The Baabnq Programming Language

1. Introduction

1.1 Description

The Baabnq Programming Language is a low-level untyped statically-compiled Language that is intended to compile to S1monsAssembly. Baabnq is meant for very small and feature-poor systems and processors, where any other languages would be too high-level.

S1monsAssembly is a simplistic assembler, that doesn't have any of the weird and arguably useless features of modern assemblers. In addition, S1monsAssembly can be retargeted easily, which comes in handy when making a custom system, and where a custom compiler would be overkill.

This whole system is designed for homebrew architectures and small private projects, to provide an easy way of quickly programming and debugging. However, doing larger projects in Baabnq is not a good idea, because code can become unmanageable quite quickly if one is not careful.

1.2 Basic information

In Baabnq every program consists of a list of statements. Every statement ends in a semicolon. All variables are prefixed with an underscore. There are no namespaces or scopes, so all variables are global. It's convention that all Baabnq files end with the extension ".baabnq".

List of valid expression operators: **+**, **-**, **|**, **&**, **^**, **<<**, **>>** (plus, minus, or, and, xor, shift l, shift r)
List of valid conditional operators: **==**, **>**, **<**, **!=** (equal, greater, smaller, not equal)

1.3 Table of contents

1. Introduction 1
1.1 Description 1
1.2 Basic information 1
1.3 Table of contents 2
2. Memory Management 3
2.1 Memory Introduction 3
2.2 put 3
2.3 new 3
2.4 free 4
2.5 static 4
3. Stack 4
3.1 Stack Introduction 4
3.2 push 5
3.3 pull 5
4. Flow control 5
4.1 Flow control introduction 5
4.2 lab 5
4.3 jump 5
4.4 sub 5
4.5 return 6
5. Input and Output 6
5.1 input 6
5.2 print 6
5.3 putchr 6
6. Extra Commands 7
6.1 Inline Assembler 7
6.2 use 7
7. Quick Reference 8
7.1 Command list 8
7.2 Text formatting 8
7.3 Syntax highlighting 9
8. Retargeting 9
8.1 Explanation 9
8.2 Instruction listing 10

2. Memory Management

2.1 Memory Introduction

Baabnq and subsequently S1monsAssembly has 16-bit memory, while the upper half is heap memory and the lower half is just normal memory.

Baabnq puts variables and temporaries into normal memory, but the user can allocate own memory at compile-time. The heap is controlled by the user and managed at run-time by the processor.

2.2 put

The **put** command is used for general variable and pointer manipulation.
Here's an example:

```
put _a = 2 + 3;
```

In this case the variable **_a** is set to the evaluation of the expression **2 + 3**, which is **5**. If a variable is not allocated, the **put** instruction will allocate it.
Besides the **=** operator to set a variable, there are two more operator **->** and **<-**,
which are for pointers. When handling a pointer, the right side is always the address and the left side is the data at that address.
-> takes an expression, variable or constant and writes it to where the pointer is pointing.
<- takes a variable and overrides it with the value, the pointer is pointing to.
Here's an example:

```
"take _data and write to the address stored in _ptr
put _data -> _ptr;

"read the value stored at _ptr and write it to _data
put _data <- _ptr;
```

Notice that both **_data** and **_ptr** are values, in this case **_ptr** is just interpreted as a pointer.
To further illustrate, here's a C- equivalent:

```
//assuming int data and int* ptr
//put _data <- _ptr;
data = *ptr;
//put _data -> _ptr;
*ptr = data;
```

2.3 new

The **new** command can allocate memory dynamically on the heap at run-time. There are two sub-classes of dynamic memory: chunk and string. Strings are encoded by the compiler and have a normal null-terminator, like in C. Chunks are not fixed and the size of a chunk is stored in its first word, yet this is not really important, because chunks are managed by the compiler.

The **new** command takes two arguments:

1. Either a string or a size, depending on what kind of memory should by allocated.
2. The destination pointer that points to the base of the newly allocated memory.

Here's an example:

```
"allocate a string "hello world"
new 'Hello World' _ptr;

"allocate a chunk based upon a dynamic size
put _size = 10;
new _size _ptr;
```

2.4 free

The **free** command can deallocate heap memory, given a pointer
Here's an example:

```
"allocate string "Hello World" and free it again
new 'Hello World' _ptr;
free _ptr;
```

The command knows the size of the allocated chunk or string because it reads the first word, which holds the size.

2.5 static

The **static** command allocates in static memory at compile-time, but otherwise works the same as **new**
Here's an example:

```
static 'Hello World' _ptr;

"here the size value need to be constant
static 10 _ptr;
```

3. Stack

3.1 Stack Introduction

The stack in Baabnq and subsequently S1monsAssembly works like a normal stack, though user data and return addresses from subroutine calls are mixed. Furthermore, the stack is not mapped to the normal memory.

3.2 push

The **push** command takes an expression, variable or constant, evaluates it and pushes it onto the stack.

Here's an example:

```
"push 10 then 5 to the stack
put _a = 3;
push 10;
push 2 + _a;
```

3.3 pull

The **pull** command pulls a value from the stack and overrides the given variable with the value.
Here's an example:

```
"set test value
push 10;
"and pull it again
pull _a;
"now _a is equal to 10
```

4. Flow control

4.1 Flow control introduction

Flow control in Baabnq work quite similar to assembly languages, it uses labels and jumps to those label, to control the flow. In addition to that, there are also subroutines jumps to labels. Both the normal jump and subroutine jump have conditional counterparts.

4.2 lab

The **lab** command defines a label with a given name.
Here's an example:

```
lab ThisIsALabel;
lab ThisIsAnotherLabel;
```

4.3 jump

The **jump** command jumps to a given label. Optionally a boolean expression can be set with a **~**; then the jump will only be executed if the expression is true.
Here's an example:

```
lab Loop1;
"here's an infinite loop that only runs while 1 is 1
jump Loop1 ~ 1 == 1;

lab Loop2;
"here's another infinite loop that runs always
jump Loop2;
```

4.4 sub

The **sub** command calls a subroutine. It does this by taking the program counter, saving it on the stack as the return address and then normally jumping to the subroutine.

4.5 return
The **return** command returns from a subroutine, it does this by pulling the return address, that has been pushed to the stack by the **sub** command and jumping there. This will effectively continue the program where it left of.
Here's an example:
```
"this program will take _FunnyValue, then add 2, then subtract 1, then repeat
put _FunnyValue = 0;
lab MainLoop;
sub Add2;
sub Sub1;
jump MainLoop;

"helper routine to add 2
lab Add2;
put _FunnyValue = _FunnyValue + 2;
return;

"helper routine to subtract 1
lab Sub1;
put _FunnyValue = _FunnyValue - 1;
return;
```
When dealing with subroutines, one has to be careful not to remove or obscure the return address from the stack.

5. Input and Output
5.1 input
The **input** command takes input in form of a number from the user and writes it to a given variable. Here's an example:
```
input _a;
input _b;
```

5.2 print
The **print** command takes a variable, expression or constant and outputs it as decimal.
Here's an example:
```
put _a = 3;

print 2;
print _a;
print _a + 1;
"the output of this program is: '2\n3\n4\n'
```

5.3 putchr
The **putchr** command takes a variable, expression or constant and outputs it as an ascii character.
Here's an example:
```
putchr 65;
putchr 10;
```

6. Extra Commands

6.1 Inline Assembler

The **asm** command takes a string, applies inline processing and put the result into the output assembler file at the same relative position as the **asm** command itself. So, in informal terms, **asm** is for inline assembly. When using inline assembler, the following things have to be considered (this only makes sense with prior knowledge of S1monsAssembly):

1. Instruction and attribute are separated by a space and all pairs end in a semicolon.
2. Each instruction can only have one attribute, not more.
3. Variables can be referenced by inclosing their name in brackets.
4. A temporary, which only exists in the scope of the command, can be referenced by prepending the name of the temporary with a dash symbol "-".

Here's an example:

```
put _test = 10;
asm '
lDA (_test);
sAD -myTemp;
out -myTemp;
';
```

6.2 use

The **use** command takes a path to an external Baabnq file and imports that files contents. This happens fully at compile-time, there is no dynamic linking. The files that you can import are raw Baabnq, they are just 'inserted' into the program. However, assembly files could be linked using a special assembler. But this is not needed, because Baabnq libraries are generally not very big and can just be recompiled with the main file. If a relative path is given, the compiler will search for it in the working directory, from where it has been called.

Here's an example:

```
"this is the file to import
"its name is 'lib.baabnq'

static 'This is a lib' _InfoPtr;
```

```
"this is the file importing
"its name doesn't matter

use 'lib.baabnq';
"now the _InfoPtr is available
```

7. Quick Reference

7.1 Command list

Command	Description
put	set variable
new	allocate heap memory
free	free heap memory
static	allocate static memory
push	push to stack
pull	pull from stack
lab	define label
jump	jump to label
sub	call subroutine
return	return from subroutine
input	get input from stdin
print	print value to stdout
putchr	output value as ascii to stdout
asm	wrapper for inline assembler
use	import from external file

7.2 Text formatting

There are some text formatting conventions in Baabnq, that are not enforced by syntax, but are generally useful. While Baabnq has no code blocks, they can be implied and usually are indented. Here's an example:

```
lab Loop;
    "This is the content of a code block
    print 1;
jump Loop;
```

There is also a way to manage labels, in the case that a program gets too convoluted.
In bigger program, the architecture is mainly broken down with subroutines. So if there is a conflict between labels, they can be prefixed by the name of the subroutine they belong to. The two names can be separated by a double colon, to avoid further confusion.

Here's an example:

```
lab MySub;
    "sub label
    lab MySub::Lab1;
    lab MySub::Lab2;
        "even a subsub label
        lab MySub::Lab2::Lab3;
```

7.3 Syntax highlighting
Here's the default syntax highlighting:

Color	Category
orange	constant
cyan	variable
red	operator
light green	general command
yellow	flow control
purple	stack
blue	memory
white	label

8. Retargeting

8.1 Explanation
(Disclaimer: This is advanced knowledge and is not required for basic use, if a cross-assembler is available for your target system)

Baabnq is designed to be retargetable, meaning that you can use the compiler for multiple systems. Baabnq achieves this by compiling to S1monsAssembly, which is the "Frankensteins Monster" equivalent of assemblers. You can chop and sew parts of S1monsAssembly and it will still run perfectly fine if you're careful. S1monsAssembly is designed to be very simplistic, only having 2 registers and a general 16-bit architecture. But even that bit width with can be tampered with. On the other hand, S1monsAssembly also supports advanced features such as dynamic memory management, which are obviously optional, both in the usage of S1monsAssembly itself and also in Baabnq. In addition to this S1monsAssembly doesn't have any quality of life improvements, such as macros or defining bytes, in the source. This also makes retargeting to a drastically different architecture simpler.

Here is a more detailed description of the S1monsAssembly System:
S1monsAssembly has 2 register, the **acc** and the **reg**. Arithmetic is done by using the **reg** as a base to modify the **acc**. For Example, the **add** instruction, takes the **reg** and adds it to the **acc**.
(That's why it's called **acc**, Accumulator)
The memory is 16-bit, both address space and word space, while the heap is mapped to the upper half of the memory. The stack is not memory mapped by default, but it can be mapped to the lower half. Important to keep in mind is, that the stack is required by Baabnq for expression evaluation.

(A tip for trying to figure out what instructions to implement, is for the custom assembler to log all instructions that it doesn't understand and add based on that. I know that's kind of unprofessional, but it works)

8.2 Instruction listing

Important to note is, that S1monsAssembly uses a **RIS (Reduced Instruction Set)**, each instruction can only take one argument.

<Value> refers to a direct value.
<Address> refers to a memory address.
<Label> refers to a label.

Instruction	Argument	Description
set	<Value>	Puts <Value> into the **reg** register
add		Adds content of **reg** to **acc**
sub		Subtracts content of **reg** from **acc**
shg		Shifts content of **acc** to the left (multiply by 2)
shs		Shifts content of **acc** to the right (divide by 2)
lor		Bitwise Ors content of **reg** to **acc**
and		Bitwise Ands content of **reg** to **acc**
xor		Bitwise Xors content of **reg** to **acc**
not		Bitwise Inverts content of **acc**
lDA	<Address>	Loads value from <Address> into **acc** register
lDR	<Address>	Loads value from <Address> into **reg** register
sAD	<Address>	Saves value from **acc** register into <Address>
sRD	<Address>	Saves value from **reg** register into <Address>
lPA	<Address>	Loads value pointed to by <Address> into **acc** register
lPR	<Address>	Loads value pointed to by <Address> into **reg** register
sAP	<Address>	Saves value from **acc** register into memory pointed to by <Address>
sRP	<Address>	Saves value from **reg** register into memory pointed to by <Address>
out	<Address>	Outputs value from <Address>
lnp	<Address>	Inputs value to <Address>
lab	<Label>	Defines <Label>
got	<Label>	Jumps to <Label> unconditionally
jm0	<Label>	Jumps to <Label> if **acc** is 0
jmA	<Label>	Jumps to <Label> if **acc** is equal to **reg**
jmG	<Label>	Jumps to <Label> if **acc** is greater to **reg**
jmL	<Label>	Jumps to <Label> if **acc** is less to **reg**
jmS	<Label>	Calls subroutine at <Label>
ret		Returns from subroutine
pha		Pushes **acc** to the stack
pla		Pulls from stack to **acc**
brk		Stops programs
clr		Clears **acc** and **reg**
putchr		Prints **acc** as character to stdout (not a standard instruction)
ahm		Allocate heap memory, size given by **reg**, base pointer put to **acc**
fhm		Free heap memory, size given by **reg**, base pointer given by **acc**