Programmiersprachen – Programming Languages
LVA 185.208, VU, 3 ECTS, 2014 S

Second Exercise

Exercise

Develop an interpreter of procedures in a language based on guarded commands as specified below. The interpreter shall be written in a dynamically typed scripting language like Perl, PHP, Python and Ruby.

The language

All variables in the language can hold strings (there are no other data types) and have a strict single-assignment property. We distinguish between bound variables holding a string and free variables without a value. A free variable becomes bound when assigning a string to it. No value must be assigned to a bound variable.

Variables can occur as parts of strings. To distinguish them from other characters in a string we enclose them into $ signs at the begin and end. For example, if variable `x` is bound to `"1"`, `y` to `"2"` and `three` to `"3"`, then the string `"a$x$$y$b$three$"` expands to `"a12b3"`. As long as any of the variables occurring in a string is free, the string cannot be expanded. Computations depending on the string have to be delayed until all variables occurring in the string are bound.

Actually, waiting for variables to become bound is all that we need to coordinate the execution. Waiting allows us to express sequences, alternatives, concurrency and synchronization. It is not necessary to force the execution into a specific sequence.

Proposed language details

A program is an unordered set of procedures. It has the following syntax (in EBNF where terminals are in quotes):

```
program ::= {procedure}
procedure ::= name {name} '-' string {string} '{' {guarded_command} '}'
```

Each procedure begins with the name of the procedure followed by a possibly empty list of input parameters to the left of “`-`”. Each variable in the input parameter list becomes bound on procedure invocation. Strings to the right of “`-`” represent results. A procedure invocation returns the results as soon as all variables in these strings are bound. There must be at least one result.

The body of the procedure contains an unordered set of guarded commands in braces:

```
guarded_command ::= {guard ':'} command ';'

command ::= name {name} '=' name {string}
          | name [name] '=' 'exec' string [string]
          | name name  '=' 'split' string string
          | name '=' string {string}

guard ::= name
        | name '==' string
        | name '!=' string
        | 'finally'
```

Each guarded command in a procedure is executable when

- all of its guards are satisfied,
- all variables occurring in strings to the right of “`=`” are bound,
- and none of the variables occurring to the left of “`=`” is bound.

A guard consisting just of a variable name is satisfied when the variable is bound. If a guard compares a variable name with a string for equality or inequality, this variable and all variables in the string have to be bound before the guard can be satisfied. A `finally` guard defers the execution to the latest possible point in time, see below.

There are several kinds of commands (described here in the same ordering as given as alternatives in the EBNF):

1. In a *procedure invocation*, we find the name of a procedure to the right of “`=`”, possibly followed by strings used as arguments. To the left of “`=`” there is a variable for each result. It is an error if no procedure of the given name exists, the number of arguments does not correspond to that of the parameters, or the number of variables does not correspond to that of the results.
2. An *external invocation* lets the operating system execute a command specified by the string immediately following `exec`. The result of the execution is bound to the first variable to the left of “`=`” (usually `"0"` for successful termination and other values for erroneous termination). If there is a second variable to the left of “`=`”, this variable is bound to a string containing the complete output written to the standard output stream of the executed command. Accordingly, if there is a second string to the right of `exec`, this string is used as standard input for executing the command. For example,

   ```
   x y = exec "cat" "test"
   ```

   will bind `x` to `"0"` and `y` to `"test"` because `cat` just copies standard input to standard output.
3. *Splitting* can be used to divide a string into parts. For example,

```
    x y = split "/" "a/b/c"
```

will bind `x` to `"a"` and `y` to `"b/c"`. This is, the first string determines a sub-string, and the second string will be split at the first occurrence of this sub-string. If the sub-string does not occur in the second string, the first variable will be bound to the whole string and the second variable to an empty string. If the sub-string occurs at the very beginning of the second string, the first variable will be bound to an empty string and the second variable to the second string shorted by the sub-string.

4. An *assignment* simply binds the variable to the left of “`=`” to the concatenation of all strings to the right of “`=`”.

Each guarded command in a procedure invocation can be executed at most once because after execution corresponding variables are no longer free. All executable commands can be executed in parallel. However, if several commands bind the same variable, at most one of them can be executed. The interpreter has to select any of them (provided that it is executable) and ignore the others.

A procedure returns its results as soon as all variables in the result strings are bound. It is possible that a procedure returns its results before the executions of its commands has terminated. In that case the execution continues after return until no executable command is left. A variable in a result string may never become bound. It is an error if a variable in a result string is free although no command in the procedure body is executable anymore. A `finally` guard helps us to avoid that case as shown in the following example:

```
maxnum a b c - "$max$" {
    ab = exec "test $a$ -le $b$";
    bc = exec "test $b$ -le $c$";
    ab == "0" : bc == "0" : max = "$c$";
    ab == "0" : bc == "1" : max = "$b$";
    ab == "1" : bc == "1" : max = "$a$";
    ab == "1" : bc == "0" : ac = exec "test $a$ -le $c$";
    ac == "0" : max = "$c$";
    ac == "1" : max = "$a$";
    finally : max = "error";
}
```

In Unix that use of `test` compares two strings supposed to represent integers. If the first integer is less than or equal to the second integer, the result is `"0"`, and if the first integer is larger than the second one, the result is `"1"`. In `maxnum`, two executions of `test` run concurrently. If possible, the result values determine the result bound to `max`. Only in one case we need a further execution of `test`. However, there is a problem because `test` will return a different unknown value in the case of an error, maybe `"2"` if any of the arguments is not an integer. The `finally` guard provides a solution: This guard is satisfied only if all other guarded commands are no longer executable.

All the details given above describe an *example* of a language that you shall develop. Please feel free to change all the details and develop your own language. However, your language must distinguish between bound and unbound variables, variables must have a single assignment property, the execution must be coordinated by waiting for variables to become bound (supporting concurrency, sequences and alternatives), and procedures must be recursively invokable.

Testing

Please develop at least one nontrivial program in your language to test the language and its implementation. Such coordination languages have advanced capabilities to deal with concurrency and rather complicated forms of coordination. They make it easy to invoke programs in a similar way as do shell scripts. Your test application(s) shall make use of these capabilities. For example, you can develop a tool to sort files into several directories according to file names, file extensions, file sizes, change dates, etc., and create a protocol telling where files have moved, thereby using a maximum of parallelism.