**Davide Cremonini – Davide Perez – Davide Sbetti**

**GINSENG PROGRAMMING LANGUAGE AND GINSENG COMPILER**

This is *ginseng 1.1.0*, our very first compiler for the *Ginseng* programming language, developed by us.

**Ginseng programming language**

*Ginseng* is a small, compact and strongly typed programming language. It has the following characteristics:

* **Set-oriented paradigm**: *Ginseng* provides a native *set* type, that reflects the properties of its mathematical counterpart (more on this in the type system section).
* **Embedded operators**: provides the developers with a collection of built-in operators to efficiently manipulate data.
* **Simple and intuitive notation**: *Ginseng* operations follow the most common notational conventions, making it an intuitive and easy to use programming language for those who have even just basic knowledge of math.

**Ginseng type system**

Ginseng is a strongly typed language: *type inference* is used during compilation to infer variable types and enforcing operations on such. The reason behind this is to keep the syntax as light as possible, making the task of writing a ginseng source file as fast and less verbose as possible, while maintaining a good level of expressiveness in the meantime. Variables are all global-scoped.

Ginseng offers the following 3 native data types:

* **number**: this is the basic data type. It is used to represent numbers of any kind, i.e. integers and floating-points.
* **array (sequences)**: this data type is the equivalent to the array data structure native to most programming languages. It fits good in abstracting mathematical sequences, as they have the same ordering and the same indexing. It is possible to declare an array variable, populate it and retrieve elements from it, either via expression or via indexing. Arrays have unbounded length and can grow dynamically as new elements are added, or as an element is inserted in a non-existent index.
* **set**: data type which abstracts the mathematical concept of a set. Basically, it represents an unordered, unbounded collection of unique numbers. Sets support the common set operations, such as membership tests, intersection with another set, union and difference. It is also possible to add single number variables to a set, and even to use them as operands in one of the previously mentioned set operations! A number used as operand along with a set is interpreted as a *singleton*, which is a special type of set containing just a single element.  
  Following the most widespread mathematical convention, sets are declared with identifiers starting with an uppercase letter.

**Syntax and keywords**

The following are language reserved keywords. They cannot be used to define variables, and, in case of operators, they cannot be part of a variable’s name.

|  |  |
| --- | --- |
| **KEYWORD** | **DESCRIPTION** |
| print | Prints the value of a variable, string, expression or a concatenation of any of the previous. Called as print varname; |
| ginseng | If supplied as argument to print, triggers a funny easter egg |
| length(arg) | Retrieve the length of a set or array |
|  |  |
| if | Keyword for conditional construct |
| then | Keyword for conditional construct |
| else | Keyword for conditional construct |
| and | Logical and for conditions |
| or | Logical or for conditions |
| == | Boolean comparison operator for equality |
| != | Boolean comparison operator for difference |
| < > <= >= | Logical comparison operators |
|  |  |
| union | Operator for set union |
| intersection | Operator for set intersection |
| difference | Operator for set difference |
| is\_subset(A, B) | Operator for subset check between sets |
| is\_equal(A, B) | Operator for equality check between sets |
| contains(A, b) | Operator for membership of an element against a set |
|  |  |
| abs | Operator for absolute value |
| % | Operator for modulo operation |
| ! | Operator for factorial |
| ^ | Operator for exponentiation |
| \_ | Operator for concatenation |
| + - \* / | Standard arithmetic operators |

Note: all binary operators are defined as **infix** operators, allowing for a more natural definition. This means, intersection of two set A and B is called by doing C = A intersection B; similarly, to the other operators listed above.  
Set membership tests and length are not operators: they are functions returning a value. They can be used in any expression where a type which match their returning type is expected, but it is required to supply their arguments between parentheses (not as in the case of infix operators).

**Syntactic elements**

A ginseng program is made of source code saved in .g format.  
The line separator is the character “;” and the program terminator is the dot “.” : a valid ginseng program must have its last instruction terminated by it.  
Comments are marked by “//” and span an entire line. They are ignored by the compiler.

**Variables declaration and operations**

**Numeric variable**

A valid **numeric variable** identifier must start with a lowercase letter and is made of alphanumeric characters only. Any expression having a number as result can be assigned to a numeric variable.

a = 1;  
b = -3.1415;  
c = 10+2-1\*4;

**Array syntax**

A valid **array** identifier is made up of any combination of alphanumeric characters enclosed between the so-called diamond: “< >”. Array elements can be any number or expression giving a number as result, and they are enclosed in square brackets and separated by a comma character “,” .  
Array initialization is bounded to the declaration: when the variable is first declared, one must supply at least one member to the array. Such members can be either numbers or numeric variables, but not expression that have not been evaluated yet. There are no empty arrays. However, it is possible to initialize the array to contain as many members as desired.

<arr1> = [1,2,3]; //valid  
<arr2> = [1, 2.12-1, -3/2, 4!, -3.14\*3, 1, 3]; //not valid  
<arr3> = []; //not valid

Elements array can be referenced with the classic zero-indexed bracket indexing and assigned to a numeric variable if needed. Note that this does not remove the element from the array, but it returns a copy of it. If the index is out of bounds, the compiler will raise an error.

<arr> = [1, -12.1, 3]  
num = <arr>[0] //num is now 1  
num2 = <arr>[5] //syntax error

Same notation can also be used to insert an element into the array at a given position, overwriting the element already present. If the index is out of bounds, array is automatically expanded to fit the index specified for the new element. Any element corresponding to indexes between the last element’s index and the newly inserted element is initialized to 0 automatically.  
Element inserted through their index can also be supplied as expression that have still to be evaluated, unlike what happens when an array is initialized.

<arr> = [1,2,3];  
<arr>[7] = 11-2\*4+2! //valid, <arr> is now [1,2,3,0,0,0,0,5]

Length of an array can be obtained through the length() function.

**Set syntax**

A valid **set** identifier starts with an uppercase letter (as the usual mathematical notation requires) and can contain any combination of alphanumeric characters. Similarly, to arrays, they must be declared and initialize to contain at least one element: set elements are listed within curly brackets and are separated by a comma. A set element can be any number or expression with a number as result, and it is possible to provide them in initialization too.

S = {12, 0.6, 0.7+0.3, 1\*1-1, 1/100}; //valid  
S = {}; //syntax error when used as initializer

Differently from arrays, elements can be removed from a set and the result can be the empty set. Note that any operation involving a set will return a result of type set, which should be reassigned to a set variable in order to be stored or reused.

S = {1};  
S difference 1; //syntax error  
S = S difference 1 //valid: 1 is converted to {1} and now S={}

Set operations (union, intersection, difference) can always have as argument either two sets, or a set and a number (except is\_subset and contains).   
To remove one or more elements, set difference is used. To add elements to a set, the union operations is used.   
Set does not allow duplicates by definition: any number which is added to a set, and is already present, is ignored.  
Set must be declared explicitly to be referenced in operations, while single elements can be added without declaring them first.

A = {1,2,3};  
A = A union 4 //A is {1,2,3,4}  
B = {3,4}  
A = A difference B //A is {1,2}  
A = A difference 2 //A is {1}  
A = A union {3,4} //syntax error: use of an undeclared set.  
A = A intersection B //A is now {}

Set membership operations, namely contains, is\_subset, and is\_equal are functions returning a numeric variable that can be interpreted as a boolean (1.000 true, 0.000 false).   
Contains is a membership test of a member (that can be a number, a variable or an expression) against a set, while is\_subset takes two sets as operands. Same goes for is\_equal, which checks equality between two sets.

A = {12, 1.5, -3};  
b = contains(A,12); //b is 1.000  
B = {2, 3, 4};  
c = is\_subset(A, B); //c is 0.000

Cardinality of a set can be obtained with the length() function. Empty set has cardinality 0.

**Flow control statements**

Ginseng offers control flow statements based on the logical *if-then-else* construct. This allows the conditional execution of some statements, based on the evaluation of a boolean condition.

A = {12, 1.1, 3, 4.2};  
b = 8;

if(contains(A, b) == 1){

A = A difference b;

}  
else{

print “Set A does not contain element ” \_ b;

}

print A.

Boolean conditions must be supplied explicitly, i.e. in case of functions the return value must be compared with 1 (true) or 0 (false) as needed. This is not required when evaluating boolean conditions using the standard logical operators (listed in the keyword table).

*if-then-else* structures can also be nested, allowing for arbitrarily complex conditional executions.

A = {1, 2, 3, 4};  
B = {-1, -2, -3, -4};  
a = 10;  
b = 2+1\*3+2;  
c = -1.111;

if(a > 5 and b < a){

if(c < 0){

A = A union a;

}  
 else{

B = A union B;

}

}  
else{

d = a + b + c;  
 print “Result is “ \_ d;

}  
print “Set A is “ \_ A;  
print “Set V is “ \_ B;

Although ginseng statements usually require to be terminated by a semicolon, there is no semicolon after *if-then-else* statements.

**Ginseng grammar**

The grammar of our language is a **context free grammar** where:

* = {P, S, ASSIGNMENT, OP, PRINTABLE, EXP, SET\_EXPRESSION, SETLIST, UNION\_EXP, INT\_EXP, DIFF\_EXP, ARRAY, ELEM, IFSTAT, CONDITION, COND\_EXP, COMPARISON}
* = {. , ; , PRINT , STRING , ARRID , \_ , GINSENG , = , [ , ] , SET , { , } , UNION , INTERSECTION , DIFFERENCE , + , - , \* , / , ^ , % , ! , ( , ) , NUMBER , LENGTH , SUBSET , SETEQUALS , CONTAINS , ID , IF , THEN , ELSE , AND , OR , EQUAL , NOTEQUAL , LESSEQUAL , LESS , GREATEREQUAL , GREATER}
* **P** is the start symbol
* **Q** is the set of productions, that is going to be explained in this section:

**Program:** the program is a sequence of statements and should end with a point.

**Statement:** the sequence of statements of the program should be separated by a semicolon. A single statement could be an assignment, an operation without return value or a control flow statement. Note: statement could be empty due to if statement

**Operation without a return value:** it represents the print statement. What could be printed is an expression, a string, a set, an array, or a concatenation of the previous constructs.

**Assignment:** an assignment statement assigns a value to an id, to an array id, to a particular array element, to a set.

**Set expressions:** the assignment of a set variable could be reached following two methods: specifying a list of elements separated by a comma and embedded into curly brackets, or as a result of set operations (union, intersection and difference).

**Expressions:** the expressions allow to perform a large variety of operation whose result is a number.

**Arrays:** an array is expressed listing elements separated by commas and embedded in square brackets. Elements can be ids or numbers.

**Control flow statements**

**Execution instructions**

**How the input should look like**

Although ginseng is a compiler, it can be accessed as a live interpreter by running it in the console and entering directly the various statements one after the other, similarly to other programming languages. This is useful for debug purposes or to test code snippets. To compile a source file, invoke the ginseng compiler and pass it the source file path as an argument.

> ginseng input\_file.g

The compiler recognizes as valid source files the file ending with a .g extension.

While using ginseng, no external libraries or imports are needed, since all functionalities are already built-in. Therefore, the first line can already contain user’s code. Each statement should be separated by a semicolon and a colon should appear at the end of the code to signal its end. After the control – flow statements “if – then” and “if – then – else”, no semicolons are needed.

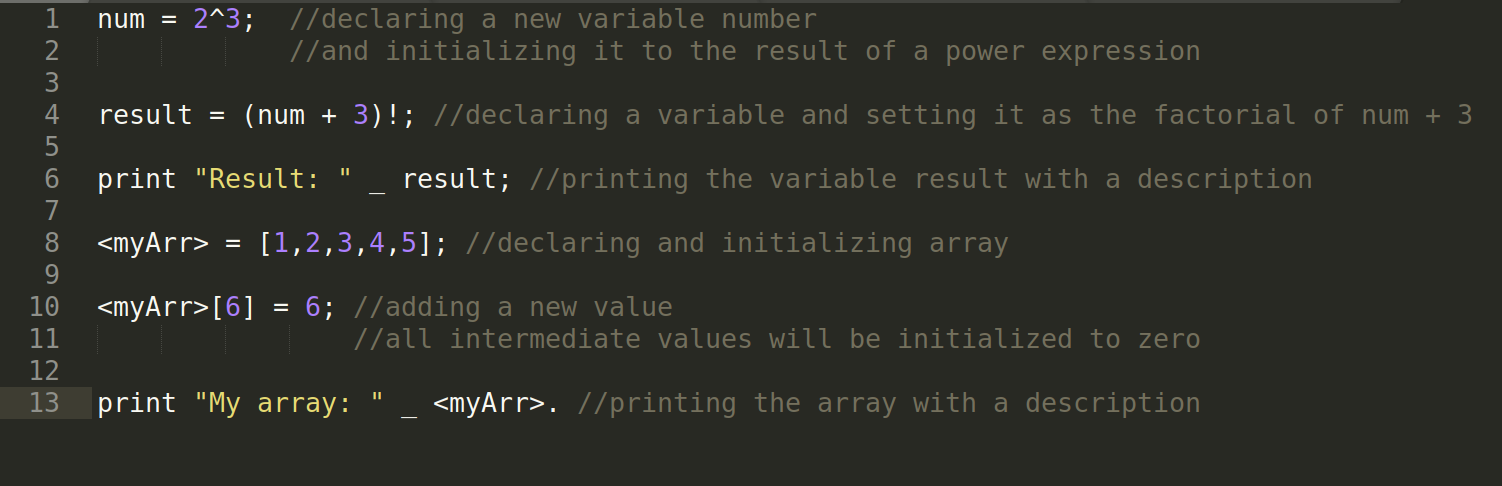
Ginseng comes already with compiled executable files, both for windows and Linux based systems (“ginseng.exe” and “ginseng”), but in case of a new compilation, a makefile is provided to automatize the process. Depending on the operating system it is launched on, this makefile contains directives to compile lex, yacc and .c files and generate a ginseng1.1.0 compiler executable for a Windows, Mac or Unix machine. the root folder of the project, called ginseng, we suggest running the following two commands:

make clean

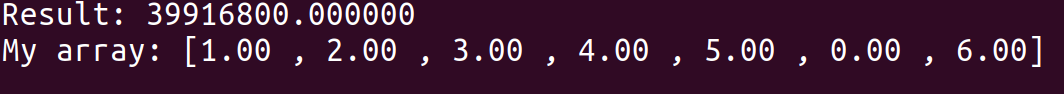
make

**Input code and output examples**

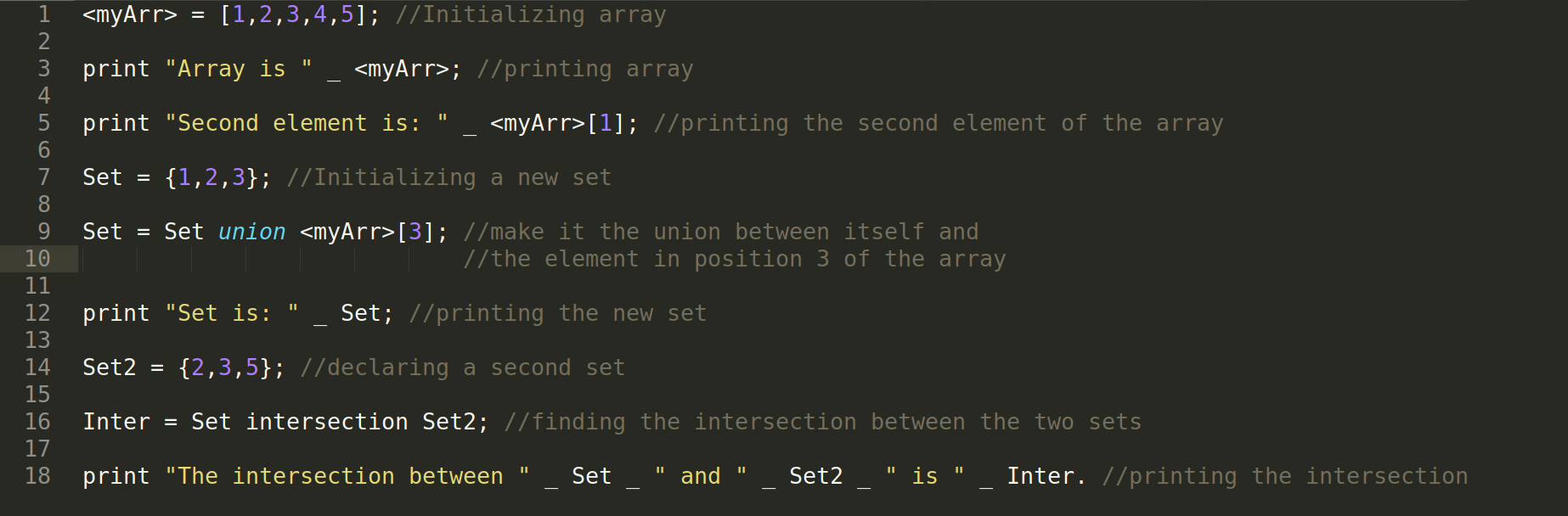
*Input*



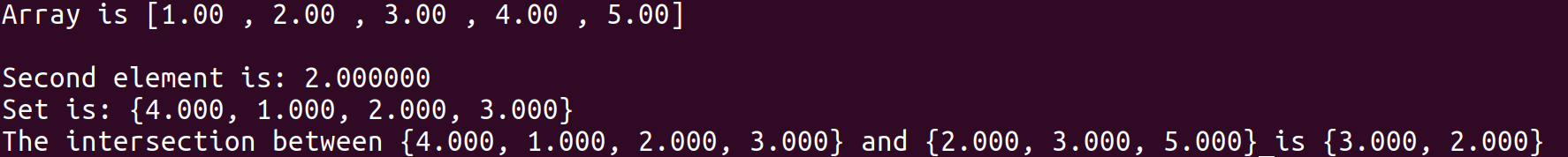
*Output*



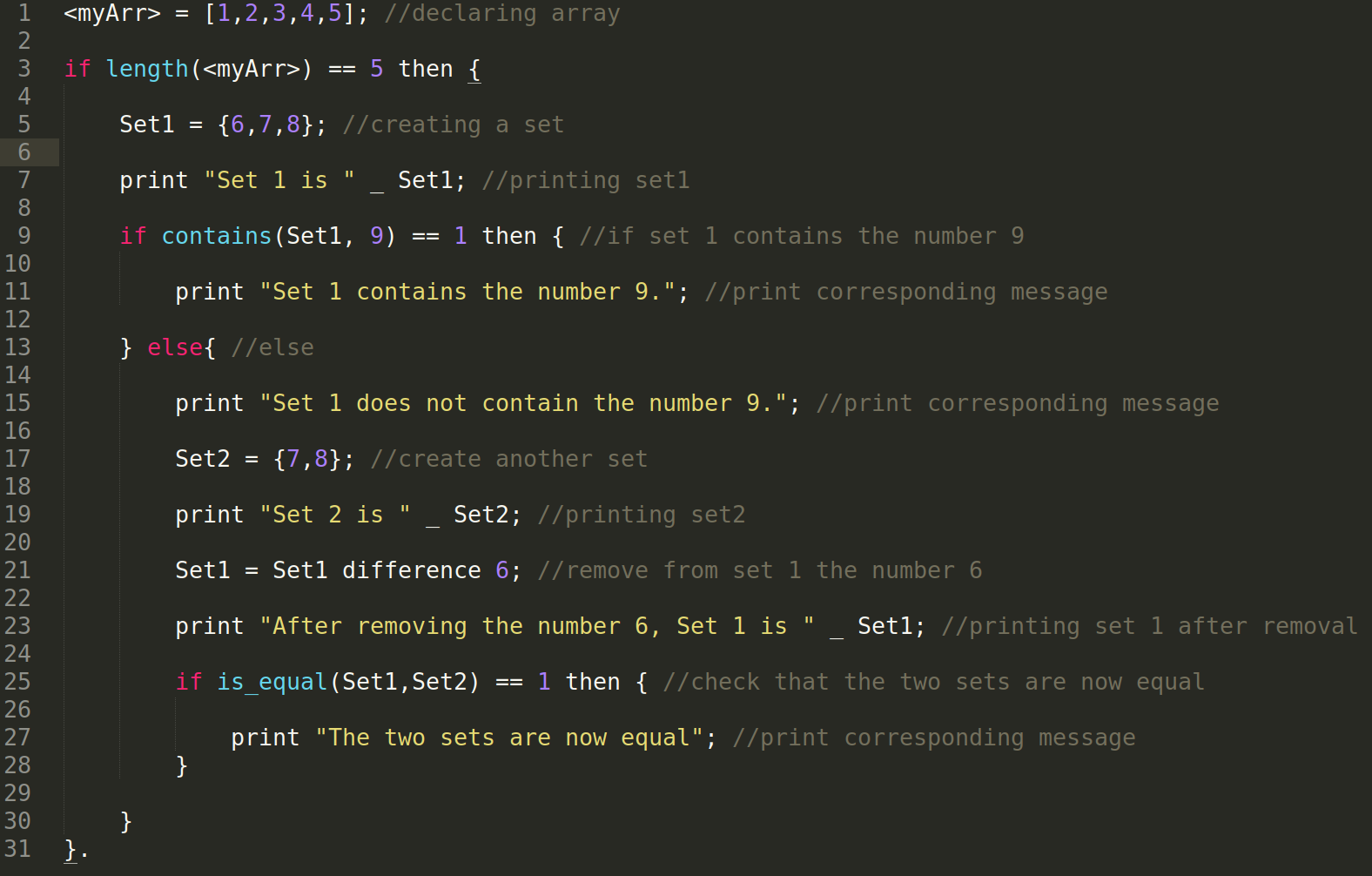
*Input*



*Output*



*Input*



*Output*

