@SubSection{Defined Types}

Types can be given names:

@Verbatim{

```
       - let type intpair = int # int;
       > type intpair = int # int

       - let p = 12,20;
       > p = (12,20) : int # int

       - p : intpair;
         (12,20) : intpair
```

}

The new name is simply an abbreviation; for example, 'intpair' and
'int # int' are completely equivalent. The system may use any of these
equivalent type names when printing types, but tries to maintain the
names which have been explicitely forced by ':'.

@SubSection{Abstract Types}

New types (rather than mere abbreviations) can also be defined.
For example, to define a type 'time' we could do:

@Verbatim{

```
       - let type time <=> int # int
       = with maketime(hrs,mins) =
       =         if  hrs<0 Or 23<hrs Or mins<0 Or 59<mins
       =         then fail
       =         else abstime(hrs,mins)
       = and  hours t = fst(reptime t)
       = and  minutes t = snd(reptime t);
       > type time = -
       | maketime = \ : (int # int) -> time
       | hours = \ : time -> int
       | minutes = \ : time -> int
```

}

This declaration defines an abstract (i.e. new) type 'time' together
with three primitive functions: 'maketime', 'hours' and
'minutes'.
In general an abstract type declaration has the form 'type ty <=> ty1 with d'
where d is a declaration, i.e. the kind of phrase that can follow 'let'.
Such a declaration introduces a new type
ty which is represented by ty1.
Only within d can one use the (automatically declared)
functions absty (of type ty'->ty) and repty (of type ty->ty'), which
map between a type and its representation. In the example above 'abstime' and
'reptime' are only available in the definitions of 'maketime', 'hours'
and 'minutes'; these latter three functions, on the other hand,
are defined throughout the scope of the declaration. Thus an abstract
type declaration simultaneously declares a new type together with primitive
functions for the type. The representations of the type (i.e. ty'), and of the
primitives (i.e. the right hand sides of the definitions in b),
are not accessible outside the with-part of the declaration.

@Verbatim{

```
       - let t = maketime(8,30);
       > t = - : time
```

```
        - hours t, minutes t;
          (8,30) : int # int

}
```

Notice that values of an abstract type are printed as '-'.

When interacting with the system, it is useful to be able to define
abstract types incrementally in order to make experiments with them.
This can be achieved by an open ended '<=>' declaration, where the
'with' part is omitted.

```
@Verbatim{

       - let type time <=> int # int;
       > type time = -
       | abstime = \ : (int # int) -> time
       | reptime = \ : time -> (int # int)

}
```

Now 'abstime' and 'reptime' are available at the top level and can be
used to define 'maketime', 'hours' and 'minutes' as before.

The 'with' construct is not a special syntax for abstract types: it is an
environment operator which behaves like 'ins' on values (making
'abstime' and 'reptime' private) and like 'enc' on types (making
'time' available). Hence abstract types are obtained from the interaction
of two orthogonal features: the @Italic{isomorphism} type constructor
'<=>' and the environment operators.

@SubSection{Type Operators}

Both 'list' and '#' are examples of type operators; 'list' has one
argument (hence '@*{}a list') whereas '#' has two (hence '@*{}a # @*{}b').
Each type operator has various primitive operations associated with it,
for example 'list' has 'null', 'hd', 'tl',... etc, and '#' has
'fst', 'snd' and the infix ','.

```
@Verbatim{

        - let z = it;
        > z = 8,30 : int # int

        - fst z;
          8 : int

        - snd z;
          30 : int

}
```

Another standard operator of two arguments is '+';
'@*{}a + @*{}b' is the disjoint union of types @*{}a
and @*{}b, and associated with it are the following primitives:

```
@Verbatim{

   isl  : (@*{}a + @*{}b) -> bool      -- tests membership of left summand
   inl  : @*{}a -> (@*{}a + @*{}b)         -- injects into left summand
   inr  : @*{}a -> (@*{}b + @*{}a)         -- injects into right summand
   outl : (@*{}a + @*{}b) -> @*{}a         -- projects out of left summand
   outr : (@*{}a + @*{}b) -> @*{}b         -- projects out of right summand
}

These are illustrated by:
@Verbatim{

       - let x = inl 1
       = and y = inr 2;
       > x = inl 1 : int + @*{}a
       | y = inr 2 : @*{}b + int

       - isl x;
         true : bool

       - isl y;
         false : bool

       - outl x;
         1 : int

       - outl y;
       Failure: outl

       - outr x;
       Failure:  outr

       - outr y;
         2 : int

}

The abstract type 'time' defined above can be thought of
as a type operator with no arguments (i.e. a nullary operator);
its primitives are 'maketime', 'hours' and 'minutes'.
The 'type...<=>...with...' construct may also be used to define
non-nullary type operators (with 'rec type' in place of 'type'
if these are recursive).
For example trees analogous to LISP S-expressions could
be defined by:

@Verbatim{

       - let rec type @*{} sexp <=> @*{} + (@*{} sexp) # (@*{} sexp)
       =  with cons(s1,s2) = abssexp(inr(s1,s2))
       =  and  car s = fst(outr(repsexp s))
       =  and  cdr s = snd(outr(repsexp s))
       =  and  atom s = isl(repsexp s)
       =  and  makeatom a = abssexp(inl a);
       > type @*{}a sexp = -
       | cons = \ : (@*{}a sexp) # (@*{}a sexp) -> @*{}a sexp
       | car = \ : @*{}b sexp -> @*{}b sexp
       | cdr = \ : @*{}c sexp -> @*{}c sexp
       | atom = \ : @*{}d sexp -> bool
       | makeatom = \ : @*{}e -> @*{}e sexp

}

Exercise: introduce references in the above definition of sexp, so that
the LISP operations 'replaca' and 'replacd' can be defined.

Exercise: modify the definition of sexp to obtain @Italic{lazy} S-expressions.
(Hints: use functional objects to implement @Italic{suspensions} which prevent
early evaluation, and use references to ensure that suspensions
are evaluated at most once.)

@SubSection{Records}
```

```
 A record is very similar to a tuple '(x@Sub{1},...,x@Sub{n})'
 where each x@Sub{i} is given a different label; then the order of the x@Sub{i}
 is not important because every component can be identified by its label.

 @Verbatim{

        - let r = (|a=3; c="1"; b=true|);
        > r = (|a=3; b=true; c="1"|) : (|a:int; b:bool; c:tok list|)

        - r = (|c="1"; b=true; a=3|);
          true : bool

 }

 The special brackets '(|' and '|)' are used to delimit records and record
types.
 Note how the labels 'a', 'b' and 'c' are rearranged in alphabetical order by
the system.

 The only operation defined on records is field selection 'r.a' where a is a
 label and r has a record type.

 @Verbatim{

        - r.a;
          3 : int

        - (|a=r.a; b=r.b|);
          (|a=3; b=true|) : (|a:int; b:bool|)

 }

 Labels are not identifiers and cannot be computed.
 In the expression '\a. r.a' the first 'a' is a variable while the second
 'a' is a label; it is not possible to parameterize with respect to a label.
 Also, labels are not tokens.

 @Verbatim{

        - \a. r.a;
          \ : @*{}a -> int

        - it ();
          3 : int

 }

 A record field of the form 'a=a' (where the first 'a' is a label and the
 second one is a variable) can be abbreviated to 'a'; this is useful
 because often one uses variables having the same name as record fields
 for mnemonic reasons. The same applies to record types.

 @Verbatim{

       - let !{type R = (|bool; int|)!}
       = and f(x,y) = (|x; y|);
       > type R = (|bool:bool; int:int|)
       | f = \ : (@*{}a # @*{}b)  -> (|x:@*{}a; y:@*{}b|)

 }

 The exact type of every record must be known; the expression
 '\r. r.a' will fail to typecheck in isolation because not all fields of 'r'
 are know (we only know it has an 'a' field). However '\r. r.a' is
```

```
accepted when the context provides enough information about 'r'.

 @Verbatim{

        - \r. r.a;
        Unresolvable Record Selection of type:  (|a:@*{}a|)
        Field selector is:                      a

        - (\r. r.a) r;
          3 : int

 }

 @SubSection{Variants}

 Variants and records can be considered as complementary concepts;
 a record type is a labelled product of types, while a variant type is
 a labelled sum (disjoint union) of types. An object of a variant type
 can belong to one of several types; these different cases are distinguished
 by the label attached to every variant object. Hence testing the label of
 a variant object is like testing its type out a finite set of possibilities.

 @Verbatim{

        - let type OneOfTwo = [|int: int; bool: bool|];
        > type OneOfTwo = [|bool:bool; int:int|]

 }

 The special brackets '[|' and '|]' are used to delimit variants and
 variant types.
 Again, the labels 'bool' and 'int' are rearranged in alphabetical order
 (they should not be confused with the types `bool` and `int`
 which follow the `:`).

 Two basic operations are defined on variants: `is` tests the label of a
 variant object, and `as` returns the object contained in the variant.

 @Verbatim{

        - let v = [|int=3|] : OneOfTwo;
        > v = [|int=3|] : OneOfTwo

        - v is int, v is bool;
          (true,false) : bool # bool

        - v as int;
          3 : int

        - v as bool;
        Failure: as

 }

 We must specify `: OneOfTwo` in the definition of v because v might
 also belong to some
 different variant containing a case `int: int`. This type specification
 is not needed in general
 if the context gives enough information about v. Here is what happens if we
 forget to specify the type:

 @Verbatim{

        - [|int=3|];
        Unresolvable Variant Type:             [|int:int|]
```

}

A very useful abbreviation concerning varian types is the following:
whenever we have a variant type with a field 'a: .' we can abbreviate that
field specification
to 'a', and whenever we have a variant object '[|a=()|]' we can abbreviate
it to '[|a|]'.

@Verbatim{

```
        - let type color = [|red; orange; yellow|];
        =      and fruit = [|apple; orange; banana|];
        > type color = [|orange; red; yellow|]
        | type fruit = [|apple; banana; orange|]

        - let fruitcolor (fruit: fruit): color =
        =    case fruit of
        =      [|apple.  [|red|];
        =        banana. [|yellow|];
        =        orange. [|orange|]
        =      |];
        > fruitcolor = \ : fruit -> color
```

}

The 'case' construct is a convenient form of saying
'if fruit is apple then [|red|]; if fruit is banana then [|yellow|];
if fruit is orange then [|orange|] else fail` (`else fail` is never
executed; a compile-time error message is given if some case is
missing).

Note that `[|orange|]` is both a color and a fruit; it is possible to
disambiguate the occurrence above because of the type declarations
in the definition of fruitcolor.

@SubSection{Varstructs}

Structured variables (varstructs) can appear wherever a formal parameter
is expected; we have already seen the simple case of tuples:

@Verbatim{

```
        - let a,b,c = 1,2,3;
        > a = 1 : int
        | b = 2 : int
        | c = 3 : int
```

}

All the variables in a varstruct must be distinct, so that no ambiguites
can arise.

Lists can also be used in varstructs, and they can be nested with tuples
and with all the other kinds of varstructs:

@Verbatim{

```
       - let [a;b] = [1;2];
       > a = 1 : int
       | b = 2 : int

       - let a::b = [1;2;3];
       > a = 1 : int
       | b = [2;3] : int list
```

```
     - let a,[();b,c] = 1,[2,3;4,5];
      | a = 1 : int
      | b = 4 : int
      | c = 5 : int

      - let [a;b] = [3];
      Failure: varstruct

}
```

List varstructs may fail when the number of elements in a list is not
the expected one.
The varstrtuct '()' can be used to ignore parts of a structure.

Record and variant varstructs are also allowed.

```
@Verbatim{

      - let (|a=x; b=[|int=y|]|) = (|a=3; b=[|int=4|]:OneOfTwo|);
      > x = 3 : int
      | y = 4 : int

}
```

Varstructs can also appear as formal parameters of functions and
lambda-expressions, and in case statements. Note that record varstructs
in functions allow to pass arguments in random order by associating them
to labels; this feature is called 'call by keyword' in some languages.

```
@Verbatim{

      - let f [a;b,c] = a,b+c;
      > f = \ : ((int # int) list) -> ((int # int) # int)

      - (\(|a; b|). a,b) (|b=3; a=4|);
        (4,3) : int # int

      - let !{type IntOrPair = [|int:int; pair:(|x:int; y:int|)|]!}
      = enc total (a: IntOrPair): int =
      =        case a of
      =          [|int=z. z;
      =            pair=(|x; y|). x+y
      =          |]
      = in total [|pair= (|x=3; y=4|)|];
        7 : int

}
```

The varstruct '(|x; y|)' above is an abbreviation for
'(|x=x; y=y|)', just like in record expressions.