F#Snake Language Specification

Markus Feng

2018-12-09

1 Introduction

The goal of this language is to create a F# implementation of a general purpose interpreted language heavily based on the Python programming language, yet with restricted functionality on the language. This language would have many features of existing general purpose imperative programming languages, including function definitions, function calls, lexical scoping, variables, control flow, operators, and types including `int`, `string`, `bool`, and `list`.

This problem needs its own programming language because it defines a specific set of restricted functionality. This means that it is easier to check the security and the correctness of the programs written in this language. The hope is that this language is a basic building block such that simple programs can be written in this language, and that features can be added to this language as needed to make it easier to write specific programs, while not losing its benefit of simplicity.

A demo of the project can be found at:
`http://assisstion.github.io/projects/cardgamelang/`.

2 Design Principles

The guiding design principle behind this design is powerful but simple and familiar. This would more likely ring true to programmers with previous experience in Python, who would find the syntax and semantics similar to that language. This language is powerful enough to be Turing complete (assuming infinite time and memory), as demonstrated by the fact that it is possible to implement a Rule 110 elementary cellular automata in this language.

3 Examples

The following example (which can be run by `dotnet run example-1.cgr`) is an implementation of a program to print out prime numbers:

```
isprime(n):
    if n < 2:
        return false
    for i in range(2, sqrt(n) + 1):
        if n % i == 0:
            return false
    return true

main():
```

```
    print("Print all prime numbers less than 100:")
    for i in range(100):
        if isprime(i):
            print(i)
    return 0
```

The following example (which can be run by `dotnet run example-2.cgr`) is an implementation of the "Fizz-Buzz" program:

```
main():
    print("For numbers from 1 to 50, inclusive")
    print("Print Fizz for numbers divisble by 3")
    print("Print Buzz for numbers divisible by 5")
    for i in range(1,51):
        let b1 = i % 3 == 0
        let b2 = i % 5 == 0
        if b1 | b2:
            let s = concat(str(i), ":")
            if b1:
                s = concat(s, " Fizz")
            if b2:
                s = concat(s, " Buzz")
            print(s)
    return 0
```

The following more complicated example (which can be run by `dotnet run example-3.cgr`) is an implementation of mergesort:

```
split(l):
    let splitted = []
    let c = len(l)
    pushf(splitted, sublist(l, c/2))
    pushf(splitted, sublist(l, 0, c/2))
    return splitted

merge(a, b):
    if len(a) == 0:
        return b
    elif len(b) == 0:
        return a
    let av = popf(a)
    let bv = popf(b)
    if av <= bv:
        pushf(b, bv)
        let nl = merge(a, b)
        pushf(nl, av)
        return nl
    else:
        pushf(a, av)
        let nl = merge(a, b)
        pushf(nl, bv)
        return nl

mergesort(l):
    if len(l) <= 1:
        return l
```

```
    let splitted = split(l)
    let left = mergesort(splitted[0])
    let right = mergesort(splitted[1])
    return merge(left, right)

randlist(length, maxval):
    let l = []
    for i in range(length):
        pushf(l, rand(maxval))
    return l

main():
    print("Merge sorting a fixed list:")
    let arr = [3,1,4,1,5,9,2,6]
    print("Original:")
    print(arr)
    print("Sorted:")
    print(mergesort(arr))
    print("Merge sorting a random list:")
    let arr2 = randlist(20, 1000)
    print("Original:")
    print(arr2)
    print("Sorted:")
    print(mergesort(arr2))
    return 0
```

Other sample programs can be found in the samples subdirectory, and can be run by dotnet run samples/<file>.cgr. Most of these are programs to test that my language is working the way that it is intended.

4 Language Concepts

The user needs to understand basic imperative programming concepts to use this language. Because the structure of this language is similar to any other type of imperative language such as python or java, the language has variables and functions, control flow such as if/else/for/while, recursion, and types such as int, boolean, string, list. In this case, variables act as primitives, while methods, control flow, and complex objects act as combining forms. In addition, there is a concept of a scope, similar to that of a class. A scope contains a set of properties and functions, and can be initialized by calling a function of the scope name with no arguments. Note that like many other general purpose programming languages, there is usually multiple ways to solve a problem in this language, so not understanding some of the previous concepts does not prevent a user from being able to use this language.

5 Syntax

The goal is for the language to have a python-like whitespace based syntax. Therefore, indentation matters in such a language, so we use a [<expr>+1] to indicate that everything inside <expr¿ (other than newlines) is indented one additional time. In addition, any non-quoted "#" character begins a comment, so that character and all successive characters of the same line are ignored. The BNF of the currently implemented grammar looks like the following:

```
<program> ::= <definition> <newlines> <definition> | <definition>
<newlines> ::= \n | <newlines> \n
```

```
<definition> ::= <fun_def> | <var_decl> | <named_scope>
<named_scope> ::= <id> : <newlines> [<program>+1]
<var_decl> ::= <id> | <assignment>

<fun_def> ::= <fun_header> <stmt_block>
<stmt_block> ::= : <newlines> [<expr_body>+1]
<fun_header> ::= <id> (<fun_args>)
<fun_args> ::= <id> | <fun_args>, <id>
<expr_body> ::= <stmt> <newlines> <stmt> | <stmt>

;Statements
<stmt> ::= <fun_call>
    | <prop_fun_call>
    | <complex_assignment>
    | <let_stmt>
    | <return_stmt>
    | <if_else_block>
    | <while_block>
    | <for_block>

<complex_assignment> ::= <lvalue> <complex_eq> <expr>
<complex_eq> ::= <op>= | =
<let_stmt> ::= let <assignment>
<assignment> ::= <id> = <expr>
<return_stmt> ::= return <expr>
<if_else_block> ::= <if_part> | <if_part> <else_block>
<else_block> ::= <else_part> | <elif_block> <else_part>
<elif_block> ::= <elif_part> | <elif_part> <elif_block>
<if_part> ::= if <expr> <stmt_block>
<elif_part> ::= elif <expr> <stmt_block>
<else_part> :: else <stmt_block>
<while_block> ::= while <expr> <stmt_block>
<for_block> ::= for <id> in <expr> <stmt_block>

;Expressions
<expr> ::= <dot_expr> | <dot_expr> <op> <expr>
<dot_expr> ::= <array_expr> | <prop_acc> | <prop_fun_call>
<array_expr> ::= <consuming_expr> | <array_acc>
<array_acc> ::= <array_expr>[<expr>]
<consuming_expr> ::= <unary_expr>
    | <fun_call>
    | <this_literal>
    | <bool_literal>
    | <id>
    | <num_literal>
    | <parens_expr>
    | <list_literal>

;Function calls and properties
<fun_call> ::= <id> () | <id> (<expr_args>)
<prop_fun_call> ::= <array_expr> . <fun_call>
<prop_acc> ::= <array_expr> . <id>
<expr_args> ::= <expr>, <expr_args> | <expr>

<lvalue> ::= <id>
    | <array_acc>
    | <prop_acc>

;Literals
```

```
<string_literal> ::= "" | "<chars>"
<num_literal> ::= <num_literal> <num> | <num>
<bool_literal> ::= true | false
<this_literal> ::= this
<parens_expr> ::= (<expr>)
<list_literal> ::= [] | [<expr_args>]

;Identifiers
<id> ::= <id> <idchar> | <idchar>
<idchar> ::= <loweralpha> | <upperalpha> | <num> | <underscore>
<loweralpha> ::= {a, b, ..., z}
<upperalpha> ::= {A, B, ..., Z}
<underscore> ::= _
<num> ::= num in {0, 1, ..., 9}

;Operators
<unary_expr> ::= <unary_op> <array_expr>
<unary_op> ::= {-, +, !}
<op> ::= {+, -, *, /, %, ==, !=, <=, >=, <, >, &, |}

;Strings
<stringchars> ::= <stringchar> <stringchars> | <stringchar>
<stringchar> ::= \ <char> | <nonquote>
<char> ::= {Unicode characters}
<nonquote> ::= {Unicode character that are not a double quote}
```

6 Semantics

The program is interpreted in order, line-by-line, with multi-line clusters forming a scope. Due to the inspiration from a `python`-like whitespace based syntax, indentation is used to define blocks, whereas line breaks are used to separate statements.

The interpreter will throw a syntax error if the result of the parsing is invalid, indicating the line number that the interpreter fails at. Currently, the interpreter does not support multi-line statements, so the same statement must be kept on a single line for parsing to succeed.

6.1 Primitive Values

Syntax	Abstract Syntax	Type	Meaning
n	NumLiteral of int	ValInt	A primitive in the language that represents a 32-bit signed integer using the F# Int32 type.
“str”	StringLiteral of string	ValString	A primitive in the language that represents a string using the F# String type.
true/false	BoolLiteral of bool	ValBool	A primitive in the language that represents a boolean using the F# bool type.
id	Identifier of string	ValReference	A primitive in the language that represents an identifier that can be assigned to and accessed later.
this	ThisLiteral	ValReference	A primitive in the language that, when called in an instance function, gives a reference to that instance, and fails otherwise.

6.2 Control flow

Syntax	Abstract Syntax	Meaning
if *pred*: stmts (elif *pred*: stmts)* (else: stmts)?	IfElseStmt of (Expr * Stmt list) * ((Expr * Stmt list) list) * Stmt list option	If the predicate is true, execute the first block of statements. Otherwise, successively go through each of the elif predicates. If any of those are true, execute that block of statements and stop. Finally if there is an else block and none of the previous blocks executed, execute the else block statements.
while *pred* stmts	WhileStmt of Expr * Stmt list	Run the predicate. If it's true, execute the block of statments and repeat the process.
for *id* in *list* stmts	ForStmt of string * Expr * Stmt list	Evaluate the list expression, and execute the block once for each value in the list, setting a temporary variable *id* to that value.
$fun(args)$	FuncCallStmt of string * Expr list	Evaluate the arguments form left to right order (where args is a comma separated list of expressions), then look for a function with the name fun and execute the function with the specified arguments. An exception will be thrown if the number of provided arguments is incorrect.

6.3 Accessors

(Note that the left hand side of accessors can be any expression that evaluates to the list or object)

Syntax	Abstract Syntax	Meaning
arr[*index*]	ArrayAccessor of Expr * string	Access the a zero-indexed list at the specified index, failing if the index is out of bounds or the object is not a list, or assigns to the list at the index (index must be a valid index).
obj.property	PropertyAccessor of Expr * string	Access a property of an object, failing if the property does not exist, or assigns a property of an object (property does not already need to exist).
$obj.fun(args)$	PropertyFunctionCall of Expr * string * Expr list	Evaluate the arguments form left to right order (where args is a comma separated list of expressions), then look for a function with the name fun of the specified object and execute the function with the specified arguments. An exception will be thrown if the number of provided arguments is incorrect.

6.4 Composite Values

Syntax	Abstract Syntax	Type	Meaning
$[e_1, e_2, ..., e_n]$	ListLiteral of Expr list	Value list	Represents a list of elements. Evaluating the list involves evaluating each individual expression in the list, and returning a lit containing the result of the evaluations.
$a\ \ [op] =\ \ b$	AssignmentStmt of LValue * BinaryOp option * Expr	Stmt	Represents an assignment of a variable a to a value b. If the optional op is included, represents taking the value of a and b and applying op to them, then storing the value to a.
let $a = b$	LetStmt of string * Expr	Stmt	Represents an assignment of a newly created local variable a to a value b. This newly created local variable can shadow and existing variable of the same name in an outer scope.
$expr$	Expr	Expr	Represents an expression that can be evaluated to produce a value.
$stmt$	Stmt	Stmt	Represents a statement that can be executed sequentially in program order.
$defn$	Defn	Defn	Represents an upper level definition. A complete program is made up of Defns.
$(expr)$	ParensExpr of Expr	Parenthases ensure that the expression inside is evaluated as a group. Use to override operator precedence.	
$fun(args)$: $stmts$	FunctionDefn of string * string list * Stmt list	ValFunc	Represents a function, defined by the function name (string), a list of arguments (string list), and a list of statements(Stmt)
$scopename$: $defns$	ScopeDefn of string * Defn list	ValFunc	Represents a scope (the equivalent of a class). The scope is instantiated by calling a function $scopename()$, and each instance has unique properties. The created instance will have type ValReference.

6.5 Operators

Syntax	Input Type	Resulting Type	Meaning
$a + b$	ValInt * ValInt	ValInt	Returns the sum of the input values. Note: does not work for string concatenation; use the `concat` builtin function instead.
$a - b$	ValInt * ValInt	ValInt	Returns the difference between the first and second values.
$a * b$	ValInt * ValInt	ValInt	Returns the product of the input values.
a/b	ValInt * ValInt	ValInt	Returns the quotient of the first and second values.
$a\%b$	ValInt * ValInt	ValInt	Returns the modulo of the first and the second values; (the modulo is the remainder after division).
$a == b$	Value * Value	ValBool	Returns true if the inputs are equal, false otherwise. Works for the following Value types (the two inputs must be the same type): ValBool, ValInt, ValString, ValFunc, ValList (reference not structural equality), ValReference.
$a! = b$	Value * Value	ValBool	Returns false if the inputs are equal, true otherwise. In other words, equivalent to $!(a == b)$.
$a <= b$	ValInt * ValInt	ValBool	Returns true if the first value is less than or equal to the second value, false otherwise.
$a >= b$	ValInt * ValInt	ValBool	Returns true if the first value is greater than or equal to the second value, false otherwise.
$a < b$	ValInt * ValInt	ValBool	Returns true if the first value is less than the second value, false otherwise.
$a > b$	ValInt * ValInt	ValBool	Returns true if the first value is greater than the second value, false otherwise.
$a\&b$	ValBool * ValBool	ValBool	Returns true if all inputs are true, false otherwise.
$a \vert b$	ValBool * ValBool	ValBool	Returns true if any input is true, false otherwise.
$!a$	ValBool	ValBool	Returns true the input is false, false otherwise.
$-a$	ValInt	ValInt	Returns the additive inverse of the input value. Equivalent to $a * -1$.
$+a$	ValInt	ValInt	Returns the input value.

6.6 Builtin Functions

Syntax	Input Type	Resulting Type	Meaning
$str(v)$	Value	ValString	Returns the string representation of the value.
$print(v)$	Value	ValNone	Prints the string representation of the value to the console, with a trailing newline.
$printraw(v)$	Value	ValNone	Prints the string representation of the value to the console, with no trailing newline.
$input([v])$	Value option	ValString	Reads input from standard input into a string. If an argument is specified, uses the string representation of that argument as a prompt for the input.
$sqrt(i)$	ValInt	ValInt	Returns the floor of the square root of the input, failing on a negative input.
$pushf(l, v)$	ValList * Value	ValNone	Pushes a value v to the beginning of a list l.
$popf(l)$	ValList	Value	Pops a value from the beginning of a list and returns it.
$concat(v1, v2)$	ValListOrString * ValListOrString	ValListOrString	Concatenates two lists or two strings together. If lists are concatenated, does not modify the reference to the original list.
$len(v)$	ValListOrString	ValInt	Returns the length of the input list or string.
$range([a], b)$	ValInt option * ValInt	ValList	If there are two arguments, return a list of ValInts from a inclusive to b exclusive (the list contains all values from a to $b-1$). If only one argument is specified, the first argument is implied to be zero.
$clone(l)$	ValList	ValList	Returns a new list that is a copy of the input list.
$sublist(l, a, [b])$	ValList * ValInt option * ValInt	ValNone	Returns a new list that contains the elements of an existing list from index of the first argument (inclusive) to the index of the second argument (exclusive), or the end of the list if the second argument is omitted.
$reverse(v)$	ValListOrString	ValListOrString	Reverses a list or string (modifying the current list), and returns it.
$rand(i)$	ValInt	ValInt	Returns a random integer from 0 to $i-1$, inclusive.
$int(str)$	ValString	ValInt	Parses the input string to its the integer value, fails if it cannot occur. It is recommended to use *tryInt* first to verify if the string can be parsed to an integer.
$tryInt(str)$	ValString	ValBool	Returns true if the string can be parsed to an int, and false otherwise.

$toCharList(s)$	ValString	ValList	Returns the list of characters of a string (characters are represented by ValInts with their ASCII/Unicode representation).
$fromCharList(l)$	ValList	ValString	Returns a string formed by a specified list of characters (characters are represented by ValInts with their ASCII/Unicode representation).
$charToStr(i)$	ValInt	ValString	Returns the string representation of a single character (characters are represented by ValInts with their ASCII/Unicode representation).
$charAt(s, i)$	ValString * ValInt	ValString	Returns the character at a specified index of a string (characters are represented by ValInts with their ASCII/Unicode representation).

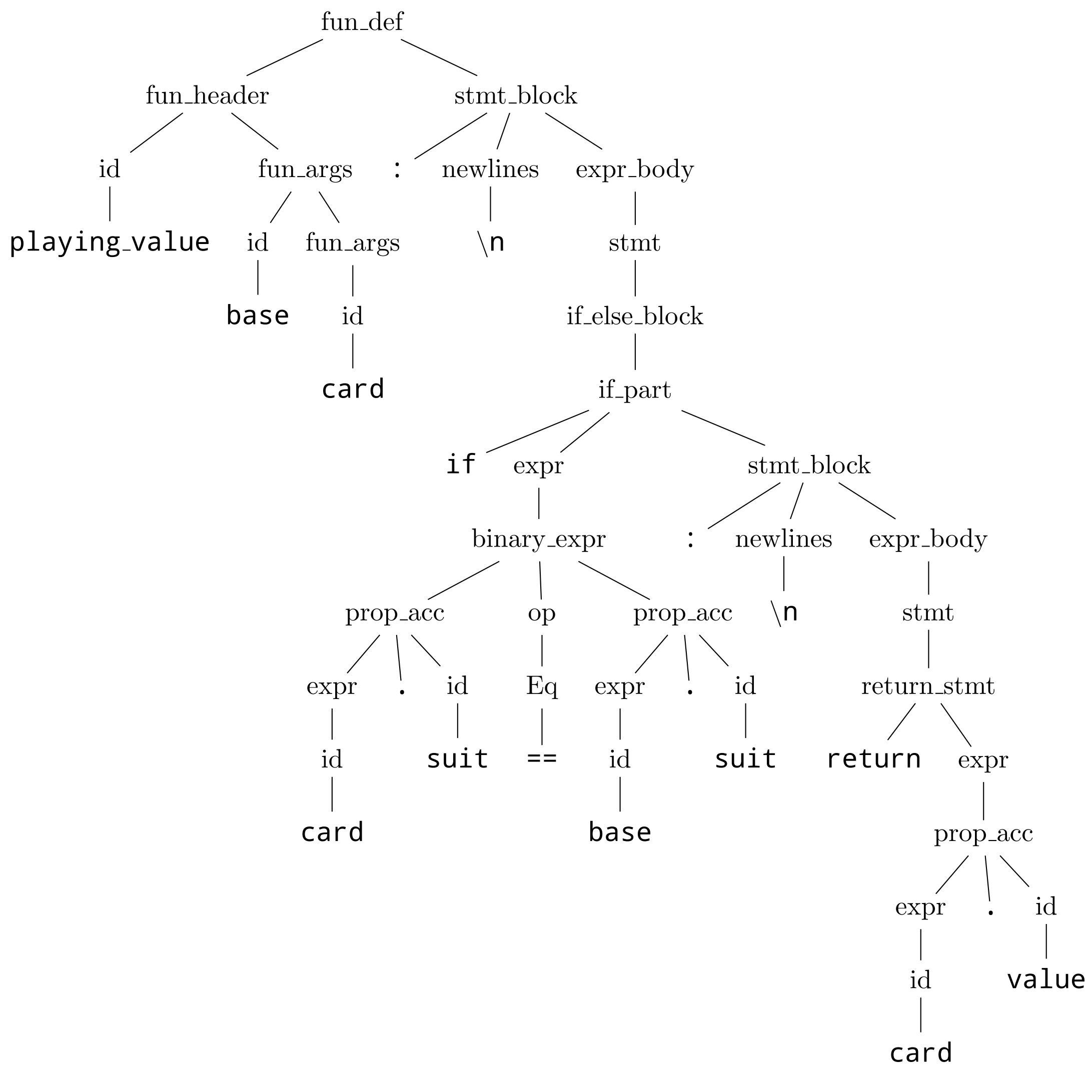
My program will be represented in an abstract syntax tree (AST) fashion in an algebraic data type. For example, the following code excerpt:

```
playing_value(base, card):
    if card.suit == base.suit:
        return card.value
```

Would be stored in the following alegbraic data type structure, subject to change:

```
Function("playing_value", ["base", "card"],
    [IfStmt(
        BinaryExpr(
            BinaryOp(Eq)
            LValue(PropertyAccessor("card", "suit"))
            LValue(PropertyAccessor("base", "suit"))
        )
        [ReturnStmt(
            LValue(PropertyAccessor("card", "value"))
        )]
    )]
)
```

This same code excerpt would be represented in the abstract syntax tree structure in the following form (ignoring indentation for now).



This language supports lexical scoping and strong dynamic typing. Any error that occurs during execution will cause the program to halt with an error message describing the problem. The interpreter will run the main function of the global scope as the entry point, with an empty state.

7 Remaining Work

Currently, my project already achieves the goal of a general purpose programming language with various features. In addition, I already have a web interface/playground to test my project on, with working "command line" output.

As a stretch goal, I could try to implement more additional language features. Some of the potential features to include are the following:

- Higher order functions
- Constructors
- Escape characters in strings

- Lambda expressions
- Line-numbers during execution
- Dictionaries
- Multi-line statements/expressions
- Floating point numbers
- Multi-file support
- External language support

Finally, though almost all programs written with my syntax can be parsed, like most general purpose programming languages, many end up being invalid programs when run. Therefore, implementing an "exception handling" system in my language could be another stretch goal for my project.