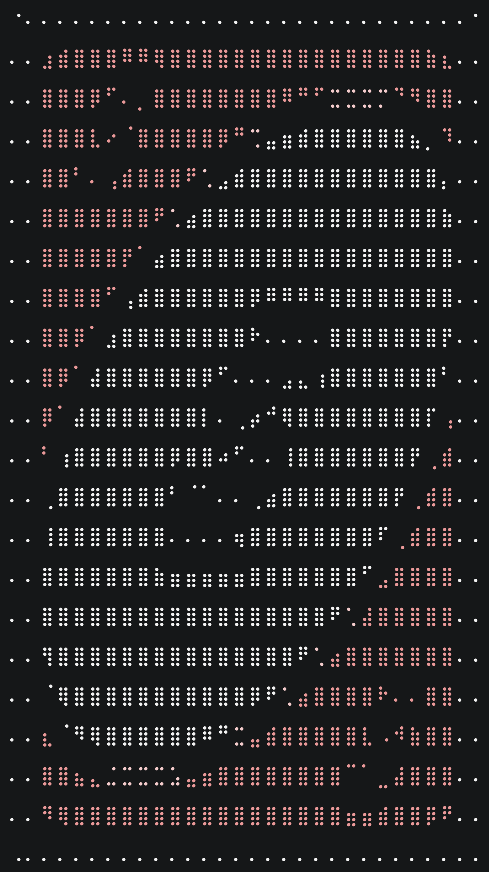
Reverse Jython

Language Design and Example Usage

Version 9.6

Cassidy Heulings

Reverse Jython was made to practice reading and recognizing words in reverse order. This is so that users could practice thinking differently than how they are used to reading, but also so they could get better at spotting palindromes and playing word games like solving anagrams, word ladders, or apps like word cookies.



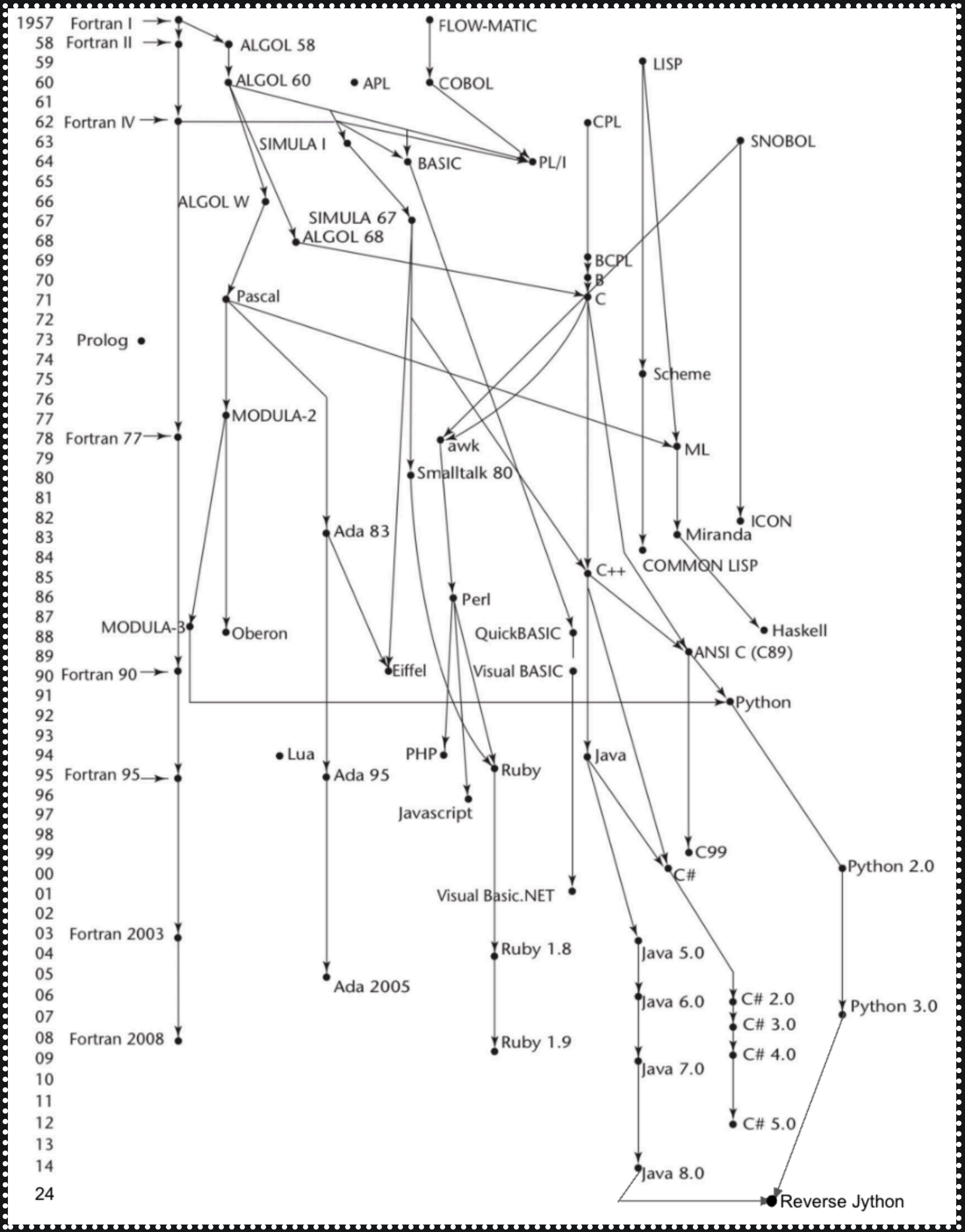
1. Introduction

Reverse Jython is a simple yet somewhat confusing, object-oriented, and strongly type-safe programming language. Based on Java and Python but differing in the following ways:

1. Keywords are reversed. However, operators are not. Here is a breakdown:
    a. Identifiers - not reversed
    b. Keywords - reversed
    c. Literals - not reversed
    d. Operators and punctuators - not reversed (logic may be reversed based on operator such as >,<, -, or /)
    e. Comments - not reversed
2. Entire lines and expressions are reversed. Phrases, including the structure of statements, are written in reverse
3. Python ’s built-in functions and types are treated as keywords
4. Functions are defined using def, and the syntax uses indentation and colons instead of curly braces. Semicolons are no longer used at the end of lines, but are still used in things like looping
5. Classes use tini (init reversed) to define constructors, and niam (main reversed) as the entry point if defined
6. file (elif reversed) is used instead of else if in conditional checks
7. While assignment statements are written in reverse, the order of elements in lists is unchanged. If we assign myList to the list [0, 1, 2, 3] ([0, 1, 2, 3] = myList), element 0 is at index 0 and element 3 is at index 3
8. Comments use Java ’s // and /* */ syntax, and are not reversed so there is at least some readability
9. Arithmetic and comparison operations have reversed syntax, and may have reversed logic based on the operation. For example, x = a - b will be b - a = x. The compiler will take care of the backwards logic, assigning x the value of a - b, even if the syntax has the opposite result. Similarly, 5 > 1 becomes 1 > 5, which may appear to be false, but as the compiler will reverse the syntax to take care of the reversed logic, will be true.

1.1 Genealogy

Reverse Jython was made in 2024 inspired by reading backwards. It is a mix of Java and Python, blending the syntax of both languages to create a hybrid syntax. It uses references, built-in types, and built-in functions similar to Python. Like Java, it uses static typing, with the rav type (var reversed) being inferred at compile time. Once the compiler locks in a type for rav, it is statically enforced.



1.2 Hello World

```
:HelloWorld ssalc
    :(fles)tini fed
        ("Hello, World!")tnirp

()HelloWorld
```

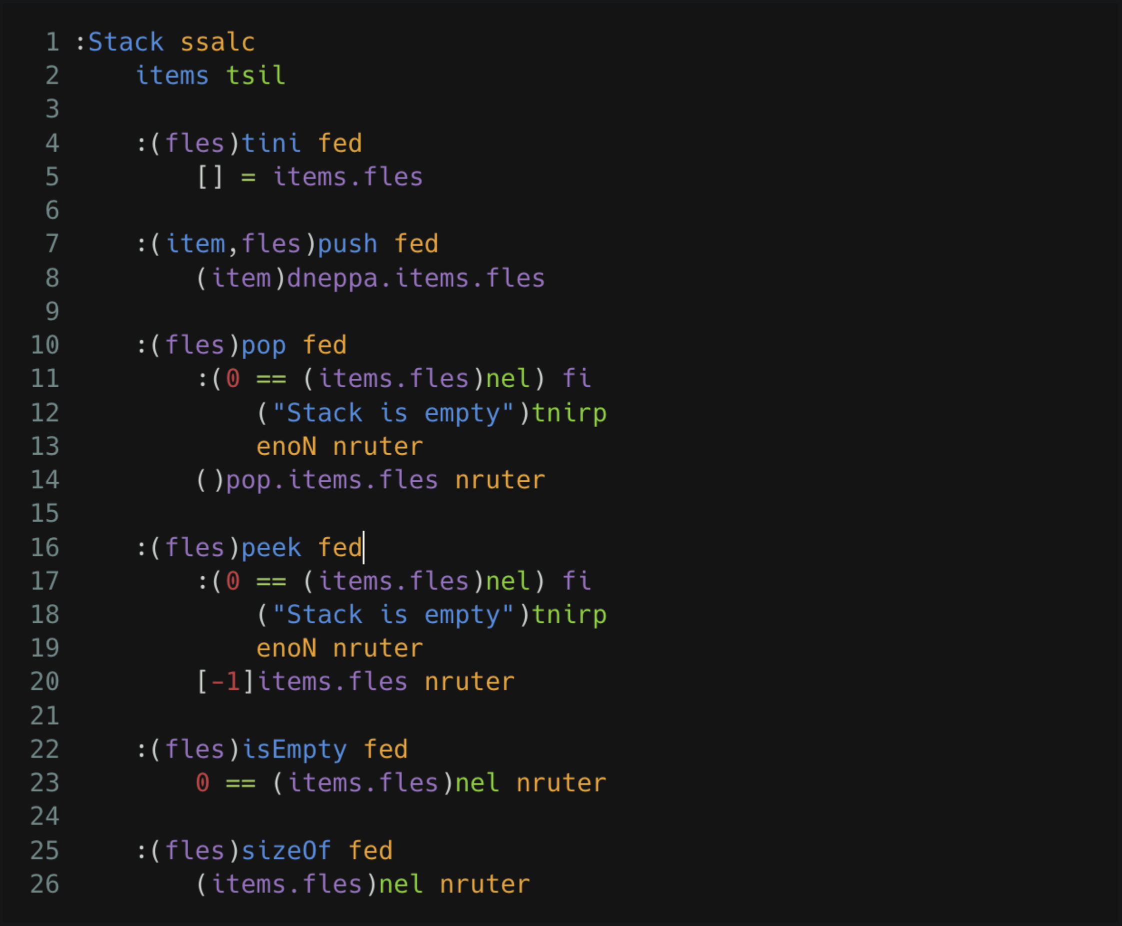
1.3. Program Structure

The key organizational concepts in Reverse Jython are as follows:

1. Code is organized by classes (with syntax ssalc). A class will hold data and methods, similar to Java. Standalone functions outside classes are allowed, but may take away from the organization of code, reducing clarity. If a class has a niam (main reversed) function, it is the entry point to the program.
2. Classes are initialized using the constructor function tini (init reversed), like Python’s __init__ or Java’s constructor. tini is automatically called by object instantiation.
3. Indentation-Driven Scope
   Scope if defined by indentation and colons, not curly braces or semi-colons, similar to Python. Control constructs like if, for, or while, use colons and are indented.

```
("\-^-")             ("-^-/")
[ 'o__o`             `o__o' ]
\_  (_Y_)            (_Y_) _/
,`.-'--`.._        _..`--'-.`,
(__)--,_(__)      (__)_,--(__)
1 ;   :7              7:   ; 1
: '-.-`,\_          _/,`-.-' :
(,_)~~-(,_)        (_,)-~~(_,)
```

This example:



```
:Stack ssalc
    items tsil

    :(fles)tini fed
        [] = items.fles

    :(item,fles)push fed
        (item)dneppa.items.fles

    :(fles)pop fed
        :(0 == (items.fles)nel) fi
            ("Stack is empty")tnirp
            enoN nruter
        ()pop.items.fles nruter

    :(fles)peek fed
        :(0 == (items.fles)nel) fi
            ("Stack is empty")tnirp
            enoN nruter
        [-1]items.fles nruter

    :(fles)isEmpty fed
        0 == (items.fles)nel nruter

    :(fles)sizeOf fed
        (items.fles)nel nruter
```

Declares a class named Stack. The fully qualified name of the class is Stack. Stack contains one field named items and five methods named push, pop, peek, isEmpty, and size. The class also includes a constructor method to initialize the items field as an empty list.

1.4 Types and variables

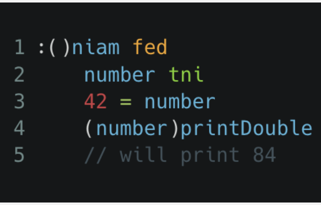
There is one kind of type in Reverse Jython: reference types. Reference types include all user-defined classes, as well as variables. Variables store references to their data, which are known as objects. One object can be referenced by more than one variable, so operations affecting one variable may change the value of another variable if both variables are referencing the same object.

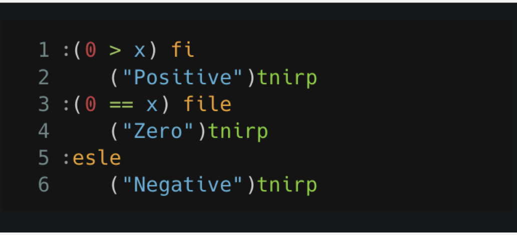
1.5 Visibility

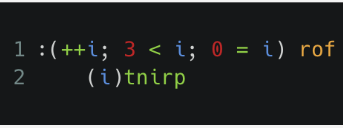
Reverse Jython supports public and private member visibility. The accessibility of class fields and methods are controlled by these visibilities, similar to Java. By default, fields within a class are private, methods within a class are public, constructors are public, and both fields and methods outside a class are public. To specify otherwise, you must add a modifier to whatever data is being modified. Add a modifier in between class name and ‘ssalc’ keyword for class, add a modifier in between method name and ‘fed’ keyword for functions, or add a modifier in between variable name and type for variables. Public members can be accessed by other classes. Private members can only be accessed within the scope they are defined in.

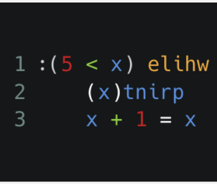
1.6 Statements differing from Java and Python

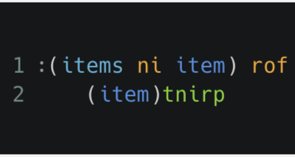
Statement	Example

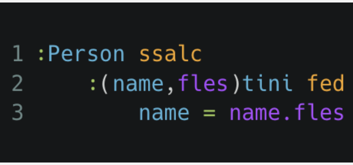
Expression statement	
	`:()niam fed`
	`    number tni`
	`    42 = number`
	`    (number)printDouble`
	`    // will print 84`

If statement	
	`:(0 > x) fi`
	`    ("Positive")tnirp`
	`:(0 == x) file`
	`    ("Zero")tnirp`
	`:esle`
	`    ("Negative")tnirp`

For loop	
	`:(++i; 3 < i; 0 = i) rof`
	`    (i)tnirp`

While loop	
	`:(5 < x) elihw`
	`    (x)tnirp`
	`    x + 1 = x`

For each loop	
	`:(items ni item) rof`
	`    (item)tnirp`

Class definition	
	`:Person ssalc`
	`    :(name,fles)tini fed`
	`        name = name.fles`

2. Lexical Structure

2.1 Programs

A Reverse Jython program consists of one or more source files, each composed of an ordered sequence or characters, typically encoded in Unicode. These source files use Reverse Jython’s reversed syntax, where lines and keywords are written in reversed order, while characters remain in their original form.

A Reverse Jython program uses three stages to be processed:

1. The source file gets transformed into Unicode characters. It uses standard character encoding and keeps individual character symbols when syntax is reversed.
2. The Unicode characters are scanned then grouped into tokens. Keywords are reversed forms of their original spelling, while identifiers, literals, operators, and numbers are unchanged. The reversed order tokens are extracted based on this reversed order.
3. Based on Reverse Jython’s syntax rules, the token stream gets parsed into executable code with syntactic analysis. The parser reconstructs logical structure by applying normal rules after mirroring the line. Therefore, the syntactic analyser is sensitive to token position and format of code.

```
       _..oo8"""Y8b.._
     .88888888o.    "Yb.
   .d888P""Y8888b      "b.
  o88888    88888)       "b
 d888888b..d8888P         'b
 88888888888888"           8
(88DWB8888888P             8)
 8888888888P               8
 Y88888888P     ee        .P
  Y888888(     8888      oP
   "Y88888b     ""     oP"
     "Y8888o._     _.oP"
       `""Y888boodP""'
```

2.2 Grammars

This specification presents the syntax of the Reverse Jython programming language where it differs from Java and Python.

2.2.1 Lexical grammar (tokens) different from Python and Java

Reverse Jython differs through its reversed syntax. Type declarations follow the variable and assignment, with optional static typing using the rav type. Operators remain their original symbols, but the directional logic is flipped as the position of the operands are reversed.

```
 <digit>       ->  0 ¦ 1 ¦ 2 ¦ 3 ¦ 4 ¦ 5¦ 6 ¦ 7 ¦ 8 ¦ 9
 <number>      ->  <digit> ¦ <digit> <number>
 <letter>      ->  a-z ¦ A-Z
 <word>        ->  <letter> ¦ <letter> <word>
 <identifier>  ->  <word> ¦ <word> <number>
 <num_type>    ->   “ tni ”  ¦  “ taolf ”  ¦  “ laer ”
 <type>        ->  "rts" ¦ "rav" ¦  “ loob ”  ¦ <num_type>
 <operator>    ->  + ¦ - ¦ * ¦ /
 <comparison>  ->  == ¦ != ¦ <= ¦ >= ¦ < ¦ >
 <crement>     ->  ++ ¦  -
 <control>     ->   “ elihw ”  ¦  “ fi ”
```

2.2.2 Syntactic (parse) grammar different from Python and Java

Reverse Jython differs through its reversed syntax. Assignment and syntax reads right to left, written in reverse order. In assignment syntax, the expression comes before the variable. Control structures start with the colon (typically how its syntax ends), and have reversed keyword order. The parser design would differ from that of a traditional language as it must recognize the reversed order of the line in order to apply semantic checks.

```
 <term>        ->  <identifier> ¦ <number>
 <expression>  ->  <term> ¦ <term> <operator> <term> ¦ <term> <comparison> <term>
 <assignment>  ->  <expression> = <identifier> <type>
 <inc_or_dec>  ->  <crament> <identifier>
                    IF <identifier> is of type <num_type>
 <cntrl_block> ->  :(<expression>) <control>
```

2.3 Lexical analysis

2.3.1 Comments

There are two forms of comments. Single-line or inline comments are made using characters // as anything following it in the rest of the line will be a comment. Delimited comments are comments that spread from the first instance of /* to the second instance of */. These can vary from one line to multiple lines, as anything types in between these characters are part of the comment.

2.4 Tokens

Tokens are the smallest units that provide meaning in the source code to the lexical analysis. Some tokens are reversed according to Reverse Jython ’ s syntax (these are noted).

- keywords (reversed)
- identifiers
- integer-literal
- real-literal
- character-literal
- string-literal
- operators-or-punctuators

2.4.1 Keywords different from Java or Python

As stated in 1.1, all keywords are reversed. Therefore, all keywords and built in functions that are not palindromes (such as pop) are new keywords. Here are some examples:

class -> ssalc	var -> rav	elif -> file	for -> rof
def -> def	public -> cilbup	file -> elif	private -> etavirp
init -> tini	in -> ni	else -> esle	return -> nruter
self -> fles	while -> elihw	if -> fi	break -> kaerb
true -> eurt	false -> eslaf	None -> enoN	continue -> eunitnoc
int -> tni	str -> rts	ord() -> ()dro	len() -> nel()

3. Type System

Reverse Jython’s type system is a strong, optionally static type system. Type mismatches in operations are not allowed and will result in an error. Type checking happens at compile time when explicitly declared. Reverse Jython supports type inference using type rav (var reversed), which is inferred at compile time: the same time as explicitly declared types. After the type is inferred, it is statically bound to that type and cannot change for the remainder of the program. So, type reassignment with rav type variables is not allowed and will result in an error. Only expressions or literals of single type can be inferred. Expressions of mixed type upon assignment to rav are not allowed and will result in an error. If a rav variable is being assigned based on return value, the return value type must be consistent throughout the entire function. So, for example, a function cannot return either a string or an integer (rts or tni), it must choose to return only strings or only integers.

```
                                                    .
                                                   ":"
                                                 ___:____     |"\/"|
                                               .'        `.    \  /
                                               |  O        \___/  |
~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^

                     .
                    ":"
      |"\/"|     ____:___
       \  /    .'        '.
       |  \___/        O  |
~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^~^
```

3.1 Type Rules

The type rules for Hybrid are as follows:

Intrinsic Types

$$\frac{s \text{ is a string literal}}{\vdash s\text{: rts}} \qquad \frac{i \text{ is an integer literal}}{\vdash i\text{: tni}} \qquad \frac{}{\vdash \text{eurT: loob}} \qquad \frac{}{\vdash \text{eslaF: loob}}$$

Addition and Multiplication

$$\frac{\begin{array}{l} S \vdash e_1 : \text{rts} \\ S \vdash e_2 : \text{rts} \end{array}}{S \vdash e_2 + e_1 : \text{rts}} \qquad \frac{\begin{array}{l} S \vdash e_1 : \text{tni} \\ S \vdash e_2 : \text{tni} \end{array}}{S \vdash e_2 + e_1 : \text{tni}} \qquad \frac{\begin{array}{l} S \vdash e_1 : \text{tni} \\ S \vdash e_2 : \text{tni} \end{array}}{S \vdash e_2 * e_1 : \text{tni}}$$

Subtraction and Division

$$\frac{\begin{array}{l} S \vdash e_1 : T \\ S \vdash e_2 : T \\ T \text{ is an intrinsic type} \end{array}}{S \vdash e_2 - e_1 : T} \qquad \frac{\begin{array}{l} S \vdash e_1 : T \\ S \vdash e_2 : T \\ T \text{ is an intrinsic type} \end{array}}{S \vdash e_2 \,/\, e_1 : T}$$

Assignment and Comparisons

$$\frac{\begin{array}{l} S \vdash e_1 : T \\ S \vdash e_2 : T \\ T \text{ is an intrinsic type} \end{array}}{S \vdash e_2 = e_1 : T} \qquad \frac{\begin{array}{l} S \vdash e_1 : T \\ S \vdash e_2 : T \\ T \text{ is an intrinsic type} \end{array}}{S \vdash e_2 == e_1 : \text{loob}} \qquad \frac{\begin{array}{l} S \vdash e_1 : T \\ S \vdash e_2 : T \\ T \text{ is an intrinsic type} \end{array}}{S \vdash e_2 > e_1 : \text{loob}}$$

$$\frac{\begin{array}{l} S \vdash e_1 : T \\ S \vdash e_2 : T \\ T \text{ is an intrinsic type} \end{array}}{S \vdash e_2 < e_1 : \text{loob}} \qquad \frac{\begin{array}{l} S \vdash e_1 : T \\ S \vdash e_2 : T \\ T \text{ is an intrinsic type} \end{array}}{S \vdash e_2 \mathrel{!=} e_1 : \text{loob}}$$

Reverse Jython has one type: Reference types.

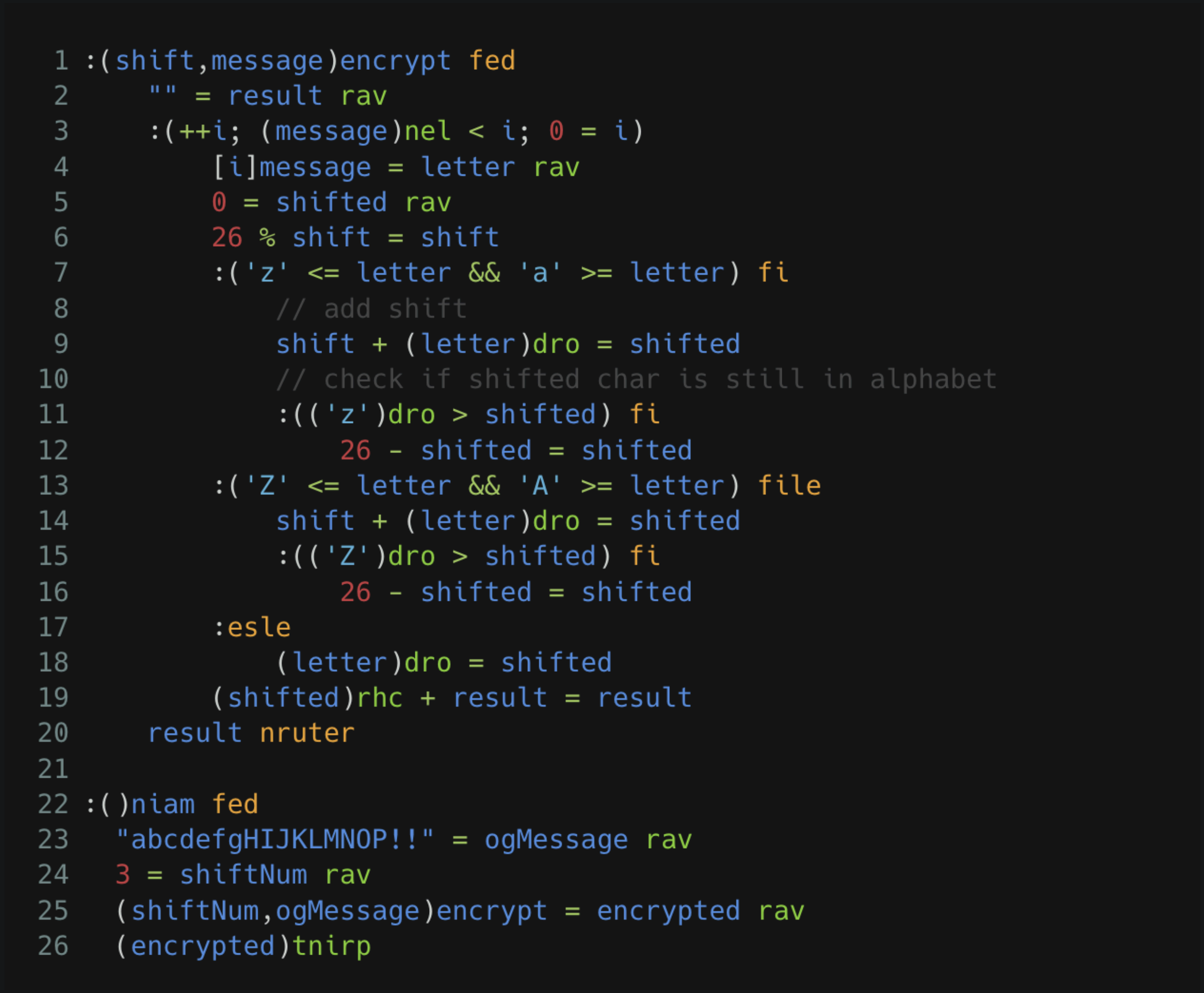
3.2 Reference types differing from Python and Java

User-defined classes and built-in types are reference types. Variables store references, or pointers, to objects in memory as opposed to directly storing values. This typing is closer to Python’s typing rather than Java’s. The difference between Python’s typing and Reverse Jython’s typing is that, regardless of mutability, Reverse Jython values all behave as references. Also, as previously mentioned, type names are reversed. This reversed syntax does not affect their behavior as reference types.

```
"cde" + "ab" = message rts //message = "abcde" for rts (strings)
2004 = year tni // year references integer object 2004
year = randNum tni // both year and randNum reference the same integer object
```

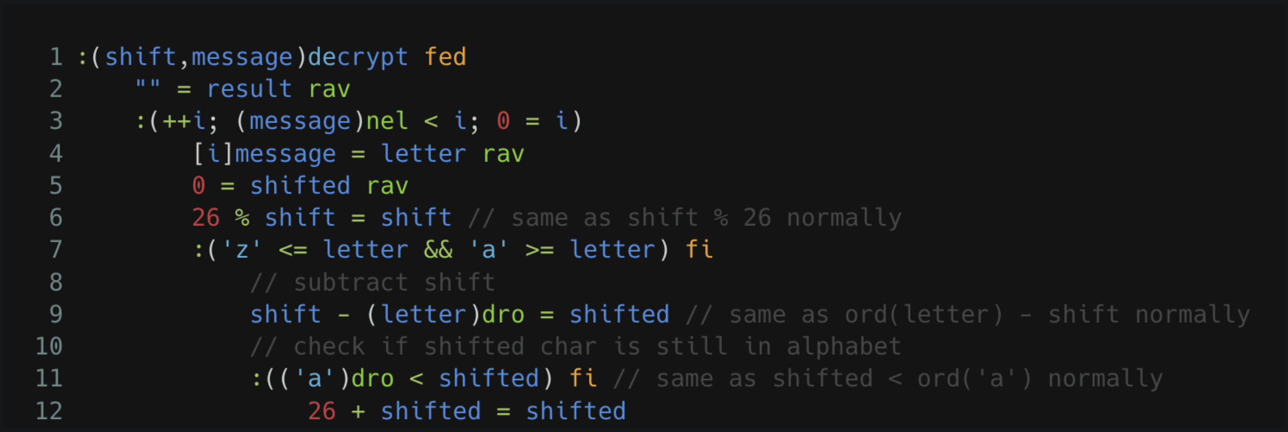
4. Example Programs

4.1 Encrypt:



```
:(shift,message)encrypt fed
    "" = result rav
    :(++i; (message)nel < i; 0 = i)
        [i]message = letter rav
        0 = shifted rav
        26 % shift = shift
        :('z' <= letter && 'a' >= letter) fi
            // add shift
            shift + (letter)dro = shifted
            // check if shifted char is still in alphabet
            :(('z')dro > shifted) fi
                26 - shifted = shifted
        :('Z' <= letter && 'A' >= letter) file
            shift + (letter)dro = shifted
            :(('Z')dro > shifted) fi
                26 - shifted = shifted
        :esle
            (letter)dro = shifted
        (shifted)rhc + result = result
    result nruter

:()niam fed
  "abcdefgHIJKLMNOP!!" = ogMessage rav
  3 = shiftNum rav
  (shiftNum,ogMessage)encrypt = encrypted rav
  (encrypted)tnirp
```

4.2 Decrypt:



```
:(shift,message)decrypt fed
    "" = result rav
    :(++i; (message)nel < i; 0 = i)
        [i]message = letter rav
        0 = shifted rav
        26 % shift = shift // same as shift % 26 normally
        :('z' <= letter && 'a' >= letter) fi
            // subtract shift
            shift - (letter)dro = shifted // same as ord(letter) - shift normally
            // check if shifted char is still in alphabet
            :(('a')dro < shifted) fi // same as shifted < ord('a') normally
                26 + shifted = shifted
        :('Z' <= letter && 'A' >= letter) file
            shift - (letter)dro = shifted
            :(('A')dro < shifted) fi
                26 + shifted = shifted
        :esle
            (letter)dro = shifted
        (shifted)rhc + result = result
    result nruter

:()niam fed
  "abcdefgHIJKLMNOP!!" = ogMessage rav
  3 = shiftNum rav
  (shiftNum,ogMessage)decrypt = decrypted rav
  (decrypted)tnirp
```

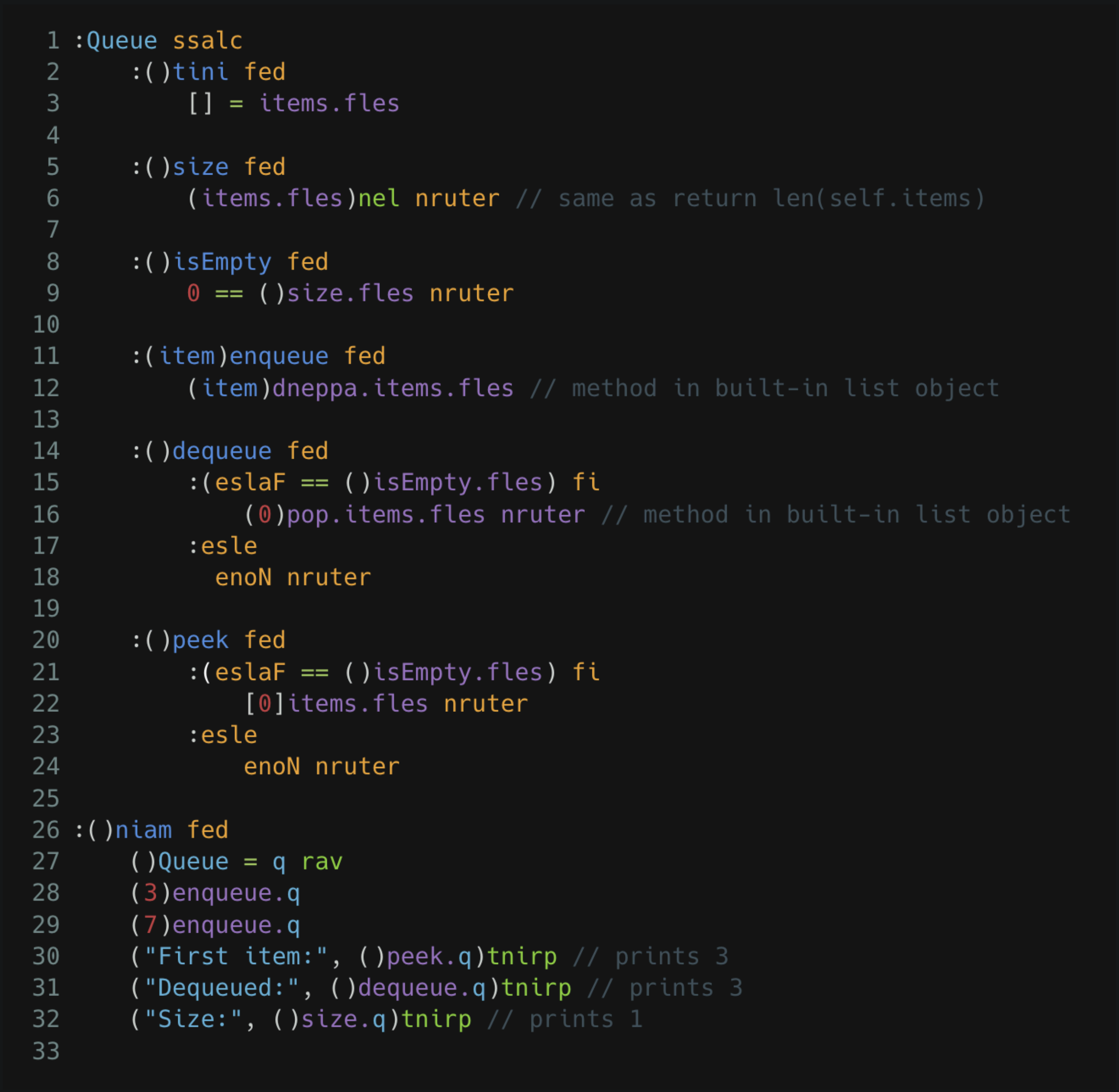
4.3 Factorial:

```
:(n)factorial fed
    :(n == 1 || n == 0) fi
        1 nruter
    :esle
        (1 - n)factorial * n nruter // same as n*factorial(n-1)

 ("Factorial of 5 is:",(5)factorial)tnirp
```

4.4 Bubble Sort:

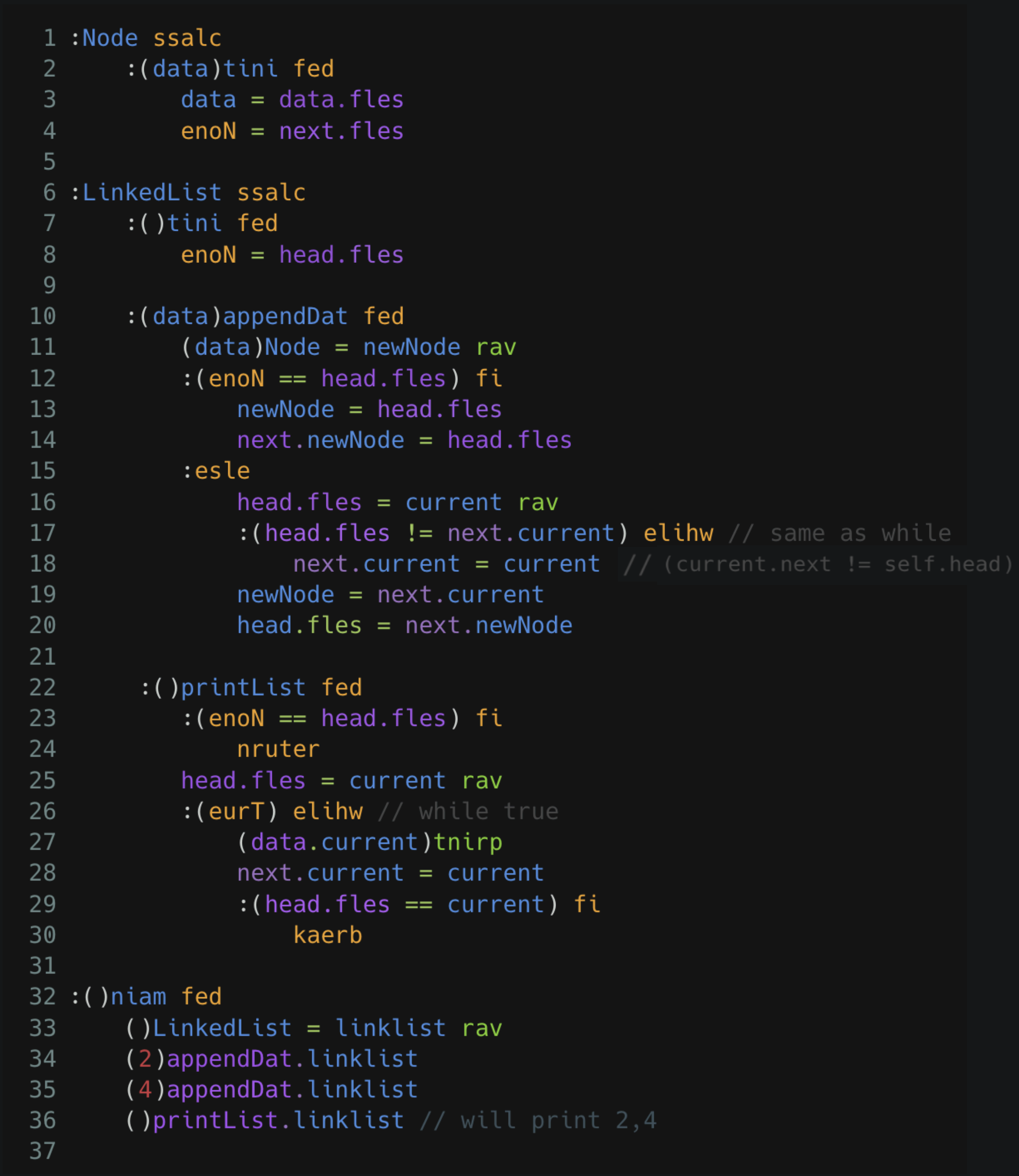
```
:(ogList)bubbleSort fed
    (ogList)nel = n rav
    :(++i; n < i; 0 = i) rof // same as while i < n i++
        :(++j; 1 - i - n < j; 0 = j) rof // same as while j < n - i - 1 j++
            :([1 + j]ogList > [j]ogList) fi // if [j] > [j + 1]
                // swap element j and j+1
                [j]ogList = temp rav
                [1 + j]ogList = [j]ogList
                temp = [1 + j]ogList
     ogList nruter

[3, 6, 1] = unsorted rav
(unsorted)bubbleSort = sorted rav
```

4.5 Queue:



```
:Queue ssalc
    :()tini fed
        [] = items.fles

    :()size fed
        (items.fles)nel nruter // same as return len(self.items)

    :()isEmpty fed
        0 == ()size.fles nruter

    :(item)enqueue fed
        (item)dneppa.items.fles // method in built-in list object

    :()dequeue fed
        :(eslaF == ()isEmpty.fles) fi
            (0)pop.items.fles nruter // method in built-in list object
        :esle
          enoN nruter

    :()peek fed
        :(eslaF == ()isEmpty.fles) fi
            [0]items.fles nruter
        :esle
            enoN nruter

:()niam fed
    ()Queue = q rav
    (3)enqueue.q
    (7)enqueue.q
    ("First item:", ()peek.q)tnirp // prints 3
    ("Dequeued:", ()dequeue.q)tnirp // prints 3
    ("Size:", ()size.q)tnirp // prints 1
```

4.6 Circular single linked list:



```
:Node ssalc
    :(data)tini fed
        data = data.fles
        enoN = next.fles

:LinkedList ssalc
    :()tini fed
        enoN = head.fles

    :(data)appendDat fed
        (data)Node = newNode rav
        :(enoN == head.fles) fi
            newNode = head.fles
            next.newNode = head.fles
        :esle
            head.fles = current rav
            :(head.fles != next.current) elihw // same as while
                next.current = current  // (current.next != self.head)
            newNode = next.current
            head.fles = next.newNode

     :()printList fed
        :(enoN == head.fles) fi
            nruter
        head.fles = current rav
        :(eurT) elihw // while true
            (data.current)tnirp
            next.current = current
            :(head.fles == current) fi
                kaerb

:()niam fed
    ()LinkedList = linklist rav
    (2)appendDat.linklist
    (4)appendDat.linklist
    ()printList.linklist // will print 2,4

```

**Ascii art by (in order): Copypasta, regality, Donovan Bake, Riitta Rasimus, b'ger