AML

Language Design and Example Programs

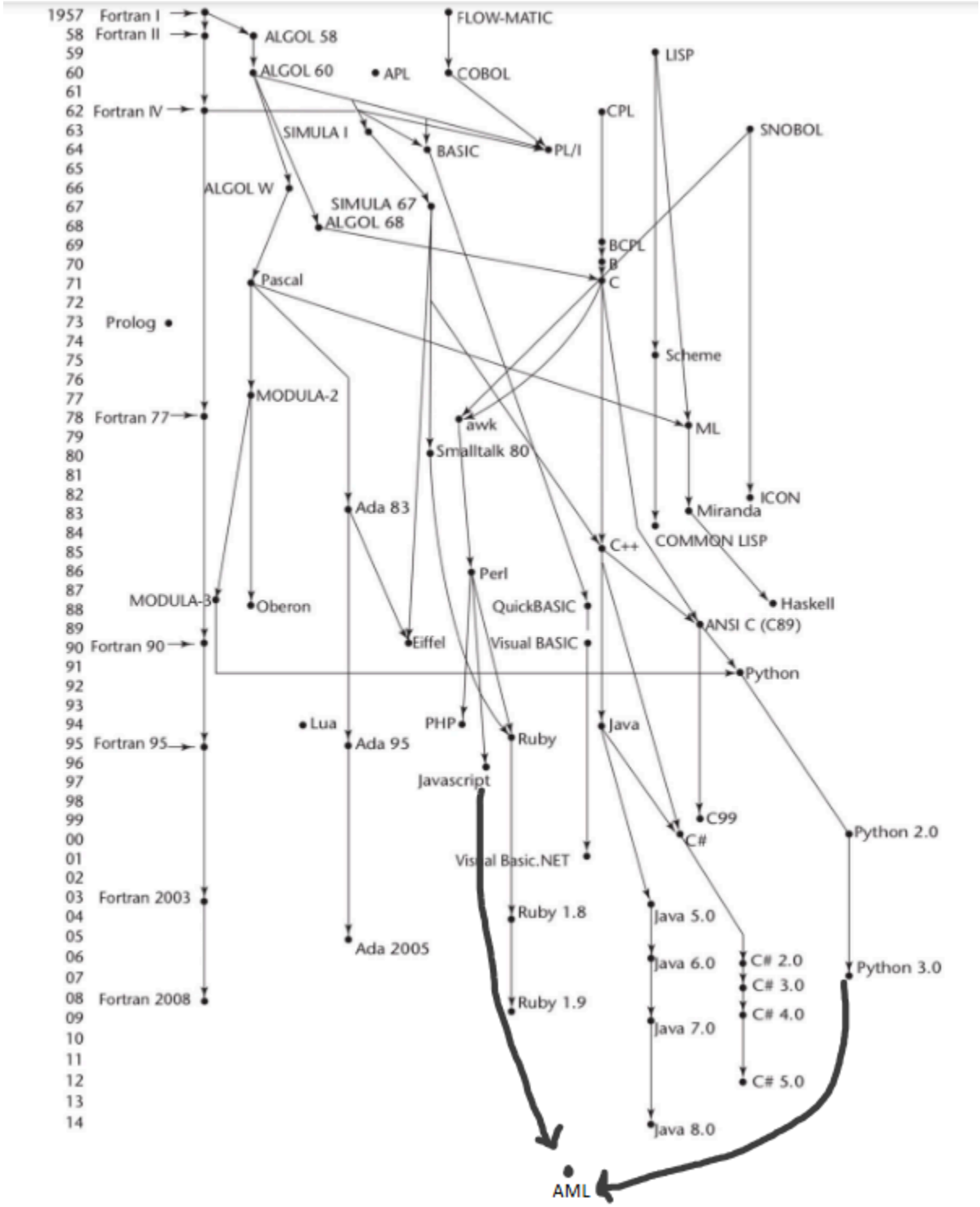
Version 0.0.1

1. Introduction

AML is a simple, modern and weakly typed functional language. This language is based on JavaScript and Python, but differs in the following ways:

1. It is purely functional
2. The syntax differs
3. There are different keywords
4. Code doesn't exist inside a class and anything typed is implied to be in "the main function" but that isn't explicitly stated
5. In order to call all code inside of "main" simply type RUN() (this must be the final line in your file).

1.1 Genealogy



1.2 Hello World

```
log(“Hello World”)

RUN()
```

1.3 Program Structure

The key organization concepts in AML are as follows:

1. Anything in scope 0 is considered the main program and will only execute if you type RUN() as the last line of your program.

2. New functions can be declared as “define *function_name*” and called with “*function_name*()”

3. Modules may be imported at top of file, in a comma separated list like so: “IMPORT = (“math”, “DIYmodule”)

4. Rather than getting dazed and confused with so many opening and close braces and ending each line with a semicolon or the like, we simply use tab to distinguish between each scope and no end of line markers are needed.

5. Single-line comments can be marked by “///” but no multi-line comments are supported.

```
IMPORT =  (“math”)

define calculate()
    x = 5
   squareroot(5)
   return x


y = calculate()
log(y)


RUN()
```

As shown above AML's program structure is very simple. We begin at scope 0, which we consider our main function. From there we import modules at this level as we all define and call any functions we want inside of it. RUN() must be the very last line of the file to call our main function (which is the entire file).

1.4 Types and Variables

AML supports both value and reference types just as both of its parent programs (JavaScript and Python) do. Value types directly store data within their variable, while reference types store a memory address (a reference) pointing to where the data is actually located. Variables may be reassigned in the same scope or in different scopes except for constants which must have the keyword CONSTANT prior to the variable in its declaration.

1.5 Visibility

While using keywords such as private or public on a function is irrelevant because AML is functional, all variables are public by default but can be changed to private if “PRIVATE” is explicitly stated directly prior to the first usage of that variable.

1.6 Statements Differing from JavaScript and Python

Statement	**Example**
Declaring a function	In order to declare a new function “define” is placed directly prior to the function name
Importing modules	As the absolute very first line of the file must be IMPORT = () where the parentheses represent a list that can be filled with module names inside double quotes separated by commas
Main Function	The main function is not declared as the entire file is automatically considered to be the “main.” To execute this, RUN() has to be the absolute very last line of the file.
Print	Displaying output to the console is done through log() where any strings outputted must be contained within double quotes
Math Functions	More advanced math functions that are built in include squareroot(x) and

	raisedtothepower(x, y) where x is the number we want to raise to a certain power and y is that power.
For & While Loops	They don’t exist since this language is functional

2. Lexical Structure

2.1 Programs

An AML **program** consists of only one source file as it is a simple language that strives to avoid being overly complex.

This single file is compiled like so:

1. Lexical Analysis: Here the sequence of tokens are evaluated to ensure that the words are correct and adhere to the language by matching its defined keywords and such.

2. Parsing: In this step, the tokens that are confirmed in lexical analysis are checked to see if they are conforming to the grammar in the order they were processed in.

3. Semantic Analysis: The meaning is then checked to make sure that it follows the language's semantics.

4. Code generation: Finally this high level language is converted into assembly code to be processed by the computer.

2.2 Grammars

2.2.1 Lexical Grammar (tokens) where different from JavaScript and Python

<Print> → log()

<Function Definition> → define()

<Comment> → ///

<Import> → IMPORT = ()

<Call Main> → RUN()

<Square Root> → squareroot()

< Power > → raisedtothepower()

2.2.2 Syntactic (“parse”) grammar where different from JavaScript and Python

<Program> → <ImportDecl>? <FunctionDecl>* <Statement>* RUN()

<ImportDecl> → "IMPORT" "=" "(" <StringList> ")"
<StringList> → <String> ("," <String>)*

<FunctionDecl> → "define" <Identifier> "(" <ParamList>? ")" NEWLINE INDENT
<Statement>*
DEDENT

<Assignment> → <Identifier> "=" <Expression>

<ReturnStmt> → "return" <Expression>

<LogStmt> → "log" "(" <Expression> ")"

2.3 Lexical analysis

2.3.1 Comments

Only **single-line comments** are supported with “///” which can either be placed on its own line or at the end of a line with valid code. **Multi-line comments** are not supported but you can just prefix each line with “///” if you want to comment over several lines.

2.4 Tokens

AML programs consist of the following token categories. Whitespace and comments (`/// …`) are not tokens but merely delimiters.

tokens:
- identifier
- keyword
- number-literal
- string-literal
- boolean-literal
- operator
- punctuator

2.4.1 Keywords different from Javascript and Python

A keyword is an identifier-like sequence of characters that is reserved and cannot be used as an identifier except when prefixed by the `@` character.

New keywords:
- IMPORT
- define
- RUN
- CONSTANT
- PRIVATE

Removed keywords:
- function
- var
- let
- const
- class
- def
- import
- for
- while
- do

3. Type System

AML employs a weak dynamic type system: types are attached to values at run time, implicit coercions are allowed, and there is no compile-time type checking.

3.1 Type Rules

$S \vdash e_1 : T$
$S \vdash e_2 : T$
T is a primitive type

$S \vdash e_1 = e_2 : T$

$S \vdash e_1 : T$
$S \vdash e_2 : T$
T is a primitive type

$S \vdash e_1 + e_2 : T$

$S \vdash e_1 : T$
$S \vdash e_2 : T$
T is a primitive type

$S \vdash e_1 - e_2 : T$

$S \vdash e_1 : Number$
$S \vdash e_2 : Number$

$S \vdash e_1 * e_2 : Number$

$S \vdash e_1 : Number$
$S \vdash e_2 : Number$

$S \vdash e_1 / e_2 : Number$

$S \vdash e_1 : T$
$S \vdash e_2 : T$

T is a primitive type

$S \vdash e_1 == e_2 : \text{Boolean}$

$S \vdash e_1 : T$
$S \vdash e_2 : T$
T is a primitive type

$S \vdash e_1 != e_2 : \text{Boolean}$

$S \vdash e_1 : \text{Number}$
$S \vdash e_2 : \text{Number}$

$S \vdash e_1 < e_2 : \text{Boolean}$

$S \vdash e_1 : \text{Number}$
$S \vdash e_2 : \text{Number}$

$S \vdash e_1 > e_2 : \text{Boolean}$

3.2 Value types (different from JavaScript and Python)

Number: Real numbers
Boolean: true, false
String: Sequences of characters in double quotes, e.g. "hello"
Function: First-class values created by `define`

3.3 Reference types (different from JavaScript and Python)

List: Sequences in brackets, e.g. [1,2,3]
Module: Imported namespaces via `IMPORT = (…)`

4. Example Programs

1. Caesar Cipher encrypt:

```
IMPORT = ()

define processChar ( c , shift )
    code = charCodeAt ( c , 0 )
    base = charCodeAt ( "A" , 0  )
    if code >= base and code <= base + 25
        offset = code - base
        newOffset = ( offset + shift ) %  26
        return fromCharCode ( base + newOffset )
    else
        return c

define caesarEncrypt ( text , shift )
    upperText = toUpperCase ( text )
    chars =  split ( upperText , "" )
    encryptedChars = map ( chars , define ( c ) processChar ( c , shift )  )
    return join (encryptedChars , ""  )

message = “Hello, World!"
shiftVal =  3
encrypted = caesarEncrypt ( message , shiftVal )
log ( encrypted )

RUN ()
```

2. Caesar Cipher decrypt:

```
IMPORT = ()

define caesarDecrypt( text, shift )
    upperText = toUpperCase( text )
    chars = split ( upperText , "" )
    decryptedChars = map ( chars , define ( c )
        code = charCodeAt ( c , 0 )
```

```
        base = charCodeAt ( "A" , 0 )
        if code >= base and code <= base + 25
            offset  = code - base
            newOffset  = ( offset - shift + 26) % 26  /// +26 to handle negative shifts
            return fromCharCode ( base + newOffset )
        else
            return c
    )
    return join ( decryptedChars , "" )

message  = "Khoor, Zruog!"
shiftVal  = 3
decrypted = caesarDecrypt ( message , shiftVal )
log ( decrypted )    /// should print "HELLO, WORLD!"

RUN()
```

3. Factorial

```
IMPORT = ()

define factorial ( n )
    if n == 0
        return 1
    else
        recursiveResult = factorial ( n - 1 )
        result = n * recursiveResult
        return result

number = 5
factVal = factorial ( number )
log ( factVal )    /// should print “120”

RUN()
```

4. Selection Sort

```
IMPORT = ()
```

```
/// Find the minimum element in a non-empty list
define findMin ( lst )
    if length ( lst ) == 1
        return lst [ 0 ]
    first = lst [ 0 ]
    rest = slice ( lst , 1 , length ( lst ) )
    restMin = findMin ( rest )
    if first < restMin
        return first
    else
        return restMin

/// Remove the first occurrence of val from lst
define removeFirst ( lst , val )
    if lst == []
        return []
    first = lst [ 0 ]
    rest  = slice ( lst , 1 , length ( lst ) )
    if first == val
        return rest
    else
        return [ first ] + removeFirst ( rest , val )

/// Selection-sort: repeatedly pick the minimum
define sort ( lst )
    if lst == [] or length ( lst ) == 1
        return lst
    minVal  = findMin ( lst )
    remaining  = removeFirst ( lst, minVal)
    sortedRest = sort ( remaining )
    return [ minVal ] + sortedRest

numbers = [ 5 , 3 , 8 , 1 , 2 ]
sortedNumbers = sort ( numbers )
log ( sortedNumbers )   /// prints [1,2,3,5,8]

RUN()
```

5. Square each Number in a List

```
IMPORT = ()
```

```
numbers = [ 1 , 2 , 3 , 4 , 5 ]
squares = map ( numbers , define ( x )
    return raisedtothepower ( x , 2 )
)
log ( squares )    /// prints [1,4,9,16,25]

RUN()
```

6. Stack

```
IMPORT = ()

define push ( stack , val )
    return [ val ] + stack

define pop ( stack )
    if stack == []
        return []
    return slice ( stack , 1 , length ( stack ) )

define peek ( stack )
    if stack == []
        return null
    return stack [ 0 ]

/// main
stk0 = []
stk1 = push ( stk0 , 10 )
stk2 = push ( stk1 , 20 )
top = peek ( stk2 )
stk3 = pop ( stk2 )

log ( "Top: " + top )       /// prints Top: 20
log  ( "After pop: " + stk3 )  /// prints After pop: [10]

RUN()
```