Next
Final Report

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

Contents

1 Introduction

The Next programming language provides a way to easily create text-based computer games. Users of the language can specify characters, locations and items that will appear in the game, and they can design the general plot of the game that ties these together. This language is ideal for creating text-based RPGs (or digital "choose your own adventure" stories).

1.1 Language Elements

Declarations

The program is primarily structured around declarations. The user will create declarations for characters, locations, items, and attributes.

Items

Items are objects that the game player can pick up and maintain in their inventory, and which can be contained in locations. The player could be required to collect item, or he might need a certain item to perform certain actions.

Characters

Characters can appear in locations throughout the game. A declaration for a character will provide the character's name, items he holds, and attribute values.

Locations

Locations provide a basic map of the game's landscape. These are the "rooms" in which a game player can find characters to interact with, items to pick up, or options for actions. For each location in the game, there are two types of declarations. It has a regular declaration, similar to a character declaration, in which its name is declared, as well as items found in that location, characters found in that location, and attributes of the location. Each location also mandatorily has an associated "start statement" which defines what gameplay will take place in that location. This start statement can contain any of the allowable statements in the language.

Actions

The way a player can interact with the game to make choices regarding gameplay is by selecting from various "actions." The programmer defines a list of possible actions, and the code that will be executed when each action is chosen. This is what allows the players to control the plot of the game.

Randomness

A key element of a language designed to create games is some element of randomness, so the gameplay will not be entirely predictable. This is implemented in our language through probability statements, a form of selection statement in which each option is assigned a probability , and the option to be executed is chosen according to what number is returned by a random number generator.

Basic Math

Our language provides the tools for performing basic math, such as addition, subtraction, and boolean operations. These can be used to declare additional variables, update the values of attributes, examining features of attributes for conditional results for actions, or anything else the programmer can think of to make use of them.

2 Language Tutorial

2.1 Tutorial 1

In this section we will create a simple program in the Next programming language that outputs Hello world. Keep in mind that in this step by step each line individually explained is part of a single program. The Next programming language can be thought of as a language made up of declarations. A programmer can declare a number or a string. We will begin by declaring an integer and setting its value to 0:

```
int num = 0;
```

We will then declare a string:

```
string announce = Not in the world;
```

In Next we can also declare items, characters and locations. Here we will declare a character and a location. What follows is a character declaration. The first parenthesis contains the characters attributes while the second contains the items the character is carrying. We will leave the second parenthesis empty because our character is not carrying anything:

```
character  you {(string slogan = Hello world!),()}
```

What follows is a location declaration. The first parenthesis contains the locations attributes, the second contains the items the location contains and the third the characters the location contains. We will leave the first two empty and only include a character in the location declaration. This code looks as follows:

```
location here  {(),(),(you)}
```

There is a special type of declaration in next in which we declare an event to start. Once a location has been declared, a start declaration must be made for that location. In order to use a location in conjunction with a start statement the location must be declared first. In order to use a character in a location that character must be declared first. In order to use an item in a character or a location the item must be declared first. This is why in the Next language it is a good programming practice to create all of the item, character and location declarations first and in that order followed by the start declarations. The start declarations must contain the programming logic for each location because aside from other declarations Next does not allow statements to be placed outside of a start declaration. The start location we will use for this program looks as follows:

```
start here end (num == 1) {
    if (exists here.you) then
    output you.slogan;
    else
    output announce;
    num = 1;

}
```

This start declaration starts the here location and specifies that it will end when integer num is equal to 1. The expression

```
if (exist here.you)  then
```

means if character you exists in location here then execute the code that follows. In this case if true then the slogan attribute from the you character is output with the line

```
output you.slogan;
```

Else the string announce is output. To finish execution of the here start statement integer num is set equal to 1 to meet the end condition. If this had not been done the start statement would iterate forever over its body of code. Because you exists in location here the program will output Hello World! Here is the complete program:

```
int num = 0;
string announce = "not in the world:";
character you {(string slogan = "Hello world!"),()}
location here {(),(),(you)}

start here end (num == 1) {
    if (exists here.you) then
    output you.slogan;
    else
    output announce;
    num = 1;
}
```

2.2 Compiling and Running a Next Program

Once you have created a next program save it with a .next file extension. Go to where the Next file is located. Use the Next file and your .next extension file as input and output the results into a file called Next.java. The command to do this would look like this

```
$ ./Next < Tutorial1.next > Next.java
```

In this example the Next source code file is Tutorial1.next which is located in the same folder as the Next file. This outputs a java file which can then be compiled and executed as any other java source code file. For example to compile the file you could use the command

```
$ javac Next.java
```

Then to run the file you could use

```
$ java Next
```

2.3 Tutorial 2

This tutorial assumes Tutorial 1 has been reviewed by the reader. In this tutorial we will make use of some of the more interesting features included in the Next language. The program that follows will utilize a probability statement to non-deterministically execute segments of code and we will use a choose statement to take simple keyboard input from the user and execute commands accordingly. The commands will make an item show or hide(disappear) from a location and make a character drop or grab an item from a location if the conditions are right or output the automatic error message if it is not possible to perform the action.

We will begin by creating an integer, a location and a character. Because no value will be given to the integer it will take the Next default value of 0. The location and character declarations will be created with empty lists which means that they will not have attributes, items or in the case of the location, characters. Those declarations look as follows:

```
int num;
character person {(),()}
location here {(),(),()}
```

We will then declare an item. Items can only have attributes and in this case we will also leave this items attribute list empty. The declaration looks as follows:

```
item object {()}
```

Next we will declare a start statement, very much like we did in the last example:

```
start here end (num == 1) {
```

Inside this start declaration we will begin by putting a probability statement. Probability statements begin with [? which in Next is known as an opening probability bracket. Probability statements end with a closing probability bracket which looks like this ?]. Inside the probability statement code that is executed with a given probability is defined with the notation prob number statement where number is an integer that specifies the probability that the statement code will be executed. The probability statement can have many such prob number statement clusters. One of them is chosen from inside

the probability statement according to the number part which specifies the probability of that cluster. It is important to remember that the sum of all the integers defined by the number part of the cluster must equal 100. The probability statement used for this program looks as follows:

```
[?
     prob 50 output "This is line 1 of a possible 2";
     prob 50 output "This is line 2 of a possible 2";
 ?]
```

This probability statement means that with a 50 percent probability the line "This is line 1 of a possible 2" will be output. If that line is not output then "This is line 2 of a possible 2" will be output (which, of course, means the second line also has a 50 percent probability of occurring as specified in the probability statement).

Following the probability statement we use two if statements. The first one checks if the item object exists in the person character. It outputs a message that notifies if it does or not. The second if statement checks if the item object exists in location here and again outputs a message that notifies the result. The code looks like this:

```
if (exists person.object) then
     output "The person has the object";
else
     output "The person does not have the object";

if(exists here.object) then
     output "The object is in the location";
else
     output "The object is not in the location";
```

These if statements become useful to easily see the results of the choose statement that follows. A choose statement contains lists of three elements that specify a variable name, a string that will assist in output and a letter that becomes mapped to the keyboard. The first list of the choose statement in this program looks as follows:

```
(grabItem, "character grab item", "g")
```

This means that when the choose statement is executed the output Press g for character grab item will be presented to the user. The g keyboard key will be mapped and if the user presses the g key followed by the enter key the code associated with the grabItem variable will be executed. These associations occur through the use of when statements which will soon be explained. An arbitrary amount of lists for action/keyboard mappings may be used in a choose statement. The choose statement and its lists used in this example are as follows:

```
choose (grabItem, "character grab item", "g")
     (dropItem, "character drop item", "d")
     (showItem, "show item in location", "s")
     (hideItem, "hide item from location", "h")
     (exit, "exit", "e")
```

The lists for a choose statement for action/keyboard bindings are followed by a series of when statements enclosed by braces. These when statements specify code to be executed for each action. At the end of the code specified for each when statement a next statement is used that specifies a location to be executed next. The first when statement in this example is:

```
when grabItem
     grab person.object;
next here
```

This means that when the grabItem action is specified (in this example by the user choosing the g key on the keyboard) the character person will grab the item object if object is present in the current location. This means that if the item object is present in the location grab will cause object to be removed from the location and added to the list of the characters items. If the item is not present in the location when grab is used an error message is output and execution continues with no changes to the character or the location. Once grab is executed in the above when statement the next statement sends execution to the beginning of the here location.

The when statement that follows in our example is:

```
when dropItem
    drop person.object;
next here
```

In this case the drop action is used to have the person character drop the object item in the current location. drop removes an item from a characters item list if the character has that item and adds it to a locations item list. If the character does not have that item an error message is output and execution continues.

After that the following two when statements follow:

```
when showItem
     show here.object;
next here

when hideItem
     hide here.object;
next here
```

show and hide statements are used to add or remove an item or character from a location. In this case the item object is added to the here location by using the show keyword or removed by using the hide keyword. show, hide, drop and grab are used in conjunction with exists to create connections between items, locations and characters in Next programs to simulate real world relationships when creating games.

Finally the last when statement is used to end execution of location here by making its end condition true. The entire program can be seen bellow.

```
int num;

character person {(),()}
location here {(),(),()}
item object {()}

start here end (num == 1) {

     [?
          prob 50 output "This is line 1 of a possible 2";
          prob 50 output "This is line 2 of a possible 2";
     ?]

 if (exists person.object) then
      output "The person has the object";
```

```
 else
      output "The person does not have the object";

 if(exists here.object) then
      output "The object is in the location";
 else
      output "The object is not in the location";


 choose (grabItem, "character grab item", "g")
      (dropItem, "character drop item", "d")
      (showItem, "show item in location", "s")
      (hideItem, "hide item from location", "h")
      (exit, "exit", "e")
 {
     when grabItem
          grab person.object;
     next here

      when dropItem
           drop person.object;
      next here

      when showItem
           show here.object;
      next here

      when hideItem
           hide here.object;
      next here

      when exit
           num = 1;
      next here
 }

}
```

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.
Keywords are used:

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

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		

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.

Table 2: Single-character operators

+	-	*	/	>	<	=	.

The multiple-character operators are shown in Table 3.

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.

3.3 Basic Concepts

3.3.1 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;
}
```

3.3.2 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.

3.4 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.

3.5 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

3.6 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.

3.6.1 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.

3.6.2 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.
- All objects used within an `item_list` or `character_list` must have been declared earlier in the program.

3.7 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.

3.7.1 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.

3.7.2 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 */
```

The previous statement is equivalent to the following:

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

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

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

```
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.

3.7.3 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`.

3.7.4 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
```

The operands must be type `int` or type `string`, and they must be of identical 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
```

3.7.5 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 primitive types; 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.

3.8 Statements

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

3.8.1 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.

3.8.2 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) then
   {
      enemy.health = enemy.health - 10;
   }
   else
   {
      fred.health = fred.health - 10;
   }
   enemy.strength = 50;
}
```

3.8.3 Expression Statements

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

3.8.4 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; ?]
```

3.8.5 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 `y`. The `hide` statement removes `x` from `y`, and the `show` statement adds `x` to `y`. The object `y` must be a `location`. The object `x` must be an `item` or `character`.

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`.

3.9 Start Statements

Start statements are where iteration takes place in Next. They 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. Every location that is declared in the declarations section must have a corresponding start statement somewhere in the following code.

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!";
  }
```

3.10 Example Program

```
int count;

item the_greatest_sword_ever {(int damage = 100000000)}

character xiaowei_the_greatest_man_ever {(int life = 100000000,
int level=99999, string haha="hahahahaha"), (the_greatest_sword_ever)}

location where_is_this_place {(int sizex = 10000, int sizey=9283), (),
(xiaowei_the_greatest_man_ever)}

start where_is_this_place end (xiaowei_the_greatest_man_ever.life < 0) {
     choose (attack,"hia!","a") (up,"up","u") (fin, "end", "f") {
          when attack
          {
              xiaowei_the_greatest_man_ever.life+1;

           } next where_is_this_place

          when up
           {
               [? prob 40 {
                    count = count-1;
                   output "count:";
                    output count;
               }
               prob 60 {
                    count = count+1;
                    output "count:";
                    output count;
               }
                ?]
          } next where_is_this_place
```

```
          when fin
          {
               xiaowei_the_greatest_man_ever.life = -1;
          } next where_is_this_place


     }
}
```

4 Project Plan

4.1 Process

We started work on our language by brainstorming what sort of features we wanted it to include, and sketching out mini programs of the kind we wanted our language to support. We filled up many sheets of notebook paper with ideas for the syntax and structure of our programs. As these ideas solidified, we began to create our parser, scanner, and AST, and develop the Language Reference Manual. The discussion while writing the code for the parser and scanner, while cross-referencing with the LRM, fine-tuned many of the details of our language.

After the parser and scanner were complete, we made the decision to make our we divided the work to create the translator into different sections for each person to work on. Each person was responsible for the code for certain elements of the language, for example expressions, declarations, and the various types of statements.

As the translator code neared completion, we also began work on the testing of our program. We created many small unit tests to test the features of our language, and running these allowed us to see what was and wasn't working properly in our code. As we tested, sometimes implementation details needed to be changed, and we discussed these in the group before changing them.

Throughout the work on our project, we had biweekly meetings to make sure everyone was on the same page and work through issues that required whole-group input.

4.2 Programming Style Guide

4.2.1 Width of the page

80 characters Column Limit

4.2.2 Height of the page

A function should always fit one screenful (of about 70 lines) , at most three

4.2.3 Using tab stops

No Tab Characters is allowed

4.2.4 Comments

Comments Go Above the Code They Reference

4.2.5 Indentation

The change in indentation between successive lines of the program is 2 spaces

indent let ... in constructs

The expression following a definition introduced by let is indented to the same level as the keyword let, and the keyword in which introduces it is written at the end

indent if ... then ... else ...

```
if cond1  then
else if cond2  then
else if cond3  then
```

indent pattern-matching

All the pattern-matching clauses are introduced by a vertical bar, including the first one. For a match or a try align the clauses with the beginning of the construct. If the expression on the right of the pattern matching arrow is too large, cut the line after the arrow.

For Pattern matching in named functions. Indent the clauses 2 spaces with respect to the beginning of the construct

4.3 Project timeline

We kept a calendar to keep track of the important dates for this project.

9/20: Begin biweekly meetings.
9/29: Project proposal submitted.
10/5: Received feedback on proposal, began work on parser/scanner/AST.
11/3: Parser, scanner, AST, and Language Reference Manual complete.
12/1: Ocaml code complete, begin testing and debugging.
12/15: All unit tests written.
12/20: Report drafts finished.
12/22: Project is done!

4.4 Roles and Responsibilities

All members of the team helped write the OCaml code for this project and prepare Next test cases.

Project Leader: Morgan - divided jobs among team members, arbitrated disagreements about language elements, put together the written reports, wrote the implementation for selection statements (if, choose, prob), and much of the supporting Java code.

Language Guru: Ernesto - primary source of ideas for our language and what it should do, wrote scanner, wrote implementation of start statements and action statements (kill, grab, drop, show, hide), wrote the larger sample Next programs.

IT Support: Danny - managed our Github repository, set up automation for test scripts, wrote the implementation of declarations and expressions.

AST Expert: Xiao - wrote much of the parser, go-to person for problem with the parser or AST, wrote code to do type-checking on AST, wrote

implementation of compound statements and output statements.

4.5 Software Development Environment

Our translator was written using OCaml. We used ocamllex and ocamlyacc to create the scanner and parser, respectively. Code was developed in text files and compiled at the command line with a make file. Since our compiler translates code into Java, we also used the Java compiler and JVM to run our programs. Source control was done via Github.

5 Architectural Design

5.1 Architecture Block Diagram

The Next compiler consists of the following major components:

1. Scanner (implemented using Ocamllex)
   Scans the input source file and generates tokens

2. Parser (implemented using Ocamlyacc)
   Parse the tokens from the scanner and generates concrete syntax tree (CST) for the given source program

3. Semantic and type checking module
   Checks the CST for semantic or type errors and also builds up a symbol table

4. Java library code
   Necessary Java class, function and method thats included in the final output java code

5. Java code generator
   Translates the CST to java code and appends the Java library code to the final output

The interaction of these components is shown in figure 1. The Next compiler takes a Next Source File (conventionally suffixed by .Next) and feeds it to the Next scanner implemented using Ocamllex. The scanner captures

the input source file, generates a token stream and passes it to the parser. On receiving the token stream, the parser checks the syntax and builds up the concrete syntax tree. The generated concrete syntax tree is first fed to the checking module to detect any semantic or type errors. During this checking pass, a symbol table is built up as well. After the checking pass, the Java code generator picks up the symbol table, goes through the concrete syntax tree again and translates the CST to java code. In the final stage, the necessary library code is included in the final output java file by the java code generator as well.

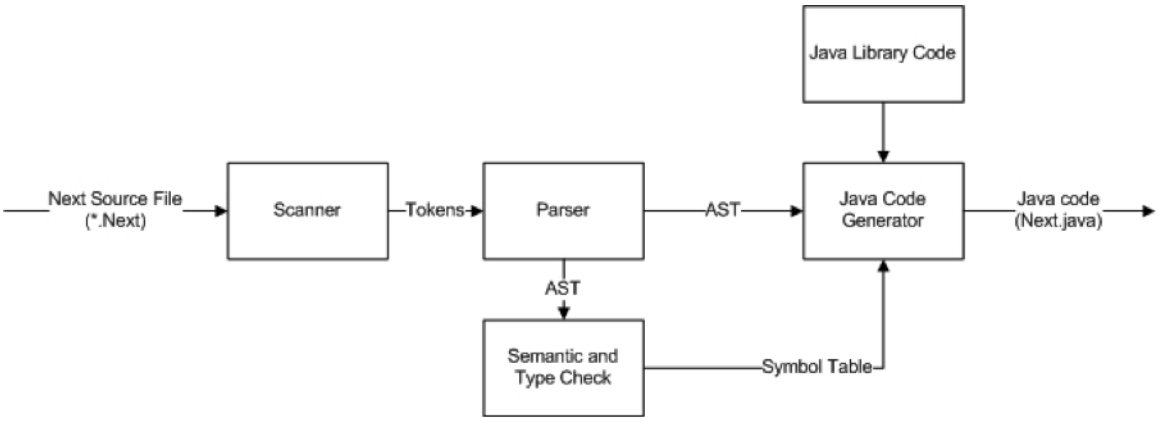

Figure 1: Block diagram of Next compiler

5.2 Interfaces Description

1. Scanner
   Scanner has one public interface which takes an input from the stdin and returns a token stream. If any invalid token is captured, an exception will be raised

2. Parser
   Parser has one public interface which takes the token stream from the scanner as input and returns the generated CST. If any syntax error is captured, a parsing exception will be raised.

3. Semantic and type checking module
   This checking module has one major public interface which takes the

CST generated by the parser and returns a symbol table. Also this module provides several query functions for the Java code generator to call to determine the type of variables or expressions. Namely, Check_id function takes in an identifier in the CST together with the symbol table and returns the type of the identifier.
Check_expr function takes in an expression in Next and returns the type of the expression These functions are called by the Java code generator to when its necessary to know the type of the expression or identifier.

4. Java code generator
The code generator provides one public interface which takes the symbol table and the CST and output the final java code to stdout. The code generator calls the check_id and check_expr functions provided by the checking module when the type information about a specific identifier or expression is need.

6 Test Plan

Tests were first created to cover all aspects of the LRM. The LRM was split up into 4 parts and each member wrote tests that covered aspects of the Next language that was discussed in the portion of the LRM that was assigned to them. These tests were automated using a Perl script (RunTests.pl) which first ran the Next compiler on the Next test code, second compiled the resulting Java code, third ran the Java program, and last compared the output of the Java program with a reference file. If any tests failed, this was recorded in a log file. Because the Next language allows the developer to create programs that are non-deterministic through manual input or through the use of probability, we decided to test these aspects of the Next language manually.

7 Lessons Learned

7.1 Ernesto

This semester was filled with lots of lessons to be learned in PLT. It was interesting to see how while creating our language, every time we hit a stumbling block on the creation of our AST the solution was to create a new type that took the other elements of the language bellow it (another level of

indirection). In general just learning about tokens and how languages work changed my way of looking at programming. It was exciting to see how expressions resolve and become statements to go on into declarations (at least for the language my team created). I also learned about the many considerations that go into making a programming language such as scope, syntax and structure. Specifically for the language we created I learned about flexibility. Our language would have been a lot more flexible and intuitive if we had used smaller statements such as separating the next statement from the when statement. It is true that less is better in this case. It was also interesting to see how programming languages currently seem to be set to work in certain typical ways but really, they can be organized and made to work in any way as long as a language is well defined. It leaves a lot of room for creativity.

Regarding team work I learned that having strong and involved team members and a person with the ultimate word really helps. Disputes are resolved with a lot more ease if one person has the last say. It is a lot more efficient.

Regarding development I learned that Ocaml is a fantastic language when it comes to creating programming languages because of its strong data type system and the way its functions are created and used. At the same time I learned about the weaknesses of trying to port a language that uses integer values to express true/false to a language that uses Boolean values. Expressions can become quite complex to handle.

For students next semester I would agree that it is definitely good to start early, especially on the details of the language. My team had the parser, scanner and AST done within a month and it gave us a lot of room to think about what we had and tweak decisions with something to fall back on rather than finding out last minute something is terribly wrong. Also a system of checks I would say helps. Have someone specialize on a certain part of the report but have someone else overview and understand what is happening. This helps for decisions to get a second opinion for debugging and error checking and when the rest of the tea does not understand something the expert is usually too immersed on the material to be able to explain things well but a person who understands but is not so impregnated is able to clarify concepts in lay-mans terms.

7.2 Danny

1. Pick a source control you are comfortable with.

I proposed to the group that we use Git as our source control program. Although learning a new source control seemed exciting and like a great idea at the time, if I had to do it again, I would have picked SVN or CVS. The reason being is that Git worked for us most of the time but when things went wrong, such as a merge, it was very frustrating and time consuming figuring out how to fix it. There were a couple of times where I lost work because of some mix-up. Time lost figuring out Git could have been used better elsewhere.

2. Set Goals

Our group had a Google Calendar that we setup early in the semester with weekly meeting times as well milestones that we wanted to hit. We did not always hit our milestones, for example being code complete by Thanksgiving, but it did provide a sense of where we were compared to where we wanted to be. This translated into us working harder in order to stay on schedule so we would not be dealing with a waterfall of work in the last week.

3. Meet Regularly (In person or electronically)

I think this was the main reason that made our group as efficient as it was. Regular meetings allowed us to clarify and refine our language and make incremental changes as we went along as opposed to changing the whole architecture of our language every 2-3 weeks. This also helped to keep everybody accountable since it is hard not to do any work when everybody else has something to show every couple days.

7.3 Morgan

It really helps to very specifically designate jobs and responsibilities to each person. This ensures that everyone is clear on what they should be doing, and it helps the work get done more efficiently. It is also useful when it comes to debugging and fixing errors in the code to know exactly who was responsible for that section, so that someone familiar with the design of that section can be the expert on fixing that problem.

Communication is also key. Our group met twice a week to discuss our project and work on issues that had come up. This system made sure that everyone was always on the same page in regard to the project. We also communicated frequently via email as problems came up, which was also effective as all members of our group responded quickly to emails.

Our project ran very smoothly overall, so I think that the systems we had in place worked well. It also helps if all members of the group have a good work ethic and are responsible in completing tasks, which we were lucky in having as well.

7.4 Xiaowei

Communication is very important, especially in a collaborative project like this. It really takes time for everyone to understand each other (ambiguous in language might be the cause, need a better parser). Source control is another important thing. If we dont have the source control, I can imagine what a mess it is going to be.

8 Appendix

Attached to the end of this report is a complete code listing of our translator.