Munchkin+

Language Summary and Example Programs

Version 0.1

Dayna Eidle

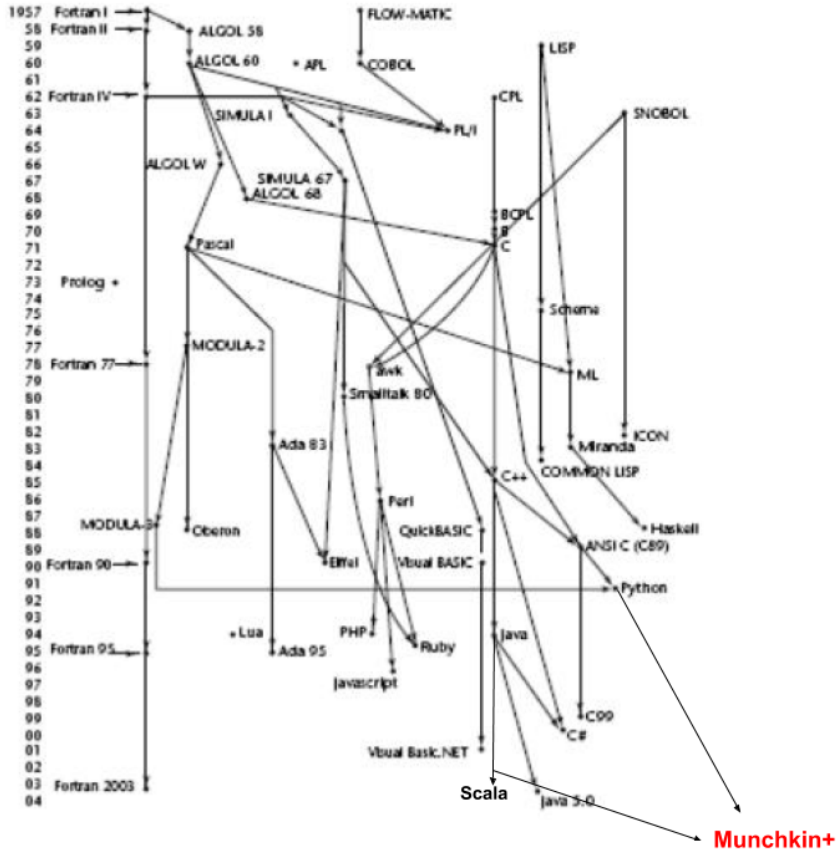
Theory of Programming Languages

1. Introduction

Munchkin+ is a modern, object oriented programming language based on Python and Scala. Munchkin+ works best with the dOnutS Operating System and lends itself to causing snacking on munchkins. It differs from Python and Scala in these ways:

1. Semicolons at the ends of lines are required.
2. If statements and any type of loop have begin and end lines. For both types, the begin comes after the conditional statement and the end comes after the last statement in the block, both being in line with the conditional statement. Both begin and end are followed by what they are referring to (ex. A for loop begin and end would say, begin for, end for).
3. There are functions and procedures. Functions return something, procedures do not.
4. Variables must have a declared type, however variables used as the counter in a for loop are assumed to be integers, unless otherwise specified.
5. Constants can be defined for variables that remain unchanged throughout execution;

1.1. Geneaology



1.2. “Hello World”

The program that is essentially a right of passage in any language: the Hello World program.

File: `helloWorld.py`

```
program helloWorld:
    print("Hello World");
```

1.3. Program Structure

Munchkin+ key organizational concepts:

- Each file is to be given a program name, defined at the beginning of the program. The program contains all classes, functions, and variable definitions. This is not necssarily in place of a main function, because main functions are still encouraged in Munchkin+.
- Any import statements for modules or other classes should be immediately following the program name header.
- Any code nested inside something else (ex. Code in a function, for loops, if statements), must be tabbed or spaced correctly. Tabs and spaces are both permitted, however it must be the same throughout the whole program.

File: `bankAccount.py`

```
program myBankAccount:
      class bankAccount:
            def __init__(self, fName, lName, accNum, balance=0):
                  self.fName = fName;
                  self.lName = lname;
                  self.accNum = accNum;
                  self.balance = balance;

            def procedure deposit(self, amount):
                  balance += amount;

            def procedure withdraw(self, amount):
                  balance -= amount;

            def procedure displayAccount(self):
                  print("Name: " + fname + " " + lname);
                  print("Account number: " + accNum);
                  print("Balance: " + balance);
```

```
acc1 = bankAccount("Dayna", "Eidle", "123456", 1000);

print(acc1.balance);
#prints 1000

acc1.deposit(100);

print(acc1.balance);
#prints 1100
```

The bankAccount program creates a bankAccount class with properties for first name, last name, account number, and balance. Balance is initialized to zero. It also has three methods: a deposit procedure, a withdraw procedure and a display account info procedure. The first two procedures modify the balance in the account.

1.4. Types and Variables

There are two kinds of types in Munchkin+: ***value types*** and ***reference types***. Value types hold data values within their own memory space, reference types contain a reference (or address) to the value being stored. The different possible data types for each are listed in parta 3.1 and 3.2.

1.5. Statements Differing From Python and Scala

Expression Statement

```
program expression:
    i: int = 10;
    x: double;
    name: string = "Dayna";
```

For loop

```
program forLoop:
    for (i = 0; i < 5; i++):
    begin for
        print(i);
    end for
```

While loop

```
program whileStmt:
    counter: int = 10;
    while (counter > 0):
    begin while
        print(counter);
        counter -= 1;
    end while
```

If statement

```
program ifElse:
    num1: int = 1;
    num2: int = 2;
    if (num1 > num2):
    begin if
        print("The first number is greater.");
    elif (num2 > num1):
        print("The second number is greater.");
    else:
        print("These numbers are equal.");
    end if
```

Constant definition

```
program constDef:
    const PI: double = 3.14159267;
```

String Concatenation

-Method 1:

```
program concat:
    name: string = "Dayna";
    newStr: string;
    newStr = "My name is " + name;
```

-Method 2:

```
program concat:
    num1: int = 1;
    num2: int = 2;
    printf("%x is less than %y", num1, num2);
    #prints 1 is less than 2
```

Function definition

```
program funct:
    def function add1(x):
        x += 1;
        return x;
```

Procedure definition

```
program proc:
    def procedure printSum(x, y):
        print(x + y);
```

Line continuation

```
program longLine:
    longString: string = (“This is a very long string and
                           so I need to break it up onto
                           multiple lines.”);
```

2. Lexical Structure

2.1. Programs

A Munchkin+ program consists of one or more source files made up of Unicode characters, all of which must have their own program definition. A program is compiled using three steps:

1. Transformation, which converts a file from a particular character repetoire and encoding scheme into a sequence of Unicode characters.
2. Lexical Analysis, which translates a stream of Unicode input characters into a stream of tokens.
3. Syntactic analysis, which translates the stream of tokens into executable code.

2.2. Grammars

2.2.1. Lexical Grammar of Munchkin+

* In some cases, the | is not part of the grammar, it is a separator between options.

<identifier>	⇒	_<identifier>
	⇒	<letter>
	⇒	<letter> <letter_list>
	⇒	<letter> <digit_list>
<digit_list>	⇒	<digit_list> <digit>
	⇒	<digit>
<digit>	⇒	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter_list>	⇒	<letter_list> <letter>
	⇒	<letter>
<letter>	⇒	*a….f*
	⇒	A….F
<operator>	⇒	+ | - | / | * | += | -=
<symbols>	⇒	( ) | ; | : | [ ]

2.2.2. Syntactic Grammar of Munchkin+

<program_stmt> ⇒
```
program <identifier>:
    <stmt>;
```
<assignment> ⇒
```
<identifier>: <type> = <value>;
```
<function_stmt> ⇒
```
def function <identifier>:
    <stmt>;
    return <stmt>;
```
<procedure_stmt> ⇒
```
def procedure <identifier>:
    <stmt>;
```
<if_else_stmt> ⇒
```
if (<condition>):
begin if
    <stmt>;
elif (<condition>):
    <stmt>;
else:
    <stmt>;
end if
```
<while_stmt> ⇒
```
while (<condition>):
    begin while
        <stmt>;
    end while
```
<for_stmt> ⇒
```
for(<identifier>; <condition>; <modification>):
    begin for
        <stmt>;
    end for
```

2.3. Lexical Analysis

2.3.1. Comments

There are two types of comments in Munchkin+: ***single line comments*** and ***multi-line comments***. The methods for writing each are shown below.

Single line comments

```
program comment:
    #This is a comment.
```

Multi-line comments

```
program comment:
    ~*These are all in the comments
    of my program because of the multi-line
    comment delimiter*~
```

*I don't see the point of nested comments and *simplicity is key* so there is no support for nested comments in Munchkin+. Just use a multi-line comment or write two separate comments. Everything will be ok. I promise.

2.4. Tokens

There are several kinds of tokens in Munchkin+:

- **Identifiers -** Commonly known as variables. These are names for symbols and values in a program that can be set and changed throughout execution.
- **Keywords/Reserved Words -** These are used throughout the program for syntax and stucture. (ex. if/else)
- **Literals -** The values given to variables.
- **Operators -** The punctuation which performs operation on numbers and variables. (ex. +, %)
- **Constants -** Variables definied (normally at the beginning of the program) that remain the same value throughout execution. They cannot change.
- **Separators -** White space or comments. These separate other tokens.

2.4.1. Keywords different from Python and Scala

A ***keyword*** is an identifier-like sequence of characters that is reserved, and cannot be used as an identifier. All Munchkin+ keywords are lowercase, however any uppercase versions of keywords are still prohibited as identifiers.

Removed keywords:
var val nonlocal lazy(because you should never be lazy)

New keywords:
begin end program procedure function rem const munchkin

Modified keywords:

New	**Old**
true	True - Python
false	False - Python
elif	else if - Scala
mod	Floor.mod - Scala, % - Python
and	&& - Scala
or	|| - Scala

3. Types

3.1. Value Type

The different value types in Munchkin+ are: ***int, double,*** and ***boolean***. ***double*** replaces Python and Scala's float type.

3.2. Reference Type

The different reference types in Munchkin+ are: ***string, list,*** and ***dict*** (for dictionaries).

list is the same as Python's list type, but replaces Python's tuple and Scala's Array type.

4. Example Programs

4.1. CeasarCipher (Encrypt, Decrypt, and Solve)

File: `caesarCipher.py`

```
program caesar:
    def procedure main():
        originalStr : string = "Hi my name is Dayna";
        println("Original: " + originalStr);
        encryptedStr : string  = encrypt(originalStr, 5);
        println("Encrypted: " + encryptedStr);
        decryptedStr : string = decrypt(encryptedStr, 5);
        println("Decrypted: " + decryptedStr);
        solve("DOG", 26);

    #encrypt a string
    def function encrypt(str, shiftAmount):
        #make string uppercase
        strUpper : string = str.map(c => c.toUpper);
        #map string to shifted characters using shift function
        newStr: string = strUpper.map(x => shift(x, shiftAmount));
        return newStr;

    #decrypt a string
    def function decrypt(str, shiftAmount):
        #make string uppercase
        strUpper: string = str.map(c => c.toUpper);
        #call encrypt and pass in negative shiftamount
        newStr: string = encrypt(strUpper, (shiftAmount * -1));
        return newStr;

     #shift a character
     def function shift(char, shiftAmount):
         #if character is a space don't change anything
         if (char.toInt == 32):
         begin if
```

```
            return char;
        else:
            ~*otherwise shift the character by shiftAmount and
handle overflow/wraparound*~
         newChar: string = (((((char.toInt - 'A'.toInt) +
shiftAmount) mod 26) + 'A'.toInt).toChar);
         return newChar;
         end if

    def procedure solve(str, maxShiftValue):
        newStr : string = "";
        shiftVal : int = maxShiftValue;

        #call encrypt for maxShiftValue -> 0 and print results
        while (shiftVal > 0):
        begin while
            newStr = encrypt(str, shiftVal);
            println(shiftVal + ": " + newStr);
            shiftVal = shiftVal - 1;
        end while
```

4.2. Factorial

File: `factorial.py`

```
program factorial:
    factNum: int = 10;
    printf("The factorial of %x is %y", factNum, factorial(factNum));
    #factorial function
    def function factorial(factNum):
        fact: int = 1;
        for (i = 1; i < factNum+1; i++):
            fact = fact * i;
        return fact;
```

4.3. Bubble Sort

File: bubbleSort.py

```
program bubbleSort:
    arr: list = [32, 15, 5, 60, 2, 10];
    arr = bubbleSort(arr);
    for (i = 0; i < len(arr); i++):
    begin for
        print(arr[i]);
    end for

    #function for bubble sort
    def function bubbleSort(arr):
        length: int = len(arr);
        #loop through the whole array
        for (i = 0; i < length; i++):
        begin for
            for (j=0; j < (length - i -1); j++):
            begin for
                #swap the elements if the lower indexed on is greater
                if (arr[j] > arr[j + 1]):
                begin if
                    temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
                end if
            end for
        end for
        return arr;
```

4.4. Fibonacci

File: `fibonacci.py`

```
program fibonacci:
    termNum: int = 16;
    def procedure fibonacci(termNum):
        a: int = 0;
        b: int = 1;
        temp: int;
        for (i = 0; i < termNum; i++):
        begin for
            temp = a;
            a = b;
            println(a);
            b = temp + b;
        end for
```

4.5. Queue

File: `queue.py`

```
program queueProg:
    class Queue:
        def __init__(self):
            self.items = [];

        def function isEmpty(self):
            return self.items == [];

        def procedure dequeue(self):
            self.items.pop();

        def procedure enqueue(self, item):
            self.items.append(item);
```

```
        def function getSize(self):
            return len(self.items);

    names = Queue();
    names.enqueue("Dayna");
    names.enqueue("Ali");
    names.enqueue("Dan");
    print(names.getSize());
    #prints 3
    names.dequeue();
    print(names.getSize());
    #prints 2
```