ALEF Language Reference Manual

Phil Winterbottom
philw@research.att.com

Introduction

Alef is a concurrent programming language designed for systems programming. Exception handling, process management and synchronisation 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. 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 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. A token is a sequence of characters separated by white space. The white space characters are space, new line, tab, form feed and vertical tab.

1.1. Tokens

The lexical analyser classifies tokens as: identifiers, typenames, keywords, constants and operators. Tokens are separated by white space. White space is ignored in the source except as needed to separate sequences of tokens which would otherwise be ambiguous. The lexical analyser is greedy: if tokens have been consumed up to a given character, then the next token will be the longest string of characters which 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         uint
ulint           unalloc         union
usint           void            while
```

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 is any sequence of alpha-numeric characters and the character _. Identifiers may not start with a digit. Identifiers are case sensitive. All characters are significant. Identifier names prefixed by `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 may contain one or more characters surrounded by the single quote mark `'`. 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     Buzz
v       VT      Vertical tab
\       \       Backslash
"       "       Double quote
```

Character constants have the type `int`. If the character is a unicode rune (see Rune(2) in the Plan9 manual) the value of the integer will be the 16-bit unicode representation of the rune.

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 the double quote marks `"`. A string has the type 'static array of byte'. The NUL 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.

1.6. Programs

An ALEF program is a list of declarations. The declarations introduce identifiers. Identifiers may define variables, types, functions, function prototypes or enumerators. Identifiers have storage classes that define their scope. A storage class applied to a complex type controls the scope of the members. 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 synchronisation operations. The term *thread* is used wherever the distinction between a process and 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: file, type, function and local.

- 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 the dereference operators `.` and `->` are applied to 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 automatics is undefined upon creation. Automatic variables are destroyed at block exit. Parameters are created by function invocation and are destroyed at function exit. Uninitialised static objects exist from invocation of the program until termination. Static objects which have not been initialised have the value 0.

3. Types

A small set of basic types are 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 character
`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

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 initialised before use. They are the size of a pointer. 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 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 */
char    c[10];          /* A vector of 10 characters */
float   *pow();         /* A function returning a pointer to a float */
```

Complex types may be built from the basic types and the deriving operators. Complex types may be either aggregates, unions 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 which 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. Derived types and basic types may be renamed using the `typedef` statement.

The integral types are `int`, `uint`, `sint`, `suint`, `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.

3.3. Conversions and Promotions

ALEF performs the same implicit conversions and promotions as C with the exception 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 lefthand 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
    type-definition
    function-declaration

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 initialised.

simple-declarations:
    type-specifier simple-decl-list **;**

simple-decl-list:
    simple-declaration
    simple-decl-list **,** *simple-declaration*

simple-declaration:
    pointer$_{opt}$ *identifier array-spec*$_{opt}$
    pointer$_{opt}$ *identifier array-spec*$_{opt}$ = *initialiser-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 bound specified by the constant expression.

4.3. Type Specifiers

type-specifier:
    storage-class$_{opt}$ *type*

type:
    `byte`
    `int`
    `uint`
    `sint`
    `usint`
    `lint`
    `ulint`
    `void`
    `float`
    typename
    channel-specifier

storage-class:
    `intern`
    `extern`

channel-specifier:
    `chan (` *chan-type* `)` *buffer-spec*$_{opt}$

buffer-spec:
    `[` *constant-expression* `]`

The storage class controls the scope of the declaration. Storage classes may only be applied to declarations at the file scope. The scope of a definition qualified with `intern` is file. A declaration qualified by `extern` references a definition in this or another file.

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

4.3.1. Channel Type Specification

chan-type:
    basic
    basic `,` *chan-type*

basic:
    type pointer$_{opt}$

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. A thread 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 deallocate the buffer. 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.

4.4. Initialisers

Only simple declarations at the file scope may be initialised.

initialiser-list:
constant-expression
`[` *constant-expression* `]` *constant-expression*
`{` *initialiser-list* `}`
initialiser-list **,** *initialiser-list*

An initialisation 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. All the components of a variable need not be explicitly initialised; uninitialised elements are set to zero. ADT types are initialised in the same way as aggregates with exception of ADT function members which are ignored for the purposes of initialisation. Elements of sparse arrays can be initialised by supplying a bracketed index for an element. Successive elements without the index notation continue to initialise the array in sequence. For example:

```
char 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 initialisation. The size of the array 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 typename `{` *memberlist* `}` **;**
complex `{` *memberlist* `}` *decl-tag* **;**
complex typename `{` *memberlist* `}` *decl-tag* **;**
enumeration-type
tupletype

complex:
`adt`
`aggr`
`union`

decl-tag:
identifier

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

typelist:
typelist **,** *type-declaration*

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. Unnamed members are referenced by type or by implicit promotion during assignment or when supplied as function arguments. A type declaration must have either a type name or a tag.

memberlist:
    member
    memberlist member

member:
    tname pointer$_{opt}$ *decl-tag array-spec*$_{opt}$ `;`
    tname decl-tag `(` *arglist* `) ;`

tname is one of the basic types or a new type introduced by `aggr`, `adt`, `union`, or `typedef`. A tuple type is a complex type whose members are all unnamed. Tuples may only be addressed by assignment into other complex types or an l-valued tuple expression. For example:

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

void
main()
{
        int     a;
        byte    *str;

        (a, str) = func();
}
```

This example demonstrates multiple return values. The return type of `func` is a complex type consisting of three integers. The tuple type returned by `func` will match any other complex type consisting of three arithmetic types. So the following is legal:

```
aggr T
{
        int a;
        byte b;
        float c;
};

void
main()
{
        (a, b, c) = func();
}
```

4.6. Abstract Data Types

An abstract data type (ADT) defines both storage for members, like an aggregate, and the operations that can be performed on that type. Access to the members of an abstract data type is restricted to enforce a policy of information hiding. The mechanism is designed to encourage modular program design and provide clean library interfaces. 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, that is the opposite behaviour 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 by member functions only */
        extern  int     y;              /* Access by everybody */

                Point set(Point*);      /* Access by everybody */
        intern  Point tst(Point);       /* Access only from Point.set */
};
```

Member functions are defined by type and name. The pair form 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 the value of the point to zero */
        a->y = 0;

        return *a;
}
```

An implicit pointer to the abstract data type may be passed to a member function. If the first argument of the member function declaration in the ADT specification is '* typename' (Note the * precedes the name) the first parameter is passed implicitly.

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

                Point set(*Point);      /* Pass &Point as first argument */
        intern  Point tst(Point);
};

void
func()
{
        Point p;

        p.set();        /* Set receives &p as first argument */
}
```

The implicit parameter passing mechanism is particularly useful when the ADT is an unnamed substructure. The receiving function is defined as:

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

4.7. 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. The members of an enumeration are called enumerators. Enumeration variables are equivalent to integer variables. Enumeration constants may appear wherever an integer constant is legal. If the values of the enumerators are undefined the compiler assigns incrementing values from 0. If a value is given to an enumeration constant, values are assigned to the following enumerators by incrementing the value for each successive member until the next assigned value is reached.

4.8. Type Definition

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

type-definition:
`typedef` *tname identifier;*

A `typedef` is required to declare complex types with mututally dependent pointers. 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;
}
```

4.9. Function Declarations

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

function-declaration:
type-specifier identifier `(` *arglist* `)` *block*
type-specifier function-id `(` *arglist* `)` `;`
type-specifier `(` *function-id* `)` `(` *arglist* `)` `;`

function-id:
$pointer_{opt}$ *identifier* $array\text{-}spec_{opt}$
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 declaration for the corresponding variable is produced in the function body.

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 behaviour 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 T’.

5.2. Primary Expressions

Primary expressions are identifiers, constants or parenthesised expressions:

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

The parameters received by a function taking variable arguments are referenced using `'...'`. 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.

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 automatics) 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. A `tuple` will match another `tuple` or complex type if each individual member of the tuple could legally be assigned to its corresponding member under the rules of assignment.

6.1. Postfix Expressions

postfix-expression:
primary-expression
postfix-expression `[` *expression* `]`
postfix-expression `(` *argument-list* `)`
postfix-expression `.` *tag*
postfix-expression `->` *tag*
postfix-expression `++`
postfix-expression `--`
postfix-expression `?`

tag:
typename
identifier

argument-list:
expression
argument-list `,` *expression*

6.1.1. 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.

6.1.2. 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 specification 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 Test */
void lock(Lock*);       /* Prototype for function lock */

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

6.1.3. Complex Type References

The operator . references a member of a complex type. The first part of the expression must be a `union`, `aggr` or `adt`. The member may be specified by either name or type. Only one unnamed member of type *typename* is permitted in the complex type when referencing members by type, otherwise the reference is 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 *(*postfix-expression).tag*.

6.1.4. Postfix Increment and Decrement

The postfix increment (++) and decrement (−−) operators return the value of expression, then add or subtract 1 to the expression. The expression must be an l-value of integral type.

6.2. Unary Operators

The unary operators are:

unary-expression:
postfix-expression
++ *unary-expression*
−− *unary-operator cast-expression*
`sizeof` *cast-expression*

unary-operator: one of
`<-  ?  *`
`+ - ~  !`

6.3. 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.

6.4. Receive and Can Receive

The 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. *unary-expression* must be of type ‘channel of T’.

6.5. Can send

The postfix operator `?` returns 1 if a thread can send on a channel without blocking, 0 otherwise. *unary-expression* must be of type 'channel of T'.

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 the interleaving of the processes using the channel is undefined.

6.6. Indirection

The unary operator `*` retrieves the 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.

6.7. 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 is the type of the promoted operand.

6.8. Bitwise Negate

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

6.9. 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 0, otherwise 1. If the operand is arithmetic and the value is 0 the result is 0, otherwise the result is 1.

6.10. 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 an 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.

6.11. Casts

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

cast-expression:
    unary-expression
    `(` *type-cast* `)` *cast-expression*

cast-type:
    type pointer

6.12. 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.

6.13. 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.

6.14. 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 shift left 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.

6.15. Relational Operators

The values of expressions can be compared as follows:

relational-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. A pointer of type `void *` or `nil` may be compared to any other pointer. Pointers of any other type may only be compared to pointers of the same type.

6.16. 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.

6.17. 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).

6.18. 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 as any component of the expression evaluate to a 1. The operands can be any mix of arithmetic and pointer types.

6.19. 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 initialisation, 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.

6.20. 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.

```
aggr Readmesg                    /* Read message to file system */
{
        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 <-= (Readmesg)(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 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’.

6.20.1. Promotion

If both 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.

6.21. Binding and Precedence

The binding and precedence of the operators is as follows:

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

7. Statements

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

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

7.1. 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 namespace and do not conflict with other names. Labels have function scope.

7.2. 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.

7.3. 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.

7.4. 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*
`alt` *cbody*

cbody:
{ *caselist* }
!{ *caselist* }

caselist:
case-item
alt-item
caselist case-item

case-item:
`case` *constant-expression : statement*
`default` *: statement*

alt-item:
`case` *expression : 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. The *expression* is evaluated and converted into an integer. The integer is compared with the value specified in each `case`. If the integers compare the statement part of the `case` is executed. The `case` expression must yield an integer constant. For a single `switch` statement each case expression must yield a unique value. If no `case` is matched, the `default`

clause is executed. If the `default` is omitted then none of the `case` statements are executed.

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 descriminate 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 abitrary 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) c;

alt {
case a = <-c:
        print("received Aggr1");
        break;
case b = <-c:
        print("received Aggr2");
        break;
case c = <-c:
        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. A channel may not be used in two `alt` statements simultaneously.

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

7.5. 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 initialise 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.

7.6. Jump Statements

Jump statements transfer control unconditionally.

jump-statement:
`goto` *identifier* `;`
`continue` $count_{opt}$ `;`
`break` $count_{opt}$ `;`
`raise` $identifier_{opt}$ `;`
`return` $expression_{opt}$ `;`
`become` *expression* `;`

count:
integer-constant

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

7.6.1. Continue Statements

The `continue` statement may only appear as part of an iteration 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 the 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 */
}
```

7.6.2. Break Statements

The `break` statement may only appear as part of an iteration statement or selection statement. If *count* is omitted in an interaction statement then the break terminates the statement portion of the iteration loop and transfers control to the statement after the iteration statement. If *count* is supplied break causes termination of the iteration statement of some nested loop. *count* is the number of nested iteration loops to terminate. `break` with no *count* is the same as `break 1` . When used in a selection statement, `break`

causes the termination of the case in the selection statement.

7.6.3. Raise Statement

By default `raise` transfers control to the last `rescue` statement declared in the code. Execution has no effect on the connection between `raise` and `rescue` statements. If an identifier is supplied, control is transferred to the named `rescue` statement. `raise` is intended for use in error recovery. For example, these two fragments are equivalent:

```
alloc p;
rescue {
        unalloc p;
        raise;
}

if(error)
        raise;
```

```
alloc p;
goto notrescue;
        dorescue:
                unalloc p;
                goto nextrescue;
notrescue:
if(error)
        goto dorescue;
```

7.6.4. 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.

7.6.5. Become Statement

The `become` statement transforms the current function into the value of the *expression* given as its argument. If `expression` is not a function call the effect of `become` is exactly the same as `return`. However a function call causes the current function to be replaced by the called function. *expression* must have exactly the same type as the caller. Functions performing recursion in this manner are guaranteed to run in constant space.

7.7. Exception Statements

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

exception-statement:
`rescue` *identifier*$_{opt}$ *block*
`check` *expression* `;`

7.7.1. Rescue Statement

Under normal execution the 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. `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;
```

7.7.2. Check Statement

The `check` statement tests an assertion. If the assertion fails a runtime error aborts the program. The file and line number of the `check` statement are printed to standard error, the program is then aborted. The expression is evaluated and compared to 0. If the compare succeeds the assertion has failed. For example:

```
alloc ptr;
check ptr != nil;       /* Program aborts if allocation fails */
```

A compiler option is provided to omit `check` statements from trusted object code. By default `check` statements are included.

7.8. Process Control Statements

These statements create processes and coroutines:

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

The `proc` statement creates a new process. The new process starts running the named function. The arguments to *function-call* are evaluated by the original process. Processes are scheduled preemptively and the interleaving of the processes is undefined. Parent and child process share memory and file descriptors. The stacks of both processes are addressable, so it is possible to pass the address of an automatic between processes. The `task` statement creates a coroutine within a process. The arguments to *function-call* are evaluated by the original process. Tasks are non-preemptive and are scheduled during message passing and synchronisation primitives. The scheduling primitives that can cause task switching 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 they were 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 process that entered the `par` is guaranteed to be the same process that exits.

All tasks within a process will block until a system call completes.

7.9. Allocation Statements

Memory management statements allocate and free memory for objects from the heap:

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

alloclist:
expr
alloclist `,` *expr*

7.9.1. Alloc Statement

The `alloc` statement takes a list of pointers, which must also be l-values. 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 the expression will be assigned the value `nil`.

If the pointer has `chan` type, the runtime system will also initialise the new channel. If the channel is asynchronous then the specified number of buffers will be allocated.

7.9.2. Unalloc Statement

The `unalloc` statement returns memory to the heap. The argument to `unalloc` must be 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 abort the program.

8. Runtime Support

The synchronisation primitives used by the runtime system are made available to ALEF programs. Prototypes for these and other library functions are provided by the include file `alef.h`. This file should be included as the first item as:

```
#include <alef.h>
```

The standard include path for ALEF on Plan 9 is `/sys/include/alef`.

8.1. Lock

The `Lock` ADT provides spinlocks. Two operations are provided. `Lock.lock` claims the lock if free, otherwise is 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.

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

9. Yacc Style Grammar

The following grammar is suitable for implementing a yacc parser. Uppercase 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 ';'

zargs           :
                | '(' arglist ')'

ztname          : tname
                | AGGR
                | ADT
                | UNION

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

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

ztag            :
                | name
                | TYPENAME

setlist         : sname
                | setlist ',' setlist

sname           :
                | name
                | name '=' expr

name            : ID

memberlist      : decl
                | memberlist decl

vardecllist     :
                | ivardecl
                | vardecllist ',' ivardecl

ivardecl        : vardecl zinit

zinit           :
                | '=' zelist

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

```
vardecl         : ID arrayspec
                | indsp ID arrayspec
                | '(' indsp ID arrayspec ')' '(' arglist ')'
                | indsp '(' indsp ID arrayspec ')' '(' arglist ')'

arrayspec       :
                | arrayspec '[' zexpr ']'

indsp           : '*'
                | indsp '*'


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

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

autolist        :
                | autolist autodecl

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

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

slist           :
                | slist stmnt

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

clist           :
                | clist case

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

rbody           : stmnt
                | ID block

zlab            :
                | ID

stmnt           : nlstmnt
                | ID ':' stmnt

nlstmnt         : error ';'
                | zexpr ';'
                | block
                | CHECK expr ';'
```

```
                | ALLOC elist ';'
                | UNALLOC elist ';'
                | RESCUE rbody
                | RAISE zlab ';'
                | GOTO ID ';'
                | PROC expr ';'
                | TASK expr ';'
                | 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
                | CONTINUE zconst ';'
                | BREAK zconst ';'

zconst          :
                | CONST

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

castexpr        : monexpr
```

```
                | '[' typecast ']' castexpr
                | '(' typecast ')' castexpr

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

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

term            : '(' elist ')'
                | SIZEOF '(' typecast ')'
                | term '(' zarlist ')'
                | term '[' expr ']'
                | term '.' stag
                | '.' TYPENAME '.' stag
                | term '->' stag
                | term '--'
                | term '++'
                | term '?'
                | name
                | '.' '.' '.'
                | CONST
                | NIL
                | ENUM_MEMBER
                | FCONST
                | STRING
                | '$' STRING

stag            : ID
                | TYPENAME

zarlist         :
                | elist

elist           : expr
                | elist ',' expr

tlist           : typecast
                | typecast ',' tlist

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

xtname          : TYPENAME
                | INT
                | UINT
                | SINT
```

```
                | SUINT
                | BYTE
                | FLOAT
                | VOID
                | CHAN '(' typecast ')' bufdim

bufdim          :
                | '[' expr ']'

sclass          :
                | EXTERN
                | INTERN
                | PRIVATE
```