programma

The Programma Language

Version 0.1

Gereon Fox

April 2013

Contents

1 Foreword

Programma development began in September 2010. This is the second version of the specification that contains many major changes and extensions in comparison to the first one.

However this version of the language is not intended for production use. It just serves as a basis for further development.

To get more information about this ongoing project visit `http://gfox.bplaced.net`.

2 Introduction

The reader will certainly ask why it should be necessary to introduce another programming language. This question shall be answered here.

As computer science and computers themselves improved over the decades the programming languages evolved to very sophisticated tools allowing us to very comfortably tell a computer what is to be done.

One major feature that most programming languages have is their universality: The fact that they are Turing complete ensures that we can describe any computational task using these languages.

The problem Programma aims to solve is the lack of universality of a very different kind: Programmers put a great deal of effort into writing code that is robust, fast and serviceable. Having accomplished this task however, there remains an unpleasant uncertainty about the program code: For how long will it keep its value. How long will it take until no one is able to execute this code either because of lack of knowledge or because of lack of a suitable runtime environment?

To sum these questions up one can say that the "time universality" of code written in these languages is not given. The code is not "eternal". It works here and now. But it probably won't work tomorrow at another place because technology changes.

The goal of the Programma project is to provide this "eternal universality". It tries to achieve it by reducing the assumptions made concerning the runtime environment: The "computer" is abstracted away and replaced by a more general "executive instance". Programma does not rely on a 32 bit or a 64 bit architecture nor does it require the availability of a network connection or a hard drive and the like. Programma is designed to avoid unnecessary requirements or determinations about the executive instance and its abilities which causes known concepts like bytes, files, or a console to be abandoned.

It remains to be seen if this ambitious approach to programming will be performant or even feasible and if it is suited to provide "eternal universality" for the programs we write.

3 Definitions

We rely on elementary set theory and predicate logic here. Anything above will be defined now:

3.1 General definitions

Let M be a set.

- Abbreviation: For a predicate $p(x)$ we define:

$$\exists^1 x \in M : \ p(x) \quad :\equiv \quad \exists x \in M : \ (p(x) \ \wedge \ \forall x' \in M : \ p(x') \leftrightarrow x' = x)$$

- $\mathbb{B} := \{0, 1\}$ – The set of ***bits***.
- $\mathbb{N}$ – The natural numbers, including 0.
- $\mathbb{N}_{<k} := \{n \in \mathbb{N} \mid n < k\}$
- $\mathbb{N}_{\leq k} := \{n \in \mathbb{N} \mid n \leq k\}$

Let s be a mathematical statement. We define:

$$bin(s) := \begin{cases} 0 & \neg s \\ 1 & s \end{cases}$$

3.2 Sequences

Let M be a set.

The set M^l of all ***sequences*** of length l over M is defined as

$$M^l := \{s \mid s : \mathbb{N}_{<l} \to M\}.$$

The ***length*** of $s \in M^l$ is defined as $|s| := length(s) := l$

Abbreviation:

$$s_i \quad := \quad s(i)$$

The only element of M^0 is written as ϵ and is called the ***empty sequence***.
The set M^* of all sequences is defined as

$$M^* \quad := \quad \bigcup_{l \in \mathbb{N}} M^l$$

An object x is said to be an element of $s \in M^*$ if

$$x \in s \quad :\equiv \quad \exists i \in \mathbb{N}_{<|s|}: \; s_i = x.$$

x is then called the ***i-th element*** of s.
Applying a function $f : M \to N$ to a sequence $s \in M^l$ is defined as

$$f(s) \quad := \quad \{(i, n) \mid i \in \mathbb{N}_{<l} \; \wedge \; f(s_i) = n\} \in N^l$$

Concatenation of two sequences $s \in M^k$, $t \in M^l$ is defined as

$$st \quad := \quad s \circ t \quad := \quad f : \mathbb{N}_{<k+l} \to M, \; f(s) = \begin{cases} s_i, & i < k \\ t_{i-k}, & i \geq k \end{cases}$$

If st contains at least one element, then ϵ is omitted in this representation.
Abbreviation:

$$s_0, ..., s_{l-1} := \begin{cases} \epsilon, & l < 1 \\ s_0, & l = 1 \\ s_0, ..., s_{l-2} \circ s_{l-1}, & l > 1 \end{cases}$$

For $b \in \mathbb{B}^n$ and $n \in \mathbb{N}$ we define:

$$[b] := \sum_{i=0}^{n-1} 2^i \cdot b_i$$

Since this function is a bijection[1] we can define $bin_n : \mathbb{N} \to \mathbb{B}^n$ as its inversion.
For a number $x \in \mathbb{N}$ and $m := min\{n \in \mathbb{N} \mid \exists b \in \mathbb{B}^n : [b] = x\}$ we define:

$$bin(x) := bin_m(x)$$

[1]Proof in section 8.1

3.3 Trees

Let L be a set. The set $\mathbb{T}(L)$ of ***trees over*** L is defined as follows: Let $l \in L$ be arbitrary.

1. Then $(l, \epsilon) \in \mathbb{T}(L)$
2. Let $c \in \mathbb{T}(L)^*$. Then $(l, c) \in \mathbb{T}(L)$ if, and only if, $t(l, c)$, where

$$t : L \times \mathbb{T}(L)^*, t(l, c) :\equiv \forall i \in c : t(i)$$

The second condition ensures that a tree is finite.

Let $t \in \mathbb{T}(L)$.

- The set $n(t)$ of ***nodes*** of this tree is defined as

$$n : L \times \mathbb{T}(L)^*, n(l, c) := \{(l, c)\} \cup \bigcup n(c)$$

- Let $i = (l, c)$. We define:
  - The ***label*** of i: $l(i) := l$
  - The ***children*** of i: $c(i) := c$
  - i being a ***leaf***: $leaf(i) :\equiv c = \epsilon$
- For two nodes $i, j \in n(t)$, i being the ***father*** of j and j being the ***n-th child*** of i is defined as

$$i \xrightarrow{t} j \quad :\equiv \quad c(i)(n) = j$$

- For an integer $n \in \mathbb{N}$ we define

$$i[n] \quad := \quad c(i)(n)$$

3.4 Syntax

Programma syntax is given as a context free grammar consisting of the following sets:

- N – The set of ***non-terminal symbols***. It contains all left hand sides of all production rules given in this specification and is a superset of $\{c' \mid \exists c, k : (c, k, c') \in R\}$
- $\mathbb{A}$ – The ***alphabet***. It contains all symbols displayable as a two-dimensional glyph.
- $T := N_T \times \mathbb{A}^*$ – The set of ***tokens***.[2].
- P – The set containing all production rules of this specification.

[2]For N_T see chapter 6

A ***production rule*** is denoted as $a ::= b$ and states that the ***left hand side*** $a \in N$ ***produces*** a sequence $b \in (N \cup T)^*$ of symbols that form the ***right hand side*** of the rule which means that a may be rewritten by b and b may be rewritten by a.

Throughout the specification the following abbreviations are used for $b, c, d \in (N \cup T)^*$:

- $a ::= b|c$ is defined as $a ::= b \wedge a ::= c$
- $a ::= b(c|d)e$ is defined as $a ::= bce \wedge a ::= bde$
- $a ::= b[c]d$ is defined as $a ::= bcd \wedge a ::= bd$

Furthermore the following rules apply:

- Sequences of alphabet symbols will often be '**`highlighted`**' for clarity.
- The specification makes use of several ***special symbols***:
  - σ – A visible symbol.
  - ψ – An invisible symbol.

A ***program*** is a sequence $p \in \mathbb{A}^*$.

A ***syntax tree*** $S_c(p) \in \mathbb{S}' := \mathbb{T}(N \cup T \cup \{\epsilon\})$ for a program p and a syntactic category $c \in N$ satisfies the following conditions:

1. $l(S_c(p)) = c$
2. $\forall i \in n(S_c(p)) : l(i) \in T \vee l(i) ::= l(c(i))$
3. $lw(S_c(p)) = tokenize(p)$ (see chapter 6)

where the ***leaf word*** of a node is defined as

$$lw(i) := \begin{cases} l(i) & leaf(i) \\ lw(i[0]) \circ \ldots \circ lw(i[n]) & |c(i)| = n + 1 \end{cases}$$

If no particular category is given, the syntax tree of a program p is to be thought of as

$$S(p) := S_{\langle program \rangle}(p)$$

For $a \in N$ and $b \in T^*$ we define that b is ***producible*** by a ($a \rightsquigarrow b$) if and only if there is a syntax tree $S_a(b)$.

From now on the specification refers to a fixed but arbitrary program p.

If $\neg(\langle program \rangle \rightsquigarrow p)$ then p is called ***syntactically invalid***. Otherwise there is not more than one[3] syntax tree and p is called ***syntactically valid***.

With the help of the definitions

[3]See section **??** for formal proof

- $R := \{(\langle call\rangle, \text{‘}\mathbf{new}\text{’}, \langle creation\rangle), (\langle eq-check\rangle, \text{‘}\mathbf{=}\text{’}, \langle eq-pos\rangle),$
  $(\langle eq-check\rangle, \text{‘}\mathbf{\neq}\text{’}, \langle eq-neg\rangle), (\langle ty-check\rangle, \notin, \langle ty-neg\rangle),$
  $(\langle ty-check\rangle, \text{‘}\in\text{’}, \langle ty-pos\rangle)\}$

- $a_1 : \mathbb{S}' \to \mathbb{S}', a_1(l, c) := \begin{cases} (l', a_1(c)) & \exists k \in K \cap l(c) : (l, k, l') \in R \\ (t, \epsilon) & l = (n, t) \in T \\ (l, a_1(c)) & otherwise \end{cases}$

- $filter : \mathbb{S}'^* \to \mathbb{S}'^*, filter(s) := \begin{cases} \epsilon & s = \epsilon \\ filter(s_1, ..., s_{|s|-1}) & l(s_0) \in K \\ s_0 \circ (filter(s_1, ..., s_{|s|-1}) & otherwise \end{cases}$

- $a_2 : \mathbb{S}' \to \mathbb{S}', a_2(l, c) := (l, a_2(filter(c))$

- $leaf_a(i) :\equiv l(i) \in \{\langle bits\rangle, \langle symbols\rangle, \langle id\rangle\}$

- $essential(i) :\equiv leaf_a(i) \vee \exists j \in c(i) : essential(j)$

- $i \twoheadrightarrow j :\equiv j = c(i)(\min\{n \in \mathbb{N} \mid \forall j' \in c(i) : essential(j') \Rightarrow j' = i[n]\})$

- $E := \{\langle creation\rangle\}$

- $a_3 : \mathbb{S}' \to \mathbb{S}', a_3(i) := \begin{cases} i & leaf_a(i) \\ a_3(j) & \neg leaf_a(i) \wedge i \twoheadrightarrow j \wedge l(i) \notin E \\ (l(i), a_3(c(i))) & otherwise \end{cases}$

we can define the ***abstract syntax tree*** for program p and a syntactic category $c \in N$ as

$$A_c(p) := a_3(a_2(a_1(S_c(p))))$$

In analogy to S, we define

$$A(p) := A_{\langle program\rangle}$$

We define $\mathbb{S} := n(imA) \subset \mathbb{T}(N \cup \mathbb{A}^*) \subset \mathbb{S}'$

3.5 Semantics

Let M be a set. An M-***context*** is a function $C : \mathbb{S} \to \wp(M)$. We define:

- $C \vdash x \triangleright m :\equiv m \in C(x)$
- For $x \in \mathbb{S}$ and $y \in M$:

$$C[x := y] : \mathbb{S} \to M, C[x := y](i) := \begin{cases} y & i = x \\ C(i) & otherwise \end{cases}$$

An ***inference rule*** of the form

$$\frac{P}{C}$$

where P and C are sets of mathematical statements is to be understood as

$$\bigwedge_P p \Rightarrow \bigwedge_C c$$

P and C will be denoted without set brackets or commata between their elements.

4 The Executive Instance

4.1 Definition

The ***Executive Instance*** (***EI***) is an object of the real world that executes the program according to this specification.

The ***execution*** of the program by the EI is modeled as transforming a start state into an end state.

A ***state*** $s \in \mathit{States}$ consists of

- $s.m : \mathit{Variables} \to \mathit{Values}$ — the content of the ***memory*** of the EI.
- $s.st \in \mathit{Statements}^*$ — the ***stack*** of statements that are currently being executed.
- $s.oc \in \mathbb{N}$ — the ***object counter***.
- $s.dev \subseteq \mathit{Id} \times \mathbb{B}^*$ — the content of the devices.

The EI operates in several distinct ***phases*** that must be absolved in the given order:

1. ***Startup*** — This phase starts as soon as the EI becomes active. The EI must carry out whatever necessary for consuming and executing a program.
2. ***Consumption*** — The EI now is meant to consume the program it has been given in any form. Exactly one program is to be consumed.
3. ***Execution/Runtime*** — This phase starts as soon as the program has been loaded and the representation of the ***initial state*** has been set up. It lasts until an ***end state*** with an empty stack is reached or an uncaught exception occurs.
4. ***Shutdown*** — As soon as execution is finished, the EI discards all resources it holds and ceases any activity.

The execution phase is modeled by the function

$$\delta : \mathit{States} \to \mathit{States}, \delta(s) = s'.$$

δ will be specified by case distinction on s. From now on s will represent the current state and s' will represent the state meant to follow s.

If the execution is ever ended, the program p is said to have ***terminated***. Otherwise p is ***divergent***.

A ***device*** is a facility controlled by the EI that can be used to interact with the real world. The state of a device is represented by a bit sequence that can be referenced under a special ***device name***. The length of this sequence never changes.

During the execution phase exceptions may occur. An ***exception*** is the occurence of a state that is not an end state but for which δ is not defined. This condition leads to the creation of an ***exception object*** which is a value of the type *exception*. This exception can either be handled[1] or not depending on the given program.

An EI is said is said to be ***correct*** if it fulfills the following requirements:

1. If, according to this specification, p diverges *in any way* for a given initial state then the EI must never end the execution with a proper end state.
2. If, according to this specification, p terminates without an unhandled exception for a given initial state then the EI must reach a proper end state in which the values of all device variables equal the states of all its devices.
3. If, according to this specification, p terminates with an unhandled exception for a given initial state then the EI must immediately end the execution phase and should report this exception to the real world.

[1]For the definition of **handle** see section **??**

5 Variables, procedures & types

5.1 Definitions

A ***variable*** is an element of $Var := Id \times Types$. The first component of a variable tuple is called the ***variable*** name and the second one is called ***variable type***. A variable v is often denoted as $name : type$.

A ***procedure*** $p \in Procedures$ consists of

- $p.rtype \in Types$ – The ***return type*** of the procedure.
- $p.atype \in Types$ – The ***parameter type*** of the procedure.
- $p.sem : States \times p.atype \to States \times rtype$ – The semantics function of the procedure: The function takes the current state[1] and an argument values. It returns the state after the execution of the procedure and its return value.

A ***type*** is an element of the set $Types$. A $t \in Types$ consists of

- $t.ancestors \subseteq Types$ – The types t is ***inheriting*** from.
- $t.public \in Id \to Procedures$ – The ***public members*** of t.
- $t.external \in Id \to Procedures$ – The ***external members*** of t.
- $t.internal \in Id \to Procedures$ – The ***internal members*** of t.
- $t.secret \in Id \to Procedures$ – The ***secret members*** of t.
- $t.range$ – A superset of possible ***values of type*** t.

We will often omit the suffix $.range$.

The set $GTypes$ of ***generic types*** is defined as follows:

1. Every $t : Types \to Types$ is an element of $GTypes$
2. Every $t : Types \to GTypes$ is an element of $GTypes$

For $t \in GTypes$ we denote the value $t(x)$ by $t\langle x \rangle$

An object v having a certain type t is denoted by $v : t \;:\equiv\; v \in t.range$.

We define: $Values := \{v \mid \exists t \in Types : v : t\}$

[1]For the definition of **state** see section 4.1

5.2 Procedure types

For any $t_1, t_2 \in \mathit{Types}$ the statement $t_1 \to t_2 \in \mathit{Types}$ holds and we define:

- $(t_1 \to t_2).\mathit{pc} := 0$
- $(t_1 \to t_2).p := \epsilon$
- $(t_1 \to t_2).\mathit{ancestors} := \{\mathit{object}\}$
- $\mathit{members}(t_1 \to t_2) = \emptyset$
- $(t_1 \to t_2).\mathit{range} = \{p \in \mathit{Procedures} \mid p.\mathit{atype} = t_1 \wedge p.\mathit{rtype} = t_2\}$

$t_1 \to t_2$ is called a ***procedure type***.

As with variables we will often denote a procedure p with type t as $p : t$. To ease reading we define that $a \to b \to c$ is to be understood as $a \to (b \to c)$, so $\to$ is right-recursive.

5.3 Internals

The ***methods***/***members*** of a type t are defined as

$$\mathit{members}(t) \quad := \quad t.\mathit{public} \cup t.\mathit{external} \cup t.\mathit{internal} \cup t.\mathit{secret}$$

5.4 Inheritance

Let $s, t \in \mathit{Types}$ be two types. The statement that s ***inherits***/***extends*** t is denoted by

$$s \triangleright t$$

which is true if and only if

1. $t \in s.\mathit{ancestors}$ – t is one of the ancestors of s.
2. $s.\mathit{range} \subseteq t.\mathit{range}$ – Every value of type s is also a value of type t.

For any $s, t \in \mathit{Types}$ and $g \in \mathit{GTypes}$ we specify:

$$s \triangleright t \quad \Rightarrow \quad g(s) \triangleright g(t)$$

5.5 Basic types

The following base types have to be supported by the EI:

$$\mathit{void}, \mathit{bit}, \mathit{symbol}, \mathit{object}, \mathit{sequence}, \mathit{exception}, \mathit{device} \in \mathit{Generics}$$

void

- $void.ancestors := \emptyset$
- $members(void) = \emptyset$
- $void.range = \{0\}$

bit The type *bit* represents the basic unit of information:

- $bit.ancestors := \emptyset$
- $members(bit) = \emptyset$
- $bit.range = \mathbb{B}$

symbol The type *symbol* contains all the symbols that are displayable as a two dimensional glyph, which we defined as T in section 3.4.

- $symbol.ancestors := \emptyset$
- $members(symbol) = \emptyset$
- $symbol.range = T$

Note, that $symbol.range \cap bit.range = \emptyset$.

object The great majority of $Values$ is of this type:

- $object.ancestors := \emptyset$
- $members(object) = \emptyset$
- $object.range = \{v \mid \exists t \in Types : v : t \wedge t \triangleright object\}$

sequence For any type $t \in Types$ we define:

- $sequence < t > .ancestors := \{object\}$
- $sequence < t > .public = \{create, length\}$
- $sequence < t > .external = sequence < t > .internal = sequence < t > .secret = \emptyset$
- $sequence < t > .range = t.range^*$

The procedures are defined as follows:

- $create.rtype = sequence\langle t\rangle$
- $create.atype = void$
- $create.sem : States \to States \times sequence\langle t\rangle, sem(s) = (s, \epsilon)$

- $length.rtype = sequence\langle bit\rangle$
- $length.atype = sequence\langle t\rangle$
- $length.sem : States \times sequence\langle t\rangle \to States \times sequence\langle bit\rangle, sem(s, v) = (s, |v|)$

exception Objects of this type are used to signal that δ is not defined:

- $exception.ancestors := \{object\}$
- $exception.public = \{create, message\}$
- $exception.external = sequence < t > .internal = sequence < t > .secret = \emptyset$
- $exception.range = \{0\} \times sequence\langle symbol\rangle$

The procedures are defined as follows:

- $create.rtype = exception$
- $create.atype = sequence\langle symbol\rangle$
- $create.sem : States \times sequence\langle symbol\rangle \to States \times exception,$
$$sem(s, m) = (s, (0, m))$$
- $message.rtype = sequence\langle symbol\rangle$
- $message.atype = exception$
- $message.sem : States \times exception \to States \times sequence\langle symbol\rangle, sem(s, (0, m)) = (s, m)$

device These values represent the EI's connections to the real world.

- $device.ancestors := \{object\}$
- $members(device) = \emptyset$
- $device.range = \{1\} \times sequence < bit >$

5.6 Predefined procedures

As well as there are predefined types in Programma, there also are some predefined procedures:

- $or : bit \to bit \to bit$

  where $or.sem(s, b_1) := (s, or')$ and $or'.sem(s, b_2) := (s, 1 - (b_1 \cdot b_2))$
- $and : bit \to bit \to bit$

  where $and.sem(s, b_1) := (s, and')$ and $and'.sem(s, b_2) := (s, b_1 \cdot b_2)$

- $not : bit \to bit$

  where $not.sem(s, b) := (s, 1 - b)$
- $>$: $sequence\langle bit\rangle \to sequence\langle bit\rangle \to bit$

  where $> .sem(s, b_1) := (s, >')$ and $>' .sem(s, b_2) := (s, bin([b_1] > [b_2]))$
- $<$: $sequence\langle bit\rangle \to sequence\langle bit\rangle \to bit$

  where $< .sem(s, b_1) := (s, <')$ and $>' .sem(s, b_2) := (s, bin([b_1] < [b_2]))$
- $+ : sequence\langle bit\rangle \to sequence\langle bit\rangle \to bit$

  where $+.sem(s, b_1) := (s, +')$ and $+'.sem(s, b_2) := (s, bin([b_1] + [b_2]))$
- $- : sequence\langle bit\rangle \to sequence\langle bit\rangle \to bit$

  where $-.sem(s, b_1) := (s, -')$ and $-'.sem(s, b_2) := (s, bin([b_1] - [b_2]))$
- $\& : sequence\langle bit\rangle \to sequence\langle bit\rangle \to bit$

  where $\&.sem(s, b_1) := (s, \&')$ and $\&'.sem(s, b_2) := (s, b_1 \circ b_2)$

6 Lexical grammar

The EI consumes a program $p \in A^*$ from left to right by splitting it up into tokens, which is defined by the sets

$$\begin{array}{rcl}
LS & := & \{?, Cmt, Smb, Bit, Wrd\} \\
N_T & := & \{\langle comment\rangle, \langle bits\rangle, \langle symbols\rangle, \langle keyword\rangle, \langle id\rangle\} \\
K & := & \{\text{`}\in\text{'}, \text{`}\notin\text{'}, \text{`}\texttt{(}\text{'}, \text{`}\texttt{)}\text{'}, \text{`}\texttt{:}\text{'}, \text{`}\texttt{?}\text{'}, \text{`}\texttt{[}\text{'}, \text{`}\texttt{]}\text{'}, \text{`}\texttt{=}\text{'}, \text{`}\neq\text{'}, \\
 & & \quad \text{`}\texttt{new}\text{'}, \text{`}\lambda\text{'}\} \\
D & := & \{w \in K \mid |w| = 1\} \\
D_I & := & \{\text{`}\texttt{'}\text{'}, \text{`}\bullet\text{'}, \text{`}\circ\text{'}\}
\end{array}$$

and the function

$$lex : LS \times A^* \times A^* \to N_T \times A^* \times A^*,$$

$$
\begin{array}{llrcll}
lex(?, & \epsilon, & \text{‘}\texttt{'}\text{’} \circ cr) & := & lex(Smb, \epsilon, cr) & \\
lex(Smb, & t, & \text{‘}\texttt{'}\text{’} \circ cr) & := & (\langle symbols\rangle, t, cr) & \\
lex(Smb, & t, & c \circ cr) & := & lex(Smb, t \circ c, cr) & c \neq \text{‘}\texttt{'}\text{’} \\
 & & & & & \\
lex(?, & \epsilon, & \text{‘}\texttt{(*}\text{’} \circ cr) & := & lex(Cmt, \epsilon, cr) & \\
lex(Cmt, & t, & \text{‘}\texttt{*)}\text{’} \circ cr) & := & (\langle comment\rangle, t, cr) & \\
lex(Cmt, & t, & c \circ c' \circ cr) & := & lex(Cmt, t \circ c, c' \circ cr) & c \circ c' \neq \text{‘}\texttt{*)}\text{’} \\
 & & & & & \\
lex(?, & \epsilon, & \text{‘}\bullet\text{’} \circ cr) & := & lex(Bit, \text{‘}\bullet\text{’}, cr) & \\
lex(?, & \epsilon, & \text{‘}\circ\text{’} \circ cr) & := & lex(Bit, \text{‘}\circ\text{’}, cr) & \\
lex(Bit, & t, & \text{‘}\bullet\text{’} \circ cr) & := & lex(Bit, t \circ \text{‘}\bullet\text{’}, cr) & \\
lex(Bit, & t, & \text{‘}\circ\text{’} \circ cr) & := & lex(Bit, t \circ \text{‘}\circ\text{’}, cr) & \\
lex(Bit, & t, & cs) & := & (\langle bits\rangle, t, cs) & cs(1) \notin \{\text{‘}\circ\text{’}, \text{‘}\bullet\text{’}\} \\
 & & & & & \\
lex(?, & \epsilon, & d \circ c' \circ cr) & := & \begin{cases} lex(?, \epsilon, c' \circ cr) & \psi \rightsquigarrow d \\ (\langle keyword\rangle, d, c' \circ cr) & d \in D \\ lex(Wrd, d, cr) & otherwise \end{cases} & d \circ c' \neq \text{‘}\texttt{(*}\text{’} \\
 & & & & & \\
lex(?, & \epsilon, & d) & := & (n(c), d, \epsilon) & \neg(\psi \rightsquigarrow d) \\
 & & & & & \\
lex(Wrd, & t, & c \circ cr) & := & \begin{cases} (n(t), t, cr) & \psi \rightsquigarrow c \\ (n(t), t, c \circ cr) & c \in D \cup D_I \\ lex(Wrd, t \circ c, cr) & otherwise \end{cases} & \\
 & & & & & \\
lex(Wrd, & t, & \epsilon) & := & (n(t), t, \epsilon) &
\end{array}
$$

where $|d| = |c| = |c'| = 1 \ \wedge \ cs \in A^* \ \wedge \ d \notin D_I$ and

$$
n(t) := \begin{cases} \langle keyword\rangle & t \in K \\ \langle id\rangle & t \notin K \end{cases}
$$

For $p \in A^*$ and $lex(?, \epsilon, p) = (l, t, r)$ we define:

$$
\begin{array}{lcl}
label(p) & := & l \\
token(p) & := & (l, t) \\
rest(p) & := & r
\end{array}
$$

The function $tokenize : A^* \to T^*$, that is defined by

$$
tokenize(p) := \begin{cases} \epsilon & \psi \rightsquigarrow p \vee p = \epsilon \\ tokenize(rest(p)) & label(p) = \langle comment\rangle \\ token(p) \circ tokenize(rest(p)) & otherwise \end{cases}
$$

converts a stream of characters into a stream of tokens.

For $i \in \{\sigma\}^*, s \in \{\sigma, \psi\}^*, b \in \{‘\circ’, ‘\bullet’\}^*, k \in K$ we define:

$$\begin{array}{lll}
\langle id \rangle & ::= & (\langle id \rangle, i) \\
\langle symbols \rangle & ::= & (\langle symbols \rangle, s) \\
\langle bits \rangle & ::= & (\langle bits \rangle, b) \\
k & ::= & (\langle keyword \rangle, k)
\end{array}$$

7 The language

Having put into place all the necessary definitions, we can now specify the actual Programma language:

7.1 Type names

⟨*type*⟩	::=	**TODO: Specify this!**

TODO: Add static rules to assign these phrases the type they refer to!

7.2 Expressions

The following grammar specifies how expressions are to be formed:

⟨*exp*⟩	::=	⟨*eq-check*⟩ [**?** ⟨*exp*⟩ **:** ⟨*exp*⟩]
⟨*eq-check*⟩	::=	⟨*ty-check*⟩ [(**=** | **≠**) ⟨*exp*⟩]
⟨*ty-check*⟩	::=	⟨*call*⟩ [($\in$|$\notin$) ⟨*type*⟩]
⟨*call*⟩	::=	⟨*projection*⟩ ⟨*call'*⟩ | **new** ⟨*id*⟩ ⟨*call'*⟩
⟨*call'*⟩	::=	[⟨*projection*⟩ ⟨*call'*⟩]
⟨*projection*⟩	::=	⟨*primitive*⟩ ⟨*projection'*⟩
⟨*projection'*⟩	::=	['**[**' ⟨*exp*⟩ [**:** ⟨*exp*⟩] '**]**' ⟨*projection'*⟩]
⟨*primitive*⟩	::=	⟨*bits*⟩ | ⟨*symbols*⟩ | ⟨*id*⟩ | **λ** ⟨*parameters*⟩ ⟨*block*⟩ | '**(**' ⟨*exp*⟩ '**)**'
⟨*parameter*⟩	::=	⟨*type*⟩ ⟨*id*⟩
⟨*parameters*⟩	::=	[⟨*parameter*⟩ ⟨*parameters*⟩]

Given a ***type context*** $T : \mathbb{S} \to Types$, the ***static semantics*** of these expressions are defined by the following inference rules:

$$\frac{l(i) = \langle \mathit{bits} \rangle}{T \vdash i \triangleright \mathit{bit}} \qquad \frac{l(i) = \langle \mathit{symbols} \rangle}{T \vdash i \triangleright \mathit{symbol}} \qquad \frac{l(i) = \langle \mathit{id} \rangle \quad i \to j \quad T \vdash l(j) \triangleright t}{T \vdash i \triangleright t}$$

$$\frac{\begin{array}{c} l(i) = \langle \mathit{parameter} \rangle \\ T \vdash i[0] \triangleright t \end{array}}{T \vdash i \triangleright t} \qquad \frac{\begin{array}{c} l(i) = \langle \mathit{parameters} \rangle \\ T \vdash i[0] \triangleright t \quad T \vdash i[1] \triangleright t' \end{array}}{T \vdash i \triangleright t \to t'} \qquad \frac{\begin{array}{c} l(i) = \langle \mathit{parameters} \rangle \\ T \vdash i[0] \triangleright t \quad l(i[1]) = \epsilon \end{array}}{T \vdash i \triangleright t}$$

7.3 Statements

$\langle \mathit{block} \rangle$ ::= **TODO: Specify this!**

8 Proofs

8.1 Binary bijection

Claim: For every $n \in \mathbb{N}$

$$[b] = \sum_{i=0}^{n-1} 2^i \cdot b_i$$

is a bijection from $\mathbb{B}^n$ to $\mathbb{N}_{<2^n}$

Proof by induction on $n \in \mathbb{N}$.

Base case: Let $n = 0$. Then:

$$\mathbb{B}^n = \{\epsilon\}$$

and

$$[\epsilon] = \sum_{i=0}^{-1} 2^i \cdot \epsilon_i = 0 < 1 = 2^n$$

Induction step: Let $[.]$ be a bijection to $\mathbb{N}_{<2^n}$ for an arbitrary but fixed $n \in \mathbb{N}$. Then every element $b \in \mathbb{B}^{n+1}$ is displayable as

$$b = r \circ a$$

where $a \in \mathbb{B}$ and $r \in \mathbb{B}^n$, so that

$$[b] := \sum_{i=0}^{n} 2^i \cdot b_i = \left(\sum_{i=0}^{n-1} 2^i \cdot b_i \right) + 2^n \cdot a$$

If $a = 0$ then

$$[b] = \sum_{i=0}^{n-1} 2^i \cdot b_i = [r]$$

So, according to the induction hypothesis, by adjusting r we can hit each element of $\mathbb{N}_{<2^n}$ exactly once and no image can be outside this set.

If $a = 1$ then

$$[b] = \left(\sum_{i=0}^{n-1} 2^i \cdot b_i\right) + 2^n = [r] + 2^n$$

So, according to the induction hypothesis, by adjusting r we can hit each element of $\{n \in \mathbb{N} \mid 2^n \leq n < 2^{n+1}\}$ exactly once and no image can be outside this set.

This shows that by applying $[.]$ to every element of $\mathbb{B}^{n+1}$ we hit every element of $\mathbb{N}_{<2^{n+1}}$ exactly once and never hit a natural number outside this set.

$\blacksquare$

TODO: We should proove the unambiguity of the grammar!

TODO: Add an index that is to contain at least all defined terms!