Language Reference Manual

This section provides a formal specification of the *Words* programming language. It is modeled heavily after the reference manual for the C programming language found in Appendix A of Kernighan and Ritchie's *The C Programming Language* (second edition).

Portions of the *Words* grammar are presented in the relevant subsections to help the reader understand the language. A complete grammar is presented in the final section.

Words is designed so that a program's execution should be displayed on a graphical user interface (GUI), allowing the user to see how the objects they have created are behaving. However, the GUI is not explicitly required by the language itself. Consequently, this manual focuses on the formal properties and requirements of the language and only mentions the GUI when it helps explain the language.

1. Lexical Conventions

1.1. Programs

A program consists of a list of statements stored in a single file or entered through a REPL interface. Programs are translated one statement at a time.

program:
    statement-list

statement-list:
    statement
    statement statement-list

The character set for *Words* is 7-bit ASCII. As such, the program file can use any character encoding that is downward-compatible with 7-bit ASCII, including UTF-8 and ISO 8859-1. (Likewise, UTF-16 and UTF-32 encoded program files are not supported because they are not downward-compatible to 7-bit ASCII.)

1.2. Tokens

There are six types of lexical tokens in *Words*: identifiers, references, keywords, literals, operators, and separators. Adjacent identifiers, references, keywords, and literals should be separated from one another by one or more whitespace characters when necessary. Whitespace characters are the space, tab, newline, and carriage return characters. Aside from separating adjacent tokens, whitespace characters are ignored.

1.3. Comments

The characters `//` designate a comment except if they occur within a string literal. All characters from the `//` to the end of the line are treated as whitespace and ignored.

1.4. Identifiers

An identifier is a sequence of letters, digits, and/or underscore characters. The first character in an identifier must be a letter or underscore. There is no length limit on identifiers. Identifiers are case sensitive: uppercase and lowercase versions of a letter are considered different. Identifiers cannot be a keyword.

1.5. References

A reference is a sequence of letters, digits, and/or underscore characters immediately followed (without any intervening whitespace) by the characters `'s`. References follow the same rules as identifiers with respect to the first character, length, case sensitivity, and the portion of the reference prior to the `'s` not matching a keyword.

1.6. Keywords

The following is a complete list of keywords. For clarity, keywords are always written in **`bold fixed-width`** font. Keywords are case sensitive, though in some cases the first character may be either upper-case or lower-case as shown in the table.

`a`	**`has`**	**`moves`**	**`that`**
`A`	**`her`**	**`means`**	**`their`**
`above`	**`Her`**	**`next`**	**`Their`**
`an`	**`him`**	**`not`**	**`them`**
`An`	**`his`**	**`nothing`**	**`then`**
`and`	**`His`**	**`now`**	**`times`**
`anywhere`	**`If`**	**`of`**	**`to`**
`as`	**`is`**	**`or`**	**`touches`**
`As`	**`it`**	**`Remove`**	**`up`**
`at`	**`its`**	**`Repeat`**	**`wait`**
`be`	**`Its`**	**`Reset`**	**`waits`**
`below`	**`left`**	**`right`**	**`Whenever`**
`can`	**`long`**	**`say`**	**`which`**
`down`	**`Make`**	**`says`**	**`While`**
`Exit`	**`move`**	**`Stop`**	**`with`**

1.7. Literals

There are two types of literals in *Words*: number literals and string literals.

1.7.1. Number Literals

A number literal is a sequence of digits, optionally followed by a decimal point and another sequence of digits. A number literal cannot begin or end with a decimal point, i.e., `.123` and `123.` are not valid number literals. All number literals are considered to be floating point decimal numbers.

1.7.2. String Literals

A string literal is a sequence of characters, excluding newline characters and double-quotes, enclosed within double-quotes, i.e., `"string"`. Within a string literal, a newline character may be represented with the escape sequence `\n`, a double-quote character may be represented with the escape sequence `\"`, and a backslash may be represented with the escape sequence `\\`.

1.8. Operators and Separators

The following characters or sequences of characters are operators in *Words*: `+`, `-`, `*`, `/`, `=`, `<`, `<=`, `>`, and `>=`, as well as the keywords `and`, `or`, and `not`.

The following characters are separators in *Words*: `.`, `(`, `)`, `{`, `}`, `[`, `]` and `,`.

2. Semantic Concepts

2.1. Semantic Items

The fundamental semantic concepts in *Words* are:

- **Classes:** A class specifies a type for objects. The class definition provides the common properties of the objects in the class and the custom actions that objects in the class can perform. All classes (except the base class) must have a single parent class, and they inherit the common properties and custom actions of their parent class. The special built-in base class *thing* exists to be an available parent class to new classes. The sequence of classes from a given class (or an object of a given class) through its parent, grandparent, etc. up to the base class *thing* is called its class chain.
- **Objects:** An object is a bundled, durable, and mutable collection of properties belonging to a single class. An object's class is immutable and set when the object is created. The properties of an object are identifier-value pairs, where the value may be a string, a number, or a pointer to another object. All objects inherit the properties of the classes in their class chain and can both override the properties of or contain additional properties beyond those provided by their class chain. All objects include special number properties *row* and *column* representing the object's position in the GUI. These special properties may be positive or negative but cannot be assigned a non-numeric value. All objects also include special string properties *name* and *class* representing the name and class given to the object when it was created. These properties are immutable, and attempting to change them generates a runtime error.

All objects include an initially empty action queue of actions that they will perform over time (see the Action Queues section below).

- **Basic actions:** The basic actions are property assignment, move, say, and wait. All objects can perform any of the basic actions. A basic action can be added to an object's action queue.
- **Custom actions:** Custom actions are sequences of statements to be performed when the custom action is invoked. Custom actions are defined within a class definition and are denoted with an identifier. An object can invoke a custom action only if it appears in its class chain. A custom action can be added to an object's action queue.
- **Strings:** A string is a sequence of characters.
- **Numbers:** All numbers are floating point.

We use the somewhat generic term 'items' to refer to any of these semantic concepts.

When a property of an object is sought using an identifier, *Words* first searches the identifier-value pairs constituting the properties of the object for a matching identifier. The search continues through the properties defined in the object's class chain and terminates as soon as a property matching the identifier is found. Likewise, when a custom action is invoked by an object, *Words* first searches the custom actions of the object's class, then up through the custom actions defined in its class chain.

2.2. Identifiers and References

An identifier, possibly immediately preceded by one or more references (a reference list), refers to a location in storage. This location contains one of five types of items: an object, a class, a custom action, a string, or a number. The location may also be empty/unused, in which case the identifier is taken to refer to the special keyword **`nothing`**.

An *unreferenced identifier* (one that is not immediately preceded by a reference list) refers to either an object, a class, a custom action, or **`nothing`**, except within the body of a custom action definition, in which case it may also refer to a string or number argument to the custom action. A *referenced identifier* (one that is immediately preceded by a reference list) refers to either an object, a number, a string, or **`nothing`**.

For example, if the following appears not in the body of a custom action definition:

> `John` may refer to either an object, a class, a custom action, or **`nothing`**.
> `John's mother` may refer to either an object, a number, a string, or **`nothing`**.

In the first example, `John` is an unreferenced identifier. In the second example, `mother` is a referenced identifier since it is immediately preceded by the reference list `John's`.

Note that both referenced identifiers and unreferenced identifiers may refer to objects. When a language construct expects an object, the grammar generates a (possibly empty) reference list followed by an identifier:

queueing-statement:
    `STOP` *reference-list identifier* **.**

reference-list:
    reference reference-list
    ε

A reference refers to the item named by the portion of the reference excluding the final characters `'s`. For example, the reference `John's` refers to the same item as the identifier `John`. The item a reference refers to must be an object. If it does not refer to an object, or refers to **`nothing`**, a runtime error is generated. References may be chained together to form a reference list. For example, `John's mother's` is a reference list.

To determine the item that a referenced identifier refers to, *Words* follows the chain of objects embodied in the reference list. For example, `John's mother's height` refers to a (presumably but not necessarily number) property on an object. The object in question is referred to by `John's mother`. `mother` in turn refers to a property (which must be an object, since it was part of a reference) on an object referred to by the identifier `John`. If the object `John` did not exist, or if it did not have a property `mother`, or if its property `mother` was not an object, a runtime error would have been generated. If merely the property height did not exist, the result would have been **`nothing`** but not an error.

The type of item that an identifier refers to is determined dynamically at runtime, so the grammar of *Words* does not enforce that referenced identifiers and unreferenced identifiers refer to an acceptable type of item. As described above, incorrect usage generates a runtime error.

3. Expressions and Predicates

3.1. Expressions

An expression is a sequence of operators and other lexical tokens that can be evaluated. An evaluated expression produces either an object, a number, a string, **`nothing`**, true, or false. Note that true and false are not keywords or a type of object property but merely an intermediate or final result of expression evaluation.

There are two types of expressions: value expressions and relational expressions, discussed below.

3.1.1. Operator Precedence and Associativity

The following table presents the operators available in expressions, their meaning, acceptable operands, their associativity, and their precedence. The table is presented in decreasing order of precedence with the highest precedence operators first. Rows are grouped, and all operators in the same group have the same precedence. Blank rows separate groups.

Operator	Type	Meaning	Operands	Associativity
`-`	unary	Negation	Number	right-to-left
`^`	binary	Exponentiation	Number	right-to-left
`*`	binary	Multiplication	Number	left-to-right
`/`	binary	Division	Number	left-to-right
`+`	binary	Addition, string concatenation	Number, string	left-to-right
`-`	binary	Subtraction	Number	left-to-right
`=`	binary	Equality	Number, string, object, **`nothing`**	N/A
`<`	binary	Less than (numeric or alphabetical)	Number, string	N/A
`>`	binary	Greater than (numeric or alphabetical)	Number, string	N/A
`<=`	binary	Less than or equal (numeric or alphabetical)	Number, string	N/A
`>=`	binary	Greater than or equal (numeric or alphabetical)	Number, string	N/A
`not`	unary	Logical negation	True, false	N/A
`and`	binary	Logical conjunction	True, false	left-to-right
`or`	binary	Logical disjunction	True, false	left-to-right

Attempting to apply an operator to an invalid operand (after type conversions are attempted, as discussed below) generates in a runtime error.

3.1.2. Value Expressions

A value expression is an expression whose evaluation results in an object, a number, a string, or **`nothing`**. Value expressions follow the usual infix arithmetic notation and permit using + for string concatenation.

value-expression:
    reference-list identifier
    literal
    `NOTHING`
    (*value-expression*)
    -value-expression
    value-expression + *value-expression*
    value-expression - *value-expression*
    value-expression * *value-expression*
    value-expression / *value-expression*
    value-expression ^ *value-expression*

3.1.3. Relational Expressions

A relational expression is an expression that compares value expressions using a relational operator and results in either true or false.

relational-expression:
    value-expression = *value-expression*
    value-expression < *value-expression*
    value-expression > *value-expression*
    value-expression <= *value-expression*
    value-expression >= *value-expression*

3.1.4. Type Coercions

For certain operators, number operands may be coerced to strings, or vice versa, based on the following table.

Operator	Left Side (LS) Type	Right Side (RS) Type	Coercion(s) Attempted
- (unary)	N/A	String	RS to number, else error
^	String	String	LS and RS to number, else error
*	String	String	LS and RS to number, else error

/	String	String	LS and RS to number, else error
+	String	Number	LS to number, else RS to string
+	Number	String	RS to number, else LS to string
- (binary)	String	String	LS and RS to number, else error
=	String	Number	LS to number, else RS to string
=	Number	String	RS to number, else LS to string
<	Number	String	Error
<	String	Number	Error
>	Number	String	Error
>	String	Number	Error
<=	Number	String	Error
<=	String	Number	Error
>=	Number	String	Error
>=	String	Number	Error

Thus, a number can be converted to a string with the expression `n + ""`, and a string can be converted to a number with the expression `s + 0`.

In all cases where a string is accepted, an object can be coerced into a string equal to the object's special *name* property.

3.2. Predicates

A predicate is a condition that can be true or false. There are two types of predicates: boolean predicates and basic action predicates.

3.2.1. Boolean Predicates

A boolean predicate is a combination of relational expressions or other boolean predicates using logical operators.

boolean-predicate:
    relational-expression
    (*boolean-predicate*)
    `NOT` (*boolean-predicate*)

boolean-predicate `AND` *boolean-predicate*
boolean-predicate `OR` *boolean-predicate*

Boolean predicates are evaluated left-to-right and employ short-circuit evaluation, where evaluation ceases as soon as the final result is definitively known.

3.2.2. Basic Action Predicates

A basic action predicate tests whether certain behavior has occurred in the current frame (see the Frames section below).

basic-action-predicate:
subject alias `MOVES` *direction*$_{opt}$
subject alias `SAYS` *value-expression*
subject alias `WAITS`
subject alias `TOUCHES` *subject alias*
subject alias adjacency-expression subject alias

subject:
reference-list identifier
`A` identifier

alias:
[*identifier*]
ε

adjacency-expression:
`IS NEXT TO`
`IS BELOW`
`IS ABOVE`
`IS LEFT OF`
`IS RIGHT OF`

In the first three forms (moves, says, waits), the predicate indicates whether an object has performed one of those basic actions in the current frame. In the case of a wait predicate, it will also be true if the object's action queue was empty in the current frame, which is semantically equivalent to a wait action. In the fourth form (touches), the predicate indicates whether two objects have come to the same position in the current frame. In the final form, the predicate indicates whether two objects have come to be directly adjacent to one another in a specific or any direction in the current frame.

There is no basic action predicate for the property assignment basic action.

A basic action predicate can use one of two types of “subjects”: a named object using a reference list and an identifier or the name of a class. Consider the following:

```
X is a thing.

Whenever X moves {
      Stop X.
}
```

In this example, the predicate and listener body refers to the object X directly. Now consider the following:

```
X is a thing.

Whenever a thing [Y] moves {
      Stop Y.
}
```

In this example, the predicate will be evaluated for every member of the class *thing*, and for each one that is true, the body of the listener will execute using the alias Y to refer to the object that satisfied the predicate (see the Scope section below).

4. Statements

A statement is the unit of translation in *Words*. Statements are translated and executed one at a time. They are terminated by the **.** separator. There are two main types of statements: *declarative statements* and *non-declarative statements*. Declarative statements declare either a class or an event listener and can only appear in the main body of a program, not within an enclosing block. Non-declarative statements are any other statement and can appear anywhere within a program. They have two sub-types: *immediate statements* and *queueing statements*. Immediate statements have an immediate impact on the execution of the program. Queueing statements add to or otherwise modify an object’s action queue (see the Action Queues section below).[1]

statement:
    declarative-statement
    non-declarative-statement

declarative-statement:
    class-declare-statement

[1] Of course, the addition or modification to an object’s action queue does take place immediately (when the statement is translated and executed), but the main semantic effect of the statement and portions of its evaluation do not occur until the action reaches the front of the action queue, hence the distinction from immediate statements.

listener-declare-statement

non-declarative-statement:
immediate-statement
queuing-statement

4.1. Declarative Statements

4.1.1. Class Declaration Statements

A class declaration statement creates a new class by specifying the name of the class, the parent class, and optionally a set of property and custom action definitions via one or more class definition statements.

class-declare-statement:
`A` *identifier* `IS A` *identifier*`.`
`A` *identifier* `IS A` *identifier* `WHICH {` *class-definition-statement-list* `}`

class-definition-statement-list:
class-definition-statement
class-definition-statement class-definition-statement-list

The identifier used to name a class must not match any existing class at the time the statement is executed, else a runtime error is generated.

4.1.1.1. Class Definition Statements

The details of a class are specified by a series of class definition statements. These statements define properties of objects in the class and custom actions that can be invoked by objects in the class.

class-definition-statement:
`HAS A` *identifier* `OF` *literal*`.`
`CAN` *identifier* `WHICH MEANS {` *non-declarative-statement-list* `}`
`CAN` *identifier* `WITH` *parameter-list* `WHICH MEANS {` *non-declarative-statement-list* `}`

parameter-list:
parameter
parameter`,` *parameter-list*

parameter:
identifier

Here, *parameter-list* provides the names of the (optional) parameters to the custom action, and *statement-list* is the body of the custom action that is executed when the custom action is

invoked. Duplicate parameter names in *parameter-list* are simply ignored. Only non-declarative statements are permissible in the definition of a custom action. Within the body of a custom action, the keywords **`him`**, **`her`**, **`it`**, **`them`**, **`his`**, **`her`**, **`its`**, **`their`**, **`His`**, **`Her`**, **`Its`**, and **`Their`** used in place of an identifier or reference, refer to the object that invoked the custom action.

4.1.2. Listener Declaration Statements

A listener declaration statement creates an event listener based on a predicate. The predicate is tested on each frame (see the Frames section below), and if the predicate is true, the given statement list is executed in that frame.

listener-declare-statement:
`WHENEVER` *predicate* { *non-declarative-statement-list* }
`AS LONG AS` *predicate* { *non-declarative-statement-list* }

In the first form, the event listener that is created is permanent for the duration of the program's execution.

In the second form, the statement list will execute each frame (including the frame in which the listener was created) while the predicate is true. Once the predicate is false for the first time (again, including the frame in which the listener was created), the listener is destroyed.

4.2. Immediate Statements

There are six main kinds of immediate statements: object creation, object destruction, object property assignment, iteration, conditionals, and runtime control.

immediate-statement:
object-create-statement
object-destroy-statement
property-assign-statement
iteration-statement
conditional-statement
runtime-control-statement

4.2.1. Object Creation Statements

An object creation statement creates a new object of a given class at a given position.

object-create-statement:
identifier `IS A` *identifier* `AT` *position*`.`

position:
value-expression`,` *value-expression*

Properties may be later added to the object using property assignment statements or queued property assignment actions.

The two value expressions for position respectively specify the column and row of the object (i.e., x and y) in the GUI, respectively, which may be zero or negative. They must each evaluate to a number or a value that can be coerced to a number, else a runtime error is generated. The actual column and row of the object are the values of the value expressions each rounded to the nearest integer.

The identifier used to name an object must not match any existing object in scope at the time the statement is executed, else a runtime error is generated.

The identifier used to name an object is assigned to the object's special *name* property and cannot subsequently be changed.

4.2.2. Object Destruction Statements

An object destruction statement completely removes an object. The identifier previously used to name the object can subsequently be reused.

object-destroy-statement:
    `REMOVE` *reference-list identifier* **.**

Note that the original identifier used to create an object need not be the identifier provided when destroying an object. Consider the following:

```
X is a thing.
Y is a thing.
X's Z is Y.
Remove X's Z.
```

This program has the effect of creating and shortly afterward destroying the object Y. The identifier Y is not used in the object destruction statement, but it is the result of the evaluation of the value expression `X's Z`.

4.2.3. Property Assignment Statements

A property assignment statement creates a new property on an object or modifies an existing property.

property-assign-statement:
    reference reference-list identifier `IS` *value-expression* **.**

Here, at least one reference is required in order to identify the object whose properties are to be modified. Consider the following (assuming X and Y are previously created objects):

```
X's P is Y.            // Valid
Z is B.                // Invalid
```

The identifier is the name of the property, and the value-expression is evaluated to the value of the property.

For the avoidance of doubt, note that the following assignment is valid since **nothing** is a value expression:

```
X's P is nothing.      // Valid
```

This form has the effect of removing property P from object X, if it previously existed. No error is generated if property P did not previously exist.

4.2.4. Iteration Statements

An iteration statement allows for a set of statements to be executed more than once.

iteration-statement:
    REPEAT *value-expression* TIMES { *non-declarative-statement-list* }
    WHILE *boolean-predicate* { *non-declarative-statement-list* }

In the first form, the statement list is executed some number of times based on the evaluation of the value expression. The value expression must evaluate to a number or a value that can be coerced to a number, else a runtime error is generated. The number of times the statement list is executed is the value of the value expression rounded to the nearest integer. If the value expression evaluates to zero or a negative number, the statement list is not executed at all.

In the second form, the statement list is executed as long as the boolean predicate is true. Naturally, the statement list should make some change (e.g., modifying object properties) so that the boolean predicate will eventually be false. If the boolean predicate is already false, statement list is not executed at all.

A common error is to use an object property in the boolean predicate but to modify that property in the statement list using a queueing statement, such as the following (assuming X is a previously created object):

```
X's P is 1.
While X's P < 5 {
    Make X's P be X's P + 1.
}
```

This will result in an infinite loop, because the change to X's property P will not take place immediately but only when that action is executed.

4.2.5. Conditional Statements

A condition statement executes a given statement list if a given boolean predicate is true.

conditional-statement:
    `IF` *boolean-predicate* `THEN {` *non-declarative-statement-list* `}`

4.2.6. Runtime Control Statements

A runtime control statement allows the user to restart or terminate a *Words* program. They are most useful when *Words* is being used in a REPL interface, but they can be used in a program stored in a file as well.

runtime-control-statement:
    `RESET.`
    `EXIT.`

The reset statement causes the environment to completely restart as if the *Words* program were just started. All existing classes, objects, and listeners are removed.

The exit statement causes the *Words* program to exit.

4.3. Queueing Statements

Queueing statements add to or modify an object's action queue (see the Action Queues section below). The five types of actions that can be added to an objects action queue — each of the four basic actions (property assignment, move, say, and wait) and custom actions — correspond to the queueing statements below. In addition, a sixth queueing statement using the `Stop` keyword completely clears an object's action queue.

queueing-statement:
    `MAKE` *reference-list queue-assign-property-list* $\texttt{NOW}_{opt}$`.`
    `MAKE` *reference-list identifier* `MOVE` *direction value-expression*$_{opt}$ $\texttt{NOW}_{opt}$`.`
    `MAKE` *reference-list identifier* `SAY` *value-expression* $\texttt{NOW}_{opt}$`.`
    `MAKE` *reference-list identifier* `WAIT` *value-expression* $\texttt{NOW}_{opt}$`.`
    `STOP` *reference-list identifier*`.`
    queueing-custom-action-statement

queueing-custom-action-statement:
    `MAKE` *reference-list identifier identifier* $\texttt{NOW}_{opt}$`.`
    `MAKE` *reference-list identifier identifier* `WITH` *argument-list* $\texttt{NOW}_{opt}$`.`

argument-list:
    argument

argument, *argument-list*

argument:
identifier value-expression

direction:
ANYWHERE
DOWN
LEFT
RIGHT
UP

Without the now keyword, the action is enqueued at the end of the object's action queue. If the now keyword is used, the action is enqueued at the front of the object's action queue and will be the next action processed in the next frame (see the Frames section below). Thus, to add several actions to the front of an object's action queue in a desired order using the now keyword, one must specify them in reverse order. Consider the following (assuming X is a previously created object):

```
Make X move left 3 now.
Make X say "example" now.
Make X's P be 5 now.
```

The first three actions on X's action queue are, in order, the property assignment, the say action, and the move action, because each enqueued action became the new front, ahead of the previously enqueued action.

4.3.1. Move Actions

The optional value expression to a queueing statement for a move action must evaluate to either a number or a value that can be coerced to a number, else a runtime error is generated. If omitted, the value is assumed to be 1. A negative number is treated as a positive number with the opposite direction. If the anywhere keyword is used, a random direction is chosen and the subsequent semantics are as if that direction were provided explicitly. The statement causes a move action to be enqueued on the object's action queue, with the number of units to move rounded to the nearest integer. A zero is treated and enqueued as a wait action for 1 frame.

4.3.2. Say Actions

The value expression to a queueing statement for a say action must evaluate to a string or a value that can be coerced to a string, else a runtime error is generated. The statement causes a say action to be enqueued on the object's action queue.

4.3.3. Wait Actions

The value expression to a queueing statement for a wait action must evaluate to either a number or a value that can be coerced to a number, else a runtime error is generated. The statement causes a wait action to be enqueued on the object's action queue, with the number of frames to wait rounded to the nearest integer. A negative number or zero generates a runtime error.

4.3.4. Custom Actions

When invoking a custom action, the argument list consists of identifier-value pairs. The value expressions are evaluated and passed as the argument values for the corresponding parameters in the custom action definition. The identifier-value pairs may appear in any order, not necessarily the order that was used in the custom action definition. Any identifier-value pairs where the identifier does not match the parameter name in the custom action definition are ignored.

5. Scope

Scope in Words is slightly different for classes, objects, listeners aliases, and parameters, but follow natural rules.

5.1. Classes

The names of classes have global scope in Words below the point in the program where they are declared. Since class declarations are declarative statements, they can only appear within the main body of the program and not within a block.

5.2. Objects

When an object is created, it continues to exist (and is displayed in the GUI) regardless of where in the program it was created. However, the identifier used to name the object has scope only within the block enclosing the object creation statement.

For iteration statements, the scope resets on each execution of the body. For example, consider the following:

```
Repeat 5 times {
      X is a thing at 0,0.
      Repeat 10 times {
            Make X move anywhere.
      }
}
```

In this example, 5 objects named X are created, and each is queued to move 10 times in a random direction each time. The use of the identifier X outside of the iteration would be a runtime error. (If some other object of that name existed, the object creation statement in the iteration body would itself have been a runtime error.)

5.3. Listener Aliases

Aliases creates for an object in a basic action predicate have scope only within the body of the listener.

5.4. Parameters

Parameters to custom actions have local scope within the body of the custom action.

6. Action Queues

Each object has an action queue. An action queue is a series of basic or custom actions that the object will perform in order over time (see the Frames section below).

An action on the queue may be either expandable or executable based on the following table:

Expandable	Executable
● Movement actions for >1 unit or based on a value expression other than a number literal ● Wait actions for >1 unit or based on a value express other than a number literal ● Any custom action	● Any property assignment action ● Movement actions for 1 unit using a number literal ● Any say action ● Wait actions for 1 frame using a number literal

6.1. Expanding Expandable Actions

When the first action on an action queue is an expandable action (see the Frames section below), it is expanded. The semantics of expansion depend on the type of action.

6.1.1. Movement Actions

The action is replaced with a number of 1 unit movement actions equal to the evaluation of the value expression. The actions are placed at the front of the action queue.

6.1.2. Wait Actions

The action is replaced with a number of 1 frame wait actions equal to the evaluation of the value expression. The actions are placed at the front of the action queue.

6.1.3. Custom Actions

The arguments (if any) to the custom action are evaluated and the statements in the custom action definition are executed. Non-queueing statements are executed immediately as normal. However, queueing statements cause their actions to be enqueued at the *front* of the object's action queue in *the order in which they appeared* in the body. In this way, the single custom action on the queue is "expanded" into its constituent actions (which, of course, can be basic or custom actions that themselves may be expandable.)

7. Frames

A frame is a length of time during the program's execution. Certain activities are performed at the end of each frame. Typically, every frame is the same fixed length of time, but it is possible for the GUI to use variable-length frames or allow the user to adjust the frame rate or pause the current frame during program execution.

Each frame consists of three phases in the following order: statement translation and execution, action queue processing, and listener evaluation. After these phases are completed, the GUI can be updated to display objects at their current positions, what the objects are saying, etc.

7.1. Phase 1: Statement Translation and Execution

In this phase, any statements received since the previous frame are translated and executed. Thus, if the program is provided as a file, all statements in the file are translated and executed (even before the first frame). If statements are provided by the user in a REPL interface, all complete statements they entered since the previous frame are translated and executed.

7.2. Phase 2: Action Queue Processing

In this phase, the first action in each object's action queue is processed. If it is an expandable action, it is expanded (see the Action Queues section above), and expansion process repeats until the first action is an executable action. The first action (now an executable action) is then dequeued and executed. If the object's action queue is empty, there is no effect. An object's action queue is not processed until the first frame *after* the frame in which it was created. In this way, an object will always first appear at the position specified when it was created.

Naturally, executing a move action updates the object's row and column properties. Movement `left` and `right` decreases or increases the column respectively, and movement `up` and `down` increases or decreases the row respectively. Executing a say action causes some implementation-defined internal state to be updated such that the object will say the provided value expression in the GUI. Executing a wait action causes no change other than advancing the action queue.

When the first action is expanded or executed, any value expressions are evaluated only at that time, and within the scope in which the queueing statement originally appeared. The value expressions are *not* evaluated when the action was first enqueued. Consider the following (assuming X and Y are previously created objects):

```
Make X move left Y's P + 1.
Make X's P be Y's Q + 10.
```

In this example, we have queued a move action on X. When this action is the first, it must be expanded, and at that time, the value expression `Y's P + 1` is evaluated to determine how

many 1-unit move actions to expand it into. Some time later after the movement is complete, the queued assignment to X's property P will be the first action on X's action queue. The value that is assigned will be determined by evaluating the value expression `Y's Q + 10` at this time.

The order in which the expansion and execution of actions occur within a frame is arbitrary and implementation-defined. Consider, for example, the following (assuming X and Y are previously created objects):

```
X's P is 10.
Y's P is 20.
Make X move left 3.
Make Y move left 3.
Make X's P be Y's P.
Make Y's P be X's P.
```

In this example, we have queued property assignments to both objects X and Y. They will be dequeued and executed in the same frame. It is implementation-defined whether X's P becomes 10 and Y's P becomes 10, or X's P becomes 20 and Y's P becomes 10, as if the actions were first executed on X and then on Y, or vice versa. The implementation may also have X's P become 20 and Y's P become 10, as if the actions were executed simultaneously.

If the first action is a custom action that includes object destruction statements, the object destructions are not processed until after the entire phase is complete.

7.3. Phase 3: Listener Evaluation

In this phase, the predicates for event listeners are evaluated and, if they are true, the body of the event listener is executed. Boolean predicate evaluation reflects any changes to object property values made in the prior two phases. Basic action predicate evaluation reflects any actions that were executed in the action queue processing phase.

All listener predicate evaluations within a frame occur "simultaneously" with respect to object properties based on the property values in place after the prior two phases in the frame. If the listener body includes object destruction statements, the object destructions are not processed until after the entire phase is complete.

If the body of the event listener includes queueing statements, they will affect the object's action queue immediately (as normal) but any enqueued actions cannot possibly be dequeued and executed until the next frame. If two different listeners both have queueing statements for the same object, the order of evaluation is arbitrary and the resultant behavior is implementation-defined.

8. Grammar

The complete grammar of *Words* is presented below.

In this presentation, the terminal symbols consist of any text or characters written in `fixed-width` font as well as the italicized symbols *identifier*, *reference*, *number*, and *string*, which correspond to identifiers, references, number literals, and string literals respectively. All other italicized symbols are non-terminals, the productions for which are given in the grammar. The start symbol is *program*.

The terminal symbols written in upper-case `fixed-width` font map one-to-one to the keywords, operators, and separators presented earlier (apart from capitalization), with the exception of the following:

Terminal Symbol	Corresponding Keywords
`A`	**`a`**, **`A`**, **`an`**, **`An`**
`WHICH`	**`that`**, **`which`**

The ε symbol indicates that the non-terminal at the head of the production may be the empty string.

The 'opt' subscript indicates that the symbol is optional in that production. Some parser generators might require these productions to be split into two separate productions, one with and one without the optional symbol.

For ease of presentation, the grammar shown below does not enforce the precedence and associativity of the operators listed earlier. When transforming the grammar into suitable input for a parser generator such as Yacc, the precedence and associativity of the operators can be supplied.

program:
    statement-list

statement-list:
    statement
    statement statement-list

statement:
    declarative-statement
    non-declarative-statement

non-declarative-statement-list:
    non-declarative-statement
    non-declarative-statement non-declarative-statement-list

declarative-statement:
    class-declare-statement
    listener-declare-statement

class-declare-statement:
    `A` *identifier* `IS A` *identifier*`.`
    `A` *identifier* `IS A` *identifier* `WHICH {` *class-definition-statement-list* `}`

class-definition-statement-list:
    class-definition-statement
    class-definition-statement class-definition-statement-list

class-definition-statement:
    `HAS A` *identifier* `OF` *literal*`.`
    `CAN` *identifier* `WHICH MEANS {` *non-declarative-statement-list* `}`
    `CAN` *identifier* `WITH` *parameter-list* `WHICH MEANS {` *non-declarative-statement-list* `}`

listener-declare-statement:
    `WHENEVER` *predicate* `{` *non-declarative-statement-list* `}`
    `AS LONG AS` *predicate* `{` *non-declarative-statement-list* `}`

non-declarative-statement:
    immediate-statement
    queuing-statement

immediate-statement:
    object-create-statement
    object-destroy-statement
    property-assign-statement
    iteration-statement
    conditional-statement
    runtime-control-statement

object-create-statement:
    identifier `IS A` *identifier* `AT` *position*`.`

object-destroy-statement:
    `REMOVE` *reference-list identifier*`.`

property-assign-statement:
    reference reference-list identifier `IS` *value-expression* `.`

iteration-statement:
    `REPEAT` *value-expression* `TIMES` { *non-declarative-statement-list* }
    `WHILE` *boolean-predicate* { *non-declarative-statement-list* }

conditional-statement:
    `IF` *boolean-predicate* `THEN` { *non-declarative-statement-list* }

runtime-control-statement:
    `RESET.`
    `EXIT.`

queueing-statement:
    `MAKE` *reference-list queue-assign-property-list* $\texttt{NOW}_{opt}$`.`
    `MAKE` *reference-list identifier* `MOVE` *direction value-expression*$_{opt}$ $\texttt{NOW}_{opt}$`.`
    `MAKE` *reference-list identifier* `SAY` *value-expression* $\texttt{NOW}_{opt}$`.`
    `MAKE` *reference-list identifier* `WAIT` *value-expression* $\texttt{NOW}_{opt}$`.`
    `STOP` *reference-list identifier* `.`
    queueing-custom-action-statement

queueing-custom-action-statement:
    `MAKE` *reference-list identifier identifier* $\texttt{NOW}_{opt}$`.`
    `MAKE` *reference-list identifier identifier* `WITH` *argument-list* $\texttt{NOW}_{opt}$`.`

predicate:
    basic-action-predicate
    boolean-predicate

basic-action-predicate:
    subject alias `MOVES` *direction*$_{opt}$
    subject alias `SAYS` *value-expression*
    subject alias `WAITS`
    subject alias `TOUCHES` *subject alias*
    subject alias adjacency-expression subject alias

boolean-predicate:
    relational-expression
    (*boolean-predicate*)
    `NOT` (*boolean-predicate*)
    boolean-predicate `AND` *boolean-predicate*

boolean-predicate `OR` *boolean-predicate*

relational-expression:
value-expression = *value-expression*
value-expression < *value-expression*
value-expression > *value-expression*
value-expression <= *value-expression*
value-expression >= *value-expression*

value-expression:
reference-list identifier
literal
`NOTHING`
(*value-expression*)
-value-expression
value-expression + *value-expression*
value-expression - *value-expression*
value-expression * *value-expression*
value-expression / *value-expression*
value-expression ^ *value-expression*

reference-list:
reference reference-list
ε

parameter-list:
parameter
parameter, *parameter-list*

argument-list:
argument
argument, *argument-list*

queue-assign-property-list:
queue-assign-property
queue-assign-property, *queue-assign-property*

argument:
identifier value-expression

parameter:
identifier

subject:
    reference-list identifier
    `A` identifier

alias:
    [*identifier*]
    ε

queue-assign-property:
    identifier `BE` *value-expression*

adjacency-expression:
    `IS NEXT TO`
    `IS BELOW`
    `IS ABOVE`
    `IS LEFT OF`
    `IS RIGHT OF`

direction:
    `ANYWHERE`
    `DOWN`
    `LEFT`
    `RIGHT`
    `UP`

position:
    value-expression`,` *value-expression*

literal:
    number
    string