查询L04B供应商
捷多邦，专业PCB打样工厂，24小时加急出货

Processadors de Llenguatge II

Functional Paradigm II

Pratt A.7

Robert Harper’s SML tutorial (Sec II)

Rafael Ramirez
Dep Tecnologia
Universitat Pompeu Fabra

User-Defined Types

- How to **define** the new type.

- The **principle** in the use of data types (**constructing** via the data constructors, and **deconstructing** via pattern matching) is the **same** **principle** that you learnt in lists.

维库一下 www.dzsc.com

User-Defined Types

In SML,

- the `list` is a predefined **type constructor**.
- The `::` and `[]` are predefined **data constructors**.
- The value `1::(2::[])` has a type of `int list`. Which can be depicted conceptually as

1
2

- Functions on list, deconstruct it by pattern matching…

```
fun len [] = 0
|   len (x::xs) = 1 + len xs
```

The function **len** pattern-matches by checking the data constructor first. Then it will know what things are stored.

If it's a '::' then it knows it can take out a head and tail.

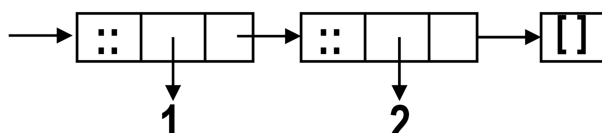
If it's a '[ ]', then it knows that there is no data stored there.

维库一下 www.dzsc.com

User-Defined Types

In SML

- the `list` is a predefined **type constructor**.
- The `::` and `[]` are predefined **data constructors**.
- The value `1::(2::[])` has a type of `int list`. Which can be depicted conceptually as



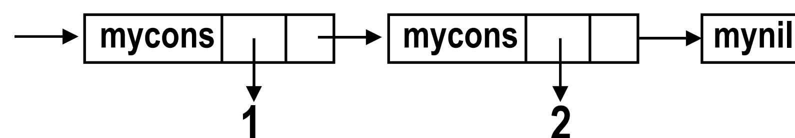
- Functions on list, deconstruct it by pattern matching…

```
fun len [] = 0
|   len (x::xs) = 1 + len xs
```

You can define your own type:

```
datatype 'a mylist
  = mynil
  | mycons of 'a * ('a mylist)
```

- Now, `mylist` is a user-defined **type constructor**.
- The `mycons` and `mynil` are the **data constructors**.
- The value `mycons(1,mycons(2,mynil))` has a type of `int mylist`. Which can be depicted conceptually as



- Functions on list, deconstruct it by pattern matching…

```
fun mylen mynil = 0
|   mylen (mycons(x,xs))
      = 1 + mylen xs
```

找芯片 维库一下 www.dzsc.com

User-Defined Types

```
datatype 'a mylist
      = mynil
      | mycons of 'a * ('a mylist)

datatype myintlist
      = intnil
      | intcons of int * myintlist
```

In defining a new datatype, you define

- A new type or a new type constructor.
- One or more data constructors.

找芯片 维库一下 www.dzsc.com

User-Defined Types

```
datatype `a mylist
      = mynil
      | mycons of `a * (`a mylist)

datatype myintlist
      = intnil
      | intcons of int * myintlist
```

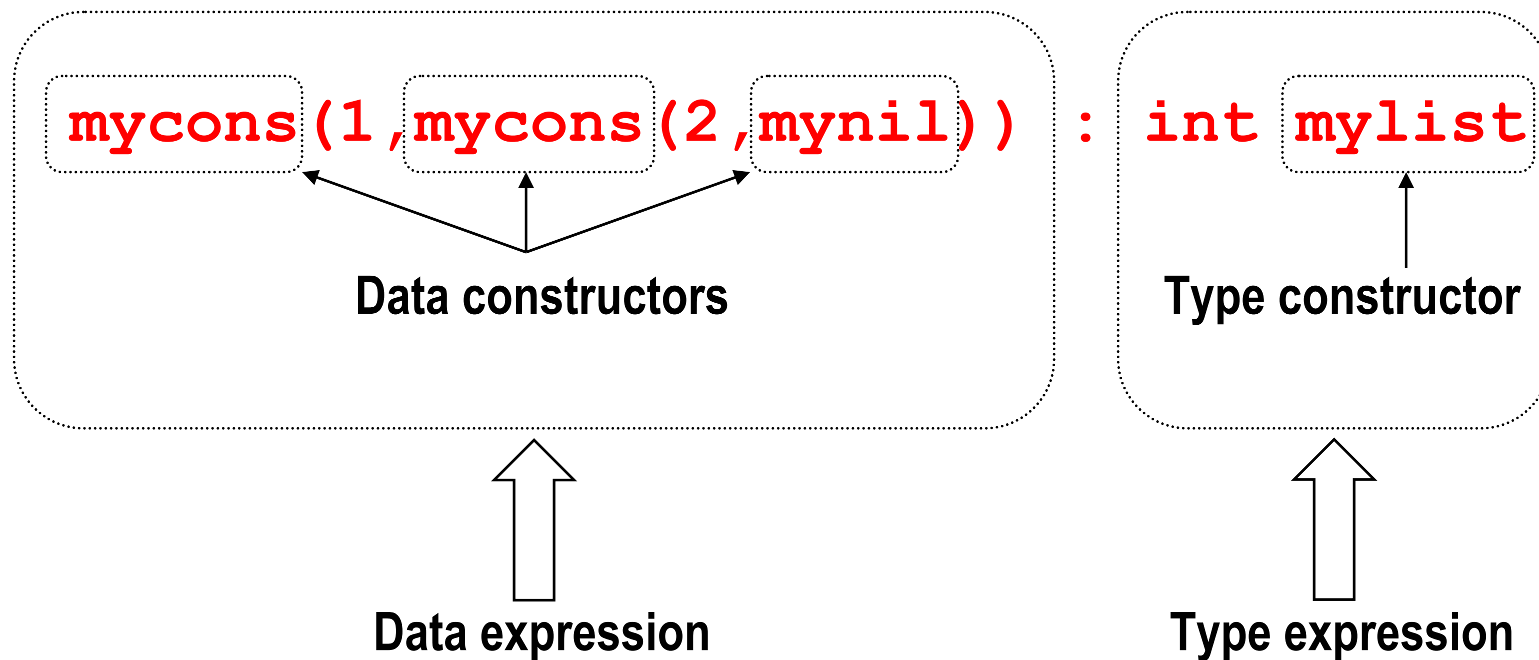
In defining a new datatype, you define

- A new type or a new type constructor.
- One or more data constructors.

找芯片 维库一下 www.dzsc.com

User-Defined Types

```
datatype 'a mylist
      = mynil
      | mycons of 'a * ('a mylist)
```



维库一下 www.dzsc.com

User-Defined Types

```
datatype 'a mylist
     = mynil
     | mycons of 'a * ('a mylist)
```

```
mycons(1,mycons(2,mynil)) : int mylist
mycons(1.1,mycons(2.2,mynil)) : real mylist
mycons("a",mycons("b",mynil)):string mylist
```

This is an example of **PARAMETRIC POLYMORPHISM**.

It is when the type takes in another type as parameter.

找芯片 维库一下 www.dzsc.com

User-Defined Types

```
datatype 'a mylist
      = mynil
      | mycons of 'a * ('a mylist)
```

In general, the BNF for datatype declaration is:

```
<DataDecl> ::= datatype [<tyvar>] <tyname> =
                         <dataAlt> {'|' <dataAlt>}

<dataAlt>  ::= <Constructor>
             |  <Constructor> of <type> {* <type>}

<Constructor> ::= <Identifier>
<tyname>      ::= <Identifier>
<tyvar>       ::= <Identifier>
```

维库一下 www.dzsc.com

User-Defined Types – Binary Tree

Defining an integer binary tree:

```
datatype IntTree
    = Leaf of int
    | Node of IntTree * int * IntTree
```

```
<DataDecl> ::= datatype [<tyvar>] <tyname> =
                         <dataAlt> {'|' <dataAlt>}

<dataAlt>   ::= <Constructor>
              |  <Constructor> of <type> {* <type>}

<Constructor> ::= <Identifier>
<tyname>      ::= <Identifier>
<tyvar>       ::= <Identifier>
```

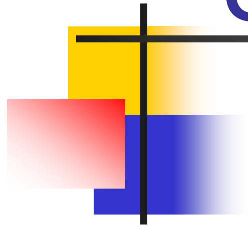
找芯片 维库一下 www.dzsc.com

User-Defined Types – Binary Tree

- Defining an integer binary tree:

```
datatype IntTree
     = Leaf of int
     | Node of IntTree * int * IntTree
```

- The above will declare
  - A new type name : `IntTree`
  - 2 new data constructors : `Leaf`, `Node`.



User-Defined Types – Binary Tree

Defining an integer binary tree:

```
datatype IntTree
     = Leaf of int
     | Node of IntTree * int * IntTree
```

Constructing a tree using the user-defined data constructors:

- **`Leaf 26`**
- **`Node (Leaf 26, 30, Leaf 10)`**
- **`Node (Leaf 26, 30, Node ( Leaf 10 , 3 , Leaf 4))`**

All will be of type IntTree

找芯片 维库一下 www.dzsc.com

User-Defined Types – Binary Tree

- Defining an integer binary tree:

```
datatype IntTree
    = Leaf of int
    | Node of IntTree * int * IntTree
```

Deconstructing a tree through pattern matching:

```
fun bsearch (Leaf y) y = y
|   bsearch (Leaf _) _ = ~1
|   bsearch (Node (l,x,r)) y =
    if (y = x) then x  (* return the value *)
    else if (y < x) then bsearch l y
    else bsearch r y;
```

找芯片 维库一下 www.dzsc.com

User Defined DataTypes

Parametric Polymorphism

- Defining a binary tree of some type:

```
datatype 'a tree
    = Leaf of 'a
    | Node of ('a tree) * 'a * ('a tree)
```

- Now we can have trees containing any type of object, but all objects stored must be the same type.

找芯片 维库一下 www.dzsc.com

User Defined DataTypes

Parametric Polymorphism

- Defining a binary tree of some type:

```
datatype 'a tree
      = Leaf of 'a
      | Node of ('a tree) * 'a * ('a tree)
```

Constructing a tree using the user-defined data constructors:

- `Leaf 1.33 : real tree`
- `Leaf 1 : int tree`
- `Node (Leaf 26, 30, Leaf 10) : int tree`
- `Node (Leaf [1], [2,3], Leaf [4]) : int list tree`

找芯片 维库一下 www.dzsc.com

User Defined DataTypes

Parametric Polymorphism

- Defining a binary tree of some type:

```
datatype 'a tree
    = Leaf of 'a
    | Node of ('a tree) * 'a * ('a tree)
```

Deconstructing a tree through pattern-matching:

```
fun height (Leaf _) = 1
|   height (Node (l,_,r)) = 1 + max(height l, height r)


height (Node (Node (Leaf 1,10,Leaf 2), 20 (Leaf 3)))
```

Will evaluate to ?.

找芯片 维库一下 www.dzsc.com

User Defined DataTypes

Parametric Polymorphism

- Defining a binary tree of some type:

```
datatype 'a tree
    = Leaf of 'a
    | Node of ('a tree) * 'a * ('a tree)
```

Deconstructing a tree through pattern-matching:

```
fun height (Leaf _) = 1
|   height (Node (l,_,r)) = 1 + max(height l, height r)
```

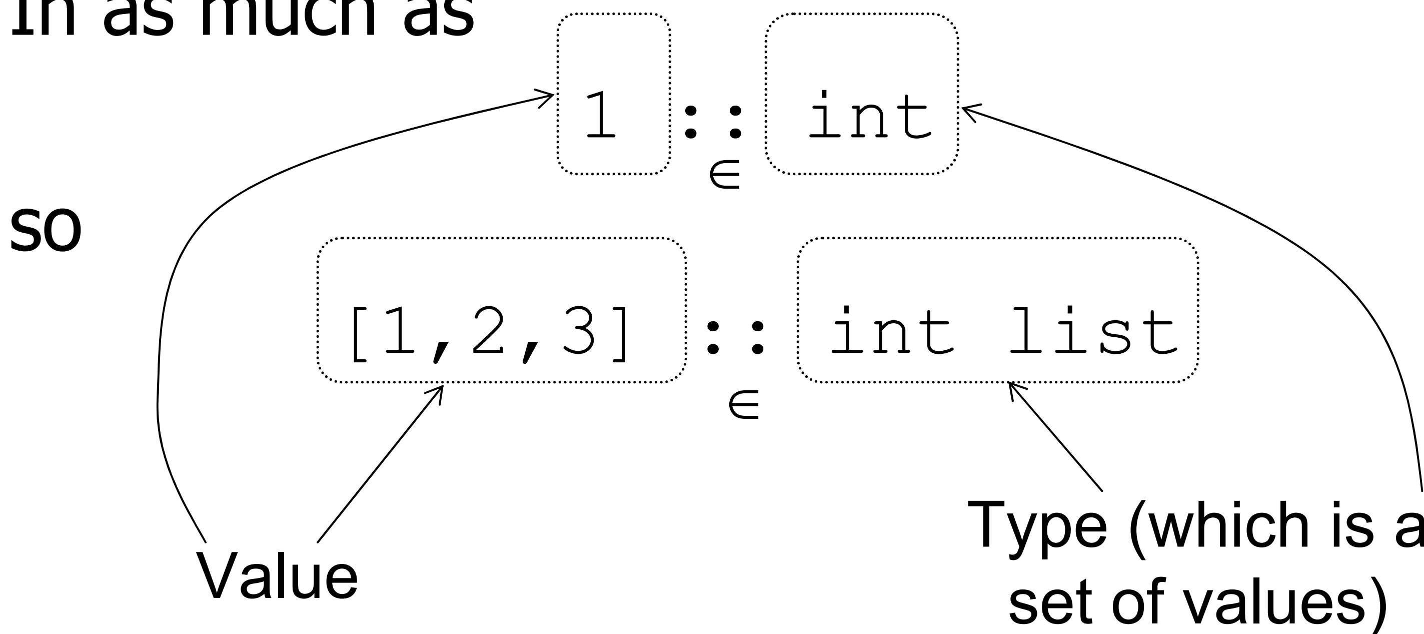
```
height (Node (Node (Leaf 1,10,Leaf 2), 20 (Leaf 3)))
```

Will evaluate to 3.

找芯片 维库一下 www.dzsc.com

Types in ML

- **Every expression will evaluate to a value.** By the word "value", we do not merely restrict ourselves to numbers, characters or booleans. For example, a list `[1,2,3]` is also considered as a value.
- In as much as



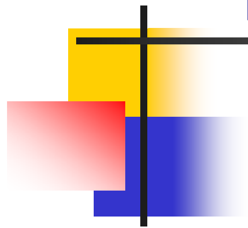
找芯片 维库一下 www.dzsc.com

Types in ML

- **Every value has a type.**

  Therefore every expression written in ML "can be assigned a type" – which is the type of the value it will evaluate to.

- If the expression that is written "has a type" or "can be assigned a type", we say that:
  - "The expression is **well-typed**"; or
  - "The expression is **typable** (by the ML type system)".



Types in ML – Type Check vs Infer

Type checking vs Type inferencing

- In most programming languages, we declare the type of an identifier, and the type system **CHECKS** that the program is type-safe.
- In ML, (most of the time) we need not declare the type of an identifier / expression, because the type system **INFERS** the type of the expression in the process of checking for type-safety.

找芯片 维库一下 www.dzsc.com

Types in ML – Decliaring Exp Type

- However, if you **insist on declaring the type of each expression**, you may do so.

```
<Exp> ::= <Constant> [: <type>]
        |  <Exp1> <op> <Exp2> [: <type>]
        |  if <Exp0> then <Exp1> else <Exp2> [: <type>]
        |  let {<Decl>} in <Exp> end [: <type>]
        |  <Exp1> <Exp2> [: <type>]
        |  fn <param> => <Exp> [: <type>]
        |  <TupleExp> [: <type>]
        |  <ListExp> [: <type>]
        |  <Identifier> [: <type>]
```

找芯片 维库一下 www.dzsc.com

Types in ML – BNF for types

<*type*> ::= **int** | **real** | **bool** | **char** | **string**

- Primitive types names
  - You have seen integers, reals and booleans.
  - Here are examples of `char` and `string` types.

```
- #"a";
val it = #"a" : char
- #"a" : char;
val it = #"a" : char
- [#"a" , #"b" , #"c"];
val it = [#"a",#"b",#"c"] : char list
- "a";
val it = "a" : string
- "a":string;
val it = "a" : string
- ["a","b","c"];
val it = ["a","b","c"] : string list
```

找芯片 维库一下 www.dzsc.com

Types in ML – BNF for types

```
<type> ::= int | real | bool | char | string
         | <type> -> <type>
```

- Function Types
  - Here are some examples:

```
- (fn x => x + 1) : int -> int;
val it = fn : int -> int
- fun add x y z = x + y + z;
val add = fn : int -> int -> int -> int
- val add2 = (add 0) : int -> int -> int;
val add2 = fn : int -> int -> int
- val add3 = (add 5 6) : int -> int;
val add3 = fn : int -> int
```

找芯片 维库一下 www.dzsc.com

Types in ML – BNF for types

```
<type> ::= int | real | bool | char | string
         |  <type> -> <type>
         |  <type> * <type> { * <type> }
```

- Tuples (product types)
  - Here are some examples:

```
- (1,2) : int * int;
val it = (1,2) : int * int
- (1.2,3) : real * int;
val it = (1.2,3) : real * int
- ((#"a",2) , 3.3 , ("abc",true)) : ((char * int)
* real * (string * bool));
val it = ((#"a",2),3.3,("abc",true)) : (char *
int) * real * (string * bool)
```

找芯片 维库一下 www.dzsc.com

Types in ML – BNF for types

```
<type> ::= int | real | bool | char | string
         |  <type> -> <type>
         |  <type> * <type> { * <type> }
         |  <typename>
```

- (User-Defined) Type names:
  - Shown earlier with user-defined **mylist** and **intTree**.

```
datatype IntTree
       = Leaf of int
       | Node of IntTree * int * IntTree
```

找芯片 维库一下 www.dzsc.com

Types in ML – BNF for types

```
<type> ::= int | real | bool | char | string
         |  <type> -> <type>
         |  <type> * <type> { * <type> }
         |  <typename>
         |  <type> <typecons>
         |  <typevar>
```

- Type constructors and type variables
  - Examples: Shown earlier with `'a mylist`.
  - Here `'a` is a type variable, and `mylist` is a type constructor,

找芯片 维库一下 www.dzsc.com

Curried vs Uncurried Functions

- How many arguments does this function take?

```
fun f(x,y) = x + 2 * y ;
```

- Ans : 1 argument, which is pattern-matched to a pair.

```
val f = fn : int * int -> int
```

找芯片 维库一下 www.dzsc.com

Curried vs Uncurried Functions

- How many arguments does this function take?

```
fun f x y = x + 2 * y ;
```

- Look at the type of the function:

```
val f = fn : int -> int -> int
```

Note that

1. the -> associates to the right. So the type is:

```
val f = fn : int -> (int -> int)
```

2. Application associates to the left. So

```
f 1 2 = ((f 1) 2)
```

维库一下 www.dzsc.com

Curried vs Uncurried Functions

- So from the type what can you deduce? How many arguments?

```
val f = fn : int -> (int -> int)
```

- If you mean in the immediate sense, ONE argument is taken in by f.
  - `f` a function which takes in an integer and returns a function.
- But if you mean ‘eventually, when fully-applied, how many arguments does it take in’, then f takes in a total of TWO arguments.
  - `f` takes in an integer (1st argument) which returns a function, which will take in another integer (2nd argument), and return an integer.

找芯片 维库一下 www.dzsc.com

Curried vs Uncurried Functions

How many arguments does the function really take?

```
fun f (x,y) = x + 2 * y ;
```

pattern 1: a pair

Ans: 1 argument which is pattern-matched to a pair

```
fun f x y = x + 2 * y ;
```

pattern 1 (x)

pattern 2 (y)

Ans: Eventually, when fully-applied, 2 arguments

找芯片 维库一下 www.dzsc.com

Curried vs Uncurried Functions

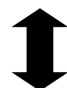
```
fun f (x,y) = x + 2 * y ;
```

UNCURRIED FUNCTION

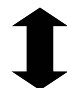
You can convert any uncurried function to a curried version. This process is known as currying. 'curry' is named after Haskell B. Curry

```
fun f x y = x + 2 * y ;
```

CURRIED FUNCTION



```
fun f x = (fn y => x + 2 * y);
```



```
val f = (fn x => (fn y => x + 2 * y));
```

The same

找芯片 维库一下 www.dzsc.com

Curried vs Uncurried Functions

- **Curried functions** provide extra **flexibility** to the language because it **enables partial application** of a function.

```
fun twice f x = f (f x);
let
    val inc2 = twice (fn x => x + x)
in  (inc2 1) + (inc2 2)
end;

val it = 12 ;
```

找芯片 维库一下 www.dzsc.com

Curried vs Uncurried Functions

- **Curried functions** provide extra **flexibility** to the language because it **enables partial application** of a function.

```
fun add_ver1 (x,y) = x+y; (* Uncurried *)
fun add_ver2 x y   = x+y; (* Curried *)

map (add_ver1 10) [1,2,3,4,5] (* will not work *)
map (add_ver2 10) [1,2,3,4,5] (* this will work*)
```

找芯片 维库一下 www.dzsc.com