Abstract

This document is a formal specification of the Alef programming language. It specifies the syntactic features of the language and explains it semantics. Any compiler or interpreter for this language must conform to these specifics. Grammar rules are expressed using EBNF:

```
Production  = production_name "=" [ Expression ] "." .
Expression  = Alternative { "|" Alternative } .
Alternative = Term { Term } .
Term        = production_name | token [ "..." token ] | Group | Option |
   Repetition .
Group       = "(" Expression ")" .
Option      = "[" Expression "]" .
Repetition  = "{" Expression "}" .   (* Repeating 0 times is allowed. *)
```

(Taken from `https://golang.org/ref/spec`.)

Notes about the current implementation and examples are in italic:
This is an example.
Code snippets and grammar rules are represented using listings:

```
    this is a code snippet.
```

This document is a draft. You can find the reference implementation on `https://github.com/ecmma/alf`.

Contents

CHAPTER 1

Execution model

Alef is a concurrent programming language designed for systems software. Exception handling, process management, and synchronization primitives are implemented by the language. 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 description of the syntax and semantics of the implementation.
In the Alef execution model the term *process* refers to a preemptively scheduled thread of execution. A process may contain severl *tasks*, which are non-preemptively scheduled coroutines 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, while 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, therefore 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 (sending or receiving on/from channels) and synchronization (lock, unlock) operations. The term *thread* is used wherever the distinction between a process and a task is unimportant.

In the current implementation, processes are implemented as plain OS forks, while tasks are implemented as green threads, using a round robin scheduler implemented in the runtime – scheduling of tasks is realized through simple calls to `_alef_yield`.

CHAPTER 2

Type system

2.1 Basic types

Alef defines a small set of basic types:

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

where the given size is the minimum number of bits required to represent that type. The `float` type should be the highest precision floating point provided by the hardware, therefore format and precision are implementation dependent. The alignment of these types is implementation dependent as well. The `chan` type is actually a pointer to a runtime-system-defined object, and must be allocated before use, and are the size of a pointer. Polymorphic types are represented by a pointer and an hash of the type it currently represents. For a given implementation the polymorphic type has the same size as the following aggregate definition:

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

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. 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.

From the point of view of the lexer types are all identifiers; however, the compiler needs to distinguish identifiers and typenames; therefore, for the grammars, identifiers naming types will be tokens of type `Typename`.

```
BaseType = Typename [ ( ChanSpec | GenericInstantiation ) ] .
GenericInstantiation = "[" Variant "]" .
Variant = TypeCast { "," TypeCast } .
ChanSpec = "(" Variant ")" [ ChanBufDim ] .
ChanBufDim = "[" Expression "]" .
```

2.1.1 Channel types

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.

2.1.1.1 Sync channels

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.

2.1.1.2 Async channels

If buffers are specified, then an asynchronous channel is produced. 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.

2.1.1.3 Variant channels

If multiple types are specified in a channel definition, 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.

2.1.2 Polymorphic types

The polymorphic type can be used to dynamically represent a value of any type. A polymorphic type is identified by an identifier defined in a polymorphic type definition or as a parameter to a polymorphic abstract data type. Distinct identifiers 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, which consists of an integer tag (an identifier of the type, such as an hash) and a pointer to a value. Like channels, storage for the data must be allocated by the runtime.

2.1.3 Enumerations

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.

2.2 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. Both the `&` prefix operator and the `()` operator have distinct rules in prefix unary expressions and basic declarations, respectively.

```
ArraySpec = "[" [ Expression ] "]" { "[" [ Expression ] "]" } .
PtrSpec = "*" { "*" } .
```

2.2.1 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.

2.3 Complex types

Complex types may be either aggregates, unions, tuples, or abstract data types. These complex types contain sequences of basic types, derived types and other complex types, called members. Members are referenced by tag or by type, and 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.

An aggregate is a simple collection of basic, derived[1] and complex types with unique storage for each member. Unions, instead, store each member in the same storage and its size is determined by the size of the largest member. Abstract data types, adt's, are comparable to aggregates, but also has a set of functions to manipulate objects of its type and a set of visibility attributes for his members, to allow information hiding.

The declaration of complex types bind identifiers to such types, and after declaration such identifier can be used wherever a basic type is permitted. New type bindings may be defined from derived and basic types using the `typedef` statement.

The complex types are `aggr`, `adt`, `union` and `tuple`[2][3].

```
Type = BaseType | [ "tuple" ] "(" TupleList ")" .
TupleList = TypeCast "," TypeCast { "," TypeCast } .
```

What follows is a in-depth analysis of the meaning of the complex types. Grammar rules for deriving such constructs will be given in the Declarations section.

[1] **@TODO**: Can a function be defined inside an aggregate?

[2] **@TODO**: Maybe we could clarify, in a "mechanical" way, how complex type fit in the type system?

[3] **@TODO**: Clarify why tuples are different from other complex types, i.e. why they can be declared as basic types can (inline).

2.3.1 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. 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 parenthesized 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 */
}
```

2.3.2 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 */
};
```

2.3.2.1 Member functions (methods)

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 `* Identifier` (with the * preceding the name of the ADT), 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 `. Identifier` then the value of the ADT will be passed to the member function, rather than a pointer to it:

```
adt Point
{
    int x;
    extern int y;
    Point set(*Point); /* Pass &Point as 1st arg */
    Point clr(.Point); /* Pass Point as 1st arg */
    intern Point tst(Point);
};

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

2.3.2.2 Polymorphic and parametrized ADTs

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;
}
```

2.3.2.2.1 Bound and unbound parametrized ADTs

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.

2.4 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.

CHAPTER 3

Lexical analysis

Compilation starts with a preprocessing phase. An ANSI C preprocessor is used, and any directive valid in such standard is valid in Alef as well. 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]. Comments are removed by the preprocessor. Any form of comment accepted by the preprocessor is valid. For example, GNU's CPP might accept (and remove) both `/*...*/` and `//..` style comments.

3.1 Tokens

The lexical analyzer classifies tokens as identifiers, keywords, literals 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.

3.1.1 Reserved words

3.1.1.1 Keywords

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

`adt`	`aggr`	`alloc`	`alt`	`become`	`break`	`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`				

Keywords that identify types, such as `int` or `chan` are, from the point of view of lexical analysis, seen as identifiers. It will be the job of the compiler to make it so that at any time, if the parser has to distinguish between an actual identifier and a typename, the symbol table at that point will reflect the fact that `int` and `chan`, for example, are predefined types. In the same fashion, the `nil` keyword isn't interpreted as a keyword, but as an always-defined constant of type `void\*` of value `0`. In other words, to correctly build the parse tree the parser needs the information to discern identifiers and typenames

[1]Note that using, for example, GCC's CPP on an Alef source file will produce a compilation unit with "#"-directives, which aren't described in this document, but is instead the objective of the compiler engineer to manage these implementation-dependent notions.

(otherwise, type declarations such as `int* a` would be undiscernible from multiplication expressions, as both would be seen as `identifier op_mul identifier`.)

3.1.1.2 Delimiters and operators

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

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

3.1.2 Identifiers

An identifier, also called a lexical name, is any sequence of alpha-numeric characters and the underscore character `\_`. An identifiers binds a name to a semantic object such as a type, a function or a variable. Identifiers starting with "ALEF" are reseved for use in the runtime system.

```
ASCII      = "a" | "b" | ... | "z" | "A" | "B" | ... | "Z" .
Digits     = "0" | "1" | ... | "9" .
Identifier = ( "_" | ASCII ) { ( "_" | ASCII | Digits ) } .
```

Identifiers may define variables, types, functions, function prototypes or enumerators. An identifier has associated a scopeScopes and storage class Storage classes.

3.1.3 Literals

Alef literals are integer and floating point numbers, characters, strings and runestrings (UTF-8 strings.) There are five types of constant:

```
Literal       = StringLit | RunestringLit | CharLit | IntLit | FloatLit .
StringLit     = '"' { ASCII | Escapes } '"' .
Escapes       = "\" ("0" | "n" | "r" | "t" | "b" | "f" | "a" | "v" | "\" |
   '"' ) .
RunestringLit = @TODO
CharLit       = @TODO .
IntLit        = [ "0" [ ( HexLit | OctalLit ) ] ] | DecimalLit .
HexLit        = ("x" | "X") { HexDigits }.
HexDigits     = "0" ... "9" | "A" ... "F" | "a" ... "f" .
OctalLit      = { OctalDigits } .
OctalDigits   = "0" ... "7"
DecimalLit    = ( "1" | ... | "9" ) { Digits } .
FloatLit      = @TODO .
```

Character literals have the type `uint` and can hold UTF-8 code points. String literals have type `static array of byte` and are NUL (`\0`) terminated (appended by the compiler); therefore, the sizeof operator applied to a string yields the number of bytes including the appended NUL. Rune string literals are sequences of UTF-8 code points and have type `static array of uint`, and are NUL (`U+0000`) terminated (appended by the compiler); therefore, the sizeof operator applied to a string yields the number of runes in the runestring, in terms of `sizeof(uint)` including the appended NUL. 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
```

Float literals have type `float`. Integer literal have type `int`.

CHAPTER 4

Declarations

4.1 Programs

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

```
Program     = { Declaration } .
Declaration = [ Visibility ] ( SimpleDecl | ComplexDecl | TypeDefs ).
Visibility  = "intern" | "extern" .
```

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.

4.1.1 Scopes

Scopes define where a named object, that is an identifier with a precise semantic meaning, can be referenced.

4.1.1.1 File scope

A declaration introduces an identifier and specifies its type. A definition is a declaration that also reserves storage for an identifier. Declaration at the file scope are implicitly assumed to be "extern" declarations, that is, visible to all compilations units. If a declaration is preceded with the "intern" keyword, its scope is narrowed to its compilation unit only. Variable declarations (that is, variable declarations preceded by the keyword "extern") aren't definitions, and therefore do not allocate space for the variable. Similarly, function prototypes do not allocate space[1].

4.1.1.2 Type scope

Members of complex types (ADT's, aggregates or union) are in scope only when an access operator is applied to objects of their appartaining type. Members of unions and aggregates are always accessible from outside. Access to members of ADT's can be restricted (or enabled) using visibility specifiers

[1] **@TODO**: The use of extern as a keyword to make a variable visible in the entire compilation unit and to make non-allocating definitions is highly unclear – we might use a different keyword to indicate non-allocating definitions.

"intern" and "extern": variable members are, by default, "intern", that is not accessible from outside. Methods (function members of an ADT) are by default "extern".

4.1.1.3 Function scope

Labels and raise statement's identifiers can be referenced from the start of a function to its end, regardless of the position of the declaration.

4.1.1.4 Local scope

Local identifiers are declared at the start of a block[2]. A local identifier has scope starting from its declaration to the end of the block in which it was declared.

4.1.2 Storage classes

While scopes define where identifiers have meaning, storage classes define the lifetime "of the meaning", that is, when variables and functions are created and deleted, and to what value they're initialized.

4.1.2.1 Automatic storage class

Automatic objects are created at the entry of the block in which they were declared, and their value is undefined upon creation.

4.1.2.2 Parameter storage class

Function parameters are created by function invocation and are destroyed at function exit. They have the value of the values passed by the caller.

4.1.2.3 Static storage class

Static objects exist from invocation of the program until termination, and uninitialized static objects have, at creation, the value 0.

4.2 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. Function pointer declarations have *per sé* rules.

```
SimpleDecl = Type [ PtrSpec ] ( FuncPtr | BaseDecl ) .
FuncPtr    = "(" [ PtrSpec ] Identifier ( FuncPtrFuncDecl | FuncPtrVarDecl )
    .
BaseDecl   = Identifier ( FuncDecl | VarDecl |  MethDecl ) .
```

[2] **@TODO**: This means that, in a function's body, automatic variables can be declared only at the start, before any statement.

4.2.1 Function pointer declarations

```
FuncPtrFuncDecl  = "(" [ ParamList ] ")" ")" "(" [ ParamList ] ")" ( ";" |
   Block ) .
ParamList         = Param { "," Param } .
Param            = SimpleParam | TupleParam  | "..." .
SimpleParam      = BaseType [ PtrSpec ] [ ( BaseParam |  FuncPtrParam ) ] .
BaseParam        = Identifier [ ArraySpec ] .
FuncPtrParam     = "(" PtrSpec [ Identifier ] [ ArraySpec ] ")"  "(" [
   ParamList ] ")" .
TupleParam       = "tuple" "(" TupleList ")" [ [ PtrSpec ] ( BaseParam |
   FuncPtrParam ) ] .
FuncPtrVarDecl   = [ ArraySpec ] ")"  "(" [ ParamList ] ")" [ "="
   InitExpression ] ( ";" |  "," [ PtrSpec ] ( "(" [ PtrSpec ] Identifier
   FuncPtrVarDecl | Identifier VarDecl ) ) .
```

The parameters received by a function taking variable arguments are referenced using the ellipsis `...`. The token `...` yields is a value of type `pointer to void`. The value points at the first location after the formal parameters.

4.2.1.1 Examples of function pointer declarations.

4.2.1.1.1 A variable of type array of function pointer.

In this example,

```
int * (* func_ptr[2] ) (int, bool);
```

4.2.1.1.2 A function returning a function.

In this example,

```
int * (func_func (float, char)) (int, bool);
```

4.2.1.1.3 A notable case

In this example,

```
int * (func) (int, bool);
```

Is interpreted as a function, not as a variable of type pointer to a function.

4.2.2 Variable, function and method declarations

```
FuncDecl = "(" [ ParamList ] ")" ( ";" | Block ) .
VarDecl  = [ ArraySpec ] [ "=" InitExpression ] ( ";" | "," [ PtrSpec ] (
   "(" [ PtrSpec ] Identifier FuncPtrVarDecl | Identifier VarDecl ) .
MethDecl = "." Identifier "(" [ ParamList ] ")" Block .
```

4.2.2.1 Initializers

Only simple declarations at the file scope may be initialized[3]. 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

[3]We may want to change this.

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.

```
InitExpression     = Expression | ArrayElementInit | MemberInit | BlockInit
   .
ArrayElementInit   = "[" Expression "]"  ( Expression | BlockInit ) .
MemberInit         = "." Identifier Expression .
BlockInit = "{" [ InitExpressionList ] "}" .
InitExpressionList = InitExpression [ "," InitExpression ] .
```

4.3 Complex type declaration

Complex declarations define new aggregates, unions, ADT's and enums in the innermost active scope.

```
ComplexDecl = ( AggrDecl | UnionDecl | AdtDecl | EnumDecl ) ";" .
```

4.3.1 Unions and aggregates

```
AggrDecl        = "aggr" [ Identifier ] "{" { AggrUnionMember } "}" [
   Identifier ] .
UnionDecl       = "union" [ Identifier ] "{" { AggrUnionMember } "}" [
   Identifier ] .
AggrUnionMember = ComplexDefs | VarMember .
VarMember       = Type [ [ PtrSpec ]  ( SimpleMember | FuncPtrMember ) { ","
    [ PtrSpec ] ( SimpleMember | FuncPtrMember ) } ] ";" .
SimpleMember    = Identifier [ ArraySpec ] .
FuncPtrMember   = "(" [ PtrSpec ] Identifier [ ArraySpec ] ")" "(" [
   ParamList] ")" .
```

4.3.2 Abstract data types

```
AdtDecl                = "adt" [ Identifier ] [ AdtGenSpec ] "{" { AdtMember
    } "}" [ Identifier ] .
AdtGenSpec             = "[" Identifier { "," Identifier "}" "]" .
AdtMember              = [ Visibility ] Type [ [ PtrSPec ] (
   AdtFuncPtrMember | AdtBaseMember ) ] ";" .
AdtFuncPtrMember       = "(" [ PtrSpec ] Identifier ( AdtFuncPtrMethodMember
    | AdtFuncPtrVarMember ) .
AdtFuncPtrMethodMember = "(" [ AdtMethodRefParam [ "," ParamList ] ] |
   ParamList ")" ")" "(" [ ParamList ] ")" .
```

```
AdtFuncPtrVarMember    = [ ArraySpec ] ")" "(" [ ParamList ] ")"  [ "," [
   PtrSpec ] ( "(" [ PtrSpec ] Identifier AdtFuncPtrVarMember | Identifier
   AdtVarMember ) ] .
AdtBaseMember          = Identifier ( AdtMethodMember | AdtVarMember ) .
AdtMethodMember        = "(" [ AdtMethodRefParam [ "," ParamList ] ] |
   ParamList ")" .
AdtMethodRefParam      = ( "*" | "." ) Identifier [ Identifier ] .
AdtVarMember           = [ ArraySpec ] [ "," [ PtrSpec ] ( "(" [ PtrSpec ]
   Identifier AdtFuncPtrVarMember | Identifier AdtVarMember ) ] .
```

4.3.3 Enumerators

```
EnumDecl   = "enum" [ Identifier ] "{" { EnumMember } "}" .
EnumMember = Identifier [ "=" Expression ]
```

4.4 Type definitions

Type definitions are declarations which start with the keyword “typedef”. Type definitions can introduce new polymorphic variables in the innermost active scope, forward references to complex types and new names for basic and derived types.

```
TypeDefs       = "typedef" ( PolyVarTypeDef | ForwardDef )
PolyVarTypeDef = BaseType [ PtrSpec ] ( DerivedTypeDef | FuncPtrTypeDef )
   ";" .
DerivedTypeDef = Identifier [ ArraySpec ] .
FuncPtrTypeDef = "(" [ PtrSpec ] Identifier [ ArraySpec ] ")" "(" [ParamList
   ] ")" .
ForwardDef     = ( "aggr" | "union" | "adt" ) Identifier ";" .
```

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;
};
```

CHAPTER 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 by the specification and is implementation dependent.

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:

```
PrimaryExpression = Identifier | Literal | "nil" | [ "tuple" ] "("
   ExpressionList ")" .
```

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

```
PostfixExpression = ( PrimaryExpression | AdtNameCall ) { PostfixOperand } .
AdtNameCall       = "." Typename "." Identifier .
PostfixOperand    = ArrayAccess | FuncCall | MemberAccess | IndirectAccess |
    UnaryPostfix .
ArrayAccess       = "[" Expression "]" .
FuncCall          = "(" [ ExpressionList ] ")" .
MemberAccess      = "." Identifier .
IndirectAccess    = "->" Identifier .
UnaryPostfix      = "++" | "--" | "?" .
ExpressionList    = Expression { "," 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

```
*((PrimaryExpression)+(Expression))
```

One of the expressions must be of type pointer, the other of integral type.

5.3.2 Function calls

Function call postfix operators 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.

5.3.2.1 Function call promotions

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 */
}
```

5.3.2.2 ADT namecalls

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.

5.3.2.3 Polymorphic promotions

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 `(*PostfixExpression).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]; /* s.data[s->tos] has been unallocated. */
```

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 Prefix expressions

```
UnaryExpression = PostfixExpression | UnaryPrefix | CastPrefix .
UnaryPrefix     = ( "<-" | "++" | "--" | "zerox" ) UnaryExpression .
CastPrefix      = UnaryOperator Term .
UnaryOperator   = ( "?" | "*" | "&" | "!" | "+" | "-" | "~" | "sizeof" ) .
```

5.4.1 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.

5.4.2 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.

5.4.3 Send and Can send

The postfix operator `<-`, on the left-hand side of an 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.

5.4.4 Indirection

The unary prefix 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`.

5.4.5 Unary plus and minus

Unary plus is equivalent to `(0+(UnaryExpression))`. Unary minus is equivalent to `(0-(UnaryExpression))`. An integral operand undergoes integral promotion. The result has the type of the promoted operand.

5.4.6 Bitwise negate

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

5.4.7 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`.

5.4.8 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 its operand. 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`.

5.4.9 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.

5.5 Term expressions

```
Term           = UnaryExpression | CastExpression | AllocExpression .
CastExpression  = "(" TypeCast ")" Term .
AllocExpression = "(" "alloc" Typename ")" Term .
TypeCast        = BaseType [ PtrSpec ] [ FuncCast ] | "tuple" "(" TupleList
   ")" .
FuncCast        = "(" [ PtrSpec ] ")" "(" [ ParamList ] ")" .
```

5.5.1 Cast expressions

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

5.6 Binary expressions

Binary operators in LL grammars lose their left associativity. A given implementation will use Dijkstra's shunting yard algorithm or a Pratt Parser. Nonetheless, a list of valid expressions follows.

```
Expression   = Term | Expression BinaryOp Expression .
BinaryOp     = SumOp | MulOp | LogOp | ShOp | CompOp | EqOp | AssOp | IterOp
    .
SumOp        = "+" | "-" .
MulOp        = "\*" | "/" .
LogOp        = BitwiseLogOp | "&&" | "||" .
BitwiseLogOp = "^" | "&" | "|" .
ShOp         = "<<" | ">>" .
CompOp       = "<" | ">" | ">= " | "<=" .
EqOp         = "==" | "!=" .
AssOp        = "=" | ":=" | "<-=" | (SumOp | MulOp | BitwiseLogOp | ShOp )
   "=" .
IterOp       = "::" .
```

5.6.1 Multiply, divide and modulus

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.

5.6.2 Add and subtract

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.

5.6.3 Shift operators

The shift operators perform bitwise shifts. If the first operand is unsigned, << performs a logical left shift by a number of bits as its right operand. If the first operand is signed, << performs an arithmetic left shift by a number of bits as its right operand. The left operand must be of integral type. The >> operator is a right shift and follows the same rules as left shift.

5.6.4 Relational Operators

The values of expressions can be compared using relational operators. 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*`.

5.6.5 Equality operators

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.

5.6.6 Bitwise logic operators

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

5.6.7 Logical operators

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.

5.6.8 Constant expressions

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

5.6.9 Assignment

The assignment operators are:

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

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`.

5.6.9.1 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.

5.6.9.2 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.

5.6.10 Iterators

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

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.

5.7 Associtivity and precedence of operators

Precedence	Assoc.	Operator
14	L to R	() [] ->
13	R to L	! ~ ++ – <- ? + - * & (cast) sizeof zerox
12	L to R	* / %
11	L to R	+ -
10	L to R	« »
9	R to L	::
8	L to R	< <= > >=
7	L to R	== !=
6	L to R	&
5	L to R	^

	Precedence	Assoc.	Operator
4	L to R		|
3	L to R		&&
2	L to R		||
1	L to R		<-= = := += -= *= /= %= &= ^= |= «= »=

CHAPTER 6

Statements

Statements do not yield values, but have side effects.

```
Statement = [ Expression ] ";" | LabelStmn | Block | SelectionStmn |
   LoopStmn | JumpStmn | ExceptionStmn | ProcessStmn | AllocationStmn .
```

6.1 Label statements

```
LabelStmn = Identifier ":" Statement .
```

6.2 Blocks

```
Block = [ "!" ] "{" [ { AutomaticDeclarations } ] | { Statement } "}" .
AutomaticDeclarations = Type [ PtrSpec ] ( FuncPtrDeclarator FuncPtrVarDecl
    |  Identifier VarDecl)
```

6.3 Selection

```
SelectionStmn = IfElseStmn | SwitchStmn | TypeofStmn | AltStmn .
IfElseStmn    = "if" "(" Expression ")" Statement [ "else" Statement ] .
SwitchStmn    = "switch" Expression SwitchBody .
SwitchBody    = ["!"] "{" { SwitchCase } "}" .
SwitchCase    = "case" Expression ":" { Statement } | "default" ":" {
   Statement } .
TypeofStmn    = "typeof" Expression TypeofBody .
TypeofBody    = ["!"] "{" { TypeofCase } "}" .
TypeofCase    = "case" CastExpression ":" { Statement } | "default" ":" {
   Statement } .
AltStmn       = "alt" SwitchBody .
```

6.4 Loops

```
LoopStmn  = WhileStmn | DoStmn | ForStmn .
WhileStmn = "while" "(" Expression ")" Statement .
DoStmn    = "do" Statement "while" "(" Expression ")" .
ForStmn   = "for" "(" [ Expression ] ";" [ Expression ] ";" [ Expression ]
   ")" Statement .
```

6.5 Jumps

```
JumpStmn     = GotoStmn | ContinueStmn | BreakStmn | ReturnStmn | BecomeStmn
    .
GotoStmn     = "goto" Identifier ";" .
ContinueStmn = "continue" [ IntLit ] ";" .
BreakStmn    = "break" [ IntLit ] ";" .
ReturnStmn   = "return" [ Expression ] ";" .
BecomeStmn   = "become" Expression ";" .
```

6.6 Exceptions

```
ExceptionStmn = RaiseStmn | RescueStmn | CheckStmn .
RaiseStmn     = "raise" [ Identifier ] ";" .
RescueStmn    = "rescue" ( Statement | Identifier Block ) .
CheckStmn     = "check" Expression [ "," StringLit ] ";" .
```

6.7 Process control

```
ProcessStmn = ProcStmn | TaskStmn | ParStmn .
ProcStmn    = "proc" ExpressionList ";" .
TaskStmn    = "task" ExpressionList ";" .
ParStmn     = "par" Block ";" .
```

6.8 Allocations

```
AllocationStmn = AllocStmn | UnallocStmn .
AllocStmn      = "alloc" ExpressionList  ";" .
UnallocStmn    = "unalloc" ExpressionList ";" .
```