Jairo Pava
COMS W4115
June 28, 2013

LEARN: Language Reference Manual

Contents

1 Introduction

Understanding concepts can sometimes be difficult for people when they are being introduced to them for the first time. Self-assessment has consistently demonstrated the ability to lead to faster understanding and longer retention of new information. In the classroom environment, self-assessment typically takes place in the form of homework or in-class exercises. After completing the exercises, students must wait for the professor to grade them and provide feedback. This grading process is slow, particularly if the professor chooses to provide high quality feedback, and may result in a significant period of time before a student receives a reply.

The Language for Exercise And Response Notification (LEARN) aims to help professors quickly create sets of exercise questions that automatically grade, provide feedback, and adapt to the student's understanding of material.

2 Lexical Conventions

2.1 Tokens

There are six categories of tokens in LEARN: identifiers, keywords, constants, string literals, operators, and separators. White space is used to separate tokens and includes blanks, horizontal and vertical tabs, newlines, form feeds, and comments.

Tokens are retrieved as sequences of characters from an input stream and are identified based on the largest matching sequence.

2.2 Comments

The characters [* begin a comment and the characters *] complete a comment. Comments do not nest. They are meant to describe the code and will be ignored by the compiler.

2.3 Identifiers

An identifier is a sequence of letters, digits, and underscores that must begin with a letter or an underscore. Identifiers are case insensitive and may have up to 32 characters.

2.4 Keywords

The following identifiers are reserved for use as keywords and may not be used for any other reason:

if	start	Question
else if	end	Execute
else	prompt	var
repeat	choice	OR
correct	answer	AND

2.5 Constants

2.5.1 Integer Constants

Integer is defined as a series of numerical digits that must begin with the digits 1 through 9, and may only begin with a zero if the number is zero.

2.6 String Literals

A string literal begins with a double quote (as in "), is followed by a sequence of zero or more ASCII characters, and terminates with a double quote. A double quote may be included within the string if it is escaped by immediately preceding it with a backslash.

3 Types

Four data types are defined: boolean, integer, array, and question. Since LEARN is dynamically typed, a type is not declared when a variable is defined. Type conversion is not supported.

3.1 Boolean

There exist two types of boolean: true and false.

3.2 Integer

Integers range from a minimum value of -2,147,483,648 to a maximum value of 2,147,483,647.

3.3 Array

The array data type describes an array of elements of the any data type except array. The size of the array must be defined at the declaration of the array.

3.4 Question

The question data type describes a question. It must define a prompt, an array of correct answers and, optionally, a set of answer choices.

4 Syntax

The following sections of this manual will use ***bold italicized*** font to identify grammar production and a `monospaced` font to identify literal symbols.

5 Expressions

Expressions consist of at least one operand and zero or more operators and produce a value or generate a side effect. Operands are typed objects such as constants and variables. Operators within this section are defined in order of precedence. Operators within the same subsection have the same order of precedence.

5.1 Primary

Identifiers, literal integers, and literal strings are primary expressions. An expression may be delimited by parentheses to change precedence of operator. An expression within parenthesis has the same type as the expression without parenthesis would have. A primary expression may also include a semicolon-delimited list of expressions.

5.2 Variable

$ ***identifier***

The $ operator is used to reference and retrieve the value stored in a variable.

5.3 Array References

$ ***identifier*** [***expression***]

The expression within the left and right brackets must evaluate to an integer that refers to an element of an array. This expression must be greater than or equal to zero and must be less than the size of the array. Otherwise, an error will be produced and the program will halt execution. The $ operator is used to reference and retrieve the value stored in the specified element of the array.

5.4 Unary

Support is provided for negation and logical negation unary operators. Unary operators are grouped from right to left.

5.4.1 Negation

- ***expression***

The - operator performs negation. The operation is defined for integer types. The resulting type is an integer.

5.4.2 Logical Negation

! ***expression***

The ! operator performs logical negation. The operation is defined for boolean types. The resulting type is a boolean.

5.5 Arithmetic

Support is provided for standard arithmetic operations: multiplication, division, modular division, addition, and subtraction.

5.5.1 Multiplicative

All multiplicative operators are grouped from left to right.

5.5.1.1 Multiplication

expression* * *expression

The multiplication operator performs multiplication of two operands. The * operator is used to indicate multiplication. The operation is defined for integer types. The resulting type is an integer.

5.5.1.2 Division

expression* / *expression

The division operator performs division of two operands. The / operator is used to indicate division. The operation is defined for integer types. The resulting type is an integer. If the divisor is not a factor of the dividend, then only the integer part of the resulting division will be returned. If the divisor is equal to zero, an error message will be printed and the program will exit.

5.5.1.3 Modulus

expression* % *expression

The modulus operator performs modular division of two operands. The % operator is used to indicate modular division. The operation is defined for integer types. The resulting type is an integer.

5.5.2 Additive

All additive operators are grouped from left to right.

5.5.2.1 Addition

expression* + *expression

The addition operator performs addition of two operands. The + operator is used to indicate addition. The operation is defined for integer types. The resulting type is an integer.

5.5.2.2 Subtraction

expression* - *expression

The subtraction operator performs subtraction of two operands. The - operator is used to indicate subtraction. The operation is defined for integer types. the resulting type is an integer.

5.6 String concatenation

expression* ^ *expression

The ^ operator is used to indicate string concatenation. The operation is defined for strings. the resulting type is a string.

5.7 Relational

Support for relational operators is provided to determine how two operands relate to each other: greater than, greater than or equal to, less than, and less than or equal to. Relational operators will always result in either a true or false boolean type. Relational operators group from left to right.

5.7.1 Greater than

expression* > *expression

The > operator is used to indicate a greater than comparison. The operation is defined for integers. The resulting type is a boolean and will return true if the left operand is greater than the right operand. Otherwise, the comparison will return false.

5.7.2 Greater than or equal to

expression* >= *expression

The >= operator is used to indicate a greater than or equal to comparison. The operation is defined for integers. The resulting type is a boolean and will return true if the left operand is greater than or equal to the right operand. Otherwise, the comparison will return false.

5.7.3 Less than

expression* < *expression

The < operator is used to indicate a less than comparison. The operation is defined for integers. The resulting type is a boolean and will return true if the left operand is less than the right operand. Otherwise, the comparison will return false.

5.7.4 Less than or equal to

expression* <= *expression

The <= operator is used to indicate a less than or equal to comparison. The operation is defined for integers. The resulting type is a boolean and will return true if the left operand is less than or equal to the right operand. Otherwise, the comparison will return false.

5.8 Equality

Support for equality operators is provided to determine if two operands are or are not equal to each other. Equality operators will always result in either a true or false boolean type. Equality operators group from left to right.

5.8.1 Equal to

expression* == *expression

The == operator is used to indicate an equal to comparison. The operation is defined for integer and string types. Integers must be compared with integers and strings must be compared with strings. Otherwise, the program will produce an error. The resulting type is a boolean and will return true if the left operand is equal to the right operand. Otherwise, the comparison will return false. String comparisons are case sensitive.

5.8.2 Not equal to

expression `!=` ***expression***

The != operator is used to indicate a not equal to comparison. The operation is defined for integer and string types. Integers must be compared with integers and strings must be compared with strings. Otherwise, the program will produce an error. The resulting type is a boolean and will return true if the left operand is not equal to the right operand. Otherwise, the comparison will return false. String comparisons are case sensitive.

5.9 Logical comparison

Support for logical comparison is provided to evaluate the boolean return type of two operands. If an evaluation may be performed by evaluating only the first operand, the second operand will not be evaluated. Logical comparison operators group from left to right.

5.9.1 OR

expression `OR` ***expression***

The OR operator is used to indicate a logical or comparison. This expression will return true if at least one of its operands evaluates to true. As a result, if the first operands evaluates to true, the second operand will not be evaluated.

5.9.2 AND

expression `AND` ***expression***

The AND operator is used to indicate a logical and comparison. This expression will return true if both of its operands evaluate to true. As a result, if the first operands evaluates to false, the second operand will not be evaluated.

6 Declarations

Declarations are used to introduce identifier names and types to be used in a program. Identifiers must be declared before they are used.

6.1 Variable

`var` ***identifier***

Variables are declared using the var keyword followed by an identifier.

6.1.1 Variable Initialization

`var` ***identifier* = *expression***

A variable may be initialized upon declaration. Initialization consists of an equal sign followed by a constant expression. The initialization declares the variable, the variable type (based on the return type of the expression), and defines the value of the variable.

6.2 Array

`var` ***identifier***`[`***expression***`]`

Arrays are declared with the var keyword followed by an identifier followed by a pair of brackets. An expression within the brackets must evaluate to an integer greater than zero that indicates the maximum number of elements allowed in the array. This expression must be omitted if the array is initialized with a list of constant expressions. Elements within the array may be of any supported data type. Arrays are single-dimensional.

6.2.1 Array Initialization

var *identifier*[] = `[`***expression-list***`]`

Arrays are initialized with a brace-enclosed, semicolon delimited list of constant expressions. The number of expressions in the expression list determines the size of the array.

6.3 Question

`Question` ***identifier***

question-compound-statement

Questions must be declared at the beginning of the program. Declaration of questions must begin with the `Question` keyword followed by an identifier followed by a compound statement. The prompt and answer keywords must be defined within the compound statement. The choice keyword may optionally be defined as well. Other statements may optionally precede these keyword declarations.

The prompt keyword must be defined with a string and will be used by LEARN to provide the user with the question. The optional choice keyword must be defined with an array and will be used by

LEARN to provide the user with a set of choices. The answer keyword must be defined with an array and will be used by LEARN to check if the user's provided answer or answers are correct. If the programmer has defined a choice keyword that does not have one or more elements defined in the answer keyword, those elements will be automatically added to the array of choices. For example:

```
Question exampleQuestion
{
    prompt = “This is the question”;
    choice = [“A” ; “B” ; “C” ; “D”];
    answer = [“D”];
}
```

The logic above will generate a multiple-choice question with one correct answer. Any variables created within the question block will be scoped within that block. For example, if a variable is defined in one question block, it will not be accessible in another question block.

LEARN will execute the logic within the question block, prompt the user, wait for the user to provide an answer, and then verify whether the answer provided is correct. If the answer provided is correct then the value of the correct keyword, as described in section 7.6, will be set to true. Otherwise, the value of the correct keyword will be set to false.

6.4 Execution

`Execute`

execute-compound-statement

After questions have been declared, the order with which they are presented to the user is declared in the `Execute` block. This block may only be declared once and it may only be declared after the last question declaration. Declaration of the `Execute` begins with the `Execute` keyword and must be followed by a compound statement. The compound statement must consist of a start state declaration, end state declaration, and question statements as described in section 7.6.

7 Statements

7.1 Expressions

expression `;`

A statement can be any valid expression by following the expression with a semicolon.

7.2 Compound

`{`***statement-list***`}`

A block, or compound statement, permits a single statement to be a sequence of statements. A compound statement begins with a left curly bracket. It is then followed by zero or more statements. It ends with a right curly bracket.

7.3 Conditional

A conditional statement selects among a set of statements depending on the true or false value of one or more controlling expressions. The conditional statement has the following syntax:

`if(`***expression***`)`

  compound-statement

`else if(`***expression***`)`

  compound-statement

`else`

  compound-statement

The compound statement following the expression is executed if the value of the expression is true. The conditional statement may contain zero or more `else if` to describe multiple controlling expressions. The conditional statement may also contain one optional `else` to describe a compound statement that is executed if the last control expression evaluates to false.

Conditional statements may be nested. In nested conditional statements, an `else` will match the most recent `if` or `else if` that does not already have an `else` and is in the same compound statement.

7.4 Loops

A loop repeatedly executes a statement until a condition is met. The loop has the following syntax:

`repeat(`***expression*** `;` ***expression***`)`

  compound-statement

A semicolon is used to separate the control expressions. The value of the expression to the left of the semicolon is used determine whether to terminate the loop. If the value of the expression is false, then the loop is terminated. The expression is evaluated before each loop iteration.

The expression to the right of the semicolon is optional and, if present, is evaluated after each iteration. It is evaluated before the expression to the left of the semicolon.

7.5 Assignment

7.5.1 Variable

identifier* = *expression

There are two operands in assignment expression: a modifiable variable represented by an identifier on the left and an expression that evaluates to the value to be assigned on the right. Assignment is represented by an equal sign between the two operands. The type of the identifier is equal to the type of the expression. A variable type may not be reassigned after it has been declared. The value of the expression is assigned to the identifier.

7.5.2 Array

identifier* [*expression*] = *expression

There are three operands in assignment expression for arrays: a modifiable array represented by an identifier, an expression that identifies the integer location within the array to perform assignment, and an expression that evaluates to the value to be assigned. Assignment is represented by an equal sign between the two operands. The type of the identifier is equal to the type of the expression. The value of the expression is assigned to the identifier.

The expression that identifies the location within the array to perform assignment must be greater than or equal to zero and must be less than the size of the array. Otherwise, an error will be produced and the program will halt execution.

7.6 Question Transition

identifier* [*expression*]$_{opt}$ -> *identifier

The -> operator is used to indicate the transition from one question to another. The operation is defined for question types. The operation does not return anything but does have the side effect of setting the value of the correct keyword to true or false depending on whether the previous question was answered correctly. For example:

```
start -> q1;
q1 [correct] -> q2;
q1 [!correct] -> q3;
q2 -> end;
q3 -> end;
```

Two states are defined in LEARN, start and end, that must be part of the question flow that is defined. Brackets may optionally be used to impose a condition through an expression that must evaluate to either true or false for transition between one questions. In this example question flow, a user will be unconditionally prompted with question 1. If the user provides a correct answer to question 1, then the user is prompted with question 2. Otherwise, the user is prompted with question 3. Regardless of the answer provided to question 2 or question 3, the user will transition to the end of the question flow and the program will exit.

8 Scope

Variables are statically scoped and separated by blocks which begin with a left curly bracket (as in {) and end with a right curly bracket (as in }).

9 Built-in functions

9.1 Random Integer

random_integer(expression, expression)

Randomly returns an integer inclusively between both expression arguments which must be of type integer.

9.2 Array Length

length(expression)

Returns the number of elements present in the array. The argument must be an expression of type array.

10 Example

10.1 Program

```
[* Create multiple-choice question with one correct answer *]
Question q1
{
    prompt = "Convert 2/4 to decimal";
    choice = [“0.25” ; “0.75” ; “1.00”];
    answer = [“0.50”];
}

[* Create multiple-choice question with multiple correct answers *]
Question q2
{
    var a = ["1/2"; "2/4"; "4/8"];
    var b = random_integer(0, 2);
    var c = random_integer(0, 2);
    prompt = "Convert 0.50 to fraction.";
    choice = ["1/3" ; "2/3"];
    answer = [$a[$b] ; $a[$c]];
}

[* Create free-response question *]
Question q3
{
    var a = random_integer(1, 10);
    var b = random_integer(1, 10);
    prompt = "Convert " ^ $a ^ "/" ^ $b ^ " to integer. Round to lowest integer."
    answer = [$a/$b];
}

[* Create flow of questions *]
Execute
{
    start["Welcome!"];
    end["Good bye!"];

    start -> q1;
    q1 [correct] -> q2;
    q1 [!correct] -> q3;
    q2 -> end;
    q3 [until correct] -> end;
}
```

10.2 Output

```
Welcome!
[Multiple-choice] Convert 2/4 to decimal.
1. 0.75
2. 0.50
3. 1.00
4. 0.25
> 2
Correct!
[Multiple-choice. Select more than one.] Convert 0.50 to fraction."
1. 1/2
2. 2/3
3. 1/3
4. 4/8
> 1;4
Correct!
Good bye!
2 out of 2 answered correctly.
```

11 Language Summary

Listed below is the grammar for LEARN. Productions that have the suffix ***opt*** are optional. That is, either no token is chosen or the production without the ***opt*** token is chosen. The following tokens are passed in from the lexer and are undefined in this grammar: ***identifier***, ***integer-literal***, ***boolean-literal***, and ***string-literal***.

learn-program:

question-declaration-list execute-declaration

question-declaration-list:

question-declaration

question-declaration-list question-declaration

question-declaration:

`Question` ***identifier question-compound-statement***

execute-declaration:

`Execute` ***execute-compound-statement***

question-compound-statement:

{***statement-list***$_{opt}$ ***prompt-var-declaration choice-var-declaration***$_{opt}$ ***answer-var-declaration***}

execute-compound-statement:

{***start-state-declaration end-state-declaration statement-list**$_{opt}$*}

prompt-var-declaration:

`prompt` = [***expression-list***]

choice-var-declaration:

`choice` = [***expression-list***]

answer-var-declaration:

`answer` = [***expression-list***]

start-state-declaration:

`start` ***block-expression***

end-state-declaration:

`end` ***block-expression***

compound-statement:

{***statement-list**$_{opt}$*}

statement-list:

statement

statement-list statement

statement:

expression ;

conditional-statement

loop-statement

assign-statement ;

var-declaration ;

array-declaration `;`

question-transition-statement `;`

conditional-statement:

***if-statement else-if-statement-list**$_{opt}$ **else-statement**$_{opt}$*

if-statement:

`if (`***expression***`)` ***compound-statement***

else-if-statement-list:

else-if-statement

else-if-statement-list else-if-statement

else-if-statement:

`else if (`***expression***`)` ***compound-statement***

else-statement:

`else (`***expression***`)` ***compound-statement***

loop-statement:

`repeat (`***expression*** `;` ***expression***`)` ***compound-statement***

assign-statement:

var-assign-statement

array-assign-statement

var-assign-statement:

identifier = expression

array-assign-statement:

identifier block-expression = expression

var-declaration:

`var` ***identifier***

`var` ***identifier*** = ***expression***

array-declaration;

`var` ***identifier block-expression***

`var` ***identifier block-expression*** = `[`***expression-list***`]`

question-transition-statement:

identifier block-expression$_{opt}$ `->` ***identifier***

block-expression:

`[`***expression***`]`

expression-list:

expression

expression-list `;` ***expression***

expression:

or-expression

or-expression:

and-expression

or-expression `OR` ***and-expression***

and-expression:

equality-expression

and-expression `AND` ***equality-expression***

equality-expression:

relational-expression
equality-expression **==** *relational-expression*
equality-expression **! =** *relational-expression*

relational-expression:
string-concat-expression
relational-expression **>** *string-concat-expression*
relational-expression **>=** *string-concat-expression*
relational-expression **<** *string-concat-expression*
relational-expression **<=** *string-concat-expression*

string-concat-expression:
additive-expression
string-concat-expression **^** *additive-expression*

additive-expression:
multiplicative-expression
additive-expression **+** *multiplicative-expression*
additive-expression **–** *multiplicative-expression*

multiplicative-expression:
unary-expression
multiplicative-expression ***** *unary-expression*
multiplicative-expression **/** *unary-expression*
multiplicative-expression **%** *unary-expression*

unary-expression:
array-reference-expression
- *unary-expression*
! *unary-expression*

array-reference-expression:

  variable-expression

  $ ***identifier block-expression***

variable-expression:

  primary-expression

  $ ***identifier***

primary-expression:

  integer-literal

  string-literal

  boolean-literal

  identifier

  (***expression***)