Alef Language Reference Manual

Phil Winterbottom
philw@plan9.att.com

Introduction

Alef is a concurrent programming language designed for systems software. Exception handling, process management, and synchronization primitives are implemented by the language. Programs can be written using both shared variable and message passing paradigms. Expressions use the same syntax as C, but the type system is substantially different. Alef supports object-oriented programming through static inheritance and information hiding. The language does not provide garbage collection, so programs are expected to manage their own memory. This manual provides a bare description of the syntax and semantics of the current implementation.

Much of the terminology used in this manual is borrowed from the ANSI C language reference manual and the Plan 9 manual. The manual expects familiarity with both.

1. Lexical

Compilation starts with a preprocessing phase. An ANSI C preprocessor is used. The preprocessor performs file inclusion and macro substitution. Comments and lines beginning with the # character are consumed by the preprocessor. The preprocessor produces a sequence of tokens for the compiler.

1.1. Tokens

The lexical analyzer classifies tokens as: identifiers, typenames, keywords, constants, and operators. Tokens are separated by white space, which is ignored in the source except as needed to separate sequences of tokens which would otherwise be ambiguous. The lexical analyzer is greedy: if tokens have been consumed up to a given character, then the next token will be the longest subsequent string of characters that forms a legal token.

1.2. Reserved Words

The following keywords are reserved by the language and may not be used as identifiers:

```
adt         aggr        alloc
alt         become      break
byte        case        chan
check       continue    default
do          else        enum
extern      float       for
goto        if          int
intern      lint        nil
par         proc        raise
rescue      return      sint
sizeof      switch      task
tuple       typedef     typeof
uint        ulint       unalloc
union       usint       void
while       zerox
```

The following symbols are used as separators and operators in the language:

```
+     -     /     =
>     <     !     %
&     |     ?     .
"     '     {     }
[     ]     (     )
*     ;
```

The following multi-character sequences are used as operators:

```
+=    -=    /=    *=
%=    &=    |=    ^=
<<=   >>=   ==    !=
--    <-    ->    ++
::    :=
```

1.3. Comments

Comments are removed by the preprocessor. A comment starts with the characters /* and finishes at the characters */. A comment may include any sequence of characters including /*. Comments do not nest.

1.4. Identifiers

An identifier, also called a lexical name, is any sequence of alpha-numeric characters and the underscore character _. Identifiers may not start with a digit. Identifiers are case sensitive. All characters are significant. Identifiers beginning with the string ALEF are reserved for use by the runtime system.

1.5. Constants

There are five types of constant:

constant:
  integer-const
  character-const
  floating-const
  string-const
  rune-string-const

An integer constant is a sequence of digits. A prefix may be used to modify the base of a number. Defined prefixes, bases, and digit sets are:

none	decimal	0–9
0x	hexadecimal	0–9 a–f A–F
0	octal	0–7

A character constant contains one or more characters surrounded by single quote marks '. If the constant contains two or more characters the first must be the escape character \. The following table shows valid characters after an escape and the value of the constant:

0	NUL	Null character
n	NL	Newline
r	CR	Carriage return
t	HT	Horizontal tab
b	BS	Backspace
f	FF	Form feed
a	BEL	Beep
v	VT	Vertical tab
\	\	Backslash
"	"	Double quote

Character constants have the type `int`. The range of values they hold depends on the character set. In Plan 9, the input text is in UTF and character constants hold the 16-bit representation of the Unicode character (see *rune*(6) in Volume 1 of the Plan 9 Programmer's Manual).

A floating point constant consists of an integer part, a period, a fractional part, the letter e and an exponent part. The integer, fraction and exponent parts must consist of decimal digits. Either the integer or fractional parts may be omitted. Either the decimal point or the letter e and the exponent may be omitted. The integer part or period and the exponent part may be preceded by the unary + or − operators. Floating point constants have the type `float`.

A string constant is a sequence of characters between double quote marks ". A string has the type 'static array of byte'. A NUL (zero) character is automatically appended to the string by the compiler. The effect of modifying a string constant is implementation dependent. The `sizeof` operator applied to a string constant yields the number of bytes including the appended NUL.

A rune string constant is a sequence of Unicode characters introduced by $" and terminated by ". A rune string has the type 'static array of usint'. A zero rune character is automatically appended to the string by the compiler. The `sizeof` operator applied to a rune string constant yields the number of runes, including the appended zero, times `sizeof (usint)`.

1.6. Programs

An Alef program is a list of declarations stored in one or more source *files.* The declarations introduce identifiers. Identifiers may define variables, types, functions, function prototypes, or enumerators. Identifiers have associated *storage classes* and *scope* (see Section 2). For functions and variables declared at the file scope the storage class determines if a definition can be accessed from another file.

1.7. Processes and Tasks

The term *process* is used to refer to a preemptively scheduled *thread* of execution. A process may contain several tasks. A *task* is a non-preemptively scheduled coroutine within a process. The memory model does not define the sharing of memory between processes. On a shared memory computer processes will typically share the same address space. On a multicomputer processes may be located on physically distant nodes with access only to local memory. In such a system processes would not share the same address space, and must communicate using message passing.

A group of tasks executing within the context of a process are defined to be in the same address space. Tasks are scheduled during communication and synchronization operations. The term *thread* is used wherever the distinction between a process and a task is unimportant.

2. Definitions and Declarations

A declaration introduces an identifier and specifies its type. A definition is a declaration that also reserves storage for an identifier. An object is an area of memory of known type produced by a definition. Function prototypes, variable declarations preceded by `extern`, and type specifiers are declarations. Function declarations with bodies and variable declarations are examples of definitions.

2.1. Scope

Identifiers within a program have scope. There are four levels of scope: local, function, type, and file:

- A local identifier is declared at the start of a block. A local has scope starting from its declaration to the end of the block in which it was declared.
- Exception identifiers and labels have the scope of a function. These identifiers can be referenced from the start of a function to its end, regardless of position of the declaration.
- A member of a complex type is in scope only when a dereferencing operator **.** or −> is applied to an object of the type. Hidden type members have special scope and may only be referenced by function members of the type.
- All definitions outside of a function body have the scope of file. Unqualified declarations at the file scope have static storage class.

2.2. Storage classes

There are three storage classes: automatic, parameter and static. Automatic objects are created at entry to the block in which they were declared. The value of an automatic is undefined upon creation. Automatic variables are destroyed at block exit. Parameters are created by function invocation and are destroyed at function exit. Static objects exist from invocation of the program until termination. Static objects which have not been initialized have the value 0.

3. Types

A small set of basic types is defined by the language. More complex types may be derived from the basic types.

3.1. Basic types

The basic types are:

name	size	type
byte	8 bits	unsigned byte
sint	16 bits	signed short integer
usint	16 bits	unsigned short integer
int	32 bits	signed integer
uint	32 bits	unsigned integer
float	64 bits	floating point
lint	64 bits	long signed integer
ulint	64 bits	unsigned long integer
chan	32 bits	channel
poly	64 bits	polymorphic type

The size given for the basic types is the minimum number of bits required to represent that type. The format and precision of `float` is implementation dependent. The `float` type should be the highest precision floating point provided by the hardware. The `lint` and `ulint` types are not part of the current implementation but have been defined. The alignment of the basic types is implementation dependent. Channels are implemented by the runtime system and must be allocated before use. They are the size of a pointer. Polymorphic types are represented by a pointer and a tag representing the type. For a given implementation the polymorphic type has the same size as the following aggregate definition:

```
aggr Polytype
{
        void*   ptr;
        int     tag;
};
```

The `void` type performs the special task of declaring procedures returning no value and as part of a derived type to form generic pointers. The `void` type may not be used as a basic type.

3.2. Derived types

Types are derived in the same way as in C. Operators applied in declarations use one of the basic types to derive a new type. The deriving operators are:

*	create a pointer to
&	yield the address of
()	a function returning
[]	an array of

These operators bind to the name of each identifier in a declaration or definition. Some examples are:

```
int     *ptr;   /* A pointer to an integer */
byte    c[10];  /* A vector of 10 bytes */
float   *pow(); /* A function returning a pointer to float */
```

Complex types may be built from the basic types and the deriving operators. Complex types may be either aggregates, unions, tuples, or abstract data types (ADT). These complex types contain sequences of basic types and other derived types. An aggregate is a simple collection of basic and derived types. Each element of the aggregate has unique storage. An abstract data type has the same storage allocation as an aggregate but also has a set of functions to manipulate the type, and a set of protection attributes for each of its members. A union type contains a sequence of basic and derived types that occupy the same storage. The size of a union is determined by the size of the largest member.

The declaration of complex types introduces *typenames* into the language. After declaration a typename can be used wherever a basic type is permitted. New *typenames* may be defined from derived and basic types using the `typedef` statement.

The integral types are `int`, `uint`, `sint`, `usint`, `byte`, `lint` and `ulint`. The arithmetic types are the integral types and the type `float`. The pointer type is a type derived from the & (address of) operator or derived from a pointer declaration. The complex types are `aggr`, `adt`, and `union`.

3.3. Conversions and Promotions

Alef performs the same implicit conversions and promotions as ANSI C with the addition of complex type promotion: under assignment, function parameter evaluation, or function returns, Alef will promote an unnamed member of a complex type into the type of the left-hand side, formal parameter, or function.

4. Declarations

A declaration attaches a type to an identifier; it need not reserve storage. A declaration which reserves storage is called a definition. A program consists of a list of declarations:

program:
    declaration–list

declaration–list:
    declaration
    declaration–list declaration

A declaration can define a simple variable, a function, a prototype to a function, an ADT function, a type specification, or a type definition:

declaration:
    simple–declarations
    type–declaration
    function–declaration
    type–definition

4.1. Simple declarations

A simple declaration consists of a type specifier and a list of identifiers. Each identifier may be qualified by deriving operators. Simple declarations at the file scope may be initialized.

simple−declarations:
  type−specifier simple−decl−list ;

simple−decl−list:
  simple−declaration
  function−prototype
  simple−decl−list , *simple−declaration*

function−prototype:
  $pointer_{opt}$ *identifier* $array\text{−}spec_{opt}$ (*arglist*) ;
  ($pointer_{opt}$ *identifier* $array\text{−}spec_{opt}$) (*arglist*) ;

simple−declaration:
  $pointer_{opt}$ *identifier* $array\text{−}spec_{opt}$
  $pointer_{opt}$ *identifier* $array\text{−}spec_{opt}$ = *initializer−list*

pointer:
  *
  pointer *

array−spec:
  [*constant−expression*]
  [*constant−expression*] *array−spec*

4.2. Array Specifiers

The dimension of an array must be non-zero positive constant. Arrays have a lower bound of 0 and an upper bound of `n−1`, where `n` is the value of the constant expression.

4.3. Type Specifiers

type−specifier:
  $scope_{opt}$ *type*

type:
  `byte`
  `int`
  `uint`
  `sint`
  `usint`
  `lint`
  `ulint`
  `void`
  `float`
  typename
  polyname
  tupletype
  channel−specifier

scope:
    `intern`
    `extern`

channel–specifier:
    `chan` (*typelist*) *buffer–spec*$_{opt}$

tupletype:
    `tuple`$_{opt}$ (*typelist*)

buffer–spec:
    [*constant–expression*]

typelist:
    ptr–type
    ptr–type , *typelist*

ptr–type:
    type–specifier
    ptr–type pointer$_{opt}$

polyname:
    identifier

The keywords `intern` and `extern` control the scope of declarations. When applied to a definition or declaration at the file scope, `intern` narrows the scope to the current file; `extern` makes the declared identifier visible to other compilation units. By default declarations at the file scope default to `extern`. The control of access to members of abstract data types is defined in the discussion of ADT's below.

Typename is an identifier defined by a complex type declaration or a `typedef` statement.

4.3.1. Channel Type Specification

The type specified by a `chan` declaration is actually a pointer to an internal object with an anonymous type specifier. Because of their anonymity, objects of this special type cannot be defined in declarations; instead they must be created by an `alloc` statement referring to a `chan`. A channel declaration without a buffer specification produces a synchronous communication channel. Threads sending values on the channel will block until some other thread receives from the channel. The two threads rendezvous and a value is passed between sender and receiver. If buffers are specified then an asynchronous channel is produced. The *constant–expression* defines the number of buffers to be allocated. A send operation will complete immediately while buffers are available, and will block if all buffers are in use. A receive operation will block if no value is buffered. If a value is buffered, the receive will complete and make the buffer available for a new send operation. Any senders waiting for buffers will then be allowed to continue.

Values of *chan–type* are passed between threads using the channel for communication. If *chan–type* is a comma-separated list of types the channel supplies a *variant* protocol. A variant protocol allows messages to be demultiplexed by type during a receive operation. A form of the `alt` statement allows the control flow to be modified based on the type of a value received from a channel supplying a variant protocol.

4.3.2. Polymorphic Type

The polymorphic type can be used to dynamically represent a value of any type. A polymorphic type is identified by a lexical name defined in a polymorphic type definition (see the section on Type Definition) or as a parameter to a polymorphic abstract data type (see the section on Polymorphic and Parameterized Abstract Data Types). Distinct lexical names represent a value of the same structure but are different for the purposes of type checking. A polymorphic value is represented by a *fat pointer*. The pointer consists of an *integer* tag and a *pointer* to a value. Like channels, storage for the data must be allocated by the runtime.

4.4. Initializers

Only simple declarations at the file scope may be initialized.

initializer–list:
  constant–expression
  [*constant–expression*] *constant–expression*
  { *initializer–list* }
  initializer–list , *initializer–list*

An initialization consists of a *constant–expression* or a list of constant–expressions separated by commas and enclosed by braces. An array or complex type requires an explicit set of braces for each level of nesting. Unions may not be initialized. All the components of a variable need not be explicitly initialized; uninitialized elements are set to zero. ADT types are initialized in the same way as aggregates with the exception of ADT function members which are ignored for the purposes of initialization. Elements of sparse arrays can be initialized by supplying a bracketed index for an element. Successive elements without the index notation continue to initialize the array in sequence. For example:

```
byte a[256] = {
                ['a']   'A',    /* Set element 97 to 65 */
                ['a'+1] 'B',    /* Set element 98 to 66 */
                        'C'     /* Set element 99 to 67 */
};
```

If the dimensions of the array are omitted from the *array–spec* the compiler sets the size of each dimension to be large enough to accommodate the initialization. The size of the array in bytes can be found using `sizeof`.

4.5. Type Declarations

A type declaration creates a new type and introduces an identifier representing that type into the language.

type–declaration:
  complex { *memberlist* } ;
  enumeration–type
  tupletype

complex:
  `adt` *typename* *poly–spec*$_{opt}$
  `aggr` *typename*
  `union` *typename*

poly–spec:
  [*typelist*]

A complex type is composed of a list of members. Each member may be a complex type, a derived type or a basic type. Members are referenced by tag or by type. Members without tags are called *unnamed.* Arithmetic types, channel types, tuples, and complex types may be unnamed. Derived types may not be left unnamed. Complex unnamed members are referenced by type or by implicit promotion during assignment or when supplied as function arguments. Other unnamed members allocate storage but may not be referenced. Complex types are compared by structural rather than name equivalence. A type declaration must have either a type name or a tag.

> *memberlist:*
>   *member*
>   *memberlist member*
>
> *member:*
>   *type* ;
>   *tname pointer*$_{opt}$ *decl–tag array–spec*$_{opt}$;
>   *tname decl–tag* (*arglist*) ;
>
> *decl–tag:*
>   *identifier*

tname is one of the basic types or a new type introduced by `aggr`, `adt`, `union`, or `typedef`.

4.6. Tuples

A `tuple` is a collection of types forming a single object which can be used in the place of an unnamed complex type. The individual members of a tuple can only be accessed by assignment.

> *tuple:*
>   (*tlist*)
>
> *tlist:*
>   *tlist* , *expression*

When the declaration of a tuple would be ambiguous because of the parenthesis (for instance in the declaration of an automatic variable) use the keyword `tuple`:

```
void
f()
{
        int a;
        tuple (int, byte, Rectangle) b;
        int c;
}
```

Type checking of tuple expressions is performed by matching the *shape* of each of the component types. Tuples may only be addressed by assignment into other complex types or l–valued tuple expressions. A bracketed list of expressions forms a tuple constructor, while a list of l–valued expressions on the left hand side forms a destructor. For example, to make a function return multiple values:

```
(int, byte*, byte)
func()
{
        return (10, "hello", 'c');
}

void
main()
{
        int     a;
        byte*   str;
        byte    c;
        (a, str, c) = func();
}
```

When a tuple appears as the left-hand side of an assignment, type checking proceeds as if each individual member of the tuple were an assignment statement to the corresponding member of the complex type on the right-hand side. If a tuple appears on the right hand side of an assignment where the left-hand side yields a complex type then the types of each individual member of the tuple must match the corresponding types of the complex type exactly. If a tuple is cast into a complex type then each member of the tuple will be converted into the type of the corresponding member of the complex type under the rules of assignment.

```
aggr X
{
        int     a;
        byte    b;
};

void
main()
{
        X x;
        byte c;

        x = (10, c);       /* Members match exactly */
        x = (X)(10, 1.5); /* float is converted to byte */
}
```

4.7. Abstract Data Types

An abstract data type (ADT) defines both storage for members, as in an aggregate, and the operations that can be performed on that type. Access to the members of an abstract data type is restricted to enable information hiding. The scope of the members of an abstract data type depends on their type. By default access to members that define data is limited to the member functions. Members can be explicitly exported from the type using the `extern` storage class in the member declaration. Member functions are visible by default, the opposite behavior of data members. Access to a member function may be restricted to other member functions by qualifying the declaration with the `intern` storage class. The four combinations are:

```
adt Point
{
                int     x; /* Access only by member fns */
        extern  int     y; /* by everybody */

                Point set(Point*); /* by everybody */
        intern  Point tst(Point);  /* only by member fns */
};
```

Member functions are defined by type and name. The pair forms a unique name for the function, so the same member function name can be used in many types. Using the last example, the member function `set` could be defined as:

```
Point
Point.set(Point *a)
{
        a->x = 0;       /* Set Point value to zero */
        a->y = 0;

        return *a;
}
```

An implicit argument of either a pointer to the ADT or the value of the ADT may be passed to a member function. If the first argument of the member function declaration in the ADT specification is * *typename* (with the * preceding the name), then a *pointer* to the ADT is automatically passed as the first parameter, similarly to the `self` construct in Smalltalk. If the declaration is of the form **.** *typename* then the *value* of the ADT will be passed to the member function.

```
adt Point
{
                int     x;
        extern  int     y;

                /* Pass &Point as 1st arg */
                Point set(*Point);
                /* Pass Point as 1st arg */
                Point clr(.Point);
        intern  Point tst(Point);
};

void
func()
{
        Point p;

        p.set();        /* Set receives &p as 1st arg */
}
```

The receiving function is defined as:

```
Point
Point.set(Point *p)
{
        ...
}
```

4.8. Polymorphic and Parameterized Abstract Data Types

Alef allows the construction of type parameterized abstract data types, similar to *generic* abstract data types in Ada and Eiffel. An ADT is parameterized by supplying type parameter names in the declaration. The type parameters may be used to specify the types of members of the ADT. The argument type names have the same effect as a `typedef` to the polymorphic type. The scope of the types supplied as arguments is the same as the ADT *typename* and can therefore be used as a type specifier in simple declarations. For example the definition of a stack type of parameter type `T` may be defined as:

```
adt Stack[T]
{
        int     tos;
        T       data[100];
        void    push(*Stack, T);
        T       pop(*Stack);
};
```

Member functions of `Stack` are written in terms of the parameter type `T`. The implementation of `push` might be:

```
void
Stack.push(Stack *s, T v)
{
        s->data[s->tos++] = v;
}
```

The `Stack` type can be instantiated in two forms. In the *bound* form, a type is specified for `T`. The program is type checked as if the supplied type were substituted for `T` in the ADT declaration. For example:

```
Stack[int] stack;
```

declares a stack where each element is an `int`. In the bound form a type must be supplied for each parameter type. In the *unbound* form no parameter types are specified. This allows values of any type to be stored in the stack. For example:

```
Stack poly;
```

declares a stack where each element has polymorphic type.

4.9. Enumeration Types

enumeration–type:
  enum *typename* { *enum–list* } ;

enum–list:
  identifier
  identifier = *constant–expression*
  enum–list , *enum–list*

Enumerations are types whose value is limited to a set of integer constants. These constants, the members of the enumeration, are called enumerators. Enumeration variables are equivalent to integer variables. Enumerators may appear wherever an integer constant is legal. If the values of the enumerators are not defined explicitly, the compiler assigns incrementing values starting from 0. If a value is given to an enumerator, values are assigned to the following enumerators by incrementing the value for each successive member until the next assigned value is reached.

4.10. Type Definition

Type definition allows derived types to be named, basic types to be renamed, polymorphic types to be named, and forward referencing between complex types.

> *type−definition:*
> `typedef` *tname identifier* ;
> `typedef` *polyname* ;

If *tname* is omitted then the identifier, *polyname*, becomes a polymorphic type specifier. To declare complex types with mutually dependent pointers, it is necessary to use a `typedef` to predefine one of the types. Alef does not permit mutually dependent complex types, only references between them. For example:

```
typedef aggr A;

aggr B
{
        A       *aptr;
        B       *bptr;
};

aggr A
{
        A       *aptr;
        B       *bptr;
};
```

4.11. Function Declarations

There are three forms of function declaration: function definition, prototype declaration, and function pointer declaration.

> *function−declaration:*
> *type−specifier identifier* (*arglist*) *block*
>
> *function−id:*
> *pointer*$_{opt}$ *identifier array−spec*$_{opt}$
> *adt−function*
>
> *adt−function:*
> *typename* . *decl−tag*
>
> *arglist:*
> *arg*
> *pointer type*
> *arglist* , *arg*
>
> *arg:*
> *type*
> *type pointer*
> *type* (*pointer*) (*arglist*)
> *type simple−declaration*
> . . .

If a formal parameter is declared without an identifier, no variable corresponding to the actual parameter is produced.

5. Expressions

The order of expression evaluation is not defined except where noted. That is, unless the definition of the operator guarantees evaluation order, an operator may evaluate any of its operands first.

The behavior of exceptional conditions such as divide by zero, arithmetic overflow, and floating point exceptions is not defined.

5.1. Pointer Generation

References to expressions of type ‘function returning T’ and ‘array of T’ are rewritten to produce pointers to either the function or the first element of the array. That is ‘function returning T’ becomes ‘pointer to function returning T’ and ‘array of T’ becomes ‘pointer to the first element of array of T’.

5.2. Primary Expressions

Primary expressions are identifiers, constants, or parenthesized expressions:

> *primary−expression:*
>   *identifier*
>   *constant*
>   `...`
>   `nil`
>   (*expression*)
>   *tuple*

The parameters received by a function taking variable arguments are referenced using the ellipsis `...`. The primary−expression `...` yields a value of type ‘pointer to void’. The value points at the first location after the formal parameters. The primary−expression `nil` returns a pointer of type ‘pointer to void’ of value 0 which is guaranteed not to point at an object. `nil` may also be used to initialize channels and polymorphic types to a known value. The only legal operation on these types after such an assignment is a test with one of the equality test operators and the `nil` value.

5.3. Postfix Expressions

> *postfix−expression:*
>   *primary−expression*
>   *postfix−expression* [*expression*]
>   *postfix−expression* (*argument−list*)
>   . *typename* . *tag* (*argument−list*)
>   *postfix−expression* . *tag*
>   *postfix−expression* −> *tag*
>   *postfix−expression* ++
>   *postfix−expression* −−
>   *postfix−expression* ?
>
> *tag:*
>   *typename*
>   *identifier*
>
> *argument−list:*
>   *expression*
>   *argument−list* , *expression*

5.3.1. Array Reference

A primary expression followed by an expression enclosed in square brackets is an array indexing operation. The expression is rewritten to be `*((`*postfix–expression*`)+(`*expression*`))`. One of the expressions must be of type pointer, the other of integral type.

5.3.2. Function Calls

The *postfix–expression* must yield a value of type 'pointer to function'. A type declaration for the function must be declared prior to a function call. The declaration can be either the definition of the function or a function prototype. The types of each argument in the prototype must match the corresponding expression type under the rules of promotion and conversion for assignment. In addition unnamed complex type members will be promoted automatically. For example:

```
aggr Test
{
        int     t;
        Lock;           /* Unnamed substructure */
};

Test yuk;               /* Definition of complex object yuk */
void lock(Lock*);       /* Prototype for function lock */

void
main()
{
        lock(&yuk);     /* address of yuk.Lock is passed */
}
```

Calls to member functions may use the type name instead of an expression to identify the ADT. If the function has an implicit first parameter, `nil` is passed. Given the following definition of `X` these two calls are equivalent:

```
adt X
{
        int     i;
        void    f(*X);
};

X val;

        ((X*)nil)->f();
        .X.f();
```

This form is illegal if the implicit parameter is declared by value rather than by reference.

Calls to member functions of polymorphic ADT's have special promotion rules for function arguments. If a polymorphic type `P` has been bound to an actual type `T` then an actual parameter `v` of type `T` corresponding to a formal parameter of type `P` will be promoted into type `P` automatically. The promotion is equivalent to `(alloc P)v` as described in the Casts section. For example:

```
adt X[T]
{
        void    f(*X, T);
};

X[int] bound;

bound.f(3);                 /* 3 is promoted as if (alloc T)3 */
bound.f((alloc T)3);        /* illegal: int not same as poly */
```

In the unbound case values must be explicitly converted into the polymorphic type using the cast syntax:

```
X unbound;

unbound.f((alloc T)3);  /* 3 is converted into poly */
unbound.f(3);           /* illegal: int not same as poly */
```

In either case the actual parameter must have the same type as the formal parameter after any binding has taken place.

5.3.3. Complex Type References

The operator `.` references a member of a complex type. The first part of the expression must yield `union`, `aggr`, or `adt`. Named members must be specified by name, unnamed members by type. Only one unnamed member of type *typename* is permitted in the complex type when referencing members by type, otherwise the reference would be ambiguous. If the reference is by *typename* and no members of *typename* exist in the complex, unnamed substructures will be searched breadth first. The operation −> uses a pointer to reference a complex type member. The −> operator follows the same search and type rules as `.` and is equivalent to the expression *(*postfix−expression).tag*.

References to polymorphic members of unbound polymorphic ADT's behave as normal members: they yield an unbound polymorphic type. Bound polymorphic ADT's have special rules. Consider a polymorphic type `P` that is bound to an actual type `T`. If a reference to a member or function return value of type `P` is assigned to a variable `v` of type `T` using the assignment operator =, then the type of `P` will be narrowed to `T`, assigned to `v`, and the storage used by the polymorphic value will be unallocated. The value assignment operator := performs the same type narrowing but does not unallocate the storage used by the polymorphic value. For example:

```
adt Stack[T]
{
        int     tos;
        T       data[100];
};

Stack[int] s;
int i, j, k;

i := s.data[s−>tos];
j = s.data[s−>tos];
k = s.data[s−>tos];     /* illegal */
```

The first assignment copies the value at the top of the stack into `i` without altering the data structure. The second assignment moves the value into `j` and unallocates the storage used in the stack data structure. The third assignment is illegal since `data[s−>tos]` has been unallocated.

5.3.4. Postfix Increment and Decrement

The postfix increment (++) and decrement (−−) operators return the value of *expression*, then increment it or decrement it by 1. The expression must be an l−value of integral or pointer type.

5.4. Unary Operators

The unary operators are:

unary−expression:
    postfix−expression
    <− *unary−expression*
    ++ *unary−expression*
    −− *unary−expression*
    unary−operator cast−expression
    `sizeof` *cast−expression*
    `zerox` *unary−expression*

unary−operator: one of
    ? * !
    + − ~

5.5. Prefix Increment and Decrement

The prefix increment (++) and prefix decrement (−−) operators add or subtract one to a *unary−expression* and return the new value. The *unary−expression* must be an l−value of integral or pointer type.

5.6. Receive and Can Receive

The prefix operator <− receives a value from a channel. The *unary−expression* must be of type ‘channel of T’. The type of the result will be T. A process or task will block until a value is available from the channel. The prefix operator ? returns 1 if a channel has a value available for receive, 0 otherwise.

5.7. Send and Can send

The postfix operator <−, on the left-hand side of an assignment (see the section called Assignment), sends a value to a channel, for example:

```
chan(int) c;

c <-= 1;          /* send 1 on channel c */
```

The postfix operator ? returns 1 if a thread can send on a channel without blocking, 0 otherwise.

The prefix or postfix blocking test operator ? is only reliable when used on a channel shared between tasks in a single process. A process may block after a successful ? because there may be a race between processes competing for the same channel.

5.8. Indirection

The unary operator * retrieves the value pointed to by its operand. The operand must be of type ‘pointer to T’. The result of the indirection is a value of type T.

5.9. Unary Plus and Minus

Unary plus is equivalent to `(0+(`*unary–expression*`))`. Unary minus is equivalent to `(0−(`*unary–expression*`))`. An integral operand undergoes integral promotion. The result has the type of the promoted operand.

5.10. Bitwise Negate

The operator ~ performs a bitwise negation of its operand, which must be of integral type.

5.11. Logical Negate

The operator `!` performs logical negation of its operand, which must of arithmetic or pointer type. If the operand is a pointer and its value is `nil` the result is integer 1, otherwise 0. If the operand is arithmetic and the value is 0 the result is 1, otherwise the result is 0.

5.12. Zerox

The `zerox` operator may only be applied to an expression of polymorphic type. The result of `zerox` is a new fat pointer, which points at a copy of the result of evaluating *unary–expression*. For example:

```
typedef Poly;
Poly a, b, c;

a = (alloc Poly)10;
b = a;
c = zerox a;
```

causes `a` and `b` to point to the same storage for the value 10 and `c` to point to distinct storage containing another copy of the value 10.

5.13. Sizeof Operator

The `sizeof` operator yields the size in bytes of its operand, which may be an expression or the parenthesized name of a type. The size is determined from the type of the operand, which is not itself evaluated. The result is a signed integer constant. If `sizeof` is applied to a string constant the result is the number of bytes required to store the string including its terminating NUL byte or zero rune.

5.14. Casts

A cast converts the result of an expression into a new type:

cast–expression:
  unary–expression
  (*type–cast*) *cast–expression*
  (`alloc` *polyname*) *cast–expression*

type–cast:
  type pointer
  function–prototype
  `tuple` *tuple*

A value of any type may be converted into a polymorphic type by adding the keyword `alloc` before the polymorphic type name. This has the effect of allocating storage for the value, assigning the value of *cast–expression* into the storage, and yielding a fat pointer as the result. For example, to create a polymorphic variable with integer value 10:

```
        typedef Poly;
        Poly p;

        p = (alloc Poly) 10;
```

The only other legal cast involving a polymorphic type converts one *polyname* into another.

5.15. Multiply, Divide and Modulus

The multiplicative operators are:

multiplicative−expression:
cast−expression
multiplicative−expression * *multiplicative−expression*
multiplicative−expression / *multiplicative−expression*
multiplicative−expression % *multiplicative−expression*

The operands of * and / must have arithmetic type. The operands of % must be of integral type. The operator / yields the quotient, % the remainder, and * the product of the operands. If b is non−zero then `a == (a/b) + a%b` should always be true.

5.16. Add and Subtract

The additive operators are:

additive−expression:
multiplicative−expression
additive−expression + *multiplicative−expression*
additive−expression − *multiplicative−expression*

The + operator computes the sum of its operands. Either one of the operands may be a pointer. If *P* is an expression yielding a pointer to type *T* then P+*n* is the same as *p*+(sizeof(T)**n*). The − operator computes the difference of its operands. The first operand may be of pointer or arithmetic type. The second operand must be of arithmetic type. If *P* is an expression yielding a pointer of type *T* then P−*n* is the same as *p*−(sizeof(T)**n*). Thus if *P* is a pointer to an element of an array, *P*+1 will point to the next object in the array and *P*−1 will point to the previous object in the array.

5.17. Shift Operators

The shift operators perform bitwise shifts:

shift−expression:
additive−expression
shift−expression << *additive−expression*
shift−expression >> *additive−expression*

If the first operand is unsigned, << performs a logical left shift by *additive−expression* bits. If the first operand is signed, << performs an arithmetic left shift by *additive−expression* bits. The *shift−expression* must be of integral type. The >> operator is a right shift and follows the same rules as left shift.

5.18. Relational Operators

The values of expressions can be compared as follows:

relational−expression:
shift−expression
relational−expression < shift−expression
relational−expression > shift−expression
relational−expression <= shift−expression
relational−expression >= shift−expression

The operators are < (less than), > (greater than), <= (less than or equal to) and >= (greater than or equal to). The operands must be of arithmetic or pointer type. The value of the expression is 1 if the relation is true, otherwise 0. The usual arithmetic conversions are performed. Pointers may only be compared to pointers of the same type or of type `void*`.

5.19. Equality operators

The equality operators are:

equality−expression:
relational−expression
relational−expression == equality−expression
relational−expression != equality−expression

The operators == (equal to) and != (not equal) follow the same rules as relational operators. The equality operations may be applied to expressions yielding channels and polymorphic types for comparison with the value `nil`. A pointer of value `nil` or type `void*` may be compared to any pointer.

5.20. Bitwise Logic Operators

AND−expression:
equality−expression
AND−expression & equality−expression

XOR−expression:
AND−expression
XOR−expression ^ AND−expression

OR−expression:
XOR−expression
OR−expression | XOR−expression

The operators perform bitwise logical operations and apply only to integral types. The operators are & (bitwise and), ^ (bitwise exclusive or) and | (bitwise inclusive or).

5.21. Logical Operators

logical−AND−expression:
OR−expression
logical−AND−expression && OR−expression

logical−OR−expression:
logical−AND−expression
logical−OR−expression || logical−AND−expression

The && operator returns 1 if both of its operands evaluate to non−zero, otherwise 0. The || operator returns 1 if either of its operand evaluates to non−zero, otherwise 0. Both operators are guaranteed to evaluate strictly left to right. Evaluation of the expression will cease as soon the final value is determined. The operands can be any mix of arithmetic and pointer types.

5.22. Constant expressions

A constant expression is an expression which can be fully evaluated by the compiler during translation rather than at runtime.

> *constant−expression:*
>   *logical−OR−expression*

Constant−expression appears as part of initialization, channel buffer specifications, and array dimensions. The following operators may not be part of a constant expression: function calls, assignment, send, receive, increment and decrement. Address computations using the & (address of) operator on static declarations is permitted.

5.23. Assignment

The assignment operators are:

> *assignment−expression:*
>   *logical−OR−expression*
>   *unary−expression* <−= *assignment−expression*
>   *unary−expression assignment−operator assignment−expression*
>   *unary−expression* = (*type−cast*) *tuple*
>
> *assignment−operator: one of*
>   = := += *= /= −= %= &= |= ^= >>= <<=

The left side of the expression must be an l−value. Compound assignment allows the members of a complex type to be assigned from a member list in a single statement. A compound assignment is formed by casting a tuple into the complex type. Each element of the tuple is evaluated in turn and assigned to its corresponding element in the complex types. The usual conversions are performed for each assignment.

```
/* Encoding of read message to send to file system */
aggr Readmsg
{
        int     fd;
        void    *data;
        int     len;
};

chan (Readmsg) filesys;

int
read(int fd, void *data, int len)
{
        /* Pack message parameters and send to file system */
        filesys <-= (Readmsg)(fd, data, len);
}
```

If the left side of an assignment is a tuple, selected members may be discarded by placing `nil` in the corresponding position in the tuple list. In the following example only the first and third integers returned from `func` are assigned.

```
(int, int, int) func();

void
main()
{
        int a, c;

        (a, nil, c) = func();
}
```

The <−= (assign send) operator sends the value of the right side into a channel. The *unary−expression* must be of type ‘channel of T’. If the left side of the expression is of type ‘channel of T’, the value transmitted down the channel is the same as if the expression were ‘object of type T = expression’.

5.23.1. Promotion

If the two sides of an assignment yield different complex types then assignment promotion is performed. The type of the right hand side is searched for an unnamed complex type under the same rules as the `.` operator. If a matching type is found it is assigned to the left side. This promotion is also performed for function arguments.

5.23.2. Polymorphic Assignment

There are two operators for assigning polymorphic values. The reference assignment operator = copies the fat pointer. For example:

```
typedef Poly;
Poly a, b;
int i;

a = (alloc Poly)i;
b = a;
```

causes a to be given a fat pointer to a copy of the variable `i` and b to have a distinct fat pointer pointing to the same copy of `i`. Polymorphic variables assigned with the = operator must be of the same polymorphic name.

The value assignment operator `:=` copies the value of one polymorphic variable to another. The variable and value must be of the same polymorphic name and must represent values of the same type; there is no implicit type promotion. In particular, the variable being assigned to must already be defined, as it must have both a type and storage. For example:

```
typedef Poly;
Poly a, b, c;
int i, j;

a = (alloc Poly)i;
b = (alloc Poly)j;
b := a;
c := a;                 /* illegal */
```

causes a to be given a fat pointer to a copy of the variable `i` and b to be given a fat pointer to a copy of the variable `j`. The value assignment `b:=a` copies the value of `i` from the storage referenced by the fat pointer of a to the storage referenced by b, with the result being that a and b point to distinct copies of the value of `i`; the reference to the value of `j` is lost. The assignment `c:=a` is illegal because c has no storage to hold the value; c is in effect an uninitialized pointer.

A polymorphic variable may be assigned the value `nil`. This assigns the value 0 to the pointer element of the fat pointer but leaves the type field unmodified.

5.24. Iterators

The iteration operator causes repeated execution of the statement that contains the iterating expression by constructing a loop surrounding that statement.

> *expression:*
> *assignment−expression*
> *assignment−expression* `::` *assignment−expression*

The operands of the iteration operator are the integral bounds of the loop. The iteration counter may be made explicit by assigning the value of the iteration expression to an integral variable; otherwise it is implicit. The two expressions are evaluated before iteration begins. The iteration is performed while the iteration counter is less than the value of the second expression (the same convention as array bounds). When the counter is explicit, its value is available throughout the statement. For example, here are two implementations of a string copy function:

```
void
copy(byte *to, byte *from)
{
        to[0::strlen(from)+1] = *from++;
}

void
copy(byte *to, byte *from)
{
        int i;

        to[i] = from[i=0::strlen(from)+1];
}
```

If iterators are nested, the order of iteration is undefined.

5.25. Binding and Precedence

The binding and precedence of the operators is in decreasing order:

binding	operator
l to r	`()` `[]` `->` `.`
r to l	`!` `~` `++` `--` `<-` `?` `+` `-` `*` `&` (*cast*) `sizeof` `zerox`
l to r	`*` `/` `%`
l to r	`+` `-`
l to r	`<<` `>>`
r to l	`::`
l to r	`<` `<=` `>` `>=`
l to r	`==` `!=`
l to r	`&`
l to r	`^`
l to r	`|`
l to r	`&&`
l to r	`||`
l to r	`<-=` `=` `:=` `+=` `-=` `*=` `/=` `%=` `=` `^=` `|=` `<<=` `>>=`

6. Statements

Statements are executed for effect, and do not yield values. Statements fall into several groups:

> *statement:*
>   *expression* ;
>   *label–statement* :
>   *block–statement*
>   *selection–statement* ;
>   *loop–statement*
>   *jump–statement*
>   *exception–statement*
>   *process–statement* ;
>   *allocation–statement* ;

6.1. Label Statements

A statement may be prefixed by an identifier. The identifier labels the statement and may be used as the destination of a `goto`. Label and exception identifiers have their own name space and do not conflict with other names. Labels have function scope.

6.2. Expression Statements

Most expressions statements are function calls or assignments. Expressions may be null. Null expressions are often useful as empty bodies to labels or iteration statements.

6.3. Block Statements

Several statements may be grouped together to form a block. The body of a function is a block.

> *block:*
>   { *autolist slist* }
>   !{ *autolist slist* }
>
> *autolist:*
>   *declaration*
>   *autolist declaration*
>
> *slist:*
>   *statement*
>   *slist statement*

An identifier declared in *autolist* suspends any previous declaration of the same identifier. An identifier may be declared only once per block. The declaration remains in force until the end of the block, after which any suspended declaration comes back into effect.

The value of identifiers declared in *autolist* is undefined at block entry and should be assigned to a known value after declaration but before use.

The symbol !{ introduces a guarded block. Only one thread may be executing the statements contained in the guarded block at any instant.

6.4. Selection Statements

Selection statements alter the flow of control based on the value of an expression.

> *selection–statement:*
> `if (` *expression* `)` *statement* `else` *statement*
> `if (` *expression* `)` *statement*
> `switch` *expression cbody*
> `typeof` *expression cbody*
> `alt` *cbody*
>
> *cbody:*
> `{` *caselist* `}`
> `!{` *caselist* `}`
>
> *caselist:*
> *case–item*
> *alt–item*
> *type–item*
> *caselist case–item*
>
> *case–item:*
> `case` *constant–expression* `:` *statement*
> `default :` *statement*
>
> *alt–item:*
> `case` *expression* `:` *statement*
>
> *type–item:*
> `case` *ptr–type* `:` *statement*

An `if` statement first evaluates *expression,* which must yield a value of arithmetic or pointer type. The value of *expression* is compared with 0. If it compares unequal *statement* is executed. If an `else` clause is supplied and the value compares equal the `else` statement will be executed. The `else` clause shows an ambiguity in the grammar. The ambiguity is resolved by matching an `else` with the nearest `if` without an `else` at the same block level.

The `switch` statement selects one of several statements based on the value of *expression*. The *expression* is evaluated and converted into an integer. The integer is compared with the value specified in each `case`. If the integers compare, control is transferred to the statement after the matching `case`. If no `case` is matched, the `default` clause is executed. If the `default` is omitted then none of the `case` statements is executed. The `case` expression must yield an integer constant. For a single `switch` statement each case expression must yield a unique value.

Within a `switch`, `alt`, or `typeof` execution proceeds normally except that a `break` statement will terminate the selection statement.

The `typeof` statement selects one of several statements based on the type of *expression*. The *expression* must be of polymorphic type. The expression is evaluated and the resulting type is compared with the type specified by each `case`. If the types match, the statement part of the corresponding `case` is executed. All the cases must have a distinct type within a single `typeof` statement. If no `case` is matched, the `default` clause is executed, if one exists; otherwise none of the `case` statements is executed. If the *expression* is a simple variable, then within the statement supplied by the `case`, the value is narrowed to the type of that `case`. In the `default` case where *expression* is a simple variable the type remains polymorphic.

```
typeof v {
case int:
        print("int=%d", v);   /* v passed as int */
        break;
case float:
        print("float=%f", v); /* v passed as float */
        break;
default:
        usertype(v);      /* still polymorphic */
        break;
}
```

The `typeof` statement is the only way to narrow the type of a polymorphic value.

The `alt` statement allows threads to perform communication on several channels simultaneously without polling. The expression in each `case` of an `alt` must contain either a send or receive operation. The `alt` statement provides a fair select between ready channels. A thread will remain blocked in `alt` until one of the `case` expressions can be evaluated without blocking. The `case` expression may be evaluated more than once, therefore care should be taken when using expressions which have side effects. If several of the `case` expressions are ready for evaluation one is chosen at random. A `break` statement terminates each case of the `alt`. If the `break` statement is omitted execution will proceed to execute the communication of the next case regardless of its readiness to communicate. For example:

```
chan(Mesg) keyboard, mouse;
Mesg m;

alt {
case m = <-keyboard:
        /* Process keyboard event */
        break;
case m = <-mouse:
        /* Process mouse event */
        break;
}
```

The `alt` statement is also used to discriminate between the type of values received from channels of variant protocols. In this form each *case-item* of the `alt` must be a simple assignment. The right hand side must contain a communication operation on a channel which supplies a variant protocol. The type of the l-value is used to match a type in the variant protocol. An `alt` may be performed on an arbitrary set of variant protocol channels so long as no type is supplied by more than one channel. There must be a `case` clause for each type supplied by the union of all channel types; Alef requires the match against types to be exhaustive. For example:

```
Aggr1 a;
Aggr2 b;
Aggr3 c;

chan (Aggr1, Aggr2, Aggr3) ch;

alt {
case a = <-ch:
        print("received Aggr1");
        break;
case b = <-ch:
        print("received Aggr2");
        break;
case c = <-ch:
        print("received Aggr3");
        break;
}
```

If an `alt` is pending on a channel the programmer must ensure that other threads do not perform an operation of the same type on the channel until the `alt` is complete. Otherwise the `alt` on that channel may block if values are removed by the other thread.

The symbol `!{` introduces a guarded *caselist*. Only one thread may be executing the statements contained in the guarded caselist at any instant.

6.5. Loop Statements

Several loop constructs are provided:

loop−statement:
 `while (` *expression* `)` *statement*
 `do` *statement* `while (` *expression* `) ;`
 `for (` *expression* `;` *expression* `;` *expression* `)` *statement*

In `while` and `do` loops the statement is repeated until the expression evaluates to 0. The expression must yield either an arithmetic or pointer type. In the `while` loop the expression is evaluated and tested before the statement. In the `do` loop the statement is executed before the expression is evaluated and tested.

In the `for` loop the first expression is evaluated once before loop entry. The expression is usually used to initialize the loop variable. The second expression is evaluated at the beginning of each loop iteration, including the first. The expression must yield either a pointer or arithmetic type. The statement is executed while the evaluation of the second expression does not compare to 0. The third expression is evaluated after the statement on each loop iteration. The first and third expressions have no type restrictions. All of the expressions are optional. If the second expression is omitted an expression returning a non−zero value is implied.

6.6. Jump Statements

Jump statements transfer control unconditionally.

jump–statement:
  `goto` *identifier* ;
  `continue` *count*$_{opt}$;
  `break` *count*$_{opt}$;
  `return` *expression*$_{opt}$;
  `become` *expression* ;

count:
  integer–constant

`goto` transfers control to the label *identifier*, which must be in the current function.

6.6.1. Continue Statements

The `continue` statement may only appear as part of a loop statement. If *count* is omitted the `continue` statement transfers control to the loop–continuation portion of the smallest enclosing iteration statement, that is, the end of that loop. If *count* is supplied `continue` transfers control to the loop continuation of some outer nested loop. *Count* specifies the number of loops to skip. The statement `continue` with no *count* is the same as `continue 1`. For example:

```
while(1) {
        while(1) {
                continue 2;     /* Same as goto contin; */
        }
contin:                         /* Continue comes here */
}
```

6.6.2. Break Statements

Define *compound* to be a selection or loop statement. The `break` statement may only appear as part of such a compound. If *count* is omitted, then the break terminates the statement portion of the compound and transfers control to the statement after the compound. If *count* is supplied, break causes termination of the some nested compound. *Count* is the number of nested compounds to terminate. A `break` with no *count* is the same as `break` 1. In a selection statement, `break` terminates execution of the case in the selection, and thus prevents 'falling through' to the next case.

6.6.3. Return Statement

A function returns to its caller using a `return` statement. An expression is required unless the function is declared as returning the type `void`. The result of *expression* is evaluated using the rules of assignment to the return type of the function.

6.6.4. Become Statement

The `become` statement transforms the execution of the current function into the calculation of the *expression* given as its argument. If *expression* is not itself a function call, then it must have the same type as the return value of the caller and the behavior is analogous to a `return` statement. If, however, it is a function call, then it need only have the same return type; the argument list may be different. When a function *P* executes a `become` whose expression is a call of *Q*, the effect is exactly as if the caller of *P* had instead called *Q* with the appropriate arguments from *P*. In particular, the stack frame for *P* is overwritten by the frame for *Q*; functions that invoke one another with `become` will execute in constant stack space.

6.7. Exception Statements

The `rescue`, `raise`, and `check` statements are provided for use in error recovery:

exception–statement:
  `raise` *identifier*$_{opt}$;
  `rescue` *identifier*$_{opt}$ *block*
  `check` *expression* ;
  `check` *expression* , *string–constant* ;

6.7.1. Raise and Rescue Statement

Under normal execution a `rescue` block is not executed. A `raise` after a `rescue` statement transfers control to the closest previously defined `rescue` statement in the same function. Execution flows through the end of the `rescue` block by default.

Execution has no effect on the connection between `raise` and `rescue` statements. If an identifier is supplied in a `raise` statement, control is transferred to the named `rescue` statement. For example, these two fragments are equivalent:

```
alloc p;                alloc p;
rescue {                goto notrescue;
        unalloc p;              dorescue:
        raise;                          unalloc p;
}                                       goto nextrescue;
                        notrescue:
if(error)               if(error)
        raise;                  goto dorescue;
```

Multiple `rescue` statements may be cascaded to perform complex error recovery actions:

```
alloc a, b;
rescue {
        unalloc a, b;
        return 0;
}

alloc c;
rescue {
        unalloc c;
        raise;
}

dostuff();

if(error)
        raise;
```

6.7.2. Check Statement

The `check` statement tests an assertion. If the assertion fails, the runtime calls an error handler. By default the handler writes a message to standard error and exits with the status `ALEFcheck`. A user–supplied error handler may be installed by setting a handler vector. The prototype for the vector is:

```
void    (*ALEFcheck)(byte*, byte*);
```

The first string argument, supplied by the compiler, is of the form `file:line:`

`func()` The second argument passed to the handler will be the *string−constant* or the default string `check` if no string is supplied by the statement. The expression is evaluated and compared to 0. If the compare succeeds the assertion has failed. For example, the runtime checks the return from memory allocation like this:

```
ptr = malloc(n);
check ptr != nil, "no memory";
```

6.8. Process Control Statements

An Alef program consists of one or more preemptively scheduled processes called *procs*, each of which consists of one or more coroutines called *tasks*. An Alef program is always executing within some task.

These statements create procs and tasks:

process−statement:
  `proc` *function−call* ;
  `task` *function−call* ;
  `par` *block*

The `proc` statement creates a new proc, which starts running the named function. The arguments to *function−call* are evaluated by the original proc. Since procs are scheduled preemptively, the interleaving of their execution is undefined. What resources a proc shares with its parent is defined by the implementation. A proc is initially created with a single task, which begins execution at the function call.

The `task` statement creates a coroutine within the same proc, which begins execution at the function call. The proc is maintained until all the tasks have completed. A task completes when it returns from its initial function or calls the library function `terminate`. All of the tasks within a proc share memory, including access to the stack of each task. Tasks are non-preemptive: they are scheduled during message passing and synchronization primitives only. In both the `proc` and `task` statements, the function call parameters are evaluated in the original task.

The synchronization primitives that can cause task switching are defined by a library. They are `QLock.lock` and `Rendez.sleep`. The communication operations which can cause task switching are `alt`, <−= (send) and <− (receive). A process that contains several tasks will exist until all the tasks within the process have exited. In turn, a program will exist until all of the processes in the program have exited. A process or task may exit explicitly by calling the function `exits` or by returning from the function in which it was invoked.

The `par` statement implements fork/process/join. A new process is created for each statement in the block. The `par` statement completes when all processes have completed execution of their statements. A `par` with a single statement is the same as a block. The processes within a `par` have the same memory sharing model as procs and share all automatics and parameters of the function executing the `par`. `Locks` or `QLocks` may be used to synchronize write access to the shared variables. The process that entered the `par` is guaranteed to be the same process that exits.

6.9. Allocation Statements

Memory management statements allocate and free memory for objects:

allocation−statement:
  `alloc` *alloclist* ;
  `unalloc` *alloclist* ;

alloclist:
  expression
  alloclist , *expression*

6.9.1. Alloc Statement

The `alloc` statement takes a list of pointers, which must also be l−values. In strictly left to right order, for each pointer, memory is reserved for an object of appropriate type and its address is assigned to the pointer. The memory is guaranteed to be filled with zeros. If the allocation fails because there is insufficient memory a `check` condition will be generated with the argument string `no memory`.

If the *expression* has `chan` type, the runtime system will also initialize the new channel. Buffers will be allocated for asynchronous channels. For example:

```
chan(int) a;
chan(int)[10] *p;

alloc a, p, *p;
```

To allocate a polymorphic value a cast expression is used as defined in the section on Casts.

6.9.2. Unalloc Statement

The `unalloc` statement releases memory. The argument to `unalloc` must have been returned by a successful `alloc` or be `nil`. Unalloc of `nil` has no effect. If an object is unallocated twice, or an invalid object is unallocated, the runtime system will generate a `check` condition with the message string `arena corrupted`.

6.10. Lock

The `Lock` ADT provides spin locks. Two operations are provided. `Lock.lock` claims the lock if free, otherwise it busy loops until the lock becomes free. `Lock.unlock` releases a lock after it has been claimed.

Lock ADTs have no runtime state and may be dynamically allocated. The thread which claimed the lock need not be the thread which unlocks it.

6.11. QLock

The `QLock` ADT provides blocking mutual exclusion. If the lock is free `QLock.lock` claims the lock. Further attempts to gain the lock will cause the thread to be suspended until the lock becomes free. The lock is released by calling `QLock.unlock`.

The thread which claimed the lock need not be the thread which unlocks it.

`QLock` ADTs have runtime state and may be dynamically allocated provided they are unallocated only when unlocked.

7. The Plan 9 Implementation

The runtime, support functions, and basic library for Alef are contained in a single library documented in section 2 of the Plan 9 Programmer’s Manual. The include file `<alef.h>` contains prototypes for the library. A `pragma lib` directive tells *vl*(1) to load the correct library elements automatically. The `pragma` directives supported by the compiler are documented by the compiler manual page *alef(1).* The compiler ignores unrecognized `pragma` directives.

In Plan 9 all procs in an Alef program share the same address space. When a task performs a system call, all tasks within its proc will be blocked until the system call completes.

Programs should not receive notes unless the programmer can guarantee the runtime system will not be interrupted.

A channel may be involved in no more than one `alt` at any time.

The compiler does not support `lint` and `ulint` even though they are defined.

Stack bounds checking is not implemented. The external variable `ALEFstack` defines the number of bytes allocated for the stack of each task. The `ALEFstack` variable can be changed any time, and is global to all task creation. The default stack size is 16000 bytes. The library function `doprint` has a 1024 byte automatic array so `ALEFstack` should not be less than 2048 bytes.

The runtime system uses the *rendezvous*(2) system call to synchronize procs within an Alef program. The rendezvous tag space is part of the name space group, so care must be taken in forking the name space with RFNAMEG. For example after an `rfork(RFNAMEG)` a task cannot synchronize or exchange messages with another proc. A good example of the code necessary to perform this operation can be found in the source to *acme*(1).

A related issue is that programs that do not fork their name space may collide with other programs sharing the space, so unless there is strong reason not to, an Alef program should call `rfork(RFNAMEG)` early.

The runtime uses the variable `ALEFrfflag` as the argument to *rfork*(2) when creating a proc. By default `ALEFrfflag` is set to RFNOWAIT|RFMEM|RFPROC. It may be modified, for example by clearing RFNOWAIT to permit a proc to *wait*(2) for a child proc to exit. The value of `ALEFrfflag` should be restored immediately after the `rfork`.

A unique identifier for each task may be found by calling the function `ALEF_tid`, declared

```
        uint    ALEF_tid();
```

in <`alef.h`>. The identifier is useful for debugging; for example, it is used in the *acid*(1) alef library.

In the symbol table of the executable, member functions of ADT types are named by concatenating the ADT name, an underscore, and the member name.

8. Yacc Style Grammar

The following grammar is suitable for implementing a yacc parser. Upper case words and punctuation surrounded by single quotes are the terminal symbols.

```
prog:           decllist

decllist        :
                | decllist decl

decl            : tname vardecllist ';'
                | tname vardecl '(' arglist ')' block
                | tname adtfunc '(' arglist ')' block
                | tname vardecl '(' arglist ')' ';'
                | typespec ';'
                | TYPEDEF ztname vardecl zargs ';'
                | TYPEDEF IDENTIFIER ';'

zargs           :
                | '(' arglist ')'

ztname          : tname
                | AGGR
                | ADT
                | UNION

adtfunc         : TYPENAME '.' name
                | indsp TYPENAME '.' name

typespec        : AGGR ztag '{' memberlist '}' ztag
                | UNION ztag '{' memberlist  '}' ztag
                | ADT ztag zpolytype '{' memberlist '}' ztag
                | ENUM ztag '{' setlist '}'

ztag            :
                | name
                | TYPENAME

zpolytype       :
                | '[' polytype ']'

polytype        : name
                | name ',' polytype

setlist         : sname
                | setlist ',' setlist

sname           :
                | name
                | name '=' expr

name            : IDENTIFIER

memberlist      : decl
                | memberlist decl

vardecllist     :
                | ivardecl
                | vardecllist ',' ivardecl

ivardecl        : vardecl zinit

zinit           :
                | '=' zelist
```

```
zelist          : zexpr
                | '[' expr ']' expr
                | '.' stag expr
                | '{' zelist '}'
                | '[' expr ']' '{' zelist '}'
                | zelist ',' zelist

vardecl         : IDENTIFIER arrayspec
                | indsp IDENTIFIER arrayspec
                | '(' indsp IDENTIFIER arrayspec ')'
                              '(' arglist ')'
                | indsp '(' indsp IDENTIFIER arrayspec ')'
                              '(' arglist ')'

arrayspec       :
                | arrayspec '[' zexpr ']'

indsp           : '*'
                | indsp '*'

arglist         :
                | arg
                | '*' xtname
                | '.' xtname
                | arglist ',' arg

arg             : xtname
                | xtname indsp arrayspec
                | xtname '(' indsp ')' '(' arglist ')'
                | xtname indsp '(' indsp ')' '(' arglist ')'
                | TUPLE tuplearg
                | xtname vardecl
                | '.' '.' '.'

tuplearg        : tname
                | tname '(' indsp ')' '(' arglist ')'
                | tname vardecl

autolist        :
                | autolist autodecl

autodecl        : xtname vardecllist ';'
                | TUPLE tname vardecllist ';'

block           : '{' autolist slist '}'
                | '!' '{' autolist slist '}'

slist           :
                | slist stmnt

tbody           : '{' ctlist '}'
                | '!' '{' clist '}'

ctlist          :
                | ctlist tcase

tcase           : CASE typecast ':' slist
                | DEFAULT ':' slist
```

```
cbody           : '{' clist '}'
                | '!' '{' clist '}'

clist           :
                | clist case

case            : CASE expr ':' slist
                | DEFAULT ':' slist

rbody           : stmnt
                | IDENTIFIER block

zlab            :
                | IDENTIFIER

stmnt           : nlstmnt
                | IDENTIFIER ':' stmnt

info            :
                | ',' STRING_CONST

nlstmnt         : zexpr ';'
                | block
                | CHECK expr info ';'
                | ALLOC elist ';'
                | UNALLOC elist ';'
                | RESCUE rbody
                | RAISE zlab ';'
                | GOTO IDENTIFIER ';'
                | PROC elist ';'
                | TASK elist ';'
                | BECOME expr ';'
                | ALT cbody
                | RETURN zexpr ';'
                | FOR '(' zexpr ';' zexpr ';' zexpr ')' stmnt
                | WHILE '(' expr ')' stmnt
                | DO stmnt WHILE '(' expr ')'
                | IF '(' expr ')' stmnt
                | IF '(' expr ')' stmnt ELSE stmnt
                | PAR block
                | SWITCH expr cbody
                | TYPEOF expr tbody
                | CONTINUE zconst ';'
                | BREAK zconst ';'

zconst          :
                | CONSTANT

zexpr           :
                | expr

expr            : castexpr
                | expr '*' expr
                | expr '/' expr
                | expr '%' expr
                | expr '+' expr
                | expr '-' expr
                | expr '>>' expr
                | expr '<<' expr
                | expr '<' expr
                | expr '>' expr
                | expr '<=' expr
                | expr '>=' expr
                | expr '==' expr
                | expr '!=' expr
                | expr '&' expr
                | expr '^' expr
                | expr '|' expr
                | expr '&&' expr
                | expr '||' expr
                | expr '=' expr
                | expr ':=' expr
                | expr '<-' '=' expr
                | expr '+=' expr
                | expr '-=' expr
                | expr '*=' expr
                | expr '/=' expr
                | expr '%=' expr
                | expr '>>=' expr
                | expr '<<=' expr
                | expr '&=' expr
                | expr '|='  expr
                | expr '^=' expr
                | expr '::' expr

castexpr        : monexpr
                | '(' typecast ')' castexpr
                | '(' ALLOC typecast ')' castexpr

typecast        : xtname
                | xtname indsp
                | xtname '(' indsp ')' '(' arglist ')'
                | TUPLE tname

monexpr         : term
                | '*' castexpr
                | '&' castexpr
                | '+' castexpr
                | '-' castexpr
                | '--' castexpr
                | ZEROX castexpr
                | '++' castexpr
                | '!' castexpr
                | '~' castexpr
                | SIZEOF monexpr
                | '<-' castexpr
                | '?' castexpr

ztelist         :
                | telist

tcomp           : expr
                | '{' ztelist '}'

telist          : tcomp
                | telist ',' tcomp

term            : '(' telist ')'
```

```
                | SIZEOF '(' typecast ')'
                | term '(' zarlist ')'
                | term '[' expr ']'
                | term '.' stag
                | '.' TYPENAME '.' stag
                | term '->' stag
                | term '--'
                | term '++'
                | term '?'
                | name
                | '.' '.' '.'
                | ARITHMETIC_CONST
                | NIL
                | ENUM_MEMBER
                | STRING_CONST
                | '$' STRING_CONST

stag            : IDENTIFIER
                | TYPENAME

zarlist         :
                | elist

elist           : expr
                | elist ',' expr

tlist           : typecast
                | typecast ',' tlist

tname           : sclass xtname
                | sclass TUPLE '(' tlist ')'
                | sclass '(' tlist ')'

variant         : typecast
                | typecast ',' variant

xtname          : INT
                | UINT
                | SINT
                | SUINT
                | BYTE
                | FLOAT
                | VOID
                | TYPENAME
                | TYPENAME '[' variant ']'
                | CHAN '(' variant ')' bufdim

bufdim          :
                | '[' expr ']'

sclass          :
                | EXTERN
                | INTERN
                | PRIVATE
```