# Chomik

chomik version 0.0.5
document version 0.1

*Pawel Biernacki*

pawel.f.biernacki@gmail.com

Vantaa October 9, 2022

# Chapter 1

# Chomik - a minimalistic programming language

## 1.1 Introduction

In this section we will discuss the difference between the Chomik language and the C/C++ languages. You have got used to the stack, loops, conditional instructions, data structures, classes, inheritance and templates? None of them is actually necessary to write a program in Chomik. In fact none of them **exists** in Chomik, and for good reasons. In spite of that Chomik provides a powerful way of programming, with the implicit loops, implicit conditional instructions and polimorphism. It also provides some constructs that, according to the author's knowledge, do not exist in any other programming language, most notably the recursive enumerations.

### 1.1.1 Identifiers

First we need to get used to the fact that a Chomik "identifier" is only a part of a variable's name. You can write in the traditional style:

```
variable a:integer;
let a=value integer 123;
```

Any time you need to use the value of the variable you need to apply the <> operator:

```
variable a:integer, b:integer;
let a=value integer 123;
let b=<a>;
```

Why the strange syntax? It is a consequence of the chomik's choice to allow multiple identifiers as a variable name. And not only identifiers.

```
variable my age:integer, another age:integer;
let my age=value integer 49;
let another age=<my age>;
```

### 1.1.2 Variable representations

What about the way how we represent the code and data in memory? In the ancient times some language designers determined that the code must be constant and controllable by the parameters only, while the data must be stored in variables. But we do not need to follow the design patterns of the old languages. We can afford something much more revolutionary. First we can store the code in many different ways. We do not necessarily need to follow the pattern of code controlled by the parameters of a function/procedure and the variables (for example integer variables) stored as separate values. Instead we can introduce a built-in type "integer" and sometimes store it in a variable, sometimes (as an "assignment event") for a family of variables or maybe even in some other ways. If we maintain a list of such assignment events than we can easily search it backwards to find the most recent assignment. It is really not part of the language to decide how the interpreter or the compiler is supposed to represent the code and data. Let me show you an example:

```
type width=1..5, height=1..4;

variable alpha_(X:width)_(Y:height):integer;

expand(1);

let variable alpha_(X:width)_(Y:height) = value integer 10;
let variable alpha_2_(Y:height) = value integer 5;
let variable alpha_2_3 = value integer 1;
```

The above three assignments the Chomik interpreter (or compiler) might store in memory in any way, we do not determine it beforehands. The point is that when we access *alpha_2_3* we expect (after the three above assignments) it to have the value 1. In the above example we might simply have 20 separate variables. We call them a **family** of variables. But consider this example:

```
type height=1..4;

variable alpha_(X:integer)_(Y:height):integer;
expand(1);

let variable alpha_(X:integer)_(Y:height) = value integer 10;
let variable alpha_2_(Y:height) = value integer 5;
let variable alpha_2_3 = value integer 1;
```

The difference is that the former **width** is the whole integer type now. Do we need to store it for all possible integers? Of course not. It is enough to store an assignment event in a list of assignment events. But the second and the third assignment can be stored traditionally - as a variable's value. The language only requires that if we access later for example $alpha\_100\_1$ then the interpreter should find the value 10 there.

### 1.1.3 Code representation

In most, if not all, imperative programming languages we got used to the stack and passing parameters to functions or procedures. The code as such is constant, and it is not even a type. This implies the assumption that no operations on code are allowed. While in Chomik indeed there are no operations on code (yet) there could be some in the future! In fact you might consider "execute" to be such an operation, but by "no operations on code" we mean no operations in the meaning of a group theory. For example there is no concatenation of two codes to produce a third one.

```
int do_something(int a, int b, int c)
{
        // some code depending on a, b and c
        printf("a+b+c=%i\n", a+b+c);
        return a+b+c;
}
```

Consider the above function written in C. Think about it in this way - a function definition is a kind of pattern how to "create" a huge amount of "real" functions, if we replace the parameters with their actual values. It is made so by design, no matter how many parameters are passed to the function and no matter how large their sets of possible values are. In a way a function that has parameters is a **family** of various functions. We could imagine replacing the above function by a family of codes without parameters, so that the actual parameters' values become a part of the name.

```
int do_something_0_0_0()
{
        printf("a+b+c=0\n");
        return 0;
}
int do_something_0_0_1()
{
        printf("a+b+c=1\n");
        return 1;
}
int do_something_0_1_1()
{
        printf("a+b+c=2\n");
        return 2;
}
...
```

Instead of calling *do_something*$(1, 2, 3)$ we call in Chomik *do_something*_1_2_3. Not a big change. You might think this just a syntactic sugar. But it is more than that. It gets rid of the unnecessary burden of the parameters. This is precisely how you treat the parameters in Chomik. They are just the items of a variable's name. We do not assume that code must be represented using the parameters and some constant "code literal" as we might call it. We require the Chomik interpreter (or compiler) to choose the optimal way to represent the code. It might be a value of type code stored in a separate variable, or it might be an assignment event, especially if the amount of the variables in the family is very large. The above example would look in Chomik as follows:

```
let variable do_something_(X:integer)_(Y:integer)_(Z:integer)
= value code { execute <print "a+b+c=x">; };
```

This is an example of a very large variable family, but of course the Chomik interpreter will store it as a single assignment event in the assignment event list, which we search then backwards when we want to find a specific code variable.

```
let variable do_something_(X:integer)_(Y:integer)_(Z:integer)
= value code { execute <print "a+b+c=">; };
let variable do_something_(X:integer)_0_0
= value code { execute <print "Hello, y=0 and y=0!">; };
let variable do_something_0_0_0
= value code { execute <print "The parameters are all 0!">; };
```

If we have assignments like above and later call for example *do_something*_0_0_0 then we should see the message "The parameters are all 0!". But the inner representation of the variables in this family will be probably mixed, the first two assignments will be stored in an assignment event list, while the third one will be represented traditionally - as a separate variable. We leave

it at the discretion of an interpreter or a compiler. It is simply NOT a part of the Chomik specification.

### 1.1.4 Class attributes

Now think about the attributes of a class. They are all similar in so far, that for every class instance there is a copy of this particular attribute. They are a **family** of variables.

```
type person={Gotrek, Gwaigilion, Gerrudir};
expand(1);

let variable age_of_(X:person) = value integer 100;
let variable age_of_Gotrek = value integer 150;

let variable strength_of_(X:person) = value integer 10;

let variable wisdom_of_(X:person) = value integer 5;
let variable wisdom_of_Gerrudir = value integer 13;
```

The above Chomik code will create the following variables' representation:

```
Gotrek has age 150, strength 10, wisdom 5
Gwaigilion has age 100, strength 10, wisdom 5
Gerrudir has age 100, strength 10, wisdom 13
```

And you are free to create other "class attributes" this way.

### 1.1.5 Class methods

What about the class methods? They are simply the families of variables of the built-in type code. The literals of this type are embraced in the curly brackets. Note that when we have a literal (a syntactic construct beginning with "value" we explicitly follow the keyword "value" with the type name).

```
type person={Gotrek, Gwaigilion, Gerrudir};
expand(1);

let variable say_hello_(X:person) =
        value code { execute <print "Hello">;};
let variable say_hello_Gotrek =
        value code { execute <print "Hi">;};
```

Now if you call the method *say_hello_Gwaigilion* it will print out "Hello", but *say_hello_Gotrek* will print out "Hi". In Chomik we have polimorphism (we will demonstrate later how it can be applied).

### 1.1.6 Tuples attributes

In C++ there is no such thing as the tuples attributes. One needs to create a special class for the tuples or push the information say about the relations between two persons into the person class, which is a little contrary to the OOP paradigm. In Chomik it is easy:

```
type person={Gotrek, Gwaigilion, Gerrudir};
expand(1);
variable (X:person)_likes_(Y:person):boolean;

let variable (X:person)_likes_(Y:person) = value boolean false;
let variable Gotrek_likes_Gwaigilion = value boolean true;
let variable Gwaigilion_likes_Gotrek = value boolean true;
```

The above code stores the information that Gotrek likes Gwaigilion and Gwaigilion likes Gotrek, but for all the other combinations the $X\_likes\_Y$ has the value $false$. The "boolean" is a built-in type, just like "integer" and "code".

### 1.1.7 Arrays

Now think about the arrays - we intuitively feel they somehow match the concept of an iteration. An array is also a family of variables, which can be accessed using their indeces. If you knew just the type of the index, i.e. its range, then you can almost mechanically write an iteration over the indices within this array. The multidimensional arrays are also families of variables, but they are indexed by tuples. If you knew just their types (ranges) then you could write nested iterations proceeding the array. Or, even better, we could create a programming language, which does the job for you. That is what Chomik is. We do not want to impose the condition on the interpreter whether there should be a sequential loop iterating over the items of an array. We do not care. In next versions we might even allow the code to be executed paralelly, in many threads. (At present we execute it sequentially).

### 1.1.8 Hello world

```
execute <print "Hello world">;
```

The above code, as you might expect, prints out the string "Hello world" to the standard output. Why the strange syntax? What it means? In Chomik code is just a built-in type, with literals that happen to be sequences of statements of the language itself. You can execute it in two ways - either execute a code value directly or execute a variable of the type code. In the above example we use the latter method. This is what the triangular brackets mean in Chomik - whenever you see <something> it means "a value of the variable named - something". But wait a minute - is 'print "Hello world"' a variable? Well - in Chomik it is a predefined variable, or to be precise, all variables with the names beginning with the identifier print are a single predefined family of variables. You could as well write:

```
execute <print "Hello " 1 2 3>;
```

... and Chomik would print "Hello " 1 2 3. In this case the name of a variable is 'print "Hello" 1 2 3'. It is just a name, it has nothing to do with passing the parameters to a code.

You can combine the <> operator, making it nested, for example:

```
execute <print "Hello, Gotrek's strength is" <strength_of_Gotrek>>;
```

The nesting can be as complex as you wish.

### 1.1.9 Names, placeholders and implicit loops

As you have noticed the names in Chomik are slightly different than the identifiers in common programming languages. You can think of the names as a sequence of name items, which can be identifiers, literals (integer, float, string or even code literals) or the placeholders, i.e. special identifiers provided with their type information. A placeholder can be recognized so that it uses a special syntax:

```
execute <print "Hello " (X:boolean) (Y:boolean)>;
```

In the above example we have two placeholders (X and Y) of type boolean. (Boolean is a built-in type).
In the above example we do not simply print "Hello " and $\{false, true\}$ twice as you might have expected. Instead we build a cartesian product of the set $\{false, true\}$, i.e. $\{false, true\} \times \{false, true\}$. Then we calculate how many items are there in the cartesian product. Here we can easily calculate that it contains four items:

```
(false,false)
(false,true)
(true,false)
(true,true)
```

In this case (when the amount of items in the cartesian product is finite and small) we will print out:

```
Hello false false
Hello false true
Hello true false
Hello true true
```

We have just encountered the implicit loops for the first time - this is the preferred way to perform iterations in Chomik.

### 1.1.10 Conditional instruction

As you might have guessed there is no conditional instruction in Chomik. Instead we use the polimorphism that is granted by its powerful concept of a variables' family for the variables of the type "code". There is a built-in type *compare_result* with the possible values *lower*, *equal* and *greater*. There is also a built-in family of variables:

*variable compare "integer"* $(X : integer)(Y : integer) : code$

This operation sets the built-in variable *the compare result* of the type *compare_result*. In order to use the machinery you need to call *compare"integer"*....

```
variable alpha:integer;

let variable alpha = value integer 5;

variable modify_alpha_on_comparison (X:compare_result):code;

let variable modify_alpha_on_comparison (X:compare_result)
= value code {};

let variable modify_alpha_on_comparison lower
= value code { let variable alpha = value integer 100; };

execute <compare "integer" <alpha> 10>;
execute <modify_alpha_on_comparison <the compare result>>;

execute <print "alpha=" <alpha>>;
```

The above code compares the value of the alpha with 10, and if it is lower then assigns alpha with the integer value 100.

### 1.1.11 include

This is a piece of code from an actual Chomik project that uses an "include" construct. It is not a command as such - it is a part of the Chomik's scanner.

```
include "../chomik/user_defined_types.chomik"
include "../chomik/global_variables.chomik"
include "../chomik/global_streams.chomik"
include "../chomik/persons.chomik"
include "../chomik/places.chomik"
include "../chomik/decisions.chomik"
include "./create_new_images.sdl_chomik"
include "./create_new_fonts.sdl_chomik"
include "./on_game_mode.sdl_chomik"
include "./gotrek_options.sdl_chomik"

let variable game mode = value game_mode_type title;

let variable sdl loop body = value code
{
        execute <on game mode <game mode>>;
};

execute <sdl loop>;
```

The above code is written in sdl_chomik, which is a dialect of chomik extended for usage of the libraries SDL2 and SDL2_image. Note that we mix here various files, either written in chomik or in sdl_chomik. This is legal.

### 1.1.12 Simplified syntax

In order to simplify the Chomik syntax we have decided to remove the requirement for a "variable" keyword after "let", and to remove the requirement for the "execute" keyword. You can still use them (they are optional), but you could write:

```
let game mode = value game_mode_type title;

let sdl loop body = value code
{
        <on game mode <game mode>>;
};

<sdl loop>;
```

The modification affects both chomik and sdl_chomik as well as any future extensions that might be created.

## 1.2 Recursive enumerations

Until now we were showing examples how to achieve in Chomik the effect you can achieve in other languages (like C/C++ or Java). Now let us introduce a feature that does not exist elsewhere (according to our knowledge): the recursive enumerations. You could have noticed the command *expand*(1); in our examples. The time has come to reveal its meaning. It is necessary to call it if you want to apply the user-defined types. The integer parameter (which cannot be a <> operator) determines how deeply the recursion of the types should go.

```
type place={Krakow,Warszawa,Wroclaw,Poznan,Gdansk},
person={Conan, Gotrek, Gwaigilion},

action={doing_nothing,
        going_to_(X:place),
        telling_(Y:person)_(X:information),
        asking_(X:person)_to_do_(Y:action),
        asking_(X:person)_whether_(Y:information),
        attacking_(X:person)},

information={(X:person)_is_in_(Y:place),
        (X:person)_is_(Y:action),
        (X:person)_thinks_(Y:information),
        (X:person)_has_told_(Y:person)_(Z:information),
        (X:person)_has_attacked_(Y:person)};

expand(1);

execute <print (X:information)>;
```

The above program will not print anything. If you replace *expand*(1); with *expand*(2); it will print out:

```
Conan_is_in_Krakow
Gotrek_is_in_Krakow
Gwaigilion_is_in_Krakow
Conan_is_in_Warszawa
Gotrek_is_in_Warszawa
Gwaigilion_is_in_Warszawa
Conan_is_in_Wroclaw
Gotrek_is_in_Wroclaw
Gwaigilion_is_in_Wroclaw
Conan_is_in_Poznan
Gotrek_is_in_Poznan
Gwaigilion_is_in_Poznan
Conan_is_in_Gdansk
Gotrek_is_in_Gdansk
Gwaigilion_is_in_Gdansk
Conan_is_doing_nothing
Gotrek_is_doing_nothing
Gwaigilion_is_doing_nothing
Conan_has_attacked_Conan
Gotrek_has_attacked_Conan
Gwaigilion_has_attacked_Conan
Conan_has_attacked_Gotrek
Gotrek_has_attacked_Gotrek
Gwaigilion_has_attacked_Gotrek
Conan_has_attacked_Gwaigilion
Gotrek_has_attacked_Gwaigilion
Gwaigilion_has_attacked_Gwaigilion
```

If you replace *expand*(1); with *expand*(3); then you will get 384 values, including:

```
Gotrek_has_told_Gwaigilion_Conan_is_in_Poznan
```

If you replace *expand*(1); with *expand*(4); then you will get 5253 values, including:

```
Gwaigilion_has_told_Conan_Gotrek_has_told_Conan_Gwaigilion_is_in_Gdansk
```

Note that in the below code we have recursive dependencies - for example some actions depend on other actions or other informations, and some informations depend on other actions or other informations. Using this concept in Chomik we can address the problem of handling situations like "I know that you know that he knows that ...".

```
type place={Krakow,Warszawa,Wroclaw,Poznan,Gdansk},
        person={Conan, Gotrek, Gwaigilion},

    action={doing_nothing,
        going_to_(X:place),
        telling_(Y:person)_(X:information),
        asking_(X:person)_to_do_(Y:action),
        asking_(X:person)_whether_(Y:information),
        attacking_(X:person)},

information={(X:person)_is_in_(Y:place),
        (X:person)_is_(Y:action),
        (X:person)_thinks_(Y:information),
        (X:person)_has_told_(Y:person)_(Z:information),
        (X:person)_has_attacked_(Y:person)};

expand(3);
```

Combining the power of the Chomik's variables' families with the recursive enumerations we can easily create unusually strong code - the code that is difficult to achieve in other, even much more complex languages.

## 1.3 Value of a placeholder

When you ask Chomik to execute a family of code variables you often need to assign a placeholder value to a variable. Think about it as if it were a local variable (but they are NOT, in Chomik all variables are global). This is achieved with the following construct:

```
type place={Krakow,Warszawa,Wroclaw,Poznan,Gdansk};
expand(1);


variable current place: place;

variable do something about the place (X:place):code;
let variable do something about the place (X:place)=value code
{

        let variable current place = value place [(X:place)];
        # we assign the current place variable with the value of X here

        execute <print <current place>>;

        # we could do other operations using the current place here
};
execute <do something about the place (X:place)>;
```

This will print out all the places. If you need to do just that (to print them out), it is more simple to write:

```
type place={Krakow,Warszawa,Wroclaw,Poznan,Gdansk};
expand(1);

execute <print (X:place)>;
```

## 1.4 Built-in constructs

### 1.4.1 Print

There is a family of variables beginning with the word "print". It prints out the values following the identifier "print" to the current output stream. By default it prints to the standard output. Note that we are free to use the placeholders, even many of them, so that our print command is executed for each possible tuple of their values. Example:

```
variable (X:boolean) and (Y:boolean):boolean;
let variable (X:boolean) and (Y:boolean)=value boolean false;
let variable true and true=value boolean true;

execute <print (X:boolean) "and" (Y:boolean)
        "=" <(X:boolean) and (Y:boolean)>>;
```

In the above example we first declare and define a family of variables of type boolean, which include variables "false and false", "false and true", "true and false" - all of them equal "false", and the variable "true and true" which has the value "true". Then we use the magic of the Chomik's implicit iteration to print out the following text:

```
false and false = false
true and false = false
false and true = false
true and true = true
```

BTW. We could as well use different placeholders to define the family of variables and to print it out.

```
variable (X:boolean) and (Y:boolean):boolean;
let variable (X:boolean) and (Y:boolean)=value boolean false;
let variable true and true=value boolean true;

execute <print (A:boolean) "and" (B:boolean)
        "=" <(A:boolean) and (B:boolean)>>;
```

**Variables controlling print**

There are some predefined variables controlling the print's behaviour. We do not have many versions (printf, fprintf, sprintf...). We do printing with a single print instruction, and we can control it setting the predefined integer variable "the print target stream index":

```
# now we redirect print to the standard error stream
let variable the print target stream index=value integer 1;

execute <print (X:boolean) "and" (Y:boolean) "="
        <(X:boolean) and (Y:boolean)>>;
```

Another predefined variable used to control the print family of variables is the string variable "the print separator".

```
# now we change the print separator
let variable the print separator=value string "";

execute <print (X:boolean) "and" (Y:boolean) "="
        <(X:boolean) and (Y:boolean)>>;
```

This will print out the text with the empty separator between the items of the name:

```
falseandfalse=false
trueandfalse=false
falseandtrue=false
trueandtrue=true
```

By default every print ends with an end of line string. It is controlled by the string variable "the print end of line":

```
# now we change the print end of line:
let variable the print end of line=value string "";

execute <print (X:boolean) "and" (Y:boolean) "="
        <(X:boolean) and (Y:boolean)>>;

let variable the print end of line=value string "\n";

execute <print (X:boolean) "and" (Y:boolean) "="
        <(X:boolean) and (Y:boolean)>>;
```

As shown above we can use the "backslash n" construct to restore the original value.

**Print memory report**

The variable "print memory report" prints out the machine report (the types defined) and the memory report (the variables defined and their values). It is useful for the debugging purposes.

### 1.4.2 Input/output

Just like in C there are three predefined streams in Chomik:

1. standard output - stream index 0
2. standard error - stream index 1
3. standard input - stream index 2

If we want to write to a file we first create a stream. We do it with the predefined family of variables of type code prefixed "create". It is somewhat similar to the concept behind the "print" family of variables.

```
execute <create new output filestream "t.txt">;
let variable the print target stream index
        = <the created stream index>;
execute <print "hello2">;    # now we print to the file t.txt
```

The family of variables "create new output filestream 'string'" updates the value of a predefined integer variable "the created stream index". We can store it somewhere, or we can use it directly to assign "the print target stream index".

### Random enums

If we have some enum type defined we can use a similar construct (creating a stream) to create a random generator that provides the places. Note that the type name must be provided as a string.

```
type place={Krakow, Warszawa, Wroclaw};
expand(1);

execute <create new input random enum stream "place">;
let variable my place generator index = <the created stream index>;

...
# this is how we read from a random enum stream:
let variable the read from stream source stream index =
        <my place generator index>;
execute <read from stream "place">;
execute <print <the read from stream result "place">>;
```

This works even for the predefined enums (like boolean). The variable "my place generator index" could have any other name, we just use it to remember the created stream index. After we execute 'read from stream "place"' the value of the automatically created variable 'the read from stream result "place"' will be assigned for us. We can use it afterwards. It is just a simple variable.

### Random integers

There is a similar construct for created the integer random generators:

```
execute <create new input random number stream "integer" 1 10>;
let variable my strength generator index =
        <the created stream index>;
```

In the above example we create a random number generator for the built-in type integer (again we need to provide the type name in quotation marks) beginning from the number 1 and ending with 10 - inclusively.

```
# this is how we use it:
let variable the read from stream source stream index =
        <my strength generator index>;
execute <read from stream "integer">;
execute <print <the read from stream result "integer">>;
```

The variable "my strength generator index" could have any other name.

### 1.4.3 Arithmetic operations

Chomik supports integer addition and substraction (at present). Each of the operations consists of executing a code variable from a built-in family of variables (add or substract). The "add" operation sets an integer predefined variable *'the add result "integer"'*, and the "substract" operation sets an integer predefined variable *'the substract result "integer"'*.

```
execute <add "integer" <current option number> 1>;
let variable current option number =
        <the add result "integer">;
```

The above two operations mean we increase the variable "current option number" by 1.

```
execute <substract "integer" <current option number> 1>;
let variable current option number =
        <the substract result "integer">;
```

The above two operations mean we decrease the variable "current option number" by 1.

### 1.4.4 Comparing

There is a predefined type "compare_result" with the possible values being 'lower', 'equal' and 'greater'. There is also a predefined variable "the compare result" of this type.

#### Comparing enums

Let us take a look at a code taken from an actual project:

```
type person={Gotrek, Gwaigilion, Gerrudir},
place={Krakow, Warszawa, Wroclaw},
action={no_action, goto (X:place)};
expand(2);

let variable decision of (X:person)=value action no_action;


variable current person goto current place (X:compare_result):code;
let variable current person goto current place lower=value code {};
let variable current person goto current place greater=value code {};
let variable current person goto current place equal=value code
{
let variable location of <current person>=<current place>;
execute <print <current person> "has moved to"
        <location of <current person>>>;
let variable decision of <current person>=value action no_action;
};

variable perform current person decision goto (A:place):code;
let variable perform current person decision goto (A:place)=value code
{
let variable current decision = <decision of <current person>>;
let variable current place = value place [(A:place)];
let variable current pattern = value action goto <current place>;

execute <compare "action" <current decision> <current pattern>>;
execute <current person goto current place <the compare result>>;
};

variable perform decision for (X:person):code;
let variable perform decision for (X:person)=value code
{
let variable current person = value person [(X:person)];
execute <print "perform decision for" <current person>>;
execute <perform current person decision goto (A:place)>;
};

variable perform all the decisions:code;
let variable perform all the decisions=value code
{
execute <print "performing all the decisions">;
execute <perform decision for (X:person)>;
};


# here we assign the value of 'decision of Gotrek' to:
# goto Krakow

let variable decision of Gotrek=value action goto Krakow;

# ...
# somewhere else in the code we execute all decisions of all persons:
execute <perform all the decisions>;
```

This will print out:

```
performing all the decisions
perform decision for Gotrek
Gotrek has moved to Krakow
perform decision for Gwaigilion
perform decision for Gerrudir
```

### Comparing integers

The below code comes from an actual project written in Chomik. It is defining two families of variables of type code. For each compare result we have three options (lower, equal, greater) therefore we have six assignments:

```
variable move current option number down if (X:compare_result):code;
let variable move current option number down if equal=value code
{
        let variable current option number = value integer 1;
};
let variable move current option number down if lower=value code
{
        execute <add "integer" <current option number> 1>;
        let variable current option number =
                <the add result "integer">;
};
let variable move current option number down if greater=
        <move current option number down if equal>;


variable move current option number up if (X:compare_result):code;
let variable move current option number up if equal = value code
{
        let variable current option number = value integer 5;
};
let variable move current option number up if lower =
        <move current option number up if equal>;
let variable move current option number up if greater=value code
{
        execute <substract "integer" <current option number> 1>;
        let variable current option number =
                <the substract result "integer">;
};
```

When we increase the "current option" we use the above code as follows:

```
execute <compare "integer" <current option number> 5>;
execute <move current option number down if <the compare result>>;
```

When we decrease the "current option" we use it as follows:

```
execute <compare "integer" <current option number> 1>;
execute <move current option number up if <the compare result>>;
```

### 1.4.5 The program return value

In order to write tests it is useful to control the value that is going to be returned by the program. There is a variable called "the program return" of type integer, by default equal 0, which can be assigned for that purpose.

```
let the program return = value integer 1;
# this program is going to return 1
```

You can assign it conditionally (using the comparison), so that a test can fail if say, the addition does not work properly.

```
#!/usr/local/bin/chomik

# this test checks the addition of integers

variable return 1 on wrong result (X:compare_result_type):code;
let return 1 on wrong result equal=value code
{
        let the program return = value integer 0;
};
let return 1 on wrong result lower=value code
{
        let the program return = value integer 1;
};
let return 1 on wrong result greater=value code
{
        let the program return = value integer 1;
};


<add "integer" 2 2>;

<compare "integer" <the add result "integer"> 4>;
<return 1 on wrong result <the compare result>>;
```

### 1.4.6 The break flag

When running an infinite loop you will want eventually to stop it. This can be achieved by assigning the built-in boolean variable "the break flag" to false. The below code iterates from 0 to 10:

```
#!/usr/local/bin/chomik

# this test checks the "infinite" loop
# it sets the break flag after the comparison of the counter
# indicates it is equal 10


variable break on (X:compare_result_type):code;
let break on equal=value code
{
        let the break flag = value boolean true;
};
let break on lower=value code {};
let break on greater=value code
{
        let the break flag = value boolean true;
        # this assignment is not necessary!
        # it is just to be on the safe side...
};


variable do something (X:integer):code;
let do something (X:integer)=value code
{
        let counter = value integer [(X:integer)];

        <print "the counter equals" <counter>>;

        <compare "integer" <counter> 10>;
        <break on <the compare result>>;
};

<do something (X:integer)>; # here we run the iteration

let the break flag=value boolean false;
# after we did the loop it is a good habit to reset the break flag
# the break flag is a predefined boolean variable
# by default it equals false
```

Note that the same global "the break flag" works for all loops, it is not a local variable. Keep this in mind when nesting infinite loops.

## 1.5 Summary

Chomik is a minimalistic language with the following types of instructions:

1. type declaration

2. variable declaration
3. expand command
4. assignment command (let variable ...)
5. execute command

# Chapter 2

# Writing libraries in Chomik

When you write a library in Chomik you usually want to provide its source code under a GPL3 license.

## 2.1 Logical operators

Although Chomik does not support say logical operators, we can easily add them using its powerful "family of variables" paradigm. We have for example the built-in type "boolean". Imagine we want the operation "not":

```
variable not (B:boolean):boolean;

let not false=value boolean true;
let not true=value boolean false;
```

With this definition in place we can use the "not" operator when constructing the names of variables. For example:

```
variable a:boolean;

...

<print "a=" <a>>
<print "not a=" <not <a>>>;
```

Using the same concept you could create infix (or prefix, if you prefer) operators "and", "or", "implies" and so on.

## 2.2 Successor

You may wish to have operations like "successor" or "predecessor" in place. Imagine you define a new type "place" and add a family of variables based on this type:

```
type place={Krakow, Warszawa, Wroclaw, Poznan, Gdansk};
extend(1);

variable (X:place) successor:place;

let Krakow successor=value place Warszawa;
let Warszawa successor=value place Wroclaw;
let Wroclaw successor=value place Poznan;
let Poznan successor=value place Gdansk;
let Gdansk successor=value place Krakow;
```

You could then easily iterate using this family of variables.

```
variable x:place;
...
<print "x=" <x>>;
<print "x successor=" <<x> successor>>;
<print "x successor successor=" <<<x> successor> successor>>;
```

# Chapter 3

# Extending chomik

It is possible to embed chomik in your own C++ projects creating new Chomik-like languages. The chomik package itself contains an example of such language, which is sdl_chomik. Feel free to look at the code (inc/sdl_chomik.h and src2/sdl_chomik.cc).

In order to execute an sdl_chomik program we need to do the following:

```
sdl_chomik::machine m;
m.create_predefined_types();
m.create_predefined_variables();
m.create_predefined_streams();

the_program.execute(m);


chomik::generic_name gn;
gn.add_generic_name_item(
        std::make_shared<chomik::identifier_name_item>("the"));
gn.add_generic_name_item(
        std::make_shared<chomik::identifier_name_item>("program"));
gn.add_generic_name_item(
        std::make_shared<chomik::identifier_name_item>("return"));
chomik::signature s0{gn};

return_value = m.get_variable_value_integer(s0);
```

The "return_value" should be then returned by the main function itself. Note that the machine we are using for an execution is NOT chomik::machine, but sdl_chomik::machine instead. It is, however, inherited from chomik::machine. The program must be first declared and initialized:

```
chomik::program the_program;
chomik::parser the_parser{the_program};

...

if (the_parser.parse(argv[1]) == 0)
{
        # execute the program
}
```

The sdl_chomik::machine overrides three functions from its base class chomik::machine:

1. create_predefined_variables
2. get_is_user_defined_executable
3. execute_user_defined_executable

Note that in the **sdl_chomik::machine::create_predefined_variables** we invoke the original **chomik::machine::create_predefined_variables** so that sdl_chomik has all the predefined variables from chomik (since it is an extension).

```
void sdl_chomik::machine::create_predefined_variables()
{
        chomik::machine::create_predefined_variables();
        ...
}
```

# Chapter 4

# sdl_chomik

The sdl_chomik is a small extension of chomik, i.e. it is a language based on chomik and adding to it a (limited) functionality of the SDL2 and SDL2_image libraries. It is delivered with the chomik itself. While in the former chapter we have written a little about how to create your own extensions on its example - we are going to introduce sdl_chomik from a user's perspective now.

## 4.1 Built-in constructs

There are some built-in entities in sdl_chomik made to support the 2D graphic programming.

### 4.1.1 sdl loop

The entry point (sdl loop) is a predefined variable.

```
execute <sdl loop>;
```

The above code is the simplest program written in sdl_chomik. It invokes sdl loop. It simply shows a black window with the hard-coded size of $800 \times 600$.

### 4.1.2 sdl loop body

The "sdl loop body" is a predefined variable of type code. Its value is originally an empty code. You are supposed to assign it with the actual sdl_chomik code so that the "sdl loop" knows what to do.

```
let variable sdl loop body = value code
{
        execute <print "hello">;
};

execute <sdl loop>;
```

The above code opens a black window, and then continuously prints out "hello" to the standard output. The text appears on the terminal screen, not on the created SDL screen. How to show an image on the SDL window instead?

### 4.1.3 Images

In order to show an image you first need to read it from a file. This is done by extended family of variables with the prefix "create". But didn't we use it already in chomik? That is correct. The sdl_chomik does not remove the old chomik functionality, but adds a new one. You can still create random generator streams or any other streams as if you would in chomik, but in addition you can create images. For example in order to create an image stored in the folder "../image/title.png" we need to run:

```
execute <create new image "../image/title.png">;
variable title image index: integer;
let variable title image index =<the created image index>;
```

Just as for the streams, there is a predefined integer variable "the created image index", which is sdl_chomik specific. It is an integer variable assigned by the "create new image '...'" family of variables. You usually need to store it in some specific variable of yours, for later use. The complete working example in sdl_chomik would be:

```
execute <create new image "../image/title.png">;
variable title image index: integer;
let variable title image index =<the created image index>;

let variable sdl loop body = value code
{
    execute <show image <title image index> 0 0>;
};

execute <sdl loop>;
```

This code reads a PNG image from the file "../image/title.png" and uses a predefined "show image" family of variables. After the "show image" you should pass three values:

1. the index of the image to display
2. the x coordinate
3. the y coordinate

It is customary in chomik to read a hamster image (delivered with the chomik) and to put it on the screen in the left upper corner, for example like that:

```
execute <create new image "../image/title.png">;
variable title image index: integer;
let variable title image index = <the created image index>;

execute <create new image "../image/chomik.png">;
variable chomik image index: integer;
let variable chomik image index = <the created image index>;


let variable sdl loop body = value code
{
        execute <show image <title image index> 0 0>;
        execute <show image <chomik image index> 0 0>;
};

execute <sdl loop>;
```

You are free to add the hamster image to your projects (it is public domain). The word "chomik" means "hamster" in Polish.

### 4.1.4 Fonts

In sdl_chomik you can also load TTF fonts. You have to use a predefined family of variables "create new font ...". It should be followed by the TTF font file name and the font size. It assignes the sdl_chomik-specific predefined variable "the created font index". You can later use it to show text.

```
execute <create new font "../font/PlayfairDisplay-Bold.ttf" 32>;
variable my large font index:integer;
let variable my large font index = <the created font index>;

execute <create new image "../image/title.png">;
variable title image index: integer;
let variable title image index = <the created image index>;

execute <create new image "../image/chomik.png">;
variable chomik image index: integer;
let variable chomik image index = <the created image index>;


let variable sdl loop body = value code
{
execute <show image <title image index> 0 0>;
execute <show image <chomik image index> 0 0>;
execute <show text <my large font index> "hello world" 100 100>;
};

execute <sdl loop>;
```

In order to show the text you can use a predefined sdl_chomik-specific "show text" family of variables. You have to pass the font index (obtained from "the created font index"), the text itself and the coordinates. The Playfair fonts are delivered with chomik and licensed:

```
This Font Software is licensed
under the SIL Open Font License, Version 1.1.
This license is available with a FAQ at:
http://scripts.sil.org/OFL
```

The Playfair font license is also included in the chomik package (file chomik/font/OFL.txt).

## 4.2 License

chomik and sdl_chomik are licensed under GPL 3.0, with the exceptions of the images (public domain) and the fonts.