SML/NJ Error and Warning Messages

This document contains lists of error and warning messages produced by the SML/NJ Version 110 compiler, sorted alphabetically, with short explanations and examples.

Parsing Errors

The parser used by SML/NJ is produced by ML-Yacc, and it uses a lexer generated by ML-Lex. The parser uses an error repair scheme that attempts to get a correct parse by deleting, adding, or substituting tokens. The parser produces error messages like:

```
- fun + => 3;
= ;
stdIn:2.2-2.7 Error: syntax error: deleting  DARROW INT SEMICOLON
```

This error message indicates how the parser attempted to "repair" the input, and in this case indicates that the parser doesn't like the "`=> 3;`" that follow the plus sign.

For more detailed discussion of syntax errors generated by the parser, see the explanations of errors [76] through [79] below.

Compiler Errors

Error messages that start with "`Error: Compiler bug:` indicate that an unexpected situation has been encountered by the compiler. Example:

```
Error: Compiler bug: ModuleUtil: getStr: bad entity
```

Such a message indicates a bug in the compiler, and it should be reported to sml-bugs@research.bell.labs.com, with self-contained code to reproduce the message if possible.

Most such messages will be *secondary* error messages, meaning that they occur immediately following a normal (i.e. non "Compiler Bug") error message. Secondary errors typically occur when a primary error disrupts the internal state or data structures of the compiler, and the corrupted state then causes further failures. The SML/NJ compiler is pretty good at recovering from errors and failing gracefully, so secondary Compiler Bug errors are rare.

General Errors

In the example code shown for the following errors, the interactive prompt symbols have been omitted to improve readability. The user input is shown in regular font, with the compiler responses in italics.

[1] **argument of raise is not an exception**

The expression following the **`raise`** keyword should evaluate to an exception value, i.e. a value of type `exn`. In this case, the value has some other, inappropriate type. E.g.:

```
raise 3;
stdIn:16.7 Error: argument of raise is not an exception [literal]
  raised: int
  in expression:
    raise 3
```

[2] **argument type variables in datatype replication**

In a datatype replication declaration, neither the type name on the left hand side nor the type path (longid) on the right hand side should be preceeded by formal type variable arguments, even if the right hand side datatype is of arity n>0.

```
datatype 'a T = A of 'a;
datatype 'a T = A of 'a

datatype 'a T1 = datatype T;
stdIn:18.1-18.28 Error: argument type variables in datatype replication

datatype T1 = datatype T;
datatype 'a T = A of 'a
```

[3] **can't find function arguments in clause**

This occurs when an formal parameter pattern is not supplied on the left hand side in a `fun` declaration, or one of the formal parameters of an infix function symbol is missing.

```
fun f = 3;
stdIn:1.5 Error: can't find function arguments in clause

infix 3 ++;
infix 3 ++

fun (x xx) = 3;
stdIn:1.5-2.6 Error: can't find function arguments in clause
stdIn:1.5-2.6 Error: illegal function symbol in clause
```

[4] **case object and rules don't agree**

The *case object* is the expression following the **`case`** keyword. It's type must agree with the type of the lhs patterns in the *rules* (*`pat => exp`*) following the **`of`** keyword. All the patterns of the rules also have to agree in type, but that is another error.

```
case 3
  of true => 1
   | false => 2;
stdIn:1.1-25.16 Error: case object and rules don't agree [literal]
  rule domain: bool
  object: int
  in expression:
    (case 3
      of true => 1
       | false => 2)
```

[5] **clauses don't all have function name**

In a `fun` definition, the function name must appear in each clause. If it is omitted from one or more clauses, this error results.

```
fun f nil = 1
  | (x::y) = x;
stdIn:1.5-17.15 Error: clauses don't all have function name
```

This error is also reported when the function name in two clauses of the function definition differ, for instance because of a misspelling.

```
fun test (SOME s) = true
  | teat (NONE) = false;
stdIn:120.5-121.24 Error: clauses don't all have function name
```

[6] **clauses don't all have same number of patterns**

In a `fun` declaration, each clause, or rule, separated by `|` (vertical bar symbol), has to have the same number of curried arguments.

```
fun f x y = 3
  | f a b c = 4;
stdIn:1.5-26.16 Error: clauses don't all have same number of patterns
stdIn:24.6-26.16 Error: types of rules don't agree [tycon mismatch]
  earlier rule(s): 'Z * 'Y -> int
  this rule: 'X * 'W * 'V -> int
  in rule:
    (a,b,c) => 4
```

[7] **constant constructor applied to argument in pattern: %**

A constant constructor like `nil` can't be applied to an argument in a pattern.

```
val nil x = [];
stdIn:1.5-24.8 Error: constant constructor applied to argument in pattern:nil
```

[8] **constructor and argument don't agree in pattern**

A nonconstant constructor in a pattern must be applied to an argument pattern of the appropriate type (i.e. the domain type of the constructor).

```
datatype t = A of int;
val A true = A 3;
stdIn:1.1-26.3 Error: constructor and argument don't agree in pattern [tycon mismatch]
  constructor: int -> t
  argument:    bool
  in pattern:
    A true
```

[9] **data constructor % used without argument in pattern**

A nonconstant constructor must be applied to an argument when it is used in a pattern (though not necessarily when it is used in an expression).

```
datatype t = A of int
val A = A 3;
stdIn:17.5-17.12 Error: data constructor A used without argument in pattern
```

[10] **datatype % does not match specification**

Usually occurs because the constructors for a datatype declared in a structure don't agree with the constructors (in names or number) of a signature that the structure must match.

```
signature S =
sig
  datatype t = A of int
end;
signature S = sig datatype t = A of int end

structure A : S =
struct
  datatype t = A of int | B
end;
stdIn:1.1-27.4 Error: datatype t does not match specification
   constructors in actual only: B
```

[11] **datatype % has duplicate constructor name(s): %, %**

The names of the constructors of a given datatype must all be distinct.

```
datatype t = A | B | A of int;
stdIn:1.1-26.5 Error: datatype t has duplicate constructor name(s): A
```

[12] **dependency cycle in instantiate**

The *instantiate* process takes a signature and creates a dummy structure matching that signature with no extraneous sharing (i.e. no types are identified that don't need to be). This process can fail because of various kinds of circularities. An example of one of the simpler forms of circularity would be:

```
signature S =
sig
  type u
  datatype s = A of u
  sharing type u = s
end;
stdIn:16.1-21.4 Error: dependency cycle in instantiate
```

By default, every signature is instantiated when it is declared, to detect errors as early as possible. However, signature instantiation is strictly only necessary when a signature is used as a functor parameter signature or in an opaque (`:>`) signature constraint.

[13] **duplicate constructor specifications for % caused by include**

A signature should have only one specification of a given value or constructor name. A common way that multiple constructor specifications for a name can occur is if a constructor is specified explicitly, and also implicitly through an included signature.

```
signature S =
sig
```

```
  datatype t = A of int
end;
signature S = sig datatype t = A of int end

signature T =
sig
  datatype u = A
  include S
end;
stdIn:27.3-28.13 Error: duplicate constructor specifications for A caused by include
```

[14] **duplicate exception declaration**
An exception name is declared multiple times in a single exception declaration.

```
exception E of int
and E of bool;
stdIn:17.1-18.14 Error: duplicate exception declaration: E
```

Note that it is ok if the same exception name is declared in different exception declarations, as in the following.

```
exception E of int;
exception E of int

exception E of bool;
exception E of bool
```

[15] **duplicate function name in val rec dec**
When declaring several functions in a single `val rec` declaration, the names of the functions must be distinct.

```
val rec f = (fn x => x)
and f = (fn y => y + 3);
stdIn:21.1-22.24 Error: duplicate function name in val rec dec: f
```

[16] **duplicate function names in fun dec**
When declaring several functions in a single `fun` declaration, the names of the functions must be distinct.

```
fun f x = x
and f y = y + 3;
stdIn:1.1-23.16 Error: duplicate function names in fun dec: f
```

[17] **duplicate label in record**
The label names in a record expression or pattern must be distinct.

```
{a=3,b=true,a="abc"};
stdIn:1.1-1.21 Error: duplicate label in record: a
fun f {a=x,a=y} = 3;
stdIn:2.2-2.11 Error: duplicate label in record: a
```

[18] **duplicate specifications for % % in signature**
Only one specification for a given name in a given name space is allowed in signatures. Values and constructors (including exception constructors) are in one name space; types, structures, and functors are disjoint name spaces. So `x` cannot be specified twice as a value or constructor, but it can be specified as a value, as a type, as a structure, and as a functor in the same signature.

```
signature S =
sig
  val x : int
  val x : bool
end;
stdIn:20.3-21.16 Error: duplicate specifications for variable or constructor x in signature

signature S =
sig
  type t
  type t
end;
stdIn:24.3-25.10 Error: duplicate specifications for type constructor t in signature
signature S =
sig
  exception Foo
  exception Foo of int
end;
stdIn:28.3-29.24 Error: duplicate specifications for variable or constructor Foo in signature

signature S =
sig
  structure A : sig end
  structure A : sig end
end;
stdIn:32.3-33.25 Error: duplicate specifications for structure A in signature

signature S =
sig
  val x : int
  datatype t = x
end;
stdIn:36.3-37.18 Error: duplicate specifications for variable or constructor x in signature

signature S =
sig
  val x : int
  type x
  structure x : sig end
end;
signature S =
  sig
```

```
    val x : int
    type x
    structure x : sig end
  end
```

[19] **duplicate specifications for functor % caused by include**

Multiple specifications for a functor name occur in a signature, with one of the later ones introduced via an `include` spec. If the included functor spec comes first, you get error [19] instead.

```
signature S1 =
sig
  functor F () : sig end
end;
signature S1 = sig functor F : (: ) :  end

signature S2 =
sig
  include S1
  functor F(X: sig val x : int end): sig end
end;
stdIn:55.3-56.46 Error: duplicate specifications for functor F in signature

signature S2 =
sig
  functor F(X: sig val x : int end): sig end
  include S1
end;
stdIn:59.3-60.14 Error: duplicate specifications for functor F caused by include
```

[20] **duplicate specifications for structure % caused by include**

Multiple specifications for a structure name occur in a signature, with one of the later ones introduced via an `include` spec. If the included structure spec comes first, you get error [19] instead.

```
signature S1 =
sig
  structure A : sig end
end;
signature S1 = sig structure A : sig end end

signature S2 =
sig
  structure A : sig val x : int end
  include S1
end;
stdIn:67.3-68.14 Error: duplicate specifications for structure A caused by include

signature S3 =
sig
  include S1
  structure A : sig val x : int end
end;
stdIn:71.3-72.37 Error: duplicate specifications for structure A in signature
```

[21] **duplicate specifications for type % caused by include**

Multiple specifications for a type name occur in a signature, with one of the later ones introduced via an `include` spec. If the included structure spec comes first, you get error [19] instead.

```
signature S1 =
sig
  type t
end;
signature S1 = sig type t end

signature S2 =
sig
  type 'a t
  include S1
end;
stdIn:79.3-80.14 Error: duplicate specifications for type t caused by include

signature S3 =
sig
  include S1
  type 'a t
end;
stdIn:83.3-84.13 Error: duplicate specifications for type constructor t in signature
```

[22] **duplicate type definition**

A type name is defined twice in a single simultaneous type declaration (i.e. type declarations separated by **`and`**. If the simultaneous declaration is split into separate declarations, there is no error.

```
type t = int
and t = bool;
stdIn:17.1-18.13 Error: duplicate type definition: t

type t = int;
type t = int
type t = bool;
type t = bool
```

[23] **duplicate type names in type declaration**

A type name is defined multiple times in a datatype declaration (including possibly in the **`withtype`** part.

```
datatype t = A
and t = B;
stdIn:1.1-19.10 Error: duplicate type names in type declaration: t

datatype t = A
withtype t = int;
stdIn:1.1-20.17 Error: duplicate type names in type declaration: t
```

[24] **duplicate type variable name**

A type variable name is repeated in a type parameter list, when defining an n-ary type or datatype constructor, or explicitly binding types in a value declaration.

```
type ('a,'a) t = 'a * 'a;
stdIn:21.4-21.11 Error: duplicate type variable name: a

datatype ('a,'a) t = A of 'a;
stdIn:1.1-21.15 Error: duplicate type variable name: a

fun ('a,'a) f(x:'a) = x;
stdIn:1.1-21.10 Error: duplicate type variable name: a
```

[25] **duplicate value specifications for % caused by include**

Multiple specifications for a value name occur in a signature, with one of the later ones introduced via an `include` spec. If the included structure spec comes first, you get error [19] instead. It does not matter whether the multiple value specifications give the same type or not.

```
signature S1 =
sig
  val x : int
end;
signature S1 = sig val x : int end

signature S2 =
sig
  val x : bool
  include S1
end;
stdIn:29.3-30.14 Error: duplicate value specifications for x caused by include

signature S3 =
sig
  val x : int
  include S1
end;
stdIn:33.3-34.14 Error: duplicate value specifications for x caused by include

signature S4 =
sig
  include S1
  val x : int
end;
stdIn:37.3-38.15 Error: duplicate specifications for variable or constructor x in signature
```

[26] **duplicate variable in pattern(s)**

A variable may only occur once in a pattern (or in the sequence of argument patterns of a curried function declaration.

```
fun f(x,x) = x;
stdIn:1.5-2.10 Error: duplicate variable in pattern(s): x

fun f x x = x;
stdIn:1.5-2.9 Error: duplicate variable in pattern(s): x

val (x,x) = (3,3);
stdIn:1.1-36.3 Error: duplicate variable in pattern(s): x
```

[27] **explicit type variable cannot be generalized at its binding declaration: %**

A type variable used in a type constraint within a value expression or declaration must be generalized at the appropriate point (determined either explicitly or implicitly). If the type variable cannot be generalized at that point because of the value restriction, this error message results.

```
val x : 'a list = (fn x => x) nil;
stdIn:1.1-37.14 Error: explicit type variable cannot be generalized at its binding declaration: 'a

val 'a (x: 'a list) = (fn x => x) nil;
stdIn:1.1-38.5 Error: explicit type variable cannot be generalized at its binding declaration: 'a
```

[28] **expression and handler don't agree**

The type of the right hand side of the each rule in an exception handler must agree with the type of the base expression that the handler is attached to, because the value returned by the entire handle expression is either that of the base expression or the value returned by one of the handler rules.

```
fun f x = (hd x)+1 handle Empty => true;
stdIn:2.6-38.7 Error: expression and handler don't agree [literal]
  body:          int
  handler range:          bool
  in expression:
    hd x + 1
    handle
        Empty => true
    | exn => raise exn
```

[29] **expression or pattern begins with infix identifier "%"**

An infix identifier cannot be the first identifier in an expression, unless it is preceded by the **`op`** keyword.

```
+(2,3);
stdIn:1.1 Error: expression or pattern begins with infix identifier "+"
```

```
op +(2,3);
val it = 5 : int
```

[30] **expression or pattern ends with infix identifier "%"**

An expression cannot end with an infix identifier. Perhaps there is a missing **op** keyword.

```
2 +;
stdIn:40.4 Error: expression or pattern ends with infix identifier "+"
stdIn:40.1-40.4 Error: operator is not a function [literal]
  operator: int
  in expression:
    2 +

(fn x => x) +;
stdIn:40.3 Error: expression or pattern ends with infix identifier "+"

(fn x => x) op +;
val it = fn : int * int -> int
```

[31] **fixity precedence must be between 0 and 9**

This one is obvious. When defining new infix operators, you have to fit them into the existing precedence ranking, which is limited to ten levels, from 0 to 9, with higher numbers giving stronger precedence. See the Top Level Environment chapter of the Basis documentation for the precedences of the predefined infix operators.

```
infix 10 ++;
stdIn:43.7-43.9 Error: fixity precedence must be between 0 and 9

infix ~2 ++;
stdIn:2.2-2.4 Error: fixity precedence must be between 0 and 9
```

[32] **found data constructor instead of exception**

In a context where an exception constructor identifier was expected, a dataconstructor identifier was found instead.

```
exception Foo = nil;
stdIn:17.1-17.20 Error: found data constructor instead of exception
```

[33] **found variable instead of exception**

In a context where an exception constructor identifier was expected, a value variable was found instead.

```
val x = 3;
val x = 3 : int
exception Foo = x;
stdIn:18.1-18.18 Error: found variable instead of exception
```

[34] **handler domain is not exn**

In the rules following the **handler** keyword, the type of the patterns on the left hand side the rule must be `exn`. In the example below, the first error message is caused by a mismatch with an implicit default rule that handles exceptions not handled by the explicit rules of the handler.

```
3 handle nil => 4;
stdIn:1.1-18.7 Error: types of rules don't agree [tycon mismatch]
  earlier rule(s): 'Z list -> int
  this rule: exn -> 'Y
  in rule:
    exn => raise exn
stdIn:1.1-18.7 Error: handler domain is not exn [tycon mismatch]
  handler domain: 'Z list
  in expression:
    3
    handle
        nil => 4
      | exn => raise exn
```

[35] **ill-formed datatype spec**

In a datatype replication specification in a signature, type parameters were found on the left hand side of the specification.

```
signature S =
sig
  datatype 'a t = datatype bool
end;
stdIn:26.3-26.33 Error: ill-formed datatype spec
```

[36] **illegal (multiple?) type constraints in AS pattern**

The value variable in front of the **as** keyword can have a type constraint, but only one. This error also occurs in other circumstances, as illustrated by the second example.

```
val x : int list : int list as y::z = [1,2];
stdIn:29.5-29.36 Error: illegal (multiple?) type constraints in AS pattern

val (x: int list) as (y::z : int list) = [1];
stdIn:1.5-24.10 Error: illegal (multiple?) type constraints in AS pattern
stdIn:1.5-24.10 Error: parentheses illegal around variable in AS pattern

val x : int list as (y::z) = [1,2];
stdIn:1.1-24.6 Warning: binding not exhaustive
          x as y :: z = ...
val x = [1,2] : int list
val y = 1 : int
val z = [2] : int list
```

[37] **illegal function symbol in clause**

In a function declaration, the left hand side between the keyword **fun** and the equal sign must be a well-formed applicative term, and the operator (i.e. the function part of the top-level application) of this term must be a simple identifier. If the application has an infix operator, it must be parenthesized (unless followed immediately by a type constraint or the equal sign); otherwise it may not be parenthesized.

```
fun (f x) = 3;  (* bad parentheses *)
stdIn:1.5-2.5 Error: can't find function arguments in clause
stdIn:1.5-2.5 Error: illegal function symbol in clause

fun (x+y) = 3;  (* ok; redefines infix + operator *)
val + = fn : 'a * 'b -> int
```

[38] **inconsistent arities in type sharing % = %: % has arity % and % has arity %**

Two types involved in a type sharing constraint have different arities.

```
signature XSIG = sig
  datatype ('a, 'b) t = A of 'a | B of 'b
end

functor F
   (type u
    structure X: XSIG
        sharing type X.t = u) =
struct
end

stdIn:49.11-54.6 Error: inconsistent arities in type sharing t = u : t
has arity 2 and u has arity 0.
```

[39] **inconsistent equality properties in type sharing**

This error occurs when type constructors with incompatible equality properties are equated by sharing constraints. When this happens, the signature is not consistent, and could not be successfully matched.

```
signature S =
sig
  eqtype t
  datatype u = U of int -> int  (* not an equality type *)
  sharing type t = u
end;
stdIn:17.1-22.4 Error: inconsistent equality properties in type sharing
```

[40] **infix operator "%" used without "op" in fun dec**

A function symbol declared to be an infix symbol is used in a function declaration used to declare nonfix functions.

```
infix foo;
infix foo
fun foo (x,y) = x + y;
stdIn:34.5-34.8 Error: infix operator "foo" used without "op" in fun dec
```

The correct definition is:

```
fun op foo(x,y) = x +y;
val foo = fn : int * int -> int
```

[41] **infix operator required, or delete parentheses**

The first term following keyword **`fun`** in a function declaration is a parenthesized application, implying an infix application, but the middle subterm is not an infix symbol.

```
fun (x f y) = ();  (* would work if "f" were infix *)
stdIn:18.8 Error: infix operator required, or delete parentheses

fun x f y = ();    (* ok, but maybe not what was intended *)
val x = fn : 'a -> 'b -> unit
```

[42] **infix symbol "%" used where a nonfix identifier was expected**

In a **`val rec`** declaration, the if the identifier being declared (on the left hand side of the declaration) is an infix symbol, it must be preceeded by the **`op`** keyword.

```
infix ++;
infix ++
val rec ++ = (fn x => x);
stdIn:17.9-17.11 Error: infix symbol "++" used where a nonfix identifier was expected

val rec op ++ = (fn x => x);
val ++ = fn : 'a -> 'a
```

[43] **install_pp: empty path**

The function `Compiler.PPTable.install_pp` installs a user-defined pretty printer function (the second argument) for a generative (i.e. datatype or abstype) designated by the first argument, which must be a nonempty list of strings that can be interpreted as a symbolic path (longTyCon) naming a datatype or abstract type in the current environment. This function should only be called at top level.

```
Compiler.PPTable.install_pp [] (fn x => fn y => ());
 Error: install_pp: empty path
```

[44] **install_pp: nongenerative type constructor**

The function `Compiler.PPTable.install_pp` installs a user-defined pretty printer function (the second argument) for a generative (i.e. datatype or abstype) designated by the first argument, which must be a nonempty list of strings that can be interpreted as a symbolic path (longTyCon) naming a datatype or abstype in the current environment. This function should only be called at top level.

```
Compiler.PPTable.install_pp ["t"] (fn x => fn y => ());
 Error: install_pp: nongenerative type constructor
```

[45] **int constant too large**

Integer literal in program is too large. Default integers are represented using 31 bits, and range from ~1073741824 to 1073741823, or from:

```
Option.valOf(Int.minInt) to Option.valOf(Int.maxInt)
```

```
val x = 1073741823;
val x = 1073741823 : int

val x = 1073741824;
stdIn:2.4-22.7 Error: int constant too large

val x = ~1073741824;
val x = ~1073741824 : int

val  x = ~1073741825;
stdIn:30.10-30.21 Error: int constant too large
```

[46] **match nonexhaustive**

Insufficient patterns in clause to match against all the possible inputs. This is an error if the flag `Compiler.Control.MC.matchNonExhaustiveError` is set to true (the default is false), otherwise it is a warning if `Compiler.Control.MC.matchNonExhaustiveWarn` is set to true. If neither of these flags is true, then the compiler does not complain about nonexhaustive matches.

```
fun f 0 = 1
  | f 1 = 1;
stdIn:1.1-22.12 Error: match nonexhaustive
          0 => ...
          1 => ...

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

[47] **match redundant**

A pattern is provided that is covered by some earlier pattern. If the compiler flag `Compiler.Control.MC.matchRedundantError` is set to false (default is true), only a warning message is given. If `Compiler.Control.MC.matchRedundantWarn` is also false (default is true), no message is generated.

```
fun f (0, true) = 1
  | f (0, false) = 2
  | f (0, _) = 3
  | f _ = 4;
stdIn:24.1-27.14 Error: match redundant
          (0,true) => ...
          (0,false) => ...
    -->   (0,_) => ...
          _ => ...
```

[48] **match redundant and nonexhaustive**

A pattern is provided that is covered by some earlier pattern, and the set of patterns do not cover all the possible inputs. Whether this message is generated, and its severity (Error or Warning), are controlled by the compiler flags

```
Compiler.Control.MC.matchNonExhaustiveError
Compiler.Control.MC.matchNonExhaustiveWarn
Compiler.Control.MC.matchRedundantError
Compiler.Control.MC.matchRedundantWarn
```

Example:

```
fun f 1 = 1
  | f 2 = 3
  | f 1 = 4 ;
stdIn:1.1-24.12 Error: match redundant and nonexhaustive
          1 => ...
          2 => ...
    -->   1 => ...
```

[49] **multiple where defs for %**

The `where` clauses of a signature expression must not bind the same type-specification multiple times.

```
signature S = sig
  type t
end
where type t = int
  and type t = bool;
stdIn:1.1-72.20 Error: multiple where defs for t
```

or even:

```
signature S = sig
  type t
end
where type t = int
  and type t = int;
stdIn:1.1-76.19 Error: multiple where defs for t
```

[50] **non-constructor applied to argument in pattern**

The value applied to an argument in a pattern is not a constructor.

```
fun f (0 0) = true;
stdIn:17.5-17.19 Error: non-constructor applied to argument in pattern
```

[51] **non-constructor applied to argument in pattern: %**

Same error as [58]. This particular error occurs when the applied value has a name that can be reported.

```
val a = 0;
val a = 0 : int

fun f (a 0) = true;
stdIn:18.5-18.19 Error: non-constructor applied to argument in pattern: a
```

[52] **nonlocal path in structure sharing: %**

A structure participating in a structure **sharing** specification was not declared in the current signature.

```
signature S = sig
  structure A : sig end
  sharing A = B.C
end;
stdIn:41.11-41.18 Error: nonlocal path in structure sharing: B.C
```

[53] **nonlocal path in type sharing: %**

A type participating in a type **sharing** specification was not declared in the current signature.

```
signature S = sig
  type t
  sharing type t = B.t
end;
stdIn:44.16-44.23 Error: nonlocal path in type sharing: B.t
```

[54] **operator and operand don't agree**

A function (operator) is applied to a value (operand) with a type different than the type expected by the function.

```
fun f true = 0
  | f false = 1;
val f = fn : bool -> int

f 3;
stdIn:25.1-25.4 Error: operator and operand don't agree [literal]
  operator domain: bool
  operand:         int
  in expression:
    f 3
```

[55] **operator is not a function**

The value used in operator position is not a function.

```
3 true;
stdIn:1.1-19.6 Error: operator is not a function [literal]
  operator: int
  in expression:
    3 true
```

[56] **or-patterns don't agree**

In a pattern that uses or-ed subpatterns (via `|`), the type of all the subpatterns must agree.

```
fun f (0 | 1 | true) = 0;
stdIn:1.1-21.4 Error: or-patterns don't agree [literal]
  expected: int
  found: bool
  in pattern:
    (1 | true)
```

[57] **out-of-range word literal in pattern: 0w%**

A word literal used in a pattern is larger than the largest representable word.

```
fun f 0w100000000000 = 0
  | f _ = 1;
stdIn:1.1-27.12 Error: out-of-range word literal in pattern: 0w100000000000
```

[58] **overloaded variable not defined at type**

An overloaded variable is being instantiated at a type for which it has no definition. Typical overloaded variables include numerical operations, overloaded over the numerical types (int, word, etc.)

```
true + true;
stdIn:19.5 Error: overloaded variable not defined at type
  symbol: +
  type: bool
```

[59] **parameter or result constraints of clauses don't agree**

In a **`fun`** declaration, each clause, or rule, separated by `|` (vertical bar symbol), has to have the same type (both in the type accepted by the clauses, and the type returned by the clauses).

```
datatype typeA = A;
datatype typeA = A
datatype typeB = B;
datatype typeB = B
fun f A = 0
  | f B = 0;
stdIn:36.1-37.12 Error: parameter or result constraints of clauses don't agree [tycon mismatch]
  this clause:      typeB -> 'Z
  previous clauses:      typeA -> 'Z
  in declaration:
    f =
      (fn A => 0
        | B => 0)
```

[60] **parentheses illegal around variable in AS pattern**

In an "as"-pattern *`pat as pat`*, where the pattern to the left of the "as" is a simple variable, the variable must not be wrapped in parentheses.

```
val ((a) as (b,c)) = (4,5);
stdIn:19.5-31.2 Error: parentheses illegal around variable in AS pattern
```

[61] **pattern and constraint don't agree**

In a pattern, the type of the pattern and the constaint type of the pattern must agree.

```
fun f (0:bool)=0;
stdIn:38.1-38.17 Error: pattern and constraint don't agree [literal]
  pattern:    int
  constraint:    bool
  in pattern:
    0 : bool
```

[62] **pattern and expression in val dec don't agree**
In a declaration `val` *pat* `=` *exp*, the type of *pat* must match the type of *exp*.

```
val s:string = 6;
stdIn:1.1-18.6 Error: pattern and expression in val dec don't agree [literal]
  pattern:    string
  expression:    int
  in declaration:
    s : string = 6
```

[63] **pattern and expression in val dec don't agree**
In a declaration `val` *pat* `=` *exp*, the type of *pat* must match the type of *exp*.

```
val s:string = 6;
stdIn:1.1-18.6 Error: pattern and expression in val dec don't agree [literal]
  pattern:    string
  expression:    int
  in declaration:
    s : string = 6
```

[64] **pattern to left of "as" must be variable**
In an "as"-pattern *pat* `as` *pat*, the first pattern must be a simple variable, not a more complex pattern using tuples or data constructors.

```
val (a,_) as (_,b) = (7,5);
stdIn:1.5-18.8 Error: pattern to left of AS must be variable
```

[65] **pattern to left of AS must be variable**
In an "as"-pattern *pat* `as` *pat*, the first pattern must be a simple variable, not a more complex pattern using tuples or data constructors.

```
val (a,_) as (_,b) = (7,5);
stdIn:1.5-18.8 Error: pattern to left of AS must be variable
```

[66] **possibly inconsistent structure definitions at: %**
When a signature contains a sharing constraint between two structure-specifications, each of which is specified using a `where` clause, the compiler is unable to calculate whether the structures are compatible. This is a bug in the compiler and will be fixed in a future version.

```
signature SIG =
sig
   structure A : sig end
   structure B : sig structure Z : sig end
                 end where Z = A
   structure C : sig structure Z : sig end
                 end where Z = A
   sharing B = C
end;

stdIn:1.1-38.4 Error: possibly inconsistent structure definitions at: B.Z
```

[67] **real constant out of range: %**
A real literal must have an exponent in the proper range for the floating-point representation of the target machine. At present all SML/NJ target machines use IEEE double-precision floating point, so real literals must be in the range ~1.79769313486e308 to 1.79769313486e308.

```
2e309;

uncaught exception BadReal
  raised at: bignums/realconst.sml:228.54-228.63
```

At present, a bug in the compiler raises an exception instead of printing the appropriate error message.

[68] **rebinding data constructor "%" as variable**
An identifier bound as a data constructor cannot be rebound as a variable in a pattern.

```
fun nil x = x;
stdIn:1.5-2.9 Error: rebinding data constructor "nil" as variable
```

[69] **redundant patterns in match**
In a multi-clause pattern match, if one of the later patterns can only match cases that are covered by earlier patterns, then the later pattern is redundant and can never be matched. In SML '97 it is an error to have useless (redundant) patterns.

```
4 handle Match => 5 | e => 6 | Bind => 7;
stdIn:1.1-20.15 Error: redundant patterns in match
          Match => ...
          e => ...
    -->   Bind => ...
```

[70] **redundant where definitions**
The `where` clauses of a signature expression must not bind the same structure-specification to different structures.

```
signature S1 =
sig
  structure A : sig type t end
end
where A=Int and A=Real;
stdIn:32.1-36.23 Error: redundant where definitions
```

[71] **rhs of datatype replication not a datatype**

The syntax `datatype id1 = datatype id2` that binds the name *id1* to the existing datatype *id2* requires that *id2* must be a datatype, and not an ordinary type.

```
datatype myint = datatype int;
stdIn:38.1-38.30 Error: rhs of datatype replication not a datatype
```

[72] **rhs of datatype replication spec not a datatype**

The specification syntax `datatype id1 = datatype id2` that binds the name *id1* to the existing datatype *id2* requires that *id2* must be a datatype, and not an ordinary type.

```
signature S = sig type t
                  datatype d = datatype t
              end;
stdIn:37.18-40.17 Error: rhs of datatype replication spec not a datatype
```

[73] **right-hand-side of clause doesn't agree with function result type**

The body of (each clause of) a function must have the type specified in the function-result type constraint (if it is present).

```
fun f(x) : int = "hello";
stdIn:1.1-37.24 Error: right-hand-side of clause doesn't agree with function result type [tycon mismatch]
  expression:  string
  result type:  int
  in declaration:
    f = (fn x => "hello": int)
```

[74] **sharing structure with a descendent substructure**

A structure cannot share with one of its components.

```
signature S = sig structure A : sig structure B : sig end end
                  sharing A = A.B
              end;
stdIn:1.1-44.20 Error: Sharing structure with a descendent substructure
```

[75] **structure % defined by partially applied functor**

Functors in SML/NJ may be higher-order, so that the functor F in the example below returns (as its result) another functor, which in turn returns a structure. The result of applying F to an argument cannot, therefore, be bound to a structure name.

```
functor F()() = struct end;
functor F :

structure S = F();
stdIn:45.15-45.18 Error: structure S defined by partially applied functor
```

[76] **syntax error found at %**

This message is produced if the parser finds a syntax error and is unable to correct the problem using its built-in heuristics (deletion, insertion, or replacement of tokens). Example:

```
x andalso val y orelse z;
stdIn:1.6 Error: syntax error found at VAL
```

Note: Error correction in the parser relies on lookahead. Different amounts of lookahead are used depending on whether input is taken from the interactive toplevel or from a source file. Therefore, error messages for the same program can vary depending on circumstances. (See also the note on error [78].)

[77] **syntax error: deleting %**

This message indicates that the error-correcting parser attempted to rectify a syntax error by deleting (ignoring) some input token(s).

For example, let's assume that file *delete.sml* contains the following code:

```
structure 99 X =
    struct
        val x = 1
    end
```

Compiling this file produces:

```
- use "delete.sml";
[opening delete.sml]
delete.sml:1.11-1.13 Error: syntax error: deleting  INT
```

Note: Error correction in the parser relies on lookahead. Different amounts of lookahead are used depending on whether input is taken from the interactive toplevel or from a source file. Therefore, error messages for the same program can vary depending on circumstances. (See also the note on error [78].)

[78] **syntax error: inserting %**

This error message, like the previous one, is generated by SML/NJ's error-correcting parser. It indicates that the parser was able to correct a syntactic error by inserting an additional token.

For example, let's assume that file *insert.sml* contains the following code:

```
let
  val x = 1; y = x + x
in
  x * y
end
```

Compiling this file produces:

```
- use "insert.sml";
[opening insert.sml]
insert.sml:2.16 Error: syntax error: inserting  VAL
```

Note: Error correction in the parser relies on lookahead. Since the interactive parser cannot use lookahead, it is likely that its syntax error messages differ from those that are generated when compiling files. For example, typing the contents of *insert.sml* directly into the interactive toplevel produces:

```
let
  val x = 1; y = x + x
in
  x * y
```

```
  end;
  stdIn:2.14-2.19 Error: syntax error: deleting  ID EQUALOP ID
  stdIn:2.20-3.3 Error: syntax error: deleting  ID ID IN
  stdIn:4.3-4.8 Error: syntax error: deleting  ID ASTERISK ID
```

[79] **syntax error: replacing % with %**
The parser found a syntax error and has attempted to fix the problem by replacing some token(s) by some other token(s).

For example, let's assume that file *replace.sml* contains the following code:

```
  fn x = x
```

Compiling this file produces:

```
  - use "replace.sml";
  [opening replace.sml]
  replace.sml:1.6 Error: syntax error: replacing  EQUALOP with  DARROW
```

Note: Error correction in the parser relies on lookahead. Different amounts of lookahead are used depending on whether input is taken from the interactive toplevel or from a source file. Therefore, error messages for the same program can vary depending on circumstances. (See also the note on error [78].)

[80] **tycon arity for % does not match specified arity**
The arity of a type constructor differs between the definition inside a structure and its declaration in the corresponding signature constraint.

Example:

```
  signature S = sig type ('a, 'b) t end;
  signature S = sig type ('a,'b) t end

  structure S : S = struct
    type 'a t = 'a list
  end;
  stdIn:75.1-77.4 Error: tycon arity for t does not match specified arity
```

[81] **type % must be a datatype**
This message indicates that the signature constraint for a given structure requires some type to be a **datatype** but the structure defines it as different type (i.e., not a datatype).

Example:

```
signature S = sig datatype t = A | B end;
signature S = sig datatype t = A | B end

structure S : S = struct
  type t = int
end;
stdIn:80.1-82.4 Error: type t must be a datatype
stdIn:80.1-82.4 Error: unmatched constructor specification: A
stdIn:80.1-82.4 Error: unmatched constructor specification: B
```

[82] **type % must be an equality type**
This error message is issued when the definition for some type inside a structure does not permit equality while the corresponding signature constraint for the structure specifies that type as an **eqtype**.

Example:

```
  signature S = sig eqtype t end;
  signature S = sig eqtype t end

  structure S : S = struct
    type t = int -> int
  end;
  stdIn:86.1-88.4 Error: type t must be an equality type
```

[83] **type constraint of val rec dec is not a function type**
Names that are defined using **val rec** must refer to function values. Therefore, their types must be function types.

Example:

```
  val rec f : int = fn x => x;
  stdIn:1.1-79.26 Error: type constraint of val rec dec is not a function type [tycon mismatch]
    constraint:          int
    in declaration:
      f = (fn x => x)
```

[84] **type constraints on val rec declaraction [sic] disagree**
This error message occurs when a declaration has the form

```
  val rec id : ty1 = exp : ty2
```

and the types *ty1* and *ty2* disagree.

```
  val rec f : int -> int = (fn x => x) : bool -> bool;
  stdIn:1.1-29.30 Error: type constraints on val rec declaraction disagree [tycon mismatch]
    this constraint:   bool -> bool
    outer constraints:   int -> int
    in declaration:
      f = (fn x => x): bool -> bool
```

[85] **type constructor % given % arguments, wants %**
A type constructor was used with the wrong number of type arguments.

Example:

```
  type ('a, 'b) t = 'a * 'b;
  type ('a,'b) t = 'a * 'b
```

```
type u = (int, bool, real) t;
stdIn:103.28 Error: type constructor t given 3 arguments, wants 2
```

[86] **type variable % occurs with different equality properties in the same scope**
This message indicates that different occurences of the same type variable have inconsistent equality properties. In practice this means that the same name of a type variable was used both with one apostrophe and with two apostrophes. (Note that this would have been ok if the two occurences are clearly separated by scope.)

Example:

```
fun f (x: 'a, y: ''a) = (x, y);
stdIn:118.2-119.12 Error: type variable a occurs with different equality properties in the same scope
```

But:

```
fun 'a f (x: 'a) = let
  fun ''a g (y: ''a) = y = y
in x end;
val f = fn : 'a -> 'a
```

[87] **type variable in exception spec: %**
Exception declarations in signatures cannot contain type variables.

Example:

```
signature S = sig
  exception E of 'a list
end;
stdIn:135.3-135.26 Error: type variable in exception spec: E
```

[88] **type variable in top level exception type**
Exception definitions at top level cannot contain type variables.

Example:

```
exception E of 'a list;
stdIn:1.1-135.4 Error: type variable in top level exception type
```

[89] **types of rules don't agree**
The right-hand sides of the rules in a match must agree in type. Matches occur both in **case**- and in **fn**-expressions.

Examples:

```
fn true => false
 | false => 1;
stdIn:144.1-144.30 Error: types of rules don't agree [literal]
  earlier rule(s): bool -> bool
  this rule: bool -> int
  in rule:
    false => 1

fn x =>
   case x
     of true => false
      | false => 1;
stdIn:144.6-144.42 Error: types of rules don't agree [literal]
  earlier rule(s): bool -> bool
  this rule: bool -> int
  in rule:
    false => 1
```

[90] **unbound functor signature: %**
This error message is related to SML/NJ's higher-order module extension to Standard ML. The constraint on a functor declaration in some signature uses an undefined functor signature name.

Example:

```
signature S = sig
  functor F: FS
end;
stdIn:145.3-145.17 Error: unbound functor signature: FS
```

[91] **unbound functor: %**
The name of the functor being used is not defined.

Example:

```
structure S = F ();
stdIn:147.15-147.19 Error: unbound functor: F
```

[92] **unbound left hand side in where (structure): %**
A **where** specification refers to a structure inside a signature that was not declared there.

Example:

```
structure A = struct end;
structure A : sig end

signature S = sig end;
signature S = sig end

signature S' = S where B = A;
stdIn:158.1-158.29 Error: unbound left hand side in where (structure): B
```

But:

```
signature S = sig structure B : sig end end;
signature S = sig structure B : sig end end
```

```
signature S' = S where B = A;
signature S' = sig structure B : sig end end
```

[93] **unbound left hand side in where type: %**
A **where type** specification refers to a type inside a signature that was not declared there.

Example:

```
type t = int;
type t = int

signature S = sig end;
signature S = sig end

signature S' = S where type u = t;
stdIn:169.1-169.34 Error: unbound left hand side in where type: u
```

But:

```
signature S = sig type u end;
signature S = sig type u end

signature S' = S where type u = t;
signature S' = sig type u = t end
```

[94] **unbound signature: %**
A signature name is used but it has not been defined; for instance S in the following example:

```
structure A : S = struct end;
stdIn:16.15 Error: unbound signature: S
```

[95] **unbound structure: %**
A structure name is used but it has not been defined; for instance B in the following example:

```
- structure A = B;
stdIn:2.10 Error: unbound structure: B
```

[96] **unbound type constructor: %**
A type constructor name is used but it has not been defined, for instance t in the following example:

```
val x : t = ();
stdIn:2.4 Error: unbound type constructor: t
```

[97] **unbound type variable in type declaration: %**
A type variable occurs on the right hand side of a type or datatype declaration, without having been bound as a formal parameter on the left hand side.

```
type t = 'a list;
stdIn:2.5-2.12 Error: unbound type variable in type declaration: 'a

datatype 'a t = A of 'b;
stdIn:1.1-18.2 Error: unbound type variable in type declaration: 'b
```

[98] **unbound variable or constructor: %**
A value variable or constructor is used without having been defined or bound as a formal parameter.

```
x;
stdIn:1.1 Error: unbound variable or constructor: x

fun f x = x+y;
stdIn:2.8 Error: unbound variable or constructor: y
```

[99] **unresolved flex record (can't tell what fields there are besides %)**
When a flexible record pattern (one containing ...) is used, the context must provide enough type information to determine what all the fields are (though not necessarily their types).

```
fun f {x,...} = x;
stdIn:37.1-37.18 Error: unresolved flex record
   (can't tell what fields there are besides #x)

fun f ({x,...} : {x: int, y:bool}) = x;
val f = fn : {x:int, y:bool} -> int
```

If more than one field occurs in the flexible record pattern, then a different variant of this error message is generated. See error [100].

[100] **unresolved flex record (need to know the names of ALL the fields in this context)**
The pattern in a pattern match was a *flexible record*. The pattern omitted some of the record's members and summarized their existence using ellipses ("..."). But in the given context there was not enough information for the type checker to be able to infer the missing field names.

```
fun f {x,y,...} = (x,y);
stdIn:118.1-118.24 Error: unresolved flex record (need to know the names of ALL the fields
 in this context)
  type: {x:'Y, y:'X; 'Z}
```

[101] **value type in structure doesn't match signature spec**
A value component of a structure has a different type than that specified in a signature that the structure is matched against.

```
signature S =
sig
  val x : int
end;
signature S = sig val x : int end

structure A : S =
struct
  val x = true
end;
```

```
stdIn:21.1-24.4 Error: value type in structure doesn't match signature spec
    name: x
  spec:   int
  actual: bool
```

[102] **variable % does not occur in all branches of or-pattern**
SML/NJ supports or-patterns, where a single rule can have several patterns separated with the | symbol. The component patterns of an or-pattern are required to have exactly the same variables with the same types.

```
fun f(nil | x::_) = 1;
stdIn:1.5-2.18 Error: variable x does not occur in all branches of or-pattern
```

Here the component patterns are nil and x::_, and the variable x doesn't occur in the first pattern.

[103] **variable found where constructor is required: %**
A symbolic path (longid) of length greater than 1 occurring in a pattern must designate a data constructor.

```
fun f(Int.+) = 3;
stdIn:1.5-2.12 Error: variable found where constructor is required: Int.+
```

[104] **vector expression type failure**
In a vector expression of the form #[exp_1,exp_2,...], all the vector element expressions must be of the same type.

```
#[1,true];
stdIn:1.1-2.5 Error: vector expression type failure [literal]

fun f(x:int) = #[x,true];
stdIn:2.11-2.20 Error: vector expression type failure [tycon mismatch]
```

[105] **vector pattern type failure**
In a vector pattern of the form #[pat_1,pat_2,...], all the vector element patterns must be of the same type.

```
fun f(#[x:int,y:bool]) = (x + 1; not y);
stdIn:1.1-2.35 Error: vector pattern type failure [tycon mismatch]
```

[106] **where defn applied to definitional spec**
SML/NJ does not allow multiple definitions of a structure in a signature (one through a definitional spec, another through a **where** clause).

```
structure A = struct end;
structure A : sig end

signature S =
sig
  structure X : sig end = A
end
where X = A;
stdIn:27.1-31.12 Error: where defn applied to definitional spec
```

[107] **where type definition has wrong arity: %**
The arity implied by a **where type** definition must agree with the arity in type specification that it applies to.

```
signature S =
sig
  type 'a t
end
where type t = int;
stdIn:1.1-26.19 Error: where type definition has wrong arity: t
```

[108] **where type defn applied to definitional spec: %**
SML/NJ does not allow multiple definitions of a type in a signature (one through a definitional spec, another through a **where type** clause).

```
signature S =
sig
  type t = int
end
where type t = int;
stdIn:1.1-22.19 Error: where type defn applied to definitional spec: t
```

[109] **withtype not allowed in datatype replication**
One can't attach a **withtype** clause to a datatype replication declaration or specification.

```
datatype t = A;
datatype t = A

datatype s = datatype t
withtype u = s list;
stdIn:37.1-38.20 Error: withtype not allowed in datatype replication
```

[110] **word constant too large**
Word constants (by default Word31.word) are limited to values less than 0w2147483648 (0wx80000000). Similarly for word literals of type Word32.word (bound 0w4294967296) and Word8.word (bound 0w256).

```
0w2147483648;
stdIn:1.1-18.3 Error: word constant too large
0wx80000000;
stdIn:1.1-18.2 Error: word constant too large

0w4294967296 : Word32.word;
stdIn:25.1-25.13 Error: word constant too large
0wx100000000 : Word32.word;
stdIn:23.1-23.13 Error: word constant too large

0w256: Word8.word;
stdIn:1.1-1.6 Error: word constant too large
0wx100 : Word8.word;
stdIn:1.1-24.2 Error: word constant too large
```

Warnings

[1] **match nonexhaustive**

Insufficient patterns in clause to match against all the possible inputs. This is an warning if the flag Compiler.Control.MC.matchNonExhaustiveError is set to false (the default), Compiler.Control.MC.matchNonExhaustiveWarn is set to true. If neither of these flags is true, then the compiler does not complain about nonexhaustive matches.

```
fun f 0 = 1
  | f 1 = 1;
stdIn:1.1-22.12 Warning: match nonexhaustive
          0 => ...
          1 => ...

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

[2] **match redundant**

A pattern is provided that is covered by some earlier pattern. This is a warning if the compiler flag Compiler.Control.MC.matchRedundantError is set to false (default is true) and Compiler.Control.MC.matchRedundantWarn is true (the default).

```
fun f (0, true) = 1
  | f (0, false) = 2
  | f (0, _) = 3
  | f _ = 4;
stdIn:24.1-27.14 Warning: match redundant
          (0,true) => ...
          (0,false) => ...
     -->  (0,_) => ...
          _ => ...
```

[3] **match redundant and nonexhaustive**

A pattern is provided that is covered by some earlier pattern, and the set of patterns do not cover all the possible inputs. Whether this message is generated, and its severity (Error or Warning), are controlled by the compiler flags

```
Compiler.Control.MC.matchNonExhaustiveError
Compiler.Control.MC.matchNonExhaustiveWarn
Compiler.Control.MC.matchRedundantError
Compiler.Control.MC.matchRedundantWarn
```

If the first two are set to false and the latter two are set to true, then this warning is generated.

```
fun f 1 = 1
  | f 2 = 3
  | f 1 = 4 ;
stdIn:1.1-24.12 Warning: match redundant and nonexhaustive
          1 => ...
          2 => ...
     -->  1 => ...
```

[4] **mixed left- and right-associative operators of same precedence**

If an infix expression like

aexp id_1 *aexp* id_2 *aexp*

involves two infix operators id_1 and id_2 of the same precedence but opposite associativity, the SML '97 Definition states that the expression is illegal. But SML/NJ only issues this warning message and associates the two operators to the left.

```
- infix 4 <<;
infix 4 <<
- infixr 4 >>;
infixr 4 >>
- fun (x>>y) = "right";
val >> = fn : 'a * 'b -> string
- fun (x<<y) = "left";
val << = fn : 'a * 'b -> string
- 1 << 2 >> 3;
stdIn:21.8-21.10 Warning: mixed left- and right-associative operators of same precedence
val it = "right" : string
- 1 >> 2 << 3;
stdIn:22.8-22.10 Warning: mixed left- and right-associative operators of same precedence
val it = "left" : string
```

[5] **nongeneralizable type variable**

This warning is given for a top level value declaration whose type has free type variables that cannot be generalized because of the *value restriction*. See the detailed discussion of the value restriction in the SML '97 Conversion Guide.

```
val x = (fn x => x) nil;
stdIn:17.1-17.24 Warning: type vars not generalized because of
   value restriction are instantiated to dummy types (X1,X2,...)
val x = [] : ?.X1 list
```

In this example, the right hand side of the declaration of x has type 'X list, where 'X is a free type variable. This type variable cannot be generalized to form a polymorphic type for x because the right hand expression is *expansive* (a function call in this case). So the compiler eliminates the free type variable 'X be inventing a new dummy type named X1 and instantiates 'X to X1. Since X1 won't match any other type, there is little one can do with x now (one could take its lenght (0), but one cannot cons any values onto x).
