Jonas Franz

Snips Documentation & Manual

Contents

1. About SNIPS

The Language

Snips is a C/Java oriented programming language. It is meant to be a simple, easy to use language, that us used best for small programs. The Language does provide a small built in Library, of which you can learn more in a later chapter.

The compiler translates the language into ARM Assembly. The output file contains all functions, imports, direct imports as well as transitive imports. The output file can be seen as a completely independent source file. All that's needed to run the code is already included. This means that no kind of linking has to be done, the assembly only has to be converted into binary.

The language features some basic features, like functions, control structures and loops. More advanced features like Templating, Predicates and Structs are supported as well. The language implements a wide set of operators, from arithmetic to boolean to bitwise and comparison operators.

An import system is also built in. Imports stated by the include directive are automatically processed and imported. This allows for a more distributed programming style and higher reusability.

Some words in advance

This project was started and still is for educational purposes. The programming language Snips, the Compiler and all included modules are not following any standards and are built to function well only for this project. Results produced by the compiler and included modules may contain errors and are not thought for any production environment. The project and all its included modules are still under development and are subject to change, both in functionality, as well in language syntax and behaviour.

Comparison to the Raspbian Buster

To bring performance and compactness of the output of the compiler into perspective, in this section, small programs are compiled with the Snips Compiler and the Raspbian Buster. The resulting program is executed on the SWARM32Pc LLVM. The measured metrics are amount of generated instructions in lines in the output file, and executed CPU-Cycles for running the programs with the same inputs.

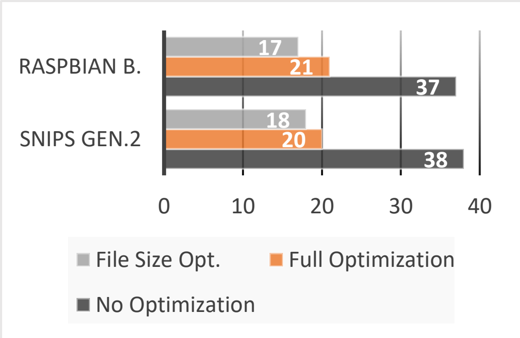
Used compiler Flags:

Full Optimization: Raspbian Buster: $-O4$, Snips Gen.2: $-sid$
File Size Optimization: Raspbian Buster: $-Os$, Snips Gen.2: $-sid, -com - ofs$

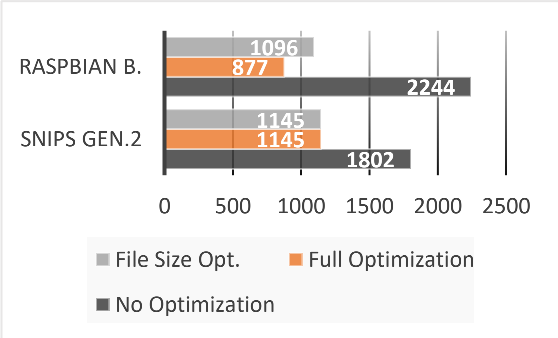
Recursive Fibonacci Sequence:

```
int fib(int n) {
    if (n < 3) return 1;
    else return fib(n - 1) + fib(n - 2);
}
int main(int a)
    return fib(a);
```

Generated Instructions:



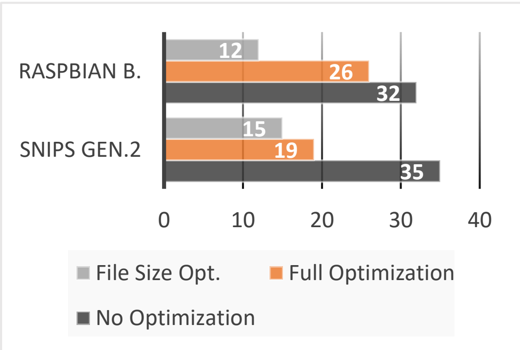
CPU-Cycles, given input for $a = 10$



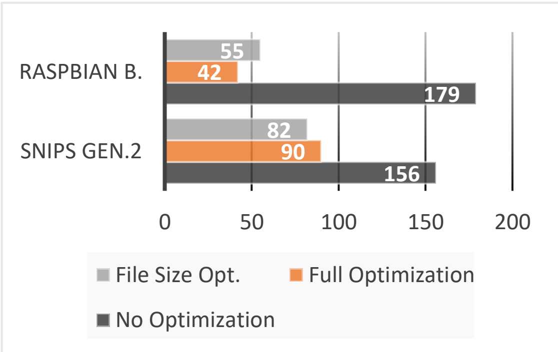
Recursive log_2:

```
int log2(int a) {
    if (a != 0)
        return 1 + log2(a >> 1);
    else return 0;
}
int main(int a)
    return log2(a);
```

Generated Instructions:



CPU-Cycles, given input for $a = 128$:



Conclusion

As seen above, the Raspbian Buster has an advantage in File Size and Performance. The Raspbian Buster can make use of AST-Transformations and deeper Optimization Techniques, while the Optimizer of Snips Gen.2 is only able to make local improvements to generated Assembly Code. The complexity introduced by AST-transformations is out of the scope of this project, and plans to implement said optimizations have been scrapped.

2. Language Syntax

Type System

Primitives

Snips has a range of built-in primitive types. All of the types are 1 32-bit word large.

Void. The void type acts as a universal primitive type. Using this type requires caution, because of its unique property to match every other type. This is not only true for primitives, but for every possible type. This can be used to create "don't care" types. But, to improve type security, this should be avoided. Also, since the void type is 1 word large, assigning an expression to this type that is larger than 1 word can cause major problems. Consider **Void Arrays** as a workaround. A Void is declared with the type specifier $void$. For example:

```
void v = 20;
```

Integer. The integer represents the most common numeric type, with a range from $-2^{31}-1$ to 2^{31}. An Integer is declared with the type specifier int. For example:

```
int x = 10;
```

Boolean. The boolean type is mostly used for logic. Its value range is the exact same as the one of the Integer, but when interpreting the value, only the value 0 is interpreted as $false$, all other possible values are interpreted as $true$. A boolean is declared with the type specifier $bool$. For example:

```
bool b = true;
```

Character. The character type can be used to store UTF-8 encoded Characters. The value Range is, because of the 32-bit size, like the one of the Integer, but the usable Range is only from 0 to 255. The 0 value takes a special role. In a String, a char with the value 0 indicates the end of the String. A character is declared with the type specifier $char$. For example:

```
char c = ‘a’;
```

Enumerations. The enumeration type is not necessarily a primitive type, but is handled like one on the low level. An enumeration has to be declared, see Enum Typedef. After Declaration, an Enum value can be selected. An Enum is declared with the name of the Enum itself. For Example:

```
State s = State.Normal;
```

Predicates. A predicate type holds a reference to a function. This means that a predicate type can be "called" like a function. You can read more about it in the Section "Predicates". A predicate type is declared using the $func$ type specifier. For example:

```
func pred = my_predicate;
```

In this example, my_predicate is the name of a function already declared. By calling this predicate the program will execute the function using the given parameters.

Warning: Using predicates requires caution, especially with anonymous predicates. See the Predicates Section for more information. When handled wrong, predicates can cause the program to crash or the stack to be misaligned.

Composite Types

Arrays. An array holds a fixed amount of values of the same type. Arrays can be created with a fixed size, or by allocating memory on the heap. Arrays can hold any type, both primitive and composite, except the proviso type. An array is declared like this:

```
int [5] arr = {1, 2, 3, 4, 5};
```

In this case, an array of the length 5 with the element type int is created. Also, using an array init expression, we assign the array a value. Arrays of arrays are also possible:

```
int [2] [2] mat = {{1, 2}, {3, 4}};
```

In this case, we declare an integer matrix of the size 2 by 2. Again, we initialize the matrix with an array init expression, except the values of this expression are array init expressions themselves. Arrays can be of any dimension. It may be worth considering to flatten the array when using pointers. Arrays can also be initialized with the following syntax:

```
void [5] arr = [10, ‘c’, true, Point::(10, 5)];
```

Using brackets instead of braces, the array will be treated as a void type array. This means that any value in the array init is accepted. Types larger than 1 data word will be split into n data words, the length of the resulting array is equal to the sum off all the word sizes of the elements.

Note: When using a don’t care array, the original types of the array elements will be lost. This means the mapping must be stored or known. Don’t care arrays are meant to be a way around having to cast every element to a void type when creating a void array.

Structs. A struct holds a collection of fields. Each of which has its own separate type. Since all fields of the struct have a fixed size, the struct has a fixed size as well. The struct has to be declared to be a type using a Struct Typedef. A struct is initialized like this:

```
MyStruct s = MyStruct::(5, true);
```

In this case, a struct called MyStruct is initialized. The Type of MyStruct has an integer and boolean field, both of which we have to give an initial value in the Structure Init Expression.

Pointers. A pointer holds a reference to another object. When dereferencing a pointer, we get the value the pointer points to. A pointer can be created manually, or by retrieving the address of an object:

```
int* p = &var;

int* p = (int*) 0;
```

In the first example, we get the address of an object with an address-of expression. This will return a pointer to the object. The type of the pointer is determined through what type it points to. The $*$ after the type signals that this is in fact a pointer. In the second example we create a Null-Pointer by casting the value 0 to an integer pointer. This can be useful in data structures to signal an invalid pointer or Null-Value. Pointers can point to a pointer as well:

```
int** p0 = &p;
```

In this example, p is an integer pointer. By getting the address of this pointer, we now have a pointer of the second grade. This means this pointer is now pointing to a pointer that points to the target value. We can retrieve the pointer by dereferencing this pointer twice.

Expressions

Operators

Snips brings a wide range of operators to the table. You can see all down below. The higher up, the higher precedence has the operator.

Operator	Description
Atom, Enum Selection: $a, 10, Enum.v$	Basic building blocks. Atoms can be immediate or variables. Enum Selection does count as a new Enum Immediate value.
Array Selection: $a\,[0], m\,[0][1]$	Select an element from an array.
Struct Select: $st.v, st{-}{>}\ v$	Select a field from a struct.
Increment, Decrement: $i++, i--$	Increment or decrement a primitive value.
Unary Minus: $-val$	Negates value.
Not: $!\,b, \sim i$	Boolean negation via !, bitwise complement with ~.
Type Cast: $(int)\ b, (void)\ k$	Cast value to type.
Dereference: $*\ val$	Treat value as address, load value at this address.
Address Of: $\&v$	Get the address of a value.
Size of: $sizeof(v), sizeof(Struct)$	Get the word size of a value or of a type.
Multiplication, Division: $a * b, a/b$	Multiply, Divide.
Addition, Subtraction: $a + b, a - b$	Add, Subtract.
Shift: $a << b, a >> b$	Shift logical.
Comparison: $a < b, <=, ==, != , >, >=$	Compare two values based on comparator.
Bitwise And, xor and or: $a\&b, a\hat{}b, a\vert b$	Perform bitwise and, xor and or operation.
Logical And, Or: $a\&\&b, a\vert\vert b$	Perform boolean and, or operation
Ternary: $(a)?\,x : y$	Select one of two values based on compared value.
Array Initialization: $\{a, b, \dots, c\}$	Create a new array of values of the same type.
Struct Initialization: $Struct::(a, b, \dots, c)$	Create a new Struct instance by providing values for the structs fields.

Building Expressions

Expressions in Snips are inductively defined. This means that operations listed above can be capsuled and used recursiveley.

For example, this gives us an expression like this:

$$S :: (10, true, \{a, b + 4\})$$

We initialize a struct, and give it the initial values for the fields, once with an int-literal, once with a boolean literal, and once with an array init, where the fields are an identifier reference and an addition to a variable.

Expressions will always have a return value. This value can be received by a declaration, function call, or another expression like in the example by the structure init.

Statements

In the following we are going to use the following macros:

<table>
<tr><th>Macro Name</th><th>Value</th></tr>
<tr><td>Assign Operator</td><td>(+, -, *, /, %, <<, >>, &, &&, |, ||, ^)?=</td></tr>
<tr><td>Type</td><td>A type denoter, like described previously</td></tr>
<tr><td>Identifier</td><td>Must match the regex
$([a - z]||[A - Z]|\_)([a - z]||[A - Z]||[0 - 9]||\_) *$</td></tr>
<tr><td>Expression</td><td>An inductively defined expression, using operators like described previously.</td></tr>
<tr><td>Statement</td><td>Any of the statement listed below.</td></tr>
<tr><td>CStatement</td><td>A compound statement, is equal to [Statement] | ‘{‘ [Statement]* ‘}’</td></tr>
</table>

Simple Statements

Simple statements in snips are linear. This means they do not add code flow or looping to the code.

<table>
<tr><th>Statement</th><th>Description</th></tr>
<tr><td>Variable Declaration:
[Type] [Identifier] ‘=’ [Expression]‘;’</td><td>Declares a new variable with a name and a value. The variable is either placed in a register or on the stack.</td></tr>
<tr><td>Variable Assignment:
[Identifier] [Assign Operator] [Expression]’;’</td><td>Assigns the resulting value of the expression to the variable, and applies the assign operator when assigning.</td></tr>
<tr><td>Function Call:
[Identifier] ‘(‘ ([Expression] ‘, ‘)* [Expression] ‘);’</td><td>Call the stated function with the results of the stated expressions, but discard the return value.</td></tr>
<tr><td>Return value:
‘return;’ | (‘return ‘ [Expression] ‘;’)</td><td>Return from the current function or return the value of the expression.</td></tr>
</table>

Flow Statements

Flow statements allow for branching and looping within the code.

Statement	Description
If-Statement: ‘if (‘ [Expression] ‘)’ [CStatement] (‘else if (‘ [Expression] ‘)’ [CStatement])* (‘else ‘ [CStatement])	Declares a new variable with a name and a value. The variable is either placed in a register or on the stack.

3. ARM Assembly & Output

Compiler Assembly Conventions

Register Usage

The Snips Compiler maps certain functionality to the target machine registers, the mapping can be seen below:

Register	Functionalities
R0	Holds the result of an arithmetic operation, can hold a parameter in a function call, holds the return value of a function call if the return type word size is equal to 1.
R1, R2	Used as operands for arithmetic operations, can hold a parameter in a function call.
R3-R9	Part of the Register Stack, used to hold variables with word size 1.
R10	Holds PC backup during syscalls.
R11	Acts as the frame pointer.
R12	Holds the exception code when an exception is thrown.
R13	Acts as the stack pointer.
R14	Acts as the link register.
R15	Acts as the program counter.

Heap Management

The heap is managed by a custom linked list implementation that is optimized for a minimal memory footprint, and easy allocation and de-allocation. A single node consists of a size entry and the data block.

The size entry contains the word size of the memory block plus the size entry itself. The memory block simply contains the payload. Starting from the heap start, memory blocks can be created. When creating a new memory block, resv iterates over the heap. It starts at the heap start, loads the first size entry field n. If $n = 0$, no entry is here, and the memory location is free. If $n > 0$, a memory block of the size n lies ahead. The routine can then jump n words further and try again. If $n < 0$, a free memory section of the size $-n$ lies ahead. If the requested size s is less than n, the memory section ahead is large enough. Write s to the current cell, write $n + s$ to the current cell plus s cells. By doing this we split the free section, use the whole section or a part, and keep the heap structure intact. A pointer that points to the heap always points to the first word of the payload. This is especially true for pointers created by the resv routine. This means that $p - 1$ would point to the size entry, where p is a pointer to the heap.

The free routine now only has to negate the size entry of a memory section to mark it as unused. The routine can locate this entry through $p - 1$, where p is the passed pointer.

The hsize routine can load the size entry using the method described earlier. Note that the function will load a random value if the pointer does not point to the heap but to the stack. The resv and free routine also implement some heap defragmentation mechanisms that are automatically executed. These mechanisms make sure to defragment multiple free memory sections after another into one big free section. This way the search time for a free section is shorter and the utilization is denser.

4. Built in Libraries

All listed libraries can be included either with their shortened path, f.E. boolean.sn, or with their full path lib/std/type/boolean.sn.

All listed functions and are $shared$ if not explicitly stated otherwise.

All listed Struct Types are $restricted$ if not explicitly stated otherwise. Libraries will offer a $create()$ method to create a new instance of the struct.

Memory & Heap Routines

resv.sn

Full Path: -
Package Type: Base Package
Namespacing: None
Description:

Contains the memory reserve routine that is responsible for allocating a requested block size in the heap. This file is included dynamically when the resv function is called.

Function Header	Description
void* resv(int size)	Reserves a memory block of given size + 1. The additional word is used for heap management. Returns a pointer to the memory location. The returned pointer points to the second data word of the block. The first word contains the heap management data. $O(n)$ where n is the number of heap elements.

free.sn

Full Path: -
Package Type: Base Package
Namespacing: None
Description:

Contains the memory free routine that is responsible for freeing memory in the heap. This file is included dynamically when the free function is called.

Function Header	Description
void free(void* p)	Frees the given pointer from the heap. This is done by negating the heap management data on the block head. The data will remain in the heap, but can from now on be overwritten. $O(n)$, where n is the amount of heap elements.

hsize.sn

Full Path: -
Package Type: Base Package
Namespacing: None
Description:

Contains the memory hsize routine that can determine the size of a heap memory block. This file is included dynamically when the hsize function is called.

Function Header	Description
int hsize(void* p)	Returns the size of the memory section. The pointer should point to a memory block in the heap. The size is determined by reading the heap management data and subtracting 1. $O(1)$

init.sn

Full Path: -
Package Type: Base Package
Namespacing: None
Description:

Contains the memory init routine that allocates a pointer and assigns a value to at the pointer location.

Function Header	Description
void* init<T>(T value)	Allocates memory using resv and assigns the given value at the memory location. Returns a pointer to the memory location.

System Operators

__op_div.sn

Full Path: -
Package Type: Base Package
Namespacing: None
Description:

Contains a routine to divide two integers. This file is included dynamically when the division operator is used.

Function Header	Description
int __op_div(int a, int b)	Calculates $\frac{a}{b}$ and returns the result.

__op_mod.sn

Full Path: -
Package Type: Base Package
Namespacing: None
Description:

Contains a routine to compute the rest of a integer division. This file is included dynamically when the modulo operator is used.

Function Header	Description
int __op_mod(int a, int b)	Calculates $a \% b$ and returns the result.

Types

boolean.sn

Full Path: lib/std/type/boolean.sn
Package Type: Includes: string.sn
Namespacing: The entire package is namespaced in 'Boolean'.
Description:
Contains utility around the boolean type.

Function Header	Description
bool parseBool(char* str)	Parses a boolean value from given String. The result will be $true$ if the String is equal to "true". In any other case, the result will be false.
char* toString(bool b)	Converts given boolean into String representation. The result will be, depending on given value, "true" or "false". Returns a pointer to the created String.

integer.sn

Full Path: lib/std/type/integer.sn
Package Type: Includes: linked_list.sn
Namespacing: The entire package is namespaced in 'Integer'.
Description:
Contains utility around the integer type.

Function Header	Description
int parseInt(char* str)	Parses a integer from given String. The String has to match the pattern: $-?\,([0-9])^*$. The result is the parsed int.
char* toString(int num)	Converts given int to String representation. The resulting string will contain a sign if the int was negative. Returns a pointer to the created String.

Namespace Array:

Function Header	Description
void sort(int* arr, int size, func (int a, int b) -> bool pred)	Sorts given integer array. The sorting is done using a bubble sort algorithm, so the runtime complexity equals $O(n^2)$. The sorting is determined by given predicate. Writes back the result in the array at the given pointer.

string.sn

Full Path: lib/std/type/string.sn
Package Type: Includes: linked_list.sn
Namespacing: The entire package is namespaced in 'String'.
Description:
Contains utility for String operations.

Function Header	Description
bool equals(char* str0, char* str1)	Checks if the two given Strings are equal on a char level. All contained chars must have the same value, and both Strings have to have the same case.
char* substring(char* str, int begin, int end)	Cuts out a part of the String specified by the bounds. The begin index marks the first char to be included in the new String, the end index marks the last char to be included. The 0 char is inserted at the end of the resulting String. The begin index should be less than the end index. In case the indexes are out of the bounds of the String, the entire String is copied. Returns a pointer to the new String on the heap.
int length(char* str)	Returns the length of the String. This includes the 0 char. Returns the length.
char charAt(char* str, int i)	Returns the char at the given index, or the 0-char if the index is out of bounds.
char* concat(char* str0, char* str1)	Concatenates the two given Strings. The resulting String will contain the entire first String followed by the entire second String. The 0 char of the first String is discarded. Returns a pointer to the new String.

Data Structures

linked_list.sn

Full Path: lib/std/data/linked_list.sn
Package Type: Base Package
Namespacing: The entire package is namespaced in 'List'.
Description:

Provides a data structure that can hold a variable number of items. Each element is capsuled in a own node. Using pointers, these nodes are chained together. The nodes are stored on the heap.

Struct Name	Struct Fields	Description
ListNode<T>	ListNode<T>* next T value	Capsules a single data element. Also contains a pointer to the next list node in the list. Is set to 0 if this is the last element in the list.
LinkedList<T>	ListNode<T>* head ListNode<T>* tail T defValue	Capsules two list node pointers, one pointing to the first, and one to the last element. Both pointers are initialized to $null$. Also contains a default value, that is returned when attempting to get a value out of bounds.

Function Header	Description
LinkedList<T>* create<T>(T defValue)	Initializes a new linked list struct. Sets the pointers to $null$. Sets the given default value. $O(1)$
void destroy(LinkedList<void>* l)	Destroys the given list and all of its contained elements by freeing every node and the list itself from the heap. $O(n)$
void add<T>(LinkedList<T>* lp, T x)	Adds given x to the list by encapsuling it in a new list node and appending the element at the end of the list. This uses the tail pointer of the list, so the complexity is $O(1)$.
T get<T>(LinkedList<T>* lp, int i)	Returns the value of the i-th element in the given list or the default value when out of bounds. $O(n)$
ListNode<T>* getNode<T>(LinkedList<T>* lp, int i)	Returns a pointer to the list node in the list at the i-th position in the list. $O(n)$
bool contains<T>(LinkedList<T>* lp, func (T, T) -> bool pred, T x)	Checks wether given value is the value of one of the contained list nodes. To check if x matches the currently searched value, the predicate pred is used. If it returns true, the values will be treated as equal and true will be returned. $O(n)$
ListNode<T>* find<T>(LinkedList<T>* lp, func (T, T) -> bool pred, T x)	Attempts to find the first list node in order in the list that has the given value x. To check if x matches the currently searched value, the

	predicate pred is used. If it returns true, the values will be treated as equal and the value will be returned. Returns a pointer to this list node. $O(n)$
int indexOf(LinkedList<void>* lp, ListNode<void>* n)	Returns the index of the given node in the list, or -1 if the node is not part of the list. $O(n)$
int size(LinkedList<void>* lp)	Returns the number of nodes in the list. $O(n)$
void remove(LinkedList<void>* lp, int i)	Returns the list node at the i-th position. Updates the pointers of the neighbours to fill the gap. Also frees the removed node. $O(n)$
void clear(LinkedList<void>* lp)	Removes all elements stored in the list. $O(n)$
bool isEmpty(LinkedList<void>* lp)	Returns if the given list contains no elements.

binary_tree.sn

Full Path: lib/std/data/binary_tree.sn
Package Type: Base Package
Namespacing: The entire package is namespaced in 'Tree'.
Description:

Provides a data structure that can hold a variable number of elements and organizes the elements to minimize search time.

Struct Name	Struct Fields	Description
TreeNode<T>	TreeNode<T>* left TreeNode<T>* right T value	Capsules pointers to left and right child nodes, and a value.

Function Header	Description
TreeNode<T>* create<T>(T value)	Initializes a new Tree. Initializes pointers to 0. Sets given value to node value. Returns pointer to newly created node. $O(1)$
TreeNode<T>* insert<T>(TreeNode<T>* root, T value)	Capsules given value in a new Node and inserts it into given Tree. The insertion location is determined by numerically comparing the value. Returns a pointer to the newly created node. $O(\log n)$

queue.sn

Full Path: lib/std/data/queue.sn
Package Type: Base Package
Namespacing: The entire package is namespaced in ‘Queue’.
Description:

Provides a data structure that can hold a fixed number of elements and acts like a bypassing FIFO. The queue is implemented with a circular array.

Struct Name	Struct Fields	Description
CyclicQueue<T>	T* storage T defValue int size int head int tail bool isEmpty	Capsules a pointer to the storage array that holds the contained elements. The field defValue holds the default value that is returned f.E. when a dequeue is attempted while the queue is empty. Also capsules a field that holds the max number of elements. Holds two indices that act as pointers in the array, used to determine where the head and tail of the queue in the array is. Finally, it holds a boolean that stores wether the queue is full or not.

Function Header	Description
CyclicQueue<T>* create<T>(int size)	Initializes a new Queue with given size. Creates a storage array on the heap and initializes the CyclicQueue struct. Returns a pointer to the created queue. $O(1)$
void destroy(CyclicQueue<void>* queue)	Frees the given queue object and frees the storage array of the queue in the heap. $O(1)$.
bool isEmpty(CyclicQueue<void>* queue)	Returns wether the queue contains any elements or not. $O(1)$
bool isFull(CyclicQueue<void>* queue)	Returns wether the number of contained elements is equal to the maximum of elements. $O(1)$
void enqueue<T>(CyclicQueue<T>* queue, T value)	Adds a new element to the tail of the queue. If the queue is full, the element wont be added. $O(1)$
T dequeue<T>(CyclicQueue<T>* queue)	Removes the head of the queue and returns it. If the queue is empty, the function will return 0. Note that this may cause problems with types larger than 1 word. $O(1)$
void clear(CyclicQueue<void>* queue)	Removes all elements from the queue. This is done by setting the pointers, rather than clearing the actual values. $O(1)$

int size(CyclicQueue<void>* queue)	Returns the number of currently stored elements in the queue. $O(1)$

stack.sn

Full Path: lib/std/data/stack.sn
Package Type: Includes: linked_list.sn
Namespacing: The entire package is namespaced in ‘Stack’.
Description:

Provides a data structure that can hold a variable amount of elements and acts like a LIFO buffer.

Struct Name	Struct Fields	Description
StackedList<T>	LinkedList<T>* list	Capsules only a linked list. This list will hold the elements.

Function Header	Description
StackedList<T>* create<T>(T defValue)	Initializes a new Stacked List and the capsuled linked list. Returns a pointer to the newly created stack. $O(1)$
void destroy(StackedList<void>* stack)	Frees the stack and the capsuled linked list and all of the lists contained elements. $O(n)$
void push<T>(StackedList<T>* stack, T value)	Adds given element to the top of the stack. $O(1)$
bool isEmpty(StackedList<void>* stack)	Returns wether no elements are stored in the stack. $O(n)$
int size(StackedList<void>* stack)	Returns the number of contained elements in the stack. $O(n)$
T peek<T>(StackedList<T>* stack)	Returns the top element of the stack, but does not remove it. Returns the default value if the stack is empty. $O(n)$
T pop<T>(StackedList<T>* stack)	Returns the top element of the stack and removes it. Returns the default value if the stack is empty. $O(n)$

hash_map.sn

Full Path: lib/std/data/hash_map.sn
Package Type: Includes: linked_list.sn, hash.sn
Namespacing: The entire package is namespaced in ‘Map.
Description:

Provides Utility to create and manage a HashMap data structure with parameterized Key and Value Types.

Struct Name	Struct Fields	Description
MapEntry<X, Y>	X first int pad	A tuple capsuling two different data types. Padding is required since SIDs may be

	Y second	enabled or not. If SIDs are enabled, the tupleKeyMatcher will match the SID and the key. If they are disabled, matcher will match the key and the 0 padding.
HashMap<K, V>	LinkedList<MapEntry<K, V>*>** int size MapEntry<K, V>* defValue	The hash map itself, capsuling a storage array out of linked lists with the size specified in the capsuled size field, and a default value that is returned when a key is not found.

Function Header	Description
restricted bool tupleKeyMatcher<K, V>(MapEntry<K, V>* t0, MapEntry<K, V>* t1)	Used internally to match the keys of map entries.
HashMap<K, V>* create<K, V>(int size, V defValue)	Creates a new HashMap instance, and initializes the storage array and default value.
void destroy(HashMap<void, void>* map)	Destroys all linked lists, frees the storage and default value and the map itself.
void put<K, V>(HashMap<K, V>* map, K key, V value)	Put given value under given key into the map. Does not check if the key already existed.
void replace<K, V>(HashMap<K, V>* map, K key, V value)	If given key is in the map, replace the value of this key with the given value.
V get<K, V>(HashMap<K, V> map, K key)	Check if given key is in the map, if yes, return the value, if not return the default value.
void remove<K, V>(HashMap<K, V>* map, K key)	If the given key is contained in the HashMap, remove the first occurrence of it.
bool contains<K, V>(HashMap<K, V>* map, K key)	Check if given key is contained in the map.
LinkedList<MapEntry<K, V>*>* getEntries<K, V>(HashMap<K, V>* map)	Retrieves all entries from the HashMap sorted by hash value ascending, and adds them into a single linked list and returns that list.
LinkedList<V>* getValues<K, V>(HashMap<K, V>* map)	Retrieves all stored values from the HashMap sorted by hash value ascending, and adds them into a single linked list and returns that list.

I/O

display.sn

Full Path: lib/std/io/display.sn
Package Type: Includes: linked_list.sn, integer.sn
Namespacing: The entire package is namespaced in 'Display.
Description:
Contains functions and utility for text input/output.

Function Headers	Description
restricted char getChar()	Returns the currently entered char via keyboard.
restricted void putChar(char c)	Stores given char at the current cursor location of the display and increments the cursor.

Namespace Print:

Function Header	Description
void println()	Sets the cursor of the display to the start of the next line.
void printString(char* str)	Prints out given String at the current cursor location.
void printlnString(char* str)	Prints out given String at the current cursor location and moves the cursor to the start of the next line.
void printf(char* c, void* f)	Prints out given String with formatting awareness. This means that escape sequences and placeholders are respected. Placeholders are: %d, %s. Where %d means print a decimal number at this position, and %s means print a string at this position. The values for these placeholders are obtained from the parameter f. f should be an array that is equal to the number of placeholders. If the function encounters the n-th placeholder, it will load the n-th data word from the array. If the placeholder is a decimal value, the loaded value will be printed out. If the placeholder is a string, the loaded value should be a pointer to the string to be printed. See $release/examples/quine.sn$ for an example. Escape sequences are: \n, \”, \t. Where \n will cause the function to jump to the start of the next line and continue printing, \” will result in a " in

	the output and \t will cause the function to print a tab or four spaces.

Namespace Scanf:

Function Header	Description
char* scanf()	Reads an input from the user via the keyboard. The input is confirmed with the character `.`, so if the user inputs it, the function will terminate, create a string out of the inputted chars and will store it in a heaped char array. Returns a pointer to the heaped char array.

Math

math.sn

Full Path: lib/std/math/math.sn
Package Type: Base Package
Namespacing: The entire package is namespaced in 'Math'.
Description:
Contains math functions and operations.

Function Headers	Description
int pow(int x, int n)	Returns x to the n-th power.
int abs(int x)	Returns the absolute value of x.
int fac(int n)	Returns the n-th faculty.

matrix.sn

Full Path: lib/std/math/matrix.sn
Package Type: Base Package
Namespacing: The entire package is namespaced in 'Matrix'.
Description:
Contains matrix utility. Currently only matrix multiplication.

Function Headers	Description
int mult(int* m, int* n, int dim0, int dim1)	Multiplies the two given matrices. Stores the result in an integer array on the heap. Returns a pointer to the result. The first matrix should have dim0 columns, the second matrix should have dim1 columns. The resulting matrix will have the dimensions $dim0 \times dim1$. Does not check matrix size compatibility.

vector.sn

Full Path: lib/std/math/vector.sn
Package Type: Base Package
Namespacing: The entire package is namespaced in 'Vector'.
Description:
Contains utility functions for vector operations, like the scalar product.

Function Headers	Description
int scalarProd(int* v, int* w, int l)	Calculates the scalar product of the two given vectors. Both vectors have to have the given length. Returns the result.

Utility

bits.sn

Full Path: lib/util/bits.sn
Package Type: Base Package
Namespacing: The entire package is namespaced in 'Bits'.
Description:

This library can be used to create something like bit-banks. Meaning a single Integer can hold 32 States for an automaton, flag-set etc. These bits can easily be modified with this library.

Function Headers	Description
bool isBitSet(int x, int i)	Checks wether the bit at the i-th place in given word x is a 1. The lowest bit has the index 0.
int setBit(int target, int i, bool val)	Sets the bit in the target word at the i-th place to given val. Returns the resulting data word.
int toggleBit(int target, int i)	Sets the bit in given target word at the i-th place to its complement. Returns the resulting word.
Int clearBit(int target, int i)	Sets the bit in given target word at the i-th place to 0. Returns the resulting word.

color.sn

Full Path: lib/util/color.sn
Package Type: Base Package
Namespacing: The entire package is namespaced in 'Colors'.
Description:

This library contains utility related to colours, specifically rgba-coloring.

Struct Name	Struct Fields	Description
Color	int rgba	Encodes the R, G, B and A color channels, by creating a single integer where each byte contains the value in range of 0-255.

Function Headers	Description
Color* create(int r, int g, int b, int a)	Encodes the four given color values and creates a new Color Struct on the heap. Returns a pointer to the struct.
int getRed(Color* c)	Returns the value of the red channel.
int getGreen(Color* c)	Returns the value of the green channel.
int getBlue(Color* c)	Returns the value of the blue channel.
int getAlpha(Color* c)	Returns the value of the alpha channel.

graphics.sn

Full Path: lib/util/graphics.sn
Package Type: Includes: color.sn
Namespacing: The entire package is namespaced in ‘Graphics’.
Description:

This library contains utility related to colours, specifically rgba-coloring.

Struct Name	Struct Fields	Description
Graphics2D	Color* color, void* targetBase, int targetWidth, int targetHeight	Encapsulates a draw color, the base address of the vbuffer of the target, and a width and height in pixels of the target.

Function Headers	Description
Graphics2D* create(void* base, int w, int h, Color* c)	Creates a new Graphics Object and sets given values.
int pixelAddress(Graphics2D* g, int x, int y)	Calculates the absolute memory address of the given pixel with respect to the specified canvas address base by the graphics object.
void setPix(Graphics2D* g, int x, int y)	Colors a single given pixel with the current color set to the graphics object.

Namespace Draw:

Function Header	Description
void drawRect(Graphics2D* g, int x, int y, int w, int h)	Draws a rectangle on the canvas, with the top left corner at the given (x, y) coordinates, and with given width and height.
void fillRect(Graphics2D* g, int x, int y, int w, int h)	Fills a rectangle on the canvas, with the top left corner at the given (x, y) coordinates, and with given width and height.