real operators: `~ + - * /` (mention "overloaded" operators)

Operators

- int operators: `~ + - * div mod`
- real operators: `~ + - * /` (mention "overloaded" operators)
- string concatenation: `^`

Operators

- int operators: `~ + - * div mod`
- real operators: `~ + - * /` (mention "overloaded" operators)
- string concatenation: `^`
- `size "hello";`

Operators

- int operators: `~ + - * div mod`
- real operators: `~ + - * /` (mention "overloaded" operators)
- string concatenation: `^`
- `size "hello";`
- comparison operators: `< > <= >=` for the types

Operators

- int operators: `~ + - * div mod`
- real operators: `~ + - * /` (mention "overloaded" operators)
- string concatenation: `^`
- `size "hello";`
- comparison operators: `< > <= >=` for the types
  - string

Operators

- int operators: `~ + - * div mod`
- real operators: `~ + - * /` (mention "overloaded" operators)
- string concatenation: `^`
- `size "hello";`
- comparison operators: `< > <= >=` for the types
  - string
  - char

Operators

- int operators: `~ + - * div mod`
- real operators: `~ + - * /` (mention "overloaded" operators)
- string concatenation: `^`
- `size "hello";`
- comparison operators: `< > <= >=` for the types
  - string
  - char
  - int

Operators

- int operators: `~ + - * div mod`
- real operators: `~ + - * /` (mention "overloaded" operators)
- string concatenation: `^`
- `size "hello";`
- comparison operators: `< > <= >=` for the types
  - string
  - char
  - int
  - real

Operators

- int operators: `~ + - * div mod`
- real operators: `~ + - * /` (mention "overloaded" operators)
- string concatenation: `^`
- `size "hello";`
- comparison operators: `< > <= >=` for the types
  - string
  - char
  - int
  - real
- String comparison is alphabetic (dictionary) order.

Operators

- int operators: `~ + - * div mod`
- real operators: `~ + - * /` (mention "overloaded" operators)
- string concatenation: `^`
- `size "hello";`
- comparison operators: `< > <= >=` for the types
  - string
  - char
  - int
  - real
- String comparison is alphabetic (dictionary) order.
- (in)equality:

Operators

- int operators: `~ + - * div mod`
- real operators: `~ + - * /` (mention "overloaded" operators)
- string concatenation: `^`
- `size "hello";`
- comparison operators: `< > <= >=` for the types
  - string
  - char
  - int
  - real
- String comparison is alphabetic (dictionary) order.
- (in)equality:
  - values of "equality type" can be compared using = and <>

Operators

- int operators: `~ + - * div mod`
- real operators: `~ + - * /` (mention "overloaded" operators)
- string concatenation: `^`
- `size "hello";`
- comparison operators: `< > <= >=` for the types
  - string
  - char
  - int
  - real
- String comparison is alphabetic (dictionary) order.
- (in)equality:
  - values of "equality type" can be compared using = and <>
  - but `real` is not an equality type

Operators

- int operators: `~ + - * div mod`
- real operators: `~ + - * /` (mention "overloaded" operators)
- string concatenation: `^`
- `size "hello";`
- comparison operators: `< > <= >=` for the types
  - string
  - char
  - int
  - real
- String comparison is alphabetic (dictionary) order.
- (in)equality:
  - values of "equality type" can be compared using = and <>
  - but `real` is not an equality type
  - `bool`, `string`, `char`, and `int` are equality type

Operators

- int operators: `~ + - * div mod`
- real operators: `~ + - * /` (mention "overloaded" operators)
- string concatenation: `^`
- `size "hello";`
- comparison operators: `< > <= >=` for the types
  - string
  - char
  - int
  - real
- String comparison is alphabetic (dictionary) order.
- (in)equality:
  - values of "equality type" can be compared using = and <>
  - but `real` is not an equality type
  - `bool`, `string`, `char`, and `int` are equality type

Basic, continued

- conditional:
  `if` … `then` … `else` … as an expression

Basic, continued

- conditional:
  `if` … `then` … `else` … as an expression
- ML won't do implicit type conversion: `3 * 4.5` is an error

Basic, continued

- conditional:
  `if` … `then` … `else` … as an expression
- ML won't do implicit type conversion: `3 * 4.5` is an error
- In such a case, have to use conversion function: `real(12);`
  `floor(3.5); ceil(7.6); round(3.2); trunc(7.1);`
  `ord(#"a"); chr(66); str(#"a");`

Basic, continued

- conditional:
  `if` … `then` … `else` … as an expression
- ML won't do implicit type conversion: `3 * 4.5` is an error
- In such a case, have to use conversion function: `real(12); floor(3.5); ceil(7.6); round(3.2); trunc(7.1); ord(#"a"); chr(66); str(#"a");`
- The argument of a function call doesn't need surrounding parentheses!

Basic, continued

- conditional:
  `if` … `then` … `else` … as an expression
- ML won't do implicit type conversion: `3 * 4.5` is an error
- In such a case, have to use conversion function: `real(12); floor(3.5); ceil(7.6); round(3.2); trunc(7.1); ord(#"a"); chr(66); str(#"a");`
- The argument of a function call doesn't need surrounding parentheses!
- `f a` is good enough, but can also say `f(a)`, `(f a)`, `(f) a`, `(f)(a)`. However, have to say `f(a+1)` because *function application* binds tighter than +

Basic, continued

- conditional:
  `if` … `then` … `else` … as an expression
- ML won't do implicit type conversion: `3 * 4.5` is an error
- In such a case, have to use conversion function: `real(12); floor(3.5); ceil(7.6); round(3.2); trunc(7.1); ord(#"a"); chr(66); str(#"a");`
- The argument of a function call doesn't need surrounding parentheses!
- `f a` is good enough, but can also say `f(a)`, `(f a)`, `(f) a`, `(f)(a)`. However, have to say `f(a+1)` because *function application* binds tighter than +
- Also, function application is left-associative, so `f g a` means `(f g) a`

Variable Definition

- variable definition:

```
val x = 3;
val y = if x = 7 then 1.0 else 2.0;
```

Variable Definition

- variable definition:

```
val x = 3;
val y = if x = 7 then 1.0 else 2.0;
```

- One can say

```
val r = 3.1;
val r = "rstring";
```

but these two `r`'s are different `r`'s!.

Tuples

- A tuple is an ordered sequence of values.

Tuples

- A tuple is an ordered sequence of values.
- Components of a tuple can be of different types, e.g.,

```
val vector1 = (3.5, 4.2);
val aTup = ("yes", 3, (4.5, #"a"));
```

Tuples

- A tuple is an ordered sequence of values.
- Components of a tuple can be of different types, e.g.,

```
val vector1 = (3.5, 4.2);
val aTup = ("yes", 3, (4.5, #"a"));
```

- `#n` is the tuple component access function

```
#2 aTup;
#1 vector1;
```

Tuples

- A tuple is an ordered sequence of values.
- Components of a tuple can be of different types, e.g.,

```
val vector1 = (3.5, 4.2);
val aTup = ("yes", 3, (4.5, #"a"));
```

- `#n` is the tuple component access function

```
#2 aTup;
#1 vector1;
```

- There's no tuple of length 1, so, for example, the expression `(3)` has type `int`.

Tuples

- A tuple is an ordered sequence of values.
- Components of a tuple can be of different types, e.g.,

```
val vector1 = (3.5, 4.2);
val aTup = ("yes", 3, (4.5, #"a"));
```

- `#n` is the tuple component access function

```
#2 aTup;
#1 vector1;
```

- There's no tuple of length 1, so, for example, the expression `(3)` has type `int`.
- However, there's something that looks like a tuple of length 0. It's called a **unit**. Its usefulness is related to functions, which will be described shortly.

Lists

- A list is also an ordered sequence of values, but all its elements must be of the same type.

```
[1, 2];
[3.1, 2.0, 3.4];
["hi", "ho"];
[(1,2), (3,4)];
[[1], [2,3,4]]
nil;
[];
```

Lists

- A list is also an ordered sequence of values, but all its elements must be of the same type.

```
[1, 2];
[3.1, 2.0, 3.4];
["hi", "ho"];
[(1,2), (3,4)];
[[1], [2,3,4]]
nil;
[];
```

- To test if a list is empty, use `null`

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

List continued

- list concatenation operator `@`:

  ```
  [1,2] @ [3, 4, 5]
  ```

  This "cons operator" is right-associative, thus, `1 :: 2 :: [3, 4]` gives `[1, 2, 3, 4]` as expected.

List continued

- list concatenation operator `@`:

  ```
  [1,2] @ [3, 4, 5]
  ```

  This "cons operator" is right-associative, thus, `1 :: 2 :: [3, 4]` gives `[1, 2, 3, 4]` as expected.

- `hd` and `tl` operators give the head and tail of the list, respectively.

  ```
  hd [1, 2, 3];
  tl [1, 2, 3];
  ```

List continued

- list concatenation operator `@`:

```
[1,2] @ [3, 4, 5]
```

  This "cons operator" is right-associative, thus, `1 :: 2 :: [3, 4]` gives `[1, 2, 3, 4]` as expected.

- `hd` and `tl` operators give the head and tail of the list, respectively.

```
hd [1, 2, 3];
tl [1, 2, 3];
```

- conversion functions:

```
explode "hello";
implode [#"a", #"b"]
```

Function continued

- function definition:

```
<fun-def> ::= fun <fn-name> [<param>] = <expression>
```

Function continued

- function definition:

  ```
  <fun-def> ::= fun <fn-name> [<param>] = <expression>
  ```

- The literal `()` is called a **unit**. The term `unit` also denotes a type that has `()` as its only value.

Function continued

- function definition:

  `<fun-def> ::= fun <fn-name> [<param>] = <expression>`

- The literal `()` is called a **unit**. The term `unit` also denotes a type that has `()` as its only value.
- Note that every ML function takes exactly one parameter. If you want zero parameter, use `()`. If you want more than 1 parameter, use a tuple.

Function continued

- function definition:

  `<fun-def> ::= fun <fn-name> [<param>] = <expression>`

- The literal `()` is called a **unit**. The term `unit` also denotes a type that has `()` as its only value.
- Note that every ML function takes exactly one parameter. If you want zero parameter, use `()`. If you want more than 1 parameter, use a tuple.
- Every function application also has "return value" which is the value of the function call expression. Even a function that works by creating a side effect only like `print` has a value—its value is unit.

Introduction to Types

- `->`, `*`, and `list` are the 3 type constructors we learned in this chapter (ordered from lowest to highest precedence).

Introduction to Types

- `->`, `*`, and `list` are the 3 type constructors we learned in this chapter (ordered from lowest to highest precedence).
- Type variables

  ```
  'a 'b ...
  "a "b ...
  ```

  and `polytype`.