Fall 2015 COMS 4115

Programming Languages & Translators

Language Reference Manual

StoryBook

Authors

Nina Baculinao (nb2406) - Systems Architect

Beth Green (blg2132) - Language Guru

Anna Lawson (aal2150) - Testing

Pratishta Yerakala (py2211) - Manager

Contents

1 Introduction

Once upon a time, the creators of Storybook were learning how to code for the first time. At first, they fumbled with the tricky and alien syntax. It took a while for them to discover the joyful creativity of computer programming.

StoryBook is a programming language targeted toward novice programmers who are just starting to understand the basics of computer science and computational thinking. The language uses intuitive, "story-like" syntax and structure to make object-oriented programming easier for children and adult-beginners to read and implement. The backend of StoryBook generates Java code.

2 Syntax Notation

The syntax notation of this manual is as follows. Any literals or words that belong to the StoryBook language will be written in `monospaced typeface.` Syntactic categories are written in *italic*.

Grammar patterns are expressed throughout the document using regular expressions. r* means the pattern r may appear zero or more times, r+ means r will appear one or more times, and r? means r will appear one or zero times. r1|r2 means that the pattern has either r1 or r2. r1r2 means that the pattern r1 is concatenated with r2.

3 Lexical Conventions

StoryBook programs are lexically composed of three elements: comments, tokens, and whitespace.

3.1 Comments

Symbol	Description	Example
`~~`	single line comment	`~~Single line comment`
`~ ~`	block comment	`~Multi-line comment~`

3.2 Tokens

A token in StoryBook is a group of characters that hold meaning when considered as a group. These consist of *keywords*, *identifiers*, *operators*, *separators*, and *constants*.

3.2.1 Keywords

These are the StoryBook keywords:

```
Plot, Chapter, Character, Action, subtype, trait, list, number,
```

`words, letter, tof, new, returns, endwith, say, repeat while, repeat for, if, else, elseif, then, is, true, false, and, or, not, null`

3.2.2 Identifiers

identifier → (['A'-'Z' 'a'-'z'] | [0-9] | _)*

Identifiers are a collection of characters, numbers, and/or underscores. The characters are the ASCII characters 'a'-'z' and 'A' -'Z', numbers are digits 0-9, and underscore '_'. StoryBook is case sensitive. Identifiers hold values that are of the type to which they are assigned.

3.2.3 Operators

operator → `+`

`-`

`*`

`/`

`%`

`<`

`>`

`<=`

`>=`

`,`

`'s`

`and`

`or`

`not`

In StoryBook there are arithmetic, comparison, boolean, `list`, and `Character` operators. The syntax and use of these expressions are described in 6.2, 6.3 and 6.4.

3.2.4 Constants

digit → `[0-9]*`

constant → `[1-9]` *digit**

`true`

`false`

Values in StoryBook that always have the same value include true, false, and digits 0-9.

3.2.5 Separators

separator → `;`

`.`

StoryBook uses `;` to separate items in a `list` data structure or in a list of function arguments. A `.` is used to mark the end of an expression.

3.2.6 Newlines
StoryBook uses newlines to identify the end of a single line comment. Otherwise, newlines are ignored by the compiler.

3.3 Whitespace
Tabs and spaces are used by StoryBookers to make their programs more readable. However, whitespace is ignored by the compiler.

4 Data Types
4.1 Primitive Data Types
There are five primitive data types in Storybook: `letter`, `words`, `tof`, `number`, and `list`.

Type	Definition
`letter`	- Single character
`words`	- Grouping of consecutive characters, a string
`tof`	- Boolean type, holds a value of `true` or `false`
`number`	- Any type of number: int, short, float, double, or long
`list`	- Can hold multiple instances of primitive or user-defined types; all values in a list must be of the same type.

4.1.1 Lists
While a list can be declared and used as a primitive data type, its methods can be called in the same way as character methods (see 8.2). For this reason, lists are treated as a special object in the compiler to separate them from other primitive data types.

4.2 Non-Primitive Data Types
A `Character` is a user-defined data type comprised of `traits` (instance variables) and `Actions` (methods). `Traits` can be of a primitive type or of a `Character` type, including itself. See section 7.1.2 for an example.

4.3 Scoping and Lifetime
A variable's scope is the block in which the variable is declared, with the exception of `traits`.

In the case of nested blocks, if a variable declared within an inner block and shares the same name as a variable declared in the outer block, then the variable declared in the inner block takes precedence, effectively overriding the one in the outer block. Thus, in this case, the outer block's variable with the shared name is inaccessible from the inner block. If two variables are declared within the same block level, consequently sharing the same scope, with the same name, the one declared later will take precedence and the earlier one will be inaccessible after the point of the later variable's declaration.

`traits` have the lifetime of their object. Local variables have a lifetime from their declaration's execution to when the program counter exits the block in which the variable was defined.

5 Purpose of Identifiers

An identifier is an alphanumeric sequence of characters that amounts to either a *keyword* or the name of a `Chapter`, `Character`, `Action` or a variable. This sections details the purpose and scope of the possible types of non-keyword identifiers.

5.1 Chapters

In Storybook, a `Chapter` is any function that is not the `Plot` (main function). `Chapters` enable users to create reusable and versatile blocks of code that can be called in the `Plot`. In this way, users can construct more concise `Plots` that are either comprised of or include sequences of `Chapter` calls. `Chapters` can take zero or more arguments. Each `Chapter` can have zero or one return value. All argument and return types must be declared in the `Chapter` header, as shown below.

```
Chapter foo() returns nothing {
  ~~Code here.
}

Chapter foo() returns number {
  ~~Code here.
  endwith (5).
}
```

5.2 Characters

In Storybook, classes are called `Characters`. `Characters` are user-defined data types that represent a type of object. Users can then instantiate `Character` objects of a specific `Character` type. Each `Character` object has its own copy of instance variables declared using the `trait` keyword and can perform `Actions`.

5.2.1 Subtypes

Inheritance can be employed to create `subtypes` of `Characters` and avoid duplication of code for shared functionality. This structure allows users to define reusable data types and to abstract the implementation details of story characters. `Characters` allow computer science novices to begin to understand the key concepts object-oriented programming in the familiar context of story characters.

5.3 Actions

`Actions` are methods that can be invoked on instances of a `Character`. `Actions` are defined inside the `Character` class definition.

5.4 Variables

In Storybook, variables are statically-typed. A variable is an identifier that is bound to a reference of a value of one of the following types: `Character`, `letter`, `words`, `tof`, `list`, or a `number`. Variables of type `number` are dynamically typed in that they can be initialized and re-assigned to any type of number. The variables in Storybook are mutable.

5.5 Traits

In StoryBook, `traits` represent the object-oriented concept of instance variables. `Traits` are variables that are defined at the scope of a `Character` type. Each instantiated object of that `Character` type has its own instance of each `trait`.

6 Expressions

This section describes the syntax of StoryBook *expressions*. StoryBook uses postfix, prefix, or infix *operators*. The precedence of expression operators mirrors the order of the major subsections of this section, highest precedence first. Within each subsection, the operators have the same precedence. The grammar of StoryBook incorporates the precedence and associativity of the operators.

6.1 Primary Expressions

```
primary-expr → constant
               identifier
               ( expression )
               [ expression ]
```

Primary expressions include *identifiers*, *constants*, or *expressions* that can be evaluated to a single value in parentheses.

6.1.1 Identifiers

An *identifier* for a variable is a primary expression, provided it has been fully declared and holds a value. A variable *a* is a primary expression whose type is the same as the type of *a*. Likewise, a `trait` *b* is a primary expression whose type is the same as the type of *b*. Evaluation of an identifier actually entails evaluation of the expression bound to that variable. Identifiers are described in section 3.2.2.

6.1.2 Constants

A *constant* is a primary expression with the same type as the type of the literal. See 3.2.4 for a discussion of constants.

6.1.3 Parenthesized Expressions

A parenthesized expression is a primary expression whose type and value are identical to the final evaluation of an un-parenthesized expression.

6.1.4 Lists

`list` → [*expression*$_1$ `;` ... `;` *expression*$_i$]
where $0 \leq i \leq 1$.
A `list` is a primary expression that can contain zero or more expressions. The expressions in a list must all be of the same type. An undeclared empty list `[]` holds no type until it stores at least one expression; at that point, it is assigned the same type as the first expression in the list.

6.2 Postfix Expressions

postfix-expr → *primary-expr*
list-postfix-expr `[` *expression* `]`
postfix-expr `(` *optional-list-of-parameters-expr* `)`
Character-or-list-postfix-expr `'s` *identifier*
Character-or-list-postfix-expr`,` *identifier*

The operators in postfix expressions group from left to right.

6.2.1 List Access

A postfix expression followed by an expression in square brackets is a postfix expression denoting a subscripted list reference. The expression `expr1[expr2]` denotes the accessing of `list` elements. First `expr1` is evaluated, then `expr2`, then the `[]` operator. It returns the value at the position denoted by `expr2` in the `list` denoted by `expr1`. Position numbers in the list begin at 1 and end with the length of the list. For instance:

```
words list pets is new words list[5] char* pets[5];
```

```
["cat", "dog", "lizard", "dragon", "unicorn"].
pets[1]. ~~cat, counting from the left
pets[5]. ~~unicorn, last element, counting from the left
pets[-2]. ~~dragon, counting from the right
```

6.2.2 Character Access

The `'s` operator is used to access a `Character`'s `traits`. The `,` operator is used to invoke a `Character`'s `Actions`.

```
words monsterName is Frankenstein's name.
Frankenstein, scare ("Boo!").
```

6.2.2 List Functions

`List` functions are invoked in the same way as `Character Actions`. The `,` operator is used to call the special `list` functions, which are discussed in section 8.2.

```
list1, length. ~~evaluates to the length of list1
```

6.2.3 Chapter Invocation

`Chapters` can be called in the scope in which they were created by the `Chapter` *identifier* and the appropriate arguments.

```
number result is Sum(1; 2).
~~calls a function called Sum that takes in two number arguments
```

6.3 Prefix Expressions

The only prefix operator in StoryBook is `not`, the logical negation expression.

6.3.1 Logical Negation

prefix-expr → `not`

The operand of the `not` operator must have a `tof` type. The result of the prefix expression `not true` is `false` and the value of `not false` is `true`. Use of `not` with comparison operators flips the condition of the comparison and is discussed in section 6.4.3.

6.4 Binary Operator Expressions

binary-expr → *expression*$_1$ *op expression*$_2$

The following categories of binary *operators* exist in StoryBook, and are listed in order of decreasing precedence: arithmetic, concatenation, comparison, logical, assignment and sequence.

6.4.1 Arithmetic Operators

Operator	**Description**	**Example**
`*`	multiply	`number fifty is 5*10. ~~fifty=50`
`/`	divide	`number five is 50/10. ~~five=5`
`%`	modulo	`number zero is 50%5. ~~zero=0`

`+`	add	`number x is 0. ~~x=0`
		`x is x+1. ~~ now x=1`

`-`	subtract	`number six is 10-4. ~~six=6`

The multiplicative operator `*`, the division operator `/` and the remainder operator `%` are all grouped left-to-right. The operands must have `number` type. The binary operator `*` denotes multiplication of the two operands. The binary `/` operator yields the quotient, which is always the result of floating point division of the first operand by the second. The `%` operator yields the remainder of a product of the floating point division. If the second operand is 0 for the `/` or `%` operator, the result is undefined.

Of lower precedence than the multiplicative operators, the additive operator + and subtractive operator − also group left-to-right. As long as the operands are of the `number` type, the result of the `+` operator is the sum of the operands. The + operator can also have operands of other types, in which case the function of the operator changes to concatenation, which is discussed in the next subsection. The result of the − operator is the difference of the operands. The operands for subtraction must be of `number` type.

6.4.2 Concatenation Operator

Pattern	**Description**	**Example**
words-expr + *words-expr*	concatenate strings	`words bestLanguageEver is "Story" + "Book". ~~bestLanguageEver evaluates to "StoryBook"`
words-expr + *number-expr* | *number-expr* + *words-expr*	concatenate string and number	`words title is "Alibaba and the " + 40 + " thieves".`
words-expr + *tof-expr* | *tof-expr* + *words-expr*	concatenate string and boolean	`"Today you are you! That is " + true + "r than " + true + "! There is no one alive who is you-er than you!"`

words-expr + *list-expr*
 | *list-expr* + *words-expr*
	concatenate string and list	`[3; 2; 1] + " Here I come!"`
		 `~~"3, 2, 1, Here I come!"`

list-expr + *list-expr*	concatenate lists	`words list colorsOfTheRainbow is ["red"; "orange"; "yellow"] + ["green"; "blue"; "indigo"; "violet"]. ~~full ROYGBIV list`

The + operator is distinguished from the other arithmetic operators because its operands do not have to be of number type, but can also be of `words` or `list` type. If the operands are of type `words`, the result of the + operator is the concatenated result of the two `words`. If one operand is of type `words` and the other operand is a different data type, the non-`words` operand is cast to type `words`; then regular string concatenation takes place, and the final concatenated result is of type `words`. If the operands are of the `list` types, the result of the + operator evaluates to a new joined `list` with elements from the two `list` operands, in the same sequence.

6.4.3 Comparison Operators

Operator	**Description**	**Operator**	**Description**
<	is less than	`not <`	is not less than
>	is greater than	`not >`	is not greater than
<=	is less than or equal to	`not <=`	is not less than or equal to
>=	is greater than or equal to	`not >=`	is not greater than or equal to
=	tests equality	`not =`	tests inequality

The final result of a comparison expression is of type `tof`. The equality operator and the inequality operator have lower precedence than the other comparison operators. Thus, `apples<oranges = pears<bananas` equals `true` if both `apples<oranges` and `pears<bananas` share the same `tof` value. In other words, it is equivalent to `(apples<oranges) = (pears<bananas)`.

StoryBook allows for comparison between all primitive data types listed in section 4.1. These operators compare the values of the elements being compared. A comparison of a `number` and a `letter` compares the ascii value of the `letter` to the `number`. All other comparisons must be made between values of the same type. A comparison between two `lists` compares the values at each index in the `list`. If a StoryBooker wishes to compare `Characters` he or she can create a comparison `Action` for that `Character`. For example:

```
Action compareRobot(Robot a; Robot b) {
  if a's model = b's model {
    endwith true;
  } else {
    endwith false;
  }
}
```

6.4.5 Logical Operators

Operator	Description	Example

`and`	logical and	`if (a and b) then {`
		`  say "Both true".`
		`}`

`or`	logical or	`if (a or b) then {`
		`  say "At least one is true".`
		`}`

The logical operators group left-to-right. The operands do not need to have the same type. When evaluating the operands, a value of `0` or `null` are both equivalent to `false`. Subsequently, discussion of an operand evaluating to `false` can also signify that the the operand is equal to `0` or `null.` Note that the result of a logical expression is always of type `tof.`

`and` returns `true` if both its operands are unequal to `false`, otherwise it returns `false`. It guarantees left-to-right evaluation and adopts short-circuit evaluation. The first operand is evaluated; if it is equal to `false`, the value of the entire expression is immediately set to `false`. Otherwise, the right operand is evaluated, and if it equal to `false`, the whole expression is `false`, otherwise `true`.

`or` returns `true` if either of its operands are not equal to `false`, otherwise it returns `false`. It also guarantees left-to-right evaluation and adopts short-circuit evaluation. The first operand is evaluated; if it is equal to `true`, the value of the entire expression is immediately set to `true`. Otherwise, the right operand is evaluated, and if it equal to `true`, the whole expression is `true,` otherwise the expression is equal to `false`.

6.4.6 Assignment Operator

The `is` assignment operator requires a mutable variable as the left operand. It can be of any of the primitives types or a `Character` type, but must not be of type `Chapter`. The left operand must be an initialized identifier. The type of an expression is that of its left operand, and the value is the value stored in the left operand after the assignment has taken place. The operand on the right must have the same type as the left operand.

```
number x is 5. ~~x is set to 5
x is (5+1). ~~x changes to 6
list1[3] is 23. ~~changes the item at index 3 to 23.
```

6.4.7 Sequence Operator

Sequence → [*expression; expression;*]+
Expressions separated by semicolons are evaluated left-to-right.

6.5 Control Flow

6.5.1 Conditional Expression

conditional-expr → `if` *tof-expr1* `then` *expr2* [`elseif` *tof-expr3* `then` *expr4*]* `else` *expr4*
Each expression after an `if` or after an `elseif` is an expression that evaluates to `true` or `false`. If the expression evaluates to `true` then the expression following the subsequent `then` is executed. Otherwise, the expression following the subsequent `else` is executed. In the case of multiple `if` statements preceding an `else` clause, then the `else` binds to the immediately preceding `if` block. Parentheses around the *tof-expr* condition are optional.

6.5.2 Loop Expressions

while-loop → `repeat while` *tof-expression*
This is the syntax of a StoryBook `repeat while` loop. The block of code defined in the loop will be executed while the *tof-expression* evaluates to `true`. The parentheses around the *tof-expression* is optional.

```
repeat while (SleepingBeauty's age not = 100) {
     SleepingBeauty, snore.
     SleepingBeauty's age is SleepingBeauty's age + 1.
}
```

for-loop → `repeat for` *expression1*`;` *expression2*`;` *expression3*
This is the syntax of a StoryBook *repeat for* loop. It is equivalent to:

expression1.

```
repeat while expression2 {
   statement.
   expression3.
}
```

The first expression is evaluated only once and initializes the loop. There is no restriction on its type. The second expression must evaluate to type `tof` and is typically a condition for the loop continue. Once the second expression evaluates to false, the loop ends. The third expression is evaluated after each iteration and specifies a re-initialization for the loop. There is no restriction on its type. The block of code defined after the `repeat for` line will be executed as long as the *tof-expression* evaluates to `true`.

The popular pattern used in a *for-loop* is *assignment-expression; tof-expression; arithmetic-expression*. In the initial declaration, an *identifier* is assigned to an initial `number` value. This is followed by the *tof-expression* to be tested on each run through the loop. Last is whether to increment or decrement the `number` *identifier* on each loop. A typical example:

```
repeat for number x is 5; x < 10; x = x + 1 {
     ~~this will print "hello" 5 times
     say("hello").
}
```

6.5.3 Return Statements

The keyword `endwith` indicates a return statement. No code following a return statement will be execute. Only a chapter (function) or action (method) can have a return statement. The return statement must match the type declared in the chapter or action header. No return is neccessary or allowed for type `nothing` (void). If the return is of type `number,` then its value must be enclosed in parenthesis. Examples below.

```
endwith (5).
endwith true.
```

7 Declarations and Types

7.1 Type Signatures

type-signature → *type identifier*

type → `number`
`letter`
`words`
`tof`
`words`
Character-identifier

When declaring a variable prepend each declaration with the data type.

```
number age.
letter initials.
words dialogue.
tof asleepOrNot.
```

7.2 Declarations

7.2.1 List Declarations

list-signature → *type* `list` *identifier*

`Lists` are treated as objects in the compiler but can be declared as a regular data type by the user. `Lists` can only contain one type of data type so that data type should prepends the list declaration. A `list` can be declared empty or with values.

```
number list dwarfAges is []. ~declares an empty list of numbers
                              called dwarfAges~
number list dwarfAges.       ~~equivalent to above expression
```

7.1.2 Character Declaration and Instantiation

Character-signature → `Character` *identifier*

`Character` variable names are capitalized. Inside the braces of a `Character` declaration the user can declare zero or more `traits` and `Actions`. To create an instance of a `Character` the `Character` *identifier* is prepended to the instance *identifier* and assigned to a `new` `Character` of that type. `Traits` are defined during instantiation by passing the values in as arguments.

```
Character Monster {
     words trait name.
     number trait size.

     Action scare(words scream)returns null {
          say scream.
     }
}

Monster Frank is new Monster(name is "Frankenstein"; size is
99).
```

```
say(Frank's name). ~~prints Frankenstein
Frank, scare("AHHHHHH"). ~~prints AHHHHHH
```

7.1.3 Character Subtype Declaration

Subtypes are declared with the same syntax as a normal Character with the addition of the subtype.

```
Character Giant is Monster { ~~can also call scare on Giant
     Action Capture(words list names) returns words list {
          repeat for number i is 1; i < names's length; i is i+1 {
               if names[i] = "Jack" {
                   names[i] = null.
               }
          }
          endwith names.
     }
}
```

7.1.4 Chapter Declarations

Chapter-signature → `Chapter` *identifier* ([*arg;*]*) `returns` *type*

`Chapters` are declared with zero or more parameters, separated by semicolons, and a return value preceded by the keyword `returns`.

`Chapter sum (number x; words y) returns number {` Chapter body `}`
`Chapter sum (number x; words y) returns nothing {` Chapter body `}`

7.1.5 Action Declarations

Action-signature → `Action` identifier([arg;]*) `returns` *type*

`Actions` are declared with zero or more parameters, separated by semicolons, and a return value preceded by the keyword `returns`.

`Action makeMoney(number initialAmnt, number salaryPerMonth,`
`number monthsWorked)returns number {` Action body `}`
`number monthsWorked)returns nothing {` Action body `}`

7.1.6 Plot Declarations

plot-signature → `Plot`([`word list args`]?)

The main function of a StoryBook program is called the `Plot`. The `Plot` can either have no arguments or it can take a `list` of command line arguments. When a StoryBooker runs a Storybook program, the first function that is called is the `Plot`.

```
~~Plot with command line args
Plot(words list args){ ~Plot body~ }

~~Plot without command line args
Plot(){ ~Plot body~ }
```

8 Library Functions

Below are the library functions defined for all StoryBookers to use.

8.1 Say

Prints `words` or `numbers` to standard output.

```
words pirateName = "Captain Jack Sparrow".
say("Ahoy " + pirateName). ~~prints "Ahoy Captain Jack Sparrow"
```

8.2 List Functions

These are functions that can only be invoked on `lists`.

8.2.1 Append

Used to add values to the end of a list.

```
number list ages is [20; 23; 26; 61].
ages, append(63).  ~~ages is now [20; 23; 26; 61; 63].
```

8.2.2 Insert

Used to add values at specified positions in a list, shifting the elements in the list if necessary. The first argument of the function is the value to add to the list. The second argument is the position at which to add the value.

```
words list names is ["Woody"; "Buzz"; "Nemo"].
names, insert("Dory"; 3). ~~names is now ["Woody"; "Buzz";
"Dory"; "Nemo"].
```

8.2.3 Remove

Used to remove values at specified positions in a list, shifting the elements in the list if necessary. The function argument indicates the position at which to remove the value.

```
words list guestList is ["Cinderella"; "Jasmine"; "Belle";
"mouse"; "Pocohontas"; "Elsa"; "Mulan"].
guestList, remove(4).
```

8.2.4 Length

Used to determine the length of a list.

```
letter list alphabet is ['a'; 'b'; 'c'; 'd'; 'e'; 'f'].
alphabet, length. ~~this will return 6
```

8.3 Character Functions

Special function that can only be invoked by a `Character`.

8.3.1 WhoAmI

Calling whoAmI on a `Character` returns a string description of that `Character`. This description will include the identifier of the `Character` of which it is an instance and all of the `Character`'s `traits`.

```
Character Soldier {
     word trait name.
     number trait age.
}
Soldier Jack is new Soldier(name is "Jack"; age is 18).
Jack, whoAmI. ~~This will return the string "Jack: I am a
Soldier. My name is Jack. My age is 18."
```