CMPT 332 – Theory of Programming Languages
Dr. Labouseur

SpellVoid

A spell-based programming language

Antonio Lopez
Antonio.Lopez2@marist.edu

May 6th, 2022

Table of Contents

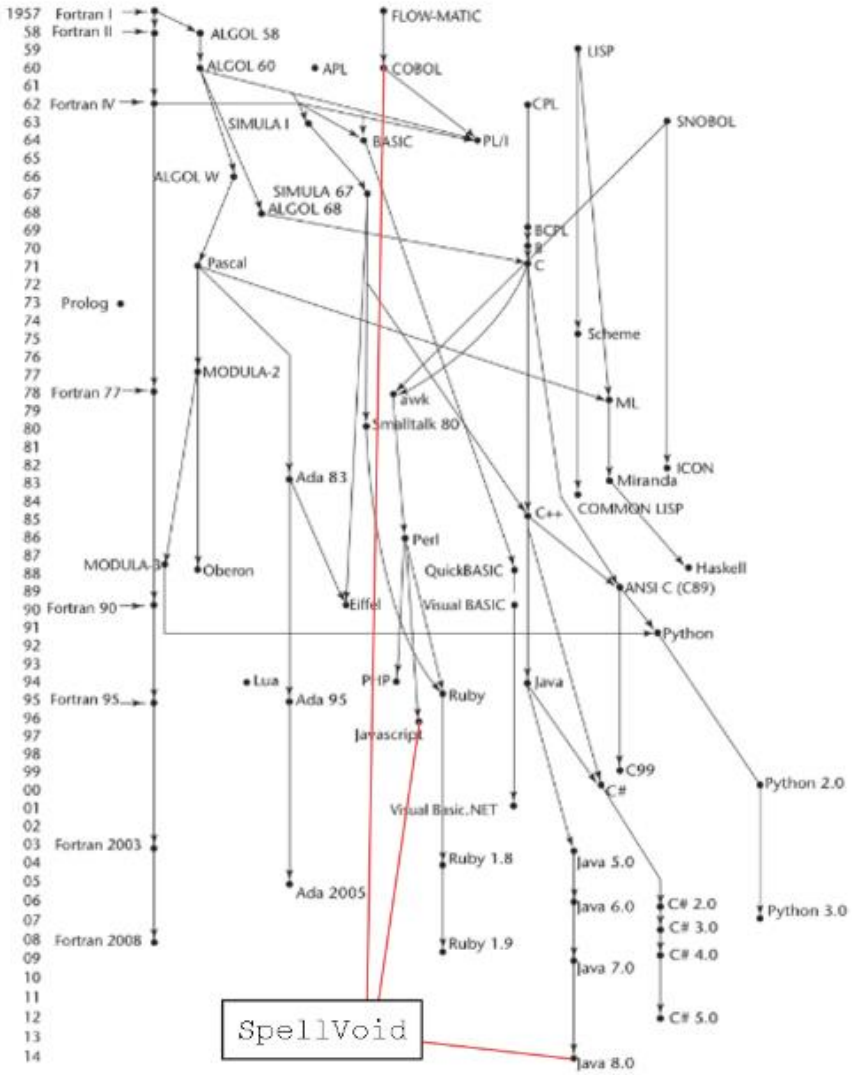
1 Introduction ……3
1.1 Genealogy……4
1.2 Hello World……5
1.3 Program Structure……6
1.4 Types and Variables……8
1.5 Visibility……8
1.6 Differing Statements……8
2 Lexical Structure……10
2.1 Programs……10
2.2 Grammars……10
2.2.1 Lexical grammar differences……10
2.2.2 Syntactic grammar differences……10
2.3 Lexical Analysis……11
2.3.1 Comments……11
2.4 Tokens……11
2.4.1 Keyword differences……11
3 Type System……12
3.1 Type Rules……12
3.2 Different Value……13
Types……13
3.3 Different Reference Types……13
4 Sample Programs……14
Encrypt……14
Decrypt……14
Factorial……14
BubbleSort……14
Fibonacci Sequence……16
Combat Simulation……17

1 Introduction

For those that are entering the world of wizardry, a complex understanding of abstract concepts (and putting up with old complex bullshit that doesn't make sense) is required to truly master the arcane arts. In an effort to familiarize students with these techniques, the head academy has developed SpellVoid as a modern-day medium to introduce pupils considering a future in wizardry to the basic concepts. SpellVoid utilizes a mixture of a little bit of Java, JavaScript and COBOL, differing in the following ways:

1. Periods are placed at the end of single-line statements. While they are entirely optional (whitespace is used from JavaScript), they can assist in debugging and readability.
2. Null values are now called "void," to gently remind students to not touch things outside this realm.
3. Functions are replaced with "spells", with the main function being the "spellbook."
4. Writing results instead of printing them reinforces concepts learned in Scribing 101.
5. Each program starts with a "cast" and ends with a "resolve," mirroring the general process of casting and resolving spells.

1.1 Genealogy



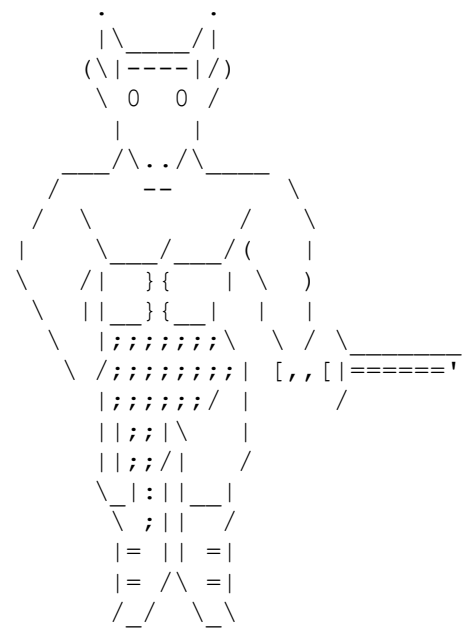
1.2 Hello World

```
------------------------------------------------------------------

SPELL-ID. SPELLBOOK.
CAST.
   WRITE “HELLO WORLD”.
   STOP RUN.
RESOLVE SPELLBOOK.

------------------------------------------------------------------
```

1.3 Program Structure

SpellVoid programs use the following concepts:

1. Rather than being a procedural language, SpellVoid starts inside any function named Spellbook function (borrowing a bit from Java). There can only be one function named Spellbook, otherwise an error is formed.
2. Classes can be declared using "SUBCLASS-ID" where you would put the program name. The constructor for the class will be placed where variables are stored under SUBCLASS-STORAGE.
3. A return type is required for non-spellbook spells (functions). If nothing is supposed to be returned, return void.
4. SpellVoid uses indentation to bind the code, rather than using JavaScript's brackets. It also makes use of periods instead of semicolons.
5. Every function has to have a "cast" and "resolve" to indicate where they start and stop respectively



Sample Program:

```
SUBCLASS-ID. SCHOOL.
SUBCLASS-STORAGE SECTION.
VERB Name.
VERB School.
SOM Grade.
CAST.
   FAIL-PASS (x).
      IF x.Grade > 80 THEN.
         WRITE "You passed!".
      ELSE.
         WRITE "Have you considered becoming a sellsword?".
   RETURN VOID.
RESOLVE SCHOOL.

SPELL-ID. SPELLBOOK.
LOCAL-STORAGE SECTION.
SUBCLASS MySchool   SCHOOL(99).
VERB Name         VALUE "Antonio".
VERB School       VALUE "Necromancy".
SOM Grade         VALUE 85.
CAST.
   BIND (Name, School, Grade) to MySchool.
   CALL 'FAIL-PASS' USING MySchool.
   STOP RUN.
RESOLVE SPELLBOOK.
```

This creates a subclass for the school with various verbal and somatic components (elaborated on in 3.1), as well as the constructor being found within the "class-storage" section. The class has a function "fail-pass" to check the Grade value. In the spellbook, it creates a subclass MySchool, then calls the fail-pass function using said subclass. This demonstrates an if/else statement, checking numerical values, and using whitespace correctly.

1.4 Types and Variables

SpellVoid contains two types: value types and reference types. Variables of value types directly contain their data whereas variables of reference types store references to their data, the latter being known as objects. With reference types, it is possible for two variables to reference the same object and thus possible for operations on one variable to affect the object referenced by the other variable.

1.5 Visibility

All functions and methods in SpellVoid are public, allowing any function in the file to access them.

1.6 Statements Differing from JavaScript and COBOL

Statement	Example

Expression	SPELL-ID. SPELLBOOK.
	LOCAL-STORAGE SECTION.
	SOM x.
	SOM y.
	CAST.
	  WRITE “Adding ”, x, “ + “, y
	  BIND 5 TO x.
	  BIND 3 TO y.
	  BIND x + y TO y.
	  WRITE “Answer: “, y.
	  STOP RUN.
	RESOLVE.

If/Else Statement	SPELL-ID. SPELLBOOK.
	LOCAL-STORAGE SECTION.
	SOM x.
	SOM y.
	CAST.
	  BIND 12 TO x.
	  BIND 6 TO y.
	  IF x IS > y THEN.
	  WRITE x, “ is greater than ”, y.
	  ELSE
	  WRITE y, “ is greater than ”, x.
	STOP RUN.
	RESOLVE.

Print/PrintIn	WRITE “Hello World”. WRITELINE “Hello World”.

Comments	SPELL-ID. SPELLBOOK.
	LOCAL-STORAGE SECTION.
	SOM x.
	SOM y.
	CAST.
	  << Setting values for variables >>
	  BIND 12 TO x.
	  BIND 6 TO y.
	  <<
	  Tests to see if x is greater than y,
	  then prints as such
	  >>
	  IF x IS > y THEN.
	  WRITE x, " is greater than ", y.
	  ELSE.
	  WRITE y, " is greater than ", x.
	STOP RUN.
	RESOLVE.

For loops	SPELL-ID. SPELLBOOK.
	LOCAL-STORAGE SECTION.
	SOM x VALUE 1.
	SOM y VALUE 9.
	CAST.
	  CONCENTRATE UNTIL x > y.
	  WRITE x.
	  BIND 1 + x TO x.
	  STOP RUN.
	RESOLVE.

2 Lexical Structure

2.1 Programs

A SpellVoid program consists of one or more source files. A source file is an ordered sequence of (probably Unicode) characters.

Conceptually speaking, a hybrid program is compiled using four steps:

1. Transformation, which converts a file from a particular character repertoire 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.
4. Interpretation, which translates the high-level language to intermediate language using an interpreter.

2.2 Grammars

This specification presents the syntax of SpellVoid, and where it differs from JavaScript and COBOL.

2.2.1 Lexical Grammar Where Different from JavaScript and COBOL

<assignmentOperator> -> BIND TO
<endOfLineCharacter> -> .
<print> -> WRITE | WRITELN
<comments> -> << … …>>
<begin/end function> -> CAST. … … RESOLVE.
<main function> -> SPELL-ID. SPELLBOOK.

2.2.2 Syntactic Grammar Where Different from JavaScript and COBOL

<function> -> SPELL-ID. <name>. <storage-type>.
<variable declaration> -> <type> <name> VALUE <value>.
<parameter-list> -> LINKAGE SECTION <parameter> <parameter-list>
-> <parameter>
<if/else-statement> -> IF (case) THEN. (expr). ELSE. (expr2).

2.3 Lexical Analysis

2.3.1 Comments

SpellVoid supports two types of comments: single-line and multi-line. Both styles of commenting use the same syntax, which is << >> both at the start and the end. Nested comments were disallowed in order to have some degree of readability as well as ease the process of debugging.

2.4 Tokens

There are several kinds of tokens: identifiers, keywords, literals, operators, and punctuators. White space and comments are not tokens, though they act as separators for tokens where needed.

tokens:

identifier
keyword
integer-literal
real-literal
character-literal
string-literal
operator-or-punctuator

2.4.1 Keywords Different From COBOL and JavaScript

A keyword is an identifier-like sequence of characters that is reserved, and cannot be used as an identifier.

New keywords: write/writeln, verb, som, fate, cast, resolve

Removed keywords: print/println, str, int, float, double, bool, procedure, end program

3 Type System

SpellVoid uses a strong static type system, meaning that type errors are caught and expressed to the programmer during compilation. Static typing means early binding compile-time type checking.

3.1 Type Rules

Assignment:

⊢ e1: T
⊢ e2: T
T is a primitive type
⊢ BIND e1 TO e2: T

Arithmetic:

S ⊢ e1: T
S ⊢ e2: T
T is a primitive type
S ⊢ e1 + e2: T
OR
S ⊢ e1 - e2: T
OR
S ⊢ e1 * e2: T
OR
S ⊢ e1 / e2: T

Comparisons:

⊢ e1: T
⊢ e2: T
T is a primitive type
⊢ e1 == e2: fate
OR
⊢ e1 > e2: fate
OR
⊢ e1 < e2: fate
OR
⊢ e1 >= e2: fate
OR
⊢ e1 <= e2: fate
OR
⊢ e1 != e2: fate

3.2 Value Types (Different than JavaScript or COBOL)

Verb: Verb (standing for verbal) represents a string, which is a set of characters.

Ex. VERB COW_NOISE VALUE "Moo".

Syl: Syl (standing for syllable) represents a Unicode character, a single part of a verbal component.

Ex. SYL FIRST_LETTER VALUE 'A'.

Som: Som (standing for somatic) represents a general numerical value. This can be written as an int, decimal or fractional component.

Ex. SOM AGE VALUE 20.

Fate: Fate is a value that can either be a boon (true) or a bane (false)

Ex. FATE RESULT VALUE BANE.

Void: A null value, representing the vast emptiness of time (also representing no value)

3.3 Reference Types (different than JavaScript or COBOL)

Ritual: Rituals represent an array of data with variable length.

```
Ex. RITUAL WEEKDAYS.
        VERB VALUE MONDAY.
        VERB VALUE TUESDAY.
        VERB VALUE WEDNESDAY.
        VERB VALUE THURSDAY.
        VERB VALUE FRIDAY.
    END RITUAL.
```

4 Example Programs

Encrypt

```
SPELL ID. ENCRYPT.
LINKAGE SECTION.
VERB Str.
SOM Amount.
WORKING-STORAGE SECTION.
VERB Newcode.
SYL Code.
CAST.
    BIND Str.TOUPPERCASE TO Str.
    BIND Str.CHARCODEAT(0) TO Code.

    IF Code >= 65 AND Code <= 90 THEN.
        BIND (Code - 65 + Amount) % 26 + 65 TO Code.
    BIND Code.TOCHAR.TOSTRING TO Newcode.

    IF Str.LENGTH > 1 THEN.
        RETURN Newcode + CALL 'ENCRYPT' USING Str.SUBSTRING(1) Amount.
    RETURN Newcode.
    EXIT PROGRAM.
RESOLVE.
```

Decrypt

```
SPELL ID. DECRYPT.
LINKAGE SECTION.
VERB Str.
SOM Amount.
CAST.
    BIND (26 - Amount) TO Shift.
    RETURN CALL 'ENCRYPT' USING Amount.
    EXIT PROGRAM.
RESOLVE.
```

Factorial

```
SPELL-ID. SPELLBOOK.
LOCAL-STORAGE SECTION.
SOM n      VALUE 10
SOM Result.
CAST.
    BIND n TO CALL 'FACTORIAL' USING n.
    WRITE "Factorial: ", n.
    STOP RUN.
RESOLVE.
```

```
SPELL-ID. FACTORIAL.
LINAKGE-STORAGE SECTION.
SOM n.
CAST.
    IF n == 0 THEN.
        RETURN 1.
    ELSE.
        RETURN n * CALL 'FACTORIAL' USING n - 1.
    EXIT PROGRAM.
RESOLVE.
```

BubbleSort

```
SPELL-ID. SPELLBOOK.
LOCAL-STORAGE SECTION.
RITUAL arr.
    SOM VALUE 1
    SOM VALUE 4
    SOM VALUE 2
    SOM VALUE 5
    SOM VALUE -2
END RITUAL.
CAST.
    CALL 'BUBBLESORT' USING arr.
    STOP RUN.
RESOLVE.
```

```
SPELL-ID. FACTORIAL.
LINAKGE-STORAGE SECTION.
RITUAL arr.
WORKING-STORAGE SECTION.
SOM i      VALUE 0.
SOM j      VALUE 0.
SOM temp.
```

```
CAST.
    CONCENTRATE UNTIL i > arr.LENGTH.
        CONCENTRATE UNTIL j > arr.LENGTH - i - 1.
            IF arr[j] > arr[j+1] THEN.
                BIND temp TO arr[j].
                BIND arr[j] TO arr[j + 1].
                BIND arr[j + 1] TO temp.
    WRITE "Sorted array: ", arr.
    RETURN VOID.
    EXIT PROGRAM.
RESOLVE.
```

Fibonnaci Sequence

```
SPELL-ID. SPELLBOOK.
LOCAL-STORAGE SECTION.
SOM number     VALUE 8.
CAST.
    CALL 'FIBONNACI' USING number.
    STOP RUN.
RESOLVE.
```

```
SPELL-ID. FIBONNACI.
LINAKGE-STORAGE SECTION.
SOM number.
WORKING-STORAGE SECTION.
SOM i      VALUE 0.
SOM num1   VALUE 0.
SOM num2   VALUE 1.
SOM sum.
CAST.
    CONCENTRATE UNTIL i > num.
        BIND num1 + num2 TO sum.
        BIND num2 TO num1.
        BIND sum TO num2.
    WRITE "Fibonnaci(", num, "): ", num2.
    RETURN VOID.
    EXIT PROGRAM.
RESOLVE.
```

Combat Simulation

```
SPELL-ID. SPELLBOOK.
LOCAL-STORAGE SECTION.
SOM p1_health     VALUE 20.
SOM p2_health     VALUE 20.
SOM p1_damage     VALUE 20.
SOM p2_damage     VALUE 20.
CAST.
    WRITE "Fight start!".
    CONCENTRATE UNTIL p1_health <= 0 OR p2_health <= 0.
        BIND CALL 'ATTACK' USING VOID TO p1_damage.
        WRITE "Player 1 deals ", p1_damage, "to Player 2."
        BIND p2_health - p1_damage TO p2_health.
        BIND CALL 'ATTACK' USING VOID TO p2_damage.
        WRITE "Player 2 deals ", p2_damage, "to Player 1."
        BIND p1_health - p2_damage TO p1_health.

    << Calculates result of fight, both players can lose >>
    IF p1_health <= 0 && p2_health > 0 THEN.
        WRITE "Player 2 wins!".
    ELSE IF p2_health <= 0 && p1_health > 0 THEN.
        WRITE "Player 1 wins!".
    ELSE.
        WRITE "Double K.O! No one wins."
    STOP RUN.
RESOLVE.

SPELL-ID. ATTACK.
WORKING-STORAGE SECTION.
SOM result.
CAST.
    BIND 1 + MATH.FLOOR(MATH.RANDOM()*6) TO result.
    RETURN result.
    EXIT PROGRAM.
RESOLVE.
```
