Project F
Functional Language

Introduction & Common Description

The F language can be considered as a reduced version of the **Lisp** language with some simplifications and modifications. It takes the basic syntax and semantics from Lisp.

Program structure and declarations

The F program is a sequence of elements. Elements are either atoms – identifiers or literals – or lists.

There are three kinds of entities defined in the language.

Literals are just values that are written explicitly. There are integer, real and boolean literals. The syntax of literals is quite simple and is presented at the end of the document. It is said that literals represent themselves.

Atoms can be considered as variables in conventional languages. They have the usual syntax of identifiers and can have a value. The atom's value is a value of a literal, or a list. Notice that in some contexts the value of an atom is not considered; in such contexts atoms represent themselves.

List is a sequence of elements separated by whitespaces and enclosed by parentheses.

The program execution starts from its very first element. Each element is evaluated in accordance with its semantics. If the current element is an atom, the F processor (interpreter) simply returns its value. If the current element is a list, it is treated as a call to a function, whose name is the name of the first list element, and the other list elements are considered arguments of the call.

Some lists have a special meaning. If a list starts with one of the following keywords it is called *special form* and is evaluated as described in the following section.

The keywords of the special forms are:

```
quote   setq   func   lambda   prog   cond   while   return  break
```

Special forms

Special forms introduce language notions that have some predefined meaning. Each special form is actually a list where the very first element is a keyword. The rest of the special form can contain a number of `Element`s that are specific for each special form.

```
( quote Element )
'Element
```

The function just returns its argument without evaluating it. The meaning of the function is to prevent evaluating its argument. Using the single quote sign in front of an element is actually the short form of the function.

Examples:

```
(setq x (plus 1 2))  // x gets the value of 3
x        // the value of 3
'x       // the atom x itself
(setq y '(plus 1 2))  // y gets the list (plus 1 2)
'5     // the same as 5
```

```
( setq Atom Element )
```

The form performs assigning a value to an atom. The second parameter gets evaluated, and the evaluated value becomes the new value of the atom from the fist parameter replacing its previous value. If there was not an atom with the given name in the current context then it is created and added to the context. Otherwise, the atom that already exists in the context gets the new value.

Examples:

```
(setq x 5)
(setq y (plus 1 2))   // atom y gets the value of 3
(setq z null)
(setq t '(plus minus times divide))
                     // t gets the list of four atoms
```

```
( func Atom List Element )
```

The form introduces a new user-defined function. The first argument becomes the name of the function. The second argument should contain a number of atoms that represent the function parameters. The third parameter of the form is considered as the body of the function.

Notice that each user-defined function introduces its own local context. This means that atoms representing parameters are considered local to the function as well as all atoms introduced by the **`setq`** form. If the **`setq`** form introduces an atom with the same name as an atom from an outer context, the local atom shadows the atom from the outer context. The local context of the function disappears after exiting from the function.

Notice that only three special forms introduce local contexts: **`func`**, **`lambda`**, and **`prog`**.

Examples:

```
(func Cube (arg) (times (times arg arg) arg))
(func Trivial () 1)
(func makeList (A) (A))
```

`( lambda List Element )`

The form introduces a new user-defined unnamed function. The meaning of its parameters is the same as for the second and the third parameters of the **`func`** form. The unnamed function can further be called by that name of an atom that gets it as the value, or directly.

Examples:

```
(lambda () (1 2 3 4))
(setq myFunc (lambda (p) (cond (less p 0) plus minus)))
((myFunc -1) 1 2)     // returns 3
(((lambda (p) (cond (less p 0) plus minus)) +1 1 2) // returns -1
```

`( prog List Element )`

The form introduces a sequence of elements that are to be evaluated sequentially. The first parameter is the list of atoms that represent the local context of the form. These atoms become known everywhere within the **`prog`** form and disappear after completing its evaluation. The second argument contains elements that are to be evaluated sequentially.

`( cond Element1 Element2 [ Element3 ] )`

The form is the construct for conditional evaluation. It contains two or three arguments. The evaluation of the form starts from evaluating its first argument. The result of the evaluation should be of type boolean. If the result is **`true`** the second argument is evaluated, and the result becomes the result of the whole form. Otherwise, the third argument is evaluated and the result becomes the result of the whole form. If the result of the first argument is **`false`** and there is no third argument in the form, the result of the whole form is **`null`**.

`( while Element Element )`

The form specifies repetitions. First, the second argument is evaluated. If the result is **`true`** then the second argument is evaluated, and the control from goes back for evaluating of the first argument again. In other words, the first argument is evaluated before each iteration. If the result is **`false`** then the evaluation of the form finishes. The result of the form is always **`null`**.

`( return Element )`

The form makes sense within a form that defines a local context (**`func`**, **`lambda`** or **`prog`**). It evaluates its argument and interrupts the execution of the nearest enclosing form with the context. If there is no such enclosing form then the whole program terminates.

`( break )`

The form makes sense within a `while` form. It unconditionally interrupts the execution of the nearest `while` form. If there is no such enclosing form then the whole program terminates.

Predefined functions

All predefined functions perform some actions on their arguments and return some result. The result can be an atom, a list, a literal, or null. Function calls are represented as lists where the first element is atom (identifier) that is the name of the function being called.

Notice that a function name is considered as the name of a predefined function only if it is written as the first atom in a list. In other contexts, a function name is treated as a name of a usual atom. For example, the list like `(4 minus times divide)` is just a list of atoms but not function calls.

The common algorithm of function call evaluation is as follows:

- All arguments are evaluated. The evaluation order is from the first argument to the last one.
- The results of argument evaluation are passed to the function.
- If the current value of an argument doesn't meet the function requirements, the whole program stops execution.
- The function accepts evaluated arguments and performs actions specific to this particular function.
- After completing the function's actions, a value is returned to the calling function.

Arithmetic functions

```
( plus Element Element)
( minus Element Element )
( times Element Element )
( divide Element Element )
```

Arithmetic functions have two parameters. After their evaluation, the arguments should be of an integer or real value. The functions perform addition, subtracting, multiplication or division on their arguments. The result of the function call is the result of the corresponding action.

Operations on lists

```
( head Element )
```

After evaluation, the argument should be a list. The function returns the first element of the list from the argument.

```
( tail Element )
```

After evaluation, the argument should be a list. The function returns the initial list without its first element.

```
( cons Element Element )
```

After evaluation, the first argument can be of any type. The second argument should be a list (perhaps, empty). The function constructs a new list adding its first argument as the first element to the list from the second argument. The function returns the list constructed.

Comparisons

```
( equal Element Element )
( nonequal Element Element )
( less Element Element )
( lesseq Element Element )
( greater Element Element )
( greatereq Element Element )
```

After evaluation, the arguments should be of type integer, real, or boolean. The functions perform usual comparisons and return a boolean value depending on the result of comparison.

Predicates

```
( isint Element )
( isreal Element )
( isbool Element )
( isnull Element )
( isatom Element )
( islist Element )
```

After evaluation, the arguments should be of any type. The functions return a boolean value: **`true`**, if the argument is of type that the function expects, and **`false`** otherwise.

Logical operators

```
( and Element Element )
( or Element Element )
( xor Element Element )
( not Element )
```

After evaluation, the arguments should be of boolean type. The functions perform usual logical operators on evaluated arguments and return a boolean value.

Evaluator

```
( eval Element )
```

After evaluation, the argument should be of any type. If the argument if a list then the function treats it as a valid program and tries to evaluate it. In that case, the function returns the value that the program issues. If the argument is a literal or atom the function just return the argument.

The Language F Grammar

The language grammar follows the ideas of its predecessors and is remarkably simple.

```
Program : Element { Element }

List : ( Element { Element } )

Element : Atom | Literal | List

Atom : Identifier

Literal : [+|-] Integer | [+|-] Real | Boolean | null

Identifier : Letter { Letter | DecimalDigit }

Letter : Any Unicode character that represents a letter

Integer : DecimalDigit { DecimalDigit }

DecimalDigit : 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Real : Integer . Integer

Boolean : true | false
```