|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Team 3   |  |  |  | | --- | --- | --- | | Name | Role | UNI | | Morris Hopkins | Project Manager | mah2250 | | Seungwoo Lee | Language Guru | sl3492 | | Lev Brie | System Architect | ldb2001 | | Alexandros Sigaras | System Integrator | as4161 | | Michal Wolski | Verification and Validation | mtw2135 | |

|  |
| --- |
| COMS W4115 PROGRAMMING LANGUAGES AND TRANSLATORS |
| SWIM |
| Language Reference Manual |



Table of Contents

1. Introduction 1

2. Lexical Conventions 1

2.1. Comments 1

2.2. Identifiers 2

2.3. Keywords 2

2.4 Escape Sequences 2

3. Notation and Punctuation 3

3.1. Syntax Notation 3

3.2. Punctuation 4

4. Types 4

4.1. Basic Types 4

4.1.1. Integers 4

4.1.2. Long Integers 4

4.1.3. Float Point Numbers 4

4.1.4. Boolean Values 4

4.1.5. Strings 5

4.1.6. None 5

4.2. Derived Types 5

4.2.1. Lists 5

4.2.2. Dictionaries 6

4.2.3. Objects 6

4.3. Type Conversion 6

4.4.  Constants and Literals 7

4.4.1 Constants 7

4.4.2 Literals 8

5. Operators 8

5.1.1. Unary Operators 8

5.1.2. Multiplicative Operators 8

5.1.3. Additive Operators 9

5.1.4. Relational Operators 9

5.1.5. Equality Operator 9

5.1.6. Assignment Expressions 9

6. Statements 10

6.1.  Expression-statement 10

6.2 Compound Statements 10

6.3 Selection Statements 11

6.4 Iteration Statements 11

6.4.1. While 11

6.4.2. For Each 12

7. Functions 12

7.1 Function calls 12

7.2 Function definitions. 12

8. Objects 13

8.1 Object definitions. 13

9. Scope and Namespacing 14

9.1 Lexical Scope 14

9.2 Function Scope 14

9.3 Block Scope 14

9.4 Global Scope 14

9.5 Nested Scope and Scope in General 14

10. File inclusion 15

Appendix. Full grammar 16

# 1. Introduction

This manual describes the first SWIM language specification developed by Lev Brie, Morris Hopkins, Seungwoo Lee, Alexandros Sigaras, and Michal Wolski at Columbia University in the Spring of 2013. Because the Language is still underdevelopment this document can be understood as a rough draft outlining the standard. That is, this document is not as thorough as a typical language standard, and the standard, if created, may reflect changes not present in this document.  For the most part the lexical and syntactic conventions outlined here are accurate, and the grammar will require minimal changes to reflect the finalized grammar of the production language.

Commentary material in this manual will be indented and written in smaller type.  For the most part commentary material will be limited to notations indicating facets of the language still under development, where known ambiguity exists, or where specific changes are still expected to occur.

# 2. Lexical Conventions

A Swim program is composed of one or more files that are translated in several phases.  The initial phase of translation is lexical tokenization in which the program is split into tokens separated by whitespace characters; comments (defined 2.2) are discarded.

## 2.1. Comments

The SWIM language is designed to be easy for novices and predictable for programmers.  Comments follow the same format found in other languages such as C and Java.  Block comments begin with /\* and end with \*/.  Block comments may span multiple lines.  Line comments begin with // and extend to the following newline character.  Comments may occur within a block of code but not within identifiers, numeric literals, string literals, or other cases where the comment would obfuscate semantically significant atomic units.

## 2.2. Identifiers

An Identifier is a sequence of letters, numbers, and underscore characters.  An identifier must begin with a letter and may be followed by any sequence of letters, numbers, or underscores.  The character case used for Identifiers is not restricted and may include any combination of lower or uppercase characters.  An identifier may represent a variable, which is used as a reference to a data type.

## 2.3. Keywords

The following table contains reserved words or *keywords* of the language.  These words cannot be used as regular identifiers and are used for type identification, control flow, and other semantically meaningful language constructs.

|  |  |  |
| --- | --- | --- |
| return  if  else  is  true  false  none | do  end  while  for  each  in  fun  object | **(if we have time)**  try  catch  continue  break |

Table 1 - Language Keywords

## 2.4 Escape Sequences

|  |  |  |
| --- | --- | --- |
| **Escape Sequence** | **Meaning** | **Notes** |
| \newline | Ignored |  |
| \\ | Backslash (\) |  |
| \' | Single quote (') |  |
| \" | Double quote (") |  |
| \a | ASCII Bell (BEL) |  |
| \b | ASCII Backspace (BS) |  |
| \f | ASCII Formfeed (FF) |  |
| \n | ASCII Linefeed (LF) |  |
| \r | ASCII Carriage Return (CR) |  |
| \t | ASCII Horizontal Tab (TAB) |  |
| \uxxxx | Character with 16-bit hex value *xxxx* (Unicode only) | (1) |
| \Uxxxxxxxx | Character with 32-bit hex value *xxxxxxxx*(Unicode only) | (2) |
| \v | ASCII Vertical Tab (VT) |  |

Table 2 - Escape Sequences

# 3. Notation and Punctuation

## 3.1. Syntax Notation

In the syntax notation used in this manual, syntactic categories are indicated by *italic* type, and literal words and characters in typewriter style.  Alternative categories, or in other words, syntactic categories produced from another syntactic category during translation, are listed on separate lines.  Optional terminal or nonterminal symbols will be indicated with the subscript **opt** to indicate the symbol may or may not exist for syntactic matching to occur.

## 3.2. Punctuation

A line in a Swim program contains statements and expressions.  Each line with the exception of lines ending in the keywords *do* and *end* must end in a semi-colon.   All whitespace including spaces, tabs, and newline characters are ignored except for the purpose of separating tokens.

# 4. Types

## 4.1. Basic Types

The basic data types of the Swim language are translated directly to the equivalent types in Python for the purpose of interpretation.  The list below should be familiar even to novice programmers.

The basic data types of the Swim language are integers, long integers, floats, booleans, strings and none. They are based on Python’s standard types and their properties should be familiar to most programmers.

### 4.1.1. Integers

Whole numbers ranging from -2147483648 through 2147483647. The type is converted to a long integer in the case where the value exceeds the boundaries.

### 4.1.2. Long Integers

Whole numbers with arbitrary length that is only bounded by the available virtual memory.

### 4.1.3. Float Point Numbers

Double precision floating point numbers whose range depends on the users machine. Any number with a decimal part is represented as a float. Results of operations consisting of both integers and floats evaluate to floating point values.

### 4.1.4. Boolean Values

Boolean values are denoted by the keywords “true” and “false” but are really nothing more than an alternative representation of the integers 1 and 0. This means that any valid arithmetic operations on integers will also work with the the boolean “true” and “false”.  So *true + true* evaluates to *2* and *true \* 3* equals *3*.

### 4.1.5. Strings

An immutable sequence of characters.

### 4.1.6. None

A single object that denotes a lack of a value. Functions that don’t specify a return value evaluate to None.

## 4.2. Derived Types

The main derived types supported by Swim are lists, dictionaries and prototypical objects. They can be recursively defined and can contain any of the types that are mentioned in this manual.

### 4.2.1. Lists

A list stores zero or more objects in a specific order. It is mutable and therefore can be manipulated after creation. It is defined using open and close brackets, so [ ] is an empty list and [1] is a list containing one integer. As in most popular languages, contents of a list are accessed using the identifier of the list followed by the index wrapped in brackets. It is implemented as an array list so it automatically resizes when more elements are added.  The bound on its size depends only on the available memory.

// define a list containing 4 items

list = [item1, item2, item3, item4];

// add a new item to the end of the list

list.append(newItem);

// add a new item to the front of the list

list.prepend(newItem);

// append contents of list2 to list

list.extend(list2);

// access the second cell of the list

list[1]

### 4.2.2. Dictionaries

Dictionaries map keys to values. The keys must be of an immutable type and unique to provide a many-to-one mapping.  A dictionary is defined using curly braces and is accessed using square brackets notation.

// create a new dictionary

dict = {};

// create a new dictionary with predefined key value pairs

dict = {key1 : val1, key2 : val2 };

// lookup the value that the key maps to

dict[key]

// update or define the value for the given key

dict[key] = val;

### 4.2.3. Objects

Swim implements a prototype based object model. This allows creation and modification of the object’s fields at runtime.

## 4.3. Type Conversion

Basic types may be converted from one another using built in functions.  For example to convert from type int to type string the following function is called

var = str(*exprk*) where *exprk*is an expression representing an int type.

     float(*exprk*) …

        int(*exprk*) …

        long(*exprk*) …

        bool(*exprk*) …

## 4.4.  Constants and Literals

### 4.4.1 Constants

#### 4.4.1.1 Boolean Constants

SWIM has two built-in constants, true and false.

false

        The value false for the boolean primitive type.

true

        The value true for the boolean primitive type

#### 4.4.1.2 Math Constants

SWIM uses Python’s math constants cmath.pi and cmath.e

math.**pi**

The mathematical constant *π*, to the precision available.

math.**e**

The mathematical constant *e*, same.

#### 4.4.1.3 String constants

String constants are useful helpers, both in regular expressions, and in complex crawling procedures.  The String constants available in SWIM are carefully chosen subset of those available in Python.

string.**letters**

The concatenation of the strings **lowercase** and **uppercase** described below.

string.**lowercase**

A string containing all the characters that are considered lowercase letters. On most systems this is the string 'abcdefghijklmnopqrstuvwxyz'.

string.**uppercase**

A string containing all the characters that are considered uppercase letters. On most systems this is the string'ABCDEFGHIJKLMNOPQRSTUVWXYZ'.

string.**digits**

The string '0123456789'.

string.**whitespace**

A string containing all characters that are considered whitespace. On most systems this includes the characters space, tab, linefeed, return, formfeed, and vertical tab.

### 4.4.2 Literals

#### 4.4.2.1 String Literals

String literals are a sequence of characters enclosed in double or single quotations: “” or ‘’.  All characters defined by an escape sequence must be escaped using the \ symbol in order to be used inside of a string literal.

For escape sequences, see Section 2.4

# 5. Operators

## 5.1.1. Unary Operators

|  |  |  |  |
| --- | --- | --- | --- |
| **symbol** | **name** | **description** | **associativity** |
| - | unary minus | negative value of operand | right |
| + | unary plus | positive value of operand | right |
| ! | unary not | logical NOT of operand | right |
| ++ | unary increment | operand value incremented by 1 | prefix-> right  postfix->left |
| -- | unary decrement | operand value decremented by 1 | prefix -> right  postfix->left |

Table 3 - Unary Operators

## 5.1.2. Multiplicative Operators

|  |  |  |  |
| --- | --- | --- | --- |
| **symbol** | **name** | **description** | **associativity** |
| \* | multiply | product of adjacent operands | left |
| / | divide | quotient of adjacent operands | left |
| % | mod | modulus of adjacent operands | left |
| ^ | pow | exponentiation of adjacent operands | left |

Table 4 - Multiplicative Operators

## 5.1.3. Additive Operators

|  |  |  |  |
| --- | --- | --- | --- |
| **symbol** | **name** | **description** | **associativity** |
| + | plus | addition of adjacent operands | left |
| - | minus | subtraction of adjacent operands | left |

## 5.1.4. Relational Operators

|  |  |  |  |
| --- | --- | --- | --- |
| **symbol** | **name** | **description** | **associativity** |
| < | less than | true if left op lt right op | left |
| <= | less than or equal to | true if left op le right op | left |
| > | greater than | true if left op gt right op | left |
| >= | greater than or equal to | true if left op ge right op | left |

Table 5 - Relational Operators

## 5.1.5. Equality Operator

|  |  |  |  |
| --- | --- | --- | --- |
| **symbol** | **name** | **description** | **associativity** |
| == | equal to | true if left op eq right op | left |
| != | not equal to | true if left op neq right op | left |

Table 6 - Equality Operators

## 5.1.6. Assignment Expressions

The **=** operator replaces the value of the left hand side of an expression with the value on the right hand side where the right hand side is an evaluated expression. Additionally, the following assignment operators exist:

|  |  |  |  |
| --- | --- | --- | --- |
| **symbol** | **name** | **description** | **associativity** |
| += | plusequals | left operand equals left operand plus right operand | left |
| -= | minusequals | left operand equals left operand minus right operand | left |
| \*= | starequals | left operand equals left operand times right operand | left |
| ^= | powequals | left operand equals left operand to the power of right operand | left |

Table 7 - Assignment Expressions

# 6. Statements

Except where otherwise noted statements are executed in order.  Statements do not have values. Statements are categorized into the following groups:

*statement:*

*expression-statement*

*...*

## 6.1.  Expression-statement

*expression-statement:*

*expressionopt*

An expression statement is either an assignment or a function call.  The result of such an assignment or function call is evaluated and used as the value for that expression.  All side effects such as subroutines started by a given *expression-statement* must terminate before the following statement may be executed.

## 6.2 Compound Statements

The compound statement is a block.  It is simply a list of several statements used in place of a single statement, as with a function definition.

## 6.3 Selection Statements

Selection statements are used for conditional execution. If statements work by evaluating each expression top to bottom until one evaluates to true, at which point the statement inside is executed.  If none evaluate to true and there is an else statement (else statements do not have their own associated expression, but serve as a catch-all), then the statement inside the else is evaluated.  If no else statement is present and all other expressions associated with the if statement evaluate to false, execution continues below the if statement, without executing any statements within the if.

if:

**if** *expr* **do** *statement*

**else**  *statement*;

**end**

**if** *expr*

*statement;*

**elif** *expr*

*statement;*

**elif** *expr*

*statement;*

**else**

*statement;*

**end**

## 6.4 Iteration Statements

The “while” and “for each” iteration statements are used for looping.

*iteration-statements*:

    while *expression* do *stmt* end

for each *item* in *iterable* do *stmt* end

### 6.4.1. While

While statements execute as long as the expression that follows the *while* keyword evaluates to true.

**while** *expression*  **do**

*stmt*;

**end**

### 6.4.2. For Each

For each statements are used to iterate over iterable objects like lists and dictionaries. On each iteration it assigns the next object to the *item*.

**for each** *item* **in** *iterable* **do**

*stmt;*

**end**

# 7. Functions

## 7.1 Function calls

A function can be executed by calling the name of the function followed by a parenthesized comma separated list of expressions that represent arguments or parameters for the function.

## 7.2 Function definitions

User-defined function objects are declared and defined together using the following syntax:

|  |
| --- |
| **fun\_def** = “fun” **id** “(“ **expression** “)” **statement**  **end** |

Function definitions define executable statements where the function block is given its own scope.   The function body (here defined as a statement) is executed when the function is called (see above).  **fun** with a new definition executed inside a function definition defines a local, nested function that can access its parent’s functions local variables.

# 8. Objects

## 8.1 Object definitions

Objects are defined in Swim using the object keyword.  When an object is defined it represents a prototype for an object with the expected attributes held as variables and recursively defined functions that allow Swim users to operate on instances of the object.

Object can be initialized with initial parameters. To do it, there is a built-in function for constructor.

**fun** fun\_name**(**parameters**)**

// Constructor

**fun** **\_\_init\_\_()**

// initialization

**end**

// Other stuffs

**end**

## 8.2 Object instance

Object instance can be initialized like,

obj\_name = **object** fun()

If you want to make an instance with constructor, convey the parameters like this,

  obj\_name = **object** fun(par1, par2, ...)

As SWIMM is the prototype-based language, user can add or change attribute and methods on the fly.

# 9. Scope and Namespacing

## 9.1 Lexical Scope

SWIM uses dynamic scoping. The rules of scope may be summarized as follows.  Distinct identifiers within a given scope must have distinct names; identifiers with the same name in distinct scopes are distinct identifiers; declarations of variables or functions within nested scopes having identifiers.

## 9.2 Function Scope

In terms of functions, this means that a variable whose scope is a certain function, then it is valid and can be used anywhere within the text of that function.  Parameters, variables, types, and functions declared within a function all have as their scope that function.

## 9.3 Block Scope

Variables declared within any block, for example within a given function, have that block as their scope.

## 9.4 Global Scope

Variables declared at the very beginning of a program have global scope, and are valid and can be used throughout the program.

## 9.5 Nested Scope and Scope in General

The most straightforward aspect of scope may be thought of simply as the region marked by matching the fun keyword indicating the start of the function with the end keyword marking the end. Everything declared within this region will have a lifetime beginning at the time of declaration and ending when the function exits (reaches the end keyword)

So, for example, in:

**fun** someFunction (someParameter)

  someVariable

**end**

By the rules of function scope, the scope of someParameter is someFunction.  The same is true of someVariable.  Nested scope is simply the hierarchy of scope created by nested functions with fun and do keywords, e.g.:

**fun** someFunction (someParameter)

  someVariable = evaluatedExpression

  conditionalStatement do

           someOtherVariable = someVariable

  end

  print(someOtherVariable)

**end**

Here the someVariable declared within the block defined by the fun and do keywords defines one variable, while the second  use of someVariable within the inner conditionalStatement marked by the keywords do and end is used to evaluate someOtherVariable.  The print statement at the end shows that although someOtherVariable was declared within an inner block it can still be accessed from other regions of someFunction.  However, by function scope both someVariable and someOtherVariable will be discarded when someFunction exits..

# 10. File inclusion

In many cases, programs are required to use external files. This file inclusion can be implemented as a simple statement.

**import** file\_name

If the file is located in a folder other than the folder containing the program it may be accessed as follows:

**from** folder\_name **import** file\_name

Additionally the right hand side of the **import** statement is not confined to only file names. It can also have any object that is explicitly defined in the file. For example, if there is a defined function in some external file, it is possible to access that function using the following syntax:

**from** file\_name **import** fun\_name

# Appendix. Full grammar

#-----------------------------------------------------------------------------#

# #

# COLUMBIA UNIVERSITY - FU FOUNDATION SCHOOL OF ENGINEERING #

# COMPUTER SCIENCE DEPARTMENT #

# #

# COMS W4115 - PROGRAMMING LANGUAGES AND TRANSLATORS #

# Professor A. Aho #

# #

# Team 3 Final Project: "SWIM" #

# Team Mentor: A. Aho #

# #

# Team members: #

# #

# Name Role UNI #

# Morris Hopkins Project Manager mah2250 #

# Seungwoo Lee Language Guru sl3492 #

# Lev Brie System Architect ldb2001 #

# Alexandros Sigaras System Integrator as4161 #

# Michal Wolski Verification and Validation mtw2135 #

# #

#-----------------------------------------------------------------------------#

#-----------------------------------------------------------------------------#

# Library Import #

#-----------------------------------------------------------------------------#

import sys, os

import re

sys.path.insert(0,os.path.join("..", "include"))

if sys.version\_info[0] >= 3:

raw\_input = input

# Parsing

from pyquery import PyQuery as pq

import urllib, getpass

# PDF

from fpdf import fpdf as pdf

#-----------------------------------------------------------------------------#

#-----------------------------------------------------------------------------#

# External Functions #

#-----------------------------------------------------------------------------#

def stripe\_quotation(string):

result = string.replace("'","") if string.startswith("'") else string.replace('"', "")

return result

#-----------------------------------------------------------------------------#

#-----------------------------------------------------------------------------#

# Declare tokens #

#-----------------------------------------------------------------------------#

reserved = {

'if' : 'IF',

'else' : 'ELSE',

'do' : 'DO',

'True' : 'TRUE',

'False' : 'FALSE',

'while' : 'WHILE',

'for' : 'FOR',

'foreach' : 'FOREACH',

}

tokens = [

'STRING1', 'STRING2', 'SELECTOR', 'ID','NUMBER', 'AND', 'OR', 'XOR', 'NOT', 'COMMA',

'PLUS','MINUS','MULTIPLY','DIVIDE','ASSIGN', 'EQUALS','POW','MOD',

'LPAREN','RPAREN'

] + list(reserved.values())

# Tokens

# Operator Tokens

t\_AND = r'and|&&'

t\_OR = r'or|\|\|'

t\_XOR = r'xor'

t\_NOT = r'not|!'

t\_PLUS = r'\+'

t\_MINUS = r'-'

t\_MULTIPLY = r'\\*'

t\_DIVIDE = r'/'

t\_ASSIGN = r'='

t\_EQUALS = r'=='

t\_POW = r'\^'

t\_MOD = r'\%'

t\_LPAREN = r'\('

t\_RPAREN = r'\)'

# Keywords

'''

t\_IF = r'if'

t\_ELSE = r'else'

t\_DO = r'do'

t\_TRUE = r'True'

t\_FALSE = r'False'

t\_WHILE = r'while'

t\_FOR = r'for'

t\_FOREACH = r'foreach'

'''

t\_STRING1 = r'u?"\\?[^"]\*"'

t\_STRING2 = r"u?'\\?[^']\*'"

t\_SELECTOR = r'@'

t\_COMMA = r','

def t\_NUMBER(t):

r'[0-9]\*\.?[0-9]+'

try:

t.value = float(t.value)

except ValueError:

print("Float value too large %s" % t.value)

t.value = 0

return t

def t\_ID(t):

r'[a-zA-Z\_][a-zA-Z\_0-9]\*'

t.type = reserved.get(t.value,'ID') # Check for reserved words

return t

t\_ignore = " \t"

def t\_newline(t):

r'\n+'

t.lexer.lineno += t.value.count("\n")

def t\_error(t):

print("Illegal character '%s'" % t.value[0])

t.lexer.skip(1)

#-----------------------------------------------------------------------------#

#-----------------------------------------------------------------------------#

# Lex #

#-----------------------------------------------------------------------------#

import ply.lex as lex

lex.lex(optimize=1)

#-----------------------------------------------------------------------------#

#-----------------------------------------------------------------------------#

# Parsing Rules #

#-----------------------------------------------------------------------------#

precedence = (

('left','PLUS','MINUS', 'COMMA'),

('left','MULTIPLY','DIVIDE', 'MOD'),

('right','UMINUS','POW'),

)

# dictionary of names

names = { }

def p\_start(t):

'''start : statement'''

# def p\_statement\_true(t):

# 'statement : TRUE'

# print True

# def p\_statement\_false(t):

# 'statement : FALSE'

# print False

def p\_function(t):

'''function : ID LPAREN expression RPAREN'''

if t[1] == "print" :

# Escaped sequence handling

escaped\_sequences = (r"\newline", r"\\", r"\'", r'\"', r"\a", r"\b", r"\f", r"\n", r"\r", r"\t", r"\v")

for seq in escaped\_sequences:

if t[3].find(seq) != -1:

t[3] = t[3].decode('string-escape')

# string containing ""

# 2 byte unicode with unicode characters

if re.match(r'u"\\u', t[3]):

print unichr(int(t[3][4:8], 16))

# 2 byte unicode with strings

elif re.match(r'^u"', t[3]):

print t[3].replace('"','')[1:].decode("utf-8")

# string containing ''

elif re.match(r"u'\\u", t[3]):

print unichr(int(t[3][4:8], 16))

# 2 byte unicode with strings

elif re.match(r"^u'", t[3]):

print t[3].replace("'",'')[1:].decode("utf-8")

else:

print t[3].replace("'",'')

elif t[1] == "pdf":

try:

#print stripe\_quotation(t[3][0])

f = pdf.FPDF()

f.add\_page()

f.set\_font('Arial','B',16)

f.multi\_cell(w=200,h=5,txt = stripe\_quotation(t[3][0]))

# for our user test

filename = stripe\_quotation(t[3][1])

filename = filename.split('.')[0] + '\_' + getpass.getuser() + '.' + filename.split('.')[1]

print filename

f.output(os.path.join("..","doc",filename),'F')

except:

print("Mismatch grammar for pdf output!")

t[0] = 0

def p\_statement\_assign(t):

'statement : ID ASSIGN expression'

names[t[1]] = t[3]

def p\_statement\_equals(t):

'statement : expression EQUALS expression'

print t[1] == t[3]

def p\_statement\_if\_else(t):

'statement : IF expression DO expression ELSE expression'

if t[2]:

t[4]

print "here1"

else:

t[6]

print "here2"

def p\_statement\_expr(t):

'statement : expression'

print t[1]

def p\_expression\_binop(t):

'''expression : expression PLUS expression

| expression MINUS expression

| expression MULTIPLY expression

| expression DIVIDE expression

| expression POW expression

| expression MOD expression

| expression AND expression

| expression OR expression

| expression XOR expression

| NOT expression

| TRUE

| FALSE'''

if t[2] == '+':

t[0] = t[1] + t[3] # add

elif t[2] == '-':

t[0] = t[1] - t[3] # subtract

elif t[2] == '\*':

t[0] = t[1] \* t[3] # multiply

elif t[2] == '/':

t[0] = t[1] / t[3] # divide

elif t[2] == '^':

t[0] = t[1] \*\* t[3] # power

elif t[2] == '%':

t[0] = t[1] % t[3] # remainder3

elif (t[2] == 'and' or t[2] =='&&'):

t[0] = t[1] and t[3]

elif (t[2] == 'or' or t[2] =='||'):

t[0] = t[1] or t[3]

elif t[2] == 'xor':

t[0] = (t[1] and not t[3]) or (not t[1] and t[3])

elif t[1] == 'not':

t[0] = not t[2]

elif t[1] == 'True':

t[0] = True

elif t[1] == 'False':

t[0] = False

elif t[2] == '<': t[0] = t[1] < t[3]

def p\_expression\_uminus(t):

'expression : MINUS expression %prec UMINUS'

print "uminus"#t[0] = -t[2]

def p\_expression\_group(t):

'expression : LPAREN expression RPAREN'

t[0] = t[2]

def p\_expression\_parameters(t):

'expression : expression COMMA expression'

# Stacking the parameters to the right as list

t[0] = [t[1], t[3]]

def p\_expression\_parse(t):

'expression : parse'

t[0] = t[1]

def p\_expression\_function(t):

'expression : function'

t[0] = t[1]

def p\_expression\_parse\_text(t):

'parse : SELECTOR LPAREN expression RPAREN'

try:

selector = stripe\_quotation(t[3][0])

if type(t[3][1]) == str:

url = stripe\_quotation(t[3][1])

d = pq(url=url, opener=lambda url: urllib.urlopen(url).read())

t[0] = d(selector)

else:

t[0] = t[3][1](selector)

except Exception:

print("Mismatch grammar for parsing!")

t[0] = 0

def p\_expression\_number(t):

'expression : NUMBER'

#print "push " + str(t[1])#t[0] = t[1]

t[0] = t[1]

def p\_expression\_name(t):

'expression : ID'

try:

t[0] = names[t[1]]

except LookupError:

print("Undefined name '%s'" % t[1])

t[0] = 0

def p\_expression\_string(t):

'expression : string'

try:

t[0] = t[1]

except LookupError:

print("Undefined name '%s'" % t[1])

t[0] = 0

def p\_expression\_string\_twoways(t):

'''string : STRING1

| STRING2'''

try:

t[0] = t[1]

except LookupError:

print("Undefined name '%s'" % t[1])

t[0] = 0

def p\_expression\_unistring(t):

'expression : ID string'

try:

if t[1] == 'u':

t[0] = t[1] + t[2]

else:

raise Exception()

except LookupError:

print("Undefined name '%s'" % t[1])

t[0] = 0

def p\_error(t):

if t:

print("Syntax error at '%s'" % t.value)

else:

print("Syntax error at EOF")

#-----------------------------------------------------------------------------#

#-----------------------------------------------------------------------------#

# Yacc #

#-----------------------------------------------------------------------------#

import ply.yacc as yacc

yacc.yacc(optimize=1)

mode = 2

if mode == 1:

s = raw\_input("SWIM REPL> ")

lex.input(s)

if len(sys.argv) > 1:

fn = open(sys.argv[1])

for line in fn.readlines():

if line == "\n": continue

if mode == 1:

lex.input(line)

while 1:

tok = lex.token()

print tok

if not tok:

break

else:

yacc.parse(line)

fn.close()

else:

while 1:

if mode == 1:

tok = lex.token()

print tok

if not tok:

break

else:

try:

s = raw\_input('SWIM REPL> ')

except EOFError:

break

yacc.parse(s)