INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

CODECRAFT

Documentation

Compiler Design
Ingeniera Elda Quiroga González
Maestro Héctor Gibrán Ceballos

Gabriel Berlanga Serrato
A01191670

Ricardo Canales Backoff
A01191463

May 3, 2017

User Manual

Project Description

Vision

The project's vision lies in the education topic in Mexico. Each day programming is becoming an essential requirement for daily life, since it bring many utilities. We are looking to facilitate and drive the learning of programming at young ages. Children that learn to program since early ages will fully understand how world works, and how it affects them.

Objective

The main objective with this new programming language is teaching children basic programming concepts through a graphical interface. The language's target audience is people with no prior knowledge of programming. Even so we want experienced programmers to be able to easily work through it as well and create complex programs. After utilizing the graphical interface users will be able to compile and execute the program through a browser.

Project's Scope

Even with our objective is to drive learning through simple and easy graphical programming we worked on providing an almost full programming experience with *Codecraft*. Includes basic programming logic from *functions* to *conditionals*, but is not intended for project use and does not support modules. So in the end it is intended for small scripts.

Requirement Analysis

Functional Requirements

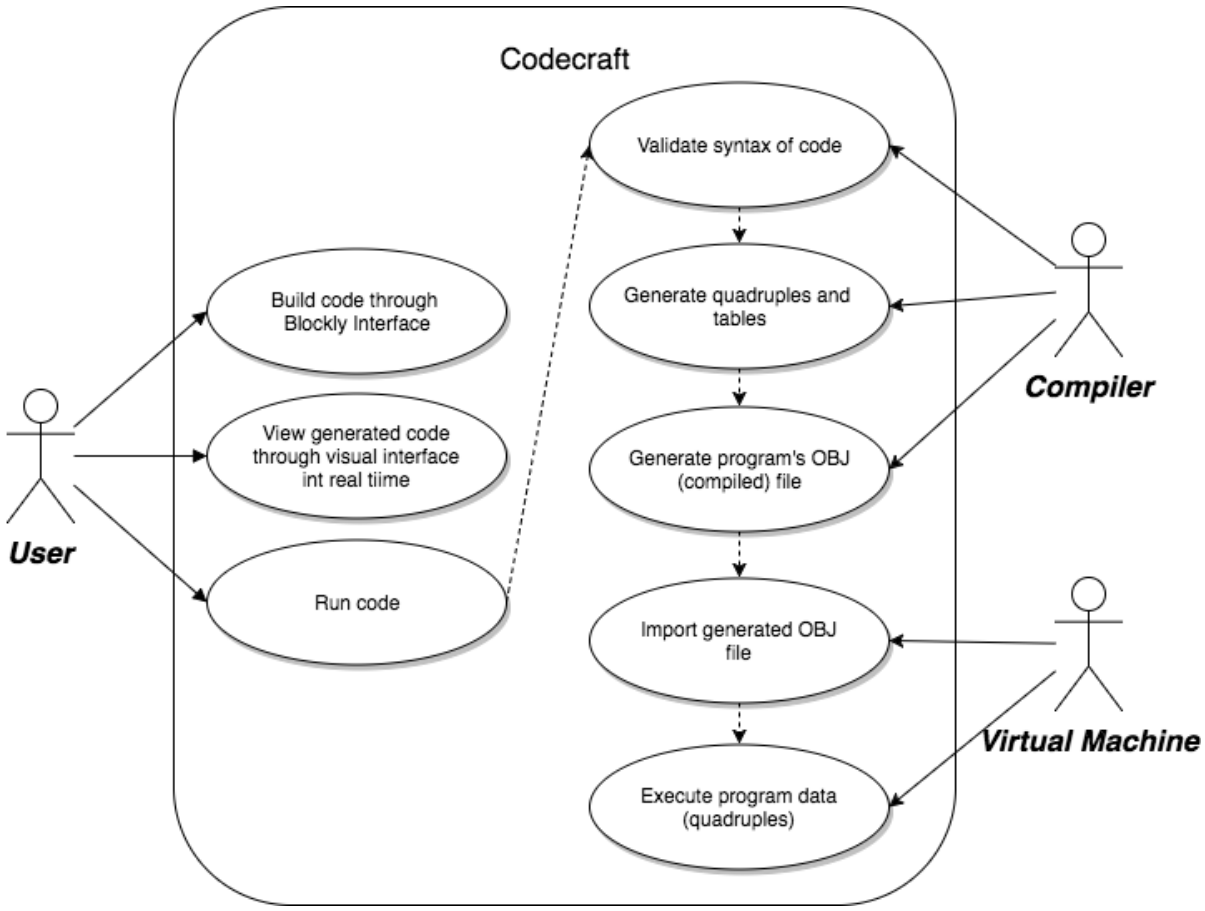
1. Compilation should be successful based on the defined rules.
2. On compilation error(rules invalid), a relevant message should be displayed.
3. The programming language should support the defined statements: variables, functions, arrays, matrix, loops, conditionals and relational, logic and arithmetic expressions.
4. The compiler should generate and export a compiled program file.

5. The *virtual machine* should import and decode the compiled *OBJ* file.
6. The *virtual machine* should be able to execute the decoded program data.
7. The graphical interface should generate syntax correct code.
8. The graphical interface should be able to compile, execute and show results to the user.

Non-Functional Requirements

- Easy syntax understanding
- Easy code building through graphical interface
- Fast execution

General Use Cases



A common user will perform 3 actions or tasks when programming with Codecraft through the graphical(web) interface. He will build the program the code with Blockly. He will verify(and learn) how the code looks like once generated by Blockly's blocks. And he will run generated code. Once the run

signal is sent, the compiler will scan, parse and generate the OBJ file base on the user input. Following the file generation the virtual machine's script will import and execute the generated quadruples from the OBJ file.

Main Test Cases Description

List of tests performed during development:

1. Tests for lexicon and syntax
2. Tests for semantics
3. Quadruple generation of:
    a. Logical, relational and arithmetic expressions
    b. Conditional statements
    c. Loop statements
    d. Function declaration and function calling
    e. Array and matrix access and declaration
4. Virtual machine execution tests of all generated quadruples
5. Memory generation and assignment tests
6. Recursion tests

General Process Description

We followed an agile development methodology occasionally working by ourselves but mostly on weekend meetings doing peer programming and discussing our approaches to certain elements of the compiler.

Reflexion

This project has definitely been one of the most demanding projects throughout my time in college. I'm proud of what we've built and of what I've learned during the development of the project. I believe the project demanded me to use most of the skills I've learned during my studies and helped me increase on them, ranging from web development to competitive programming. It has also increased my lower level understanding of how programming languages work.

_____

Ricardo Canales A01191463

This project has been the most difficult one of my college years. To know what goes behind every program and how to understand every step a computer does to run a program is something amazing. Through the semester even though we did not upload every deadline, we never stayed behind on the project. Working constantly every week helped us to meet the requirements we proposed. I really liked this kind of projects because of the challenges.

_____

Gabriel Berlanga A01191670

Language Description

Language Name

The language is called **Codecraft**, and is based on the famous *children's* game Minecraft. We've included an *unofficial* logo:

Principle Characteristics Description

Codecraft syntax and functionality is based on many languages. It's syntax looks similar Javascript without ';' at the end of lines. But it's general behaviour is similar to C++. There is no pointers or memory management, but Codecraft behaves similar with variable and array declarations. Arrays cannot be declared dynamically, but they can have unlimited dimensions. We believe this way new programmers will get a better understanding of how programming works while still working with simple syntax.

Error Description (compilacion y ejecucion)

The compiler will notice the user of the following errors:

- Syntax error
- Incompatible matching operators
- Using not previously declared variables or functions
- Missing expressions
- Calling function with wrong number of parameters
- Type-mismatch on assignment or arguments
- No boolean expression in conditionals

The virtual machine will notice the user of the following errors:

- Division by Zero
- Out of bounds access on array
- Accessing undefined variables

Compiler Description

Description

Codecraft was made with Mac, Linux and Windows machines. It is based on Python v2.7.12.

Lexical Analysis Description

Construction Patterns

Regular expressions of matching operators and types

```
t_ID = r'[a-zA-Z_][a-zA-Z_0-9]*'
t_EQ  = r'=='
t_UNEQ  = r'!='
t_GREATER_EQ  = r'>='
t_LESS_EQ  = r'<='
t_CTE_INT = r'\d+'
t_CTE_FLOAT = r'[-+]?[0-9]+\.[0-9]+([Ee][\+-]?[0-9+])?'
t_CTE_CHAR = r'\'(?:\\.|[^\'\\])\''
t_CTE_STRING = r'\"((?:\\.|[^"\\])*)\"'

# Literals
literals = [ '+','-','*','/', '%','=','(',')','{','}','[',']','<','>', ',']
```

Tokens

List of directly matching tokens.

```
    'craft' : 'CRAFT',
    'var' : 'VAR',
    'true' : 'TRUE',
    'false' : 'FALSE',
    'and' : 'AND',
    'or' : 'OR',
    'input' : 'INPUT',
    'output' : 'OUTPUT',
    'outputln' : 'OUTPUTLN',
    'return' : 'RETURN',
    'function' : 'FUNCTION',
    'void' : 'VOID',
```

```
    'if' : 'IF',
    'else' : 'ELSE',

    'while' : 'WHILE',
    'break' : 'BREAK',
    'continue' : 'CONTINUE',

    'int' : 'INT',
    'float' : 'FLOAT',
    'char' : 'CHAR',
    'bool' : 'BOOL',
    'string' : 'STRING'
```

Syntax Analysis Description

```
  program ::= 'CRAFT' "{" canvas_block "}"
  canvas_block ::= block canvas_block
                | function canvas_block
                | empty
  type ::= 'VOID'
         | 'INT'
         | 'FLOAT'
         | 'CHAR'
         | 'BOOL'
         | 'STRING'
  vars ::= 'VAR' type saveVariableType var eraseVariableType
  var ::= 'ID' saveVariableName addVariable var_array_init var_assignment eraseVariableName var_repeater
  var_repeater ::= "," var
                | empty
  var_assignment ::= pushIdOperand "=" pushOperation super_expression addAssignmentQuadruple
                   | empty
  var_free_assignment ::= 'ID' lookupId pushIdOperand eraseVariableName eraseVariableType "="
pushOperation super_expression addAssignmentQuadruple
  var_array_init ::= var_array_dimension generateDimensionSpace
                   | empty
  var_array_dimension ::= "[" 'CTE_INT' addDimension "]" var_array_dimension1
  var_array_dimension1 ::= "[" 'CTE_INT' addDimension "]" var_array_dimension1
                         | empty
  var_arr_free_assignment ::= 'ID' lookupId index_selector "=" pushOperation super_expression
addAssignmentQuadruple
  index_selector ::= saveVariableDimension "[" addFakeBottom exp removeFakeBottom validateArrayIndex "]"
index_selector1 addAddressBase eraseVariableName eraseVariableType eraseVariableDimension
  index_selector1 ::= offsetForDimension "[" addFakeBottom exp removeFakeBottom validateArrayIndex "]"
accummulateDisplacement index_selector1
                    | empty
  function ::= FUNCTION setLocalScope type saveFunctionType 'ID' saveFunctionName "("
parameters_definition ")" addFunction "{" block_repeater "}" endFunction setGlobalScope
  parameters_definition ::= type 'ID' addParameter parameters_definition1
                          | empty
  parameters_definition1 ::= "," type 'ID' addParameter parameters_definition1
                           | empty
  parameters ::= super_expression addArgument parameters1
               | empty
  parameters1 ::= "," super_expression addArgument parameters1
                | empty
```

```
  return ::= 'RETURN' returnFunction
           | 'RETURN' super_expression returnFunctionValue
  function_call ::= 'ID' "(" lookupFunction startFunctionCall addFakeBottom parameters removeFakeBottom
")" verifyArguments endFunctionCall
                  | 'ID' "(" lookupFunction startFunctionCall ")" endFunctionCall
  block_repeater ::= block block_repeater
                   | empty
  block ::= vars
          | var_free_assignment
          | var_arr_free_assignment
          | if
          | cycle
          | return
          | 'BREAK'
          | 'CONTINUE'
          | output
          | input
          | function_call
  output ::= 'OUTPUT' "(" super_expression addOutputQuadruple output1 ")"
           | 'OUTPUTLN' "(" super_expression addOutputQuadruple output1 addNewLineQuadruple ")"
           | 'OUTPUTLN' addNewLineQuadruple
  output1 ::= "," super_expression addOutputQuadruple output1
            | empty
  input ::= 'INPUT' "(" 'ID' lookupId pushIdOperand eraseVariableName eraseVariableType ")"
addInputQuadruple
          | 'INPUT' "(" 'ID' lookupId index_selector ")" addInputQuadruple
  if ::= 'IF' "(" super_expression ")" ifConditional "{" block_repeater "}" endIfConditional
       | 'IF' "(" super_expression ")" ifConditional "{" block_repeater "}" else
  else ::= 'ELSE' elseConditional else_if
         | 'ELSE' "{" elseConditional block_repeater "}" endIfConditional
  else_if ::= 'IF' "(" super_expression ")" ifConditional "{" block_repeater "}" endElseIfConditional
            | 'IF' "(" super_expression ")" ifConditional "{" block_repeater "}" else_if_else
  else_if_else ::= 'ELSE' elseIfConditional else_if
                 | 'ELSE' "{" elseIfConditional block_repeater "}" endIfConditional
  cycle ::= 'WHILE' startLoop "(" super_expression ")" loopConditional "{" block_repeater "}" endLoop
  super_expression ::= expression tryLogicalQuadruple
                     | expression tryLogicalQuadruple AND pushOperation super_expression
                     | expression tryLogicalQuadruple OR pushOperation super_expression
  expression ::= exp tryRelationalQuadruple
               | exp tryRelationalQuadruple '>' pushOperation expression
               | exp tryRelationalQuadruple '<' pushOperation expression
               | exp tryRelationalQuadruple 'EQ' pushOperation expression
               | exp tryRelationalQuadruple 'UNEQ' pushOperation expression
               | exp tryRelationalQuadruple 'LESS_EQ' pushOperation expression
```

```
              | exp tryRelationalQuadruple 'GREATER_EQ' pushOperation expression
exp ::= term tryAddSubQuadruple
      | term tryAddSubQuadruple '+' pushOperation exp
      | term tryAddSubQuadruple '-' pushOperation exp
term ::= uminus tryMultDivQuadruple
       | uminus tryMultDivQuadruple '*' pushOperation term
       | uminus tryMultDivQuadruple '/' pushOperation term
uminus ::= '-' factor generateUMinusQuadruple
         | factor
factor ::= '(' addFakeBottom super_expression removeFakeBottom ')'
         | value
value ::= constant addConstant pushConstantOperand
        | 'ID' lookupId pushIdOperand eraseVariableName eraseVariableType
        | 'ID' lookupId index_selector
        | function_call
constant ::= 'CTE_INT'
constant ::= 'FALSE'
           | 'TRUE'
constant ::= 'CTE_FLOAT'
constant ::= 'CTE_STRING'
           | 'CTE_CHAR'
```

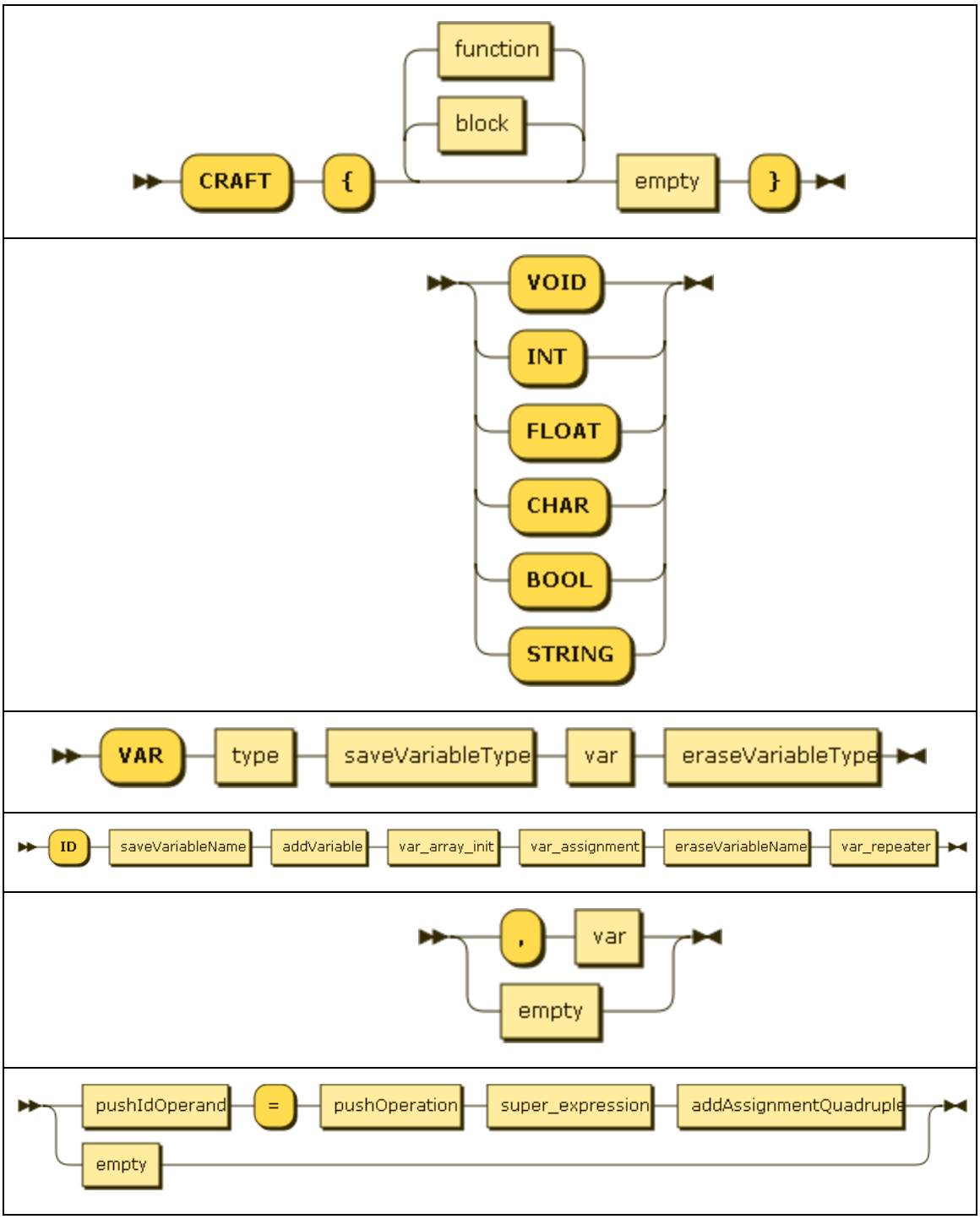
Intermid Code Generation and Semantic Analysis Description

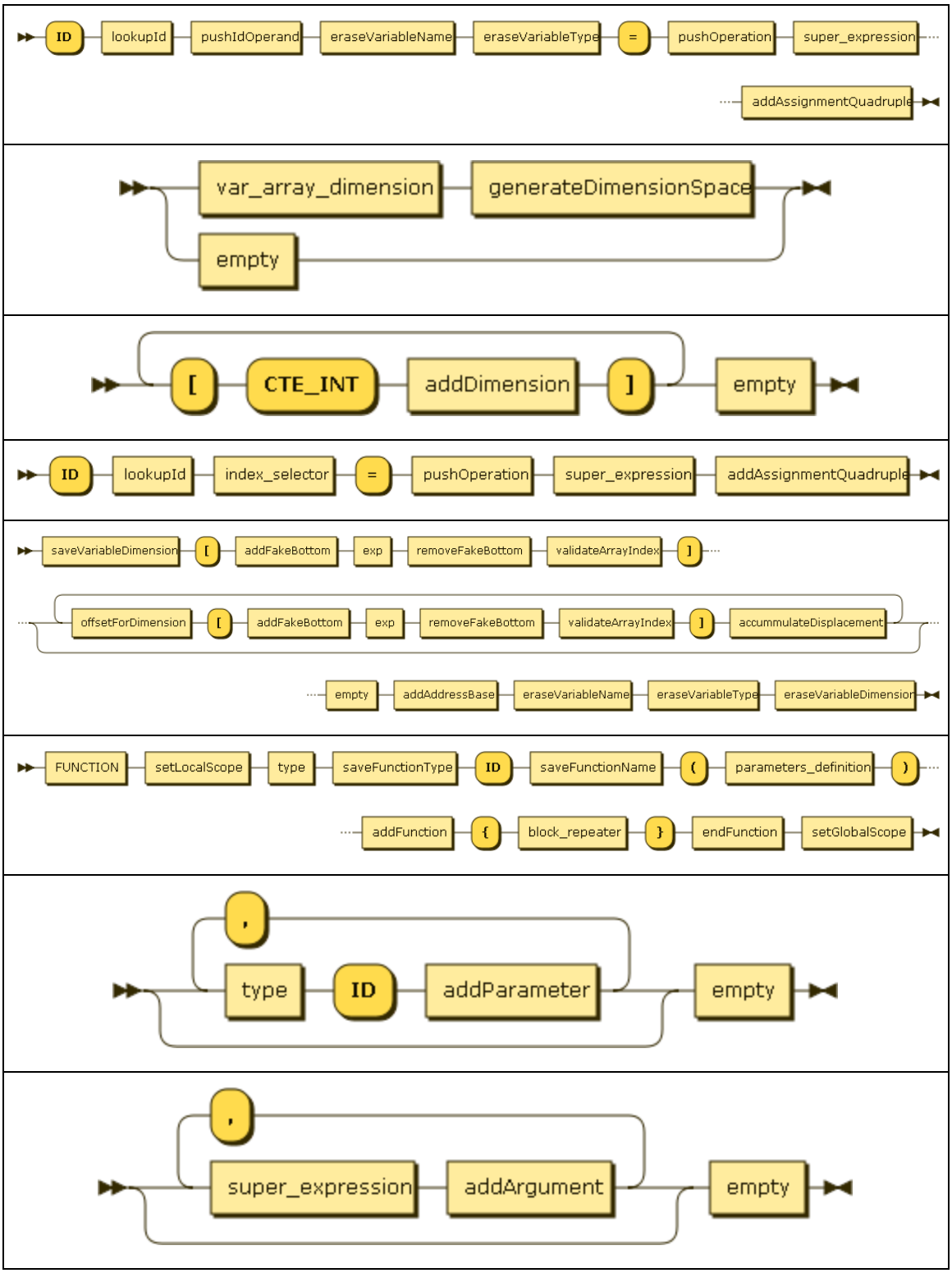
Operation Code and Associated Virtual Directions

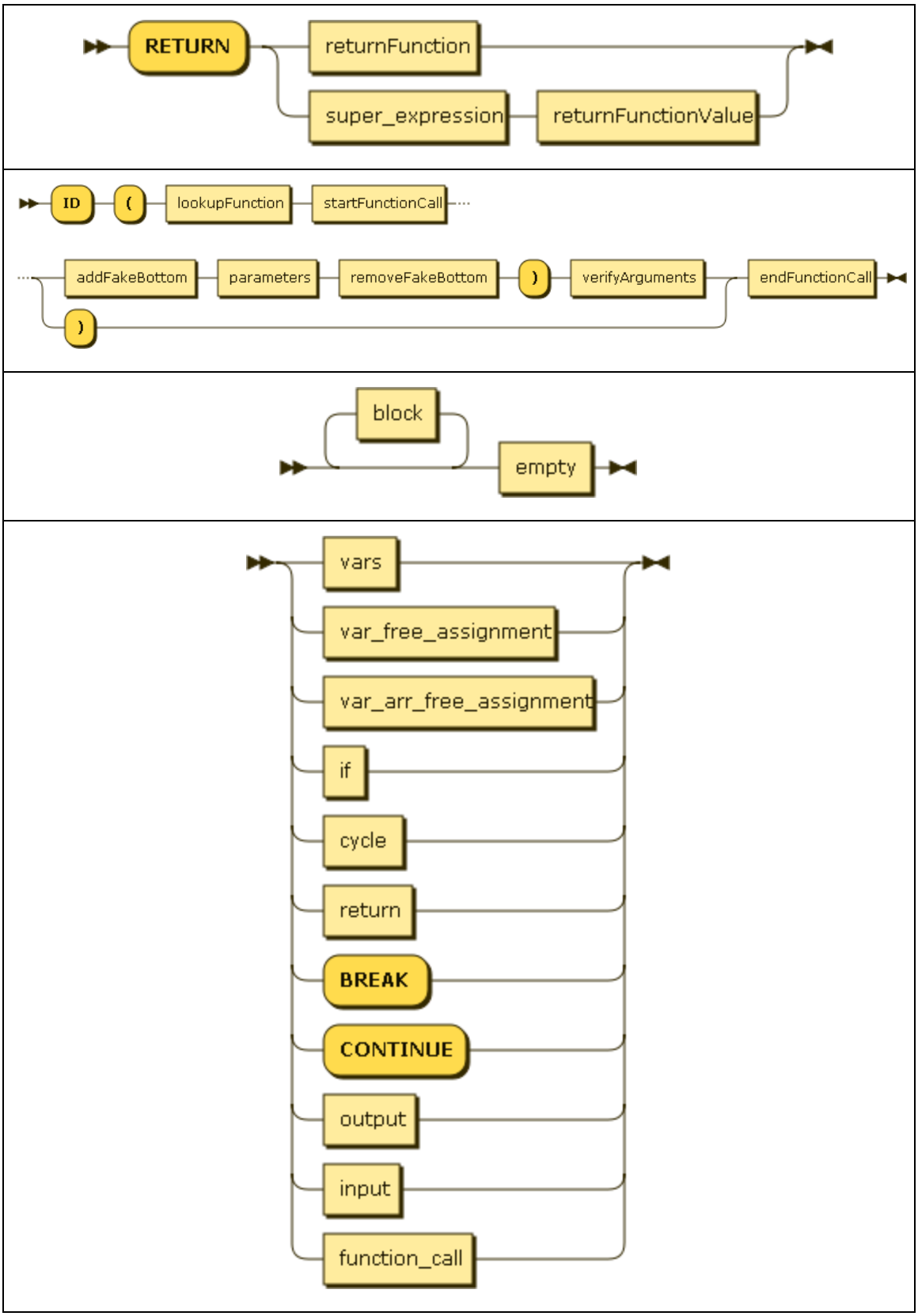
	Local	Temporary	Global	Constant
Bool	0-999	5000-5999	10000-10999	15000-15999
Int	1000-1999	6000-6999	11000-11999	16000-16999
Float	2000-2999	7000-7999	12000-12999	17000-17999
Char	3000-3999	8000-8999	13000-13999	18000-18999
String	4000-4999	9000-9999	14000-14999	19000-19999

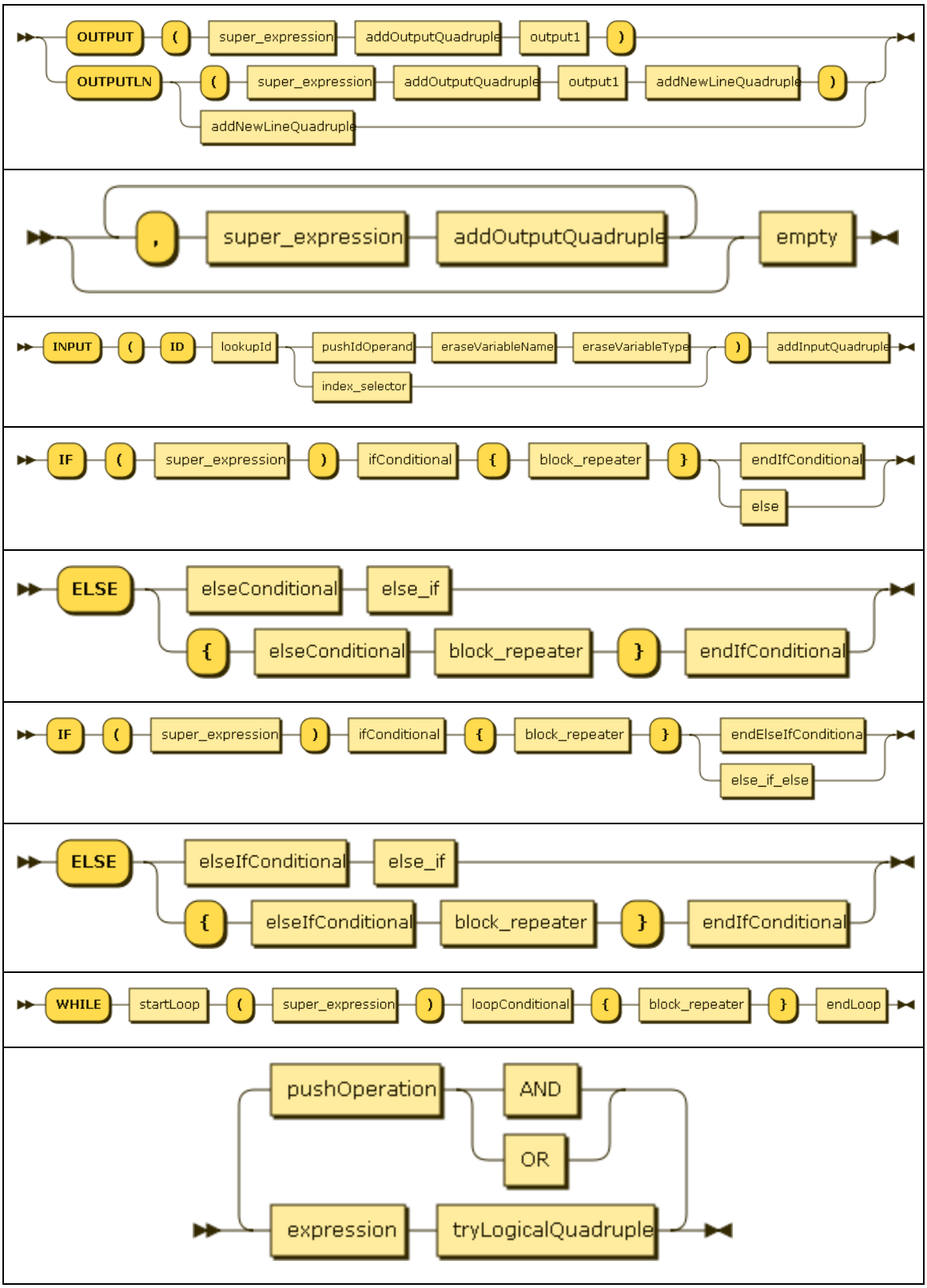
Space per type: 1000

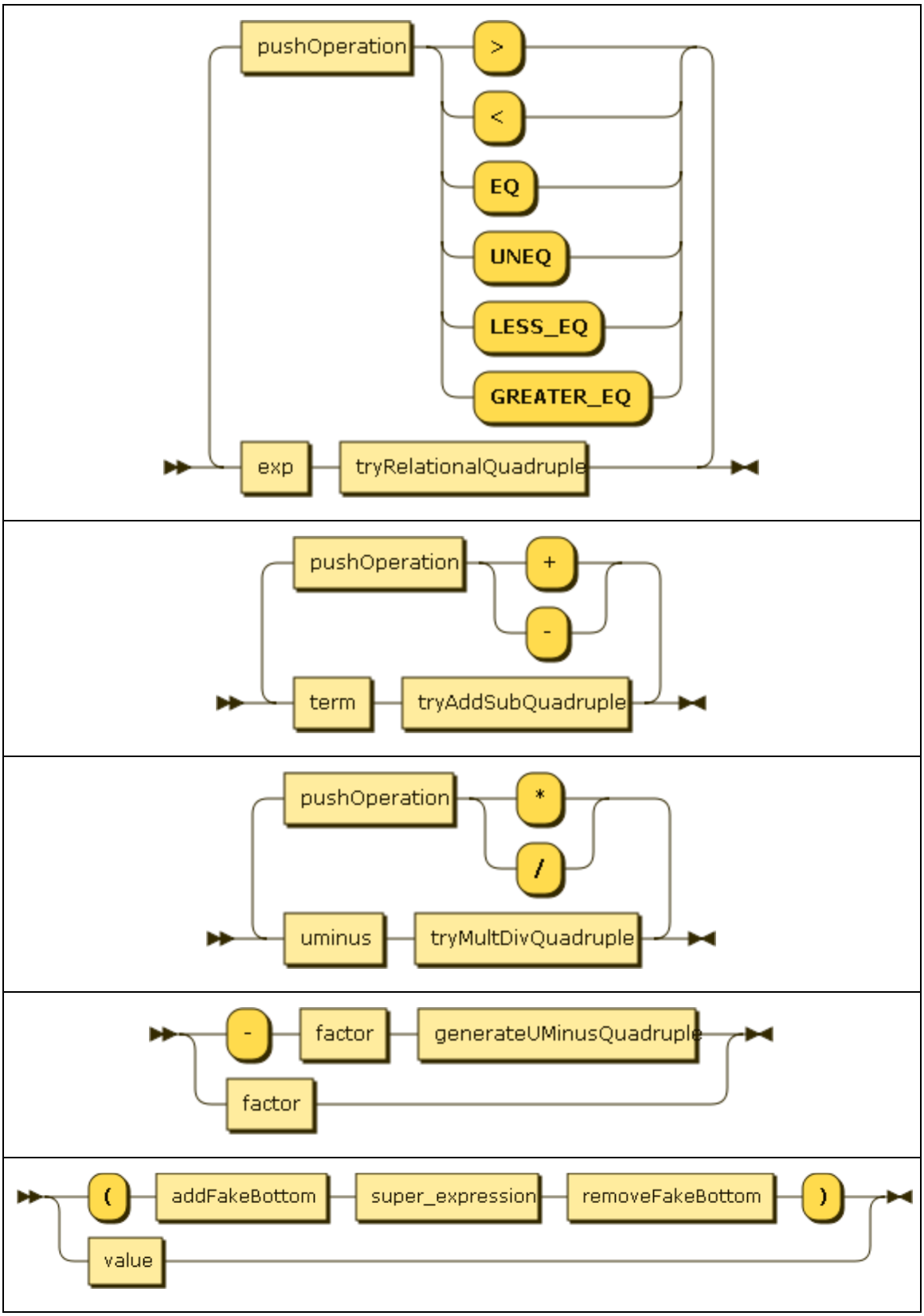
Syntax Diagrams

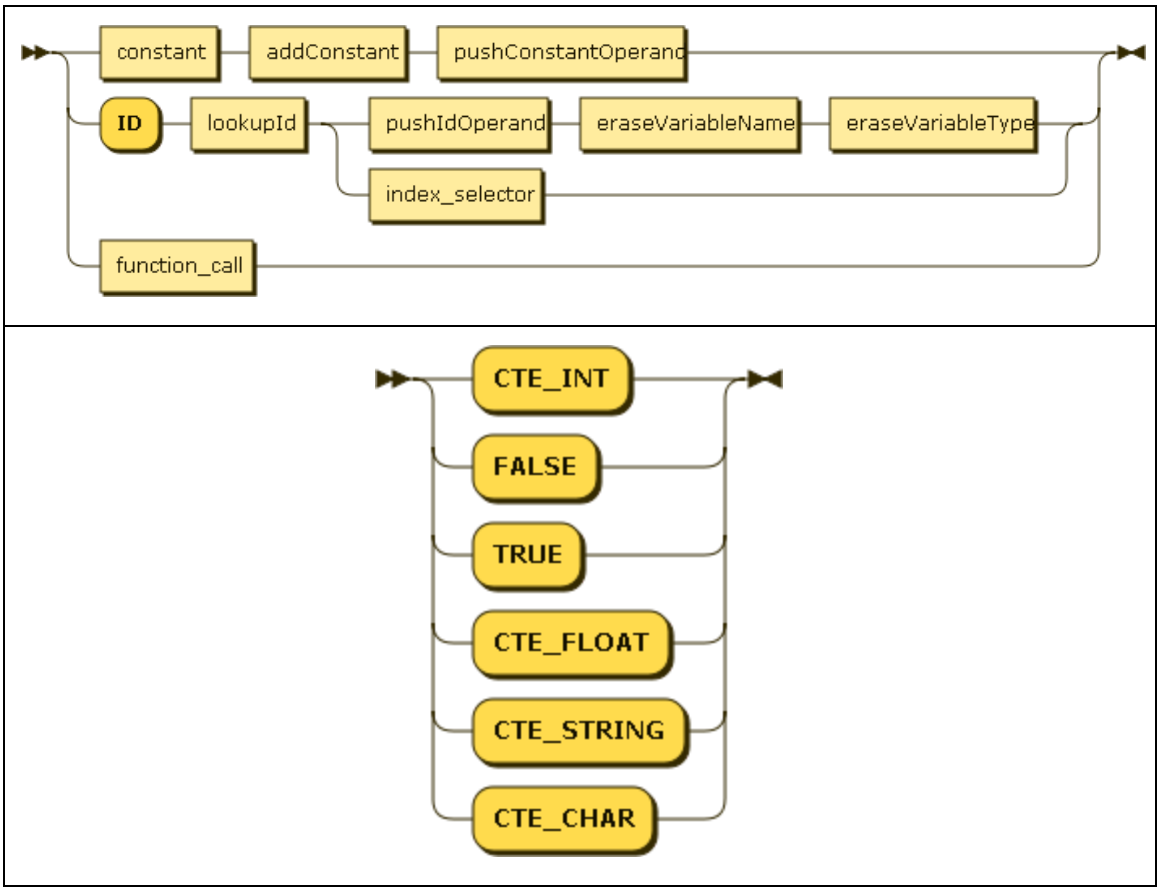



ID
lookupId
pushIdOperand
eraseVariableName
eraseVariableType
=
pushOperation
super_expression
addAssignmentQuadruple
var_array_dimension
generateDimensionSpace
empty
[
CTE_INT
addDimension
]
empty
ID
lookupId
index_selector
=
pushOperation
super_expression
addAssignmentQuadruple
saveVariableDimension
[
addFakeBottom
exp
removeFakeBottom
validateArrayIndex
]
offsetForDimension
[
addFakeBottom
exp
removeFakeBottom
validateArrayIndex
]
accummulateDisplacement
empty
addAddressBase
eraseVariableName
eraseVariableType
eraseVariableDimension
FUNCTION
setLocalScope
type
saveFunctionType
ID
saveFunctionName
(
parameters_definition
)
addFunction
{
block_repeater
}
endFunction
setGlobalScope
,
type
ID
addParameter
empty
,
super_expression
addArgument
empty


RETURN
returnFunction
super_expression
returnFunctionValue
ID
(
lookupFunction
startFunctionCall
addFakeBottom
parameters
removeFakeBottom
)
verifyArguments
endFunctionCall
)
block
empty
vars
var_free_assignment
var_arr_free_assignment
if
cycle
return
BREAK
CONTINUE
output
input
function_call


OUTPUT
(
super_expression
addOutputQuadruple
output1
)
OUTPUTLN
(
super_expression
addOutputQuadruple
output1
addNewLineQuadruple
)
addNewLineQuadruple
,
super_expression
addOutputQuadruple
empty
INPUT
(
ID
lookupId
pushIdOperand
eraseVariableName
eraseVariableType
)
addInputQuadruple
index_selector
IF
(
super_expression
)
ifConditional
{
block_repeater
}
endIfConditional
else
ELSE
elseConditional
else_if
{
elseConditional
block_repeater
}
endIfConditional
IF
(
super_expression
)
ifConditional
{
block_repeater
}
endElseIfConditional
else_if_else
ELSE
elseIfConditional
else_if
{
elseIfConditional
block_repeater
}
endIfConditional
WHILE
startLoop
(
super_expression
)
loopConditional
{
block_repeater
}
endLoop
pushOperation
AND
OR
expression
tryLogicalQuadruple


pushOperation
>
<
EQ
UNEQ
LESS_EQ
GREATER_EQ
exp
tryRelationalQuadruple
pushOperation
+
-
term
tryAddSubQuadruple
pushOperation
*
/
uminus
tryMultDivQuadruple
-
factor
generateUMinusQuadruple
factor
(
addFakeBottom
super_expression
removeFakeBottom
)
value



Semantic Actions Description

1. **setLocalScope**: Sets the scope to local
2. **setGlobalScope**: Sets the scope to global

Variable/Constant Actions

3. **saveVariableType**: Stores the variable type in the temporary stacks
4. **eraseVariableType**: Pops a variable type from the temporary stacks
5. **saveVariableName**: Stores the variable id in the temporary stacks
6. **eraseVariableName**: Pops a variable id from the temporary stacks
7. **saveVariableDimension**: Stores the index dimension in the temporary stacks
8. **eraseVariableDimension**: Pops the index dimension from the temporary stacks
9. **addVariable**: Sends variable data to variable creation method
10. **addConstant**: Sends constant data to constant creation method

11. **lookupId**: Searches the variable symbol table for the ID
12. **addNewLineQuadruple**: Creates a simple newline quadruple

Array Actions

13. **addDimension**: Adds a dimension to a looked up variable, making it an array
14. **generateDimensionSpace**: Generates dimension addresses based on total dimension space in variable
15. **validateArrayIndex**: Generates index validation quadruple
16. **offsetForDimension**: Generates offset quadruple for multiple dimensioned arrays
17. **accummulateDisplacement**: Generates displacement quadruple for array index address
18. **addAddressBase**: Adds variable base address quadruple for array index address

Expression Actions

19. **pushOperation**: Pushes operation to *operationStack*
20. **pushIdOperand**: Pushes variable address to *operandStack*
21. **pushConstantOperand**: Pushes constant address to *operandStack*
22. **tryLogicalQuadruple**: Tries to generate a logical expression
23. **tryRelationalQuadruple**: Tries to generate a relational expression
24. **tryAddSubQuadruple**: Tries to generate a add/sub expression
25. **tryMultDivQuadruple**: Tries to generate a mult/div expression
26. **generateUMinusQuadruple**: Generates 'uminus' quadruple
27. **addFakeBottom**: Adds the *fake* bottom to *operationStack*
28. **removeFakeBottom**: Removes the *fake* bottom from operaitonStack
29. **addAssignmentQuadruple**: Generates assignment quadruple

Conditional Actions

30. **ifConditional**: Generates GOTOF quadruple, adds location to jumpStack
31. **endIfConditional**: Updates last jump position to current position
32. **elseConditional**: Generates a GOTO quadruple, updates last jump position and adds current position to jump stack
33. **elseIfConditional**: Updates last 2 jump positions and generates GOTO quadruple, adds current position to jumpStack

34. **endElseIfConditional**: Updates last 2 jump positions to current position
35. **startLoop**: Pushes current position to jumpStack
36. **loopConditional**: Generates GOTOF quadruple, adds location to jumpStack (same as *ifConditional*)
37. endLoop: Sets jump to start of loop and then updates conditional to end of loop

Function Actions

38. **saveFunctionType**: Save function type to temporary variable
39. **saveFunctionName**: Save function id to temporary variable
40. **endFunction**: Clears all local data, sets the functions address limit, generates ENDPROC
41. **addParameter**: Creates variable in local scope, appends parameter to temporary list
42. **addFunction**: Adds SKIP quadruple with pos to *jumpStack*, creates function object
43. **lookupFunction**: Lookup function in *functionTable*, adds name and type to temporary stacks
44. **startFunctionCall**: Generates ERA quadruple, sets argument count to 0 in temporarry stack
45. **addArgument**: Adds verifies parameter type and adds PARAM quadruple, increases arg count
46. **verifyArguments**: Verifies the correct number of argument
47. **endFunctionCall**: Generate GOSUB quadruple with return a temporary address if needed, clear all function call temporary stacks
48. **returnFunctionValue**: Generates RETURN quadruple with return value
49. **returnFunction**: Generates RETURN quadruple w/o return value (void)
50. **addOutputQuadruple**: Generate OUTPUT quadruple
51. **addInputQuadruple**: Generate INPUT quadruple (with address)

Semantic Table

Addition (+)				
	bool	int	float	char

bool	ERROR	ERROR	ERROR	ERROR
int		int	float	string
float			float	string
char				string

Subtraction, Multiplication, Division, Modular (-, *, /, %)				
	bool	int	float	char
bool	ERROR	ERROR	ERROR	ERROR
int		int	float	ERROR
float			float	ERROR
char				ERROR

Relational Operation (<, >, <=, >=, ==, !=)				
	bool	int	float	char
bool	ERROR	ERROR	ERROR	ERROR
int		bool	bool	ERROR
float			bool	ERROR
char				ERROR

Logical Operations (AND, OR)				
	bool	int	float	char
bool	bool	ERROR	ERROR	ERROR
int		ERROR	ERROR	ERROR
float			ERROR	ERROR
char				ERROR

Memory Administration Process in Compilation Phase

Graphic Specification of Every Data Structure Used

Memory management during the compilation phase consists in storing data in Python custom object files. Some temporary data is held in Stacks to keep track of order when parsing the program. We use enumerators to keep track of types. Dictionaries to manage variables and functions. And most importantly the quadruple list.

Generally speaking there are three types of data kept track on the compiler phase. These use similar “tables” based on dictionaries. These are used by Constants, Variables and Functions objects which each have their own classes as well. At it’s very bone these tables have two important methods, *insert* and *lookup:*

- Insert: will lookup for the variable in the dictionary and if it does not exist it will create it. If it does exist, it will show error to user.
- Lookup: Will look for the variable in the dictionary and return it if it does exist.

For example the SymbolTable (used by variables and constants):

```
class SymbolTable:
    def __init__(self):
        self.symbols = {}
    def insert(self, variable):
        if self.lookup(variable.name) is not None:
            print "Symbol error : ", variable.name, "is already declared"
        else:
            self.symbols[variable.name] = variable
            return self.symbols[variable.name]
        return 0

    def lookup(self, name):
        if name in self.symbols:
            return self.symbols[name]
        else:
            # Symbols does not exist
            return None

    # ... print, size, and other helper methods ...
```

The dictionary holds the variables/constants using their identifiers for keys.

A variable and constant object look very similar with only some attributes set differently. Here are their classes:

```
class Var:
    def __init__(self, name, symbolType):
        self.id = -1
        self.name = name
        self.symbolType = symbolType
        self.dimensions = []

    # ... and other helper methods ...

class Constant:
    def __init__(self, id, name, symbolType, value):
        self.id = id
        self.name = name
        self.symbolType = symbolType
        self.value = value

    # ... and other helper methods ...
```

Functions are similar but carry parameter types and data, they still use their own table:

```
class Function:
    def __init__(self, name, functionType, parameters, position):
        self.name = name
        self.functionType = functionType
        self.parameters = parameters
        self.quadruplePosition = position
        self.limits = []

    def parametersSize(self):
        return len(self.parameters)

    # ... and other helper methods ...
```

Quadruple objects contain: *operator, operand1, operand2, result*. They are stored in a list which is later saved to the OBJ file. Here is the very Quadruple class and it's parent class with a list:

```
class Quadruple:
    def __init__(self, operator, operand1, operand2, result):
        self.operator = operator
        self.operand1 = operand1
        self.operand2 = operand2
        self.result = result

    # ... and other helper methods ...

class QuadrupleList:
    def __init__(self):
        self.list = []
```

```
    def add(self, quadruple):
        self.list.append(quadruple)

    def lookup(self, index):
        return self.list[index]

    # ... and other helper methods ...
```

Also used is the **AddressSystem** which is a very imporant part of the Quadruple generation phase. This systems job is to generate and maintain the addresses used in Quadruples. These are usually *Local*, *Temporary, Global* and *Constant* addresses. The Local and Temporary addresses have special usage. These are reset every time there is a new local scope.

```
class AddressSystem:
  def __init__(self):
    self.tempPointer = {}
    self.gtempPointer = {}
    self.localPointer = {}
    self.globalPointer = {}
    self.constantPointer = {}

    # Generate Memory Maps
    self.space = 10000 # memory space per type
    for t in Type:
      if t is Type.VOID:
        continue
      self.tempPointer[t] = t.value * self.space
      self.gtempPointer[t] = t.value * self.space
      self.localPointer[t] = (t.value + 5) * self.space
      self.globalPointer[t] = (t.value + 10) * self.space
      self.constantPointer[t] = (t.value + 15) * self.space

    self.memory = [None]*self.space*20

  def generateTemporary(self, variableType, size=1):
    if variableType is not None:
      return self.__generateAddress(self.tempPointer, variableType, size)
    else:
      print("Memory error: variable type not found")
      return None

  # ... and other generation and helper methods ...

  def __generateAddress(self, memoryMap, variableType, size):
    address = memoryMap[variableType]
    memoryMap[variableType] += size
    return address
```

Virtual Machine Description

Memory Administration Process in Execution Phase

Graphic Specification of every Data Structure used

During the start of the execution phase the first thing that is done is initialization of memory and parsing of OBJ file. The data retrieved from OBJ file is Quadruples, Constants and Functions. Firstly the limits from the AddressSystem are imported. Then Quadruples are imported and saved on the same QuadrupleList class as in compilation phase. After that, constants are saved into the MemorySystem. Finally the FunctionTable is filled up, this is also the same one used in the compilation phase.

The most important classes are the MemorySystem and ActivationRecord classes. First the MemorySystem is initialized with the ***Global*** and ***Constant*** limits imported from the AddressSystem. This basically tell us the number of space each type of data needs per scope. Inside the MemorySystem we also have the control stack, which holds ActivationRecord. These are instance of memories for each instance of a method. ActivationRecord hold ***Temporary*** and ***Local*** data. Note: there is one ActivationRecord created initially to hold ***Temporary*** data in the Global scope.

Here is the initialization of MemorySystem.

```
class MemorySystem():

  def __init__(self, limits):
    self.gtempLimit = limits[0]
    self.localLimit = limits[1]
    self.globalLimit = limits[2]
    self.constantLimit = limits[3]

    self.space = 10000 # memory space per type

    # Global Memory
    self.globalMemory = range(5)
    for t in Type:
      if t is Type.VOID:
        continue
      self.globalMemory[t.value] = [None]* (self.globalLimit[t] % self.space)

    # Constant Memory
    self.constantMemory = range(5)
```

```
    for t in Type:
      if t is Type.VOID:
        continue
      self.constantMemory[t.value] = [None]* (self.constantLimit[t] % self.space)

    # Holds ActivationRecord
    self.controlStack = Stack()
    self.callStack = Stack()
    self.controlStack.push(ActivationRecord([self.gtempLimit, self.localLimit]))

  # ... rest of class ...
```

Values are accessed with:

```
  # ... declaration of class ...

  def getValue(self, address):
    if address[0] == '(':
      pAddress = self.getValue(address[1:-1])
      return self.getValue(str(pAddress))
    address = self.__validateAddress(address)
    scope = self.__getAddressScope(address)
    aType = self.__getAddressType(address)
    address = address % self.space

    if scope is Scope.GLOBAL:
      for t in Type:
        if aType is t:
          return self.__safeValue(self.globalMemory[t.value][address], t)

    elif scope is Scope.CONSTANT:
      for t in Type:
        if aType is t:
          return self.__safeValue(self.constantMemory[t.value][address], t)

    elif scope is Scope.TEMPORARY or scope is Scope.LOCAL:
      return self.__safeValue(self.controlStack.top().getValue(address, aType,
scope), aType)

  # ... rest of class ...
```

And values are set with:

```
  # ... declaration of class ...

  def setValue(self, address, value):
    if address[0] == '(':
      pAddress = self.getValue(address[1:-1])
      self.setValue(str(pAddress), value)
      return
    address = self.__validateAddress(address)
    scope = self.__getAddressScope(address)
    aType = self.__getAddressType(address)
    address = address % self.space

    if scope is Scope.GLOBAL:
      for t in Type:
        if aType is t:
```

```
          self.globalMemory[t.value][address] = value
          return

    elif scope is Scope.CONSTANT:
      for t in Type:
        if aType is t:
          self.constantMemory[t.value][address] = value
          return

    elif scope is Scope.TEMPORARY or scope is Scope.LOCAL:
      self.controlStack.top().setValue(address, value, aType, scope)

  # ... rest of class ...
```

Similarly the ActivationRecord has the same *setValue()* and *getValue()* functions, but these are called only from MemorySystem's *setValue()* and *getValue()* functions.

```
class ActivationRecord():

  def __init__(self, limits):
    self.tempLimit = limits[0]
    self.localLimit = limits[1]

    self.parameters = []
    self.callPosition = 0
    self.returnAddress = None

    self.space = 10000

    # Temporary Memory
    self.tempMemory = range(5)
    for t in Type:
      if t is Type.VOID:
        continue
      self.tempMemory[t.value] = [None]* (self.tempLimit[t] % self.space)

    # Local Memory
    self.localMemory = range(5)
    for t in Type:
      if t is Type.VOID:
        continue
      self.localMemory[t.value] = [None]* (self.localLimit[t] % self.space)

  def setValue(self, address, value, aType, scope):
    if scope is Scope.TEMPORARY:
      for t in Type:
        if aType is t:
          self.tempMemory[t.value][address] = value
          return

    elif scope is Scope.LOCAL:
      for t in Type:
        if aType is t:
          self.localMemory[t.value][address] = value
          return

  def getValue(self, address, aType, scope):
```

```
if scope is Scope.TEMPORARY:
  for t in Type:
    if aType is t:
      return self.tempMemory[t.value][address]

elif scope is Scope.LOCAL:
  for t in Type:
    if aType is t:
      return self.localMemory[t.value][address]
```

Association between Virtual(compilation) and Real Addresses

During the compilation phase addresses are assigned to variables and set directly in the quadruples with no value yet. So during this phase the address is really just a way create a distinction from other values. The addresses range from 0-20000 (adjustable), covering 4 different scopes (global, constant, local, temporary) and 5 different types per scope, with each type holding up to 1000 values. Once the addresses are read on execution you could have a variable with address 8001 and this does not mean you have 8000 other variables. The range in the address also symbolizes the scope and type of value the address should carry.

The first data imported in the execution phase is the address *limits* which are the last variable generated per type and scope. With these values the memory is able to generate the appropriate amount of spaces it should generate:

```
8001 % 1000 == 1
# there is 1 space needed for this type
```

Additionally each imported *function* also includes it's local and temporary address limits. These are used when adding new ActivationRecords to the control stack. When a ActivationRecord is created, it is created with its defined limits and therefore is able to generate the correct amount of memory spaces needed.

So finally, when an operation is performed by the virtual machine, the address read is parsed to obtain the the type and scope of the value and return or set the value from the simulated memory *list*.

Language Functionality Test

Fibonacci

```
craft {

 var int cache[100], index = 0
 while(index < 100) {
   cache[index] = -1
   index = index + 1
 }
 cache[0] = 0
 cache[1] = 1

 function int fibonacciRecursiveCached(int n) {
   if(cache[n] >= 0) {
     return cache[n]
   } else {
     cache[n] = fibonacciRecursiveCached(n - 1) + fibonacciRecursiveCached(n - 2)
     return cache[n]
   }
 }

 function int fibonacciRecursive(int n) {
   if(n == 0) {
     return 0
   } else if (n == 1) {
     return 1
   } else {
     return fibonacciRecursive(n - 1) + fibonacciRecursive(n - 2)
   }
 }

 function int fibonacciIterative(int n) {
   var int x=0, y=1, z=1, i=0
   while (i < n) {
     x = y
     y = z
     z = x + y
     i = i + 1
   }
   return x
 }

 outputln("Fibonnacci Recursive Cached: ", fibonacciRecursiveCached(50))
 outputln("Fibonnacci Iterative: ", fibonacciIterative(50))
 outputln("Fibonnacci Recursive: ", fibonacciRecursive(20))
}
```

Intermediary Code - https://pastebin.com/Qapn5QYZ (full)

```
0 . [        =,       160000,      None,    110100]
1 . [        <,       110100,    160001,         0]
2 . [    GOTOF,            0,      None,        10]
3 . [      VER,       110100,         0,       100]
```

```
4 . [     ACUM,       110100,    110000,   (10000)]
5 . [        *,       160003,    160002,     10001]
6 . [        =,        10001,      None,   (10000)]
7 . [        +,       110100,    160002,     10002]

# ...

85. [    PARAM,       160005,      None,         0]
86. [    GOSUB, fibonacciIte,      None,     10006]
87. [   OUTPUT,         None,      None,     10006]
88. [ OUTPUTLN,         None,      None,      None]
89. [   OUTPUT,         None,      None,    190002]
90. [      ERA, fibonacciRec,      None,      None]
91. [    PARAM,       160006,      None,         0]
92. [    GOSUB, fibonacciRec,      None,     10007]
93. [   OUTPUT,         None,      None,     10007]
94. [ OUTPUTLN,         None,      None,      None]
```

Result

```
Fibonnacci Recursive Cached: 12586269025
Fibonnacci Iterative: 12586269025
Fibonnacci Recursive: 6765

% Execution ended, total time: 3.964301s
```

Factorial

```
craft {

  function int factorialRecursive (int n) {
    if (n <= 1) {
      return 1
    }
    return n * factorialRecursive(n - 1)
  }

  function int factorialIterative (int n){
    var int i = 2, sum = 1
    while (i <= n) {
      sum = sum * i
      i = i + 1
    }
    return sum
  }

  outputln("Factorial Iterative: ", factorialIterative(15))
  outputln("Factorial Recursive: ", factorialRecursive(15))
}
```

Intermediary Code

```
0 . [     SKIP,         None,      None,        11]
1 . [       <=,        60000,    160000,         0]
2 . [    GOTOF,            0,      None,         4]
3 . [   RETURN,         None,      None,    160000]
4 . [      ERA, factorialRec,      None,      None]
5 . [        -,        60000,    160000,     10000]
6 . [    PARAM,        10000,      None,         0]
7 . [    GOSUB, factorialRec,      None,     10001]
8 . [        *,        60000,     10001,     10002]
9 . [   RETURN,         None,      None,     10002]
10. [  ENDPROC,         None,      None,      None]
11. [     SKIP,         None,      None,        23]
12. [        =,       160001,      None,     60001]
13. [        =,       160000,      None,     60002]
14. [       <=,        60001,     60000,         0]
15. [    GOTOF,            0,      None,        21]
16. [        *,        60002,     60001,     10000]
17. [        =,        10000,      None,     60002]
18. [        +,        60001,    160000,     10001]
19. [        =,        10001,      None,     60001]
20. [     GOTO,         None,      None,        14]
21. [   RETURN,         None,      None,     60002]
22. [  ENDPROC,         None,      None,      None]
23. [   OUTPUT,         None,      None,    190000]
24. [      ERA, factorialIte,      None,      None]
25. [    PARAM,       160002,      None,         0]
26. [    GOSUB, factorialIte,      None,     10000]
27. [   OUTPUT,         None,      None,     10000]
28. [ OUTPUTLN,         None,      None,      None]
29. [   OUTPUT,         None,      None,    190001]
30. [      ERA, factorialRec,      None,      None]
31. [    PARAM,       160002,      None,         0]
32. [    GOSUB, factorialRec,      None,     10001]
33. [   OUTPUT,         None,      None,     10001]
34. [ OUTPUTLN,         None,      None,      None]
```

Result

```
Factorial Iterative: 1307674368000
Factorial Recursive: 1307674368000

% Execution ended, total time: 0.005753s
```

Merge Sort

```
craft {

  # ================================
  # Merge Sort
  # ================================
  var int a[50], array_size = 20
```

```
function void generateArray() {
  var int index = 0, x = 33
  while (index < array_size) {
    a[index] = x
    index = index + 2
    x = x - 2
  }

  index = 1
  while (index < array_size) {
    a[index] = index
    index = index + 2
  }
  index = 1
}

function void printArray() {
  var int c = 0
  while (c < array_size) {
    output(a[c]," ")
    c = c + 1
  }
  outputln("")
}

function void merge(int low, int mid, int high) {
  var int h = low, i = low, j = mid + 1, b[50], k

  while (h <= mid and j <= high) {
    if (a[h] <= a[j]) {
      b[i] = a[h]
      h = h + 1
    } else {
      b[i] = a[j]
      j = j + 1
    }
    i = i + 1
  }

  if (h > mid) {
    k = j
    while (k <= high) {
      b[i] = a[k]
      i = i + 1
      k = k + 1
    }
  } else {
    k = h
    while (k <= mid) {
      b[i] = a[k]
      i = i + 1
      k = k + 1
    }
  }

  # outputln(low,  " <> ",high)
  k = low
  while (k <= high) {
    # outputln(b[k])
    a[k] = b[k]
```

```
      k = k + 1
    }
  }

  function void merge_sort(int low, int high) {
    var int mid
    if (low < high) {
      mid = low + (high - low) / 2
      merge_sort(low, mid)
      merge_sort(mid + 1, high)
      merge(low, mid, high)
    }
  }

  outputln("===== Merge Sort =====")
  generateArray()
  outputln("Unsorted Array:")
  printArray()
  merge_sort(0, array_size - 1)
  outputln("Sorted Array:")
  printArray()
}
```

Intermediary Code - https://pastebin.com/mUQaEy0q (full)

```
0 . [        =,       160000,      None,    110050]
1 . [     SKIP,         None,      None,        25]
2 . [        =,       160001,      None,     60000]
3 . [        =,       160002,      None,     60001]
4 . [        <,        60000,    110050,         0]
5 . [    GOTOF,            0,      None,        14]
6 . [      VER,        60000,         0,        50]
7 . [     ACUM,        60000,    110000,   (10000)]
8 . [        =,        60001,      None,   (10000)]
9 . [        +,        60000,    160003,     10001]
10. [        =,        10001,      None,     60000]
11. [        -,        60001,    160003,     10002]

# ...

146. [    PARAM,        10000,      None,         1]
147. [    GOSUB,   merge_sort,      None,      None]
148. [   OUTPUT,         None,      None,    190003]
149. [ OUTPUTLN,         None,      None,      None]
150. [      ERA,   printArray,      None,      None]
151. [    GOSUB,   printArray,      None,      None]
```

Result

```
===== Merge Sort =====
Unsorted Array:
33 1 31 3 29 5 27 7 25 9 23 11 21 13 19 15 17 17 15 19
Sorted Array:
1 3 5 7 9 11 13 15 15 17 17 19 19 21 23 25 27 29 31 33

% Execution ended, total time: 0.089740s
```

Simple Arithmetic

```
 craft {
  var bool test
  var int A = 2 * -(1 + 1 / -3)
  outputln(-(1 + 1 / -3))
  var float B = 3 / -2.0

  output(A + B)
}
```

Intermediary Code

```
0 . [        *,       160003,    160002,     10000]
1 . [        /,       160001,     10000,     10001]
2 . [        +,       160001,     10001,     10002]
3 . [        *,       160003,     10002,     10003]
4 . [        *,       160000,     10003,     10004]
5 . [        =,        10004,      None,    110000]
6 . [        *,       160003,    160002,     10005]
7 . [        /,       160001,     10005,     10006]
8 . [        +,       160001,     10006,     10007]
9 . [        *,       160003,     10007,     10008]
10. [   OUTPUT,         None,      None,     10008]
11. [ OUTPUTLN,         None,      None,      None]
12. [        /,       160002,    170000,     20000]
13. [        =,        20000,      None,    120000]
14. [        +,       110000,    120000,     20001]
15. [   OUTPUT,         None,      None,     20001]
```

Result

```
-1
-3.5
% Execution ended, total time: 0.000602s
```

If elseif else

```
craft {
  var int A = 2, B = 3, C = 4, D = 5

  if (A + B > D) {
    D = B + A * C
  } else if (A < B) {
    A = 0
    B = B + D
  } else if (B >= A) {
    D= B
  } else {
    D = A + C
  }
```

```
  outputln("A: ", A)
  outputln("B: ", B)
  outputln("C: ", C)
  outputln("D: ", D)
}
```

Intermediary Code

```
0 . [        =,       160000,      None,    110000]
1 . [        =,       160001,      None,    110001]
2 . [        =,       160002,      None,    110002]
3 . [        =,       160003,      None,    110003]
4 . [        +,       110000,    110001,     10000]
5 . [        >,        10000,    110003,         0]
6 . [    GOTOF,            0,      None,        11]
7 . [        *,       110000,    110002,     10001]
8 . [        +,       110001,     10001,     10002]
9 . [        =,        10002,      None,    110003]
10. [     GOTO,         None,      None,        16]
11. [        <,       110000,    110001,         1]
12. [    GOTOF,            1,      None,        17]
13. [        =,       160004,      None,    110000]
14. [        +,       110001,    110003,     10003]
15. [        =,        10003,      None,    110001]
16. [     GOTO,         None,      None,        20]
17. [       >=,       110001,    110000,         2]
18. [    GOTOF,            2,      None,        21]
19. [        =,       110001,      None,    110003]
20. [     GOTO,         None,      None,        23]
21. [        +,       110000,    110002,     10004]
22. [        =,        10004,      None,    110003]
23. [   OUTPUT,         None,      None,    190000]
24. [   OUTPUT,         None,      None,    110000]
25. [ OUTPUTLN,         None,      None,      None]
26. [   OUTPUT,         None,      None,    190001]
27. [   OUTPUT,         None,      None,    110001]
28. [ OUTPUTLN,         None,      None,      None]
29. [   OUTPUT,         None,      None,    190002]
30. [   OUTPUT,         None,      None,    110002]
31. [ OUTPUTLN,         None,      None,      None]
32. [   OUTPUT,         None,      None,    190003]
33. [   OUTPUT,         None,      None,    110003]
34. [ OUTPUTLN,         None,      None,      None]
```

Result

```
A: 0
B: 8
C: 4
D: 5

% Execution ended, total time: 0.000626s
```

Code Fragments

Relevant snippets of code

Memory addresses parsing. This is used when getting a value and setting a value.

```
  # ... rest of memory system ...

  def __validateAddress(self, address):
    if not isinstance(address, int):
      address = int(address)
    return address

  def __getAddressType(self, address):
    address = address % (self.space * 5)
    for t in Type:
      if t is Type.VOID:
        continue
      if address >= t.value * self.space and address < t.value * self.space +
self.space:
        return t
    print "Memory error: address not in type bounds"
    return None

  def __getAddressScope(self, address):
    if address < self.space * 5:
      return Scope.TEMPORARY
    elif address >= self.space * 5 and address < self.space * 10:
      return Scope.LOCAL
    elif address >= self.space * 10 and address < self.space * 15:
      return Scope.GLOBAL
    elif address >= self.space * 15 and address < self.space * 20:
      return Scope.CONSTANT
    else:
      print "Memory error: address out of scope"
      return None

  # ... rest of memory system ...
```

The OBJ file generation. We use a CSV type file to easily import and export the data.

```
def export(filename):
  with open(filename, 'wb') as fp:
    writer = csv.writer(fp, delimiter=',')
    # Export Pointer Limits
    writer.writerow(limitDictToArray(__address.gtempPointer))
    writer.writerow(limitDictToArray(__address.localPointer))
    writer.writerow(limitDictToArray(__address.globalPointer))
    writer.writerow(limitDictToArray(__address.constantPointer))
    writer.writerow(['END', "MEM_LIMITS"])
```

```
    # Export Quadruples
    for q in __quadruples.list:
      writer.writerow([q.operator, q.operand1, q.operand2, q.result])
    writer.writerow(['END', "QUADRUPLES"])
    # Export Main Memory
    for k, c in __constantTable.symbols.iteritems():
      writer.writerow([c.id, c.value])
    writer.writerow(['END', "CONSTANTS"])
    # Export Function Table
    for f in __funcsGlobal.functions.itervalues():
      writer.writerow([f.name, f.functionType.value, len(f.parameters),
f.quadruplePosition])
      writer.writerow(limitDictToArray(f.limits[0]))
      writer.writerow(limitDictToArray(f.limits[1]))
      for v in f.parameters:
        writer.writerow([v.id, v.name, v.symbolType.value])
    writer.writerow(['END', "FUNCTIONS"])
```

Constants generation. We use the parser to detect the data type that belongs to the constant and easily generate.

```
def p_constant_int(p):
  "constant : CTE_INT"
  p[0] = int(p[1])

def p_constant_boolean(p):
  '''constant : FALSE
              | TRUE'''
  if p[1] == 'false':
      p[0] = False
  elif p[1] == 'true':
      p[0] = True

def p_constant_float(p):
  "constant : CTE_FLOAT"
  p[0] = float(p[1])

def p_constant_strings(p):
  '''constant : CTE_STRING
              | CTE_CHAR '''
  p[0] = str(p[1])[1:-1].decode('string_escape')

# ...

def addConstant(const):
  global __constantTable
  # Scan raw
  if __constantTable.lookup(str(const)) is None:
    if isinstance(const, bool):
      constType = Type.BOOL
      constValue = const
    elif isinstance(const, int):
      constType = Type.INT
      constValue = const
    elif isinstance(const, float):
      constType = Type.FLOAT
      constValue = const
```

```
    elif isinstance(const, str) and len(const) == 1:
      constType = Type.CHAR
      constValue = const
    else:
      constType = Type.STRING
      constValue = const
    # Create constant
    cid = __address.generateConstant(constType)
    # Save Constant to table
    __constantTable.insert(Constant(cid ,str(const), constType, constValue))
```

Cube type matching. We made the matching extremely simple to understand.

```
# ... rest of cube ...

# ‘and’ and ‘or’
cube['BOOLandBOOL'] = Type.BOOL
cube['BOOLorBOOL'] = Type.BOOL

def getResultType(leftType, operator, rightType):
  key = leftType.name + operator + rightType.name
  if not key in cube.keys():
    print "Expression error : result type mismatch. [", key, "]"
    return None
  else:
    return cube[key]
```

Enumerators. We used enumerators for easy matching,

```
class Type(Enum):
  BOOL = 0
  INT = 1
  FLOAT = 2
  CHAR = 3
  STRING = 4
  VOID = 5
```

Quadruple generation. We generate quadruples all around the grammar actions.

```
# addExpressionQuadruple receives a list of operators
def addExpressionQuadruple(operators):
  operator = __operationStack.top()
  if operator in operators:
    operator = __operationStack.pop()
    # Operands
    rightOp = __operandStack.pop()
    leftOp = __operandStack.pop()
    # Types
    rightType = __typeStack.pop()
```

```
    leftType = __typeStack.pop()
    resultType = getResultType(leftType, operator, rightType)
    if resultType is not None:
      # Generate quadruple
      if __scope == Scope.GLOBAL:
        address =  __address.generateGlobalTemporary(resultType)
      elif __scope == Scope.LOCAL:
        address =  __address.generateTemporary(resultType)
      __quadruples.add(Quadruple(operator, leftOp, rightOp, address))
      # Update stacks
      __operandStack.push(address)
      __typeStack.push(resultType)
    else:
      summary()
      exit(1)
```

Expressions. These the relevant grammar rules.

```
# ============================================================================
# Expressions
#
# Defines expression rules. Follows a recursive precedence starting at
# 'super_expression' (first rule is last quadruple generated):
#
# Logical -> Relational -> Sum/Sub -> Mult/Div -> uMinus -> Parenthesis
# -> value,id,function_calls
#
# ============================================================================

def p_super_expression(p):
  '''super_expression : expression tryLogicalQuadruple
                      | expression tryLogicalQuadruple AND pushOperation
super_expression
                      | expression tryLogicalQuadruple OR pushOperation
super_expression'''

def p_expression(p):
  '''expression : exp tryRelationalQuadruple
                | exp tryRelationalQuadruple '>' pushOperation expression
                | exp tryRelationalQuadruple '<' pushOperation expression
                | exp tryRelationalQuadruple EQ pushOperation expression
                | exp tryRelationalQuadruple UNEQ pushOperation expression
                | exp tryRelationalQuadruple LESS_EQ pushOperation expression
                | exp tryRelationalQuadruple GREATER_EQ pushOperation expression'''

def p_exp(p):
  '''exp : term tryAddSubQuadruple
         | term tryAddSubQuadruple '+' pushOperation exp
         | term tryAddSubQuadruple '-' pushOperation exp '''

def p_term(p):
  '''term : uminus tryMultDivQuadruple
          | uminus tryMultDivQuadruple '*' pushOperation term
          | uminus tryMultDivQuadruple '/' pushOperation term'''

def p_uminus(p):
  '''uminus : '-' factor generateUMinusQuadruple
```

```
            | factor'''

def p_factor(p):
  '''factor : '(' addFakeBottom super_expression removeFakeBottom ')'
            | value'''

def p_value(p):
  '''value : constant addConstant pushConstantOperand
           | ID lookupId pushIdOperand eraseVariableName eraseVariableType
           | ID lookupId index_selector
           | function_call'''
```

User Manual

Codecraft
V1.0

Requirements

Python 2.7.13

To download python head to https://www.python.org/downloads/ and select the version 2.7.13.

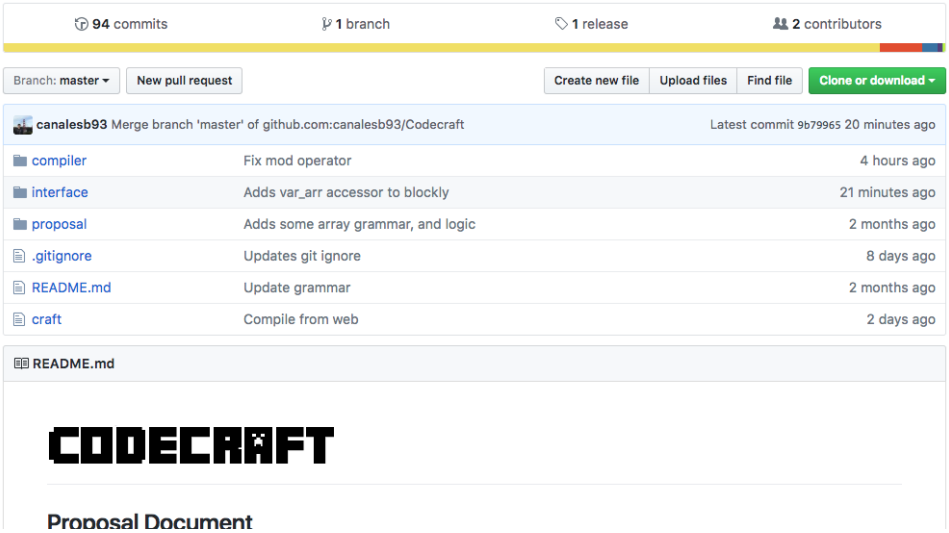
PHP

To download PHP head to http://php.net/downloads.php and download a 7.1 version or higher. PHP is needed for the web interface.

Installation and Setup

Download

The first thing you need to do is head to our github page and download Codecraft files. You can find them at https://github.com/canalesb93/Codecraft.



Press the *"Clone or download"* button and run the link in your terminal at the desired installation directory.

```
$ git clone git@github.com:canalesb93/Codecraft.git
```

Project Directory

After the files have been installed, move into the codecraft directories.

```
$ cd Codecraft/
```

The Codecraft project has three main directories:

- *interface/* - Holds the web application with the graphical input.
- *compiler/* - Holds the compiler files
- *samples/* - Holds example programs

Usage

There are 2 main ways to run the program, through the web interface or directly through your terminal.

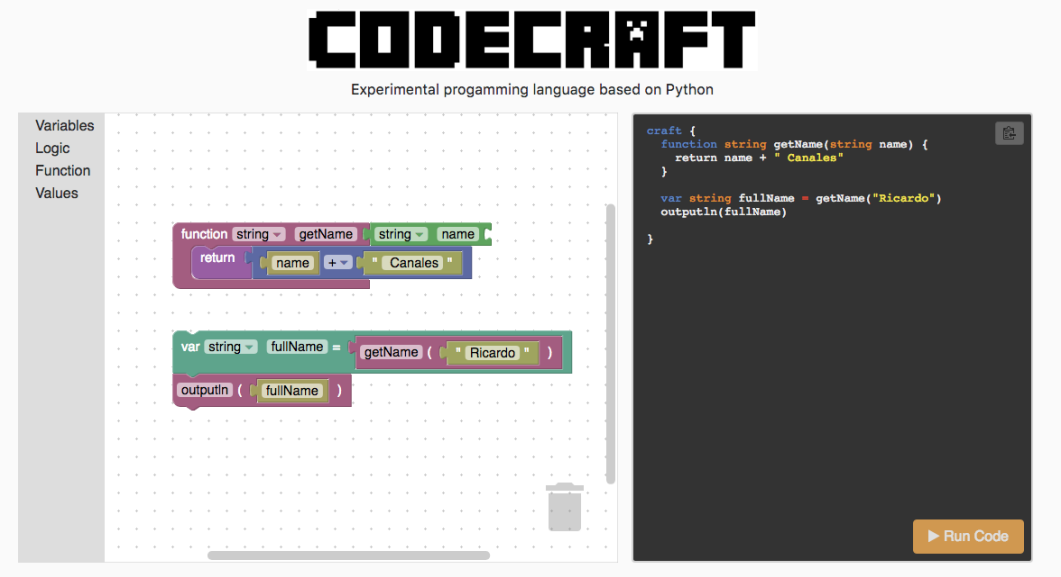
Running Web Interface

To fully run the web interface you must navigate to the *interfaces/* directory and run locally run a file hosting server. We recommend you use PHP for this since it allows platform code execution.

```
$ cd interfaces/
$ php -S localhost:8000
```

This will run the PHP built-in server feature. You can now open your browser and go to “http://localhost:8000”.

Once in the site you can use blockly or the custom text editor provided to run programs.



Running with terminal

There are two ways to execute a program through the terminal.

Bash Script

To run the full execution script you must first give the file permissions. Go to the main project's directory. And run:

```
$ chmod +x craft
```

After that you can run your program with:

```
$ ./craft path/to/program.craft
```

Python Execution

You can also run the compiler by phase. First the compilation phase which generates the "*.crafted*" file. And then the execution phase which runs the generated file.

From the main project directory run this to compile:

```
$ python compiler/parser.py path/to/program.craft
```

This will generate the "*.crafted*" file and show some information about the compilation. You must then give to the file to the execution script.

```
$ python compiler/virtual_machine.py path/to/program.crafted
```

Quick Reference Sheet

Codecraft Ver1.00 - May 2, 2017

Operator Precedence Table

func_name(*args*, …)	Function Call
(...)	Parenthesis
*, /, %	Multiply, divide, mod
+, -	Add, subtract
>, <, <=, >=, !=, ==	Relational
and, or	Logical

Common Syntax Structures

Assignment Statement
var TYPE *name* = exp

Selection
```
if (boolean_exp) {
     Stmt…
} else if (boolean_exp) {
     Stmt…
} else {
     Stmt…
}
```

Cycle
```
while(boolean_exp){
   Stmt …
}
```

Function Definition
```
function TYPE func_name(TYPE name){
     Stmt …
}
```

Function Call
```
func_name(args, ...)
```

Arrays Definition
```
var TYPE name[index]...[index]
```

Common Data Types

Type	Description	Literal Ex.
int	32 bit integer	-1, 10
float	Floating point number	3.33, -5.4
bool	boolean	True, false
char	Single character	‘A’, ‘z’
string	Character sequence	“codecraft”

Built-in Function

output(*args*, ...)	Prints all data types
outputln(*args*, …)	Prints all data types and a newline
input(*var*)	Sets the value of the variable

File definition

Program file	*.craft*
Compiled file	*.crafted*

Program
```
craft {
     Stmt…
     Func…
}
```

Resources

- https://github.com/canalesb93/Codecraft
- http://codecraft.canalesb.com

By **Ricardo Canales** and **Gabriel Berlanga**