AGUDA is an imperative programming language composed of expressions alone.

A program in AGUDA is a sequence of declarations, each announced with keyword `let`. Value declarations introduce an identifier and a type. For example:

```
let maxSpeed : Int = 120
```

Declarations may also introduce functions, as in

```
let succ (n) : Int -> Int = n + 1
```

Functions can be recursive:

```
let add (n, m) : (Int, Int) -> Int =
  if n == 0 then m else succ(add (n - 1, m))
```

or even mutually recursive:

```
let even (x) : Int -> Bool =
  x == 0 || odd(x - 1)

let odd (x) : Int -> Bool =
  x != 0 && even(x - 1)
```

Imperative means state changing. How do we change state? By means of (imperative) variables and assignment. Here's addition in imperative style:

```
let AddI (n, m) : (Int, Int) -> Int =
  let sum : Int = m;
  while n > 0 do (
    set sum = sum + 1;
    set n = n - 1
  );
  sum
```

A few points to notice:

- The body of the function is composed of three *expressions*, *separated* by a semicolon;
- The value of the function is given by the last expression in the semicolon separated sequence, namely, `sum`;

- The body of the **while** *expression* is a sequence of two expressions; we enclose them in parenthesis.
- Assignments are announced by the **set** keyword

Each expression has a value, including the **while** loop. The value of **while** is **unit**, the only value of type **Unit**. This means that we can "store" a while loop in a variable, as in

```
let x : Unit = while i > 0 do set i = i - 1
```

or "return" a while loop from a function, as in

```
let f (n) : Int -> Unit = while n > 0 do set n = n - 1
```

An assignment stores a value in a variable and returns the value. Given the below declaration, a call to `f(unit)` yields value `3`.

```
let f (_) : Unit -> Int = let x : Int = let y : Int = 3
```

Equipped with the **unit** value, an if-then expression **if** `exp1` **then** `exp2` is an abbreviation for an if-then-else **if** `exp1` **then** `exp2` **else unit**.

Let us now look at the support provided for arrays. We start with an example: creating an `n` × `n` matrix, where all entries are `0`, except for the diagonal which is filled with `1`.

```
let diagonal (n) : Int -> Int[][] =
  let a : Int[][] = new Int[] [n | new Int [n | 0]] ;
  let i : Int = 0 ;
  while i < length(a) do (
    set a[i][i] = 1 ;
    set i = i + 1
  ) ;
  a
```

Expression **new Int** `[n | 0]` creates an integer array of size `n`, each cell initialised to `0`. Likewise, **new Int**`[]` `[n | new Int [n | 0]]` creates a `n` × `n` matrix of zero values. Expression **set** `a[i][i] = 1` writes `1` in the `i`-th position of the `i`-th array of matrix `a`, that is, in line `i`, column `i` of `a`.

Below is a function that prints a matrix, line by line, each value terminated with a space. It includes two *primitive* functions: **length** returns the number of elements in an array; **print** sends to the prints `stdout` the textual representation of a given value. Both functions are overloaded: **length** works on any array; **print** accepts any value. More primitive functions shall be announced later.

```
let printMatrix (a) : Int[][] -> Unit =
  let i : Int = 0 ;
  while i < length(a) do (
```

```
    while j < length(a[0]) do (
      print(a[i][j]) ; print(" ") ;
      set j = j + 1
    );
    print("\n") ;
    set i = i + 1
  )
```

A "little" main function exercises the two array-related functions.

```
let main : Unit =
  printMatrix(diagonal(10))
```

Expressions

- Variable: `id`
- Literals: `...`, `-1`, `0`, `1`, `...`, **`true`**, **`false`**, **`null`**, in addition to quote-enclosed string literals
- Binary operators: `;` `+` `-` `*` `/` `%` `^` `==` `!=` `<` `<=` `>` `>=` `!` `||` `&&`
- Unary operators: `-` `!`
- Function call: `id(exp1,...,expn)` with $n \geq 1$
- Assignment: **`set`** `LHS = exp`
- Variable declarations: **`let`** `id : type = exp`
- Conditionals: **`if`** `exp1` **`then`** `exp2` **`else`** `exp3` and **`if`** `exp1` **`then`** `exp2`
- While loop: **`while`** `exp1` **`do`** `exp2`
- Array creation: **`new`** `type [ exp1 | exp2 ]`
- Array access: `exp1[exp2]`
- Parenthetical expression: `(exp)`

The left-hand-side (`LHS`) of an assignment, that is, the part at the left of =, can be:

- A variable
- An array location: `LHS[exp]`

Operator precedence and associativity

- For `+ - * / % == != < <= > >= ! || &&` take the precedence and associativity of the Java programming language
- The unary minus binds tighter than any other arithmetic operator
- The power operator, `^`, is right associative and binds tighter than any other arithmetic operator (different from unary minus). For example, `2 ^ - 3 ^ 4 * 5` is to be understood as `(2 ^ ((-3) ^ 4)) * 5`
- The sequencing operator, `;`, associates to the right and binds loser than all other operators. For example, `1 ; 2 || 3 ; 4` is to be understood as `1 ; ((2 || 3) ; 4)`

Top-level declarations

- Variables: as in expressions
- Functions: `let id (id1,...idn) : type = exp` with $n \geq 1$

Programs A non-empty sequence of declarations.

Types

- Basic: `Int`, `Bool`, `Unit`, `String`
- Array: `type[]`
- Function: `type -> type` or `(type1,...,typen) -> type` with $n \geq 1$. There are no zero-ary functions. If needed, use `f (_) : Unit -> type = exp` and call as `f(unit)`.

Lexing

- Identifiers (variable or function names) start with a letter and are followed by zero or more letters, digits, underscore symbols (_) and single quotes (')
- Integer values start with an optional sign, followed by a non-zero digit and then followed by zero or more digits
- Strings are sequences of characters (not including new line), enclosed in quotes (")
- Comments: Line comments only, starting with `--`.