NAIGAMA PARSING SYSTEM (COMPENDIUM)

KEES-JAN HERMANS

This document contains a description of Naigama, a software implementation of a Parsing Expression Grammar compiler, assembler, and bytecode execution engine.

Accompanies release	0.3.9
Author	Kees-Jan Hermans / kees.jan.hermans@gmail.com
Classification	-
Generated on	July 9, 2022

Contents

1. Introduction

1.1. **Parsing Expression Grammars.** This is the documentation describing Naigama, which is a collection of specifications, libraries and tools enabling data parsing. It is based, by now loosely, on Lua Parsing Expression Grammars, or LPEG [REF], which implements a Packrat [REF] parsing algorithm.

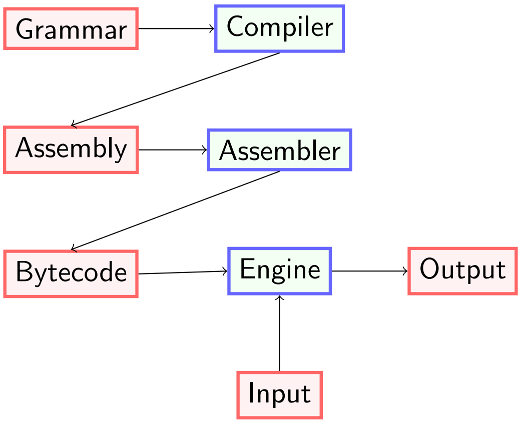
1.2. **Naigama.** Naigama adds to Lua PEG the following features:

- Discoverability features:
  - A fully fledges assembler step in between grammar and bytecode.
  - A capable engine debugger.
- Extra tools:
  - A disassembler.
  - A decompiler.
- Security features:
  - Bit upset event detection and traps.
  - Endless loop detection.
- Interoperability with C (library, header files).
- As well as Java, Perl and Rust.
- Added instructions for matchers (ranges, quads).
- Added instructions for counters (to enable efficient looping).
- Added instructions for parsing binary inputs.

2. Overview

This document describes the Naigama parsing system, which is a modular system to process structured arbitrary length data inputs, in order to extract meaning (matching, capturing from data), or to manipulate (replacements within data). The system was designed with a focus on information and system security.

The system comes in three parts: a grammar compiler, an assembler, and a bytecode execution engine. Each of the parts has their own input file format specification (grammar description, assembly specification, and bytecode specification), and outputs the format of the next stage. The process in brief: the compiler takes a grammar description that the human end user has made, and turns it into an assembly language. The assembler takes the assembly, and turns it into a bytecode. The engine takes the bytecode and the input, and processes it, resulting either in (various measures of) failure or success.



Schematic overview of Naigama transformations: data formats (red) and tools (blue).

The reason for this modularity is strict separation of tasks, and openness: anyone should be able to take out a module, and replace it with one of their own. It should be expressly possible to take out the compiler, and replace it with another tool that produces the Naigama assembly, for example. Or have a different engine. Or write an optimizer for the assembly.

2.1. **On Grammar.**

Naigama grammar is the human interface of the system. It can be used to define complex syntax definitions in order to match, capture from, and replace in, structured inputs.

People familiar with regular expressions [2], Backus-Naur syntax descriptions [3], and / or Lex and Yacc tools [4], should find this part of the document relatively easy to understand. The ideas underlying Naigama grammar, assembly and bytecode are heavily borrowed from Lua PEG (LPEG) [1] by Roberto Ierusalimschy et al.

2.2. **On Assembly.** The Naigama assembly language is a mixture of two languages: For one, it is the outcome instruction set, according to the aforementioned LPEG whitepaper, of the Naigama grammar compilation process. However, it also contains instructions that won't be produced by this compiler, but may either be produced by a separate optimizer, or other compilers altogether (for example, one that focuses on network packet parsing).

The Naigama assembly language, in all cases, is the human readable variant of the Naigama bytecode, and its instructions are one-on-one translatable from the one to the other, and vice versa (although labels will be lost in the reverse process).

2.3. **On Bytecode.** Naigama bytecode is the machine interface of the system. It runs in the Naigama engine against the input provided by the user. It is designed to be:

- Easy and quick to interpret.
- Resilient against bit upset events.
- Resilient against endless loops.
- Usable while keeping both bytecode and input as read-only buffers.

3. Generations

Naigama is written in generations. That is to say: each subsequent generation of Naigama uses the tools generated in the previous one, to build itself ('the grammar parses the grammar').

The following generations exist, with the following functions:

- Generation zero: This generation consists solely of a bootstrap grammar compiler and assembler written in perl.
- Generation one: This generation is built in C, based on the bytecode to parse grammar grammar and assembly grammar, generated by generation zero.
- Generation two: equal to generation one, but then based on the bytecode generated by the tooling in generation one. Generation two is 'pure' in that it has compiled itself, and it only provides parsing.
- Generation three: This generation extends the concept of the language further in that it also allows scripting. The parser of the scripting language is not 'active' however, because it's performed by generation two. It's equivalent to generation zero, in that it is the bootstrap compiler for scripting.
- Generation four: This generation has the compiler and assembler defined as active scripts, including grammar. There is no support required for their execution, save for the engine's implementation in C.
- Generation five: This generation is 'pure' again, in that it is equivalent to generation four, but then compiled by itself.

4. Grammar

A Naigama grammar file or buffer consists either of a single unnamed expression, or of a sequence of named expressions (called rules), whitespace and comments, denoted in ASCII text. Naigama grammar is defined with the express purpose of matching, validating and/or capturing from an arbitrary data input. Naigama grammar is taken by the Naigama compiler program (naic) and turned into Naigama assembly [section 5]. In this, it resembles greatly other grammar description and parsing methods such as Backus-Naur, Regular Expressions, Lex and Yacc.

```
-- JSON; JavaScript Object Notation.
TOP          <- JSON
__prefix     <- %s*
JSON         <- HASH END
HASH         <- CBOPEN OPTHASHELTS CBCLOSE
OPTHASHELTS  <- HASHELTS / ...
HASHELTS     <- HASHELT COMMA HASHELTS / HASHELT
HASHELT      <- STRING COLON VALUE
ARRAY        <- ABOPEN OPTARRAYELTS ABCLOSE
OPTARRAYELTS <- ARRAYELTS / ...
ARRAYELTS    <- VALUE COMMA ARRAYELTS / VALUE
VALUE        <- STRING / FLOAT / INT / BOOL / NULL / HASH / ARRAY
STRING       <- { ’"’ ( ’\\’ ([nrtv"] / [0-9]^3) / [^"\\] )* ’"’ }
INT          <- { [0-9]+ }
FLOAT        <- { [0-9]* ’.’ [0-9]+ }
BOOL         <- { ’true’ / ’false’ }
NULL         <- { ’null’ }
CBOPEN       <- ’{’
CBCLOSE      <- ’}’
ABOPEN       <- ’[’
ABCLOSE      <- ’]’
COMMA        <- ’,’
COLON        <- ’:’
END          <- !.
```

Example of a Naigama grammar meant to validate and capture from JSON[6] *text.*

4.1. **Caveats.** This section highlights a few things to keep in mind when using Naigama, or PEGs in general.

4.1.1. *Greedy and Blind Behavior.* PEG is, by itself, both greedy and blind. That is to say, it consumes as much input as possible (greedy), and it does not look ahead whether a next token is perhaps a better match for the input (blind). It can however, also implement non-blind behavior, both in a greedy as well as a non-greedy manner.

Having a greedy, blind parser - standard PEGs - implies that the following rule will always fail (because the '.*' will consume all input):

```
RULE <- .* 'foo'
```

Grammars or patterns like the above, are usually treated in a more friendly way by, for example, Regex parsers. They implement a recursive algorithm around 'does the whole pattern match' question, which backtracks 'from zero to any amount' quantifiers such as the one used above, leaving the user with a much more freedom to implement patterns but, in the end, a much more unpredictable system (for example, what if 'foo' happens twice in the text?)

Naigama is not so understanding. To make it match something while also doing a lookahead (therefore 'non blind'), you can implement the following pattern recursively (non greedy, so catching the first lookahead match):

```
S <- E2 / E1 S
```

Or, not recursively:

```
S <- ( !E2 E1 )* E2
```

And greedy, so catching only the last lookahead match:

```
S <- E1 S / E2
```

4.1.2. *Recursiveness.* Just like with greedy matching, Naigama will simply jump headlong into your rule's first matcher, and will not try to second-guess your intentions. So while in many grammar systems, left recursive is usually considered the best way to describe repetition, PEG will get stuck in an endless loop if you do the following:

```
S <- S SOMEOTHER
```

4.2. **Comments.** A comment starts with two minus signs and ends at the end of the line.

```
-- Example of a single line comment.
```

A multiline comment is also avaiable: these start with two minus signs followed by two angle braces. The multiline comment is closed by two closing angle braces.

```
--[[
     Example of a
     multiline comment.
  ]]
```

4.3. **Rules.** A rule is defined as an *Identifier*, followed by a left-pointing arrow (composed of a less-than and a minus sign), followed by a matching expression.

When a Naigama grammar consists of a sequence of rules (as opposed to a single line expression), the first rule is used as the starting point for matching inputs.

```
RULE1 <- ’a’ / RULE2
RULE2 <- ’b’ / RULE3
RULE3 <- ’c’
```

Definition of a set of rules

4.3.1. *Identifiers.* Identifiers, in Naigama, are defined as a combination of letters, numbers and the underscore character, not starting with a number, of between one and 64 characters long.

```
IDENTIFIER <- [a-zA-Z_][a-zA-Z0-9_]^-63
```

Identifiers are used to start rule definitions, and as references to rules in expressions.

4.3.2. *Special Rules.* There is currently one special rule in Naigama grammar: a rule called '__prefix' will be treated specially. This rule will be, from its definition onwards, called before the execution of any subsequent rule definition expression. The aim is to make whitespace and comment filtering in program language parsing easier. See also the earlier 'JSON' grammar example.

4.4. **Expressions.** Expressions are lists of terms, optionally separated by the OR operator (denoted by the forward slash sign).

```
ALTERNATIVES <- ALT1 / ALT2 / ALT3
```

A sequence of terms has a higher precendence than the OR operator, so any other desired order of precedence has to be forced by using brackets. In the example below, the expressions evaluate differently:

```
RULE1 <- ’a’ ’b’ / ’c’
RULE2 <- ’a’ ( ’b’ / ’c’ )
```

4.5. **Terms.** A term is defined as a matcher, potentially adorned with prefix or postfix operators (though not at the same time). Prefix operators are the NOT and CONFIRM modifiers. Postfix operators are the quantifiers.

4.5.1. *The NOT Modifier.* The NOT modifier is an exclamation mark. It does not advance the input position, but succeeds when matching fails (including reaching the end of input), and fails when matching succeeds. Since Naigama is a greedy parser, this can be used to implement a non-greedy lookahead, for example:

```
CMULTILINECOMMENT <- ’/*’ (!’*/’ .)* ’*/’
```

4.5.2. *The CONFIRM Modifier.* The CONFIRM modifier is an ampersand. It does not advance the input position, and succeeds as matching succeeds, and fails when matching fails. It can be used to process (parts of the) input twice. It is equivalent to using the NOT modifier twice.

Example where the input is first confirmed to be valid UTF-8, and only after that, processed again from the beginning, for well-formedness.

```
JSON <- & UTF8 WELLFORMED
```

4.5.3. *Quantifiers.* Naigama denotes quantifiers using a circonflex, followed by either an absolute number, or a range. Shorthand exists for certain, often-used ranges. These shorthand ranges are common in Regex as well:

Naigama Quantifiers

Quantifier	**Semantics**
*	Zero or more matches
+	One or more matches
?	Zero or one matches
ˆ*n*	*n* matches
ˆ-*n*	Zero up to and including *n* matches
ˆ*n*-	*n* matches or more
ˆ*n*-*m*	From *n* up to and including *m* matches

Example of the use of quantifiers:

```
RULE1 <- ’a’? ’b’ˆ-5
```

4.6. Matchers.

4.6.1. *The ANY Matcher.* The ANY matcher is denoted (like in Regex) with a single dot. It matches any character of input and it succeeds - advancing the input pointer by one - if it can find one at the input pointer. It fails only at the end of input.

```
RUNGREEDILYTOTHEENDOFINPUT <- .*
```

4.6.2. *The SET Matcher.* The ANY matcher, together with the NOT modifier, also allows you to define end-of-input, like so:

```
ENDOFINPUT <- !.
```

4.6.3. *The SET Matcher.* The SET matcher is denoted, like in Regex, as a series of literals and ranges, enclosed between angled brackets. It can also be negated, in which case it matches any character not in the set.

```
ALPHABETIC     <- [a-zA-Z]
NONALPHABETIC  <- [ˆa-zA-Z]
```

The denotation of the set elements has a backslash as escape character (to encode the minus and the closing angled brace). It also has a form of binary escaping, to encode non-ASCII characters as part of the set: a backslash followed by three digits of the characters octal notation.

4.6.4. *The STRING Matcher.* The STRING matcher is denoted as a string literal, between single quotes. The string may be postfixed with an lower case ’i’ to indicate (alphabetic) case insensitive matching.

```
CASEINSENSTIVESTRING <- ’peg’i
```

4.6.5. *The BITMASK Matcher.* The BITMASK matcher allows you to match bits.

4.6.6. *The HEXLITERAL Matcher.* The HEXLITERAL matcher allows you to match single characters by through their hexadecimal representation.

4.6.7. *The MACRO Matcher.* All macroes are shorthand equivalents of the SET matcher.

Naigama Macroes

Macro	**Semantics**	**Set**
%s	Whitespace	[\n\r\t\v]
%w	Alphabetic	[a-zA-Z]
%a	Alphanumeric	[a-zA-Z0-9]
%n	Numeric	[0-9]

4.6.8. *The GROUP Matcher.*

4.6.9. *The CAPTURE Matcher.*

4.6.10. *The VARCAPTURE Matcher.*

4.6.11. *The VARREFERENCE Matcher.*

4.6.12. *The REFERENCE Matcher.*

4.6.13. *The LIMITEDINPUT Matcher.*

5. Assembly

A Naigama assembly file or buffer consists of sequences of whitespace, comments, labels, and (parametrized) instructions, separated by new lines, and denoted in ASCII text.

Naigama assembly is taken by the Naigama assembler program (naia) and turned into bytecode.

```
-- Compilation of: TEST <- { ’a’ } { ’a’ } { ’a’ / ’b’ }
  call TEST
  end
-- Rule
TEST:
  opencapture 0
  char 61
  closecapture 0 0
  opencapture 1
  char 61
  closecapture 1 0
  opencapture 2
  catch __LABEL_30 -- alternative
  char 61
  commit __LABEL_31
__LABEL_30:
  char 62
__LABEL_31:
  closecapture 2 0
  ret
```

Example of a piece of Naigama assembly

5.1. Comments.

Just like in Naigama grammar, comments in Naigama assembly start with two minus signs, and end at a new line. They can be given on separate lines, or they can be postfixed to instructions or labels.

5.2. Labels.

Labels are identifiers (the same identifiers as in their definition in the grammar chapter) or numbers, followed by a colon sign. They are used as positions for the bytecode to jump to, when used by instructions.

When performing the assembly, the Naigama assembler program resolves each label to their offset in the resultant bytecode, and replaces each reference to a label with that offset.

Labels don't quite disappear though - you have the option to make the assembler program emit a so called 'labelmap' file, which stores the old label names, mapped to the bytecode offsets given by the assembler, and which can be used later for debugging purposes by the bytecode execution engine.

When disassembling (bytecode to assembly), each instruction is always prefixed by a label in the form of the instruction's offset in decimal, so that jumps are always correct (but non descriptive).

5.2.1. *Special Labels.* Certain instructions allow a special label called '__NEXT__' to be used as parameter. Its function is to make instruction simply jump to the next instruction on success. This special label is there to avoid generating a special purpose label only to put it right after the instruction using it.

The instructions allowing the use of the '__NEXT__' label are:

- backcommit
- commit
- condjump
- partialcommit
- testany
- testchar
- testquad
- testset

5.3. **Instructions.**

Naigama Assembly Instructions

Mnemonic	**Param1**	**Param2**
'any'		
'backcommit'		LABEL
'call'		LABEL
'catch'		LABEL
'char'	char	
'closecapture'	slot	
'commit'		LABEL
'condjump'	register	LABEL
'counter'	register	value
'end'	code	
'endreplace'		
'fail'		
'failtwice'		
'jump'		LABEL
'maskedchar'	char	mask

Mnemonic	**Param1**	**Param2**
'noop'		
'opencapture'	slot	
'partialcommit'		LABEL
'quad'	quad	
'range'	from	until
'replace'	slot	LABEL
'ret'		
'set'	set	
'skip'	number	
'span'	set	
'testany'		LABEL
'testchar'	char	LABEL
'testquad'	quad	LABEL
'testset'	set	LABEL
'trap'		
'var'	slot	

See table [5.3].

5.4. **Parameters.** Parameters follow the instruction in assembly text. They are separated from the instruction and any other parameters by spaces. The following types of parameters exist:

5.4.1. *Characters.* Character parameters are denoted as two byte hexadecimal values.

5.4.2. *Quads.* Quad parameters are denoted as eight byte hexadecimal values.

5.4.3. *Sets.* Set parameters are denoted as 64 byte hexadecimal values, representing a bitmask of 256 possible booleans.

5.4.4. *Registers, Slots, Codes and Numbers.* These parameters are all denoted as decimal numbers.

5.4.5. *Caveats.* Note that the order of parameters sometimes reverses between the assembly and bytecode specification. This is done because in assembly, it is considered more aesthetically pleasing to have labels at the end of an instruction, while in bytecode they are usually given as the first argument.

Consider, for example, the 'testchar' instruction, which in assembly is given as:

```
TESTCHARINSTR <- { ’testchar’ } S HEXBYTE S LABEL
S <- %s+
HEXBYTE <- { [0-9a-fA-F]^2 }
LABEL <- { [a-zA-Z0-9_]^1-256 }
AMPERSAND <- ’&’
```

Whereas in bytecode, this instruction is encoded as:

00	00	08	03	9a
04	00	00	00	00
08	00	00	00	00

Where:

Bytes 0-3 denote the instruction opcode.

Bytes 4-7 denote the bytecode offset to jump to, after this instruction (big endian 32 bit unsigned).

Bytes 8-11 denote the character to match.

6. Bytecode

6.1. **Instruction sets.** Naigama, generation 3, provides two instruction sets (with a few overlapping instructions) that each have their own stack and other memories: one for matching, and one for script execution.

Naigama Bytecode Instructions

Mnemonic	**Opcode**	**Param1**	**Param2**	**Length**
any	000003e4			4
backcommit	000403c0	address		8
call	00040382	address		8
catch	00040393	address		8
char	000403d7	char		8
closecapture	00040300	slot		8
commit	00040336	address		8
condjump	00080321	register	address	12
counter	00080356	register	value	12
end	000400d8	code		8
endreplace	00000399			4
fail	0000034b			4
failtwice	00000390			4
jump	00040333	address		8
maskedchar	00080365	char	mask	12
noop	00000000			4
opencapture	0004039c	slot		8
partialcommit	000403b4	address		8
quad	0004037e	quad		8
range	000803bd	from	until	12
replace	00080348	slot	address	12
ret	000003a0			4
set	002003ca	set		36
skip	00040330	number		8
span	002003e1	set		36
testany	00040306	address		8
testchar	0008039a	address	char	12
testquad	000803db	address	quad	12
testset	00240363	address	set	40
trap	ff00ffff			4
var	000403ee	slot		8

6.2. **Bytecode Structure.** A Naigama bytecode file or buffer consists of a sequence of binary instructions which, in turn, each consist of a binary encoded opcode, plus their parameters, should they have any.

The amount and kind of parameters following an opcode, is strictly defined: the same opcode will always be followed by the same kinds of parameters and therefore, an instruction type will always be the same size (see table [6.1]).

Naigama bytecode is taken by the Naigama engine program (naie) or library, and run against an input, to produce an output.

6.3. **Opcode Values.** Opcode values are determined through

- Grouping;
- Hamming distance;
- Instruction size;

6.4. **Noop Slides and Canaries.** Implementations that want each instruction to have exactly the same size, can choose to pad the encoding of shorter instructions with either no-ops, or canaries.

6.5. Encoding of Parameters.

6.5.1. *Address.*

6.5.2. *Char.*

6.5.3. *Slot.*

6.5.4. *Register.*

7. Output

8. Executables

8.1. The Compiler.

The Naigama grammar compiler is called 'naic'. It takes a grammar file as input, and outputs assembly text. It can be invoked as follows:

```
$ naic -i myfile.niag -o myfile.asm
```

Example of the invocation of the Naigama grammar compiler

Bear in mind that both the input (grammar) file, as well as the output (assembly) file may be omitted (in which case they will be assumed to be stdin and stdout, respectively), or be denoted as a minus sign ('-'), which will have the same effect.

You may also use the following options:

- `-m <path>` Tells the compiler to emit the slotmap file in <path>.
- `-D` Creates a lot of debugging output.
- `-t` Tells the compiler to surround generated rule code with 'trap' instructions.

8.2. The Assembler.

The Naigama grammar assember is called 'naia'. It takes an assembly file as input, and outputs bytecode. It can be invoked as follows:

```
$ naia -i myfile.asm -o myfile.byc
```

Example of the invocation of the Naigama assembler

Bear in mind that both the input (assembly) file, as well as the output (bytecode) file may be omitted (in which case they will be assumed to be stdin and stdout, respectively), or be denoted as a minus sign ('-'), which will have the same effect.

You may also use the following options:

- `-l <path>` Tells the assembler to emit the labelmap file in <path>.
- `-D` Creates a lot of debugging output.

8.3. The Engine.

The Naigama bytecode execution engine is called 'naie'. It takes a bytecode file as input, as well as an input file. It can be invoked as follows:

```
$ naie -c myfile.byc -i myfile.dat -o myfile.out
```

Example of the invocation of the Naigama engine

8.4. **The Disassembler.** The Naigama disassembler is called 'naid'. It takes a bytecode file as input, and outputs assembly.

```
$ naid -i myfile.byc -o myfile.asm
```

Example of the invocation of the Naigama disassembler

The output of the disassembler will differ from the assembly generated by the compiler in that:

- Textual labels will be gone, instead:
- Every instruction will be prefixed by a numeric label which is identical to that instruction's position offset in the bytecode, and
- All jumps will therefore also be using those position labels.

9. File Formats

This section describes all the user available memory and file formats associated with Naigama.

9.1. **Text Formats.**

9.1.1. *Grammar.* The Naigama grammar text format is descibed in [REF]. Various precompilers can potentially produce these format however, these are out of scope of this documentation. The grammar format is consumed by the Naigama assembler, naic. For the use of naic, see [REF].

9.1.2. *Assembly.* The Naigama assembly text format is descibed in [REF]. This format is produced by the Naigama compiler, naic, and is consumed by the Naigama assembler, naia. For the use of naia, see [REF].

9.1.3. *Slotmap.h.* The Naigama compiler assigns a slot number to each capture it encounters. It then assigns an internal unique name to those capture numbers, or slots. Finally, the compiler can be requested, through the command line, to 'rain down' these named mappings to slot number in a C-header file, so that API users can use easy-to-remember defines to address their captures.

9.1.4. *Disassembly.* The Naigama disassembler, naid, will take bytecode and reproduce the assembly. This assembly will be different from the original assembly in that:

- It will not contain textually understandable labels.
- It will prefix every instruction with a numeric label, which is equivalent to the bytecode offset of the instruction's opcode.
- These offsets will then also be used in jumps.

9.1.5. *Engine Debug Logs.* Running the Naigama engine, naie, in debug mode will yield, on standard error, a log, which will contain, per instruction executed, a line representing the engine's internal state, most notably its stack, like so:

```
CHAR bc 2316 in 482 0403020100______ st (014 prec.) ALT:1484
CLL:1476 C LL:1820 ALT:1944 CLL:1928 ALT:1652 CLL:1644 CLL:2440
```

These lines are built up as follows:

- The instruction being executed (in this case, 'char').
- The bytecode offset (in this case, 2316 decimal).
- The input offset (in this case, 482 decimal).

- A subset of the input, from the input offset (in this in hexadecimal).
- An overview of the stack (in this case, with 14 preceding items left out), followed by the last stack items, either 'call' items (with their return addresses), or 'alternative' items (with their jump-to addresses in case they catch a FAIL condition).

9.2. **Binary Formats.** This section describes the formats used by the Naigama tooling, other than the ones described in the sections on grammar, assembly and bytecode.

9.2.1. *Slotmap Format.* The slotmap file exists to help developers by easing access to captures in complex grammars.

The Naigama grammar compiler can be instructed to emit a slotmap file. This is a file which maps a unique name to a capture region's index. This is provided so that, instead of using the index of capture region (which requires hand counting them in your grammar file, which can be tedious and error-prone, and something that would not survive a grammar reshuffle, or the introduction of a capture region before the one you're interested in), you can use a naming scheme for your capture regions.

Names in the slotmap file are bound semantically to the capture region: they are made up of the name of the rule, an underscore, and all alphabetic characters in the capture region, cast to uppercase. Should any name occur twice, it will be postfixed with a counter until it's unique.

For example, the following rule with capture region definition:

```
RULE <- { IDENT } OPTARGS LEFTARROW EXPRESSION
```

will result in slotmap identifier 'RULE_IDENT'.

The binary format of a slotmap file is composed as a sequence of binary records: the slot index, denoted as a 32 bit network order unsigned integer, a field of 32 bit all ones, and the slot name, denoted as a zero-terminated string.

```
00 00 00 00 ff ff ff ff 52 55 4c 45 5f 49 44 45  ........RULE_IDE
4e 54 00 00 00 00 01 ff ff ff ff 45 58 50 52 45  NT.........EXPRE
53 53 49 4f 4e 5f 54 45 52 4d 53 00 00 00 00 02  SSION_TERMS.....
ff ff ff ff 45 58 50 52 45 53 53 49 4f 4e 5f 54  ....EXPRESSION_T
45 52 4d 53 5f 31 00 00 00 00 03 ff ff ff ff 45  ERMS_1.........E
58 50 52 45 53 53 49 4f 4e 5f 54 45 52 4d 53 5f  XPRESSION_TERMS_
32 00 00 00 00 04 ff ff ff ff 54 45 52 4d 53 5f  2.........TERMS_
54 45 52 4d 00 00 00 00 05 ff ff ff ff 54 45 52  TERM.........TER
4d 5f 4e 4f 54 41 4e 44 00 00 00 00 06 ff ff ff  M_NOTAND........
```

Example of the head of a slotmap file, hex dumped

9.2.2. *Labelmap Format.* The labelmap format is optionally emitted by the assembler, and exists to:

- Allow for better debugging, because offsets in bytecode can be reduced to more intuitively named sections (rule names always make it to the labelmap unchanged).
- Allow for calling the bytecode at symbols directly, which in turn allows you to use a bytecode blob more as a database of functions.

The binary format of the labelmap file is composed as a sequence of binary records: the bytecode offset, denoted as a 32 bit network order unsigned integer, followed by the labelstring, followed by a zero byte.

```
00 00 00 10 47 52 41 4d 4d 41 52 00 00 00 00 28  ....GRAMMAR....(
5f 5f 4c 41 42 45 4c 5f 31 31 00 00 00 00 38 5f  __LABEL_11....8_
5f 4c 41 42 45 4c 5f 31 32 00 00 00 00 40 5f 5f  _LABEL_12....@__
4c 41 42 45 4c 5f 36 00 00 00 00 48 5f 5f 4c 41  LABEL_6....H__LA
42 45 4c 5f 37 00 00 00 00 54 5f 5f 70 72 65 66  BEL_7....T__pref
69 78 00 00 00 00 5c 5f 5f 4c 41 42 45 4c 5f 32  ix....\__LABEL_2
39 00 00 00 00 7c 5f 5f 4c 41 42 45 4c 5f 34 33  9....|__LABEL_43
00 00 00 00 a8 5f 5f 4c 41 42 45 4c 5f 34 34 00  ......__LABEL_44.
00 00 00 b8 5f 5f 4c 41 42 45 4c 5f 33 32 00 00  ....__LABEL_32..
00 00 e4 5f 5f 4c 41 42 45 4c 5f 35 35 00 00 00  ...__LABEL_55...
01 10 5f 5f 4c 41 42 45 4c 5f 35 36 00 00 00 01  ..__LABEL_56....
10 5f 5f 4c 41 42 45 4c 5f 33 33 00 00 00 01 18  .__LABEL_33.....
5f 5f 4c 41 42 45 4c 5f 33 30 00 00 00 01 1c 45  __LABEL_30.....E
4e 44 00 00 00 01 34 5f 5f 4c 41 42 45 4c 5f 35  ND....4__LABEL_5
38 00 00 00 01 38 44 45 46 49 4e 49 54 49 4f 4e  8....8DEFINITION
00 00 00 01 4c 53 49 4e 47 4c 45 5f 45 58 50 52  ....LSINGLE_EXPR
45 53 53 49 4f 4e 00 00 00 01 60 52 55 4c 45 00  ESSION....‘RULE.
00 00 01 a0 45 58 50 52 45 53 53 49 4f 4e 00 00  ....EXPRESSION..
00 01 e4 5f 5f 4c 41 42 45 4c 5f 31 30 36 00 00  ...__LABEL_106..
```

Example of a section of labelmap file, hex dumped

9.2.3. *Engine Output Format.* The Naigama bytecode execution engine, naie, produces, on success, output for digital processing, in the form of a binary table, which is structured as follows:

- Each record is four 32 bit integers, in network order.
- The first record contains the end code of the matching process. This is the same code as was given as a parameter to the 'end' instruction that resulted in the execution finishing. This is the first field, by default it is zero. The second field contains the amount of subsequent records.
- Subsequent records have as fields, either:

  - Type, slot, start, length (when of 'capture' type), or:
  - Type, slot, start, char (when of 'replace' type).

Bear in mind the following:

- 'Capture' type is denoted as 1, 'Replace' as 3.
- Offsets and lengths of captures refer to the input buffer.
- Offsets and lengths of replacements refer to the bytecode.

```
00 00 00 00 00 00 03 64 00 00 00 00 00 00 00 00  .......d........
00 00 00 01 00 00 00 00 00 00 00 1c 00 00 00 23  ...............#
00 00 00 01 00 00 00 03 00 00 00 32 00 00 00 59  ...........2...Y
00 00 00 01 00 00 00 04 00 00 00 32 00 00 00 55  ...........2...U
00 00 00 01 00 00 00 14 00 00 00 32 00 00 00 55  ...........2...U
00 00 00 01 00 00 00 02 00 00 00 34 00 00 00 3f  ...........4...?
00 00 00 01 00 00 00 04 00 00 00 34 00 00 00 3f  ...........4...?
00 00 00 01 00 00 00 17 00 00 00 34 00 00 00 3e  ...........4...>
00 00 00 01 00 00 00 06 00 00 00 3e 00 00 00 3f  ...........>...?
00 00 00 01 00 00 00 03 00 00 00 42 00 00 00 53  ...........B...S
00 00 00 01 00 00 00 04 00 00 00 42 00 00 00 53  ...........B...S
00 00 00 01 00 00 00 17 00 00 00 42 00 00 00 53  ...........B...S
```

Example of the head of engine output with a zero end code and 868 matches (all captures), hex dumped

10. API's

10.1. **Languages.**

10.2. **C API Concepts.**

10.2.1. *Main Structures.*

10.2.2. *Error Handling.* Naigama C library function prototypes are (almost) all typed to return the NAIG_ERR_T type.

10.3. **Compiler.**

10.4. **Assembler.**

10.5. **Engine.**

10.6. **All in one API.** The compound API provides an interface to all of the action in one place.

10.6.1. *Includes.*

```
#include <naigama/naigama.h>
```

10.6.2. *Initialization.*

10.6.3. *Using the parser.*

```
NAIG_ERR_T naig_compile
  (naig_t* naig, char* grammar, int traps);
```

```
NAIG_ERR_T naig_run
  (
    naig_t*         naig,
    unsigned char*  input,
    unsigned        input_length,
    naio_result_t*  result
  );
```

10.6.4. *Using the Result Structure.* To include the proper types and functions, use:

```
#include <naigama/memio/naio.h>
```

The definition of the associated types:

```
typedef struct {
  uint32_t             action;
  uint32_t             slot;
  uint32_t             start;
  uint32_t             length; // dubs as char/quad in replace
}
naio_resact_t;

typedef struct
{
  int                  code;
  naio_resact_t*       actions;
  unsigned             length; // allocated
  unsigned             count;  // used
}
naio_result_t;
```

You can step through each action (up to result->count), and for each result->actions[i], check its action (which can be one of NAIG_ACTION_OPENCAPTURE, NAIG_ACTION_DELETE, NAIG_ACTION_REPLACE_CHAR, or NAIG_ACTION_REPLACE_QUAD), its slot number (this will be the iterator that the compiler uses for each encountered capture region), and its start and length. You'll still need the input buffer handy, because this type does not provide you with the capture data itself.

If you're only interested in capturing, then you'll only need the NAIG_ACTION_OPENCAPTURE action type. Also when your grammar does not contain any replacement definitions, you'll never encounter any other action type in the capture list.

10.6.5. *Performing Replacements.* ...

10.6.6. *Getting a Malloced Capture Tree.* To handle capture data independently of the input buffer, and preprocessed to represent a capture tree, you can use the following API:

```
extern
naio_resobj_t* naio_result_object
  (
    const unsigned char*  input,
    unsigned              inputlength,
    naio_result_t*        result
  )
  __attribute__ ((warn_unused_result));
```

```
struct naio_resobj
{
  unsigned              type;
  char*                 string;
  unsigned              stringlen;
  unsigned              origoffset;
  naio_resobj_t*        parent;
  naio_resobj_t**       children;
  unsigned              nchildren;
  unsigned              slotnumber;
  void*                 auxptr; /* this one is for you */
};
```

Below is a simplified version of the naio_result_object_debug function, which shows you how you can recursively step through your capture list and print an indented capture result tree.

```
void naio_result_object_debug
  (naio_resobj_t* object, unsigned indent)
{
  unsigned i;

  if ((int)(object->type) == -1) {
    fprintf(stderr, "TOP  - ");
  } else {
    fprintf(stderr, "%.4u ", object->type);
  }
  for (i=0; i < indent; i++) {
    fprintf(stderr, "--");
  }
  fprintf(stderr, "| %s\n", object->string);
  for (i=0; i < object->nchildren; i++) {
    naio_result_object_debug(object->children[ i ], indent + 1);
  }
}
```

11. Colofon

List of Tables

References

[1] A Text Pattern-Matching Tool based on Parsing Expression Grammars https://www.inf.puc-rio.br/~roberto/docs/peg.pdf

[2] Regular Expressions https://en.wikipedia.org/wiki/Regular_expression

[3] Backus Naur Form https://en.wikipedia.org/wiki/Backus-Naur_form

[4] Yacc Yet Another Compiler Compiler https://en.wikipedia.org/wiki/Yacc

[5] JavaScript, or ECMAScript https://www.ecma-international.org/publications/standards/Ecma-262.htm

[6] JSON, JavaScript Object Notation https://www.json.org/

[7] Perl, the Perl Programming Language https://www.perl.org/

Appendices

A. Appendix: Instructions

In this section, each Naigama instruction is treated in detail in its own subsection. In these subsections, an instruction is disected as follows:

- 'Summary'. There is a short summary of the function of the instruction.
- 'Grammar and Compiling'. This subsection details how and why this instruction may be emitted by the compiler, given the grammar. This may include compilation patterns.
- 'Assembly Syntax'. This subsection provides the exact syntax of the instruction in Naigama assembly.
- 'Bytecode Encoding'. This subsection provides a byte-for-byte specification of how this instruction will be laid out in memory and on disk.
- 'Execution State Change'. This subsection contains the formal description of the execution of the instruction.
- 'PseudoCode'. This subsection contains a pseudo code implementation of the instruction.

A.1. **Instruction: any.**

A.1.1. *Summary.* This instruction advances the input pointer by one, if possible. If successful, the input pointer is advanced and the next instruction is executed. If not successful (because end-of-input has not been reached) then the FAIL state is invoked.

A.1.2. *Grammar and Compiling.* The 'any' instruction is emitted whenever the 'dot' (.) token is used in rule production. For example:

```
CONSUME_ONE_BYTE <- .
CONSUME_ALL_BYTES <- .*
```

A.1.3. *Assembly Syntax.*

```
ANYINSTR <- { 'any' }
```

A.1.4. *Bytecode Encoding.* This instruction has a size of 4 bytes and is structured in bytecode as follows:

00	00	00	03	e4

Where:

Bytes 0-3 denote the instruction opcode.

A.1.5. *Execution State Change.* .

Original state: *(p, i, e, c)*

Operation: **any** ; $i < |S|$

Failure state: *(***Fail***, i, e, c)*

Success state: *(p + 1, i + 1, e, c)*

A.1.6. *Pseudo Code.*

```
if (input_offset < input_length) then
        input_offset = input_offset + 1
        bytecode_offset = bytecode_offset + instruction_size
else
        fail
endif
```

A.2. **Instruction: backcommit.**

A.2.1. *Summary.* Backcommit restores the engine state wrt the input offset and action list length, but then jumps to the bytecode offset given as a parameter to the instruction. The backcommit instruction is emitted to implement the '&' matcher operator, which tests the matcher without increasing the input pointer.

A.2.2. *Grammar and Compiling.* The backcommit instruction is emitted to implement the '&' matcher operator. For example:

```
RULE <- & 'foo'
```

The following assembly pattern is used to produce this behavior in the engine (assuming [[matcher]] matches 'foo' as above):

```
catch failure
[[matcher]]
backcommit safety
failure:
fail
safety:
```

A.2.3. *Assembly Syntax.*

```
BACKCOMMITINSTR <- { 'backcommit' } S LABEL
S <- %s+
LABEL <- { [a-zA-Z0-9_]^1-256 }
```

A.2.4. *Bytecode Encoding.* This instruction has a size of 8 bytes and is structured in bytecode as follows:

00	00	04	03	c0
04	00	00	00	00

Where:

Bytes 0-3 denote the instruction opcode.

Bytes 4-7 denote the bytecode offset to jump to, after this instruction (big endian 32 bit unsigned).

A.2.5. *Execution State Change.* .

Original state: (p, i, e, c)

Operation: **backcommit**

Resultant state: $(p + 1, i + 1, e, c)$

A.2.6. *Pseudo Code.*

```
let s = stack_pop()
let p = instruction_param(1)
bytecode_offset = p
input_offset = s.input_offset
while (actions.length > s.actions_length)
        actions_pop()
endwhile
```

A.3. **Instruction: call.**

A.3.1. *Summary.* This instruction calls a rule or function.

A.3.2. *Grammar and Compiling.* This instruction is emitted whenever a reference to a rule or function is made in the grammar.

```
RULE1 <- RULE2 -- creates a call instruction to 'RULE2'
RULE2 <- ...
```

A.3.3. *Assembly Syntax.*

```
CALLINSTR <- { 'call' } S LABEL
S <- %s+
LABEL <- { [a-zA-Z0-9_]^1-256 }
```

A.3.4. *Bytecode Encoding.* This instruction has a size of 8 bytes and is structured in bytecode as follows:

00	00	04	03	82
04	00	00	00	00

Where:

Bytes 0-3 denote the instruction opcde.

Bytes 4-7 denote the bytecode offset to jump to (the address of the called rule or function) (big endian 32 bit unsigned).

A.3.5. *Execution State Change.* .

A.4. **Instruction: catch.**

A.4.1. *Summary.*

A.4.2. *Grammar and Compiling.*

A.4.3. *Assembly Syntax.*

```
CATCHINSTR <- { ’catch’ } S LABEL
S <- %s+
LABEL <- { [a-zA-Z0-9_]^1-256 }
```

A.4.4. *Bytecode Encoding.* This instruction has a size of 8 bytes and is structured in bytecode as follows:

00	00	04	03	93
04	00	00	00	00

A.4.5. *Execution State Change.* .

A.5. **Instruction: char.**

A.5.1. *Summary.*

A.5.2. *Grammar and Compiling.*

A.5.3. *Assembly Syntax.*

```
CHARINSTR <- { 'char' } S HEXBYTE
S <- %s+
HEXBYTE <- { [0-9a-fA-F]^2 }
```

A.5.4. *Bytecode Encoding.* This instruction has a size of 8 bytes and is structured in bytecode as follows:

00	00	04	03	d7
04	00	00	00	00

A.5.5. *Execution State Change.* .

A.6. **Instruction: closecapture.**

A.6.1. *Summary.*

A.6.2. *Grammar and Compiling.*

A.6.3. *Assembly Syntax.*

```
CLOSECAPTUREINSTR <- { 'closecapture' } S SLOT (S TYPE)?
S <- %s+
SLOT <- NUMBER
```

A.6.4. *Bytecode Encoding.* This instruction has a size of 8 bytes and is structured in bytecode as follows:

00	00	04	03	00
04	00	00	00	00

A.6.5. *Execution State Change.* .

A.7. **Instruction: commit.**

A.7.1. *Summary.*

A.7.2. *Grammar and Compiling.*

A.7.3. *Assembly Syntax.*

```
COMMITINSTR <- { ’commit’ } S LABEL
S <- %s+
LABEL <- { [a-zA-Z0-9_]^1-256 }
```

A.7.4. *Bytecode Encoding.* This instruction has a size of 8 bytes and is structured in bytecode as follows:

00	00	04	03	36
04	00	00	00	00

A.7.5. *Execution State Change.* .

A.8. **Instruction: condjump.**

A.8.1. *Summary.*

A.8.2. *Grammar and Compiling.*

A.8.3. *Assembly Syntax.*

```
CONDJUMPINSTR <- { ’condjump’ } S REGISTER S LABEL
S <- %s+
REGISTER <- NUMBER
LABEL <- { [a-zA-Z0-9_]^1-256 }
```

A.8.4. *Bytecode Encoding.* This instruction has a size of 12 bytes and is structured in bytecode as follows:

00	00	08	03	21
04	00	00	00	00
08	00	00	00	00

A.8.5. *Execution State Change.* .

A.9. **Instruction: counter.**

A.9.1. *Summary.* This instruction supports counter loops for matchers, to avoid having to write out every matcher individually, when a matcher quantifier is used in grammar.

A.9.2. *Grammar and Compiling.* This instruction is emitted in conjunction with a 'condjump' instruction, when a matcher quantifier is used. For example:

```
RULE <- 'a'^5
```

Will wrap the matcher bytecode between a counter and a condjump instruction, initial value of the counter of which will be five. Typically, as follows (pattern):

```
counter 0 5
jumpbacklabel:
[[matcher]]
condjump 0 jumpbacklabel
```

The 'condjump' instruction will either jump back (when the counter value in the register has not yet reached zero), or ignore the jump and move on (when it has). Thus implementing a loop.

A.9.3. *Assembly Syntax.*

```
COUNTERINSTR <- { 'counter' } S REGISTER S NUMBER
S <- %s+
REGISTER <- NUMBER
NUMBER <- { [0-9]+ }
```

A.9.4. *Bytecode Encoding.* This instruction has a size of 12 bytes and is structured in bytecode as follows:

00	00	08	03	56
04	00	00	00	00
08	00	00	00	00

Where

Bytes 0-3 denote the instruction opcde.

Bytes 4-7 denote the register number used (big endian 32 bit unsigned).

Bytes 8-11 denote the initial counter value (big endian 32 bit unsigned).

A.9.5. *Execution State Change.* .

A.10. **Instruction: end.**

A.10.1. *Summary.*

A.10.2. *Grammar and Compiling.*

A.10.3. *Assembly Syntax.*

```
ENDINSTR <- { 'end' } (S CODE )?
CODE <- NUMBER
```

A.10.4. *Bytecode Encoding.* This instruction has a size of 8 bytes and is structured in bytecode as follows:

00	00	04	00	d8
04	00	00	00	00

A.10.5. *Execution State Change.* .

A.11. **Instruction: endreplace.**

A.11.1. *Summary.*

A.11.2. *Grammar and Compiling.*

A.11.3. *Assembly Syntax.*

```
ENDREPLACEINSTR <- { 'endreplace' }
```

A.11.4. *Bytecode Encoding.* This instruction has a size of 4 bytes and is structured in bytecode as follows:

$$00\ \boxed{00}\ \boxed{00}\ \boxed{03}\ \boxed{99}$$

A.11.5. *Execution State Change.* .

A.12. **Instruction: fail.**

A.12.1. *Summary.*

A.12.2. *Grammar and Compiling.*

A.12.3. *Assembly Syntax.*

```
FAILINSTR <- { ’fail’ }
```

A.12.4. *Bytecode Encoding.* This instruction has a size of 4 bytes and is structured in bytecode as follows:

00	00	00	03	4b

A.12.5. *Execution State Change.* .

A.13. **Instruction: failtwice.**

A.13.1. *Summary.*

A.13.2. *Grammar and Compiling.*

A.13.3. *Assembly Syntax.*

```
FAILTWICEINSTR <- { 'failtwice' }
```

A.13.4. *Bytecode Encoding.* This instruction has a size of 4 bytes and is structured in bytecode as follows:

00	00	00	03	90

A.13.5. *Execution State Change.* .

A.14. **Instruction: jump.**

A.14.1. *Summary.*

A.14.2. *Grammar and Compiling.*

A.14.3. *Assembly Syntax.*

```
JUMPINSTR <- { 'jump' } S LABEL
S <- %s+
LABEL <- { [a-zA-Z0-9_]^1-256 }
```

A.14.4. *Bytecode Encoding.* This instruction has a size of 8 bytes and is structured in bytecode as follows:

00	00	04	03	33
04	00	00	00	00

A.14.5. *Execution State Change.* .

A.15. **Instruction: maskedchar.**

A.15.1. *Summary.*

A.15.2. *Grammar and Compiling.*

A.15.3. *Assembly Syntax.*

```
MASKEDCHARINSTR <- { 'maskedchar' } S HEXBYTE S HEXBYTE
S <- %s+
HEXBYTE <- { [0-9a-fA-F]^2 }
```

A.15.4. *Bytecode Encoding.* This instruction has a size of 12 bytes and is structured in bytecode as follows:

00	00	08	03	65
04	00	00	00	00
08	00	00	00	00

A.15.5. *Execution State Change.* .

A.16. **Instruction: noop.**

A.16.1. *Summary.* This instruction does nothing.

A.16.2. *Grammar and Compiling.* This instruction may be emitted to align bytecode.

A.16.3. *Assembly Syntax.*

```
NOOPINSTR <- { 'noop' }
```

A.16.4. *Bytecode Encoding.* This instruction has a size of 4 bytes and is structured in bytecode as follows:

00 $\boxed{\text{00}}$ $\boxed{\text{00}}$ $\boxed{\text{00}}$ $\boxed{\text{00}}$

A.16.5. *Execution State Change.* .

Original state: *(p, i, e, c)*

Resultant state: *(p, i, e, c)*

A.17. **Instruction: opencapture.**

A.17.1. *Summary.*

A.17.2. *Grammar and Compiling.*

A.17.3. *Assembly Syntax.*

```
OPENCAPTUREINSTR <- { 'opencapture' } S SLOT
S <- %s+
SLOT <- NUMBER
```

A.17.4. *Bytecode Encoding.* This instruction has a size of 8 bytes and is structured in bytecode as follows:

00	00	04	03	9c
04	00	00	00	00

A.17.5. *Execution State Change.* .

A.18. **Instruction: partialcommit.**

A.18.1. *Summary.*

A.18.2. *Grammar and Compiling.*

A.18.3. *Assembly Syntax.*

```
PARTIALCOMMITINSTR <- { 'partialcommit' } S LABEL
S <- %s+
LABEL <- { [a-zA-Z0-9_]^1-256 }
```

A.18.4. *Bytecode Encoding.*

This instruction has a size of 8 bytes and is structured in bytecode as follows:

00	00	04	03	b4
04	00	00	00	00

A.18.5. *Execution State Change.* .

A.19. **Instruction: quad.**

A.19.1. *Summary.*

A.19.2. *Grammar and Compiling.*

A.19.3. *Assembly Syntax.*

```
QUADINSTR <- { 'quad' } S QUAD
S <- %s+
QUAD <- { [0-9a-fA-F]^8 }
```

A.19.4. *Bytecode Encoding.* This instruction has a size of 8 bytes and is structured in bytecode as follows:

00	00	04	03	7e
04	00	00	00	00

A.19.5. *Execution State Change.* .

A.20. **Instruction: range.**

A.20.1. *Summary.*

A.20.2. *Grammar and Compiling.*

A.20.3. *Assembly Syntax.*

```
RANGEINSTR <- { ’range’ } S NUMBER S NUMBER
S <- %s+
NUMBER <- { [0-9]+ }
```

A.20.4. *Bytecode Encoding.* This instruction has a size of 12 bytes and is structured in bytecode as follows:

00	00	08	03	bd
04	00	00	00	00
08	00	00	00	00

A.20.5. *Execution State Change.* .

A.21. **Instruction: replace.**

A.21.1. *Summary.*

A.21.2. *Grammar and Compiling.*

A.21.3. *Assembly Syntax.*

```
REPLACEINSTR <- { ’replace’ } S LABEL S LABEL
S <- %s+
LABEL <- { [a-zA-Z0-9_]^1-256 }
```

A.21.4. *Bytecode Encoding.* This instruction has a size of 12 bytes and is structured in bytecode as follows:

00	00	08	03	48
04	00	00	00	00
08	00	00	00	00

A.21.5. *Execution State Change.* .

A.22. **Instruction: ret.**

A.22.1. *Summary.*

A.22.2. *Grammar and Compiling.*

A.22.3. *Assembly Syntax.*

```
RETINSTR <- { ’ret’ }
```

A.22.4. *Bytecode Encoding.* This instruction has a size of 4 bytes and is structured in bytecode as follows:

00	00	00	03	a0

A.22.5. *Execution State Change.* .

A.23. **Instruction: set.**

A.23.1. *Summary.*

A.23.2. *Grammar and Compiling.*

A.23.3. *Assembly Syntax.*

```
SETINSTR <- { 'set' } S SET
S <- %s+
SET <- { [0-9a-fA-F]^64 }
```

A.23.4. *Bytecode Encoding.* This instruction has a size of 36 bytes and is structured in bytecode as follows:

00	00	20	03	ca
04	00	00	00	00
08	00	00	00	00
12	00	00	00	00
16	00	00	00	00
20	00	00	00	00
24	00	00	00	00
28	00	00	00	00
32	00	00	00	00

A.23.5. *Execution State Change.* .

A.24. **Instruction: skip.**

A.24.1. *Summary.*

A.24.2. *Grammar and Compiling.*

A.24.3. *Assembly Syntax.*

```
SKIPINSTR <- { ’skip’ } S NUMBER
S <- %s+
NUMBER <- { [0-9]+ }
```

A.24.4. *Bytecode Encoding.* This instruction has a size of 8 bytes and is structured in bytecode as follows:

00	00	04	03	30
04	00	00	00	00

A.24.5. *Execution State Change.* .

A.25. **Instruction: span.**

A.25.1. *Summary.*

A.25.2. *Grammar and Compiling.*

A.25.3. *Assembly Syntax.*

```
SPANINSTR <- { 'span' } S SET
S <- %s+
SET <- { [0-9a-fA-F]^64 }
```

A.25.4. *Bytecode Encoding.* This instruction has a size of 36 bytes and is structured in bytecode as follows:

00	00	20	03	e1
04	00	00	00	00
08	00	00	00	00
12	00	00	00	00
16	00	00	00	00
20	00	00	00	00
24	00	00	00	00
28	00	00	00	00
32	00	00	00	00

A.25.5. *Execution State Change.* .

A.26. **Instruction: testany.**

A.26.1. *Summary.*

A.26.2. *Grammar and Compiling.*

A.26.3. *Assembly Syntax.*

```
TESTANYINSTR <- { ’testany’ } S LABEL
S <- %s+
LABEL <- { [a-zA-Z0-9_]^1-256 }
```

A.26.4. *Bytecode Encoding.* This instruction has a size of 8 bytes and is structured in bytecode as follows:

00	00	04	03	06
04	00	00	00	00

A.26.5. *Execution State Change.* .

A.27. **Instruction: testchar.**

A.27.1. *Summary.*

A.27.2. *Grammar and Compiling.*

A.27.3. *Assembly Syntax.*

```
TESTCHARINSTR <- { 'testchar' } S HEXBYTE S LABEL ( S AMPERSAND HEXBYTE )?
S <- %s+
HEXBYTE <- { [0-9a-fA-F]^2 }
LABEL <- { [a-zA-Z0-9_]^1-256 }
AMPERSAND <- '&'
```

A.27.4. *Bytecode Encoding.* This instruction has a size of 12 bytes and is structured in bytecode as follows:

00	00	08	03	9a
04	00	00	00	00
08	00	00	00	00

A.27.5. *Execution State Change.* .

A.28. **Instruction: testquad.**

A.28.1. *Summary.*

A.28.2. *Grammar and Compiling.*

A.28.3. *Assembly Syntax.*

```
TESTQUADINSTR <- { 'testquad' } S QUAD S LABEL
S <- %s+
QUAD <- { [0-9a-fA-F]^8 }
LABEL <- { [a-zA-Z0-9_]^1-256 }
```

A.28.4. *Bytecode Encoding.* This instruction has a size of 12 bytes and is structured in bytecode as follows:

00	00	08	03	db
04	00	00	00	00
08	00	00	00	00

A.28.5. *Execution State Change.* .

A.29. **Instruction: testset.**

A.29.1. *Summary.*

A.29.2. *Grammar and Compiling.*

A.29.3. *Assembly Syntax.*

```
TESTSETINSTR <- { 'testset' } S SET S LABEL
S <- %s+
SET <- { [0-9a-fA-F]^64 }
LABEL <- { [a-zA-Z0-9_]^1-256 }
```

A.29.4. *Bytecode Encoding.* This instruction has a size of 40 bytes and is structured in bytecode as follows:

00	00	24	03	63
04	00	00	00	00
08	00	00	00	00
12	00	00	00	00
16	00	00	00	00
20	00	00	00	00
24	00	00	00	00
28	00	00	00	00
32	00	00	00	00
36	00	00	00	00

A.29.5. *Execution State Change.* .

A.30. **Instruction: trap.**

A.30.1. *Summary.*

A.30.2. *Grammar and Compiling.*

A.30.3. *Assembly Syntax.*

```
TRAPINSTR <- { ’trap’ }
```

A.30.4. *Bytecode Encoding.* This instruction has a size of 4 bytes and is structured in bytecode as follows:

00	ff	00	ff	ff

A.30.5. *Execution State Change.* .

A.31. **Instruction: var.**

A.31.1. *Summary.*

A.31.2. *Grammar and Compiling.*

A.31.3. *Assembly Syntax.*

```
VARINSTR <- { 'var' } S SLOT
S <- %s+
SLOT <- NUMBER
```

A.31.4. *Bytecode Encoding.*

This instruction has a size of 8 bytes and is structured in bytecode as follows:

00	00	04	03	ee
04	00	00	00	00

A.31.5. *Execution State Change.* .