CS 334 F#Snake Language Specs

Markus Feng

Due 2018-11-29

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.

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.

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
```

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`, `array`. In this case, variables act as primitives, while methods, control flow, and complex objects act as combining forms. 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> ::= <fun_name_args>
<fun_name_args> ::= <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> <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.

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.

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.

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.
$aop = b$	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.
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.
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)

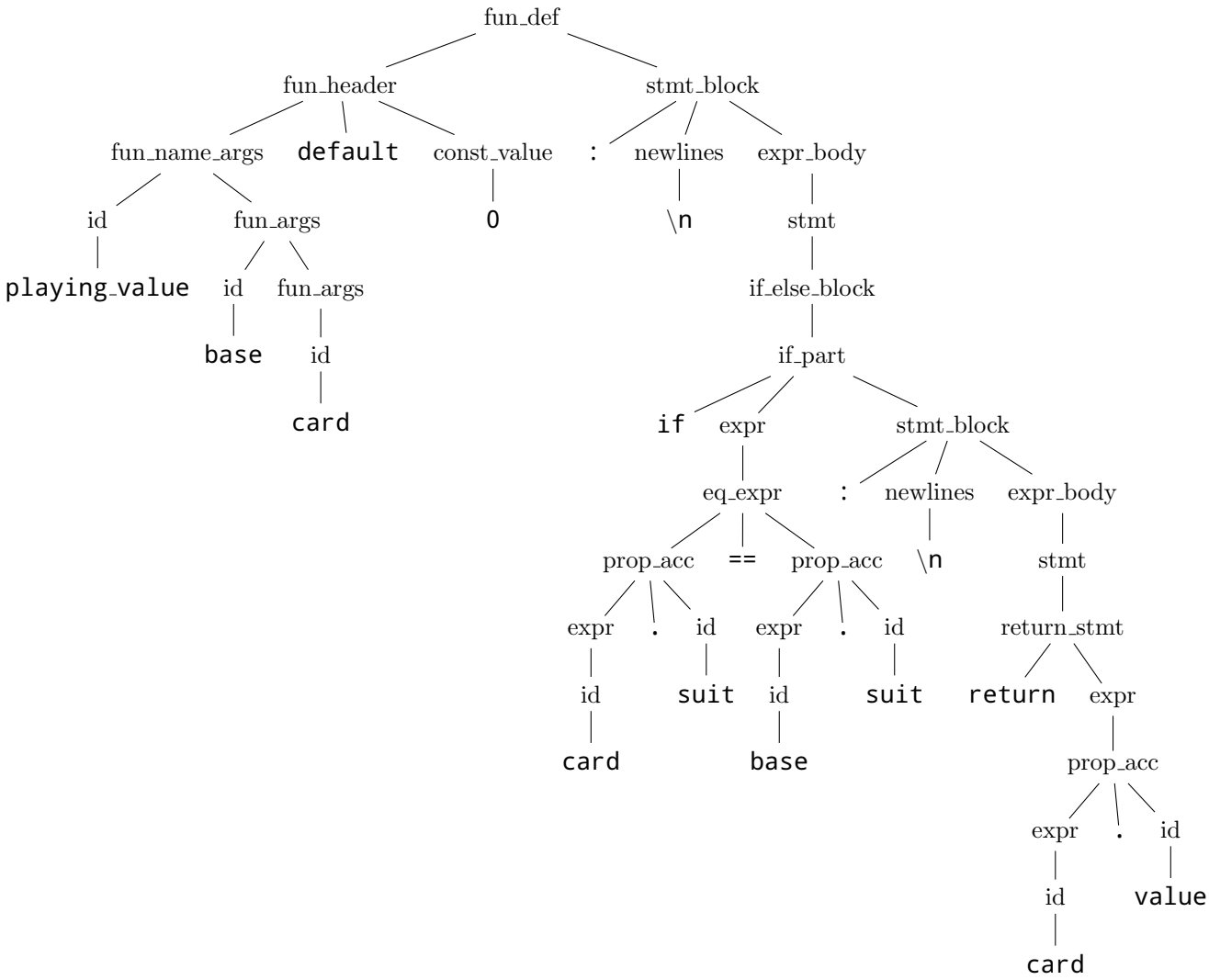
My program will be represented in an abstract syntax tree (AST) fashion in an algebraic data type. For example, the following code excerpt:

```
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 is currently planned to support lexical scoping and strong dynamic typing, due to it being an interpreted language. Any error that occurs during execution will cause the program to halt with an error message describing the problem. Upon being interpreted, will run the main function of the global scope, 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. For my project, a stretch goal would be to allow programs written in my language the ability to receive input. One of the challenges for this to work is to adapt such an input so that it works on the web client. From my knowledge, Javascript does not have the ability to perform blocking calls, so I may need to modify my program so that part of the program is executed prior to the blocking call, and that the program resumes execution from where it left off after the input.

Additionally, 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.