Code Snippet

Aiman

October 6, 2019

1 A-SMUL Code Structure

$$\textbf{let}\ n = 1; \textit{// Howdy}$$

1.1 Comments

A comment is simply any line that starts with "//".

// This is a commment. Hodgepodge

1.2 Variables, Aliases

In general, one defines a new variable by:

$$\textbf{let}\ \underbrace{<\textbf{qualifier}>\ <\textbf{type}>}_{\textit{optional}}\ <\text{name}> = <\text{rvalue}>$$

One can define an *alias*; a reference to another variable by using the arrow "=>" instead of the equal sign "=":

$$\textbf{let}\ \underbrace{<\textbf{qualifier}>\ <\textbf{type}>}_{\textit{optional}}\ <\text{name}> => <\text{lvalue}>$$

By "lvalue", we mean a value that can sit in the left hand side of an assignement, such as a variable identifier or an identifier of an element in a defined container.

The type of a value can be:

- **number**: A real number; something between $-\infty$ and ∞.

- **integer**: These are the "counting" numbers: $\cdots, -54319, \cdots, -2, -1, 0, 1, 2, 124135, \cdots$
- **text**: Should come inside quotes: e.g. "This is some text".
- **none**: Empty. Void. Nothing at all. The only value of type none is **None**.
- **bool**: A boolean can take one of two values, **True** or **False**.
- **variant**: A variant can take any value at all of all types.
- **list[<type>]**: A list of values of a given type.
- Combined types: Types can be combined together using |. For example, a variable of type **bool|none** can contain **True**, **False** or **None**.
- Other types include:
  - Tuples: **tuple[<type1>, ···, <typeN>]**
  - Dictionaries: **dictionary[<key type> => <value type>]**
  - Bitfields: **bitfield**
  - Anonymous functions: **function[<argument type>:<return type>]**.

If a type is not specified in a variable definition, then the type of rvalue is assumed.

1.3 Flow control (if, else, while ..etc)

For flow control, the most general form of an if clause is the usual:

```
if <condition> do
    <commands>
elseif <condition> do
    <commands>
else
    <commands>
end
```

Whereas a general while clause is written as:

```
while <condition> do
    <commands>
end
```

1.4 Namespaces

One can create namespaces using:

namespace <namespace name>
  <commands and definitions>
end

Accessing variables defined in a namespace is done via ":". For example, if a variable **a** is defined in a namespace called **hodgepodge**, then the variable can be accessed as **hodgepodge:a**.

1.5 Markup

The key feature of A-SMUL is that it seamlessly integrates markup structures into the language. In place of commands, one can place one of the following markup components:

- Main components:

  {<name>}

  $$< switchname1 > | switchname2 : \underbrace{\text{<bool\_rvalue>}}_{optional} | < switchname3 >$$

  <**tagname1**> = <,rvalues>   <**tagname2**> = <,rvalues>
  // With possibility to add tags, switches, or nest other components

  {/<name>}

- Sub components:

  <name>[
    // Tags, switches, nested components
  ]

- Id components:

  [<id name>= <const rvalue of type text|integer>]
    // Tags, switches, nested components
  [<id name>]

There is no symantic/real difference between a main component and a subcomponent. The difference in syntax is for applications of A-SMUL to take advantage of.

To access data in the markup structures in the components, one begins with **@**. Then main and sub-components are namespaces that contains variables of type **tuple[<,rvalue types>]** for tags, and functions that return a value of type **bool** for switches (returns **True** if switch is simply placed there. And to the rvalue given to it if there is one.). For example, to access the first value (0th value) given to a tag called **mytag** inside a main component called **mymain**, one writes **@mymain:mytag[0]**. Id components are dictionaries with keys of type **text|integer**.

1.6 Example

```
// A function definition
function say_hello(text name) none :
    print("Hello" + name)
    return  value
end

// Some variable definitions
let const number PI = 3.14
let text hi = "Hi"
let bool is_happy = True
let bool|none user_preference = None
let list[number|text] some_list = [2, 1.2, "blabla"]
let list[number|text] some_list = [2, 1.2, "blabla"]
let dictionary[text|number => variant] some_dict = [1 => "Hello", "b" => True]
let tuple[integer, text, bool] some_3_tuple = {-1, "Howdy", False}
// An alias of hi
let hello => hi
```

```
// Anonymous functions
let function[number:tuple[number,number]] my_lambda =
                        <number x: tuple[number, number] {x/2, sqrt(x)} ;>
// Alternatively, type inference should allow for:
let function[number:tuple[number,number]] my_lambda2 = <x: {x/2, sqrt(x)} ;>
// Or even:
let my_lambda3 = <number x: {x*x, 2*x} ;>

 {mybutton}
   disabled : user_preference
     position = 20, 30     button_label = hello     colour = "purple"
     on_click = say_hello
 {/mybutton}

 let tuple[number, number] position = @mybutton:position
 if  PI < 22/7 do
     @mybutton:on_click (" Jacky")
 else
     print(some_dict[1])
 end
```