1 Types

1.1 Primitives

Primitive types are types that do not have fields. Instead, they merely have values. These can take the form:

- `void`: a type representing a the return type of a function that does not return a value. It has no enumerated values and variables of type `void` cannot be created.
- `number`: a numeric type. This includes both integers and floating point numbers. Calculations internally are handled using high precision, large numbers.
- `character`: a character type. This is basically an integer that will be shown as a character.
- `bool`: a boolean type. `true` or `false`.
- `type`: a type type. This type is the type of the *values* (not types!) `number`, `type`, `array of string`.

1.2 Defined types

Defined types are the types of structures with fields. These fields can be themselves defined types or primitives.

1.3 Compound types

Compound types are types that take other types as arguments, similar to Haskell's type constructors. A common example is `array` which can only be declared as `array of T`, where `T` is the type stored in the array. Compound types are always defined types, because types are fields.

2 Variables

Although variables without prefixes would normally be beneficial in a natural-like language like 41++, they could possibly cause ambiguity with Arbitrary Syntax Functions. Therefore, all variables must start with the underscore character _. The other restriction is that variable names cannot contain spaces or start with a number or `'`.

3 Expressions

3.1 Literals

Literals are expressions that are hard-coded into the code. They take one of four forms.

3.2 Numeric Literals

These must start with a digit, a plus sign, a minus sign, or a period.

3.3 Boolean Literals

Either `true` or `false`.

3.4 Character and String Literals

These must start and end with a single quote `'`. What is in between is interpreted as a string. To use an actual single quote mark, use `\'`. Standard escapes can also be used. Determining the type of a string falls into three cases.

- `''`: This is automatically a string literal representing an empty string.
- A single character: depending on the context, this is interpreted as a string or character.
- Multiple characters: always a string.

3.4.1 Examples

- `2`, `-56543234565`, `41`, `-.02345654321`, `12.`: Numeric literals.
- `'"'`, `'1'`, `'\r'`, `'\n'`, `'\t'`, `'\0123'`: Character literals
- `''`, `'\'\''`, `'41++'`: String literals.

3.5 Algebraic Expressions

Algebraic Expression	Type
=	a = a -> bool
>	number > number -> bool and char > char -> bool
<	number < number -> bool and char < char -> bool
>=	number >= number -> bool and char >= char -> bool
<=	number <= number -> bool and char <= char -> bool
+	number + number -> number and char + char -> char and char + number -> number
-	number - number -> number and char - char -> char and char - number -> number
*	number * number = number
/	number / number = number[1]
//	number // number = number[2]
%	number % number = number[3]

3.6 Function Expressions

Any function call of a non-void returning function can be used as an expression.

3.7 The Role of Parentheses

While 41++ is designed to be a English-like language, it is often very difficult to tease out syntactical ambiguity without parentheses. Therefore, in 41++, parentheses must surround any value that is not a single word or string literal.

4 Statements

There are a limited number of valid statement forms. All start with a capital letter and end with a period.

4.1 Definition

4.1.1 Declaration

A minimal declaration simply provides a variable with a name and associates it with a type.

```
Define a[n] <type> called <name>.
```

[1] this is standard division. 11/2 = 5.5
[2] this is floor division. 11/2 = 5, 1.5 / 1 = 1, -1.5 / 1.2 = -1
[3] remainder

4.1.2 Field Initialization

A variable can also have its fields initialized. It can also be directly set to a value by using the special field `value`.

```
Define a[n] <type> called <name> with a[n] <field1> of <value1>,
a[n] <field2> of <value2>, and a[n] <field3> of <value3>.
```

Commas and `and` are all technically unnecessary, but included to insure readability. Similarly, `a` and `an` are equivalent but both are included to avoid statements like `Define a integer called x`.

4.1.3 Examples

```
Define an integer called _x.  Define a string called _name
with a value of '41++'.  Define a matrix called _M with a
width of 3 and a height of 2.  Define a matrix called _M2
with a value of _M.
```

4.2 Assignment

Assignment comes in two forms, value assignment and field assignment.

```
Set the <field> of <name> to <value>.
```

4.3 Function Calls

Any function call can be a statement in one of these two ways.

- `Run <non-void function expression>.`
- `<Void expression>.`

5 Control Flow

5.1 If

The syntax of If is as follows.

```
If <expression>:  <Statement executed in case of expression>.
```

An Otherwise block can also be appended.

```
If <expression>:  <Statement executed in case of expression>;
otherwise:  <Statement executed otherwise>.
```

5.2 While

The syntax of While is as follows.

```
While <expression>:  <Statement>.
```

5.3 Statement Concatenation

Notice that control flow statements above only take a single statement as an argument. However, having to define a procedure for every small set of instructions would be difficult. Instead, we can use the following syntax to convert multiple statements into a single one:

```
<First statement>; <Second statement>; <Third statement>.
```

6 Functions

6.1 Declaration

A function is declared as follows:

```
Define a function called <function expression> that takes
a[n] <type1> called <field1>, a[n] <type2> called <field2>,
and a[n] <type3> called <field3>[ and outputs a <return type>].
```

Where, in the function expression, variable names are enumerated as they appear in the type declarations later. Note that the function expression doesn't actually have a defined syntax. For example, this is a valid declaration:

```
Define a function is _n prime that takes a number called _n
and outputs a bool.
```

Note that if `and outputs a <return type>` is omitted, then the return type defaults to `void`.

6.2 Body

The body consists of statements, in which any variables that are made cannot be used outside the function.

6.3 Conclusion

A function return is declared as follows.

```
Exit the function[ and output <output>].
```

Note that the output must be omitted if the return type is `void` because `void` has no values.

7 Structures

A structure is declared as follows:

```
Define a structure called <structure expression>; which contains
a[n] <type1> called <field1>, a[n] <type2> called <field2>,
and a[n] <type3> called <field3>.
```

The structure expression may contain variables, whose `type` is automatically `type`. A concrete example is probably best in this situation.

```
Define a structure called _k maps to _v; which contains an
_k called key and a _v called value.
```

This defines an entry structure that can be later defined as follows.

```
Define a (number maps to number) entry with a key of 2 and
a value of 2.
```

Note the use of types to replace `_a` and `_b`.