SMAP Project Proposal


```
Tushar  Arora    -  ta2673
Andrew  Magid    -  aam2302
Swapnil Paliwal  -  sp3911
Emily   Sillars  -  ems2331
```


1. Description/Overview

Sections: Motivation, Key Aspects of language

This is a C-like language that is statically typed with the ability to have dynamically typed arrays. We call these arrays polymorphic lists (see section 2.12 below) and are a foundational building block of the language. Strings are implemented using monotypic character lists and allow for very easy string manipulation. This allows text-based games to be created very easily.

We introduce the probability structure that allows the code to execute nondeterministically and efficiently facilitate randomized gameplay. We also introduce function counters which will enable a function only to execute a certain number of times. This allows for the simplification of code logic and has security benefits for one-time run functions.

SMAP is a string manipulation language with probability-based primitives designed to make ASCII animation and command line games easier to code.

2. Syntax/Language Details

Primitives, structures, premade types, var declaration, functions, control flow, ex. of library functions, keywords, comments, operators

2.1 Data types

Type	Description
int	integer
char	character

boolean	true, false. syntactic sugar for 1 or 0
string	string literal, dynamically sized
prob	Constant integer between 0 and 100 representing probability as a percentage
struct <Name> {<type, fieldName>*};	User-defined type containing fields
void F (<type><argName>...)	A function that takes parameters with no return type
<type> F (<type><argName>...)	A function that takes parameters and returns something
List <type> [<optionalLabel>:<value>...]	A dynamic monotypic list
List [<optionalLabel>:<value>...]	A dynamic polytypic list
null	Uninitialized type

2.2 Structs

Structs are defined in the same way as they are in C: they allow for a programmer-defined data type which is a fixed memory size that contains primitives and/or other structs.

```
struct player{
     int health = 100;
     int xp;
     string name;
};
```

2.3 String-builder syntax

Strings can be defined with quotes, “aString”, but can also be defined with the following square bracket syntax, reminiscent of printf in C.

Bracket Syntax	**Description**
[“%e*”, e*]	A string in quotes can be mixed with any number of format characters %e (and their corresponding values e after the comma) to make a new string.

Format character	**meaning**

%s	Insert a string
%d	Insert an int in decimal
%c	Insert a char

Example of equivalent string builder syntax:

```
// equivalent statements
string s = “Wow Sam, 5 points!”;
string s = [“Wow %s, %d points!”, “Sam”, 5];
```

2.4 Operators

Operator/type	prob	int	char	bool	string	Description
+, -, *, /		✓	✓		Only + for concatenation	Standard arithmetic
&, |, ^, ~, <<, >>		✓	✓			Bitwise operators
>, <, >=, <=	✓	✓	✓			Comparison
==, /=	✓	✓	✓	✓	(deep comparison)	Equality

2.5 Operations on Strings, Monotypic Lists, and Polytypic Lists

2.5.1 String Operators

Strings are a case of our more general polymorphic list type. Specifically, they are monotypic lists of characters. We use syntactic sugar to hide these details from the user, resulting in a separate string type that is easy to work with.

String Operator	Description	Mini Example
==	Deep comparison	`“howdy” == “howdy” yields 1`
+	Concatenation	`“Hi” + “hello” yields “Hihello”`
<< :	Left shift, with designated filler	`“..*..” << 2 : ‘E’ yields`

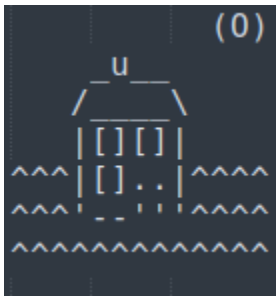
	char	“*..EE”
>> :	Right shift, with designated filler char	“..*..” **>>** 2 **:** ‘.’ yields “....*”
^^ : , .. , ;	Overlap, with a list of designated “clear” chars	“---O_O---” **^^** “.........” **:** “-” yields “...O_O...”
<Type>	Cast from one type to another	<int>”123” + 10 yields 133

Example string manipulation program:

```
list string house = ["**************",
                     "****_u__*****",
                     "***/____\****",
                     "***|[][]|****",
                     "***|[]..|****",
                     "***'--'''****",
                     "**************"];
list string bg = ["            (0)",
                  "              ",
                  "              ",
                  "              ",
                  "^^^^^^^^^^^^^^",
                  "^^^^^^^^^^^^^^",
                  "^^^^^^^^^^^^^^"];

for(int i = 0; i < house.size(), i++){
    //overlap house on top of background,
    //where '*' is treated as a clear tile
    string row = house[i]^^bg[i]:"*";
    print("%s\n", row);
}
```

Output:



Note that the string overlap operator de-sugars to a function on a monotypic list of characters:

```
list char overlap (list char top, list char bottom, list char clearTiles){
    list char overlay = [];
    if(top.size != bottom.size)
       return overlay;
    for (int i = 0; i < top.size; i++){
       if (top[i] elem clearTiles)
             row.append(bottom[i]);
       else
             row.append(top[i])
    }
    return overlay;
}
// house[i]^^bg[i]:"*" is equivalent to overlap(house, top, ['*'])
// overlap could be made more concise by using a zipwith std lib function
```

2.5.2 Polymorphic List Operators

List Operator	Description	Monotypic	Polytypic
==	Deep comparison	✓	
+	Concatenation	✓	✓
<< :	Left shift, with designated filler value	✓	✓
>> :	Right shift, with designated filler value	✓	✓
elem	Returns whether the value is an element of the list	✓	

2.6 Built in Functions

The idea is to enable users to achieve standard functionalities without having to re-write the code again. Further, the aim is to equip developers with efficient methods, thereby writing optimized codes—our language features following state-of-the-art methods.

Function	Description
abs()	Returns absolute value of an integer
allElem()	Prints all elements of the list to console
ascii()	Returns ascii value of input

canPerform()	Returns a boolean true if two input types are compatible
ceil()	Returns a ceiling value of a number
concat()	Returns concatenation of two strings
currElemType()	Returns the current element type of the list
funcCount()	Returns an int specifying execution count of a given function
floor()	Returns floor value of a number
isCompDiv()	Returns a boolean true if two numbers are divisible
inRange()	Returns an int in the range
isFuncExec()	Returns a boolean true if a function can be executed now
isEmpty()	Returns a boolean true if a string or a list is empty
isSortable()	Returns a boolean true if the list contains either a string or an int
len()	Returns the length of a string
listLen()	Returns the length of the list
ofType()	Returns the type of an element
print()	Prints object to console
probCount()	Returns the sum of all the probabilities
probDist()	Creates a probability distribution of a given type
rand()	Returns a random number in range [0,2^31)
rangeList()	Returns the range of the list
randGraffiti()	Returns a random string sequence forming a graffiti
randomize()	Returns a randomized list
regex()	Returns a boolean of true if regex matches

2.7 Comment

Comment Type	Description
//	A single line comment
/* */	A multi line comment

2.8 Keywords

“prob” keyword is applied to a type that creates a probabilistic version of that type. When a probabilistic type is evaluated, it returns one out of a group of possible values. Using the prob keyword requires the following syntax:

`prob <type> <varid> = { <prob>:<type>,* }`	Where the sum of the prob primitives inside the block must add up to 100.

```
prob char eyeball = {40:'O', // big eye
                     25:'^', // happy eye
                     15:'=', // closed eye
                     20:'*'  // star eye };
while(...){
      char eye = eyeball;    //value changes each loop execution
      print(["%c_%c\n", eye, eye]);
}
// each execution of the while loop, one of four
// faces will be printed with following frequency:
// 40% of the time: "O_O"
// 25% of the time: "^_^"
// 15% of the time: "=_="
// 20% of the time: "*_*"
```

We can use this to probabilistically call functions too.

```
void callBoss(){
     //do boss stuff
}

void callMinion(){
     //do minion stuff
}

prob void enemy = {   50: callBoss(),
                50: callMinion()};
enemy();
```

2.9 Control Flow

- `if, elif, else`
- `switch, case`
- `continue, break`
- `for loops, while loops`

2.10 Basic IO

Basic IO will be included using C's -ncurses library

- Start screen
- Enable/disable typing echo
- Get line
- Get char
- Get char (immediately, do not wait for user to hit enter)

2.11 Functions

Functions are defined in the following order: return type, function name, arguments encased in parentheses, curly braces.

```
int gimmeNum(int num){
     return num
}
```

2.11.1 Function Counters

Function counters are an optional parameter that allow a function to be run only a specified number of times from the start of program execution.

The functions follow an execute and kill strategy, which runs once or a given number of times and exits the application. Further, once the function has exhausted the count, no action will occur.

```
int 1 gimmeNum(int num){

     return num
}
```

A use case is to use a function as an init method which should only be called once for the entire lifetime of the program. For example:

```
void 1 generateEnemy() {
EnemyCurrent.enemyCount = inRange(75,90)
}

while(current.level == 5){
     generateEnemy()
     //Some other stuff
}
```

2.12 Polymorphic Lists

Pablo Picasso is often misquoted, “good artists copy, but great artists steal”. Polymorphic lists are a construct designed to be a compromise between the way many statically and dynamically typed languages store data by pulling from the designs of both.

In the static case, let’s consider how a C program would define an array.

```c
//c code snippet
int arr[10]; //statically sized int array of size 10
arr[0] = 1; //legal
//arr[1] = 'a'; //illegal because element must be of type 'int'

int *arr = (int *) malloc(sizeof(int)) * 10); /* dynamically sized
array. we can continue to increase the size by malloc-ing*/
```

We can see that in either the static or dynamic C array definition, the entire array must contain one type only, that is of type `int`.

Dynamically typed languages allow for more flexibility.

```
#Python snippet
a = [1, 'a', True, [1, 2, "hello world"]] # legal
```

We can store any type inside Pythonic lists, including other nested lists! The biggest problem is the lack of type safety, leading to a potential runtime error if a function is called on an incompatible type within the list.

Our list allows for a generalizable way to store data like a Python list, but with the option of C-like type safety. Let's take a look at how we define this in our language and the additional features we include that allow for data to be accessible in a similar way to an array, a class, or both.

```
// polytypic list
int y = 100;
list example_list = [x_cord: 50, y_cord: 100, "hello world", levels:
[one: 1, two: 2, three: 3]];
example_list.append('z');
```

What's going on here? We define our list with the `list` keyword and we initialize it using square brackets. Similarly to Python, we can include data of any type (even developer-defined structs and other lists!) hence the name, polytypic.

There are two ways to access data from a list: by index or by label. By index, a programmer could type `example_list[2]` which would return `"hello world"`. By label, a programmer could type `example_list.y_cord` and the program would return `100`. Similarly, `example_list.levels.three` would return `3`. These can also be combined: `example_list.levels[2]` would also return `3`. Labels are like keys and must be unique within a list (like a Python dictionary or a C struct).

Interestingly, because we can hold *any* data, we can include function calls. We can include functions inside lists and be able to pull them out to call on anything else we would like. We use the `fn` keyword to denote that the following is a function name and not a variable name.

```
string foo(string input){
     return input;
```

```
}

list another_list = [“test”];
another_list.append(foo: fn foo);

// the following calls are equivalent
foo(“test”);
foo(another_list[0]);
another_list.foo(another_list[0]);
another_list[1](another_list[0]);
```

So far we looked at polytypic lists that can hold any data type. However, we often are manipulating arrays of only integers, chars, or other programmer defined types. **Monotypic** lists only allow values of a given type to be added to the list.

```
// monotypic list example
int x = 2; int y = 3;
list int integer_list = [1, two: x, y, four: 4];
/* integer_list.append("this is an illegal operation and will be
caught at compile time"); */

list list int list_of_integer_lists = [[1,2,3], [4,5,6], [1,2,3]]; /*
legal */
list list char illegal_list = [[‘a’,’b’, ’c’], [‘d’, ’e’, ’f’],
[1,2,3]]; // not legal! compile time error
```

The `list` keyword is followed by the `int` keyword which denotes a monotypic list and will only allow elements of type `int`. Most importantly, this means that type safety is ensured at compile time, not runtime! Monotypic lists are therefore not required, but give the developer the tools for type safety which can prevent many errors. The type need not only be primitives. A programmer-defined struct or nested list structure are all fair game.

A natural question to ask at this point is what will happen with code that looks like this?

```
//polytypic list declaration and initialization
list example = [];

if (getKey() == LEFT_ARROW){
    example.append(13);
}
```

```
else{
    example.append("this is a string");
}

int x = example[0]; //this is undefined behavior
```

What if the user inputs any key other than the left arrow key? This behavior is undefined and will result in a runtime error. There are other examples of runtime errors. What if we append a label and a value and then reference it by the wrong label? What if we append a label that already exists within the list? Dynamic typing begets runtime errors and there is no way to get around this.

We provide the tools for type safety and it is up to the developer to make use of them. This includes keeping track of the labels appended and making sure they are all unique. However, if we define a monotypic list such as `list int example = [];` we will ensure that a line like `example.append("this is a string")` will result in a compile time error rather than a runtime error which is preferable.

2.12 ncurses Library

- Ncurses is a library that acts like a wrapper to improve the terminal capabilities
- It is mostly used to create a text based User Interface (UI) to help users create a GUI-like application software that runs under a terminal emulator.
- The packages required to install the library are -
    - `libncurses5-dev` : Developer's libraries for ncurses
    - `libncursesw5-dev` : Developer's libraries for ncursesw
- The API that we intend to use for our program is -
    -

```
WINDOW* win = initscr;  // this enables the curses mode

keypad(); // It enables the reading of function keys like F1, F2,
//arrow keys etc.

raw(); //used to disable line buffering - usually the terminal
//driver buffers the characters a user types until a new line
//or carriage return is encountered

cbreak(); //used to disable line buffering (same thing as raw())

halfdelay(3); //similar to cbreak() and so the characters typed are
```

```
//immediately available to the user
```

```
curs_set(0)     // this API is used to hide the cursor

printw();       // same thing as printf in  C

refresh();      // print everything from the printw calls to the
                //console

int name = getch();  //getting the character

endwin();  // this exits the curses mode

Some Keypad Codes:
KEY_UP
KEY_LEFT
KEY_RIGHT
```

2.12 Linking an external library

- External libraries are not usually known by ocaml build so they have to be written at the root of the project in the myocamlbuild.ml file.
- Example of adding external libraries *ablgl* and *lablglut to the project.*

```
open Ocamlbuild_plugin
open Command

let () =
  dispatch begin function
  | After_rules ->
     ocaml_lib ~extern:true ~dir:"+lablGL" "lablgl";
     ocaml_lib ~extern:true ~dir:"+lablGL" "lablglut";
  | _ -> ()
  end
```

References

https://www2.ocaml.org/learn/tutorials/ocamlbuild/Using_an_external_library.html

https://tldp.org/HOWTO/NCURSES-Programming-HOWTO/