lispy Language Manual

Contents

1 Introduction

lispy is a compiled, statically typed, lexically scoped, and type-inferred LISP-like language. Rather than being based on the more modern and complex dialects of LISP, *lispy* is based on LISP 1.5, as described in *LISP 1.5 Programmer's Manual* (McCarthy et al., 1985) and *LISP 1.5 Primer* (Weismann, 1967). However, it has significant adjustments and deviations from LISP 1.5. This helps the language stay small and manageable, but a lot of the differences also stem from the need to make the language statically typed rather than dynamically typed.

2 S-expressions

The fundamental syntactic element of the language is the *S-expression*. An S-expression is a branching binary tree structure of an indefinite length. In its simplest form, an S-expression is defined as either an atom or an expression `(A . B)` where `A` and `B` are S-expressions. However, *lispy* does not support the dot notation for S-expressions; it only supports the list notation e.g. `(A B)`. This is not equivalent to the previous expression. In the list notation, there is an implicit `nil` as the last element i.e. it is equivalent to `(A . (B . nil))`. This has a significant implication in that it's impossible to represent data like `(A . B)`. In the context of a binary tree, this restriction means that the right leaves are always `nil`. `nil` will be discussed in more detail later.

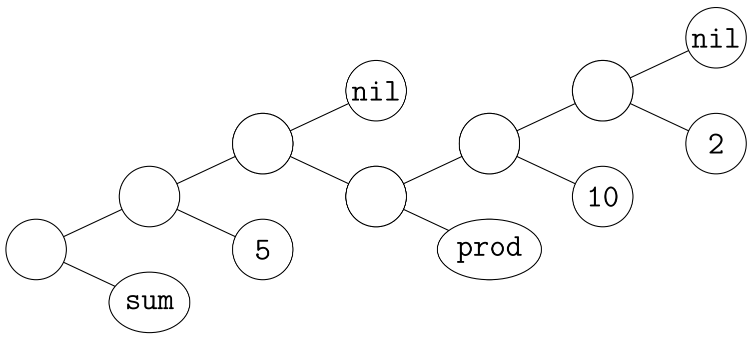

Figure 1: A binary tree representation of `(sum 5 (prod 10 2))`.

An *atom*, also knows as an *atomic symbol* can be either numeric, Boolean, or neither. The latter is known as a *literal atom*, and in some contexts is referred to as a *variable*.

There are some syntactic differences from LISP 1.5. First, commas are not valid delimiters for elements in the list notation. Second, atomic symbols can be a mix of upper/lower-case letters, digits, and underscores. These are case-sensitive. Finally, the names of constants and functions built into the language are all in lowercase.

3 Forms

As in LISP, computation in *lispy* is done by evaluating *forms*. Forms are specific kinds of S-expressions, typically involving specific types and arrangements of elements within lists. All forms have value, and the value of the form is the result of evaluating it. Forms have already been mentioned, such as for the bodies of lambdas, and lambda expressions themselves. In *lispy*, there are four kinds of forms: elementary forms, simple forms, composed forms, and special forms.

3.1 Elementary Forms

3.1.1 Variables

All variables are elementary forms. A *variable* is an atomic literal that is associated with some value. The atomic literal serves as an identifier or name for the value, meaning the atomic literal can be evaluated to get the associated value. Thus, evaluating a variable results in its associated value.

The process of associating a value with an identifier is called *(name) binding*. A pair of a bound identifier and its value is called a *binding*. An identifier can be re-bound i.e. *assigned* a new value. There is further discussion in Binding.

3.1.2 Constants

All constants are elementary forms. All numbers are constants. Built-in values such as the Booleans `true` and `false` are also constants. The result of evaluating a constant is simply that constant itself. For example, the value of `2.4` is 2.4.

3.2 Simple Forms

Simple forms consist of a left parenthesis, a function name, 0 or more function arguments, and a right parenthesis. A simple form is used to evaluate a function.

A *function* is a named form that can be re-used in evaluations with different sets of inputs. The form is known as the *body* and the inputs are known as *arguments*. The function is defined with variables known as *parameters*, which are accessible to the body. When a function is *called*, or evaluated, the arguments are bound to the parameters, and the body is evaluated using the bound parameters. The result of evaluation is the *return value*.

$$(\underbrace{\texttt{sum}}_{\text{name}}\ \underbrace{\texttt{1 2 3 4}}_{\text{arguments}})$$

Figure 2: A simple form.

The function name is either a built-in function name or any variable whose value has a type of `func`. The arguments are 0 or more variables or constants.

3.2.1 Evaluation

1. The function name is evaluated to a built-in function or a `func` value.
2. All arguments are evaluated from left to right.
3. The function is called with the evaluated arguments.
4. The value of the simple form is the value of the function called with the arguments.

3.3 Composed Forms

Composition of forms is possible, which allows for more complex programs. A *composed form* is a more generalised version of a simple form. Unlike simple forms, each argument can be *any* form (elementary, simple, composed, or special).

$$(\underbrace{\texttt{prod}}_{\text{function name}}\ \underbrace{\texttt{(div 4 2)}\ \underbrace{\texttt{(div 6 2)}}_{\text{simple form}}}_{\text{arguments}})$$

Figure 3: A composed form which evaluates to $(4 \div 2) \times (6 \div 2) = 2 \times 3 = 6$.

3.3.1 Evaluation

1. The function name is evaluated to a built-in function or a `func` value.
   (a) If the argument is a constant, the constant itself is the value of the argument.

(b) If the argument is a variable, the associated value is the value of the variable.

(c) If the argument is a simple form, it is evaluated using the previously described process for simple forms.

(d) If the argument is a composed form, the partially evaluated arguments are temporarily saved, and steps 1 to 4 are applied recursively to that composed form.

2. The function is called with the evaluated arguments.

3. The value of the composed form is the value of the function called with the arguments.

However, this is not the complete description of composed forms. The function name can actually be any form which evaluates to a function. Thus, it is more appropriate to call it an expression than a name. This will be shown when discussing the func type.

3.4 Special Forms

The language has built-in functions that are available to the programmer. Some superficially appear to be like most functions, but are actually treated differently. These functions are known as *special forms*. Special forms differ from regular functions in how they're evaluated or how they're defined. They can generally be categorised into one or more of the following:

1. Special forms with an indefinite number of arguments.

2. Special forms that don't evaluate some or all of their arguments.

3. Special forms that allow an argument or return value to be more than one type.

Note that these kinds of qualities are not achievable with user-defined functions; special forms are implemented using internal facilities not available to the programmer.

Built-in functions are all accessible through variables. The variables for special forms cannot be assigned new values.

4 Top-level Program

The top level may have zero or more S-expressions with any amount of whitespace between them. However, consecutive atoms must be separated by at least one whitespace character.

As an example, given the LISP 1.5 program `MAX (1 2)`, the equivalent in *lispy* is `(max 1 2)`. For those familiar with LISP 1.5, *lispy* effectively uses `EVAL` at the top level rather than `EVALQUOTE`, which makes it more akin to the modern LISP dialects.

5 Types

lispy is statically typed. This means type safety (i.e. that there are no discrepancies between expected types of values and actual types of values) is verified at compile time by analysing the source code. The language has a small set of types and a limited (but simple) type system. The language has type inference for everything except function parameters, meaning it can determine the type of a value at compile time without relying on being explicitly told the type by the programmer.

5.1 int and float

There are both integer and rational numbers. The former are `int`s and the latter are `float`s (a fixed-point representation). The syntax for them is a bit laxer than it is in LISP 1.5 (see Syntax). However, octal number literals are unsupported. There are also `float` constants for `inf` (floating-point positive infinity) and `nan` ("not a number").

The behaviour of precision and overflow is implementation-defined. For the reference implementation, these types are implemented using Python's `int` and `float` types, so they are subject to the same limitations in *lispy* as they are in Python. That further depends on the Python interpreter used to run the compiled code.

5.2 bool

The `bool` type is represented by the literal atoms `true` and `false`. This is a replacement for the `T` and `NIL` used by predicates in LISP 1.5. Having a single Boolean type makes it simple from a type checking perspective to implement predicates in *lispy*.

5.3 func

A `func` is a function as described in Simple Forms. The fundamental way to create a function is with a special form know as a *lambda expression*. The resulting function is called a *lambda function*. The distinction from just *function* is that a lambda function is not bound to any name. Thus, to evaluate a lambda function, a composed form has to be used. In fact, the use of a lambda expression in place of a function name is what was being alluded to briefly in the description of Composed Forms. As with function names, the lambda expression is evaluated first in the form, and then the arguments are evaluated.

$$\texttt{(}\underbrace{\texttt{(lambda}\ \underbrace{\texttt{((}\underbrace{\texttt{x}}_{\text{atomic literal}}\ \texttt{int) (y}\ \underbrace{\texttt{int}}_{\text{type}}\texttt{))}}_{\text{parameters}}\ \underbrace{\texttt{(div x y)}}_{\text{body}}\texttt{)}}_{\text{lambda expression}}\ \underbrace{\texttt{1 2}}_{\text{arguments}}\texttt{)}$$

Figure 4: A lambda function being evaluated.

A function's evaluated value (the *return value*) is the result of evaluating the function body with the function variables, which are bound to some values (the *arguments*) prior to evaluation.

Function parameters are defined as pairs, of which there can be 0 or more. The first element of the pair is the name of the parameter. The second element is the type of that parameter. All parameters must be specified with their types; they are never type-inferred. Note that neither element of this pair is evaluated (this is possible because a lambda expression is a special form).

Unlike parameters, the return value and arguments *will* be type inferred. In fact, there is no mechanism in the language for explicitly specifying the type of a return value or argument.

5.3.1 First-class Functions

Functions are *first-class citizens* in the language. This means that, like other values, a function can be passed as an argument to another function, can be returned from other functions, and can be assigned to variables. As a consequence, *higher-order functions* are supported, which are functions that takes a function as an argument and/or return a function.

```
(lambda ((f (func (int int) int))
         (i int))
  (f 3 i)
)
```

Figure 5: A higher-order function that takes a function `f` as an argument. `f` has two `int` parameters and an `int` return value.

Special Forms Unlike regular functions, special forms are not first-class citizens. However, this can be worked around by defining a lambda function which "wraps" the special form.

```
((lambda ((f (func (int int) int))
         (i int))
  (f 3 i))
(lambda ((a int) (b int)) (prod a b)) 2)
```

Figure 6: A lambda function wrapping a special form and being passed to another function. This is effectively doing $3 \times 2 = 6$.

5.3.2 Named Functions

Lambda functions are not inherently bound to any variable. However, it can be quite useful to define a function so it can be re-used later. Since lambda functions are first-class citizens, it is valid to assign a lambda expression to a variable.

One way to do this has actually already been shown: rely on the binding mechanism of a lambda expression. A lambda can be bound to another lambda's parameters, and then the lambda will be accessible via a variable within the other lambda. However, this is tedious. A more convenient way to define functions with names is to use the `let` special form. This will be discussed in Binding.

5.4 list

The `list` type stores a homogenous (i.e. all of the same type) sequence of elements. Lists always have `nil` as the final element.

Lists can be created with the special form `cons`. For example, `(cons 1 (2 (3 nil)))` creates the list `(1 2 3)` (the presence of `nil` is implicit). Notice that the second argument must be a list which holds elements that have the same type as the first argument. Alternatively, the special form `list`, which takes an indefinite number of arguments, can create the same list: `(list 1 2 3)`.

The first element of a list can be retrieved using the special form `car`. For example, `(car (list 1 2 3))` returns the value `1`. Conversely, `cdr` can be used to retrieve the remaining list: `(cdr (list 1 2 3))` returns the value `(2 3)`. Both `car` and `cdr` only accept one argument, which must be a `list`.

```
(lambda ((n int) (l (list int)))
  (prod n (car l)))
```

Figure 7: A lambda with a parameter that's a `list` of `int`s.

5.4.1 nil

The literal atom `nil` represents an empty list. Thus, it also has a type of `list`. In fact, an alternative way to represent `nil` is `()`. For `nil`, `car` and `cdr` are unsupported – attempting to do so will result in a runtime error.

`nil` is a built-in "global" variable that's accessible everywhere. However, it's special as far as variables go because it cannot be assigned a new value.

6 Conditional Expressions

A *conditional expression* is a special form that can branch evaluation conditionally on the value of forms. *lispy* has two special forms for conditional expressions: `cond` and `select`.

6.1 cond

`cond` has one or more pairs of a predicate p_i and a form e_i followed by a single form e at the end. A *predicate* is a form that evaluates to a `bool`. All forms $e_1 \dots e_n$ and e must evaluate to the same type.

$$\texttt{(cond (}p_1\ e_1\texttt{) (}p_2\ e_2\texttt{)}\ \dots\ \texttt{(}p_n\ e_n\texttt{)}\ e\texttt{)}$$

Figure 8: The `cond` special form.

From left to right, the predicate of each pair is evaluated. If p_i evaluates to `true`, then e_i is evaluated and returned from `cond`; no further evaluation is performed. Otherwise, e_i is not evaluated and the evaluation step is repeated using p_{i+1} until p_n. If all predicates are `false`, the form e is evaluated and returned, making it a default value for the conditional expression.

6.2 select

`select` has a form p followed by one or more pairs of forms (p_i e_i) and ends with a form e. All forms p and $p_1 \ldots p_n$ must evaluate to the same type. Furthermore, all forms $e_1 \ldots e_n$ and e must evaluate to the same type.

(select p (p_1 e_1) (p_2 e_2) ... (p_n e_n) e)

Figure 9: The `select` special form.

First, p is evaluated once and its value is temporarily stored. From left to right, p_i is evaluated and compared to the value of p using the built-in predicate `equal`. If this comparison evaluates to `true`, then e_i is evaluated and returned from `select`; no further evaluation is performed. Otherwise, e_i is not evaluated and the evaluation step is repeated used p_{i+1} until p_n. If no equality is found, then e is evaluated and returned, making it a default value for the conditional expression.

Due to the use of `equal`, p must not be a `func` or a `list` which contains a `func` at the top level or in any nested list.

7 Evaluating Multiple Forms

The body of a lambda can be any form, including composed forms. This means the lambda can evaluate multiple forms by continuously nesting lambdas within its body. While this is very useful, it is quite tedious. The special form `progn` solves this by taking 2 or more forms as arguments, evaluating all forms, and returning the value of the last form.

(progn e_1 e_2 ... e_n)

Figure 10: The `progn` special form.

The forms are evaluated left to right, from e_1 to e_n. The return value of `progn` is the value of e_n. Because only the value of e_n is used, each form may evaluate to a different type.

8 Binding

Recall that variables can be bound to values, and existing variables can be assigned new values. This section deals with how these operations are performed in the language.

But first, clarification is needed on what constitutes a valid name for a binding. Generally, a valid name is any atomic literal. This means numbers and the Boolean constants `true` and `false` are not valid names. Special forms and `nil` cannot be assigned new values, nor can they be used as names for new bindings. However, other built-in function names can be used as names for new bindings. This results in *shadowing*, a subject explained in Scope.

8.1 let

A way to bind variables was introduced with lambda expressions. However, as with the problem `progn` solves, writing lambdas for this becomes tedious. Instead, the special form `let` can be used to bind variables. It can be thought of as a more powerful version of `progn`.

(**let** ((a_1 d_1) (a_2 d_2) ... (a_n d_n)) e_1 e_2 ... e_n)

Figure 11: The `let` special form.

`let` has one or more pairs of bindings (a_i d_i) followed by one or more forms $e_1 \ldots e_n$. The forms $e_1 \ldots e_n$, collectively known as the *body*, have the same semantics as in `progn`, except for the benefit of also having access to the variable bindings that precede them. The binding is performed left to right from (a_1 d_1) to (a_n d_n). d_i is the form to evaluate and bind to a_i, which is the name of the variable. Just like in lambda expressions, the variable name a_i is not evaluated. As with `progn`, `let` returns the value of the last form, e_n.

The bindings are performed in parallel, meaning d_i cannot use any of $a_1 \ldots a_n$.

```
(let ((a 2) (b (prod a 2))) b)
```

Figure 12: An invalid use of bindings in `let`; `a` cannot be used for `b`'s definition.

Notice that each variable must have an initial value it's bound to. Because of this requirement, it is possible to infer the type of each variable. Hence, there is no need (nor is there a way) to explicitly specify the type of each variable.

8.2 set

The special form `set` assigns a new value to a variable.

name and *value* are similar to the binding pairs of `let`. *name* must be the name of an existing variable that is currently accessible. The variable is

(`set` *name* *value*)

Figure 13: The `set` special form.

rebound to the new value, and the value must have the same type as value being overwritten. `set` returns *value*.

9 Scope

An important aspect of binding is understanding where that binding is valid i.e. its scope. A binding is said to be *valid* if the binding's name is considered to be associated with the binding's value in the current part of the program. In other parts, the same name may be bound to a different value or not bound at all. A *scope* is the set of all bindings that are valid within a particular context (part) of the whole program.

Lexical Scope and Context *lispy* is *lexically scoped*, meaning a bound name is resolved based on where the binding was defined. Thus, a "particular context" is specifically a lexical context. Generally, a *lexical context* is the boundary of a *lexical unit*. For the discussion of scope, the lexical units that need to be considered are a `let` expression, a function body, and the whole program.

Nested Scopes A scope can be nested within another scope. A nested scope inherits the bindings of its enclosing scope. Resolution of a bound name is first attempted in the local lexical context. If it fails to find the binding in the local lexical context, then it tries again with the outer lexical context. This is repeated until the outer-most lexical context is reached. Past that, a name is considered to be unbound, and a compiler error occurs.

A variable may be defined using a name that already has a binding in an outer scope. This variable *shadows* the one in the outer scope; according to the name resolution algorithm just described, the inner binding has precedence over the outer binding.

9.1 Scope Levels

There are three levels of scope: global scope, function scope, and `let` scope.

Global Global scope is the top level of the program. There is only one global scope, and all other scopes are nested within the global scope. Built-in functions are in the global scope.

Function In a function scope, a binding defined within a function does not extend outside of that function. With lexical scoping, a binding only has scope within the lexical context of the function i.e. the boundaries of that function's definition. This means that if the function calls another function, the bindings go out of context (because the other function is defined elsewhere); the bindings cannot be accessed from the called function. When that called function returns, the bindings come back into context.

let In a `let` scope, a binding has scope within the lexical context of that `let` expression (again, this is lexical scoping). As with functions, bindings go out of context when other functions are called and come back into context when that called function returns. Thus, the initial bindings (the $(a_i\ d_i)$ pairs) are accessible to all forms in the body, but not outside the `let` expression.

9.2 set

For `set` to rebind an existing variable, that variable has to be in an enclosing scope of the `set` call i.e. it should be possible to resolve that name as described earlier. Otherwise, the name is considered unbound and the program is considered invalid.

9.3 Closures

Nested functions complicate the matter. First, because a nested function is defined within the local lexical context, calling it does not cause bindings to go out of context. The nested function will have access to its own bindings and those of its enclosing scope. However, the enclosing scope will not have access to the nested function's bindings.

Second, because functions are first-class citizens, they can be returned and called from other contexts. This means the compiler must create a closure to store copies of the required non-local variables (names that only resolve in an enclosing context). A *closure* is a pair of a function and the set of non-local variables that are used within the function. Thus, even if the function is called outside its defined context, it will have access to all the variables it depends on.

10 Built-in Functions

First, recall the following built-ins, which have already been presented and will not be discussed any further in this section:

`list`	`cons`	`car`	`cdr`
`cond`	`select`	`let`	`set`
`lambda`	`progn`		

While discussing functions here, all arguments can be assumed to be evaluated unless explicitly noted otherwise. For example, stating an argument a must be a `bool` really means that it must be some form that evaluates to a `bool`. Furthermore, all arguments can be assumed to be evaluated left to right unless explicitly noted otherwise. Finally, recall that a special form is not a first-class citizen, so it cannot be used as an argument (but it can be called so that its return value is used as an argument).

10.1 Predicate Functions

(`eq` e_1 e_2) — *Special form* Returns `true` if the values of e_1 and e_2 have the same internal memory address. Otherwise, returns `false`. The values must be of the same type. The values cannot be `int`s, `float`s, or special forms. This is always `true` if both arguments are `true` or `false`.

(`equal` e_1 e_2) — *Special form* Returns `true` if the values of e_1 and e_2 are equivalent. Otherwise, returns `false`. The values must be of the same type. The values must not be `func`s.

`int`s are equivalent following the rules of mathematics. `float`s are similar, except that due to a lack of infinite precision, they're equivalent if they're within an implementation-defined distance of each other.

`list`s are compared element-wise (sensitive to ordering), recursing into nested lists if needed. Lists of differing lengths are trivially not equivalent. Lists must not contain a `func` at either the top level or in any nested list.

For the Booleans, `true` is trivially only equivalent to itself and so is `false`.

10.1.1 Arithmetic Predicates

(`greaterp` n_1 n_2) — *Special form* Returns `true` if $n_1 > n_2$. Otherwise, returns `false`. Both arguments must be `int`s or `float`s. However, they do not have to the same type; an `int` can be compared to a `float`.

(`evenp` n) — Returns `true` if n is an even number. Otherwise, returns `false`. n must be an `int`.

(`lessp` n_1 n_2) — *Special form* Returns `true` if $n_1 < n_2$. Otherwise, returns `false`. Both arguments must be `int`s or `float`s. However, they do not have to the same type; an `int` can be compared to a `float`.

10.1.2 List Predicates

(`null` l) — *Special form* Returns `true` if l is `nil`. Otherwise, returns `false`. l must be a `list`. Equivalent to `(equal` l `nil)`.

(`member` e l) — *Special form* Returns `true` e is an element of a list l. Otherwise, returns `false`. l must be a `list` and e must have the same type as l's elements (except for the implicit terminating `nil`).

e is said to be an element of a list l of length n if `(equal` e l_i`)` is `true` for any i in $[1, n]$.

10.1.3 Logical Connectives

(`not` p) — Returns `true` if p is `false`. Otherwise, returns `false`. p must be a `bool`. Equivalent to `(eq` p `true)`.

(`and` e_1 e_2 ... e_n) — *Special form* Returns `true` if all $e_1 \ldots e_n$ are `true`. Otherwise, returns `false`. $e_1 \ldots e_n$ must all be `bool`s. There must be at least two arguments.

(`or` e_1 e_2 ... e_n) — *Special form* Returns `true` if any of $e_1 \ldots e_n$ are `true`. Otherwise, returns `false`. $e_1 \ldots e_n$ must all be `bool`s. There must be at least two arguments.

`and` and `or` don't always evaluate all arguments. When a `false` argument is encountered for `and`, the evaluation ends early and the rest of the arguments remain unevaluated. When a `true` argument is encountered for `or`, the evaluation ends early and the rest of the arguments remain unevaluated.

10.2 Arithmetic Functions

The arithmetic functions only accept numeric types (`int` and `float`) as arguments. However, the arguments do not have to be homogenous in type. If there is at least one `float`, then an arithmetic function returns a `float`. Otherwise, it returns an `int`. If the initial result of the computation is not of the right type, then it will be implicitly converted using the `trunc` and `float` functions described in the next section.

A runtime error occurs when calling any of these functions with arguments outside the domain of the equivalent mathematical function e.g. division by zero.

(`sum` x_1 x_2 ... x_n)	*Special form* Returns $\sum_i^n x_i$, the sum of all arguments. There must be at least two arguments.
(`prod` x_1 x_2 ... x_n)	*Special form* Returns $\prod_i^n x_i$, the product of all arguments. There must be at least two arguments.
(`diff` x y)	*Special form* Returns $x - y$, the difference of x and y.
(`neg` x)	*Special form* Returns $-x$, the negation of x.
(`inc` x)	*Special form* Returns $x + 1$, x incremented by 1.
(`dec` x)	*Special form* Returns $x - 1$, x decremented by 1.
(`div` x y)	*Special form* Returns x/y, a single number that is the result of division with remainder.
(`mod` x y)	*Special form* Returns $x - y\mathrm{trunc}(\frac{x}{y})$, the remainder of dividing x by y through truncated division. This is the modulo operation.
(`expt` x y)	*Special form* Returns x^y, x to the power of y.
(`sqrt` x)	*Special form* Returns $\sqrt{x}$, the square root of x.
(`log` x y)	*Special form* Returns $\log_y x$, the logarithm of x to base y.

`(lb x)`	*Special form* Returns $\log_2 x$, the binary logarithm of x.
`(lg x)`	*Special form* Returns $\log_{10} x$, the common logarithm of x.
`(ln x)`	*Special form* Returns $\log_e x$, the natural logarithm of x.
`(recip x)`	*Special form* Returns $\frac{1}{x}$, the reciprocal of x.
`(abs x)`	*Special form* Returns $\lvert x \rvert$, the absolute value of x.
`(min` x_1 x_2 $\ldots$ x_n`)`	*Special form* Returns the smallest value in $x_1 \ldots x_n$. There must be at least two arguments. Note that even if the smallest value is an `int`, it will be returned as a `float` if there's at least one other argument that is a `float`.
`(max` x_1 x_2 $\ldots$ x_n`)`	*Special form* Returns the largest value in $x_1 \ldots x_n$. There must be at least two arguments. Note that even if the largest value is an `int`, it will be returned as a `float` if there's at least one other argument that is a `float`.

10.3 Numeric Conversion Functions

`(float x)`	Returns the `float` equivalent of an `int` x. For example, `(float 3)` is 3.0.
`(floor x)`	Returns $\lfloor x \rfloor$. x must be a `float`.
`(ceil x)`	Returns $\lceil x \rceil$. x must be a `float`.
`(trunc x)`	Returns the `float` x truncated to an `int`. This rounds towards 0, so -1.5 becomes -1 and 1.5 becomes 1.

(`round` x)	Returns the `float` x rounded to the nearest `int`. If x is exactly halfway, then it rounds towards the even choice. For example, 0.5 and -0.5 round to 0, and 1.5 rounds to 2.

10.4 Bit-wise Functions

The following functions only accept `int`s as arguments.

(`logand` x y)	Returns $x \wedge y$, the bit-wise AND of x and y.
(`logior` x y)	Returns $x \vee y$, the bit-wise OR of x and y.
(`logxor` x y)	Returns $x \oplus y$, the bit-wise EXCLUSIVE OR of x and y.
(`lognot` x y)	Returns $\neg x$, the bit-wise NOT of x.
(`shift` x y)	Returns x arithmetically shifted by y bits. If y is positive, x is shifted to the left. If y is negative, x is shifted to the right.

10.5 List Functions

(`append` e l)	*Special form* Returns copy of the `list` l with a new element e at the end of it (it is still terminated with `nil`). The copy behaves like the built-in `copy` function. e must have the same type as the elements of l.
(`extend` l_1 l_2)	*Special form* Returns a new `list` which is the combination of the elements of the `list`s l_1 and l_2. The elements are copied in order like with the built-in `copy` function. The elements of l_2 follow those of l_1. l_1 and l_2 must have the same type.

`(copy l)` — *Special form* Returns a shallow copy of the `list` l. In a shallow copy, all the top-level elements are copied, but any nested lists are not recursively copied. Thus, for some list `A`, `B = (list A)`, and `C = (copy B)`, `(eq C B)` is `false`, `(equal C B)` is `true`, and `(eq (car C) A)` is `true`.

`(reverse l)` — *Special form* Returns a new `list` which contains the elements of `list` l in reversed order. Only the top-level elements are reversed; nested keep their internal order. `nil` is excluded from the reversal and is still at the end to terminate the new list.

`(length l)` — *Special form* Returns an `int` which is the number of items in the `list` l. The terminating `nil` is not counted towards the length, and `(length nil)` is 0.

`(efface e l)` — *Special form* Returns a copy of the `list` l with the first occurrence of the element e removed. e must have the same type as the elements of l. Elements are compared using the built-in `equal` function, so e must not be a `func` or a `list` which contains a `func` at the top level or in any nested list. The copy behaves like the built-in `copy` function.

10.6 Input and Output

Values can be *printed* i.e. written to the standard output (*stdout*). However, input is not supported. Values are output using their S-expression representation. The exception to this is functions, which are output as "<function at *memory-address*>". Special forms cannot be printed.

`(print` e_1 e_2 … e_n`)` — *Special form* Prints a single line of all arguments concatenated with a space delimiter. The arguments do not need to be homogenous in type. Zero arguments may be given, in which case only a single space character is printed. Returns `nil`.

`(println` e_1 e_2 $\ldots$ e_n`)` *Special form* Similar to `print` except that the line is also terminated with a newline character. Zero arguments may be given, in which case only a single newline character is printed.

10.7 Mapping Functions

`(map` l f`)` *Special form* Returns a `list` of the results of applying the function f to the list l and to successive `(cdr` l`)` until l is reduced to `nil`. f must be a `func` with a single parameter whose type matches the type of the elements of `list` l.

`(mapcar` l f`)` *Special form* Returns a `list` of the results of applying the function f to each element of the list l, excluding the terminating `nil`. f must be a `func` with a single parameter whose type matches the type of the elements of `list` l.

Appendix A Syntax

A.1 Abstract

Note: * denotes zero or more of the preceding term. + denotes one or more of the preceding term.

```
<name> ::= <string of letters and numbers>
<number> ::= <a decimal number> | "inf" | "nan"
<const> ::= <number> | "true" | "false"

<type> ::= "int" | "float" | "bool"
         | "(" "list" <type> ")"
         | "(" "func" "(" <type>* ")" <type> ")"

<nil> ::= "(" ")"

<func-param> ::= "(" <name> <type> ")"
<lambda> ::= "(" "lambda" "(" <func-param>* ")" <form> ")"

<list> ::= "(" "list" <form>* ")"
<cons> ::= "(" "cons" <form> <form> ")"
<car> ::= "(" "car" <form> ")"
<cdr> ::= "(" "cdr" <form> ")"

<progn> ::= "(" "progn" <form> <form>+ ")"
<set> ::= "(" "set" <name> <form> ")"

<let-binding> ::= "(" <name> <form> ")"
<let> ::= "(" "let" "(" <let-binding>+ ")" <form>+ ")"

<branch> ::= "(" <form> <form> ")"
<cond> ::= "(" "cond" <branch>+ <form> ")"
<select> ::= "(" "select" <form> <branch>+ <form> ")"

<elementary-form> ::= <const> | <name>
<composed-form> ::= "(" <form> <form>* ")"
<special-form> ::= <lambda> | <define> | <list> | <cons> | <car> | <cdr>
                 | <progn> | <set> | <let> | <cond> | <select> | <eq> | <equal>
                 | <evenp> | <lessp> | <null> | <member> | <and> | <or> | <sum>
                 | <prod> | <diff> | <neg> | <inc> | <dec> | <div> | <mod>
                 | <expt> | <sqrt> | <log> | <lb> | <lg> | <ln> | <recip>
                 | <abs> | <min> | <max> | <append> | <extend> | <copy>
                 | <reverse> | <length> | <efface> | <print> | <println>
                 | <map> | <mapcar>
<builtin-form> ::= <greaterp> | <not> | <float> | <floor> | <ceil> | <trunc>
                 | <round> | <logand> | <logior> | <logxor> | <lognot>
                 | <shift>
<form> ::= <elementary-form> | <composed-form> | <special-form> | <builtin-form>
```

```
<program> ::= <form>*

<eq> ::= "(" "eq" <form> <form> ")"
<equal> ::= "(" "equal" <form> <form> ")"

<greaterp> ::= "(" "greaterp" <form> <form> ")"
<evenp> ::= "(" "evenp" <form> <form> ")"
<lessp> ::= "(" "lessp" <form> <form> ")"

<null> ::= "(" "null" <form> ")"
<member> ::= "(" "member" <form> <form> ")"

<not> ::= "(" "not" <form> ")"
<and> ::= "(" "and" <form> <form>+ ")"
<or> ::= "(" "or" <form> <form>+ ")"

<sum> ::= "(" "sum" <form> <form>+ ")"
<prod> ::= "(" "prod" <form> <form>+ ")"
<diff> ::= "(" "diff" <form> <form> ")"
<neg> ::= "(" "neg" <form> ")"
<inc> ::= "(" "inc" <form> ")"
<dec> ::= "(" "dec" <form> ")"
<div> ::= "(" "div" <form> <form> ")"
<mod> ::= "(" "mod" <form> <form> ")"
<expt> ::= "(" "expt" <form> <form> ")"
<sqrt> ::= "(" "sqrt" <form> ")"
<log> ::= "(" "log" <form> <form> ")"
<lb> ::= "(" "lb" <form> ")"
<lg> ::= "(" "lg" <form> ")"
<ln> ::= "(" "ln" <form> ")"
<recip> ::= "(" "recip" <form> ")"
<abs> ::= "(" "abs" <form> ")"
<min> ::= "(" "min" <form> <form>+ ")"
<max> ::= "(" "max" <form> <form>+ ")"

<float> ::= "(" "float" <form> ")"
<floor> ::= "(" "floor" <form> ")"
<ceil> ::= "(" "ceil" <form> ")"
<trunc> ::= "(" "trunc" <form> ")"
<round> ::= "(" "round" <form> ")"

<logand> ::= "(" "logand" <form> <form> ")"
<logior> ::= "(" "logior" <form> <form> ")"
<logxor> ::= "(" "logxor" <form> <form> ")"
<lognot> ::= "(" "lognot" <form> <form> ")"
<shift> ::= "(" "shift" <form> <form> ")"

<append> ::= "(" "append" <form> <form> ")"
```

```
<extend> ::= "(" "extend" <form>  ")"
<copy> ::= "(" "copy" <form> ")"
<reverse> ::= "(" "reverse" <form> ")"
<length> ::= "(" "length" <form> ")"
<efface> ::= "(" "efface" <form> <form> ")"

<print> ::= "(" "print" <form>* ")"
<println> ::= "(" "println" <form>* ")"

<map> ::= "(" "map" <form> <form> ")"
<mapcar> ::= "(" "mapcar" <form> <form> ")"
```

A.2 Concrete

```
(* -------------------------- Character sets -------------------------- *)
letter = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K"
       | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V"
       | "W" | "X" | "Y" | "Z" | "a" | "b" | "c" | "d" | "e" | "f" | "g"
       | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r"
       | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" | "_";
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
ws = " " | "\r" | "\n" | "\t" | "\f" | "\v";

(* ---------------------------- Numbers ------------------------------ *)
sign = "+" | "-";
radix = ( digit, "." ) | ( ".", digit );
decimal = { digit }, radix, { digit }
        | digit, { digit };
exponent = ( "e" | "E" ), [ sign, digit ], { digit };
number = [ sign ], ( "inf" | "nan" )
       | [ sign ], decimal, [ exponent ];

(* ----------------------------- Atoms ------------------------------- *)
(* Literal atoms must start with a letter. *)
literal_atom = letter, { letter | digit };
bool = "true" | "false";
atom = number | bool | literal_atom;

(* -------------------------- S-expressions -------------------------- *)
(* Empty lists are valid too. *)
(* List elements must be delimited by >= 1 ws. *)
sexpr_elements = { s_expression, ws }, s_expression;
list_elements = sexpr_elements | s_expression | { ws };

sexp_non_atomic = "(", list_elements, ")";
s_expression = { ws }, ( atom | sexp_non_atomic ), { ws };

(* ---------------------------- Program ------------------------------ *)
(* Consecutive atoms must be separated by >= 1 ws. *)
atoms = { ws }, atom, { ws }, { ws, { ws }, atom, { ws } };
program = { [ { ws }, sexp_non_atomic, { ws } ], [ atoms ] };
```