ALSO **Language Reference Manual**

Andrew Myers (andru@cs.cornell.edu)

April 2021

Contents

1 Introduction

ALSO is a persistent, dynamically typed object-oriented language for building extensible servers. It was originally developed during the winter of 1990 when I was a graduate student at MIT, but I then abandoned it. Although it predates or was contemporary with languages like Python and JavaScript, it has surface similarities. It also has some unique and useful features, such as nested transactions and queryable properties. I resurrected it in 2018 and updated the expression syntax (in particular, that of lambda expressions) to look more familiar.

ALSO borrows features from several programming languages. Like JavaScript, ALSO is a prototype-based object-oriented language, inspired by the language Self [US87]. ALSO supports first-class, lexically scoped functions as in Scheme, and it has exceptions and exception handling. ALSO is not statically typed but it is strongly typed. The statement syntax is from the C family. Semicolons are optional, making code look cleaner, but unlike in Python and JavaScript, whitespace is *not* significant.

ALSO is intended to be used to implement various kinds of networked servers, particularly ones that are intended to be extended over time. It was designed originally to support MUD ("multi-user dungeon") systems, which are interactive, internally reprogrammable multi-player adventure environments. A MUD system combines aspects of an operating system, a language, and a database. As a result, the ALSO language has several unusual features:

Persistence. ALSO is a persistent programming language: [ACC82] reachable objects created during execution are automatically persisted. There is no need for a back-end database to save persistent state. Persistence is implemented via periodic checkpointing of reachable objects.

Invertible properties. An *invertible property* can be queried to find the set of objects that have a given value for that property, automatically maintaining many-to-one/one-to-many relations. Invertible properties make it easy to construct hierarchically nested containers without worrying about data structure inconsistencies arising between parent and children.

Nested, atomic transactions. Especially important for extensibility is that ALSO supports nested transactions, similar to Argus [LS83]. Also transactions are integrated with exception handling, which simplifies error recovery because unhandled exceptions automatically cause side effects to be rolled back, avoiding inconsistent changes to data structures.

Property objects. Unlike in other object-oriented languages such as JavaScript, properties (methods and fields) are named by objects rather than by strings. All objects are essentially arbitrary hash tables with inheritance. Because properties are not strings, different namespaces can be used to manipulate the same objects, for example for internationalization. Property objects themselves have properties that control aspects of their use: for example, whether they are invertible. Arrays can be implemented by using integers as properties.

Encapsulation via access control. During execution of an ALSO program, there is an ambient *authority* that controls which object properties may be accessed. Each property defines what authority is required to read, write, or invoke that property. Unlike in class-based object-oriented languages, encapsulation of object state is achieved through this fine-grained access-control mechanism. The access control mechanism also helps avoid confused-deputy attacks by automatically attenuating authority.

Eval with controlled namespaces. An operation `eval` allows arbitrary strings to be evaluated as ALSO code, while retaining control over what names in the environment are made available to the code. ALSO supports constructing local namespaces, so access from code extensions to objects or to methods can be controlled by exposing only the properties that are safe to expose.

Primitive prototypes. Primitive values such as integers do not have their own properties, but can still be treated as objects that inherit properties from standard prototype objects, such as `Int`.

A small MUD system called "A-mud" has been developed on top of ALSO, and is the largest ALSO program — more than 3,500 lines long at present. When run in HTTP mode, it implements a simple web server that is conveniently accessible using a simple JavaScript-based client. The server is currently reachable at `http://www.cs.cornell.edu/andru/amud`.

To give an idea of the syntax of ALSO, here are some short code samples:
A function that computes whether a number is prime.

```
is_prime(n) is {
    var i = 2
    loop {
        if (i*i > n) break
        if (n%i == 0) return F
        i = i+1
    }
    return T
}
```

An inherited method that determines recursively whether a person is holding an object or holding an object that contains it.

```
person.is_holding(o) =
    o.location == self || self.is_holding(o.location)
```

A Quine (self-reproducing program)

```
p is ["p␣is␣","print␣p.first␣|␣p.unparse;␣print␣p.rest.first"]
print p.first | p.unparse; print p.rest.first
```

2 Values

There are nine different kinds of values in ALSO: integer, character, boolean, string, nil, list, function, record (ordinary object), and connection. Unlike in many languages, each of these values acts like an object. For example, they all have a prototype object that can be used to augment them with new operations.

Among these values, only records and connections are *mutable*, meaning that the data they contain can be modified.

2.1 Integer

Integers may be positive or negative. Integer constants are named in the usual way. Integer literals may be given in hexadecimal if preceded by `0x`. Most of the usual C arithmetic operations may be performed on

integers: addition, subtraction, multiplication, integer division, modulo, bitwise-and and or. The prototype object for integers is `Int`.

```
1, 0, -56, 234, 0x0A
```

2.2 Character

Characters can be any Unicode character except character 0 (`NUL`). Characters can be converted to other types, and used for output. Their prototype object is `Char`.

Character literals are expressed with single quotes. A backslash may be used to denote special characters, as in C (*e.g.* `'\n'` is the newline character). In string literals, the escapes `\x`, `\u`, and `\U` may be used to specify character codes using 2, 4, or 8 hexadecimal digits. In addition, most Unicode characters may be given directly, encoded in UTF-8.

```
'a', '\n', '\\', '\u2200', '∀'
```

2.3 Boolean

There are two booleans: true (`T`) and false (`F`). Booleans are used by the `if` statement. The boolean operators `&&` (logical and) and `||` (logical or) are supported. Their prototype object is `Bool`.

```
T, F
```

2.4 String

A string is a character sequence of any length including zero. A string constant is expressed using double quotation marks, and characters in the string are specified as in character literals, except that embedded newlines are allowed, and whitespace immediately following a newline is ignored (a backslash can be used to prevent it from being ignored).

Strings may be manipulated in several ways: substrings or individual characters may be extracted, and a string may be split into tokens around delimiter characters. Strings also support searching and substitution operators.

```
"hello", "string with newline\n", "",
"string with embedded
   newline"
```

2.5 Null

`NIL` is a special value that is used to mean "no information". Under certain conditions, some built-in operators will return this value. Its prototype object is `Null`.

```
NIL
[].first == NIL
```

2.6 List

A *list* contains a sequence of values of any type. The first item of a list may be found using the `first` method, and the remainder of the list is obtained with the `rest` method.

A list expression is specified with square brackets, and the items in the list are separated by commas. When a list expression is evaluated, the result is a list containing the values of each of the item expressions.

```
[1, 2, 3]
[]
[[ 1 ],[ 'x' ], [[]]]
[‘a’, "string", 0]
```

2.7 Function

A function in ALSO acts like a mathematical function: it takes in arguments and returns a result. ALSO functions do not demand that their arguments be of any fixed type, or even that a particular number of arguments be supplied.

A function is a value. It can be passed to other functions as an argument; it can be returned as a result.

Functions are created using lambda expressions, which specify a fixed list of argument variables that arguments are placed into. If you supply too few arguments, the remaining argument variables are filled with the value `NIL`; too many arguments, and the extra argument values are ignored.

There are a number of built-in functions that implement the methods of built-in types. For example, `Int.rand:` is a native function that computes a random number.

The following example uses a lambda expression to create a function that squares its argument and returns the result:

```
f is {x -> return x*x}
f(5) == 25
```

The "`f is`" part has nothing to do with the function. It just defines `f` to take on the value of the expression after the word "`is`". Formal parameters to the function are ordinary identifiers, separated by commas. This function has one argument, named `x`. If the function is called with the expression `f(5)`, the parameter variable `x` will contain `5`. The parameter list is optional — for example, `{ 5 }` is the constant function that always returns 5. Argument variables are just ordinary variables — they can be assigned to.

Functions can also be introduced conveniently in applicative form, by writing the arguments with the name of the function:

```
f(x) is x*x
f(5) == 25
```

The first example function contains one statement, a `return` statement, which causes the function to terminate immediately and return the specified result. However, a `return` is not needed to return a result. The value returned by a function is always the value of the last statement executed by the function. In fact, we could rewrite the function as `{x->x*x}` and it would have the same effect. The statement `x*x` (any expression may be used as a statement) will be the first, last, and only statement executed, so its value will be returned as the result of the function.

If a function contains no statements, it will return `NIL`.

Some other examples using function expressions are:

```
{return "hello"}
{x,y -> if (x) y else y - 1}
{}
{f,x -> var x10 = x*10; f(x10) }({y -> y + 1}, 2) == 21
church-zero(f) is {x->x}
church-one(f) is {x->f(x)}
church-increment(f) is {n->{f->{x->n(f)(f(x))}}}
```

2.8 Record

A record corresponds to the usual notion of an object, so we use the two terms largely interchangeably. A record is a collection of mutable *properties*. A property is a pair of two items: an immutable *property key* and a (usually mutable) *property value*. The property key is used to look up the associated property value. The value of a property can be changed by assigning to it. Typically, other records are used as keys. However, any kind of value (except a function or a list containing a function) may be used as a property key, so objects can be used as associative arrays.

Each object, including primitive values, has a *prototype object*, also known as its *parent object* — a record from which it *inherits* any properties that it has not specifically overwritten. An inherited property is still owned by the parent object, so any modifications to the value of that property are visible in all the children that inherit from it. Objects are typically created as a modification of some parent by using the `with` operator.

Property operations

Various property operations may be performed on an object. Properties may be *invoked* using the dot (`.`) operator. On the left side of the dot is the object; on the right side is the property key. Arguments to the invocation may be specified as well.

If a property with the specified property key is contained in the object, the property value is invoked. If the property is not contained in the object, the object's parent is checked for the property, and so on up the parent chain. If no object in the parent chain has the property, the `no-such-property` failure occurs.

Invoking a value is the same as returning the value unless the value is a function. If the property value is a function, the function is invoked. Invoking a property is implicit, because this allows attributes of objects to be redefined as pieces of code that compute the attribute value rather than simply storing it, blurring the distinction between value-like and functional properties. This feature makes the system more extensible.

When a functional property value is invoked, the object is always passed as the first argument to this function, and any arguments supplied to the property invocation are passed to the function along with the optionally specified arguments. Even when a property is inherited, the original object — and not the parent in which the property was found — is the object passed as the first argument to the property value function.

```
object.key == value
object.key(arg1, arg2) == value
hair.color
andru.gender.nominative
me.put-in(wand, bag)
andru.name == "andru"
factors.6 == [1, 2, 3]
10.rand == 7
```

It is also possible to check whether an object has a property using the `has` method. It returns either `T` or `F`.

```
if (o.has(p))
x = o.p
```

To avoid invoking any functions, a property may be read without invocation by using a dollar sign (`$`) instead of a dot. The semantics are the same as for property invocation, except that the property value is simply returned. No arguments may be supplied.

```
object$key == value
hair$color
me$put-in
month-index$"jan" == 1
andru$names == ["andru"]
person$name == {self -> self.names.first }
```

Property values may be written using the standard assignment syntax for fields of records. If no property contained in the object has the property key that is assigned to, a new property will be created and attached to the object. This new property will break the chain of inheritance.

```
object.key = new-value
male.possessive = "his"
me.put-in = {obj, container ->
             obj.transfer(obj.location, container) }
me.put-in(container) = {
             self.transfer(self.location, container) }
factors.28 = [1, 2, 4, 7, 14]
```

Properties may be assigned as methods by providing identifier arguments to the assignment. In this case, the right hand-side must be a lambda expression; an extra formal parameter `self` is automatically prepended to the argument list.

Properties whose property key is *invertible* make available a fourth operation: they may be *queried*. Property keys are invertible if they are objects whose `invertible` property has the value `T`.

Properties are queried by the query operator: the question mark, `?`. On the left side of the operator is placed the property to be queried, and on the right side is placed the property value. The result of a query `p?v` will be a list of all objects `o` such that `o.p == v`. The value `v` must be a record object, since values of invertible properties may only be objects. An attempt to assign a non-record to an invertible property will result in a run-time type-check error.

```
property?value == [object1, object2, ...]
color?blond == [hair]
```

Finally, properties may be removed from objects using the `delete` method. When a property is deleted, the inheritance chain is restored: the property is subsequently inherited from the parent object, assuming that the parent either has or inherits that property.

```
o.delete(p)
a.delete(i+1)
```

Creating objects

Objects are created with the **`new`** and `with` operators. The **`new`** operator creates a new object using an *initialization list*. An initialization list specifies new properties that are to be created at the same time as the object and attached to it. An object created with `new` has no parent object. The `with` operator takes a parent object and an initialization list, and constructs a new object that inherits from the parent but is modified by the list.

```
new()
var point = new (x = 10, y = 20)
var new-hair = hair with (color = red, length = 24)
factors = new (15 = [1, 3, 5], 17 = [1])
```

Special property keys

Certain objects are *well-known objects* that have a special meaning to ALSO when used as property keys (See also section 10).

`parent` This immutable property contains the parent of the object. All objects except `Object` have a parent property; the parent of `Object` is `NIL`.

`properties` This property cannot be written to; it contains a value that is a list of all the properties currently contained in the object, except for the special property keys discussed here.

`immutable` This property determines whether the object containing it is an immutable property. Immutable properties cannot be changed; they can be given values only by at object initialization. The `immutable` property can only have values of `T` or `F`, and is itself immutable. That is, whether a property is immutable or not cannot be changed.

`invertible` This immutable property determines whether the object containing it is an invertible property. Invertible properties can be queried, as described above.

`endorse` This immutable property determines whether methods invoked using this property key automatically possess the effective authority of the property key. See Section 8 for details.

2.9 Connection

A connection represents the state of a communication connection to the external environment. Connection objects support operations to query the state of this connection, modify the way that input from them is handled, send output, and ultimately close the connection.

Connections cannot be created by the user. Once a connection comes into existence, any input which arrives on the connection is passed to the connection's attached *dispatch function*. The dispatch function may be changed, but its initial value is always defined by the `login` object (See section 10).

The dispatch function should expect two arguments. The argument is the connection itself; the second is ordinarily a string value containing received input. If a connection is closed from the other end, `NIL` is sent as the second argument, and the connection is then automatically closed.

Enqueuing output on a connection is treated as a mutation, so the output is never delivered to the other side of the connection if the transaction that enqueued the output fails.

3 Identifiers and Environments

Identifier names consist textually of a string which begins with a letter (upper- or lower-case) and continues with any number of letters, digits, underscores, or hyphens. An identifier may not be more than 100 characters long.

```
a, z29, name-of-the-rose, p_sample_2
```

The fact that hyphens are allowed in identifiers can lead to unexpected results if you are using subtraction: be sure to place spaces around minus signs, since `x-y` is interpreted as a single identifier.

Identifiers refer either to local variables or to constants. Local variables must be declared before use, and thus are easy to identify. Constants are defined by the *compilation environment*, which is provided at the time that the program containing the identifier is executed.

local variables Like C, ALSO allows you to nest scopes to arbitrary depth. Variable names can be created in a scope with a `var` declaration, and identifiers will resolve to the nearest preceding scope declaration, if any.

Assignment can only be performed on local variables, and not on constant definitions. An attempt to assign to a constant will produce a compile-time error.

Local variables are created either by being declared as arguments to a function, or by the `var` statement. Argument variables may be used throughout the body of the function. `var` variables may be used in the part of the nearest enclosing function expression which follows the `var` statement.

For example, in the function `f`:

```
f(x) is {
        loop {
          x = x - 1
          var n = x
          if (n == 0) break
        }
        print x
    }
```

the argument variable `x` is available throughout the entire function expression, whereas the local variable `n` is only available in the `if` statement following its declaration.

compilation environment The compiler looks for identifiers that are determined not to be local variables in the compilation environment. The compilation environment is an argument to the compiler when it is invoked by the `eval` operator. An environment is simply an ALSO value, and the interpretation of this value depends on its type:

list environment A list may specify definitions for variables. The list must contain two-item lists, in which the first item is the identifier and the second item is the value to which the identifier is bound. For example, the list `[["a",2],["b",["c", 4]]]` binds the identifier `a` to the integer `2` and the identifier `b` to the list `["c", 4]`.

function environment Every function value contains definitions for all of the compilation environment identifiers that are used inside the function. If a function value is used an environment, it will provide exactly those definitions. Using a function value `f` as an environment is exactly equivalent to using the list value `defns f` as an environment.

object environment An object environment resolves the definition of identifiers by invoking its `lookup` method, if any. If the invocation of the `lookup` method fails, the identifier is considered to be unbound.

Object environments provide a powerful hook into the internal name mechanism of the compiler, since they can be used to provide an arbitrary mapping from names to values — a mapping that can even change during compilation. This feature is particularly useful with the `eval` operation, since a later statement can use identifiers whose binding is created by the execution of a previous statement.

Identifiers can also be formed by explicitly specifying an environment, using the `::` operator. The left side of the operator must be an identifier that evaluates at compile time to a valid environment of one of the three types specified above, and the right side must be a valid identifier name.

```
myModule::foo
a::b::c
```

4 Programs

A program in ALSO is a piece of text that is evaluated. A program is a sequence of commands: statements, definitions, and imports.

A program produces two results: a value, which is the result of the last statement executed in the program, or the value of the last definition; and a list environment. This environment contains all of the public definitions in the program.

A definition is public if it is declared to be so. All identifiers defined in the program may be used within the program, but only public identifiers are exported in the returned environment.

statements The syntax of statements is described in section 5.

```
a[i][j] = b[i][k] * c[k][j]
if (fact(4) == 24) print "Evaluation test ok."
me.say("hi")
```

definitions Definitions bind the result of evaluating an expression to a particular identifier. The identifier is placed on the left side of `is`, and the expression on the right. The definition will be in force for the remainder of the program. An identifier may not be used before it is defined, except in its own definition.

A definition may include parameters, in which case it defines a function. This kind of definition is syntactic sugar for defining the identifier to be the corresponding lambda expression.

A definition may be recursive; that is, an identifier may be used in its own definition. This feature is normally used to define recursive functions, though it can be used more generally. However, if evaluating the definition causes evaluation of the identifier, a `bottom-error` exception results.

```
four is 2 + 2
plus-one(x) is x + 1
factorial is { n -> if (n == 0) 1 else n * factorial(n-1) }
fib(n) is { if (n < 2) 1 else fib(n-1) + fib(n-2) }
```

`public` The words `public` may be placed before a definition to make it public.

```
public public-id is 1
  private-id is 2
```

`import` Additional environments may be added in the middle of the program, using the `import` command. Any environment value may be imported and it has the effect of adding a new environment in which identifiers are resolved during compilation. The expression that is imported must be a constant identifier. It may evaluate to any of the three possible environment types described in section 3.

Imported environments take precedence over other external environments (including previous imported environments), but not over public or private identifiers defined within the program.

```
env is [["a", 10]]
import env
a == 10
```

An example of a simple program is:

```
is_prime(n) is {
    var i = 3
    loop {
        if (i*i > n) break
        if (n%i == 0) return F
        i = i+2
    }
    return T
}

public list-primes(max) = {
  var i = 3
  while (i <= max) {
    if (is_prime(i)) print i
    i = i+2
  }
}

print "is_prime␣is␣"| is_prime.unparse
print "list-primes␣is␣"| list-primes.unparse

list-primes(200)
```

This program prints all the prime numbers less than 200 and exports the following environment containing just the identifier `list-primes`:

```
[["list-primes",
  {max -> var i = 3;
  loop { if (is_prime(i)) print i;
         i = i+2; if (i>max) break }
  }]]
```

Note that the vertical bar operator `|` represents string concatenation in this context.

5 Statements

The code of a function is a statement (if it starts with an open brace) or an expression, otherwise. Statements are similar in form to C statements, though the separating semicolons are optional. It is considered good form to separate statements on the same line by semicolons but to leave them out at the ends of lines. In rare cases, semicolons may be necessary to avoid ambiguity.

Each statement returns some value, though this value is only accessible if the statement is the last executed by a function, since the value returned by a function is the value of its last-executed statement.

Within parentheses, many statements can be used as expressions; however, new variables can only be introduced inside braces. For example, the `if` statement is particularly useful as an expression:

```
var y = (if (x < 0) -x else x)
```

assignment Assignment is a statement rather than an expression. Assignments may be made to local variables and to properties of objects. Assignments may not be made to identifiers declared in the compilation environment, which are global constants. The value of an assignment is the assigned value.

```
x = a + 2
hair.color = red
i = [i,0]
y.i.j = 'b'
```

block A sequence of statements in braces, optionally separated by semicolons, may be used as a statement.

expression Any valid non-function expression may serve as a statement. The value of the statement is simply the value of the expression. However, the expression cannot start with an open parenthesis or open brace; An open brace indicates the start of a block of statements and an open parenthesis indicates a continuation of the previous statement.

```
f(1,2)
2 * x
person.get(thing)
```

`break` Breaks the flow of control out of the nearest enclosing `for` or `loop` loop. The value of a `break` is `NIL`. It may only be used inside a loop.

```
loop { n = n - 1; if (n == 0) break }
```

`fail` This statement raises an exception. It is similar to `return`, except that it causes the flow of control to return to the nearest dynamically enclosing `except` statement that can handle it.

Two arguments must be supplied to `fail`. The first argument is the exception object, which is used to select the `except` clause to invoke. The second argument is the exception value, which is supplied as an argument to the except clause.

```
fail(lookup-failure, id)
```

`for` The first argument to this statement is a parenthesized expression that is expected to evaluate to a list value. The second argument is a function expression. The function is evaluated once for each item in the list

```
for i in ([1,2,3,4,5]) { check-num(i) }
```

`if` The syntax of this statement is the same as in C. `if` evaluates its first argument, then invokes the second argument (a function expression) if the first argument was true when converted to a boolean. An optional `else` clause will be invoked if the conditional was false.

The value of an `if` is the value of whichever clause gets executed, or `NIL` if none is.

```
if (a==2 && b==3) { c = f(4); }
if (person.hair.color == red)
  person.chroma = red
else
  person.chroma = unknown
```

`loop` Executes an infinite loop. The `break` statement may be used to break out of the loop, just as in C.

```
loop { a = a+1; if (a>=10) break; }
```

`while` Executes a loop with a guard expression, as in C. The `break` statement may be used to break out of the loop.

```
while (a < 10) { a = a+1 }
```

`print` Prints the value of the expression on the debugging connection, if any. Printing has the advantage that it is *not* subject to rollback when transactions fail, so you can see things that happen inside transactions.

```
print "At that point"
```

`return` Causes the nearest lexically enclosing function that was invoked through ordinary function invocation (rather than by one of the built-in statements such as `if` or `loop`), to exit. Unlike C, an expression *must* be supplied as the return value of the function.

```
return a + b
```

`try/except` The `try/except` statement executes a statement and catches exceptions it generates. The `try/except` statement is often used in conjunction with an `atomic` statement (See also Section 7).

A `try/except` has some number of `except` clauses. Each `except` clause matches a specified exception, given in parentheses. It may be a user-defined exception or an exception corresponding to a run-time error, such as `divide-by-zero`. Each `except` clause provides a function taking two arguments. The first argument is the failure value or the error message. The second argument is a list with two items. The first item is the line number of the code where the failure occurred. The second item is the name of the program that the failure occurred in — the name that was provided as the second argument to the `eval` operation when the code was compiled.

A final `except` clause that matches all exceptions may be provided as well, by writing two or three identifiers in place of the parenthesized exception object. These are bound to *three* arguments that are supplied by the source of the failure. The first argument is the exception object, and the second and third arguments respectively are exactly like the first and second arguments in an ordinary except clause.

If no except clause catches the failure, the exception propagates up to the next containing except statement, if any.

```
try a = hair.color
  except e, v in (peroxide) {
     print "Peroxide␣error␣in␣line␣" | i.first
     a = unknown_color
  }
  except e, v, i {
    fail(hair-error,
        i.rest.first | ",␣line␣" | i.first |
    ":␣Unhandled␣failure␣" | e | "␣while␣getting␣hair␣color:␣"| v)
  }
```

atomic Creates a new transaction. The atomic is immediately followed by a statement, which is executed in a new child transaction. If the statement fails (terminates with an exception), all mutations performed by the transaction will be erased. The exception will also propagate through the atomic statement (See also Section 7).

An atomic statement may have some number of attached except clauses, which cannot be wildcard handlers. If one of these clauses handles the exception, the transaction is *not* rolled back.

```
atomic a = b/c
try { atomic { me.look() } }
    except e { me.tell("Failed!") }
```

val The val statement is used to evaluate an arbitrary expression as a statement.

```
val {x->x} == {x->x}
```

var A new local variable may be declared in the current scope at any time with the var declaration. The syntax is similar to C. A list of variables may be supplied, and each variable in the list may be initialized with an expression. An uninitialized variable will always contain NIL initially.

Variables in ALSO are lexically scoped and may be referred to and mutated by lambda expressions within their scope.

A variable may not be declared with the same name as another local variable that is already in scope; local variables do not have holes in scope.

Items in the variable list must be separated by commas, just as in C. The value of a var statement is NIL.

```
f(a,b) is {
      var c = a + b, d
      d = c * c
      var a2 = d
      return(a2)
    }
}
```

6 Expressions

Expressions in ALSO strongly resemble C expressions, although the set of predefined operators is slightly different.

The expression-building operators described below have been sorted into different lists, depending on what types they manipulate.

6.1 Arity

Nullary operators take no arguments.

```
main
```

Unary operators are all prefix operators — they expect one argument, which is placed to the right of the operator.

```
!x
-y
```

Binary operators are infix — they expect two arguments, placed on either side of the operator.

```
1 + 2
x == 5
```

6.2 Precedence

In general, the larger the number of arguments, the lower the precedence of an operator. There are exceptions to this. Object operators have the highest precedence. Comparison operators have very low precedence. The actual precedences are, from top to bottom:

`. ()`
`!`
`* / %`
`+ -`
`< > <= >= != == =>`
`&`
`|`
`with`

6.3 Conversions

The various built-in ALSO operators often expect argument types different from what one has available. Many of the different value types automatically convert themselves to the desired type.

If a value cannot be converted to the desired type, the runtime error `typecheck-error` occurs.

to integer `NIL` converts to 0, `T` converts to 1, `F` converts to 0. Characters convert to their ASCII index, strings convert via the C library function `atoi`. Other types usually do not convert.

to boolean `NIL` converts to `F`, integers other than zero convert to `T` and zero converts to `F`. Lists and strings convert to `T` if and only if they contain any items. All record objects convert to `T`. Other types do not convert.

to string `NIL` converts to an empty string. Characters convert to a string containing just that character. Integers convert to a standard ASCII representation. Other types do not convert.

to list Values other than lists usually convert to a list containing just that one value. NIL converts to an empty list.

6.4 Integer operators

arithmetic (Binary), (Unary)

ALSO supports the usual C arithmetic operators: `+`, `-`, `*`, `/`, `%`. The arguments are converted to integers. The `%` and `/` operator fail with the exception `divide-by-zero` if their right-hand side is zero.

bitwise (Binary)

The operators `&&` and `||` (equivalently, `&`, `|`) are treated as bitwise-and and bitwise-or when the left argument to the operator is an integer.

`rand` (Method)

The `rand` method returns a random number ranging from 0 up to one less than the receiver. For example, `100.rand` gives a number between 0 and 99. The random seed can be specified on the command line when ALSO is started.

`toChar` (Method)

This method converts an integer to the corresponding ASCII character.

```
65.toChar == 'A'
```

Comparisons See below.

6.5 Boolean operators

`!` (Unary)

Takes the logical negation of the value provided.

```
!T == F
```

`&` **or** `&&` (Binary)

Takes the logical conjunction (and) of its boolean arguments.

```
T & F == F
```

`|` **or** `||` (Binary)

The action of this operator depends on the type of the left operand. If boolean, it performs a logical or.

```
T | F == T
```

Comparisons See below.

6.6 List operators

`[]` (Special)

A bracket expression produces a list value. The expression contains a sequence of expressions, separated by commas. The expressions are evaluated, and a list value is constructed from the results.

```
[1,2,3]
[]
[["Hello"],'y'],{}]
```

`first` (Method)

Returns the first item of the receiving list value. If the value is not a list, the value is returned. If the list value is an empty list, NIL is returned.

```
[1,2,3].first == 1
 [].first == NIL
```

`rest` (Method)

Returns a list containing all the items from the list receiver, except the first item. Returns `[]` if the list is empty.

```
[1,2,3].rest == [2,3]
 [].rest == []
```

`length` (Method)

Yields the number of items in the list.

```
['1',2].length == 2
 [].length == 0
```

`reverse` (Method)

Returns a list that is the list handed it in reverse order.

```
[1,2,"hello"].reverse == ["hello",2,1]
```

`cons` (Method)

Constructs a new list out of a value and an existing list. The new list will have the value as its head, and the existing list as its tail.

```
[2,3,4,5].cons(1) == [1,2,3,4,5]
 [].cons({x -> x + x}) == [{x -> x + x}]
```

`contains` (Method)

Returns true if an element is contained in the list.

```
[1,2,"hello"].contains("hello") == T
[1,2,"hello"].contains(0) == F
```

6.7 String operators

`|` **or** `||` (Binary)

Concatenates two string arguments, returning a new string with the arguments spliced together.

```
"Hello " | "there" == "Hello there"
"he" | 'y' == "hey"
```

All values may be used on the right-hand side of this operator, and are converted to strings using the same mechanism as when printing them.

`length` (Method)

Yields the number of characters in the string.

```
"ab".length == 2
"".length == 0
```

`unparse` (Method)

Generates a string representation of the argument value. Functions yield an approximation of the code that was used to compile them. Record objects do not support this method by default, but one can be defined for all record objects by attaching a method definition to `Object`. Lists produce a pseudo-list-expression. If the list is too long, ellipses (...) will be used.

An optional integer argument may be provided to control the depth to which subexpressions are reported. This feature is primarily useful for lists.

```
1.unparse == "1"
'a'.unparse == "'a'"
T.unparse == "T"
NIL.unparse == "NIL"
[[1,2],3].unparse == "[[1,2],3]"
[[1,2],3].unparse(1) == "[<list>,3]"
val {x,y -> loop {x = x - y; if (x<=0) break} x}.unparse ==
"{x,y->␣loop␣{
```

```
␣␣␣␣␣x␣=␣x␣-␣y;
␣␣␣␣␣if␣(x␣<=␣0)␣break␣};
␣␣␣x}"
```

`at` (Method)

Returns the character located at a specified position in a string. If the position is out of range for the string, a `range-error` exception is raised.

```
"andru".at(0)== 'a'
```

If invoked with two arguments, it extracts a subsequence of the string and returns it. The first argument specifies the index of the first character in the subsequence. The second integer specifies the length of the subsequence. If the requested subsequence is too long to fit within the string, a correspondingly shorter string is returned.

```
"Hello".at(1,3) == "ell"
"Hello".at(4,3) == "o"
```

The time required to perform this operation increases with the number of characters that must be traversed from the beginning of the string.

`search` (Method)

This operator searches in the receiver string for the first occurrence of a smaller string past a certain index. The smaller string is the first argument, and the index is the second. The returned value is the index at which the smaller string was located, or the length of the larger string if it was not. If the second argument is omitted, it defaults to zero.

```
"test test test".search("test", 7) == 10
"Hello there".search("here") == 7
```

`replace` (Method)

Globally substitutes all occurrences of one string for another in a larger string, returning the result. No wildcards are supported. The first argument is the string in which substitution is done. The second string argument is a pattern to substitute for. The third string is the string that replaces every occurrence of the pattern. If the pattern is the empty string (`""`), no substitution is performed.

```
"XXXla".replace("X", "ho") == "hohohola"
```

`split` (Method)

Breaks a string into two pieces around any character in a *delimiter set*. The left argument is the string to be broken; the right argument is a string whose characters are the delimiter set. `split` finds the first sequence of non-delimiter characters in the string, and the sequence of characters that follow after skipping over any immediately following delimiters. It returns a two-item list. The first list item is the token at the beginning of the string; the second list item is the rest of the string.

```
" andru is author." split(" x") ==
                    ["andru","is author "]
```

`lowercase` (Method)

Returns an all-lowercase version of the same string.

```
"Hello There".lowercase == "hello there"
```

`sha1` (Method)

Returns a SHA-1 hash of the string.

```
"hello".sha1 == "aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d"
```

`substr` (Method)

Returns the substring located at a specified position in a string. If the position is out of range for the string, an empty string is returned. It may be invoked with one or two arguments. With one argument it returns the substring starting from that position. It can be expected that `substr` is especially efficient for small arguments; using this method is the recommended way to iterate through a string.

```
"andru".substr(2) == "dru"
```

If invoked with two arguments, `substr` acts just like `at`.

`eval` (Method)

This operator evaluates a string as an ALSO program. Up to four additional arguments may be provided. The first is the name of the program, and is placed in any resulting error messages. The second argument provides an environment in which undefined variables are evaluated. See section 3 for more information on environments. The third argument is a boolean specifying whether the program should be evaluated in interactive mode, so that identifier bindings are affected by previously executed top-level statements. The fourth argument is a boolean specifying whether definitions should be automatically be considered public, and hence exported from the program.

The program is compiled at the top lexical environment, so it may not reference any local variables that it does not define.

If a syntactic error is encountered while evaluating the string, the operator fails with the exception `parse-error`. The failure value will be an appropriate error message.

The result of a successful evaluation is a list. The first item in the list is the value returned by the last statement executed in the program. The second item in the list is a list environment containing all public definitions made in the program.

Interactive mode affects how evaluation and parsing are performed. In non-interactive mode, `eval` parses the entire program at once, then executes it. In interactive mode, the program is parsed and executed one statement at a time. During an invocation of `eval`, it cannot be performed, else a `nested-eval` failure occurs. Note that in interactive mode, `eval` may partially execute a program, then encounter a syntax error and fail with `parse-error`.

```
"2 + 3".eval
     ("simple-add", []) == [5, []]
"a is 10; 45".eval
     ("simple-defn", []) == [45, []]
"public a is 1 + b".eval
     ("defn-with-env", [["b", 2]]) ==
                    [3, [["a", 3]]]
"a is 1 + b".eval
     ("defn-with-env2", [["b", 2]]) == [3, []] "a is 1 + b; a*2".eval
     ("defn-with-env2", [["b", 2]], F, T) == [6, [["a", 3]]]
```

Comparisons See below.

6.8 Function operators

Invocation (Special)

Functions are invoked similarly to C. Arguments to the function are parenthesized and separated by commas.

```
f(1,2)
g()
proc-array(a,b,c)
```

If too many arguments are provided, the extra argument values are dropped. If not enough arguments are supplied, argument variables not provided values are initialized to `NIL`.

`apply` (Method)

Invokes a function, passing the elements of a supplied list as the arguments to the function. The result is the result of invoking the function.

```
val {x,y->x*y}.apply([4,5]) == 20
```

`map` (Method)

This method is applied to a function expecting n arguments. The arguments are n lists. The function is invoked once for each item in the lists, receiving the corresponding elements from all n lists at once. The result of a `map` is a list containing the return values of all the function invocations.

If any of the right-side arguments are not lists, then they are supplied as the corresponding argument to *all* of the invocations of the function. If the list arguments on the right-hand side are of different length, the function will be called a number of times that is equal to the length of the shortest lists. If none of the right-side arguments are lists, `map` invokes the function once.

```
{x,y,z->x*y+z}.map ([2,3],[5,7],1) == [11,22]
```

`defns` (Method)

Extracts the set of global environment definitions used by a function, and returns a list environment containing them. This means that for any function `f` at the top lexical level, the result of `f.unparse.eval ("foo", f.defns)` will be a function equivalent to `f` itself.

```
a is 10
{x->x+a}.defns == [["a",10]]
{x->x + x}.defns == []
```

6.9 Connection operators

`make` (Method)

If socket creation is enabled via the command-line option `-x socket`, then `Connection.make` can be used to create TCP sockets. It expects two arguments: the DNS name of the host to connect to, and the TCP port number. It establishes a TCP connection to that port on the specified host, and returns a new connection object which can be used to talk to the system on the other end of the connection. The dispatch function of the new connection will be the default one, and should be modified immediately.

If a connection cannot be established, the failure `e_external` happens, with an appropriate operating-system error message.

For example, we can write a trivial version of the program `telnet` as follows:

```
conn is {
    var c = Connection.make(argv.first, argv.rest.first + 0);
    c.dispatch({c, s ->
        if (s == NIL)
            System.shutdown("Connection closed.")
        else
            io.enqueue(s)
    })
    c
}()

login is new()
dispatch is new()
login.dispatch(c,s) = conn.enqueue(s)
```

`enqueue` (Method)

Enqueues a value onto the output of a connection. Output sent to a connection is transacted appropriately. Values such as strings and characters are sent in the obvious way; other values will be converted into an appropriate string in an implementation-dependent fashion.

The left argument is a connection; the right argument is the value to be enqueued.

```
c.enqueue("Hello\n")
```

`dispatch` (Method)

Changes the dispatch function of a connection. The dispatch function of a connection is invoked for each piece of data that arrives on the connection. The connection and the received data (a string) are then passed to the new dispatch function whenever data subsequently arrives. The initial value of the dispatch function is determined by the value `login.dispatch` at the time that the connection is created.

`info` (Method)

Given a connection value, this method returns information about the connection. The information is returned in a list, where the first item in the list is the string name of the remote host that has connected to the system; the second item is the total number of seconds that the connection has existed; and the third item is the number of seconds that the connection has been idle.

`close` (Method)

Closes a connection. After the connection is closed, it is useless.

6.10 Record object operators

execute (Special)

A property of a record object may be invoked using the unadorned "." operator. Optionally, arguments for invocation may be supplied after the property, like a function call.

If the value of the property is a function, the object containing the property is implicitly passed as the first ("self") argument to the function. The optionally supplied arguments are passed as the second and succeeding arguments to the function. If the value of the property is not a function, the value of the property will be returned when the property is executed, and arguments, if any, will be ignored.

```
o.p
o.p(q r s)
var c = a.b(d)
```

read (Special)

The value of a property may be read without invoking functions by using a dollar sign instead of a dot:

```
v = o$p
var c = a$b
```

If no value has been assigned to a property of an object (or to the parent(s) from which it inherits), the result of reading that property will be a `no-such-property` exception.

write (Special)

The value of a property may be modified by assigning to the property. This is actually a statement rather than an expression. Formal parameters may be used on the left-hand side to define a function value.

```
o.p = v
a.b = {x->x*x}
fact.f(x) =  if (x == 0) 1 else x*fact.f(x-1)
```

query (Binary)

Some properties are *invertible*. Invertible properties can only be assigned values that are record objects. Querying yields a list of those objects from which, if the designated property were read, the specified value would result.

```
[o] == p?v
```

`new` (Special)

Produces a new object with no parent object. The right side of the operator is a parenthesized *modifier list*, which specifies new values for properties that override the values from the old object.

The modifier list consists of a series of modifiers of the form `p = v`. The expression to the left of the equal sign should evaluate to a valid property key. The right side should evaluate to a value. Together, the property key and the property value specify a property to be attached to the new object, possibly replacing one of the inherited properties. Modifiers may be optionally separated by commas.

`with` (Special)

The `with` operator takes an existing object and produces a new object that inherits all of its properties. A modifier list is used to override existing properties or to supply new ones.

```
hair is new (color=brown, preferred=F, length=6)
blond-hair is hair with (color=blond, preferred=T)
blond-hair.color == blond blond-hair.length == 6
```

6.11 Comparisons

ALSO supports the six usual comparison operators: `>`, `<`, `>=`, `<=`. They work only on characters, strings, and integers. Operators `==` and `!=` work on any values.

Two values are equal in ALSO if no operation applied to the values results in discernably different behavior. Equality is defined on integers, characters, and strings in the obvious way. Two lists are defined to be equal if all of their elements are equal. Two objects are equal if they are the same object. Two functions are equal only if they are the result of evaluating the same function expression in code produced by the same invocation of `eval`. This rule is appropriate since failure results encapsulate information about the compilation that produced the failure.

Comparing two values will never result in failure. Values of different types are incomparable, and comparing them with an ordering operator (`<`, `>`, `<=`, `>=`) will always produce `F`.

```
2 < 3 == T
"andru" < "mrx" == T
'Z' <= 'a' == T
apple != orange == T
'a' < "b" == F
(""|'a') < "b" == T
3 < {x->x} == F
{x->x+x} != {x->2*x} == T
{x->x+x} != {x->x+x} == T
var a = {x->x+x}; (!(a != a) && (a == a)) == T
```

The binary operator => compares the level of trust of two values when interpreted as authority. See Section 8 for more detail.

6.12 System operators

`System.exec` (Method)

This method is available only if the option `-x exec` is provided at the command line. It executes a command in the local operating system. It expects an argument list as the first argument, and an environment as the second. The argument list must be represented as a list of strings, and the environment must be a list of lists containing two strings. The first string is the name of an environment variable, and the second string is the value of that variable.

The value returned by the `exec` operator is a connection that is connected to the standard input and output of the exec'ed program. Output generated by the program is sent to the dispatch function of the connection, and output enqueued on the connection arrives at the standard input of the program.

When the `exec`'ed program terminates, the connection's dispatch function is called with a second argument of `NIL`, and a third argument which is equal to the exit status of the program as returned by *wait*(2).

```
(System.exec(["ls", "-C"], environ).dispatch {c,s -> io.enqueue(s)}
```

`System.shutdown` (Method)

This method of the `System` object causes the current ALSO universe to shut down, closing all connections, and save its state persistently. The argument to `shutdown` is sent as output to all open connections before they are closed. This operator has no effect if executed during the bootstrap program.

`System.time` (Method)

This method of the `System` object reports the current time in seconds since January 1, 1970, or if that value is not less than 2^{31}, then that value minus 2^{32}.

`System.transaction-stats` (Method)

This method of the `System` object reports statistics about the current tranaction. Currently the reported statistics are simply the number of updates performed in the current transaction. This is useful for aborting tranactions that are causing too many updates or creating too many objects.

6.13 Miscellaneous operators

`main` (Nullary)

Returns the value of the environment that was returned by the bootstrap program. (See section 9).

```
Main is main;
Main::list-primes(200)
```

`authority` (Nullary)

The current effective authority of the program. (See section 8).

`caller` (Nullary)

The effective authority of the caller of the current executing function. (See section 8).

`arguments` (Nullary)

A list containing the arguments passed to the currently executing function.

`typestring` (Unary)

Returns the type of the argument value. The type is returned as a string object; the possible return values are "bool", "int", "char", "string", "list", "array", "record", "function", and "connection".

```
typestring {} == "function"
typestring 'a' == "char"
typestring new() == "record"
```

7 Transactions and Exceptions

All operations performed by ALSO take place inside a *transaction*. Transactions may either succeed or fail. If a transaction fails, no mutation performed by the transaction will be visible to the environment that began the transaction – except for the exception returned by the transaction, it will be as if the transaction never occurred.

Transactions may contain other transactions. The failure of a nested transaction does not imply that the containing transaction will also fail. However, if a containing transaction fails, transactions nested inside *it* will also fail in the sense that their mutations will be erased.

A transaction is created by use of the `atomic` statement. `atomic` creates a new nested transaction and tries to execute the statement in that transaction. If the statement generates an exception that is not caught by an `except` clause, the transaction will fail. The exception will propagate up into the containing transaction.

Except clauses may be attached to the `atomic` statement itself, allowing the statement to generate exceptions while also committing. This behavior is equivalent to nesting a `try/except` immediately inside the `atomic` and then rethrowing the same exceptions after the transaction commits.

Failure of a transaction may be caused by a run-time error (*e.g.* division by zero) or by an explicit `fail` statement executed in that transaction. Run-time errors and user-initiated failures are handled in exactly the same way. When an unhandled failure occurs, mutations performed by the transaction are erased and the transaction is terminated, and the flow of control is immediately transferred back to the nearest dynamically enclosing `try/except` statement.

Updates to objects created during a transaction are not rolled back if that transaction fails, so it is possible to construct complex data structures that are returned as exception results.

The following example will print the message "Division by zero occurred", and the condition `a == 1` will be true after it executes, since the assignment `a = 0` is a mutation that takes place inside the failed transaction.

```
var a = 1;
try atomic { a = 0; a = 10/a }
except
  when (divide-by-zero) {
      print "Division by zero occurred"
  }
```

Every exception, whether generated by a run-time error, or by an explicit `fail` statement, has an associated *exception object*. The exception object defines the kind of failure that occurred. Run-time errors such division by zero or run-time typecheck errors have pre-defined well-known exception objects: `divide-by-zero` and `typecheck-error`. User exceptions are defined by simply creating new objects, and are treated no differently from the built-in exceptions.

The `fail` statement generates an exception. The first argument to a `fail` is the exception object, which may be a pre-defined exception object or a user-defined exception object. The second argument to a `fail` is the *failure result*, which is returned to the appropriate `except` clause. The failure result may be an object or other mutable value created during the contained transaction.

```
f(n) is {
  if (typestring n != "int")
    fail(typecheck-error,
         "Sorry, an integer is required for n")
}
```

Other than from the bootstrap program, top-level transactions are created when input arrives on a connection. The connection's dispatch function is invoked inside a new top-level transaction. If the transaction fails, an appropriate error message is sent to the connection.

When a statement of the bootstrap program generates an exception, the system halts and prints an error message that contains the exception value.

8 Access control

ALSO has a built-in access-control mechanism that mediates access to record objects. At any given point during execution, there is an *effective authority* that determines the privileges currently possessed by the execution. The current effective authority can be obtained through the expression `authority`. Whether certain actions are permitted is governed by whether this authority suffices to perform the action in question. When there is insufficient authority to perform an action, an `access-error` exception occurs.

In addition to the effective authority at a given point during execution, there is also a current *maximum authority*, which corresponds to how trusted the running code is. In general, the effective authority of code is less than the maximum authority, but code can elect to increase the effective authority up to the maximum authority. This must be done with care to avoid confused-deputy attacks.

8.1 Using authority

Operations on properties are mediated by the properties themselves, based on authority. Properties that are not records have no access controls; they are always permitted. Record properties can impose controls on

reading, writing, and invoking them. An invocation of a property `o.p` using authority `a` is permitted if the expression `p.may-invoke(a, o)` evaluates to true. Similarly, a read of a property `o$p` is checked using `p.may-read(a, o)`, and a write to the property is checked using `p.may-write(a, o)`.

If one of these methods is absent from the property, then the corresponding access is implicitly authorized. In addition, the access is authorized if the method property object trusts the current authority as described below.

8.2 Authority

Authority is simply a value in the language, but the language attaches special significance to some values when used as authority. Record objects are a typical way to represent authority; in particular, when the authority is a record object, that authority controls access to the corresponding property.

Hence, access to the different properties of an object is determined in a fine-grained way that may be distinct for each property. In a typical object-oriented language, classes would determine the access restrictions to properties, either through public/private visibility modifiers or through more explicit security mechanisms like the Java stack inspection mechanism. ALSO does not have classes, and properties serve these roles instead, in a unified way.

Authorities are ordered. Higher authority allows more actions to be performed. When authority a is at least as powerful and trusted as authority b, we say that b *trusts* a. Whether this relationship exists can be tested using the expression a `=>` b. For some values a and b, the ordering of authority is hard-coded into the language; however, for record objects this ordering is controlled by the program.

- Record objects are the normal way to represent authority. By default, record objects trust only themselves (and, like every other value, they trust the empty list).

  A record may extend the trust ordering by defining a `trusts` method. When an access control check requires determining whether record `o` trusts value `v`, the expression `o.trusts(v)` is evaluated. If this expression successfully returns `T`, the trust relationship holds. Any other outcome, including an exception, means the relationship does not hold.

- A list of values represents the greatest lower bound (or join or disjunction) of the authorities represents by its elements. In other words, the authority `[A,B]` represents the greatest authority that trusts both `A` and `B`. For example, the authority `[1,2]` is equivalent to the authority `2`. There is no built-in way to construct the conjunction of authorities.

  The empty list represents the greatest possible authority, which is the authority of a program when it starts. Every value trusts the empty list: `[]` `=>` v for all values v.

- The authority `NIL` is the minimal authority. It trusts all other authority: v `=>` `NIL` for all values v.

- Integers used as authority are totally ordered according to ordinary integer comparison ($\leq$), so that `0` is more trusted than `1` and `-1` is more trusted than `0`. This ordering allows integers to be used, if desired, in a manner analogous to processor rings.

- Booleans used as authority are ordered by implication, so that `F` is strictly more trusted than `T` (`F => T`).

- Connections used as authority are not ordered; a connection trusts only itself.

Method calls. Authority changes, in general, at entry to method calls, and is restored on return. The authority of the caller can be accessed using the special expression `caller`. How it changes depends on the property object that is being invoked. In the ordinary case, the effective authority on entry to the method call becomes the join of the authority of the caller and of the property object being invoked. This join prevents the caller's authority from being exploited by the method and prevents the method from acting as a confused deputy. However, it may prevent access to needed resources. Hence, the maximum authority is set on a method call to the authority of the property, and the `as` statement can be used in the method code to increase authority up to that level.

Calls via a property object that endorses work differently. The call automatically sets effective authority to be the same as maximum authority; that is, the authority of the property object. Implementation of methods for endorsing properties must be done with care, since any caller can obtain the full power of the property object.

Note that ordinary function calls do not have any effect on authority. Hence, whereas a call of the form `x.f()` would increase maximum authority during the call to `f`, a call of the form `(x$f)(x)` would run the same code with the same argument but the maximum authority within the call would be the same as in the caller.

8.3 Changing authority

Effective and maximum authority can also be changed by two explicit mechanisms: `as` and `become`. Both of these mechanisms interact with another aspect of the access control system: each stack frame may have a current *locked authority*. By default, the locked authority is `NIL` (there is no lock), but the locked authority can be increased by the use of `as`. Authority is locked to prevent *reentrancy attacks* in which untrusted code calls back into trusted code that has called it.

`as`. The statement `as (a) s` performs a statement `s` with the authority `a`. The authority `a` must trust the current maximum authority. There is also a reentrancy check: the current authority must flow to the join of `a` with all locked authorities. If both of these checks succeed, the authority `a` is locked at the current stack frame, and the current effective authority becomes `a`.

`become`. The statement `become a` changes the current authority to be authority `a`. The authority `a` must trust the current effective authority. The locked authority of the current stack frame is joined with `a`, relaxing the condition checked by the `as` statement.

9 Executing ALSO

ALSO starts by loading a bootstrap program from either standard input or a file specified with the `-f` option. The program is evaluated using the `eval` operator in interactive mode, with the program name set to either `"standard input"` or the name of the bootstrap file. The bootstrap program is executed in debugging mode.

A special bootstrap environment is used to compile the bootstrap program. This environment contains definitions for identifiers `environ` and `argv`. `environ` is a list environment containing definitions for all environment variables. For example, the expression `environ::HOME` contains the value of the `HOME` environment variable. The identifier `argv` is bound to a list containing all command-line arguments not

parsed by the `also` program itself. There is no binding for `argc`, but `argv.length` suffices. The name `io` is bound to the console connection — that is, standard input and output.

After the bootstrap program completes successfully, the returned environment can be thereafter retrieved using the `main` operator. This environment also serves as a root for garbage collection purposes. If the bootstrap program fails, an appropriate error message is printed, and the system halts.

This structure makes it easy to have small sessions with raw ALSO (where the bootstrap file is standard input) and to write small "shell scripts" to be executed.

ALSO enters multi-user mode after the bootstrap program ends if the bootstrap program defines the well-known objects `login` and `dispatch` and sets the property `login.dispatch` to a function value. In multi-user mode, the system will accept new TCP connections from the outside world,

New connections are created with their dispatch function equal to the current value of `login$dispatch`. Input arriving on these connections is then dispatched through that dispatch function, with the first argument equal to the connection object providing the input, and the second argument equal to a string containing the data that has arrived. When a new connection is created, the message `login.value`, if defined, is dispatched to the connection.

Once multi-user mode is entered, ALSO begins saving the state of the world in a database. This database can then be used to restart ALSO in the same configuration at any future time, using the `-r` option.

During multi-user mode, ALSO uses the well-known `heartbeat` object to initiate periodic actions. The `heartbeat$dispatch` function, if any, is invoked at regular intervals. The default interval is 5 seconds, but if `heartbeat$value` is defined, it specifies the heartbeat interval in milliseconds.

Ordinarily, ALSO creates new connection objects to service users when input arrives on standard input, or when a connection is made to the connection socket. By default, the socket port is 4201, although the `-p` option may be used to change it. Standard input is the console connection. If the `-C` option is supplied, ALSO runs in the background, with no console connection.

10 Well-Known Objects

Certain objects are called "well-known" objects because they are specially recognized by the ALSO language system. These objects do not have to exist; but once created, they become well-known and assume their function as described below.

Creation of a well-known object is accomplished by defining an identifier of the appropriate name in the bootstrap program, and giving it a value which is an object. This causes the object to be registered as the well-known object named.

The list of well-known objects and their purposes, at this time of this writing, is as follows. In addition to the objects in this list, all prototype objects and built-in methods described earlier are also well-known objects.

`invertible` determines invertibility of a property. Must be `T` or `F`.

`immutable` determines mutability of a property. Must be `T` or `F`. The evaluator automatically sets `immutable.immutable = T` when this property is created. If a property is immutable, it cannot be written.

`parent` The property containing an object's parent object, from which it inherits properties.

`endorse` controls whether a property endorses execution with its own authority when invoked as a method. Must be `T` or `F`.

`lookup` The property that is executed to look up unresolved identifiers in an object environment.

`mayInvoke` controls whether a property may be invoked as a method.

`mayRead` controls whether a property may be read.

`mayWrite` controls whether a property may be written.

`value` A property that is used to attach information to the `login` and `heartbeat` objects.

`dispatch` A property that is used to attach information to the well-known `login` and `heartbeat` objects.

`properties` The property that contains the list of properties of an object.

`login` This object contains information about how to start up the multi-user environment. See section 9 for more details.

`heartbeat` This object controls periodic behavior of the ALSO system. See section 9 for more details.

`could-not-open` Exception generated when ALSO is unable to open a file.

`divide-by-zero` Exception generated when ALSO tries to use zero as a divisor or modulus.

`typecheck-error` Exception generated when an operator or statement encounters an argument of an unexpected type.

`immutable-property` Exception generated when an assignment is performed to an immutable property.

`not-implemented` Exception generated when an unimplemented language feature is used. Nothing described in this document should result in this failure.

`overflow` Exception generated when an arithmetic overflow occurs.

`parse-error` Exception generated when the compiler is unable to parse a string.

`range-error` Exception generated when a string or array index is out of range.

`nested-eval` Exception generated when `eval` is invoked during an `eval`.

`bottom-error` Exception generated when a loop iterates more than some maximum number of times (default: 100,000), or the function stack grows to more than some maximum depth (default: 300).

Acknowledgments

My thanks to the many people who have helped to improve this document and have been willing to discuss MUDs, programming languages, and other related topics. They include especially Jim O'Toole, Robert Reimann, Dave Ciemiewicz, Franklyn Turbak, and Vivek Myers.

References

[ACC82] Malcolm Atkinson, Ken Chisholm, and Paul Cockshott. PS-Algol: An Algol with a persistent heap. *SIGPLAN Notices*, 17(7):24–31, July 1982.

[LS83] B. Liskov and R. W. Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Trans. on Programming Languages and Systems*, 5(3):381–404, July 1983.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. In *2nd ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 227–241, Orlando, FL, October 1987. Published as *SIGPLAN Notices* 22(12), December, 1987. Also published in *Lisp and Symbolic Computation* 4(3), Kluwer Academic Publishers, June, 1991.