Alpes Language Universe

Throughout the course we are going to implement a programming language covering different characteristics of the topics covered in class. This language is called the Alpes Language Universe, or ALU for short.

ALU programs are defined by different modules used to define the different kinds of abstractions available in the language, which can be functions, and data abstractions. Programs in ALU are sequences of modules, in which modules cannot be nested (all modules exist at the same level).

Functions are defined as abstractions that can receive a set (possibly empty) of parameters, execute a set of expressions, and always return a value. As a convention the value returned by functions is the last expression evaluated in the function definition.

Data abstractions are used to define objects and the operations (functions and expressions) to manipulate the created objects. Following the definition on the data abstraction a definition of its operations is given. By definition in the language, data abstraction must have a recursive definition. That is, data must have a bases and build from it, or explicitly construct the abstraction; that is, null pointers do not exist in the language.

Variables in ALU are declared by giving the list of new variable identifiers to introduce into the system. Each of the identifiers should be given in a single line separated by commas. For example `a, tree, new-var` introduces three new variables, namely a, tree, and new-var. Values are given to variables by means of assignments (*e.g.,* `a = 5`).

As a means of reusability, both functions and data defined can be associated with a variable identification through the assignment (=) operator. The assignment of a variable with a particular value (say a function) automatically introduces said variable to the system.

ALU has three control flow expressions, namely `if`, `cond`, and `for` expressions. All control flow expressions are delimited by their keywords and conclude with an `end` keyword. If expressions are fully defined, that is, they always contain the condition and the two branches (`then` and `else`). Conditional expressions are used to define the pattern matchings over multiple expressions, for example matching the value of variable to positive or negative integers could be expressed as

```
cond x do
  x > 0 -> "positive"
  x < 0 -> "negative"
end
```

For expressions define an iteration variable to iterate over ranges or iterative structures. Ranges create an iterative structure between two positive numbers. Iterative structures can also be build from data abstractions like sequences, maps, trees, etc. The idea of iterative structures is that they realize an entity of the structure and assign it to a loop variable. Once the loop is done, this gives way to the iterative structure to yield the next object in the structure.

```
from x to y //create a range from x to y
it = iterator (tree) yielding (node) //create an iterative
    structure for each of the nodes in a tree
//full loop example
for i from 1 to 100 do
  tree = tree.create(i)
  leaves = iterator (t) yielding (node)
  for leaf in leaves(tree) do
    tree.value(leaf)
  end
end
```

ALU offers programmer the basic data values (*i.e.,* numbers (both floats and integers), booleans, chars) (strings are defined as lists of chars) with their basic operations. Additionally, ALU defines two basic structures to store data, sequences (defining linked lists) and represented between parenthesis and tuples (two element grouped values) represented as parenthesis as well. Again both sequences and tuples. It is also possible to define structures as sets of data to be used. Structures are used in the creation of data abstractions defining the names of the parameters to the values making up the structure. Structures can be thought of as maps of names and values, implemented in ALU as lists of tuples.

Finally all ALU expressions are delimited by a newline, that is, a change of line determines the end of an expression (which can now be evaluated), and the beginning of the following expression

1 ALU examples

As example programs of the use of ALU we provide the following definitions (in Snippet 1) of the different kinds of abstractions available to the system, and the programs we could write in the language

Note from the examples that there are two distinctive ways to access functions. Functions within the same data abstraction are called with the `$` operator. While access to functions of a particular object of a data abstraction, or the elements of an struct is done with the . operator.

2 Reserved words

Based on the language examples, the following are reserved words of the language

cond	do	data	elseif	end
for	from	then	function	else
if	in	iterator	sequence	struct
to	tuple	type	with	yielding

3 Identifiers

Identifiers are defined as a sequence of letters, digits, and dashes (-) that begin with a letter, so that they are not reserved words. For the definition of ALU we ignore capitalization rules for identifiers and reserved words; all should be understood as small caps words.

4 Literals

The following are the sequences of symbols used as operators and punctuation symbols.

*	/	-	+	**	%
<	>	<=	>=	<>	=
and	or	neg	->	:	$
.	(	)	[	]	

Note that this is an initial definition of ALU which you will need to grow as more concepts, features, and restrictions are introduced. Therefore, you should take into account the clean design directives for the language.

Task 1. For the first part of the project, your task is to provide the grammar definition of ALU. Your definition should clearly state the sets of non-terminals and symbols (the alphabet) as well as the distinguished element of the non-terminals and the production rules.

To define the production rules you should use EBNF notation (where non-terminals are defined as barewords and terminals are defined between quotes ("")).

Hand-in a pdf file, with the names of the two students in your group, and the definition of four components of the grammar.

```
complex = data with create, add, get_theta, equal //data abstraction definition
  rep = struct(real,img)

  create = function (x,y) do
    rep$(real:x, img:y)
  end create

  add = function (cmplx1, cmplx2) do
    rep$(real: cmplx1.real + cmpl2.real, img: cmplx1.img + cmplx2.img)
  end add

  get-theta = function (cmplx) do
    arctan2(cmplx.real, cmplx.img)
  end get-theta
end complex

sqrt = function (x, g) do //function definition
  if good-enough(x, g)
  then g
  else sqrt(x, improve(x,g))
  end
end sqrt

good-enough = function(x, g) do
  abs(x-g*g) < 0.001
end good-enough

improve = function (x, g) do
  average(g, x/g)
end improve

average = function (x, y) do
  (x+y)/2.0
end average
```

Snippet 1: ALU examples