COMP2212 Programming Language Concepts

The Paragon Programming Language User Manual

Andrei Rusu
Eugen Robert Patrascu

April 28, 2017

Contents

1 General Information

The Paragon Programming Language is a domain specific language aimed at performing simple computations on languages (i.e. a language is a set of lexico-graphically ordered words over an alphabet). It performs operations such as the union, intersection and difference of languages, or concatenation between a language and a string. However, it also includes operations for booleans, integers and strings, which are necessary to compute language operations.
It includes four types of variables (boolean, integer, string and language), and it contains a number or basic control structures such as simple-if statements, if-else statements, and while loops.
In terms of input and output operations, the language has specific 'read' and 'write' operations depending on the variable types.
Overall, Paragon is a domain specific programming language with a simple syntax and helpful additional features, which make it easy to write and understand.

2 Variable Types and Operations

2.1 Variable Types

The language supports four types of variables: *boolean*, *integer*, *string* and *language*. The name of a variable must be composed of a specific symbol depending on the variable type: '?' for boolean, '#' for integer, '$' for string and '&' for language, followed by a number of lowercase or uppercase letters, digits or underlines. This convention has been established for making the code easier to write and understand, and it offers a certain level of type-safety. The declaration of a variable begins with the keyword 'def'. The declaration and initialisation of variables are of the form $def < variable\_name >=< variable\_value >$, as demonstrated in the following code bit:

```
/* Variables */
def ?boolVar = true; /*declaring a boolean variable with the value true */
def #intVar = 5; /* integer variable of value 5 */
#intVar = 7; /* changing the value of the integer variable to 7*/
def $strVar = "comp2212"; /*initialising a string variable */
def &setVar = {"a", "b"}; /*declaring and initialising a language variable
   */
```

Observation: variables must be assigned a value using the assignment operator '=' when they are declared; there are no implicit values for the variables, which means a declaration of the type 'def ?b;' is incorrect.

2.2 Variable operations

Each of the four types of variables existing in the language supports specific operations. As Paragon is a domain specific language used for solving language problems, certain operations are used to support the specific computations. For example, boolean operations are used in while loops for traversing the elements of a language, integer operations are used for defining counter variables in the same context, and string operations are used for doing language concatenations.

2.2.1 Boolean operations

The operations supported by the language, which return Boolean values are: NOT (!), AND (&&), OR (||), EQUALS (==), LESS THAN (<), GREATER THAN (>).
The associativity of the variables is as follows: AND, OR are left associative, whereas the others are non-associative. The following code snippet demonstrates the usage of boolean operations:

```
def ?b = !true;
def ?c = (1<2) && (2<3);
?b = (2+3) == (5-2);
```

2.2.2 Integer operations

The operations supported by the language for Integer variables are: addition (+), substraction (-), multiplication (*), division (/) and modulus (%). It supports both negative and positive integer operations. All operations are left associative, and multiplication, division and modulus have higher precedence than addition and substraction. However, paranthesis can be used to modify this precedence. The following code snippet demonstrates the usage of integer operations:

```
def #i = 15 - 4 * 2; /* #i is 7 because of the operations order */
#i = (15 - 4) * 2; /* #i is now 22 due to the parantheses */
#i = #i + 1; /* incrementing #i by 1*/
def #j = #i * 5;
```

2.2.3 String operations

The operations supported by the language for String variables are concatenation (^) and getting an element from a language by specifying its position (lang_get) . The following code snippet demonstrates them:

```
def $s1 = "abc";
def $s2 = "def";
def $conc = $s1 ^ $s2; /* $conc will become "abcdef" */
$conc = $conc ^ "ghi" /* and now it becomes "abcdefghi" */
def &l1 = {"a", "b", "c"};
def $s = lang_get &l1 1; /* $s will be "b" */
```

2.2.4 Language operations

The operations supported by the language for Language variables are: union (lang_union), intersection (lang_inter), difference (lang_diff), the addition of an element to a language (lang_add) and the removal of an element from a language (lang_remove) .
The following code snippet illustrates the usage of Language operations:

```
def &l1 = {"a", "b", "c"};
def &l2 = lang_union &l1 {"a", "d"}; /* &l2 will be {"a", "b", "c", "d"}
    */
def &l3 = lang_inter &l2 {"e", "c"} /* &l3 will be {"c"} */
def &l4 = lang_diff &l1 &l2; /* &l4 will be {"b", "c"} */
&l1 = lang_add &l1 "d"; /* &l1 will be {"a", "b", "c", "d"} */
&l1 = lang_remove &l1 "a"; /* &l1 will be {"b", "c", "d"} */
```

3 Control Structures

The control structures supported by the language are: simple if statements, if-else statements and while loops.

3.1 Conditionals

If-statements have the following structure: $if < condition > then < statement > end$. If-else statements have the following structure: $if < condition > then < statement1 > else < statement2 > end$. The following code snippet illustrates their usage:

```
def #x = 3;
if (5>3) then #x = 5 end; /* #x will be 5 */
if (2>3) then #x = 8 else #x = 9 end; /* #x will be 9 */
```

3.2 Loops

The only type of loop supported is the while loop, which has the following structure: $while < condition > do < statement > end$. A 'break' can be used to stop the loop. The following code snippet illustrates its usage:

```
def #x = 3;
while (#x > 1) do
    #x = #x - 1
end;
```

4 Input and Output

The language supports a number of different 'read' and 'write' operations, which may differ depending on the type of variables written.

4.1 Input

There are two 'read' operations, necessary for solving language specific problems: *read_lang* which reads a language, and *read_int* which reads an integer. The position of each variable value in the input must be stated. Before reading the values, the input must be loaded from standard input using the command load_input. The following code snippet illustrates the operations:

```
load_input;

/* reads one set from position 0 and k from position 1 */
def &L1 = read_lang 0;
def #k = read_int 1;
```

4.2 Output

There are five different 'write' operations: *print* for printing any type of variable or value, *print_line* which is equivalent to *print* followed by printing a new line, *print_newline* which prints a new line, *print_lang* which prints the first 'n' elements of a language, and *print_lang_line* which is equivalent to *print_lang* followed by printing a new line. Paragon contains the specific language 'write' operations, as it is a domain specific programming language. The following code snippet demonstrates how to use 'write' operations:

```
def #x = 5;
def &l = {"a", "b", "c"};
print #x; /* prints 5 */
print_line #x /* prints 5 followed by a blank line */
print_newline /* prints a blank line */
print_lang &l until 2; /* prints the language containing first 2 elements
   of &l, {"a", "b"} */
print_lang_line &l until 0 /* prints the empty language {} */
```

5 Additional features

5.1 Programmer convenience

A programmer can easily use Paragon to solve language manipulation problems, as it supports Language as a basic data type, providing operations to easily construct new Languages with different properties. Even though the types and their supported operations are named so that the code is self-documented, the programmer can add comments to the code, using the structure $/*$ *add your comment here* $*/$.

5.2 Type checking

The type checking is made implicit as the parser constructs different branches in the abstract syntax tree for each type and each operation on types, disallowing inconsistencies. The variables make no exception from this as their initial character uniquely identifies their type, disallowing them to be assigned to any other type.

5.3 Error messages

Lexing and Parsing errors are well separated when printed to stderr for a cleaner error reporting. To aid in debugging, Paragon indicates roughly where the error was produced, showing the approximate line and column numbers. Moreover, the token around which the error appeared is also indicated. Internal errors, like an unbound variable, are clearly reported using OCaml's internal mechanisms.