Defining New Types

- New names for old types:

  type <*identifiers*> **=** <*type expression*>

- Parametrized type definitions:

type(<*list of type parameters*>**)**<*identifiers*>**=**<*type expression*>

- ML uses **'a**, **'b**, and so on for type variables

```
- type ('s , 'i) mapping = ('s * 'i) list;
type ('a,'b) mapping = ('a * 'b) list
- val words = [("a", 1)];
val words = [("a",1)] : (string * int) list
- val words = [("a", 1)] : (string, int) mapping;
val words = [("a",1)] : (string,int) mapping
```

Type Definitions

- Predefined, but not primitive in ML:

```
datatype bool = true | false;
```

- Type constructor for lists:

```
datatype 'element list = nil |
   :: of 'element * 'element list
```

- Defined for ML *in ML*

Example

```
- datatype day = Mon | Tue | Wed | Thu | Fri | Sat | Sun;
datatype day = Fri | Mon | Sat | Sun | Thu | Tue | Wed
- fun isWeekDay x = not (x = Sat orelse x = Sun);
val isWeekDay = fn : day -> bool
- isWeekDay Mon;
val it = true : bool
- isWeekDay Sat;
val it = false : bool
```

- **day** is the new type constructor and **Mon**, **Tue**, etc. are the new data constructors
- Why "constructors"? In a moment we will see how both can have parameters…

No Parameters

```
- datatype day = Mon | Tue | Wed | Thu | Fri | Sat | Sun;
datatype day = Fri | Mon | Sat | Sun | Thu | Tue | Wed
```

- The type constructor **day** takes no parameters: it is not polymorphic, there is only one **day** type
- The data constructors **Mon**, **Tue**, etc. take no parameters: they are constant values of the **day** type
- Capitalize the names of data constructors

Data Constructors In Patterns

```
fun isWeekDay Sat = false
|   isWeekDay Sun = false
|   isWeekDay _ = true;
```

- You can use the data constructors in patterns
- In this simple case, they are like constants
- But we will see more general cases next

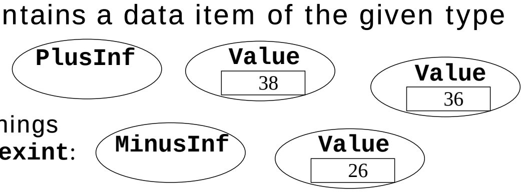
Wrappers

- You can add a parameter of any type to a data constructor, using the keyword **of**:

```
datatype exint = Value of int | PlusInf |
MinusInf;
```

- In effect, such a constructor is a wrapper that contains a data item of the given type

Some things of type **exint**:



*Value Constructor

```
- datatype exint = Value of int | PlusInf | MinusInf;
datatype exint = MinusInf | PlusInf | Value of int
- PlusInf;
val it = PlusInf : exint
- MinusInf;
val it = MinusInf : exint
- Value;
val it = fn : int -> exint
- Value 3;
val it = Value 3 : exint
```

- **`Value`** is a data constructor that takes a parameter: the value of the **`int`** to store
- It looks like a function that takes an **`int`** and returns an **`exint`** containing that **`int`**

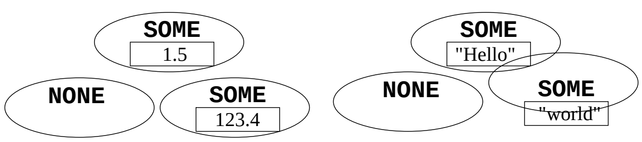
*An Example

```
- val x = Value 5;
val x = Value 5 : exint
- x+x;
Error: overloaded variable not defined at type
  symbol: +
  type: exint
```

- **`Value 5`** is an **`exint`**
- It is not an **`int`**, though it contains one
- How can we get the **`int`** out again?
- By pattern matching...

Type Constructors With Parameters

- Type constructors can also use parameters:
  `datatype 'a option = NONE | SOME of 'a;`
- The parameters of a type constructor are type variables, which are used in the data constructors
- The result: a new polymorphic type



Values of type **`real option`**

Values of type **`string option`**

Parameter Before Name

```
- SOME 4;
val it = SOME 4 : int option
- SOME 1.2;
val it = SOME 1.2 : real option
- SOME "pig";
val it = SOME "pig" : string option
```

- Type constuctor parameter comes before the type constructor name:
  `datatype 'a option = NONE | SOME of 'a;`
- We have types **`'a option`** and **`int option`**, just like **`'a list`** and **`int list`**

Uses for **`option`**

- Predefined type constructor in ML
- Used by predefined functions (or your own) when the result is not always defined

```
- fun optdiv a b =
=   if b = 0 then NONE else SOME (a div b);
val optdiv = fn : int -> int -> int option
- optdiv 7 2;
val it = SOME 3 : int option
- optdiv 7 0;
val it = NONE : int option
```

Longer Example: **`bunch`**

```
datatype 'x bunch =
   One of 'x |
   Group of 'x list;
```

- An **`'x bunch`** is either a thing of type **`'x`**, or a list of things of type **`'x`**
- As usual, ML infers types:

```
- One 1.0;
val it = One 1.0 : real bunch
- Group [true,false];
val it = Group [true,false] : bool bunch
```

Example: Polymorphism

```
- fun size (One _) = 1
= |   size (Group x) = length x;
val size = fn : 'a bunch -> int
- size (One 1.0);
val it = 1 : int
- size (Group [true,false]);
val it = 2 : int
```

- ML can infer **bunch** types, but does not always have to resolve them, just as with **list** types

Example: No Polymorphism

```
- fun sum (One x) = x
= |   sum (Group xlist) = foldr op + 0 xlist;
val sum = fn : int bunch -> int
- sum (One 5);
val it = 5 : int
- sum (Group [1,2,3]);
val it = 6 : int
```

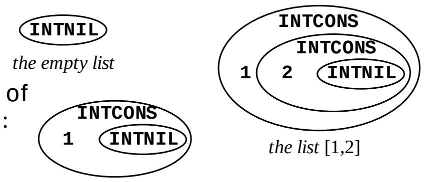
- We applied the **+** operator (through **foldr**) to the list elements
- So ML knows the parameter type must be **int bunch**

Recursively Defined Type Constructors

- The type constructor being defined may be used in its own data constructors:

```
datatype intlist =
  INTNIL |
  INTCONS of int * intlist;
```

Some values of type **intlist**:



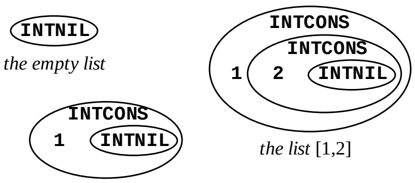
the empty list

the list [1]

the list [1,2]

Constructing Those Values

```
- INTNIL;
val it = INTNIL : intlist
- INTCONS (1,INTNIL);
val it = INTCONS (1,INTNIL) : intlist
- INTCONS (1,INTCONS(2,INTNIL));
val it = INTCONS (1,INTCONS (2,INTNIL)) : intlist
```



the empty list

the list [1]

the list [1,2]

An `intlist` Length Function

```
fun intlistLength INTNIL = 0
  | intlistLength (INTCONS(_,tail)) =
      1 + (intListLength tail);

fun listLength nil = 0
  | listLength (_::tail) =
      1 + (listLength tail);
```

- A length function
- Much like you would write for native lists
- Except, of course, that native lists are not always lists of integers…

Parametric List Type

```
datatype 'element mylist =
  NIL |
  CONS of 'element * 'element mylist;
```

- A parametric list type, almost like the predefined **list**
- ML handles type inference in the usual way:

```
- CONS(1.0, NIL);
val it = CONS (1.0,NIL) : real mylist
- CONS(1, CONS(2, NIL));
val it = CONS (1,CONS (2,NIL)) : int mylist
```

Some **mylist** Functions

```
fun myListLength NIL = 0
  | myListLength (CONS(_,tail)) =
      1 + myListLength(tail);

fun addup NIL = 0
  | addup (CONS(head,tail)) =
      head + addup tail;
```

- This now works almost exactly like the predefined **list** type constructor
- Of course, to add up a list you would use **foldr**...

Defining Operators (A Peek)

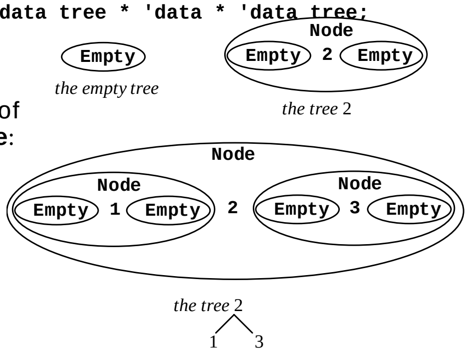
- ML allows new operators to be defined
- Like this:

```
- infixr 5 CONS;
infixr 5 CONS
- 1 CONS 2 CONS NIL;
val it = 1 CONS 2 CONS NIL : int mylist
```

Polymorphic Binary Tree

```
datatype 'data tree =
     Empty |
     Node of 'data tree * 'data * 'data tree;
```

Some values of type **int tree**:



Constructing Those Values

```
- val treeEmpty = Empty;
val treeEmpty = Empty : 'a tree
- val tree2 = Node(Empty,2,Empty);
val tree2 = Node (Empty,2,Empty) : int tree
- val tree123 = Node(Node(Empty,1,Empty),
=                    2,
=                    Node(Empty,3,Empty));
```

Increment All Elements

```
fun incall Empty = Empty
  | incall (Node(x,y,z)) =
      Node(incall x, y+1, incall z);
```

```
- incall tree123;
val it = Node (Node (Empty,2,Empty),
               3,
               Node (Empty,4,Empty)) : int tree
```

Add Up The Elements

```
fun sumall Empty = 0
  | sumall (Node(x,y,z)) =
      sumall x + y + sumall z;
```

```
- sumall tree123;
val it = 6 : int
```

Tree Traversals

- Three orders
  - Preorder
  - Inorder
  - Postorder

```
fun listall Empty = nil
    | listall (Node(x,y,z)) =
        listall x @ y :: listall z;
```

```
- listall tree123;
val it = [1,2,3] : int list
```

Tree Search

```
fun isintree x Empty = false
  | isintree x (Node(left,y,right)) =
      x=y
      orelse isintree x left
      orelse isintree x right;
```

```
- isintree 4 tree123;
val it = false : bool
- isintree 3 tree123;
val it = true : bool
```

Polymorphic Functions

- ML is strongly typed
  - It is possible to determine the type of any variable or the value returned by any function by examining the program but without running the program
- Polymorphic functions
  - However, in ML we can define functions whose type are partially or completely flexible
  - **‘a list**
  - Polymorphism (poly=many; morph=form)
    - The ability of a function to allow arguments of different types

Polymorphism

- One extreme application: identity function

```
- fun identity (x) = x;
val identity = fn : 'a -> 'a
- identity (2) + floor(identity (3.5));
val it = 5 : int
- identity (safeDiv);
val it = fn : int * int -> int
```

- First class values:

```
-fun polyDiv(a,b) = if b=0 then safeDiv else op div;
val polyDiv = fn : 'a * int -> int * int -> int
- val ff = polyDiv(2,0);
val ff = fn : int * int -> int
- ff(2,0);
val it = ~10000 : int
```

Restrictions on Polymorphic Functions

- A type variable **‘a** is generalizable: can represent any one type that we choose. However, once that type is selected, the type cannot change.
- In **val x = e;** we can give **x** a polymorphic type only if **e** is a syntactic value (also known as a nonexpansive expression)
  - literals and identifiers (e.g. **3**, **n**)
  - function expressions (e.g. **(fn n=>n)**)
  - constructors applied to syntactic values (e.g. **(12, x::nil)**)

Restrictions (cont.)

- Syntactic (nonexpansive expression) values do not include function calls

```
- val x = rev [];
stdIn:34.1-34.15 Warning: type vars not generalized
because of value restriction are instantiated to
dummy types (X1,X2,...)
val x = [] : ?.X1 list
```

- SML does not give **x** a polymorphic type, since **rev []** is not a syntactic value. But of course SML can't tell what the type of **x** should be--it could be a list of anything (**‘Z** in the textbook).

Restrictions (cont.)

- Type determination is complex
- When does a type problem arise
  - The expression is at the top level (not subexpression)
  - The type of the expression involves at least one type variable
  - The form of the expression does not meet the conditions for it to be nonexpansive

```
- identity(identity);
val it = fn : ?.X1 -> ?.X1
- (identity, identity);
val it = <poly-record> : ('a -> 'a) * ('b -> 'b)
- let val x = rev[] in 3::x end;
val it = [3] : int list
```

Operators and Polymorphism

- Polymorphism-destroying operators
  - Arithmatic +, -, *, /, div, mod
  - Inequality <, <=, >=, >
  - Boolean andalso, orelse, not
  - String concatenation ^
  - Type conversion operators ord, chr, ...
- Operators that allow polymorphism
  - Tuple (), #1, #2
  - List ::, @, hd, tl, nil, []
  - Equality =, <>

The Equality Operators

- Equality types
  - Allow equality to be tested among values of that type

```
- val x = (1, 2);
val x = (1,2) : int * int
- x = (1, 2);
val it = true : bool
- val L = [1, 2, 3];
val L = [1, 2, 3] : int list
- M = (2, 3);
val M = [2, 3] : int list
- L <> M
val it = true : bool
```

Type ''Z

- ''Z type
  - Type variables whose values are restricted to be an equality type
  - Operator = requires arguments of the same equality type
  - Functions can not be compared for equality

```
- identity = identity;
stdIn:44.1-44.20 Error: operator and operand don't
agree [equality type required]
  operator domain: ''Z * ''Z
  operand:         ('Y -> 'Y) * ('X -> 'X)
  in expression:
    identity = identity
```

Type ''a

- ''a type
  - Type variables whose values can be any type provided it is an equality type

```
- fun rev1 (L) = if L = nil then nil
             else rev1(tl(L)) @ [hd(L)];
val rev1 = fn : ''a list -> ''a list
```

```
- fun rev2 (L) = if null L then nil
             else rev1(tl(L)) @ [hd(L)];
val rev2 = fn : 'a list -> 'a list
```

```
- rev2 ([floor, trunc, ceil]);
- rev1 ([floor, trunc, ceil]);
```

Quick Sort in C

```
void  qsort ( void * base, size_t num, size_t width,
   int (*fncompare)(const void *, const void *) );
```

```
#include <stdio.h>
#include <stdlib.h>

int values[] = { 40, 10, 100, 90, 20, 25 };
int compare (const void * a, const void * b){
  return ( *(int*)a - *(int*)b );
}

int main (){
  int * pItem;  int n;
  qsort (values, 6, sizeof(int), compare);
  for (n=0; n<6; n++)  {
    printf ("%d ",values[n]);
  }
  return 0;
}
```