Stop Language Reference Manual

Jonathan Barrios (jeb2239), Jillian Knoll (jak2246), Lusa Zhan (lz2371), and James Stenger (jms2431)

Introduction

Stop is a general purpose programming language, syntactically similar to Scala, that compiles to the LLVM Intermediate Representation (LLVM IR). It is object-oriented and has type inference capabilities.

As a general purpose language, Stop has limitless possibilities for use cases. Users can harness the speed of a relatively expressive language compiled to bytecode, allowing users to combine Stop in a library with the language of their choosing. Stop can be utilized for a wide range of purposes, ranging from high level data computation to application building.

Since Stop is object-oriented, it allows for robust user created libraries. Every function and variable will be considered intrinsically as an object. This system will allow users of our language to easily create easily reusable inheritance structures to allow libraries to interact with additional languages.

Types

- Primitive Data Types
    - Int
        - The type integer stores numerical values in 32 bits.
        - Methods of declaring an int
            - Type inferred method
                - var a = 10;
                - var b = square(a);
            - Non type inferred method in function declaration
                - def square = (var u:Int):Unit { print(u*u); }
    - Float
        - The type float stores numerical values in 64 bits.
        - Methods of declaring a float
            - Type inferred method
                - var a = 10.5;
                - var b = square(a);
            - Non type inferred in function declaration
                - def square = (var u:Float):Unit { print(u*u); }
    - Unit
        - The type unit indicates that a function does not return an object at the point of the function call.

                - This is utilized in functions that print values rather than returning an object.
                - def square = (var u:Float):Unit { print(u*u); }
                - The above function is of type Unit as the print(u*u) statement does not create an object to return.
    - Char
        - The type char indicates a single character stored as 1 byte integer. char is declared by enclosing the single character with a single set of quotation marks.
        - Methods of declaring a char
            - Type inferred method
                - var a = ‘a’;
            - Non type inferred in function declaration
                - def print_letter = (var u:Char):Unit { print(u);}
    - Bool
        - The boolean type is a binary value that stores a true or false in one bit. A type null cannot be used to declare a boolean variable.
        - Methods of declaring a boolean
            - Type inferred method
                - var b = true;
            - Non type inferred method in function declaration
                - def true_or_false = (var u:bool):Unit {print(u);}

- Immutability
    - Both primitive and nonprimitive data types can be declared as immutable by placing the reserved word let before the declaration
        - let a:Bool = true;
        - let x = 3;
- Scope of primitive data types
    - Both primitive and nonprimitive data types can be declared to have a global scope by placing the reserved word global before the declaration
        - pub let a = 3;
- Casting
    - Casting is prohibited and will result in a compile time error.
- Non-Primitive Data Types
    - Arrays
        - Arrays are a data structure used to store objects, consisting of both primitive types and other user created or standard library datatypes,. Array indexing begins at 0.
        - Array declaration
            - Array declaration
                - Arrays can be declared in the following methods

        - Type Declaration method, empty array:
          Array<Type> name[size]
            - Array<Int> arr[3]; //declared array initialized to 0's
        - Type Inferred method:
            - arr = ({<list of literals separated by objects>});
            - arr = ({1,2,3,4,5}); //inferred array of ints
        - Non Type Inferred method
            - arr = Array<Type>({<list of literals separated by objects>});
            - arr = Array<Int>({1,2,3,4,5});
- Array access
    - Accessed through the <index>] operator in shorthand, representative of the .operator[](<index>) in longhand
        - int x = arr[3]; //x now contains the integer stored at position 3
        - int x = arr.operator[](3); //x now contains the integer stored at 3.

Lexical conventions

The types of tokens include identifiers, keywords, literals, comments, separators, white space, and operators. White space serves to separate tokens.

- Identifiers
    - Only alphabetical letters, digits, and the underscore character '_' may be used in variable, object, function, and class declaration. All such names must begin with a lowercase 'a'-'z' letter but may include any other listed characters in the remainder of the name.
    - We also have type identifiers, these must start with an uppercase letter 'A'-'Z' but can contain any letter afterwards
- Keywords
    - Keywords are reserved words within the language. These words cannot be overloaded by another function declaration. Keywords include the names of primitive and nonprimitive data types as well as words defining conditional statements, modifiers, and function declaration components.
    - if, else, elseif, for, while, return,
    - Int, Float, Bool, Char, Unit
    - def, class, #include, operator , trait
    - let, pub
- Literals
    - Literals are the notation in source code for representing primitive data types in source code.
        - Integer Literals

- Integer literals are indicated using optionally signed decimal notation, denoted by a string of repeating digits. Integers cannot be declared using scientific or exponential notation.
  an integer is matched with the regular expression
  int = ['0'-'9']+

- Character Literals
  - Character literates are denoted by a single, lowercase, a-z character enclosed by two single quotes. These are the only forms of information which may be stored within a character variable
    a char is matched with the regular expression
    char = '['a'-'z']'
- Float Literals
  - A float literal is denoted by an integer part, a decimal point, 0 or more digits after the decimal point, an e, and an optionally signed integer exponent. Either the integer or digits after the decimal point may be missing, either the decimal point or the exponent portion may be missing.
- Boolean Literals
  - A boolean literal is denoted by one of two reserved words, true or false.
    a bool is matched with the regular expression
    bool = ["true"|"false"]
- String Literals
  - A string literal is denoted by a series of ASCII characters and white space enclosed with double quotes. Escape sequences must be used for the identification of whitespace literals within a string.

- Separators
  - Separators are used to denote the distinction between multiple tokens in source code.
    - rule token = parse
    - [' ' '\t' '\r' '\n'] { token lexbuf }
    - | "//" { single_comment lexbuf }
    - | "/*" { multi_comment lexbuf }
    - | '\n' { NEWLINE }
    - | '(' { LPAREN }
    - | ')' { RPAREN }
    - | '{' { LBRACE }
    - | '}' { RBRACE }
    - | '[' { LSQUARE }
    - | ']' { RSQUARE }
    - | ':' { COLON }

        - | ';' { SEMI }
        - | ',' { COMMA }
- Operators
  The following operators are in use as lexical tokens. These operators
    - (* Operators *)
    - | '+' { PLUS }
    - | '-' { MINUS }
    - | '*' { TIMES }
    - | '/' { DIVIDE }
    - | '=' { ASSIGN }
    - | '%' { MODULO }
    - | "==" { EQ }
    - | "!=" { NEQ }
    - | '<' { LT }
    - | "<=" { LEQ }
    - | ">" { GT }
    - | ">=" { GEQ }
    - | "&&" { AND }
    - | "||" { OR }
    - | "!" { NOT }
    - | “.” {DOT}
- White Space
    - The following white space characters are in use and must be referenced as literals using the escape character ‘\’
        - ‘ ‘, ‘\t’, ‘\r’, ‘\n’
- Capitalization
    - Capital letters are used to define classes as a user defined object type.
    - Variables must be instantiated by names beginning with a lowercase ‘a’-’z’ letter. Capital letters can be used within the remainder of the variable name. The only reserved word that is capitalized is the void return type of “Unit”.
- White Space
    - One single space, ‘ ‘, is required to separate tokens
    - All other white space is ignored. Thus the following forms are equivalent
        - def true_or_false = (var u:bool):Unit
          {print(u);}
        - def true_or_false
          = (var u:bool):Unit
          {print(u);}
        - def true_or_false = (var u:bool):Unit {print(u);}
- Comments
    - Comments are denoted with the ‘//’ or ‘/*’ sign, differentiating a single line or comments or multiple lines of comments

- #include is a reserved keyword though the word include is not otherwise reserved.
  - // single line of comments
  - /* multiple lines
    of comments */

Expressions

- Function definition
  - Functions are defined with the def keyword, the function name, the assignment operator, a list of arguments, and a type declaration for the return type of the function. A return statement must exist within the function if the function does not return type unit.

```
def <function Name> = (<var <varname>:<Var Type>, additional args):<ReturnType>
{ statements to execute
        return <Object of Return Type>;

def negate = (var i:Int):Int
{ return -i;}

def count =(var arr:Array):Unit
{ for(vara=0;a<10;a++)
{ print(arr[a]); } }
```

- Object instantiation
  - An object must either be a primitive datatype, standard library datatype, or user defined datatype as denoted by a class definition.
  - Objects are instantiated with either a inferred type or type declarative format.

    var x = “abc”; //a is type inferred as type string
    var x = 1; //a is type inferred as type int

    var x:int = 1; //declared that x is of type int
  - When specified in function calls as arguments or return types the type of the object must be specified.
    - def funct_name = (var var_name:<Var_type> <with additional args separated by commas>):<Return_type>
    - def square = (var u:Int):Int {return(u*u); }
    - The type int is specified after var u with a colon in the argument list of the function to produce (var u:Int)
    - The type int is specified as a return type outside of the argument list for the function with the colon to produce (var u:Int):Int
- Function calls

- Functions are called by referencing the function name and any arguments in parentheses. Unless the return type is of type Unit, the function call must be enclosed within a conditional statement or on the right hand of the assignment operator.

```
def increm(var a:Int):Int {return a + 1;}
var x = increm(3);

if(increm(x) == 2
```

Operators

The following methods are associated with the operator tokens described in the Keywords section.

- Assignment
  - =
    - The' =' operator assigns values to variables, function, and class declaration.
    - The left side of the '=' operator contains a reference to the type of expression that is being called, def for function declaration, var for variable instantiation, or class for class definition followed by the name of the variable. The first letter of the variable corresponds to whether this is the declaration of a new type of object, as denoted by a capital first letter, or a lowercase letter for function declaration or variable declaration.
    - The right side of the '=' operator contains an expression that will produce the corresponding type specified by the left hand side of the operator. Otherwise a compile time error will be produced.

```
def square = (var u:Int):Unit { print(u*u); }
var x = 3;
```

- Arithmetic

  Arithmetic operators must take two objects of the same type as arguments.

```
var  x = 1 ;
var y = 4.2;
var z = x + y; //cannot perform operation on two objects of different types.
```

  - +
    - The addition operator.

```
var x = 4 + 2; //6 is stored in x
var x = 4.0 + 2.0; //6.0 is stored in x
```

  - -
    - The subtraction operator.

```
var x = 4 - 2; //2 is stored in x
var x = 4.0 - 2.0; //2.0 is stored in x
```

    - The subtraction operator can also be used for negation.

```
var x = 2; //2 is stored in x
```

var y = -x; //-2 is stored in x

- *
  - The multiplication operator.

var x = 4 * 2; //8 is stored in x
var x = 4.0 * 2.0; //8.0 is stored in x

- /
  - The division operator.

var x = 4 / 2; //2 is stored in x
var x = 4.0 / 2.0; //2.0 is stored in x

- %
  - The modulo operator. Only integer variables or literals may be taken as arguments/

var x = 10 % 2; //2 is stored in x
var y = 12.5 % 2; //This will not compile as one argument is a float literal

- Array
  - Access and Declaration
    - Array access is performed with the [<index>] operator. Array indexes start at 0.
    - In array declaration when using a specified list of objects the objects must be separated with a comma, enclosed with ({<comma separated list>}).
      arr = Array<Int>({1,2,3,4,5});
      var x = arr[3]; //4 is stored in x
- Conditional Operators

Conditional operators return a boolean value when the condition defining the operator is met. Comparisons to null are acceptable for the '==' and '!=' operators only.

The '<=', '>=', '<', and '>' operators are only defined for the int, float, and char primitive datatypes. User defined object types must contain a definition of the '<=', '>=', '<', and '>' operators. These operators can be overloaded through function definition.

- ==
  - The conditional operator '==' will evaluate whether the two objects surrounding the operator are equivalent. The condition will evaluate to true for equivalence and false otherwise.
  - The '==' operator should not be used to evaluate the equivalence of type float variables or literals due to rounding error.

var x = 1;
var y = 1;
if(x == y) {return true;} //condition evaluates to true

- !=
  - The conditional operator '!=' will evaluate whether the two objects surrounding the operator are equivalent. The condition will evaluate to true for unequal objects and false otherwise.

        - The ‘!=’ operator should not be used to evaluate the equivalence of type float variables or literals due to rounding error.

```
var x = 1.5;
var y = 2.5;
if(x != y) {return true;} //condition evaluates to true
```

- <=
    - The ‘<=’ operator will evaluate whether the argument on the right hand side contains a value that is less than or equal to the value contained on the right hand side.

```
var x = 1.5;
var y = 2.5;
if(x <= y) {return true;} //condition evaluates to true

var x = ‘c’;
var y = 1.5;
if (x <= y) {return true;} //cannot attempt to compare two objects of different types.
```

- >=
    - The ‘>=’ operator will evaluate whether the argument on the right hand side contains a value that is greater than or equal to the value contained on the right hand side.

```
var x = 1.5;
var y = 2.5;
if(y >= x) {return true;} //condition evaluates to true

var x = ‘c’;
var y = 1.5;
if (x > y) {return true;} //cannot attempt to compare two objects of different types.
```

- >
    - The ‘>’ operator will evaluate whether the argument on the right hand side contains a value that is greater than the value contained on the right hand side.

```
var x = 1.5;
var y = 2.5;
if(y > x) {return true;} //condition evaluates to true

var x = ‘c’;
var y = 1.5;
if (x > y) {return true;} //cannot attempt to compare two objects of different types.
```

- <
    - The ‘<’ operator will evaluate whether the argument on the right hand side contains a value that is less than the value contained on the right hand side.

```
var x = 1.5;
```

```
var y = 2.5;
if(x < y) {return true;} //condition evaluates to true

var x = 'c';
var y = 1.5;
if (x > y) {return true;} //cannot attempt to compare two objects of different types.
```

- Dot operator
  - Shorthand
    Operators can be referenced by writing the string literal of the lexical token in the source code.
    ```
    var a = 3 + 5; // the + operator is referenced by its string literal
    ```
  - Longhand
    Operators can be referred to with the .operator<type>(<argument>) notation
    ```
    var x = 1 + 2; //shorthand notation
    var x = 1.operator+(2) //longhand notation produces logically equivalent outcome
    ```
- Logical operators
  - Logical operators can be used to separate multiple conditions within conditional statements.
  - ||
    - The OR operator will evaluate to true when at least one of the conditions on either side of the OR operator evaluates to true. The OR operator can be used to sequence multiple logical conditions. One of the conditions must evaluate to true in order for the sequenced OR statements to evaluate to true.

    ```
    var x = false;
    var y = true;
    var z = false;
    if( x || y) //the condition within the if statement evaluates to true
    if ( x || y || z) //the condition within the if statement evaluates to true
    ```
  - &&
    - The AND operator will evaluate to true if both conditions on the either side of the AND operator evaluate to true.

    ```
    var x = true;
    var y = false;
    var z = true;
    if( x && y) //the condition within the if statement evaluates to false
    if (x && z) // the condition within the if statement evaluates to true
    ```
  - !
    - The NOT operator will negate the logical value provided by the conditional statement following the NOT operator.

    ```
    var x = false;
    var y = !x; //true is stored in x
    if (!(3 == 4)) //the condition within if statement evaluates to true
    ```

- Precedence
    - The order of precedence is as follows, ordered by from highest to lowest precedence. The order of precedence refers to the lexical tokens utilized for each operator. If the operator is overloaded then the overloaded operator will maintain the original level of precedence. When multiple operators have equivalent precedence the expressions are evaluated from left to right.
        - Calls to functions
        - Array access operators
        - Arithmetic negation and logical negation
        - Multiplication and division operators
        - Addition and subtraction operators
        - Greater than or equal to, less than or equal to, greater than, or less than operators (‘>=’,’<=’, ‘<’, ‘>’)
        - Equals (‘==’) and not equals (‘!=’) conditional statements
        - Logical AND operator
        - Logical OR operator
        - Assignment operations

Statements

- Expressions
    - The format for expressions involving arithmetic, array, assignment, conditional statements, and logical operators is described in the Operators section. These declarations must be terminated by a semi-colon.
- Declarations
    - The declaration format for variables, functions, and classes is described in the Expression section. These declarations must be terminated by a semi-colon.
- Control Flow
    - if
        - The if statement provides a series of steps to execute when the condition inside the parentheses is met. If the condition is not met then the next instruction after the {<statements>} will be executed. if statements must be accompanied by an else statement however the else statement need not contain any statements to execute when the if condition is not met.
        - Declaration

      if (<one or more logical conditions>)
      { <statements to execute when the condition evaluates to true>}
        - Example

      if(a == 0)
      {return true;}
      else
      {return false;}
    - elseif

        - The elsif statement provides a condition and accompanying series of steps to execute when the conditions in the above if and optional elseif statements do not evaluate to true. An else statement must accompany a series of if and elsif statements.
        - Declaration

    elsif(<one or more logical conditions>)
    {<statements to evaluate when the condition evaluates to true>}

        - Example

    if(a == 1)
    {return true;}
    elseif(a % 2 == 0)
    {return true;}
    else
    {return false;}

- else
    - The else statement provides a condition and series of steps to execute when the above series of if and optional elsif statements does not produce conditions which evaluate to true.
    - Declaration

    else
    {<statements to evaluate when no if or elsif conditions evaluate to true>};

    - Example

    if(a == 0)
    {return true;}
    else
    {return false;}

- looping
    - while
        - A while loop will compute the statements enclosed in brackets below the while loop as long as the logical condition within the while loop evaluates to true. The execution path will then move to the next statement after the brackets below the while loop.

    while(<logical conditions>)
    {<statements>}

    var i = 1;
    while(i < 5)
    { print(i); i = i + 1

    - for

        - A for loop will compute the statements enclosed in the brackets below the for loop as long as the series of logical conditions within the for loop evaluates to true. The computation within the for loop will occur at the beginning of every iteration of the for loop.

```
arr = ({1,2,3,4,5});
for(var a=0;a<10;a++){ print(arr[a]); } //prints 12345
```

- return
    - Functions must be accompanied by a return statement unless the return type is specified as type Unit.

```
//functions of type unit do not require a return statement
def declaring=():Unit   { var a=0; }

//if a non-void return type is specified then a return statement is required that will return a variable of the specified type.

def increment=(var a:int):int   { var b=a + 1; return b; }
```

    - If there is no return statement accompanying all logical outcomes of conditional statements the program will not compile.

```
//The following program will not compile
def even_odd=(var a:int):bool{
if(a % 2==0)
{return true};
else
{} //there is no return statement when this logical condition is met
```

Standard Library

The following objects and their methods are included in the standard library.

- String
    - Constructor
        - Strings are instantiated using the sequence
            var x= “abc”;
        - Strings contain a sequence of ASCII characters and whitespace references enclosed with double quotations.

    - Methods in the Standard Library

```
string1.equals(string2);
//will return true if string1 contains the same content as string2
print(string)
```

//will print the character sequence contained within the string

- Print
    - The print method is defined for string literals.
        - print(“abc”) //prints abc
    - Separate print methods are included in the standard library for primitive and nonprimitive data types.
        - bool x = true;
        - print(x) #prints true
        - char y = ‘a’;
        - print(y) #prints a

Inspiration

The Dice Language Reference Manual
http://www.cs.columbia.edu/~sedwards/classes/2015/4115-fall/lrms/Dice.pdf

The C Reference Manual
https://www.bell-labs.com/usr/dmr/www/cman.pdf