



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 (x) : 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 $n \times n$ 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 $n \times n$ 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`, `unit`, 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)`
- The precedence of **while** loops and conditionals (both **if-then-else** and **if-then**) seats between that of `;` and `||`. For example, **while** `b` **do** `false || true` should be understood as **while** `b` **do** `(false || true)`, but **while** `b` **do** `false ; true` should be understood as `(while b do false) ; true`. Similarly for conditionals
- Keywords **then** and **else** associate to the right, so that **if** `a` **then** **if** `b` **then** `c` **else** `d` is to be understood as **if** `a` **then** `(if b then c else d)`
- Array access, `[` binds tighter than unary minus, so that `-a[0]` denotes `-(a[0])`
- The arrow type operator, `->`, associates to the right

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 `--`.

Typing The types for expressions are as follows.

- The type of a literal is the corresponding type. For example, the type of 5 is **Int**
- The type of array creation **new** type [exp1 | exp2] is type[]. Furthermore, exp1 should be of type **Int** and exp2 should be of type type
- The type of array access exp1[exp2] is type if exp1 is of type type[] and exp2 of type **Int**
- The type of call **print**(exp1, ..., expn) is **Unit** if n = 1 and exp1 has a type (any type)
- The type of call **length**(exp1, ..., expn) is **Int** if n = 1 and exp1 has a type type[]
- The type of call id(exp1, ..., expn) (with id ≠ **print**, **length**) is type if id is of type (type1, ..., typem) → type and n = m and each expi has type typei
- The type of variable declaration **let** id : type = exp is **Unit** if exp is of type type. Furthermore, if the declaration appears at the left of a semicolon **let** id : type = exp1 ; exp2, then the type of id is used to validate exp2
- The type of an identifier is that more recently introduced by a **let** (either expression or top-level **let** declaration)
- The type of conditional **if** exp1 **then** exp2 **else** exp3 is type if exp1 is of type **Bool** and both exp2 and exp3 are of type type. In the case of **if** exp1 **then** exp2, expression exp2 must be of type **Unit**
- The type of while loop **while** exp1 **do** exp2 is **Unit** if exp1 is of type **Bool** and exp2 has a type (any type)

- The type of assignment **let** LHS = e is **Unit** if LHS and e share the same type
- The type of sequential composition `exp1; exp2` is the type of `exp2` if `exp1` has a type (any type)
- The type of the remaining operators (binary and unary) are as customary in programming languages

The types for LHS are as follows.

- Variables, as in expressions
- The type of `LHS[exp]` is `type` if LHS is of type `type[]` and `exp` is of type **Int**

Top-level declarations have no types. The validation rules are as follows.

- Variables, as in expressions
- Function declaration **let** id (id1, ..., idn) : type = exp is valid if `type = (type1, ..., typem) -> type'` and `n = m` and `exp` is of type `type'` in a context augmented with `exp1:type1, ..., expn:typen`
- No variable of function may be named as **print** or **length**
- Declarations may be (mutually) recursive
- Functions cannot be declared twice, even if with different signatures

The validation rule for programs is as follows.

- A program is valid if all its declarations are