The Design of an Experimental Programming Language and its Translator

The Nuua Programming Language

Èrik Campobadal Forés

June 19, 2019

Universitat Politècnica de Catalunya

Table of Contents

Introduction

1.1 Introduction: Objectives

MAIN OBJECTIVES

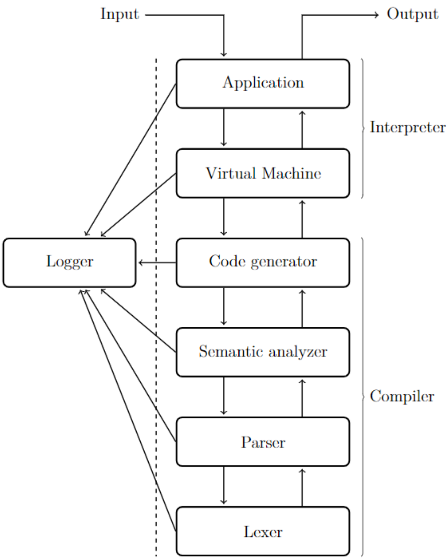
- Design an experimental programming language.
- Build a compiler and an interpreter for the language.
- Use a robust system architecture.
- Build a small standard library.

1.1 Introduction: Objectives

ADDITIONAL OBJECTIVES

- The interpreter must be memory efficient and performant.
- Cross platform support (at least on windows and linux).
- Functions, Modules and Classes.
- Static type safety.

1.2 Introduction: System Architecture



1.3 Introduction: The Nuua Programming Language

What is Nuua?

Nuua is a general-purpose high level programming language with an imperative paradigm and a statically typed system.

```
fun main(argv: [string]) {
    print "Hello, World"
}
```

1.3 Introduction: The Nuua Programming Language

```
class Triangle {
    b: float
    h: float
    fun area(): float -> (self.b * self.h) / 2.0
}

fun main(argv: [string]) {
    t := Triangle!{b: 10.0, h: 5.0}
    print "The area is: " + t.area() as string
}
```

```
fun rec_fib(n: int): int {
    if n < 2 => return n
    return rec_fib(n - 2) + rec_fib(n - 1)
}

fun main(argv: [string]) {
    print rec_fib(25)
}
```

```
use list_int_map from "list"

class Collection {
    numbers: [int]
    fun map(f: (int -> int)): Collection {
        list_int_map(self.numbers, f)
        return self
    }
}

fun multiply(n: int): int -> n * 2

fun main(argv: [string]) {
    c := Collection!{numbers: [1, 2, 3, 4, 5]}
    c.map(multiply).map(multiply)
    print c.numbers
}
```

1.4 Introduction: Error Logging

```
> C:\Presentation\code\high.nu, line: 3, column: 1

  Parsing 'class' declaration

  class Collection whops {
  ^


> C:\Presentation\code\high.nu, line: 3, column: 18

  Expected '{' after 'class' name.

  class Collection whops {
                   ^
```

Figure 1 – Example of error logging

1.4 Introduction: Error Logging

INFORMATION NEEDED

1. The module name (source file).
2. The line.
3. The column.
4. The error message.

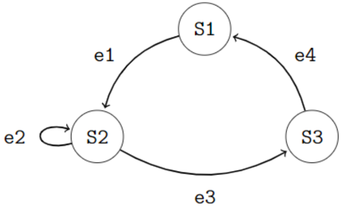
Lexical Analysis

Lexical Analysis

Transform a list of characters found in the source file into a list of tokens defined in the language.

Figure 2 – Lexical analysis overview

2.1 Lexical Analysis: Scanning the Source File



(i) Diagram

State	Explanation
S1	Initial state of the state machine.
S2	Build string. `s += c`
S3	Create token

(ii) State table

Event	Condition	Action
e1	`c == '"'`	`c = next_char()`
e2	`c != '"'`	`c = next_char()`
e3	`c == '"'`	–
e4	–	`c = next_char()`

(iii) Event table

Figure 3 – Example finite state machine for strings

2.1 Lexical Analysis: Scanning the Source File

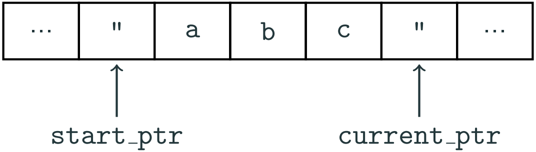


Figure 4 – Lexer scan technique

$$\texttt{length} = \texttt{current\_ptr} - \texttt{start\_ptr} + 1$$

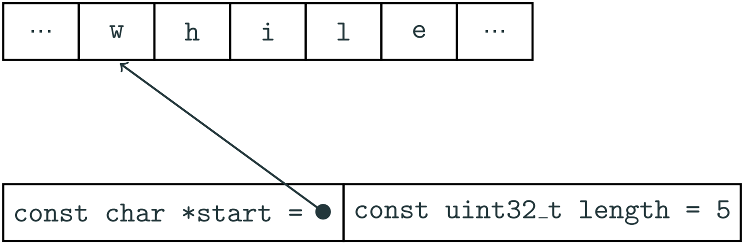


Figure 5 – Token instance

Syntactic Analysis

Syntactic Analysis

Transform the list of tokens returned by the Lexer into an abstract syntax tree.

Figure 6 – Parser overview

3.1 Syntactic Analysis: Technique

MUST BE

- Controlable (for error reporting).
- Easy to build.
- Fast enough.
- Handwritten (no generators).

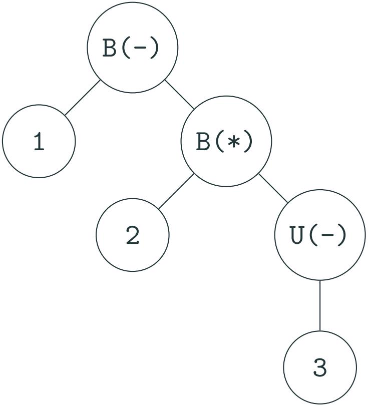
CANDIDATES

- Recursive descend parser.
- Pratt parser.

3.2 Syntactic Analysis: Abstract Syntax Tree

```
1 - 2 * -3
```

(i) Program



(ii) Nuua AST

Figure 7 – Example abstract syntax tree

```
fun main(argv: [string]) {
    print 1 - 2 * -3
}
```

(i) Program

```
Function[main:
↪   <no-return>]
  Print
    Binary[-]
      Integer
      Binary[*]
        Integer
        Unary[-]
          Integer
```

(ii) Nuua AST

Figure 8 – Code example abstract syntax tree

PATH SYSTEM

- Modules can be relative.
- Modules can be absolute.
- Modules can be on different paths.

Semantic Analysis

Semantic Analysis

Append information to the AST and perform checks to ensure it's a valid program. If this checks succeed, the program is considered valid.

IMPLEMENTED

- Creation of the symbol table for each module and block scope.
- Expression type inference.
- Static type checking.
- Node information attachment.

ADDITIONAL CHECKS

- Variable declaration.
- Function return value match.
- Valueless return types.
- Argument type match.
- Assignment type match.
- Iterator index type.
- Variable lifetime.
- Iterator check on required nodes.
- `main([string])` required on the main module.

Code Generation

Code Generator

Transforms the AST into bytecode instructions ready to be executed by the virtual machine.

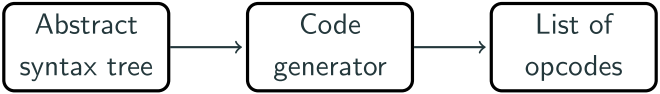


Figure 9 – Code generator overview

5.1 Code Generation: Instructions

IMPLEMENTED

- Bytecode instructions similar to hardware instructions.
- 113 register-based instructions.
- Constant pool and globals creation.
- Frame sizes (registers needed).
- Optimizations
- Position independent functions.

5.2 Code Generation: Bytecode and Source File Relationship

PROBLEM

- Runtime exceptions require the source file information.
- Requires the file name, the line and the column.
- Each opcode require this information because it can fail.
- Very memory inefficient.

SOLUTION

- Register only the changes on the file, line or column.
- Guess the file, line and column based on the current opcode.

5.2 Code Generation: Bytecode and Source File Relationship

```
a: int = 10
b: int = a - 10
print a / b
```

(i) Input program

```
LOAD_C   R-00001 C-00000
LOAD_C   R-00003 C-00001
SUB_INT  R-00002 R-00001 R-00003
DIV_INT  R-00003 R-00001 R-00002
PRINT    R-00003
```

(ii) Bytecode generated

CONDITION

Highest index that is lower or equal to the crash index.

Opcode Index	Line number
0	1
3	2
10	3

(iii) Registered line changes

Figure 10 – Runtime exception

Optimizations

6.1 Optimizations: Constant Folding

IMPLEMENTED

- Very basic constant folding on lists and dictionaries.

IMPROVEMENT

- Reduce the number of opcodes.
- Improve runtime performance.

```
print [1, 2, 3]
a: int = 3
print [1, 2, a]
```

(i) Input program

```
PRINT_C C-00000
LOAD_C  R-00001 C-00001
LOAD_C  R-00002 C-00002
LPUSH_C R-00002 C-00003
LPUSH_C R-00002 C-00004
LPUSH   R-00002 R-00001
PRINT   R-00002
```

(ii) Optimized bytecode generated

Figure 11 – List constant folding

6.2 Optimizations: Register allocation

PROBLEM

- How long is the value of a register needed?
- Can we re-use the registers?

IMPLEMENTED

- Linear scan register allocation.

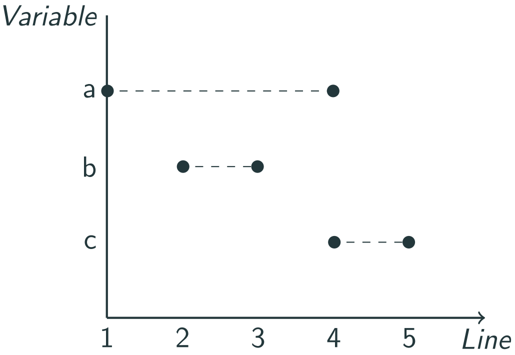
IMPROVEMENT

- Significant memory reduction.

6.2 Optimizations: Register allocation

```
a: int = 10
b: int = 20
print a + b
c: int = a
print c
```

(i) Input program



(ii) Variable lifetime

Figure 12 – Variable lifetime of a program

6.2 Optimizations: Register allocation

```
a: int = 10
b: int = 20
print a + b
c: int = a
print c
```

(i) Input program

```
LOAD_C  R-00000 C-00000
LOAD_C  R-00001 C-00001
ADD_INT R-00001 R-00000 R-00001
PRINT   R-00001
MOVE    R-00001 R-00000
PRINT   R-00001
```

(ii) Optimized bytecode generated

Figure 13 – Register allocation optimization of a program

Virtual Machine

7.1 Virtual Machine: Value Stack and Call stack

Virtual Machine

Execute the bytecode instructions generated.

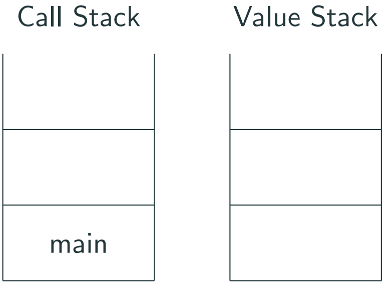
IMPLEMENTED

- A call stack.
- A value stack to pass parameters.
- Automatic call to the `main` function with `argv` argument.

```
fun mul(a: int): int {
    return a * 2
}

fun main(argv: [string]) {
    print mul(10)
}
```

(i) Input program



(ii) Stacks

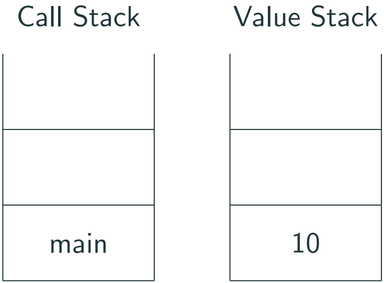
Figure 14 – Stack usage on program execution

7.1 Virtual Machine: Value Stack and Call stack

```
fun mul(a: int): int {
    return a * 2
}

fun main(argv: [string]) {
    print mul(10)
}
```

(i) Input program



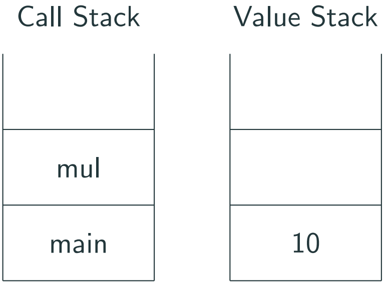
(ii) Stacks

Figure 14 – Stack usage on program execution

```
fun mul(a: int): int {
    return a * 2
}

fun main(argv: [string]) {
    print mul(10)
}
```

(i) Input program



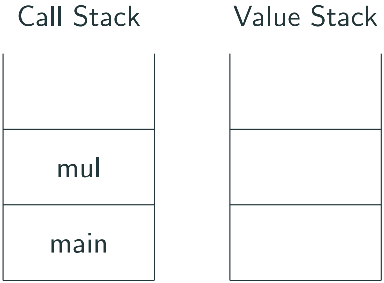
(ii) Stacks

Figure 14 – Stack usage on program execution

```
fun mul(a: int): int {
    return a * 2
}

fun main(argv: [string]) {
    print mul(10)
}
```

(i) Input program



(ii) Stacks

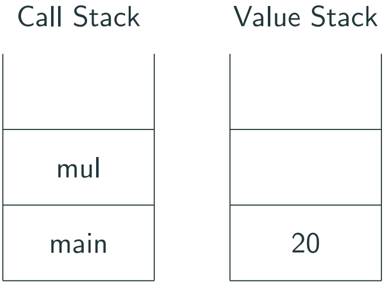
Figure 14 – Stack usage on program execution

7.1 Virtual Machine: Value Stack and Call stack

```
fun mul(a: int): int {
    return a * 2
}

fun main(argv: [string]) {
    print mul(10)
}
```

(i) Input program



(ii) Stacks

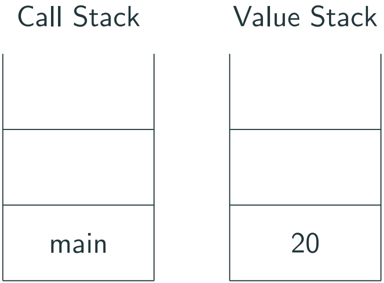
Figure 14 – Stack usage on program execution

7.1 Virtual Machine: Value Stack and Call stack

```
fun mul(a: int): int {
    return a * 2
}

fun main(argv: [string]) {
    print mul(10)
}
```

(i) Input program



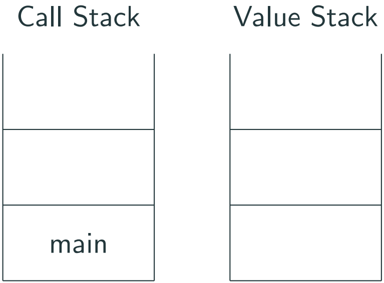
(ii) Stacks

Figure 14 – Stack usage on program execution

```
fun mul(a: int): int {
    return a * 2
}

fun main(argv: [string]) {
    print mul(10)
}
```

(i) Input program



(ii) Stacks

Figure 14 – Stack usage on program execution

7.2 Virtual Machine: Instruction Dispatch

PROBLEM

- Threaded dispatch is not ANSI C compilant.

SOLUTION

- Use a switch dispatch

TRADEOFF

- ANSI C compilant.
- Less efficient than a threaded dispatch.
- Simple to implement.

Application

8.1 Application: Job

Application

Setup the application and fire up the virtual machine.

IMPLEMENTED

- Command line argument parsing.
- Extension to multiple application types (Prompt, stdin, File, etc.).
- Creation of the initial `argv: [string]` argument for the `main` function.

Conclusions

9.1 Conclusions: General Objectives

IN GENERAL

- Primary objectives completed.
- Competitive and performant interpreter.
- Benchmarking is hard.

FURTHER EVOLUTION

- C foreign function interface (FFI).
- Propper I/O interface.
- Function overloading and generics.
- Extended standard library.
- Website (`https://nuua.io`) and package manager.

9.2 Conclusions: Curiosities

- $5k$ `C++` lines.
- $2.3k$ `C++` header lines.
- $1.4k$ Comment lines.
- $7.5k$ Total project lines.
- $500KB$ Total release binary size (Windows).
- Nuua is available for Linux and Windows under the MIT license at `http://nuua.io/latest`