Next
Final Report

Ernesto Arreguin (eja2124)
Danny Park (dsp2120)
Morgan Ulinski (mu2189)
Xiaowei Zhang (xz2242)

Contents

1 Introduction

2 Language Tutorial

3 Language Manual

3.1 Introduction

The Next programming language provides a way to easily create text-based computer games. It is particularly good for creating text-based RPGs (or digital "choose your own adventure stories"). Users of the language can specify locations, characters, and items that will appear in the game, and using Next language elements they can design the plot of the game, the actions that can take place, and beginning and end conditions.

3.2 Lexicon

The Next programming language uses a standard grammar and character set. Characters in the source code are grouped into tokens, which can be punctuators, operators, identifiers, keywords, or string literals. The compiler forms the longest possible token from a given string of characters; tokens end when white space is encountered, or when it would not be possible for the next character to be part of the token. White space is defined as space characters, tab characters, return characters, and newline characters.

The compiler processes the source code and identifies tokens and locates error conditions. There are three types of errors:

- Lexical errors occur when the compiler cannot form a legal token from the character stream.
- Syntax errors occur when a legal token can be formed, but the compiler cannot make a legal statement from the tokens.

- Semantic errors, which are grammatically correct and thus pass through the parser, but break another Next rule. For example, it is possible to `kill` a character or item, but not a location.

3.2.1 Character Set

The Next programming languages accepts standard ASCII characters.

3.2.2 Identifiers

An identifier is a sequence of characters that represents a name for a:

- Variable
- Location
- Character
- Item
- Action (only within a `choose` statement)

Rules for identifiers:

- Identifiers consist of a sequence of one or more uppercase or lowercase characters, the digits 0 to 9, and the underscore character (_).
- Identifier names are case sensitive.
- Identifiers cannot begin with a digit or an underscore.
- Keywords are not identifiers.

3.2.3 Comments

Comments are introduced by `/*` and ended by `*/`, except within a string literal, where the characters `"/*"` would be displayed directly. Comments cannot be nested. If a comment is started by `/*`, the next occurrence of */ ends the comment.

3.2.4 Keywords

Keywords identify statement constructs and specify basic types. Keywords cannot be used as identifiers. The keywords are listed in Table 1.

Table 1: Keywords

`if`	`then`	`else`	`and`	`or`
`start`	`end`	`when`	`choose`	`kill`
`grab`	`hide`	`exists`	`drop`	`output`
`character`	`location`	`not`	`item`	`int`
`string`	`next`	`show`		

Keywords are used:

- To specify a data type (`character, location, item, int, string`)
- As part of a statement (`if, then, else, start, end, when, choose, kill, grab, hide, drop, output, next, show`)
- As operators on expressions (`and, or, exists, not`)

3.2.5 Operators

Operators are tokens that specify an operation on at least one operand and yield a result (a value, side effect, or combination). Operands are expressions. Operators with one operand are unary operators, and operators with two operands are binary operators.

Operators are ranked by precedence, which determines which operators are evaluated before others in a statement.

Some operators are composed of more than one character, and others are single characters.

The single character operators are shown in Table 2.
The multiple-character operators are shown in Table 3.

Table 2: Single-character operators

+	-	*	/	>	<	=	.

Table 3: Multiple-character operators

>=	<=	==	!=	and	or	not	exists

3.2.6 Punctuators

Table 4 shows the punctuators in Next. Each punctuator has its own syntactic and semantic significance. Some characters can either be punctuators or operators; the context specifies the meaning.

Table 4: Punctuators

{ }	Compound statement delimiter
( )	Member variable list; also used in expression grouping
,	Member variable separator
;	Statement end
" "	String literal
[? ?]	Probability statements

3.2.7 String and integer literals

Strings are sequences of zero or more characters. String literals are character strings surrounded by quotation marks. String literals can include any valid character, including white-space characters, except for quotation marks.

Integers are used to represent whole numbers. Next does not support floating point numbers. Integers are specified by a sequence of decimal digits. The value of the integer is computed in base 10.

4 Basic Concepts

4.0.8 Blocks and Compound Statements

A block is a section of code consisting of a sequence of statements. A compound statement is a block surrounded by brackets { }.

This is a block:

```
int x = 1;
x = x + 1;
output x;
```

This is a compound statement:

```
{
   int x = 1;
   x = x + 1;
   output x;
}
```

4.0.9 Scope

All declarations of locations, characters, items, strings, and integers have global scope.

Actions are listed within a `choose` statement, and their scope is that `choose` statement. Actions are variables that are given values within that scope, but they are not explicitly declared in a declaration statement.

4.1 Side Effects and Sequence Points

Any operation that affects an operand's storage has a side effect. This includes the assignment operation, and operations that alter the items, attributes, etc. within a location or a character.

Sequence points are checkpoints in the program where the compiler ensures that operations in an expression are concluded. The most important example of this in Next is the semicolon that marks the end of a statement. All

expressions and side effects are completely evaluated when the semicolon is reached.

4.2 Data Types

A type is assigned to an object in its declaration. Table 5 shows the types that are used in Next.

Table 5: Data Types

`int`
`string`
`location`
`character`
`item`

4.3 Declarations

Declarations introduce identifiers (variable names) in the program, and, in the case of the complex types (`character, location, item`), specify important information about them such as attributes. When an object is declared, space is immediately allocated for it, and it is immediately assigned a value. Declarations of integers and strings do not have to include an explicit value; they will be assigned default values of 0 and " ", respectively. Next does not support complex declarations without a value for the variable.

4.3.1 Primitive Types

The primitive types in Next are integer and string. These can stand on their own, or they can serve as attributes within a complex type. Primitive types are declared as follows:

```
int identifier = value
string identifier = value
```

or, receiving default values:

```
int identifier
string identifier
```

where:

- `identifier` stands in for a variable name.
- `value` stands in for an expression.

4.3.2 Complex Types

Each of the complex types in Next (`item, character`, and `location`) has its own declaration syntax. The declarations are as follows:

```
item identifier = { primitive_declaration_list }

character identifier = { primitive_declaration_list,
                         item_list }

location identifier = { primitive_declaration_list,
                        item_list,
                        character_list }
```

where:

- `identifier` stands in for a variable name.
- `primitive_declaration_list` stands in for a list of attribute declarations in the form `(declaration_1, declaration_2, ... declaration_n)`, and each `declaration` is in the form described above in the description of primitive types. These represent the attributes (and values for those attributes) for a given item, character, or location.
- `item_list` and `character_list` stand in for lists of item and character variable names, respectively, in the form `(name_1, name_2, ... name_n)`. These represent the list of items a character is carrying or are found in a location, and the characters that are physically in a location.

- Empty lists are permitted in place of any `primitive_declaration_list`, `item_list` or `character_list`, to indicate that there are none of that type of member variable.

4.4 Expressions and Operators

An expression is a sequence of Next operators and operands that produces a value or generates a side effect. The simplest expressions yield values directly, such as ints, strings, and variable names. Other expressions combine operators and subexpressions to produce values. Every expression has a type as well as a value. Operands in expressions must have compatible types.

4.4.1 Primary Expressions

The most simple type of expressions are those that denote a value directly. These include identifiers that refer to variables that have already been declared, and integer and string literals. In addition, any more complicated expression can be enclosed in parentheses and still be a valid expression.

4.4.2 Overview of the Next Operators

Variables and literals can be used in conjunction with operators to make more complex expressions. Table 6 shows the Next operators.

The Next operators fall into the following categories:

- Unary prefix operators, which precede a single operand.
- Binary operators, which take two operands and perform some arithmetic or logical operation.
- Assignment operators, which assign a value to a variable.

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:

```
x = 7 + 3 * 2;           /* x is assigned 13, not 20 */
```

Table 6: Next Operators

Operator	**Example**	**Description/Meaning**
`.`	`c.a`	Attribute selection in a character, location, or item
`-` [unary]	`-a`	Negative of a
`+`	`a + b`	a plus b
`-` [binary]	`a - b`	a minus b
*	`a * b`	a times b
`/`	`a / b`	a divided by b
<	`a < b`	1 1 if a < b; 0 otherwise
>	`a > b`	1 if a > b; 0 otherwise
`<=`	`a <= b`	1 if a ≤ b; 0 otherwise
`>=`	`a >= b`	1 if a ≥ b; 0 otherwise
`==`	`a == b`	1 if a equal to b; 0 otherwise
`!=`	`a != b`	1 if a not equal to b; 0 otherwise
`and`	`a and b`	Logical AND of a and b (yields 0 or 1)
`or`	`a or b`	Logical OR of a and b (yields 0 or 1)
`not`	`not a`	Logical NOT of a (yields 0 or 1)
`=`	`a = b`	a, after b is assigned to it
`exists`	`exists x.a`	1 if a exists in location x or in the inventory of character x; 0 otherwise

The previous statement is equivalent to the following:

```
x = 7 + (3 * 2);
```

Using parentheses in an expression alters the default precedence. For example:

```
x = (7 + 3) * 2;            /* (7 + 3) is evaluated first */
```

In an unparenthesized expression, operators of higher precedence are evaluated before those of lower precedence. Consider the following expression:

```
A+B*C
```

The identifiers `B` and `C` are multiplied first because the multiplication operator (`*`) has higher precedence than the addition operator (`+`).

Table 7 shows the precedence the Next compiler uses to evaluate operators. Operators with the highest precedence appear at the top of the table; those with the lowest precedence appear at the bottom. Operators of equal precedence appear in the same row.

Table 7: Precedence of Next Operators

Category	**Operator**	**Associativity**
Dot	`.`	Left to right
Gameplay operators	`exists`	Non-associative
Unary	`- not`	Right to left
Multiplicative	`* /`	Left to right
Additive	`+ -`	Left to right
Relational	`< <= > >=`	Left to right
Equality	`== !=`	Left to right
Logical AND	`and`	Left to right
Logical OR	`or`	Left to right
Assignment	`=`	Right to left

Associativity relates to precedence, and resolves any ambiguity over the grouping of operators with the same precedence. Most operators associate left-to-right, so the leftmost expressions are evaluated first. The assignment operator and the unary operators associate right-to-left.

4.4.3 Unary Operators

Unary expressions are formed by combining a unary operator with a single operand. The unary operators in Next are `-`, `not`, and `exists`. The operators `-` and `not` have equal precedence and both have right-to-left associativity, and `exists` has a higher precedence and is non-associative.

Unary Minus

The following expression:

```
- expression
```

represents the negative of the operand. The operand must have an arithmetic type.

Logical Negation

The following expression:

```
not expression
```

results in the logical (Boolean) negation of the expression. If the value of the expression is 0, the negated result is 1. If the value of the expression is not 0, the negated result is 0. The type of the result is `int`. The expression must have a scalar type.

Gameplay Operators (exists)

The following expressions:

```
exists location.x
exists character.y
```

tells us whether `x` is present either in the given location (`x` must be an `item` or `character`) or in the given character's inventory (`x` must be an `item`). The expression returns 1 if x exists and 0 otherwise. The result is type `int`.

4.4.4 Binary Operators

The binary operators are categorized as follows:

- Dot operator (`.`)
- Multiplicative operators: multiplication (`*`) and division (`/`)
- Additive operators: addition (`+`) and subtraction(`-`)
- Relational operators: less than (<), less than or equal to (<=), greater than (>), and greater than or equal to (>=)
- Equality operators: equality (==) and inequality (!=)
- Logical operators: AND (`and`) and OR (`or`)
- Assignment operator (=)

Dot operator

The dot (`.`) operator is used to select from among the attributes, items, or characters within an item, character, or location. An `item` has only attributes; a `character` has attributes and items; and a `location` has attributes, items, and characters. The syntax is:

```
x.y
```

where `x` is the containing object and `y` is the name of the sub-object or attribute which we are trying to access. The result of the expression is a reference to the sub-object `y`.

Multiplicative Operators

The multiplicative operators in Next are `*` and `/`. Operands must have arithmetic type.

The * operator performs multiplication.

The / operator performs division. If two integers don't divide evenly, the result is truncated, not rounded.

Additive Operators

The additive operators in Next are + and -. They perform addition and subtraction respectively.

Relational Operators

The relational operators compare two operands and produce an integer literal result. The result is 0 if the relation is false, and 1 if it is true. The operators are: less than (<), greater than (>), less than or equal (<=), and greater than or equal (>=). Both operands must have an arithmetic type.

The relational operators associate from left to right. Therefore, the following statement first relates a to b, resulting in either 0 or 1. The resulting 0 or 1 is compared with c for the expression result.

```
if ( a < b < c)
{
   statement;
}
```

Equality Operators

The equality operators in Next, equal (==) and not-equal (!=), like relational operators, produce a result of an integer literal. In the following statement, the result is 1 if both operands have the same value, and 0 if they do not:

```
a == b
```

Both operands must have an arithmetic type.

Logical Operators

The logical operators are `and` and `or`. These operators have left-to-right evaluation. The resulting integer literal is either 0 (false) or 1 (true). Both operators must have scalar types. If the compiler can make an evaluation by examining only the left operand, the right operand is not evaluated.

In the following expression,

```
E1 and E2
```

the result is 1 if both operands are nonzero, or 0 if one operand is 0.

In the same way, the following expression is 1 if either operand is nonzero, and 0 otherwise. If expression `E1` is nonzero, expression `E2` is not evaluated.

```
E1 or E2
```

4.4.5 Assignment Operator

There is only one assignment operator (`=`) in Next. An assignment results in the value of the target variable after the assignment. It can be used as subexpressions in larger expressions. Outside of the declaration section, assignment operators can only operate on attributes in Next; these are integer and string literals. Assignment expressions have two operands: a modifiable value on the left and an expression on the right. In the following assignment:

```
E1 = E2;
```

the value of expression `E2` is assigned to `E1`. The type is the type of `E1`, and the result is the value of `E1` after completion of the operation.

4.5 Statements

Statements are executed in the sequence in which they appear in the code, unless otherwise specified.

4.5.1 Labeled Statements

Next uses labeled statements within the content of a `choose` statement. The location name included in a `start` statement is also a type of label. Both of these statements will be discussed in more detail in later sections.

4.5.2 Compound Statements

A compound statement, or block, allows a sequence of statements to be treated as a single statement. A compound statement begins with a left brace, contains (optionally) statements, and ends with a right brace, as in the following example:

```
{
   fred.speed = 5;
   if (fred.strength > enemy.strength)
   {
      enemy.health = enemy.health - 10;
   }
   else
   {
      fred.health = fred.health - 10;
   }
   enemy.strength = 50;
}
```

4.5.3 Expression Statements

Any valid expression can be used as a statement by following the expression with a semicolon.

4.5.4 Selection Statements

A selection statement selects among a set of statements depending on the value of a controlling expression, or, in the case of `prob` statements, the value of a random number. The selection statements in Next are the `if` statement, the `choose` statement, and the `prob` statement.

The if Statement

The `if` statement has the following syntax:

```
if expression then
   statement
else
   statement2
```

The statement following the control expression is executed if the value of the control expression is true (nonzero). The statement in the `else` clause is executed if the control expression is false (0). Next does not allow `if` statements without a corresponding `else` clause.

```
if x + 1 then
   output "x was not negative one";
else
   output "x was negative one";
```

When `if` statements are nested, an `else` clause matches the most recent `if` statement that does not have an `else` clause, and is in the same block.

The choose Statement

The `choose` statement maps keyboard keys to specific actions and executes the statements associated with that action, given user input.

Syntax:

```
/* associates action1 with key1 and action2 with key2 */
choose (action1,action1name,key1) (action2, action2name, key2)
{
    when action1
    {
        statement
    } next location
    when action2
    {
        statement
```

```
    } next location
 }
```

For example:

```
choose (boom,"blowup","a") (punch,"punch","u")
{
    when boom
    {
        link.life = link.life - 80;
    } next palace
    when punch
    {
        [? prob 40 link.life = link.life - 1;
           prob 60 link.life = link.life - 33; ?]
     } next dungeon
}
```

If the game player enters "a" on the keyboard, code within the `boom` block is executed. If the game player enters "u" on the keyboard, code within the `punch` block is executed.

The prob Statement

A probability statement executes a statement randomly according to the given probabilities. It selects and executes a statement randomly given a certain probability distribution. The probabilities listed within a given `prob` statement must add up to 100.

Syntax:

```
[?  prob expression statement
      ...
 ?]
```

For example:

```
/* Increments count by 1 60% of the time
   and decrements count by 1 40% of the time */
```

```
[?  prob 40 count = count-1;
    prob 60 count = count+1; ?]
```

4.5.5 Iteration (start) Statements

Iteration statements in Next begin with the `start` keyword. The statements inside the following block are executed until an end condition is met. The end condition must have a scalar type. An iteration statement is specified by the location in which the action occurs and an end condition that clarifies when gameplay should stop.

The syntax of a `start` statement is as follows:

```
start location_identifier end (expression)
   statement
```

where `location_identifier` is the identifier representing the location, and `expression` represents the end condition. For example:

```
start myplace end (junior.life < 1)
{
   junior.life = junior.life - 1;
   output "You have reached the mountains of myplace!";
}
```

4.5.6 Gameplay Statements

Grab and Drop

The statements:

```
grab y.x;
drop y.x;
```

operate on the item `x` in character `y`'s inventory. The `grab` statement removes an item from the current location and adds in to the character's inventory. The `drop` statement removes an item from the character's inventory and adds it to the current location. The object `x` must be of type `item`.

Hide and Show

The statements:

```
hide y.x;
show y.x;
```

operate on the item or character `x` within the location or character inventory `y`. The `hide` statement removes `x` from `y`, and the `show` statement adds `x` to `y`. The object `y` must be a `location` or `character`. If `y` is a `location`, `x` must be an `item` or `character`; if `y` is a `character`, `x` must be an `item`.

Output

The statement:

```
output expression;
```

outputs `expression` to the screen. Object `expression` must be of type `string` or `int`.

Kill

The statement:

```
kill x;
```

removes `x` permanently from global memory, not just for a given character or location. There is no way to retrieve `x` from any part of the game after this operation has taken place. If you would like to make a character or item disappear temporarily, use the `hide` command. The object `x` must be of type `item` or `character`.

Next

The statement:

```
next l;
```

moves the main character from the current location to location `l`. Code continues execution at the beginning of the `start` statement marked with identifier `l`.