**University of Amsterdam**

*Programming Research Group*

# Molecular Scripting Primitives

B. Diertens

Report PRG0401 June 2004

B. Diertens

Programming Research Group
Faculty of Science
University of Amsterdam

Kruislaan 403
1098 SJ   Amsterdam
The Netherlands

tel. +31 20 525.7593
e-mail: bobd@science.uva.nl

Programming Research Group Electronic Report Series

# Molecular Scripting Primitives

*Bob Diertens*

Programming Research Group, Faculty of Science, University of Amsterdam

*ABSTRACT*

The notations so far in program algebra are too low level to program in easily. In this article, two new basic instruction sets are introduced to solve this. One that contains more complex instructions, which can be seen as a shorthand for several instructions in an existing basic instruction set. And one that extends the other with instructions that operate on values. It is shown how these basic instruction sets can be mapped onto existing basic instruction sets.

*Keywords:* program algebra, scripting, programming, programming languages

## 1. Introduction

In [1], Program Algebra is introduced (see also section 1.1). Here, simple program notations are given which are build up from primitive instructions with as parameter a set of basic instructions that perform some operation on a environment. The program notations can be mapped onto each other. This gives us a hierarchy of program notations in which the more complex ones can be mapped onto the simpler ones.

The program notations so far, are too low level to program in easily. We set out to get at a level that is still primitive, but also acceptable as a programming language. We do this by developing basic instruction sets on top of existing basic instruction sets, and which can be translated to these lower level basic instruction sets (in combination with a primitive instruction set).

Our start point are the Molecular Programming Primitives (MPP) described in [2] (see section 1.2 for a short description). There, also an extension with values of type booleans, and integers is described (MPPV). But no operations on these values are added, so that for manipulation of the values large chunks of code are needed that have to be repeated when the same operation has to be done elsewhere in the program.
If we use a primitive instruction set which supports methods, such as PGLEcm, the code for the operations on values only appears once, but performing these operations takes a lot of time.

We add values of type string to MPPV and raise it to a higher level by adding some stronger instructions for manipulating molecules, which we call High-level MPPV (HMPPV). We use this as a base for adding powerful instructions on values, and so reach a level which is acceptable for programming. These powerful instructions can be considered replacements for large chunks of code, or scripts, and hence we call them Molecular Scripting Primitives (MSP).
We show how to translate HMPPV to MPPV and MSP to HMPPV. And to show the usefulness of MSP, we give a projection of PGLB to PGLA in PGLEc and MSP in Appendix A.

In order to be able to execute arbitrary code (in the form of a string), we extend MSP with instructions for evaluation, which we call MSPea. The code will be compiled into a molecule which can be evaluated. One of the instructions can not be translated to a lower form. For the others we show how to do this.

In the remainder of this chapter we give a short overview of program algebra, and molecular dynamics.

## *1.1 PGA*

PGA is an algebraic framework for sequential programming. It is intended to contribute to a better understanding of sequential programming. A very simple program notation is used as basis for development of other program notations.

The syntax of program expressions in PGA is generated from a set of constants, the primitive instructions, and two composition mechanisms. The primitive instructions have as parameter a set of basic instructions. These basic instructions can be viewed as requests to an environment to provide some service. Upon execution, a basic instructions returns a boolean value.

Primitive instructions:

- a basic instruction: `a`
  - After performing the basic instruction `a` execution continues with the next instruction.
- termination: `!`
  - Execution stops.
- positive test: `+a`
  - If the basic instruction `a` returns true, execution is continued with the next instruction. If it returns false, the next instruction is skipped.
- negative test: `-a`
  - If the basic instruction `a` returns false, execution is continued with the next instruction. If it returns true, the next instruction is skipped.
- forward jump: `#k`
  - The instruction itself and the following `k - 1` instructions are skipped. If `k` is `0`, a jump to the instruction itself is made.

Compositions:

- concatenation of `X` and `Y`: `X;Y`
- repetition of `X`: $X^{\omega}$

PGLA is a program notation for representing PGA expressions. For dealing with repetition, PGLA has an additional primitive instruction, which replaces repetition.

- repeat: `\\#n`
  - Here, `n` is a natural number greater than zero. A program text ending with this instruction will repeat its last `n` instructions, excluding the repeat instruction itself.

On top of PGLA, other program notations are designed, which can be mapped on PGLA. Such a mapping towards PGLA is called a projection. Several projections to intermediate languages may be used to get the result that is needed. A mapping away from PGLA is called an embedding.

## *1.2 Molecular dynamics*

In molecular dynamics the memory state of a system is modeled as a fluid consisting of a collection of atoms which may have bindings between them to form molecules. A molecule consists of a number of atoms all reachable from one of the atoms by sequences of directed links. A directed link from one atom to another atom exists if the former has a so-called *field* containing the latter. By means of actions causing a change of state, fields can be added to and withdrawn from atoms, and contents of fields can be modified. Selected atoms can also be brought into *focus*.
The basic instruction set is called MPP (Molecular Programming Primitives). There is also a version with values (MPPV). Both have been extended with garbage collection instruction as described in [2].

We give here an overview of the instructions of MPP. Where not mentioned, an instruction returns `true`, except when a failure occurs, in which case it returns `false`.

- `x = new`
  - Create a new atom and assign it to `x`.
- `x.+f`

Add a field `f` to the atom in focus `x`. If this atom already has a field `f` then `false` is returned.

`x.-f`
: Remove the field `f` from the atom in focus `x`. If this atom does not have a field `f` then `false` is returned.

`x.f = y`
: Assigns the atom in focus `y` to the field `f` of the atom in focus `x`. If the atom in focus `x` does not have a field `f` then `false` is returned.

`x = y.f`
: Assign to atom in field `f` of the atom in focus `y` to the focus `x`. If the atom in focus `y` does not have a field `f` then `false` is returned.

`x = y`
: Assign the atom in focus `y` to the focus `x`.

`x == y`
: Return `true` if the foci `x` and `y` contain the same atom, and `false` otherwise.

`x/f`
: Return `true` when the atom in focus `x` has a field `f`, and `false` otherwise.

In addition to the instructions of MPP, MPPV contains the following instructions.

`x.+f:t`
: Add a value field `f` of type `t` for an atom in focus `x`. Possible types are integer (non negative numbers) and boolean. The value will be initialized by `0` in the case of integer, and `false` in the case of boolean.

`x.f = u`
: Assign the value `u` to the field `f` of the atom in focus `x`. The value must be of the appropriate type, otherwise `false` is returned.

`x.f = y`
: Assign the value in focus `y` to the field `f` of the atom in focus `x`. The value must be of the appropriate type, otherwise `false` is returned.

`x = u`
: Assign the value `u` to the atom in focus `x`.

`x = y`
: Assign the value in focus `y` to the atom in focus `x`.

`x == u`
: Return `true` if the value in focus `x` equals the value `u`.

`x == y`
: Return `true` if the value in focus `x` equals the value in focus `y`.

## 2. High-level MPPV

HMPPV is based on MPPV (Molecular Programming Primitives with Values). It has some instructions that are a shorthand for commonly used combinations of instructions in MPPV. Furthermore, in addition to the type of values provided by MPPV, it has strings. We describe here the instruction set of HMPPV, and give some examples of projecting HMPPV instructions to PGLEc and MPPV instructions. We also show how we can represent values as molecules in MPP and so justify their introduction in MPPV and HMPPV.

### *2.1 Instruction set*

With the instructions described here, *`extfocus`* (extended focus) is used to denote either a focus, or a field selection that may also be compound. When more than one *`extfocus`* is used in an instruction, they are followed by a number. A field selection that does not exist results in a failure of the instruction. Where not mentioned, an instruction returns `true`, except when a failure occurs, in which case it returns `false`. To make the description of the instructions easier, we consider an atom a value of type atom.

*`extfocus`* `= new`
: Create a new atom and assign it to *`extfocus`*. When *`extfocus`* is a field selection, it must be

of type atom.

`extfocus1 = extfocus2`
: Assign the value in `extfocus2` to `extfocus1`. When `extfocus1` is a field selection, the field must be of the same type as the value in `extfocus2`.

`extfocus.+f`
: Add a field of type atom to the atom selected by `extfocus`.

`extfocus.+f = new`
: Add a field of type atom to the atom selected by `extfocus`, and assign it a newly created atom.

`extfocus1.+f = extfocus2`
: Add a field of type atom to the atom selected by `extfocus1`, and assign it the atom selected by `extfocus2`.

`extfocus.+f:t`
: Add a field of type `t` to the atom selected by `extfocus`.

`extfocus.+f:t = u`
: Add a field of type `t` to the atom selected by `extfocus`, and assign it the value `u`.

`extfocus1.+f:t = extfocus2`
: Add a field of type `t` to the atom selected by `extfocus1`, and assign it the value selected by `extfocus2`.

`extfocus.-f`
: Removes the field `f` from the atom selected by `extfocus`. Returns `false` when the atom has no field `f`.

`extfocus/f`
: Returns `true` when the atom selected by `extfocus` has a field `f`, and `false` otherwise.

`extfocus1 == extfocus2`
: Comparison of the values selected by `extfocus1` and `extfocus2`. Returns `true` when the values selected by `extfocus1` and `extfocus2` are the same or contain the same atom, and `false` otherwise.

`extfocus?`
: Returns `true` when the value selected by `extfocus` is an atom, and `false` otherwise.

`extfocus?t`
: Returns `true` when the value selected by `extfocus` is of the type `t`, and `false` otherwise.

The type of a value field can be one of bool, int, or str, representing boolean, integer, or string. Boolean values can either be `true` or `false`. An integer must be a non-negative number. In notation, string values must be delimited by double quotes ("), and a backslash (\) and a double quote inside a string must be preceded (escaped) by an additional backslash. For the moment, we restrict the set of characters that may appear in a string to the alphanumeric, the punctuation characters, the space, and the newline in the form of \n.

## *2.2 Projection of HMPPV to MPPV*

We only give here projections for some of the instructions of HMPPV. Projections of the other instructions can be done in a similar manner. For the projections we use PGLEc as primitive instruction set.

The instruction `x.f.+g:int = y.f.g` can be translated to MPPV as follows.

```
+ y/f {;
    hy = y.f;
    + hy/g {;
        hy = hy.g;
        + x/f {;
            hx = x.f;
            - hx/g {;
                hx.+g:int;
                hx.g = hy;
                };
            };
        };
    }
```

If the same instruction appears in the context of an if construction (`+ x.f.+g:int = y.f.g {; ... }`), the return value is of importance. In this case the translation should be done as follows.

```
    result = false;
    + y/f {;
        hy = y.f;
        + hy/g {;
            hy = hy.g;
            + x/f {;
                hx = x.f;
                - hx/g {;
                    hx.+g:int;
                    + hx.g = hy {;
                        result = true;
                    };
                };
            };
        };
    };
    + result == true {;
        ...
    }
```

The instruction to check the type of an extfocus, such as `x?int`, in the context of an if construction can be translated as follows.

```
    h = new;
    h.+f:int;
    + h.f = x {;
        ...
    }
```

If `x` is not of the type int, the assignment to `h.f` fails, and so it can be used as a test.

## *2.3 Molecular representation of values*

In this section we show that it is not necessary to add values to our basic instruction sets for molecular dynamics, but that we can represent values as molecules.

### *2.3.1 Booleans*

The boolean values true and false can be represented by 2 foci with the names `true` and `false`, initialized by the instructions

```
    true = new;
    false = new
```

Assignment and comparison of the foci true and false to foci or field selections do work as expected in this way. However, nothing prevents from new assignment to the foci true and false, which may cause unexpected behavior of programs.

*2.3.2 Naturals/Integers*

A natural number can be represented in binary form by a list of booleans. For instance the natural number 5 can be represented by the list shown in Figure 1.

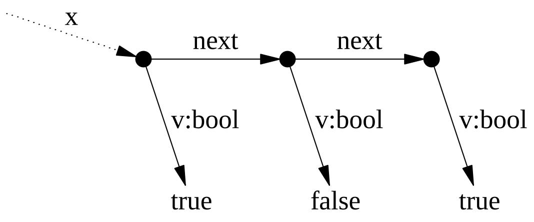

**Figure 1.** Representation of the number 5 as a list of booleans

If we introduce foci for the numbers 0 through 9 we can also represent numbers in a decimal way. However, names of foci starting with a number are not allowed. Operations on numbers can be implemented by operations on a list. Below we show the code for incrementing the value in focus `x` (the list starts with the low bit) in PGLEc with HMPPV as basic instruction set (but only using booleans as value).

```
h = x;
L0;
+ h.v == true {;
    h.v = false;
    + h/next {;
        h = h.next;
        ##L0;
    }{;
        h.+next = new;
        h.next.+v:bool = true;
    };
}{;
    h.v = true;
}
```

To make the more complex operations easier, we can represent numbers by a double linked list with links to the fi rst and last atom.

*2.3.3 Strings*

A string is just a list of characters and a character can be represented by a number. So, strings can easily be represented by a list of numbers (see Figure 2). And as with numbers, operations on a string can be implemented by operations on a list.

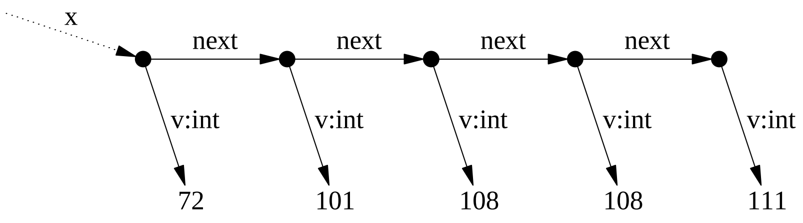

**Figure 2.** Representation of the string 'Hello' as a list of numbers (ASCII)

For example, the following program appends the string in focus `y` to the string in focus `x`.

```
h = x;
L0;
+ h/next {;
```

```
        h = h.next;
        ##L0;
    };
    h.+next;
    h.next = y
```

Note that the string in `y` is shared after execution of the program. But it is easy to implement copying of the string in focus `y`.

## 3. Molecular Scripting Primitives

MSP is an extension of HMPPV. Here, we only describe the instructions that are part of the extension. We give an example of a program which uses MSP to show that this can be done easily. Further, we indicate how MSP can be translated to HMPPV.

### *3.1 Instruction set*

With the instructions described here, an `extfocus` is used to denote either a focus, or a field selection that may also be compound. A field selection that does not exist results in a failure of the instruction. A value taken from a focus and all field selections must be of the proper type, otherwise the instructions fails. Where not mentioned, the instructions return `true`, except when a failure occurs, in which case it returns `false`.

Operations on numbers:

`incr` *`extfocus`*
: Increments the integer value selected by *`extfocus`* with `1`.

`incr` *`extfocus n`*
: Increments the integer value selected by *`extfocus`* with *`n`*.

`incr` *`extfocus1 extfocus2`*
: Increments the integer value selected by *`extfocus1`* with the integer value selected by *`extfocus2`*.

`decr` *`extfocus`*
: Decrements the integer value selected by *`extfocus`* with `1`. Return `false` if the value was already `0`.

`decr` *`extfocus n`*
: Decrements the integer value selected by *`extfocus`* with *`n`* if the value is larger than or equal to `n`. Returns `false` otherwise.

`decr` *`extfocus1 extfocus2`*
: Decrements the integer value selected by *`extfocus1`* with the integer value selected by *`extfocus2`* if it is larger than or equal to the value it must be decremented with. Returns `false` otherwise.

Operations on strings:

`first` *`extfocus1 extfocus2`*
: Takes the first character of the string selected by *`extfocus1`* and assign it as a string of length 1 to *`extfocus2`*. If the selected string is the empty string, `false` is returned.

`delfirst` *`extfocus`*
: Removes the first character from the string selected by *`extfocus`*. If the string is the empty, `false` is returned.

`append` *`extfocus1 extfocus2`*
: Appends the string selected by *`extfocus2`* to the end of the string selected by *`extfocus1`*.

Type conversions:

`int` *`extfocus1 extfocus2`*
: Converts the string selected by *`extfocus1`* to an integer value and assigns it to *`extfocus2`*. If no integer value can be obtained from the string, `false` is returned.

`str` *`extfocus1 extfocus2`*
: Converts the integer value selected by *`extfocus1`* to a string and assigns it to *`extfocus2`*.

## *3.2 Projection of PGLB to PGLA in PGLEc.MSP*

To show that MSP is strong enough to write a program in an easy way, we give a projection from PGLB to PGLA in PGLEc with MSP. The code can be found in Appendix A.

## *3.3 Translation of PGLEc.MSP to PGLEcm.HMPPV*

For the translation of MSP to HMPPV we use PGLEc as source primitive instruction set and PGLEcm as target primitive instruction set, since it contains methods. The PGA Toolset [3][1] contains projections from PGLEcm to PGLEcr and from PGLEcr to PGLEc, which brings us back to the source primitive instruction set.

In order to translate MSP to HMPPV we need a library of methods that implement operations on the molecular representation of values. We are not going to describe an implementation of such a library, but we assume values are implemented as depicted in Figure 3. Here, the actual values can be implemented as described in 2.3.

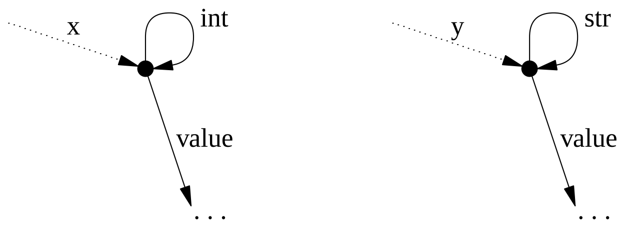

**Figure 3.** Focus `x` with a value of type int, and focus `y` with a value of type str

With such a library an instruction like `incr x` can be translated to the method call `incr(x)`. And in the context of an if construction, such as `+ incr x {; ... }` the translation would be `result = incr(x); + result == true {; ... }`.

If an assignment is necessary, as is the case with `first x y`, then we cannot translate it to `first(x, y)`, because PGLEcm uses call by value for the arguments of a method, hence it is impossible to set focus `y` to the proper value. Alternatively, we can translate it to `y = first(x)`, but then we cannot check for the result. So we translate it to the following:

```
result = first(x);
- result == false {;
    y = result;
};
```

and let `first` return the proper value, or `false` in case of an error.

Because we use a molecular representation of values, making new value fields, assignments and comparisons must be translated as well. We do not give translations for all these possibilities, but we give some examples that indicate how these translations can be done.

Consider the instruction `+ x.+f:int = y.f {; ... }`. This can be translated as follows.

```
result = false;
```

1. website: www.science.uva.nl/research/prog/projects/pga/

```
    + y.f/int {;
        + x.+f {;
            x.f = copy-int(y.f);
            result = true;
        };
    };
    + result == true {;
        ...
    }
```

And the instruction `+ x.f = y.f {; ... }` can be translated as follows.

```
    result = false;
    + y.f/int {;
        + x.f/int {;
            x.f = copy-int(y.f);
            result = true;
        };
    }{;
        + y.f/str {;
            + x.f/str {;
                x.f = copy-str(y.f);
                result = true;
            };
        }{;
            + y.f?bool {;
                + x.f?bool {;
                    x.f = y.f;
                    result = true;
                };
            }{;
                x.f = y.f;
                result = true;
            };
        };
    };
    + result == true {;
        ...
    }
```

## 4. MSP with evaluation

We already showed that we can write a projection of PGLB to PGLA in PGLEc.MSP easily, but we also want to simulate PGLA. It is not possible with MSP to execute arbitrary code. So we introduce here the following instructions for evaluation as an extension of MSP and call this MSPea (MSP with eval/apply).

`compile` *`extfocus`*
: Compiles the string selected by *`extfocus`* into a molecule and assigns it to *`extfocus`*. It is capable of compiling strings representing programs in the primitive instruction set PGLA with MSP as basic instruction set. Returns `false` when there are errors detected.

`apply` *`extfocus`*
: If *`extfocus`* is a string, it is executed as a basic instruction and returns the value returned by the basic instruction. And otherwise it fails.

`eval` *`extfocus`*
: If *`extfocus`* is a string, it compiles the string into a molecule, after which the molecule is

evaluated. If *`extfocus`* is an atom, it is evaluated. And otherwise evaluation fails. It does no assignment of the molecule to *`extfocus`* like `compile` does. The returned value of `eval` is the returned value of the last evaluated basic instruction.

Compilation results in a molecule that represents a null-terminated list of instructions connected by `next` fields. The instructions are represented by an atom with additional fields. Below follows a list of possible instructions.

end
: This instruction consists of an atom with an `end` field, and stops the evaluation.

goto
: Consists of an atom with a `goto` field which selects the next instruction to be evaluated.

test
: Consists of an atom with the fields `test`, `basic`, `T`, and `F`. On the returned value of the basic instruction, selected by the `basic` field, either the `T` (`true`) field or the `F` (`false`) field selects the next instruction to be evaluated.

basic
: Consists of an atom with the field `basic` which selects a basic instruction to be evaluated. Evaluation continues with the next instruction in the list.

The basic instructions are represented by a string. For compilation the basic instruction may be anything, but for evaluation it has to be a MSP instruction. In evaluation, if a string is not recognized as a MSP instruction `false` is returned as the result of this instruction.

Consider the following program, that counts until 10 and then stops:

```
x = 0;
incr x;
+ x == 10;
!;
\\#3
```

We can put this program in a string with the instruction (note the escaping of the backslashes):

```
count = "x = 0; incr x; + x == 10; !; \\\\#3"
```

Compilation of `count` gives the molecule in Figure 4.

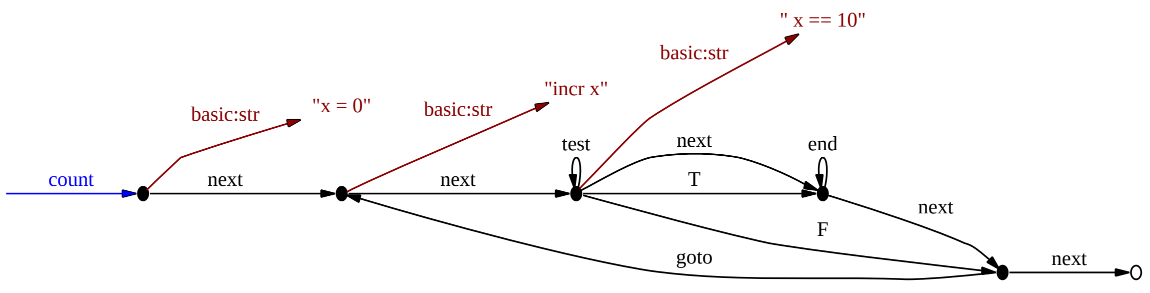

**Figure 4.** Result of compilation of `count`

This is the result of the current implementation in the PGA Toolset.

### *4.1 Translation of PGLEc.MSPea to PGLEc.MSP*

The instruction `eval` can be expressed in PGLEc and MSP as follows. It assumes that `x` contains the string or compiled molecule to be evaluated and puts the return value in the focus `result`.

```
program = x;
+ program?str {;
    - compile program {;
        result = false;
        ##L1;
    };
}{;
    - program? {;
        result = false;
        ##L1;
    };
};
pc = program;
result = true;
L0;
+ pc == null {;
    result = false;
    ##L1;
};
+ pc/end {;
    ##L1;
};
+ pc/goto {;
    pc = pc.goto;
    ##L0;
};
+ pc/test {;
    + apply pc.basic {;
        pc = pc.T;
        result = true;
    }{;
        pc = pc.F;
        result = false;
    };
    ##L0;
};
+ apply pc.basic {;
    result = true;
}{;
    result = false;
};
pc = pc.next;
##L0;
L1;
pc = null;
program = null;
!
```

The `compile` instruction can be expressed in PGLEc and MSP too. The code that does this can be found in Appendix B. However, the `apply` instruction can not be expressed in PGLEc with MSP. The reason for this is that we do not have the possibility to turn a string into a focus or fi eld selection.

## 5. Conclusions

We introduced the basic instruction set MSP that has instructions that operate on values. Values were introduced with the basic instruction set MPPV, but that was lacking operations on them. The example in Appendix A shows that MSP is powerful enough in order to program in them easily. Although we did not give full projections of MSP and HMPPV to a lower form, we showed that it is possible to do so.

We also introduced MSPea to evaluate arbitrary code. We showed that we can express the new instructions in PGLEc with MSP, apart from the `apply` instruction.

## References

[1] J.A. Bergstra and M.E. Loots, ‘‘Program algebra for sequential code,’’ *Journal of Logic and Algebraic Programming*, vol. 51, no. 2, pp. 125-156, 2002.

[2] J.A. Bergstra and I. Bethke, ‘‘Molecular dynamics,’’ *Journal of Logic and Algebraic Programming*, vol. 51, no. 2, pp. 193-214, 2002.

[3] B. Diertens, ‘‘A Toolset for PGA,’’ report PRG0302, Programming Research Group - University of Amsterdam, 2003.

## Appendix A.  Projection of PGLB to PGLA in PGLEc.MSP

In the following, a listing is given of the program in PGLEc with MSP to convert a string containing a program in PGLB to a string containing the projection of this program to PGLA.

```
strB = x;
instrlist = null;
nrinstr = 0;
L0;
instr = "";
- first strB h {;
   ##L100;
};
L10;
first strB h;
+ h == " " {;
   delfirst strB;
   ##L10;
};
space = "";
L11;
+ first strB h {;
   + h == " " {;
      append space " ";
      delfirst strB;
      ##L11;
   };
   + h == "\"" {;
      append instr space;
      append instr h;
      space = "";
      L12;
      delfirst strB;
      first strB h;
      append instr h;
      + h == "\\" {;
         delfirst strB;
         first strB h;
         append instr h;
         ##L12;
      };
      - h == "\"" {;
         ##L12;
      };
      delfirst strB;
      ##L11;
   };
   - h == ";" {;
      append instr space;
      append instr h;
      space = "";
      delfirst strB;
      ##L11;
   };
   delfirst strB;
};
+ instrlist == null {;
   pinstr = new;
   instrlist = pinstr;
}{;
   pinstr.+next = new;
   pinstr = pinstr.next;
};
+ instr == "!" {;
   pinstr.+end;
   ##L20;
};
first instr h;
+ h == "+" {;
   pinstr.+testt;
   delfirst instr;
   L21;
   first instr h;
   + h == " " {;
      delfirst instr;
      ##L21;
   };
   pinstr.+basic:str = instr;
   ##L20;
};
+ h == "-" {;
   pinstr.+testf;
   delfirst instr;
   L22;
   first instr h;
   + h == " " {;
      delfirst instr;
      ##L22;
   };
   pinstr.+basic:str = instr;
   ##L20;
};
+ h == "#" {;
   delfirst instr;
   + int instr s {;
      pinstr.+goto:int = s;
      ##L20;
   };
};
+ h == "\\" {;
   delfirst instr;
   first instr h;
   + h == "#" {;
      delfirst instr;
      + int instr s {;
         pinstr.+gotob:int = s;
         ##L20;
      };
   };
};
pinstr.+basic:str = instr;
L20;
incr nrinstr;
##L0;
L100;
pinstr = instrlist;
ic = 0;
+ pinstr == null;
##L200;
L110;
incr ic;
+ pinstr/goto {;
   h1 = ic;
   incr h1 pinstr.goto;
   h2 = nrinstr;
121    - decr h2 h1 {;
122       pinstr.goto = 0;
123       ##L190;
124    };
125 };
126 + pinstr/gotob {;
127    h1 = pinstr.gotob;
128    - decr h1 ic {;
129       h1 = nrinstr;
130       incr h1 2;
131       decr h1 pinstr.gotob;
132    }{;
133       h1 = 0;
134    };
135    pinstr.-gotob;
136    pinstr.+goto:int = h1;
137    ##L190;
138 };
139 L190;
140 + pinstr/next {;
141    pinstr = pinstr.next;
142    ##L110;
143 };
144 L200;
145 pinstr.+next = new;
146 pinstr = pinstr.next;
147 pinstr.+goto:int = 0;
148 pinstr.+next = new;
149 pinstr = pinstr.next;
150 pinstr.+goto:int = 0;
151 incr nrinstr 2;
152 pinstr.+next = new;
153 pinstr = pinstr.next;
154 pinstr.+repeat:int = nrinstr;
155 pinstr = instrlist;
156 strA = "";
157 L300;
158 + pinstr/end {;
150    instr = "!";
151    ##L310;
152 };
153 + pinstr/testt {;
154    instr = "+ ";
155    append instr pinstr.basic;
156    ##L310;
157 };
158 + pinstr/testf {;
159    instr = "- ";
160    append instr pinstr.basic;
161    ##L310;
162 };
163 + pinstr/goto {;
164    instr = "#";
165    str pinstr.goto s;
166    append instr s;
167    ##L310;
168 };
169 + pinstr/repeat {;
170    instr = "\\\\#";
171    str pinstr.repeat s;
172    append instr s;
173    ##L310;
174 };
175 instr = pinstr.basic;
176 L310;
177 + strA == "" {;
178    strA = instr;
179 }{;
180    append strA "; ";
181    append strA instr;
182 };
183 + pinstr/next {;
184    pinstr = pinstr.next;
185    ##L300;
186 }
```

The lines 1 through 110 parse the string in focus `x` and build a list of instructions in focus `instrlist`. The list of instructions in PGLB is converted to a list of instructions in PGLA in the lines 111 through 154. The remainder of the code converts the list of instructions to a string in focus `strA`.

## Appendix B. Compilation of PGLA in PGLEc.MSP

Here we give a listing of the program in PGLEc with MSP that expresses the `compile` instruction of MSPea.

```
 1 strA = x;
 2 instrlist = null;
 3 nrinstr = 0;
 4 L0;
 5 instr = "";
 6 - first strA h {;
 7    ##L100;
 8 };
 9 L10;
10 first strA h;
11 + h == " " {;
12    delfirst strA;
13    ##L10;
14 };
15 space = "";
16 L11;
17 + first strA h {;
18    + h == " " {;
19       append space " ";
20       delfirst strA;
21       ##L11;
22    };
23    + h == "\"" {;
24       append instr space;
25       append instr h;
26       space = "";
27       L12;
28       delfirst strA;
29       first strA h;
30       append instr h;
31       + h == "\\" {;
32          delfirst strA;
```

```
         first strA h;
         append instr h;
         ##L12;
      };
      - h == "\"" {;
         ##L12;
      };
      delfirst strA;
      ##L11;
   };
   - h == ";" {;
      append instr space;
      append instr h;
      space = "";
      delfirst strA;
      ##L11;
   };
   delfirst strA;
};
+ instrlist == null {;
   pinstr = new;
   instrlist = pinstr;
}{;
   pinstr.next = new;
   pinstr = pinstr.next;
};
pinstr.+next = null;
+ instr == "!" {;
   pinstr.+end;
   ##L20;
};
first instr h;
+ h == "+" {;
   pinstr.+testt;
   delfirst instr;
   L21;
   first instr h;
   + h == " " {;
      delfirst instr;
      ##L21;
   };
   pinstr.+basic:str = instr;
   ##L20;
};
+ h == "-" {;
   pinstr.+testf;
   delfirst instr;
   L22;
   first instr h;
   + h == " " {;
      delfirst instr;
      ##L22;
   };
   pinstr.+basic:str = instr;
   ##L20;
};
+ h == "#" {;
   delfirst instr;
   + int instr s {;
      pinstr.+goto:int = s;
      ##L20;
   };
};
+ h == "\\" {;
   delfirst instr;
   first instr h;
   + h == "\\" {;
      delfirst instr;
      first instr h;
      + h == "#" {;
         delfirst instr;
         + int instr s {;
            pinstr.+repeat:int = s;
            i = nrinstr;
            + decr i pinstr.repeat {;
               n = instrlist;
               L23;
               - i == 0 {;
                  decr i;
                  n = n.next;
                  ##L23;
               };
               pinstr.next = n;
            }{;
               i = pinstr.repeat;
               n = instrlist;
               L25;
               decr i;
               - i == 0 {;
                  nh = new;
                  nh.+next = n;
                  nh.+goto;
                  n = nh;
                  ##L25;
               };
               pinstr.next = n;
            };
            ##L100;
         };
      };
   };
};
pinstr.+basic:str = instr;
L20;
incr nrinstr;
##L0;
L100;
pinstr = instrlist;
L110;
+ pinstr == null;
##L200;
+ pinstr/repeat {;
   pinstr.+goto = pinstr.next;
   pinstr.next = null;
   pinstr.-repeat;
   ##L190;
};
+ pinstr/goto {;
   i = pinstr.goto;
   n = pinstr;
   L111;
   + i == 0 {;
      pinstr.-goto;
      pinstr.+goto = n;
      decr i;
      n = n.next;
      ##L111;
   };
   ##L190;
};
```

```
163 + pinstr/testt {;
164    pinstr.+T = pinstr.next;
165    + pinstr.next == null {;
166       pinstr.+F = pinstr.next;
167    }{;
168       pinstr.+F = pinstr.next.next;
169    };
170    pinstr.-testt;
171    pinstr.+test;
172    ##L190;
173 };
174 + pinstr/testf {;
175    pinstr.+F = pinstr.next;
176    + pinstr.next == null {;
177       pinstr.+T = pinstr.next;
178    }{;
179       pinstr.+T = pinstr.next.next;
180    };
181    pinstr.-testf;
182    pinstr.+test;
183    ##L190;
184 };
185 L190;
186 pinstr = pinstr.next;
187 ##L110;
188 L200;
189 x = instrlist
```

The lines 1 through 139 parse the string in focus `x` and build a list of instructions in focus `instrlist`. This list is translated to a molecule suitable for evaluation as described in section 4, in the remainder of the code.