IRIs

UNIs:

xl2784
st3174
hg2498
pt2508

Introduction

Inspired by the powerful macOS tool Alfred, we decide to design a workflow description language called IRIs. IRIs not only stands for the flower but stands for the reverse of the Apple so-called smart assistant Siri.

IRIs is used to describe and generate a workflow that with the input of the user, completes a sequence of tasks. IRIs will mainly focus on the macOS, and some of its code can also work on Linux distributions.

Ingredients

Basic Structure (Not Syntax Tree)

IRIs is mainly made up of two key structures: the expression and the workflow.

Expression

The expression is similar to what other awful or terrific languages which can be defined as a single instruction. But it is slightly different that expressions in IRIs is connected by a flow symbol (or pipe, whatever) `|` and the value of an expression will flow down to the next line if connected so that you do not need to use ~~silly~~ temporary variables to transmit information and will also make your program subtle and neat.

For example, in C++, if we want to receive an input from a keyboard and add 1 to that input, what we needto do is:

```
int a;
a << stdin;
a = a + 1;
cout << a << endl;
```

But in IRIs, it will be:

```
input()
|
```

```
+ 1
|
ShowResult()
```

We have to admit that the C++ way has many advantages, especially in big programming projects. However, what most of us really need is a simple way to handle an "input -> process -> output" flow. In most circumstances, the process could be not much complicated that a "plus 1" operation.

So here we will introduce the second principal structure of our language - the workflow.

Workflow

A workflow just likes the function part in other imperative languages. It can accept an argument, process a sequence of instructions and finally, give an output. What is more, a workflow can read an input from the result or the output from an expression or just another workflow if they are connected by the flow symbol. It just likes the pipe in the Linux shell. Also, a workflow can be called recursively to complete the repeat process of other imperative languages.

A workflow should start with the keyword `HeySiri` followed by a workflow name and end with the keyword `GoodbyeSiri`.

Data Types

- **num**: Integer or floating number
- **text**: just likes the string
- **list**: a list of option that can be chosen
- **dictionary**: a set of key-value pairs

Key Words

- `HeySiri`:
  Define a workflow
- `GoodbyeSiri`:
  Mark the end of a workflow
- `repeat`:
  Mark the start of a loop
- `RepeatIndex`:
  A repeat counter. It will count the number of times a loop runs. If there is no appendix number behind, it stands for the innermost loop; if there is a number appended, such as RepeatIndex2, it stands for the second layer of the nested loop.
- `end`:
  Mark the end of a loop
- `if/elif/fi`:
  Those set of keywords are used to perform a condition expression just as a Bash script does.
- `$`:
  A placeholder for output. Explain below.

Basic Workflows

- `input()`:
  Just like the input function of other languages, it accepts the input from a user and will pass it to others by the flow symbol.
- `list[]`:
  The `list[]` keyword defines a basic data type in IRIs. This data type is an indispensible part of the IRIs. A list is not only an array like data type that stores a set of data, but also connect with the user interaction tightly.
- `choose{}`:
  It likes the switch-case expression of other languages, but slightly different. In other languages, the switch-case expression likes another form of an if expression (although the low-level implementation is different, I know). In IRIs, `choose{}` is more of an interactive instruction. It accepts an input of a list, and prompts the items from the list to the user, waits user choosing an item (or an option), then output flow of data.
- `ShowResult()`:
  Like the print function of other languages. It accepts input from a flow, too. And inside the parenthesis, it uses the keyword `$` as a placeholder of the input.

Sample Code

Caclculate the tip

```
HeySiri CalcTip()

input()
|
bill

list[0.12, 0.15, 0.18, 0.20]
|
choose{
    1:
        0.12
    2:
        0.15
    3:
        0.18
    4:
        0.20
}
|
rate
|
* bill
|
tip
|
+ bill
|
total
```

```
ShowResult("Your tip is ", tip, " and your total is ", total)

GoodbyeSiri
```

Explanation: At first, the program starts with a `HeySiri` keywords to define a workflow. There is no concept of the main function as the entry of the program. The program will begin at the very first line.

Then, there is an `input()` workflow connecting to a variable `bill` which means "read the users input and put it in the variable."

Then, there is a list within the `[]` symbol. This is one of the basic data type called list, and it will flow into a `choose expression` with a pair of `{}`. These few lines will prompt the list and ask the user to choose one of the items in the list, then match the value within the `{}` and pass it to the next line. Here, the number `1/2/3/4` stands for the index of the items in the list, and the number behind the `:` stands for the result of the `choose expression`.

Notice that the result of a `choose expression` does not have to identify to the input list. For example, the input list could be `["large", "medium", "little"]` and the result of the `choose expression` could be `10, 5 and 1`.

After the choice, there is a long flow a calculate. In IRIs, there is no `=` expression, the operator `=` only means *equal to*. All of the assignments will be done by using the flow symbol `|`. Line 19 to 28 of code above means - "assign the result to variable `rate`, and then multiply it by the variable `bill` and assign the result to variable `tip`(but the value of the rate will not change), then add the value of `tip` and `bill`, which we get from the input, and assign the result to variable `total`. Similarly, the value of `tip` will not change."

Then, the `ShowResult()` will show the content inside the parenthesis and the workflow will terminate after the `Goodbye Siri` keyword.

GCD

```
HeySiri GCD()

intput()
|
a

input()
|
b

if (a != b)
    if (a > b)
        a - b
        |
        a
        GCD(a, b)
```

```
    elif (a < b)
        b - a
        |
        b
        GCD(a, b)
    fi
fi

GoodbyeSiri
```

99-Bottles-of-Beer

```
HeySiri _99_Bottles_of_Beer()

input()
|
repeat

99 - RepeatIndex + 1
|
left
|
ShowResult($, "bottles of beer on the wall, ", $, " bottles of beer.")

left - 1
|
ShowResult("Take one down and pass it around, ", $, " bottles of beer on the
wall.")

end

ShowResult("No more bottles of beer on the wall, no more bottles of beer.")
ShowResult("Go to the store and buy some more, 99 bottles of beer on the wall.")

GoodbyeSiri
```