SNL

Stage (null) Language Reference Manual

James Lin
Alex Liu
Daniel Maxson
Andre Paiva

October 27, 2014
Columbia University

Table of Contents

1. Introduction

This is a reference manual for the SNL programming language as implemented by James Lin, Alex Liu, Daniel Maxson, and Andre Paiva. This manual aims to describe the various features of SNL for programming.

SNL, which stands for Stage Null Language, is a language designed to model role-playing game scenarios based on the structure of state machines. It allows users to programmatically re-create sequential scenarios as "stories." Its syntax aims to be simple so that adolescents can easily understand how to use it. Where possible, SNL uses intuitive keywords in place of symbols, since symbols commonly known in the CS community can be confusing to those who are unfamiliar with programming. This will encourage children to learn how to write code in a controlled, fun environment while allowing them to exercise their creativity. SNL is robust enough to program typical computer science algorithms like GCD or factorials as well as more creative applications like interactive fiction or psychology studies.

The SNL language is a product for COMS W4115, a course at Columbia which the four creators took in the Fall of 2014.

2. Lexical Elements

2.1. Comments

All comments are single-line and denoted by the # character. Any content to the right of the # will be ignored.

2.2. Identifiers

Identifiers are sequences of characters used for naming variables, functions, and stages. All characters must be alphanumeric or the underscore character. The first character must be an alphabetic character.

2.3. Keywords

`if`	`else`	`not`
`and`	`or`	`do`
`to`	`start`	`next`
`is`	`local`	`true`
`false`	`return`	`recipe`
`done`	`input`	`of`

2.4. Literals

There are several literals in SNL. These include integer literals, float literals, boolean literals, string literals, and list literals.

2.4.1. Integer Literals

An integer literal is a sequence of digits. All digits are taken to be decimal. 12 is an example of an integer constant.

2.4.2. Float Literals

A float literal consists of a decimal point, and either an integer part or a fraction part or both. 5.0 and 5. and .5 are all valid floating constants.

2.4.3. Boolean Literals

A boolean literals is either `true' or `false'.

2.4.4. String Literals

A string literal is a sequence of chars. These are sequences of characters surrounded by double quotes. Two examples of string literals are “hello” and “world”.

2.4.5. List Literals

A list literal is a sequence of literals that have been placed between square brackets `[]' and separated by commas `,'. Lists can contain one or more types and are mutable. [1,2,3,4] and [1,2,true,”peggy”] are both examples of lists.

2.5. Operators

An operator is a special token that performs an operation on two operands. More information about these are provided in the Expressions and Operations section (4).

2.6. Separators

A separator separates tokens. These are not seen as tokens themselves, but rather break our language into discrete pieces.

2.6.1. White Space

White space is the general purpose separator in SNL. More information is provided in the White Space section (2.7).

2.6.2. Comma ,

The comma is a separator, specifically in the context of creating lists (and their elements) and also for parameters passed to a function which is being called.

2.6.3. Colon :

The colon is a separator in the context of starting a new stage or recipe. The separator will be placed right after the name of the stage or after the recipe declaration.

2.7. White Space

2.7.1. Spaces

Spaces are used to separate tokens within a single line outside of the creation of list and the first line of a stage.

2.7.2. New Line

New lines are used to separate expressions from one another. There is only one expression allowed per line.

3. Data Types

3.1. Variables in SNL are dynamically typed, similar to Python or Perl. Variables are implicitly assigned a type depending on the value assigned to it. You can find more information about these constants in the section about Literals (2.4).

3.2. The following are SNL's built-in data types:

`int`	A series of digits
`float`	A series of digits with a single '.'
`bool`	Boolean values of True or False
`string`	A sequence of characters within " "
`list`	A sequence of items enclosed by [ ]

4. Expressions and Operators

Operator	Use
+	Addition, String Concatenation
–	Subtraction
*	Multiplication
/	Division
=	Equals
!=	Not Equals
<	Less Than
<=	Less Than or Equals
>	Greater Than
=>	Greater Than or Equals
( )	Grouping expressions/statements
is	Assignment
of	Access element from list

5. Recipes

5.1. Recipe Definitions

A recipe is set of stages with an implicit Character containing any items passed into it. If there are any arguments passed into the recipe, the `to' keyword must come before the

comma-separated list of arguments. The 'return' keyword will return at most one item back to the stage from which it was initially called.

An example of a recipe built using multiple stages:

```
start example_program:
      lst is [3, 4, 5, 6]
      do inc_list to lst
      show lst


recipe inc_list to my_list:  #declaration of recipe

      start start_inc_list:
            length is do get_len to my_list  #calling a recipe
            index is 0
            next loop_start

      loop_start:
            if index < length
            (next s_list_modifier)
            else (return my_list)  #returning out of our recipe

      s_list_modifier:
            index of my_list is index of my_list + 1
            index is index + 1
            next loop_start
done
```

5.2. Calling Recipes

The keywords `do' and `to' mark recipe calls, and the comma is used to separate function arguments. For example:

```
      do foo to bar, baz
```

When there are no arguments to a recipe, `to' must be omitted such as:

```
      do foo
```

6. Program Structure and Scope

6.1. Program Structure

Each program must be written within one source file and are a combination of a single Universe along with Stages and Recipes. These can each be defined anywhere within the file.

6.2. Stages

A Stage will consist of a series of statements. The starting Stage for each recipe or program will be specified by the the 'start' keyword. Next will come the name of the Stage followed by a colon.

For all Stages outside of the starting Stage of a recipe or program, only the name of the Stage and the colon should be used.

Within a Stage, the 'next' keyword will designate the following Stage to jump to. These will control the movement of the Character between different Stages, particularly by utilizing conditional statements to vary between different next Stages.

6.3. Scope

6.3.1. Global Scope

All variables defined either in a Stage are by default part of the global scope and can be accessed and modified from any of the other stages within the program.

6.3.2. Scope within a Stage

To declare a variable at a Stage scope you will use the reserved keyword 'local' followed by the variable name. For example:

```
local colour_of_ball is "blue"
```

6.3.3. Scope within a Recipe

A recipe does not have any access to the Universe scope but will only have access to any items passed in or declared within this recipe. Users must be careful to remember which recipe they are declaring variables in at each stage.

7. A Sample Program

```
# These are comments
# Recipe to calculate health lost

recipe calc_damage_done to attack, defense:
      start calc:
            if attack < defense
            return 1
            else
            return attack - defense
done

recipe does_hit_land to attacker_speed, defender_speed:
      start calc:
            return attacker_speed >= defender_speed
done

Universe:
      Character_HP is 10
      Character_Attack is 10
      Character_Defense is 5
      Character_Speed is 5
```

```
      Ogre_HP is 10
      Ogre_Attack is 20
      Ogre_Defense is 5
      Ogre_Speed is 2
      weapon_room_seen is false

start dungeon_entrance:
      do show to "You are at the entrance of the dungeon. There are three doors
in front of you. These are labelled 1, 2, and \"BOSS\". Enter the label of the
door you want to go through. Choose wisely: "
      choice is input
      if choice = 1 and not weapon_room_seen
            (next weapon_room)
      else if choice = 1 and weapon_room_seen
            (do show to "You already went there!"
            next dungeon_entrance)
      else if choice = 2
            (next trap_room)
      else if choice = "BOSS"
            (next boss_room_intro)
      else
            (do show to "That's not a valid door label! Try again."
            next dungeon_entrance)

weapon_room:
      do show to "You found a shiny sword! +5 attack and -1 speed. You return
to the previous room."
      Character_Attack is Character_Attack + 5
      Character_Speed is Character_Speed - 1
      weapon_room_seen is true
      next dungeon_entrance

trap_room:
      do show to "The door closes behind you and never opens again. SADNESS.
THE END."

boss_room_intro:
      do show to "An ogre appeared!"
      next boss_room

boss_room:
      if Character_HP < 1
            (next character_death)
      if Ogre_HP < 1
            (next ogre_death)

      do show to "Will you try to hit or dodge? Type \"hit\" or \"dodge and try
hitting\"."
```

```
	battle_move is input

	if battle_move = “hit”
		(dam is do calc_damage_done to Character_Attack, Ogre_Defense
		Ogre_HP is Ogre_HP - dam
		do show to “You hit the ogre inflicting “ + dam + “ damage!”
		dam is do calc_damage_done to Ogre_Attack, Character_Defense
		Character_HP is Character_HP - dam
		do show to “The ogre hit you inflicting “ + dam + “ damage!”)

	else if battle_move = “dodge and try hitting”
		(success is do does_hit_land Ogre_Speed, Character_Speed

		if success
			(dam is do calc_damage_done to Ogre_Attack, Character_Defense
			Character_HP is Character_HP - dam
			do show to “The ogre hit you inflicting “ + dam + “ damage!”)

		else
			(dam is do calc_damage_done to Character_Attack, Ogre_Defense
			Ogre_HP is Ogre_HP - dam
			do show to “You dodged and hit the ogre for “ + dam + “
		damage!”)
		)

	else
		(dam is do calc_damage_done to Ogre_Attack, Character_Defense
		Character_HP is Character_HP - dam
		do show to “The ogre hit you inflicting “ + dam + “ damage!”)

	next boss_room

character_death:
	do show to “You died. Sadness. THE END.”

ogre_death:
	do show to “The ogre died. You win!!! THE END.”
```