Strlang tutorial

The strlang language features a C-like structure wrapped in a minimalistic syntax. Its focus is on making manipulation of text strings easy and efficient.

2.1 Basics

strlang has three types of variables: strings (`$`), numbers (`#`) and maps (`%[k;v]`). Strings are used to store text, number for integer values, and maps to hold key-value pairs. Keys and values in a map may be strings or numbers.

Comments

Comments are notes by the programmer in the code. They are ignored by the compiler. Comments begin with `//` and run through the end of the line. For readability, comments will be shown in red, code in green.

```
... // this is a comment and will be ignored
```

Declarations

Variables must be declared before they are used. Declarations specify the variable's type.

```
$ str1;             // declaring a string called 'str'
# num1;             // declaring a number called 'num'
%[$;#] city_temps;  // declaring a map with string keys and number values
```

Assignment

The most common operation on variables is to assign them the value of some expression using the assignment operator (`<-`). The value of the expression on the right side is stored in the variable on the left side. The variable and expression must have the same type (i.e. both string or number or...)

```
str2 <- str1;       // value of the variable str1 stored in str2
num2 <- num1;       // value of num1 stored in num2
num2 <- str1;       // ERROR: this won't work (different variable types)
```

Literals

Strings and numbers that are known may be used directly in place of variables.

```
num1 <- 0;          // num1 assigned the value 0
str1 <- "hello";    // str1 assigned the value "hello"
```

Operators

Variables and constants can be transformed using unary and binary operators. Each operator has rules regarding the types of its inputs and outputs. Here are some of the basic operators:

```
num1 <- num1 + 1 * 3 - 7 / 5;     // standard arithmetic operations
str1 <- str1 + " world";          // concatenating two strings
str1 <- "hello world" - 5;        // removing the last 5 characters
str1 <- "hello" / "l";            // search one string for the other
num1 <- city_temps[str1];         // get value associated with key str1
city_temps["Ojai"] <- 44;         // associate value 44 with key 'Ojai'
```

Expressions

An expression is simply some operation that can be executed and evaluated. Literals, variable names and binary and unary operations are expressions, among others.

```
3*7                 // arithmetic expression
"hello " + str1     // string expression
num1                // arithmetic expression
```

Functions

Functions are the logical programming unit for strlang. Each function is a semi-autonomous chunk of code which receives certain input at the beginning from whomever invokes it, and when it finishes, it may choose to send back certain output.

The function header defines the name the function is invoked by, as well as the types and numbers of the inputs to the function, and how they will referred to inside the function. The header also defines the type of value the function will output. The function's body contains the actual expressions and statements executed when the function is invoked.

Syntax

```
name(type input1; type input2; ... ) -> output_type // header
{
      // body
}
```

Functions that do not take input from their caller, or do not return output use `^` in place of those parts of the header.

```
empty_func(^) -> ^ // no input, no output
...
no_input_func(^) -> # // no input, number output
...
```

To get values out of a function, a return statement is used. The statement consists of the return operator (`->`) followed by an expression (or no expression if the function is not defined as outputting a value).

```
add(#a; #b) -> #    // add takes in two numbers, outputs a number
{
      -> a + b;     // output sum of num1 and num2 to the caller
}
```

Invoking Functions

Defining functions isn't much use if they can't invoked. Invoking a function requires specifying the values or variables that will be given as input. These values are mapped to the input variables defined in the function header in the order given. For functions that output a value, the result can be assigned using the standard assignment operator.

```
num1 <- add(num1; num2);   // function 'add' receives num1 and num2
                           // as input and its output is saves in num
                           // in this case, 'a' gets num1, 'b' gets num2
empty_func();              // a function with no inputs
```

Every program must define a function named 'main' which takes no inputs and outputs a number. This function is where program execution begins.

Built-in functions

A few functions are built in to the language to make input/output and other lower-level processes easier. The most important are:

```
read(^) -> $        // read in a line of text (by default from
                    // the keyboard) and return it as a string.
write($msg) -> ^    // output the string 'msg' (by default to the screen)
```

2.2 Example

Hello World code

```
main(^) -> #        // function header
{                   // mark beginning of function body
      $ name;       // variable declaration

      write("Enter your name: "); // call to function, with parameter
      name <- read(); // call to function, assigning result to 'name'

      write("Hello " + name + "!\n");  // call to another function
      // expression is evaluated before function is called

      -> 0;  // leaving the function, giving back the value '0'
}            // end of function body
```

Running the program

The compiler is invoked using the strlang executable. The compiler expects the name of the source file and the name to save the executable output as. Saving the above file to hello.str, the process is:

```
$ ./strlang
usage:
./strlang
        -a file.str                (dump AST of source)
        -c file.str [file.cpp]     (compile to C++)
        -e file.str file.exe       (compile to executable)
        -h                         (display this message)
        -v                         (display version number)

$ ./strlang -e hello.str hello
$ ./hello
Enter your name: Dara H
Hello Dara H!
$
```

2.3 Other language features

Conditionals

strlang includes the standard if-else construct. The test expression, which must be of type number, is evaluated first. If the result is non-zero, the block directly after the expression is executed. If not, the block associated with the else-clause is executed. The else clause is optional.

```
[num1 < 0]                        // test clause: num1 < 0 is test expr
{ write("num1 is negative"); }    // if num1 < 0, this executes
![]                               // otherwise (else)...
{ write("num1 is nonnegative"); }// this block executes
```

Loops

strlang's loops behave like while loops in C. The expression in the loop header is evaluated. If the result is non-zero, the loop block is executed. If not, control jumps to the first statement after the loop block. When the last statement of the loop block executes, the program returns to the loop header.

```
<num1>                     // loop header: num1 (!= 0) is test expr
{
       write(to_string(num1) +    // print message...
              "bottles of beer on the wall.\n");
       num1 <- num1 - 1;          // decrement loop counter
}      // end of loop
```

Operators

strlang has many operators for manipulating strings, and several for manipulating maps. See the Language Reference Manual for further details.

2.4 Invoking the compiler

The strlang has 3 modes: one for generating code, one for generating an executable and one for displaying a program's internal representation.

In compilation mode, the compiler will either display or save to a file the C++ code it has generated for the given program.

```
$ ./strlang -c hello.str hello.cpp
```

In executable mode, the compiler will first generate C++ code and then invoke the C++ compiler to build an executable that can be run directly.

```
$ ./strlang -e hello.str hello
```

In display mode, the compiler will dump to the screen the abstract syntax tree representation of the program after parsing it.

```
$ ./strlang -a hello.str
```

2.5 Installation

strlang assumes a UNIX-like environment. It requires make, bash, diff and several other standard UNIX utilities to build and execute. It also requires a C++ compiler, including the C++ standard library (STL).

1). Obtain and install the ocaml 3.1 distribution, hosted at http://caml.inria.fr/. The ocaml binaries must be added to the $PATH environment variable.

2). Obtain and install the pcre regular expression library, located at http://www.pcre.org/. If building from source, make sure to enable the C++ wrapper: (./configure --enable-cpp). The libraries and headers need to be installed in directories in the standard search path (/usr/local/include and /usr/local/lib are usually fine).

3). Obtain and decompress the strlang source distribution.

4). Adjust strlang.ml's cpp_compiler variable if you are using a C++ compiler other than g++. If PCRE is not installed in the default search path, you may need to change the pcre_* variables as well.

5). From the top-level of the source distribution, type 'make'.