PROLAC LANGUAGE REFERENCE MANUAL

Revised version of August 29, 1999

1 Introduction

This is a draft of the Prolac language reference manual. Prolac is an object-oriented language designed for creating efficient but readable protocol implementations. This draft attempts to be definitive, but does not always define Prolac semantics precisely.

A Prolac specification is stored in a single file; the Prolac compiler reads the file, analyzes it, and produces two output files in the C language. The first output file is a header file containing C structure definitions corresponding to Prolac structures; the second is a C source file containing definitions for any exported Prolac methods (§5.5). Prolac is completely order-independent: anything can be used before it is declared or defined.

In the rest of this manual, the largest Prolac structures, modules, are discussed first, while types and expressions are saved for last.

 The most current version of this manual, as well as technical papers on Prolac and source for the Prolac compiler, is available from http://www.pdos.lcs.mit.edu/prolac/.

1.1 Terminology

A *name* is either a *simple name*—that is, an identifier—or a *qualified name*, which is a member expression 'X.n' (§9.6) where 'X' is a name and 'n' is an identifier. (Note that a pointer-to-member expression 'X->n' is not a name.) The keywords all, allstatic and constructor can be used as name components in some contexts. Specifically, all and allstatic are allowed in module namespace operators (§4.6) and constructor is allowed as a method name (§5.3).

A *feature* is simply something that has a name. Modules (§3), namespaces (§4), methods (§5), fields (§6), and exceptions (§7) are features.

2 Lexical analysis

Prolac programs are stored as a sequence of ASCII characters. As in C, whitespace—spaces, horizontal and vertical tabs, formfeeds, carriage returns, newlines, and comments—is ignored except when it separates other tokens. Prolac supports both C's comment syntax '/* ... */' and C++'s '// ... *newline*'.

2.1 Identifiers

An *identifier* is an arbitrarily long sequence of letters, digits, underscores, and hyphens '–'. Identifiers must start with a letter or underscore; an identifier cannot contain two consecutive hyphens or end in a hyphen. Identifiers which differ only in substituting hyphens for underscores and vice versa are considered identical; thus, 'one–thing' and 'one_thing' are the same identifier. (In generated C code, Prolac substitutes underscores for all hyphens.)

Some examples:

```
a–pretty–long–identifier   is one identifier
–23x_x–23  ≡  '– 23 x_x–23'  ≡  '– 23 x_x_23'
x––  ≡  'x ––'
```

Identifiers containing double underscores '__' or an equivalent ('–_', '_–') are reserved for the implementation.

2.2 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

all	exception	long	static
allstatic	export	module	super
bool	false	noinline	then
catch	field	notusing	true
char	has	outline	uchar
class	hide	pathinline	uint
constructor	if	rename	ulong
defaultinline	in	self	ushort
else	inline	seqint	using
elseif	int	short	void
end	let	show	

These single characters serve as operators or punctuation:

```
!  %  ^  &  *  (  )  –  +  =  {  }  |  ~
[  ]  \  ;  '  :  "  <  >  ?  ,  .  /
```

These multi-character sequences are also single tokens:

```
–>   ++   ––   :>   <<   >>   <=   >=   ==
!=   &&   ||   +=   –=   *=   /=   %=   ^=
&=   |=   <<=  >>=  ==>  |||  ::=  %{   %}
```

2.3 Numbers and literals

Prolac's definitions for number, string, and character literals are the same as C's. However, Prolac does not support floating-point types or values, so any floating-point literal encountered is an error. The current Prolac compiler only has partial support for string literals.

2.4 Preprocessing

The Prolac compiler has some features to facilitate preprocessing Prolac files with *cpp*, the C preprocessor. In particular, it understands '# *line*' directives, and will generate error messages with appropriate line numbers. The compiler also generates '# *line*' directives in its C output files.

2.5 Including C code

C code may be included in a Prolac file in two ways. First, *C blocks* (§9.5) occur within method bodies, where they specify code to run during the method's execution. An open brace '{' within an expression introduces a C block; to read the C block, the lexer copies characters without interpretation, respecting nested braces, until the next unbalanced '}' character not in a string or character literal or a comment. C blocks can refer to some Prolac objects using Prolac names; see §9.5 for details.

The '%{' and '%}' operators specify *support C code* not relating to any method; support C code is passed unchanged to the output file at file scope. Support C code cannot refer to Prolac objects using Prolac names.

'%{' can occur wherever a definition is expected; the lexer then copies characters without interpretation until the next '%}' sequence not in a string or character literal or a comment. Note that support C code can therefore contain unbalanced braces.

Support C code occurring in the input file before any actual Prolac code is collected, in order of definition, and placed in the output C source file before any Prolac-generated code. All other support C code is collected in order of definition and placed at the very end of the output source file.

3 Modules

Modules are Prolac's basic means of organizing programs. Modules contain methods (§5), which represent computation, and fields (§6), which represent data. Each module is also a namespace (§4). A module is wholly self-contained; it must explicitly import other modules if it wants to refer to them (§3.3). Modules can be subtypes of other modules (§3.2). A module can have a special method which initializes objects of its type (§5.3). Module definitions cannot be nested.

A module definition looks like this:

```
module name [:> parents...] [has imports...] {
  ...definitions...
}
```

The reserved word class is a synonym for module.

3.1 The module header

The *module header* defines the interface between a module and the rest of the program. The interior of a module can only refer to other modules if they have been explicitly specified in the module's header (or a supertype's header).

The module header has two parts, *parents* and *imports*. Each part is a comma-separated list of module expressions, where a module expression is a module name possibly modified by module operators (§3.4).

The rightmost components of all parent and import names must be distinct. These name components can be used within the module to refer to the parents and imports. Thus, this example is illegal:

```
Alice {
  module M { ... }
}
Bob {
  module M { ... }
}
module N has Alice.M, Bob.M {
  // error: two definitions for 'M'
}
```

Use module equations (§3.4.1) to get around this restriction.

3.2 Supertypes

A module may have one *parent*, which is another module. The parent, if any, must be explicitly listed in the module's header. This parent is one of the module's *supertypes*. The supertype relation is the transitive closure of the parent relation: if A is a supertype of B and B is C's parent, then A is a supertype of C.

A module inherits its supertypes' features—their imports, fields, namespaces, and methods. The parent's features are generally available without qualification under their own names, as if the module's definition was inserted into the parent's definition; see §4.5.2 for a detailed description of how parents' namespaces are combined into a module's namespace.

If P is a supertype of M, then a pointer to M may be used wherever a pointer to P is expected; or, in notation, *M :> *P (§8.6). (Unlike C++, actual objects of type M cannot be used where objects of type P are expected.)

A module can *override* some of its supertypes' methods. On an object of that module type, the overriding definitions will be used whenever the parents' overridden methods are called. This process, called *dynamic dispatch*, is described in §5.2.

It is not an error to explicitly mention another parent's supertype as a parent. This is not actually multiple inheritance; only one copy of the supertype in question is inherited. For example, this is legal:

```
module M { ... }
module N :> M { ... }
module O :> M, N { ... }
```

This can be useful to redefine some of M's module operators—for example, its inline levels (§9.8.1).

3.3 Imports

If code inside a module M must refer to another module I that isn't a supertype of M, then M must list I as an *import* in the has clause of its module header. Imports' names, unlike supertypes' names, are not copied into a module's namespace; you must use qualified names to refer to features in an import (but see §5.4). If M imports I, it is not necessarily true that M :> I.

3.4 Module operators

Prolac has a powerful collection of *module operators*, which are operators that act on modules instead of values. The particular operators are described later in this manual; there are operators that control a module's namespace (hide, show, and rename, §4.6), operators that control how implicit methods (§5.4) are found (using and notusing, §5.4.3), and an operator controlling how methods are inlined (inline, §9.8.1). A module operator expression has a module value, so module operators may be used anywhere a module is expected.

Each module operator described in this manual changes a module's secondary features (its namespace, its exported methods, its preferred inline levels) without affecting its primary features (its rule complement, its signatures, its supertypes). In fact, two module expressions differing only in module operators have the same type.

3.4.1 Module equations

A *module equation* defines a new name for an old module. The syntax is:

```
module new-module ::=
  old-module [module operators...] ;
```

Other modules can then refer to *new-module* as an abbreviation for '*old-module* [*module operators*...]'. A module equation does not define a new type.

3.4.2 After-module operators

A module definition can specify operators that will be applied to the module by default. The syntax is to write module operators directly after the module definition's closing

brace. For example:

```
module M {
  public-method ::= ...;
  private-method ::= ...;
} hide private-method;
```

A module definition with after-module operators 'module M { ... } *operators*' is effectively equivalent to this definition with a module equation:

```
module __M { ... }
module M ::= __M operators;
```

4 Namespaces

A *namespace* maps names to features. The most common examples of namespaces in Prolac are modules—each module defines a namespace. The programmer can also create *explicit namespaces*, either outside all modules (to organize modules into groups) or within a module (to organize methods into groups). An explicit namespace is created by writing the name of the namespace followed by an open brace '{':

```
namespace-name {
  ...definitions...
}
```

Namespaces may be nested—a namespace may have a *parent namespace* which is used during name lookup. Most namespaces are open, meaning you can define a namespace in parts through several definitions. (Module namespaces are not open: each module namespace is defined in exactly one place.)

4.1 Defining names

Every feature (module, namespace, method, field, and exception) is defined in a namespace using a name. That *defining name* may be either simple or qualified. If it is simple, the feature is simply added to the current namespace under that name. A definition with a qualified name '*n1*. · · · .*nk* ::= *F*' is an abbreviation for a definition within nested namespaces, '*n1* { ... { *nk* ::= *F* } ... }'. Thus, these two examples are completely equivalent:

```
module Package.M {
  namespace.method ::= ...;
}
Package {
  module M {
    namespace {
      method ::= ...;
    }
  }
}
```

Because namespaces are open, the intermediate namespaces created by such a definition can be extended by other definitions.

It is an error to define two features in the same namespace with the same name (but see §4.5.1 and §5.2 for information on method overriding). For example:

```
module M has D {
  D ::= ...;   // error: 'D' redefined
  namespace.f ::= ...;
  namespace {
    f ::= ...;   // error: 'namespace.f' redefined
  }
}
```

4.2 Name lookup

A recursive algorithm is used to look up a name *N* in a namespace *S*. This happens when a name is used (for example, in an expression).

If *N* is a simple name *n*, then Prolac searches for *n* in *S*, then in *S*'s parent, then its grandparent, and so on; the first definition found is used. If no definition is found, the lookup fails.

If *N* is a qualified name '*Nsub.n*', Prolac first uses this algorithm recursively to look up the name *Nsub* in *S*. If the result is a namespace *S2*, then Prolac searches for *n* in *S2* (but not its parents). If no result is found, or the original search failed or didn't produce a namespace, then the lookup fails. This algorithm is described in more detail in §9.6.

4.3 Global names

The direct member operator '.' can also be used as a prefix; for example, '.M' is a name. To look up a name '*.n*', Prolac first finds the current *most global namespace*, *GS*. Within a module, the most global namespace is the module's top-level namespace; outside any module, it is the file namespace. Once it has found *GS*, the name lookup proceeds as if the expression was '*GS.n*'. This allows an expression in an inner namespace to refer to features in an outer namespace, whether or not their names have been reused in the inner namespace.

4.4 Namelike expressions

Prolac allows methods which take no arguments to be called without parentheses (§9.2). This means that an expression which looks like a name may actually contain method calls; for example:

```
module M1 {
  a.b.c.d.e ::= ...;
  method ::= a.b.c.d.e;   // really just a name
}
module D {
  d.e ::= ...;
}
module M2 has D {
  a :> M2 ::= ...;
  b.c :> D ::= ...;
  method ::= a.b.c.d.e;   // not just a name!
  // same as:
  method2 ::= let temp1 :> M2 = a() in
    let temp2 :> D = temp1.b.c() in
      temp3.d.e()
    end end;
}
```

These name-like expressions are not allowed where a name is required—here, 'M2.a.b.c.d.e' cannot be hidden by the hide module operator.

4.5 Module namespaces

Every module is a namespace. Module namespaces have no parent—they are sealed off from surrounding namespaces. To illustrate:

```
module FindMe { ... }
N {
  module A :> FindMe ...   // finds FindMe
}
module M {
  find-me-2 ::= ...;
  inner {
    ... find-me-2 ...   // finds M.find-me-2
  }
  ... FindMe ...   // does not find FindMe
}
```

The outside world—specifically, other modules—can only be reached through a module's parents and imports, which the module explicitly declares in its module header (§3.1).

4.5.1 Original and default namespaces

Each module M has an *original namespace*, which is the namespace seen inside the definition of M. It contains a union of M's parents' default namespaces, as well as every method, field, and nested namespace defined in M, and every implicit method (§5.4) used by M.

Clients of M (modules that subtype or import M) see a different namespace, M's *default namespace*. This is equal to M's original namespace with all implicit methods hidden and any after-module operators (§3.4.2) applied. (Note that the after-module operators might show some of the implicit methods again.)

A module M's original namespace ONS is created by merging all of M's parent's default namespaces, and then merging M's internal namespace into that. Finally, any implicit methods referred to in M are stored in M's original namespace.

Imports coming from M's parents are not merged into the original namespace. Name conflicts during the merging process are generally an error, unless the two names refer to the same feature (the same supertype or the same method).

The remainder of this section describes this process in more precise detail. We start with an empty ONS and add definitions to it from different sources.

4.5.2 Parents and imports

If M has parent P, we first create P's original namespace PNS. We then copy all definitions from PNS into ONS, except for any of its parent or import definitions.

Next, definitions for M's parent and any imports specified in its module header are added to ONS. Each parent or import is defined using the identifier n that is the rightmost component of its name. Thus, the original namespace for M will have a feature named n representing that parent or import.

Any conflict between a feature FP, inherited in the last stage from the parent P, and FM, a parent or import being added, is resolved silently in favor of FM (that is, FM replaces FP in ONS).

4.5.3 The internal namespace

Next, we merge M's *internal namespace*, INS, into ONS. A module's internal namespace contains the definitions the user provided inside the module body; in particular, changing a module's module header does not change its internal namespace. Any conflicts between features FM, from ONS, and FI, from INS, are resolved as follows:

1. If FM and FI are both methods, then FM must have been inherited from some supertype. The method FI *overrides* FM; there is no namespace conflict, although there may be a type conflict. See §5.2 for details on overriding.

2. If FM and FI are both namespaces, the contents of FI are merged into FM. Conflicts found during the merge are resolved using this algorithm.

3. Otherwise, an error is reported.

4.5.4 Implicit methods

Finally, all methods that were defined in M have their bodies scanned for implicit methods. If a simple name in a method body cannot be found in ONS or its appropriate subnamespaces (§4.2), then that name is an implicit method (§5.4) attached to some field or import, or it is undefined. For now, the name in question is defined in ONS as a new implicit method. Its definition will be found, or an error reported if there is no definition, at a later stage of compilation.

The resulting namespace ONS is M's original namespace.

4.6 Namespace module operators

This section describes the three module operators which affect module namespaces, hide, show, and rename. These operators are powerful tools for namespace manipulation and access control, but may cause confusion when overused, so be careful.

4.6.1 'hide'

The hide operator hides some of a module's names. The left operand to hide is a module expression; the right operand is a name, a comma-separated list of names, or 'all', which hides all of the module's names. Some examples:

```
M hide internal-operation
M hide (evil, kill, horrible.death, maim)
M hide all
```

The list of names must be enclosed in parentheses because of module operators' high precedence (§9.1).

You may also hide names that originate in a particular supertype. For example:

```
module A {
  x ::= ...;
  y ::= ...;
}
module B :> A {
  z ::= ...;
}
... B hide A.x ...   // hides x from B's namespace
... B hide A.all ...   // hides x and y from B's namespace
```

However, you cannot hide names inside an import.

A group of names can be hidden with 'hide *"pattern"*', which hides all names that match the *pattern*. The usual shell metacharacters (*, ?, and character classes [...]) may be used within *pattern*. For example:

```
module A {
  f1 ::= ...;
};
module B :> A {
  _a ::= ...;
  _bbb ::= ...;
  _c ::= ...;
  f2 ::= ...;
} hide "_[a-b]*"   // hides _a and _bbb
   hide "B.f*";   // hides f2, doesn't hide f1
   // (f1 didn't originate in B)
```

If E is name expression, then 'M hide E' is equivalent to 'M hide "E1"', where E1 is E with all parentheses and whitespace removed. Metacharacters can only be used in the string form.

4.6.2 'show'

Show is the converse of hide. Hide makes names inaccessible; show makes hidden names accessible again.

The left argument to show is a module expression, and the right argument is a list containing any number of names 'n' and name assignments '(new = old)'. A single name 'n' is essentially equivalent to the name assignment '(n = n)'.

To evaluate a show operator applied to a module M, Prolac first looks up the old name, 'old', in M's original namespace (§4.5.1). This name may be further qualified through M's supertypes. It is an error if the name is undefined.

Prolac then evaluates the new name 'new' in M's *current* namespace. This name must not be qualified through M's supertypes and imports.[1] It is an error if a feature with this name already exists; otherwise, Prolac binds F to this name in the resulting module.

It is an error to show a module's constructor in a nested namespace or under a name other than constructor.

Note that show can be used to make a single feature available under multiple names; for example:

```
module M {
  bad ::= ...;
}
module M2 ::= M show (good = bad);
```

Since M2.bad and M2.good are the *same* feature—not two copies of a feature—overriding either one will effectively override them both.

Because old names are looked up in the module's original namespace, hidden renamed features cannot be shown using the new name. This code will not work:

1. A show operation 'M show A.n', where there is only one name and it is qualified through an supertype or import, is not automatically an error; it is equivalent to 'M show (n = A.n)'.

```
module M { r ::= ...; }
module N :>
  M rename (r = weird)
    hide weird  // OK
    show weird  // error: no weird in M
    show r  // OK
{}
```

The 'show all' operation is not yet implemented.

4.6.3 'rename'

Rename changes the name you use to access a feature. The left operand must be a module expression; the right operand must be a name assignment '(new = old)' or a list of name assignments.

'M rename (new = old)' is equivalent to 'M show (new = old) hide old'. It is an error for either old or new to be qualified through supertypes or imports; for new to conflict with an existing name in M; or to attempt to rename a module's constructor.

5 Methods

Prolac code is written in *methods*. Like functions in most programming languages, methods can take parameters and can return a value. Methods can call one another, possibly recursively.

Methods can be *static* or *dynamic*. Static methods are equivalent to normal functions, while dynamic methods are called with an implicit reference to some object of module type.

A module may provide new definitions for some of its supertypes' dynamic methods. This process is called *overriding* the supertypes' methods. When an overridden method is called on an object with that module's type, the call will be handled by the new, overriding definition instead; this is called *dynamic dispatch*.

Each method has an *origin*, which is the module that provided the first, non-overriding definition for the method. Each method definition comes from some module, which is called its *actual*. For example:

```
module M {
  f ::= ...;   // origin = M, actual = M
}
module N :> M {
  f ::= ...;   // override: origin = M, actual = N
}
```

Method definitions look like this:

method–name(*parameters...*) :> *return–type* ::= *body* ;

Each *parameter* has the form *name* :> *type*. If there are no parameters, the parentheses may be omitted. If the return type is void (§8.3), ':> *return–type*' may be omitted. *Body* is an expression (§9); it may be omitted, in which case any call of the method will result in a run-time error.[2] Here is the shortest method definition possible:

```
x::=;
```

Static method definitions have a static keyword before the method name.

Methods can be arbitrarily recursive, and the Prolac compiler turns tail-recursive methods into loops.

5.1 Static and dynamic methods

A dynamic method defined in module M is called with reference to some object whose type is either M or one of M's subtypes. Within the method body, this object is called self. Any dynamic method call must provide a value for self; this is done with member operator syntax (§9.6). The intuition is that the method is also a member of the module. For example:

```
module M {
  r ::= ...;
}
module N has M {
  ... let m :> M in
    m.r
  end ...
}
```

It is an error to refer to a dynamic method without reference to an object. This is a special case of the rules for static and dynamic context described in §9.6.

Within a dynamic method, 'self.' can be elided in field and method references. For example, these two method definitions are identical:

```
module M {
  method ::= ...;
  field f :> ...;
  d1 ::= self.method, self.f;
  d2 ::= method, f;
}
```

It is an error to refer to 'self', explicitly or implicity, in a static method.

2. Such a method can still be overridden, however.

5.2 Overriding and dynamic dispatch

A method declared with the same name as a supertype's method is an *overriding method* (§4.5.3). This section describes the semantics of overriding.

Both dynamic and static methods may be overridden, but dynamic dispatch only occurs on dynamic methods.

5.2.1 Correctness

Every override of a static method MA by a static method MB is legal.

An override of dynamic method MA by dynamic method MB is correct only if MB's signature—i.e., the number and types of its parameters and its return type—agree with MA's by the usual contravariance rule. Specifically:

1. MA and MB must take the same number of parameters.
2. For corresponding parameter types PA and PB, we must have PA :> PB; that is, PA and PB are equal, or PB is a supertype of PA. Thus, the overriding parameters are the same as, or more general than, the overridden parameters. This ensures that any value passed as a parameter to MA is also valid as a parameter to MB.
3. For the methods' return types TA and TB, we must have TB :> TA; that is, TA and TB are equal, or TB is a subtype of TA. Thus, the overriding return type is the same as, or more specific than, the overridden return type.

It is an error to define an incorrect override.

Every override of a static method by a dynamic method, or a dynamic method by a static method, is an error.

5.2.2 Method selection

The process of deciding which method definition to use for a given dynamic method call is called *method selection*. Method selection depends on only one factor: the *run-time type* of the object which will become the method's self.

In Prolac, an object of type M, where M is a module, can be used in place of any of M's supertypes. Therefore, an object's run-time type, or the actual type of the object used at run time, can differ from its static type, or the type used to declare the object.

For objects of simple module type, the static type is always identical to the run-time type; Prolac's semantics are call-by-value, like C, rather than call-by-object, like Clu. An object's static and run-time types can differ only if the object is referenced through a pointer (§8.7) or the object is self. (Unlike any other value, self is actually a reference.) For example:

```
module M {
  f ::= ...;
}
module N :> M { ... }
module U has M, N { ...
  let m :> *M = ..., n :> *N = ... in
    m = n,   // *m has static type M but run-time type N,
    // as the m pointer actually points to the n object.
    m->f   // Within f, self will have run-time type N.
    ...
}
```

The method selected for a method call O.f depends then on the run-time type of O. Let the run-time type of O be T; then we select the most specific definition for f existing in T and its supertypes. More precisely, consider all possible definitions for f coming from T and all of its supertypes. Let these definitions be d1,…, dk, coming from modules M1,…, Mk. Because Prolac is restricted to single inheritance, the modules M1,…, Mk must form a total order under the supertype relation. Let Ms ∈ {M1,…, Mk} be the most specific module in this order—that is, we have Ms :> Mi for all Mi ∈ {M1,…, Mk}. The definition selected for the method call is then ds, the method definition from Ms.

5.2.3 'super'

A module can specify that its parent's definition for a method be used by calling the method through the special object super. For example:

```
module One {
  f :> int ::= 1;
}
module Two :> One {
  f :> int ::= 1 + super.f;   // returns 2
}
module Three :> Two {
  f :> int ::= 1 + super.f;   // returns 3
}
```

Except for its behavior in relation to dynamic methods, super acts exactly like self.

Any dynamic method in the module can use super to call any inherited method; it is not limited to calling the parent's version of the current method. For example:

```
module One { f :> int ::= 1; }
module Two :> One {
  f :> int ::= 2;
  old-f :> int ::= super.f;   // returns 1
}
```

Using super is not the same as calling a method through the parent's name, since calling the method through the parent's name still refers to the most specific definition of the method. For example:

```
module One { f :> int ::= 1; }
module Two :> One {
  f :> int ::= 2;
  test ::= f,   // calls Two.f
    One.f,   // also calls Two.f
    super.f,   // calls One.f
    super.One.f,   // also calls One.f
    One.super.f;   // error
}
```

5.3 Constructors

Each module may contain a special method, called its *constructor*, which is called when objects of the module type are created. (See §9.2.2 for more information on when constructors are called.) The constructor is distinguished by its name, which is the keyword 'constructor'. A constructor must appear in the module's top-level namespace; it must not be static and must not define a return type, but it can take parameters.

The body of a constructor method is parsed differently than those of normal methods. It consists of zero or more *subobject constructor expressions* separated by commas, followed by a normal expression. A subobject constructor expression is just a constructor call (§9.2.2) for the module's parent or one of its fields. For example:

```
module Counter {
  field c :> int;
  constructor(n :> int) ::= c = n;
}
module Counter2 :> Counter {
  field f :> Counter;
  constructor(n1 :> int, n2 :> int) ::=
    Counter(n1),   // parent constructor expression
    f.constructor(n2);   // field constructor expression
}
```

In a normal context, the subexpression 'Counter(n1)' would have no visible effect; it'd create a new Counter object, then throw away the result. As a parent constructor expression, however, it does have a visible effect—specifically, initializing the Counter module.

If a parent or field is not mentioned in a subobject constructor expression, its constructor is called without arguments. It is an error to omit a subobject constructor expression for a parent or field whose constructor requires arguments.

If no constructor is provided for a module, Prolac will generate a default constructor which calls any necessary parent or field constructors. The parent's constructor may take arguments, in which case the generated constructor will take the same number and types of arguments and pass them to the parent's constructor.

5.4 Implicit methods

Explicit methods are methods the user explicitly defines. *Implicit methods*, on the other hand, are created automatically when a Prolac expression refers to an undefined name (§4.5.4). The compiler will fill in the implicit method's definition by looking through the module's fields and imports, subject to any using and notusing module operators, until it finds a method with the same name. Implicit methods can considerably simplify the text of a module by eliding frequently-used object or module names.

A motivating example seems in order. Consider a module Segment-Arrives implementing part of the TCP protocol. This module will frequently refer to the current transmission control block, tcb, which has type *TCB. Here is a partial definition for a hypothetical TCB module:

```
module TCB {
  field state :> int;
  field flags :> int; ...
  // Which state are we in?
  listen ::= state == 0;
  syn-sent ::= state == 1;
  syn-received ::= state == 2;
  ...
}
```

Now, how should we implement Segment-Arrives? We want to divide computation into many small methods, so we could make tcb a parameter to each; however, passing the parameter would quickly become tiresome. Therefore, we make tcb a field in Segment-Arrives. Here is a sample of what our code might look like, considerably simplified for didactic purposes:

```
// Example 1
module Segment-Arrives has TCB {
  field tcb :> *TCB;
  check-segment ::=
    (tcb->listen ==> do-listen)
    || (tcb->syn-sent ==> do-syn-sent)
    || (tcb->syn-received ==> do-syn-received)
    || (tcb->established ==> do-established)
    ...;   // and much more!
}
```

The repetition of 'tcb->' is tedious and hinders quick comprehension of the code. We know Segment-Arrives deals with only one tcb; why should we have to tell the compiler which tcb we mean again and again?

One solution is to generate forwarding methods in Segment-Arrives. We hide these forwarding methods using after-module operators (§3.4.2), since they are artifacts of the implementation.

```
// Example 2
module Segment-Arrives has TCB {
  field tcb :> *TCB;
  check-segment ::=
    (listen ==> do-listen)
    || (syn-sent ==> do-syn-sent)
    ...;
  listen ::= tcb->listen;
  syn-sent ::= tcb->syn-sent;
  ...
} hide (listen, syn-sent, ...);
```

This is better; however, the forwarding methods clutter the module definition and, again, are tedious and error-prone to write.

The solution in Prolac is to use implicit methods. We use the using module operator (§5.4.3) to open tcb for implicit method search. When the compiler creates Segment-Arrives's original namespace, it searches its methods for undefined names, entering them in Segment-Arrives's top-level namespace as undefined implicit methods. Later, it creates their definitions through a search process. It marks the implicit methods as highly inlineable and hides them in the default namespace. Thus, the compiler transforms the following code into something like Example 2:

```
// Example 3
module Segment-Arrives has TCB {
  field tcb :> *TCB using all;
  check-segment ::=
    (listen ==> do-listen)
    || (syn-sent ==> do-syn-sent)
    ...;
}
```

In Example 3, unlike the earlier examples, nothing distracts the reader from exactly what the module is doing.

Implicit methods are meant to make code more readable rather than less. Their overuse can make code very difficult to understand, however; moderation is required.

Only methods and exceptions can be found implicitly: referring to a field always requires explicit syntax.

The remainder of this section describes the various mechanisms supporting implicit methods, specifically the implicit method search algorithm and the using and notusing module operators.

5.4.1 Implicit method search

Prolac allows implicit methods to be found in a module's supertypes. Therefore, this example will work:

```
module I {
  implicit ::= ...;
}
module S has I using all { }
module M :> S {
  ... implicit ...   // finds I.implicit
}
```

This is behavior is required for predictable programming. Consider, for example, inheriting from a module in order to extend it—not being able to refer to implicit names which the parent module used would be very counterintuitive. The following algorithm implements implicit method search, including parents, without surprising behavior.

To find an implicit method named *n* in module M, a breadth-first search is performed. First all of M's imports and fields are checked for a top-level method named *n*; then M's parent's imports and fields; then M's grandparent's imports and fields; and so on. The search continues backwards in the module hierarchy until there are no more ancestors or a definition is found.

A definition is found for *n* in some import or field iff:

1. That import or field has a module or pointer-to-module type;
2. That module has a visible method, exception, or namespace call (§9.2.1) named *n*; and
3. A 'using *n*' or 'using all' directive (§5.4.3) is in effect on that module.

A warning is given if, at any point during the search, the algorithm finds a field or a namespace that cannot be called (§9.2.1) instead of a method or exception.

If two or more definitions for *n* are found in the same generation of the search, the implicit method is ambiguous and an error is reported. For example, if two of M's fields define *n*, *n* is ambiguous; but also, if M's parent has two fields f1 and f2 which both have an *n* method, then *n* is ambiguous, and so on.

If a unique definition for *n* is found in any generation of the search, that definition is used. It is an error (specifically, an undefined variable error) if no definition is found.

There are some caveats. First, if the parent (grandparent, etc.) is closed off to implicit method search by an explicit 'notusing all', neither that parent nor its supertypes are searched.

Second, only static methods (§5.1) are considered in imports, and only dynamic methods are considered in fields. Thus, there is no ambiguity in this example:

```
module I {
  dyn ::= ...;
  static stat ::= ...;
}
module M has I using all {
  field f :> I using all;
  test ::=
    dyn,   // unambiguously f.dyn (I.dyn not considered)
    stat;  // unambiguously I.stat (f.stat not considered)
}
```

For purposes of implicit method search, exceptions are treated like static methods.

5.4.2 Implicit method definitions

Once the compiler finds an unambiguous definition *D* for an implicit method named *n*, it writes a forwarding definition for *n* which simply calls *D*. The forwarding definition depends on *D*: specifically, the new definition takes the same number of parameters with the same respective types and returns the same type as *D*.

If *D* was found in an import *I*, the definition will look like this:

```
static n(parameters) ::= I.n(parameters);
```

If *D* was found in a field *f*, there are two possibilities, depending on whether *f* has pointer-to-module type:

```
n(parameters) ::= f.n(parameters);
n(parameters) ::= f->n(parameters);
```

5.4.3 'using' module operator

The user controls implicit method search through the using and notusing module operators. The using operator makes a module's names available for implicit method search. Its left operand must be a module expression; its right operand must be a list of simple names (qualified names can never be implicit methods, anyway), or either 'all' or 'allstatic'.

Here is a simple example:

```
module M {
  static f ::= ...;
}
module U has M {
  ... f ...   // error: 'f' undefined
}
module U2 has M using f {
  ... f ...   // OK
}
```

'using all' makes all of a module's top-level names available for implicit method search, while 'using allstatic' makes all of a module's *static* top-level names available for implicit method search.

Note that any field module types are actually references to a module's import list. This can lead to more using directives than you want:

```
module U1 has M using all {
  field m1 :> M;
  field m2 :> M;
}
// is equivalent to...
module U2 has M using all {
  field m1 :> M using all;
  field m2 :> M using all;
}
```

Any reference to a dynamic implicit method from M will result in an ambiguity between m1's definition and m2's. To fix this situation, use notusing, or say 'using allstatic' in the module header instead of 'using all'.

5.4.4 'notusing' module operator

The notusing operator hides a module's names from implicit method search. Its left operand must be a module expression; its right operand must be a list of simple names or 'all'. (You can't say 'notusing allstatic'.)

5.4.5 Notes

Once they are defined through the implicit method search defined above, implicit methods are treated identically to normal methods by the language. In particular, this means that *implicit methods can be overridden* in a module's subtypes, although this will not normally happen because implicit methods are hidden by default.

5.5 Export specifications

Prolac does not, by default, generate a C function definition for every method in the Prolac program; rather, *export specifications* tell Prolac which methods to generate. Export specifications are placed outside of any module in the Prolac input file. Here is the syntax for an export specification:

```
export module.method [, module.method ...]
```

Module must be a module name (possibly preceded by namespace qualifiers); *method* can be a method from that

module or 'all', which means "export all methods defined in *module*".

Prolac collects all export specifications and generates code for the methods they mention in arbitrary order. It then recursively generates code for all the methods they call, the methods those methods call, and so on, until it reaches closure.

6 Fields

Fields are module-specific variables. Like methods (§5.1), fields can be static or dynamic; dynamic fields are like instance variables or slots in other object-oriented languages, while static fields are more like global variables. Each object of a module type has its own copy of each of the module's dynamic fields, while only one copy exists of each of a module's static fields. Any field, static or dynamic, must be part of some module.

Fields are declared with the following syntax:

```
[static] field name :> type;
```

Remember that, if *type* is a module type, it must be a visible supertype of the current module or it must have been explicitly imported (§3.3).

A dynamic field can be referred to only in a dynamic context (§9.6); in a dynamic method, 'self.' can be omitted when referring to self's dynamic fields. Unlike methods, fields cannot be overridden and they cannot be found with any kind of implicit search.[3]

7 Exceptions

Exceptions in Prolac signal conditions that cause normal processing to terminate. Prolac exceptions have termination semantics, like exceptions in C++ or Java; but unlike those languages, a Prolac exception carries no information other than its name. You declare an exception inside a module as follows:

```
exception exception-name ;
```

Every exception declaration introduces a unique exception. Thus, two exceptions with the same name are distinct if they were declared in different modules. There are no restrictions on where exceptions can be raised or caught—an exception can be raised or caught wherever that exception can be named. In particular, the module that raises an exception need not be a subtype of the module that declared the exception.

3. Implicit method search abbreviates something the user could do herself (write forwarding methods). However, the user cannot create a "forwarding field".

7.1 Raising exceptions

You raise an exception simply by calling it, as if it were a method taking no parameters:

```
module M {
  exception invariant-failed;
  f ::= !check-invariant ==> invariant-failed;
}
```

If check-invariant returns false, the invariant-failed exception will be raised and control transferred to the most recently executed 'catch invariant-failed' or 'catch all'. If there was no such catch, the topmost Prolac method will return with a negative value.

Only methods returning void or bool may raise an exception. Methods returning more complex types must use catch expressions (§7.2) to handle any exceptions their bodies may throw. The compiler checks this invariant. For example:

```
module M {
  exception bad;
  f1 ::= bad;   // OK: f1 returns bool
  f2 :> int ::= bad;   // error
  f3 :> int ::= (bad catch bad), 5;
        // OK: the bad exception cannot escape f3
  f4 :> int ::= (f1 ? 1 : 2);
        // error: f1 can throw an exception f4 does not handle
}
```

Exceptions behave in expressions as if they had arbitrary type, so they may be used in any context without causing a type error.

7.2 Handling exceptions: 'catch'

The 'catch' operator is used to catch one or more exceptions within a subexpression. Its left operand is the subexpression; its right expression is an exception name, a comma-separated list of exception names, or 'all', which catches all exceptions. Some examples:

```
do-something catch X1
do-something, do-something-else catch (X1, X2, X3)
do-something-else-again catch all
```

An expression 'A catch X' has type bool. Its return value is true if one of the exceptions X was raised but not caught inside A; it is false otherwise.

The precedence of catch is very low on the left and very high on the right (§9.1). This lets you write sequential exception handlers without using parentheses. These two expressions are equivalent:

```
A catch X1 ==> H1          (( (( ((A catch X1) ==> H1)
  catch X2 ==> H2     ≡        catch X2) ==> H2)
  catch all ==> H3           catch all) ==> H3)
```

Here, H1 is executed if the exception X1 is raised within A; H2 is executed if X2 is raised within A or H1; and H3 is executed if any exception gets through the other handlers.

8 Types

This section describes the Prolac type system, including Prolac's built-in types and their values and allowable conversions between types. Prolac supports void, boolean, and arithmetic types, module types, and pointer and array types. It does not support function or method types.

The type declaration operator ':>' (§9.7) is used in Prolac to declare the types of fields, parameters, methods, and supertypes. We write 'V :> T' for "V has type T" or "V is a T".

Types can occur in Prolac expressions only in casts and to the right of ':>' (§9.7). Module types can also appear to the left of '.' (member access, §9.6) and as constructor calls (§9.2.2). It is an error if a type expression occurs inside a value expression in another context.

8.1 Converting and casting

The two processes of *conversion* (or, equivalently, *implicit conversion*) and *casting* convert a value from one type to another. Casting is strictly more powerful than conversion.

Prolac automatically invokes implicit conversion in any context where a value is expected to be of some type—for example, the test argument to a choice operator '?:' (§9.3.5) is expected to have type bool. Integral values and pointer values both implicitly convert to bool, as defined below, so these values are also acceptable as test arguments. The phrase "V is converted to T" means "V is implicitly converted to T if this is possible; if not, an error is reported."

Prolac never automatically casts a value; the user explicitly invokes a cast by using the type cast operator (§9.7). Whenever a value V can be implicitly converted to a type T, the explicit cast '(T)V' is also possible and has the same result.

8.2 Common types

Two types T1 and T2 always have a *common type*, which is used when T1 and T2 are combined in an expression; for example, the choice expression 'test ? V1 : V2', whose value may be either V1 :> T1 or V2 :> T2, has the common type of T1 and T2 as its type. Implicit conversions (never casts) are used to convert each operand to the common type.

The common type for two types T1 and T2 is found as follows:

1. If T1 and T2 are the same type T, the common type is T.
2. If either T1 or T2 is bool and the other can be implicitly converted to bool (i.e., it is bool or an integral or pointer type), the common type is bool.
3. If both T1 and T2 are integral types, the common type is the larger of them (see §8.5).
4. If both T1 and T2 are pointer types, then:
   (a) If either T1 or T2 is *void, the common type is *void.
   (b) If T1 and T2 are pointers to module types M1 and M2, then if either module is an supertype of the other, the pointer type to the supertype is returned.
5. Otherwise, the common type is void.

8.3 void

The void type signifies the absence of any value. Any expression can be implicitly converted to void. A void expression cannot be cast to any other type.

Because void implies the absence of a value, it is an error to declare a value (object, parameter, field, etc.) of type void. void is most useful as a method return type and as the base for the generic pointer type *void.

8.4 bool

bool is the Boolean type. It has two values, true and false.

Integral values implicitly convert to bool, with 0 converting to false and any non-zero value converting to true. Pointer types also implicitly convert to bool: the null pointer converts to false, any non-null pointer to true.

bool values may be explicitly cast to integral types; false casts to 0 and true casts to 1.

8.5 Integral types

Prolac has nine integral types: four signed types, char, short, int, and long; four unsigned types, uchar, ushort, uint, and ulong; and one unsigned type with circular comparison (§9.10.4), seqint. Their properties are summarized in this table; **size** is in bits:

Size	Signed	Unsigned	Circular
8	char	uchar	
16	short	ushort	
32	int	uint	seqint
64	long	ulong	

The common type of two integral types is the larger of the two types, as defined below.

- If either type is ulong, the common type is ulong.
- Otherwise, if either type is long, the common type is long.
- Otherwise, if either type is seqint, the common type is seqint.
- Otherwise, if either type is uint, the common type is uint.
- Otherwise, the common type is int.

8.6 Module types

Each defined module M is a distinct type. Note that two versions of the same module with different module operators (§3.4) do not define two different types; in terms of type, M is equivalent to 'M hide all inline x'.

If S is a supertype of M, a pointer of type *M may be implicitly converted to a pointer of type *S (§8.7). However, unlike most object-oriented languages, an object of type M cannot be converted or cast to an object of type S: only pointers exhibit true subtyping behavior.

8.7 Pointer types

If T is a type, then *T is also a type, representing a pointer to a value of type T. (Note that Prolac differs syntactically from C, where the '*' operator attaches to the declared name, not the type.)

As in C, *void is the generic pointer type: any pointer can be implicitly converted to type *void, and an object of type *void can be implicitly converted to any pointer type.

A pointer to a module M can be implicitly converted to a pointer to a module S, where S is an supertype of M. Furthermore, a pointer to S can be explicitly cast to type *M.

The integer constant 0 can be implicitly converted to any pointer type, resulting in a *null pointer* of the given type. The semantics of such a null pointer are the same as in C. Except for '0' itself, integer constant expressions evaluating to 0 are not acceptable null pointers.

8.8 Array types

If T is a type and k is a nonnegative static integer constant, then 'T[k]' is also a type, representing an array of k T objects. The array dimension k may be a symbolic constant (§9.2.3).

An object of type T[k] may be implicitly converted to a pointer object of type *T.

9 Expressions

This section describes the Prolac operators. Prolac is an expression-based language; unlike C, but like Lisp and ML, Prolac has no concept of statement (a control structure which is not an expression). This means that Prolac has even more operators and precedence levels than C (which, some might argue, already had too many); it also means that once you understand the Prolac operators, you can understand any computation expressed in Prolac.

Prolac operators fall into several categories. *Method calls* are discussed first (§9.2), followed by *control flow operators*, which control Prolac's order of computation. The *let operator* (§9.4) and *C blocks* (§9.5) come next, followed by *member operators* (§9.6), *typing operators* (§9.7), *code motion operators* (inline and outline, §9.8), and *C operators*, whose meanings are the same in Prolac as in C (§9.10).

9.1 Operator precedence

Table 9.1, on page 15, lists all of Prolac's operators and their precedences. Some operators do not have precedence as they textually contain all their subexpressions; these are listed at the bottom of the table. Of course, grouping parentheses can be used to override any default precedence or associativity.

9.2 Method calls

A method call expression '*f*(...)' expresses the execution of the specified method. Any *actual parameters* to the method are given inside the parentheses, separated by commas; they must match the method's declared parameters in number and type. The types need not match exactly: Prolac attempts to convert the actual parameters to the types of the declared parameters.

Calls to methods without parameters need not provide parentheses. For example:

```
module M {
  f ::= true;
  g ::= f(),   // call M.g
      g;   // also call M.g
}
```

Operators with precedence

23.	*e.n* .*n*	member §9.6
	e–>*n*	pointer to member §9.6
22.	*f*(*e1*, *e2*, ...)	method call §9.2
	e1[*e2*]	array reference
	e++ *e*––	postincrement, postdecrement
21.	(*T*)*e*	type cast §9.7
20.	module operators (§3.4):	
	M hide *n* *M* show *n*	namespace control §4.6
	M rename (*n1*=*n2*)	
	M using *n* *M* notusing *n*	implicit methods §5.4.3
	M inline[*k*] *n*	inlining §9.8.1
	M noinline *n* *M* defaultinline *n* *M* pathinline *n*	
19.	*e* :> *T*	type declaration §9.7
18.	**e*	dereference
	&*e*	address of
	+*e* –*e*	unary plus/minus
	~*e*	bitwise not
	!*e*	logical not
	++*e* ––*e*	preincrement, predecrement
	inline[*k*] *e*	inlining §9.8.2
	noinline *e* defaultinline *e*	pathinline *e*
17.	*e1* * *e2* *e1* / *e2* *e1* % *e2*	multiply, divide, remainder
16.	*e1* + *e2* *e1* – *e2*	add, subtract
15.	*e1* << *e2* *e1* >> *e2*	left and right shift
14.	*e1* < *e2* *e1* > *e2*	arithmetic compare
	e1 <= *e2* *e1* >= *e2*	
13.	*e1* == *e2* *e1* != *e2*	equality tests
12.	*e1* & *e2*	bitwise and
11.	*e1* ^ *e2*	bitwise xor
10.	*e1* | *e2*	bitwise or
9.	*e1* && *e2*	logical and
8.	*e1* || *e2*	logical or
7.	*e1* ? *e2* : *e3*	choice (R) §9.3.5
6.	*e1* = *e2* *e1* += *e2* *e1* –= *e2*	assignment and compound
	e1 min= *e2* *e1* max= *e2*	assignment (R)
	e1 *= *e2* *e1* /= *e2* *e1* %= *e2* *e1* <<= *e2* *e1* >>= *e2*	
	e1 &= *e2* *e1* ^= *e2* *e1* |= *e2*	
5.	outline[*k*] *e*	outlining §9.8.3
4.	*e1*, *e2*	comma §9.3.1
	e1 {*C*} *e2*	C block §9.5
3.	*e1* ==> *e2*	arrow (R) §9.3.3
2.	*e1* ||| *e2*	case §9.3.7
1.	*e* catch *n*	handle exception §7.2

Operators without precedence

(*e*)	grouping
if *e1* then *e2* else *e3* end	if-then-else §9.3.6
let *decls* in *e* end	let §9.4
min(*e1*, *e2*) max(*e1*, *e2*)	minimum, maximum §9.10.5

Table 9.1: Prolac operators and precedence levels. Operators higher in the table bind more tightly; (R) denotes right-associative operators. If a reference is not given, the operator is described in §9.10.

Note that raising an exception (§7) looks exactly like calling a method with no parameters.

9.2.1 Namespace call

To facilitate the use of namespaces, a namespace name may be treated as a method or exception call. If a namespace originally named n[4] is found within an expression where a namespace was not expected, Prolac looks in that namespace for a method or exception named *n* and uses that if it is found. For example:

```
module M {
  nest { nest ::= ...; }
  f ::= nest.nest,   // call M.nest.nest
     nest;   // also call M.nest.nest
}
```

The search is only performed one level deep. Thus, this is an error:

```
module M {
  nest { nest { nest ::= found-the-prize; } }
  f ::= nest;   // error:
     // namespace 'M.nest' cannot be called
}
```

In a normal Prolac expression (that is, any expression except supertype and import lists and operands to module operators), namespaces are expected in only one context: to the left of a direct member operator '.' (§9.6). Thus, this search is performed everywhere except to the left of '.'.

9.2.2 Constructor calls

Calls to a module's constructor (§5.3) are used to initialize an object of Prolac module type. There are several ways to call constructors; some of them allocate stack memory to hold the object, while some initialize externally allocated memory.

It is an error to call a constructor in any manner when the constructor has been hidden (§4.6.1). This allows a module to suggest that no objects of that module type should be created except through visible interface functions.[5]

Normal constructor calls. An expression 'M(*args*)' where M is a module is a normal constructor call. This expression allocates stack memory for an M object, initializes that object by calling M's constructor with *args* as arguments, and returns the object.

4. That is, before any show or rename operations were applied.
5. The module's user could always show the constructor.

Just as with method calls, the actual arguments to a constructor call must match the constructor's declared parameters in number and type. Parentheses cannot be omitted from a normal constructor call. That is, an expression like 'M.f' is always interpreted as a reference to a static feature of M, rather than a constructor call like 'M().f'. To force the constructor interpretation, simply use parentheses.

Direct constructor calls. Externally allocated memory can be initialized by calling its constructor directly, as if it was a normal method. For example:

```
module M {
  constructor(i :> int) ::= ...;
  static new :> *M ::= let ptr :> *M in
    { ptr = malloc(sizeof(M)); }
    ptr->constructor(97),   // direct constructor
    ptr end;
}
```

Again, the number and types of arguments must match the declared constructor.

Implicit constructor calls ('let'). A let binding (§9.4) without a value expression will call constructors implicitly whenever the specified type is a module type. For example:

```
module M {
  constructor(i :> int) ::= ...;
  f ::= let m1 :> M   // same as 'm1 :> M = M()':
      // error: too few arguments to 'M.constructor'
    in let m2 :> M(5)   // same as 'm2 :> M = M(5)': OK
     in 0 end end;
}
```

9.2.3 Symbolic constants

Symbolic constants may be defined in Prolac by creating a static method whose body evaluates to a constant:

```
module M {
  static six :> int ::= 6;
  static thirty-two :> int ::= 1 << (six - 1);
}
```

The Prolac compiler will inline calls to such methods by default.

Calls to static constant methods are allowed in Prolac wherever a constant integer is expected—in the optional argument to inline, for example. An explicit noinline operator will prevent a static constant method from being usable as a Prolac constant (§9.8.1). If a static constant method without parameters is called inside a C block (§9.5), its constant value is used instead.

9.3 Control flow operators

Control flow operators are semistrict: they do not always evaluate all of their operands. Different control flow operators express sequencing, choice or if-then-else semantics, conjunction, disjunction, conditional execution, and case statements.

9.3.1 Comma: ','

The comma operator ',' expresses sequencing: an expression 'A, B' first evaluates A, then throws the result away and returns the result of B.

A and B can each have any type. The expression's type is the type of B.

9.3.2 Logical and: '&&'

The logical and operator '&&' expresses conjunction: an expression 'A && B' evaluates to true iff both A and B are true. A is evaluated first; if it is false, the whole expression must be false, and B is not evaluated at all.

Both A and B are converted to bool. The expression also has type bool.

9.3.3 Arrow: '==>'

The arrow operator '==>' expresses conditional execution. An expression 'A ==> B' evaluates to true iff A is true—but in that case, B is evaluated before the expression returns.

A is converted to bool, but B can have any type. The expression has type bool (but see below §9.3.7).

The expression 'A ==> B' is exactly equivalent to 'A && (B, true)', except for its behavior within case bars (§9.3.7). The arrow operator is useful for building "case statements"; for example, this code will evaluate else-case only if condition-1 and condition-2 are both false:

```
(condition-1 ==> case-1)
|| (condition-2 ==> case-2)
|| else-case
```

At most one of case-1, case-2, and else-case will be executed.

Case statements that return non-bool values can be built from arrow operators and case bars (§9.3.7).

9.3.4 Logical or: '||'

The logical or operator '||' expresses disjunction: an expression 'A || B' evaluates to true iff either A is true, B is true, or both. A is evaluated first; if it is true, the whole expression must be true, and B is not evaluated at all.

Both A and B are converted to bool. The expression also has type bool.

'A || B' is also legal if B has type void. In this case, 'A || B' has type void, and is shorthand for 'A ? (void)0 : B'. Prolac never tries to convert B to void—this definition is only used if B's type is void without conversion.

9.3.5 Choice: '?:'

The question mark–colon operator '?:'—also called the choice operator—expresses choice. An expression 'A ? B : C' first evaluates A. If A is true, it returns B (without evaluating C); otherwise, it returns C (without evaluating B).

A is converted to bool; the type of the expression is the common type of B and C.

9.3.6 'if-then-else'

The if-then-else operator is another way to express choice. The full syntax for if-then-else is as follows:

```
if condition
then case–1
[elseif condition–2 then case–2]...
[else else–case]
end
```

This expression is a synonym for:

```
condition ? case–1
  [: condition–2 ? case–2]...
  : else–case
```

The type of the if-then-else expression is the common type of all *case*s. The *else–case* can be omitted, forming an if-then expression; if it is omitted, the type of the expression is void.

'if–then–else' is provided primarily as an alternative to '?:' for larger expressions—the '?:' syntax becomes difficult to read very quickly when its operands are large.

9.3.7 Case bars: '|||'

The case bar operator '|||', in conjunction with the arrow operator (§9.3.3), expresses a case statement returning a meaningful value.

A case statement has this general form:

```
condition–1 ==> consequent–1
||| condition–2 ==> consequent–2 ...
||| else–case
```

Exactly one of the *consequent*s or *else–case* is executed, depending on which, if any, *condition* is true. The result of that *consequent* or *else–case* is returned as the value of the expression.

The case statement's syntax is based on matching constructs from functional programming languages; here, however, the *condition*s are all converted to bool. The case statement is exactly analogous to Lisp's cond special form.

Case bars are actually syntactic sugar for choice operators (§9.3.5). Given an expression containing '|||', the compiler repeatedly applies the following transformations until no '|||'s remain:

1. A ==> B ||| X ⇒ A ? B : X
2. (A ? B : C) ||| X ⇒ A ? B : (C ||| X)
3. (A, B) ||| X ⇒ A, (B ||| X)
4. A ||| X ⇒ A || X

Note that '|||'s used outside the context of a case statement reduce to normal logical ors, '||' (§9.3.4).

Here is a demonstration of the transformation rules:

```
A ==> B ||| C ==> D ||| E
⇒ (A ==> B ||| C ==> D) ||| E   // left-associative
⇒ (A ? B : (C ==> D)) ||| E   // rule 1
⇒ A ? B : ((C ==> D) ||| E)   // rule 2
⇒ A ? B : (C ? D : E)   // rule 1
```

The type of the whole expression is therefore the common type of B, D, and E.

A case statement without a final *else–case* usually has type bool. To see why, consider this expansion:

```
A ==> B ||| C ==> D
⇒ A ? B : (C ==> D)  // rule 1
```

Since there are no case bars remaining, expansion is over. The type of the expression is then the common type of B and 'C ==> D'; but the type of 'C ==> D' is just bool, so B will be converted to bool if possible.

9.4 'let'

The let operator, like the let operator in many functional languages, introduces new statically-bound variables within a subexpression. The syntax of let is as follows:

```
let variable [:> type] [= value]
   [, variable [:> type] [= value] ...]
in body end
```

Variable is just an identifier. *Type* is a type expression and *value* is a value expression; either *type* or *value* may be omitted, but not both. If *value* is omitted, *variable* is constructed implicitly if it has module type (§9.2.2), or left uninitialized otherwise; if *type* is omitted, *variable*'s type is the type of the *value* expression. If neither is omitted, the *value* expression is converted to type *type*. It is an error if *value* cannot be converted to *type*.

To evaluate a let expression, Prolac first evaluates the *value* expressions and any necessary *type* constructors in an arbitrary order. The resulting values are then bound to the *variable*s. Finally, *body* is evaluated with these bindings in force. The value of the let expression is the value of *body*; the type of the let expression is the type of *body*. Note that a let expression's variables are not visible to any of its *type* or *value* expressions.

9.5 C blocks: '{...}'

A C block '{...}' is used to execute C code at a given point during the execution of a method. The type of a C block is bool and its value is always true.

C blocks are special syntactically: a C block acts like a Prolac value—specifically, 'true'—with implicit comma operators on either side. Some examples will make things clearer; the implicit true is shown when necessary.

```
{A}          ≡ {A}, true
{B} X        ≡ {B}, X
X {C}        ≡ X, {C}, true
X {D} Y      ≡ X, {D}, Y
X ==> {E}    ≡ X ==> ({E}, true)
X ==> Y {F}  ≡ X ==> (Y, {F}, true)
X = Y {G} Z  ≡ (X = Y), {G}, Z
X ? Y : Z {H} ≡ (X ? Y : Z), {H}, true
```

Some Prolac names can be used in a C block to refer to the C equivalents of those Prolac objects. Specifically, the following objects are available under their Prolac names:

1. In a dynamic method, self.
2. In a dynamic method, any of self's fields or fields of its ancestors.
3. Any static fields of self, its ancestors, or its imports.
4. Parameters from the current method.
5. Variables bound by surrounding let expressions (§9.4).
6. Static constant methods without parameters (§9.2.3).

This list does not include arbitrary Prolac methods: you cannot call most Prolac methods from C blocks using Prolac syntax.

Unlike Prolac, C does not allow hyphens in identifiers (§2.1). Prolac follows C's rules while parsing C blocks; thus, this C block will not work as expected:[6]

```
let thing-1 = 0 in { return thing-1; } end
```

6. Or maybe it is the Prolac code that does not work as expected.

To refer to an object with a hyphen in its name, simply change the hyphen to an underscore:

```
let thing-1 = 0 in { return thing_1; } end
```

Objects of type seqint are treated as unsigned integers inside C blocks. In particular, comparisons like 'seq1 < seq2' are unsigned, not circular (§9.10.4).

9.6 Member operators: '.' and '->'

The member operators '.' and '->' express finding a feature in a namespace or object. The right operand of a member operator must be an identifier. The pointer-to-member operator '->' is used on pointer types (§8.7); the expression 'A->x' is exactly equivalent to '(*A).x'. The rest of this section discusses only the direct-member operator, '.'.

The left-hand operand, or "object operand", of a member expression must be a namespace or have a module type. The object operand may be either a type or value expression. If it is a type expression, the member has *static context;* if it is a value expression, it has *dynamic context*. It is an error to refer to a static feature in a dynamic context, or a dynamic feature in a static context. To illustrate:

```
module M {
  field d :> int;
  static s ::= 0;
}
module N has M {
  field m-object :> M;
  test ::=
    M.s,   // OK: static context, static method
    m-object.d,   // OK: dynamic context, dynamic field
    M.d,   // error: static context, dynamic field
    m-object.s;   // error: dynamic context, static method
}
```

Fields, parameters, and objects always have dynamic context, while Imports always have static context. Supertypes are a special case: In a static method, supertypes have static context. In a dynamic method, supertypes have dynamic context; however, in this case and this case only, you may refer to a supertype's static feature, even though the supertype has dynamic context.

The direct member operator '.' also has a prefix version '.x', used to look up names in the global namespace (§4.3).

9.7 Type operators: ':>' and '(cast)'

The type declaration operator ':>' is used elsewhere in Prolac to declare the types of objects; inside a value expression, ':>' expresses a type assertion. The right operand of ':>'

should be a type expression, T. The value of an expression 'V :> T' is the value of V converted (§8.1) to type T. Note that the ':>' operator will only use implicit conversions on V; an error is given if V cannot be implicitly converted to T. Thus, ':>' can be used to guarantee that V has type T without invoking a possibly dangerous type cast.

The type casting operator '(*type*)' is used to change the type of its value operand. The value of an expression '(T)V', where T is a type expression, is the value of V cast (§8.1) to type T.

9.8 Code motion operators

Prolac provides two operators to control optimization and code motion, inline and outline. inline is also available as a module operator.

9.8.1 'inline' module operator

The inline module operator controls how a module's methods are inlined. Its left operand must be a module expression; its right operand must be a list whose elements are one of the following kinds of expressions:

1. A simple method name, or the name of a namespace that can be called (§9.2.1); the corresponding method will be affected.
2. 'NS.all', where NS is a namespace; all methods defined in NS or any of its nested namespaces will be affected.
3. 'M.all', where M is a supertype of the module expression; all methods defined by M (specifically, whose actual is M; see §5) will be affected.
4. 'all'; all methods will be affected.

In addition, inline can take an optional *inline level* argument, which must appear in brackets directly after the inline keyword. An inline level must evaluate to an integer constant whose value is between 0 and 3. 0 means "do not inline under any circumstances", 1 means "do not inline", 2 means "inline", and 3 means "path inline" (described below). If no argument is given, the inline level defaults to 2.

When a method call expression (§9.2) is evaluated, Prolac checks the corresponding module for that method's inline level; if it is high enough and the method call is unambiguous (i.e., no dynamic dispatch is possible), the method call will be inlined. For example:

```
module M {
  method ::= ...;
}
module N has M {
  test(r1 :> M, r2 :> M inline all) ::=
    r1.method,  // not inlined
    r2.method;  // inlined
}
```

A method call labeled with inline level 3 is *path inlined*: any methods called by the first method are inlined as well. For example:

```
module M {
  a ::= { XXX; };
  b ::= a;
  d ::= inline[3] b;
  // generates code like 'd ::= { XXX; };'
  // rather than 'd ::= a;'
}
```

This recursive inlining can be stopped by inline[0]. Recursive or mutually recursive methods are currently inlined only one level deep, except for tail recursive methods, which are inlined into loops.

The operators noinline, defaultinline and pathinline are equivalent to inline[0], inline[1] and inline[3], respectively.

9.8.2 Expression 'inline'

Expression inline is a prefix unary operator appearing in method bodies (the inline module operator is a binary operator appearing in module expressions). Like the inline module operator, expression inline can take an optional inline level argument in brackets, and the noinline, defaultinline, and pathinline module operators have expression equivalents.

An expression 'inline[*n*] X' simply evaluates X and returns its value; the type of 'inline[*n*] X' is the type of X. However, any method calls within X are inlined with inline level *n*. Expression inline overrides any inline module operators.

9.8.3 'outline'

The prefix unary operator outline appears in method bodies and controls *code outlining*, the removal of infrequently-executed code from a computation's critical path. The outline operator tells the compiler that the current branch of control flow is relatively unlikely; the compiler will move that branch to the end of the function body in the code it generates. This tends to improve i-cache utilization.

Like the inline operators, outline it takes an optional static integer constant argument which must be between 0 and 10. Here, 0 means "never outline", while 10 means "outline as far as possible"; or, equivalently, 0 means "this is the most common branch" and 10 means "this is the least common branch".

An expression 'outline[*n*] X' evaluates X and returns its value; its type is the type of X.

The outline operator is only meaningful on the destination of a code branch, such as to the right of '&&', '||', or '==>', or on either consequent of a choice operator (§9.3.5, §9.3.6, §9.3.7). To see why, consider the expression 'A + (outline B)'. This expression suggests that B is less likely to be executed than A; but this is impossible, since (unless A generates an exception) B will be executed whenever A is. Prolac does not warn on such expressions; rather, it floats the outline into the expression until it finds a control flow operator, and attaches the outline onto that operator's rightmost operand. Thus, these pairs of expressions are equivalent:

```
A + (outline B ? C : D)  ≡  A + (B ? C : (outline D))
outline A || B           ≡  A || (outline B)
```

9.9 Lvalues

Some expressions are *lvalues,* meaning that they can appear on the *l*eft side of an assignment expression. Only the following expressions are lvalues:

1. Field (§6), parameter, or let-bound variable (§9.4) references;
2. Dereferences '*X', where X is an object of pointer type (§9.10.1);
3. '(X)', where X is an lvalue.

9.10 C operators

This section describes the remaining operators, whose definitions are generally borrowed from C.

9.10.1 Dereference: unary '*'

The unary star or dereference operator '*' acts differently depending on whether its operand X is a type or value expression.

If X is a value expression, then it must have some pointer type *T, but not *void. The expression '*X' then has type T; its value is the value of the object to which X points.

If X is a type expression, then '*X' is also a type expression defining the type "pointer to X" (§8.7).

9.10.2 Address of: unary '&'

The operand in an address expression '&X' must be an lvalue (§9.9). The value of the expression is a pointer to X; it has type '*T', where T is the type of X.

9.10.3 Equality tests: '==', '!='

Any two non-module values may be compared for equality. In an expression 'X == Y', X and Y are both converted to their common type (§8.2), which must not be void; as a special case, any pointer can be compared with the integer constant 0, which is converted to a null pointer. The result has type bool, and is true iff X and Y are equal.

The expression 'X != Y' is a synonym for '!(X == Y)'.

9.10.4 Arithmetic compare: '<', '<=', '>', '>='

The operands to an arithmetic compare operation are converted to their common type, which must be an integral or pointer type. Any integer value can be compared to any other, and two pointers of the same type can be compared. As a special case, any pointer can be compared with the integer constant 0, which is converted to a null pointer (§8.7). An arithmetic compare expression has type bool.

If the common type is seqint, a circular comparison is performed modulo 2^{32}. The circular comparison operators are defined as follows:

```
seq1 < seq2   ≡  (int)(seq1 - seq2) < 0
seq1 <= seq2  ≡  (int)(seq1 - seq2) <= 0
seq1 > seq2   ≡  (int)(seq1 - seq2) > 0
seq1 >= seq2  ≡  (int)(seq1 - seq2) >= 0
```

This definition corresponds to comparison on TCP sequence numbers.

9.10.5 Minimum and maximum

The minimum and maximum operators are defined as follows, except that complex subexpressions will be evaluated exactly once.

```
min(X, Y)  ≡  X < Y ? X : Y
max(X, Y)  ≡  X > Y ? X : Y
X min= Y   ≡  X = min(X, Y)
X max= Y   ≡  X = max(X, Y)
```

'min' and 'max' use circular comparison on seqints.

9.10.6 Logical not: '!'

The operand to a logical not expression '!X' is converted to bool; the expression has type bool. If the value of X is true, the expression has value false, and vice versa.

9.10.7 Arithmetic operators

The binary arithmetic operators are addition '+', subtraction '-', multiplication '*', division '/', remainder '%', left '<<' and right '>>' shift, bitwise and '&', bitwise or '|', and bitwise exclusive or '^'. The unary arithmetic operators are unary plus '+' and minus '-' and bitwise not '~'.

The operands to most arithmetic operators must have integral type. The type of a unary arithmetic expression is the type of its operand; the type of a binary arithmetic expression is the common type of its operands.

Binary addition and subtraction also support some combinations of pointer operands. In an addition expression 'A + B':

- Either A or B may be a pointer; the other must have integral type. The result has the type of the pointer operand.

In a subtraction expression 'A – B':

- A and B may be pointers of the same type. The result has type int.
- A may have pointer type and B may have integral type. The result's type is the type of A.

All arithmetic operators behave as they do in C.

9.10.8 Array reference: '[]'

The bracket operator '[]' expresses array reference. The expression 'X[Y]' acts differently depending on whether the left operand X is a type or value expression.

If X is a value expression, then it must have some pointer or array type *T, but not *void. The right operand Y must have some arithmetic type, and the type of the whole expression is T. The expression behaves similar to the similar array reference in C.

If X is a type expression, then Y must evaluate to a nonnegative static integer constant. If so, 'X[Y]' is a type expression defining the type "array of Y Xs" (§8.8).

9.10.9 Assignment: '='

The assignment operator '=' expresses variable assignment. In an expression 'X = Y', X must be an lvalue (§9.9); Y is converted to the type of X. The expression has the type of X; its value is the value of X after the assignment is performed.

9.10.10 Compound assignment

A compound assignment expression 'X @= Y', where @ is a binary arithmetic operator (§9.10.7), is exactly equivalent to the expression 'X = X @ Y' except that X is evaluated only once.

9.10.11 Increment and decrement: '++', '––'

The increment and decrement operators '++' and '––' are used to increment or decrement an lvalue by 1. Their operand must be an lvalue with arithmetic or pointer type; the result of the expression has the same type. The following table shows equivalent expressions for each increment and decrement operator, except that postfix increment and decrement evaluate their operand only once.

++X ≡ X += 1
––X ≡ X –= 1
X++ ≡ let *temp* = X in X += 1, *temp* end
X–– ≡ let *temp* = X in X –= 1, *temp* end