Jairo Pava (jap2216)
COMS W4115
August 16, 2013

LEARN: Final Report

Contents

1 Introduction....2
2 Language Tutorial....2
2.1 How to compile and run....2
2.2 Example....2
3 Language Reference Manual....4
3.1 Lexical Conventions....4
3.2 Types....5
3.3 Syntax....6
3.4 Expressions....6
3.5 Declarations....11
3.6 Statements....13
3.7 Scope....14
3.8 LEARN Program....15
3.9 Built-in Variables....15
3.10 Example....16
3.11 Language Summary....17
4 Project Plan....22
4.1 Roles....22
4.2 Development Environment....22
4.3 Time-line....22
5 Architectural Design....23
6 Test Plan....24
6.1 Goals....24
6.2 Unit Testing....24
6.3 Automated Regression Testing....24
7 Lessons Learned....25
7.1 Test early and often....25
7.2 Start early....25
7.3 Address a meaningful problem....25
7.4 Keep it simple....25
8 Complete Listing....26
8.1 Scanner....26
8.2 Parser....27
8.3 Abstract Syntax Tree....32
8.4 Bytecode....36
8.5 Compiler....38
8.6 Executor....47
8.7 Driver....51

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 Language Tutorial

2.1 How to compile and run

LEARN programs are compiled and run from the learn executable. The executable may be generated by running the command 'make' inside the LEARN directory. Once generated, the executable may be run as follows:

learn [options] File

Options

-a

Prints the abstract syntax tree

-b

Compiles and prints the bytecode

-c

Compiles and runs the program. This is the default option.

File

Path to LEARN file to process

Example

./learn -c my_progam.learn

2.2 Example

A LEARN program consists of the following declarations: global variables, functions, questions, and an execute statement. Global variables and functions are optional. The order of the declarations may not be changed.

The following example will be used to highlight the main features of the LEARN programming language.

```
[* This program will quiz a user on the GCD of two numbers *]

[* Global variable *]
var a = 12;

[* Function to calculate GCD of two integers *]
[* Receives two integer arguments *]
[* Returns one integer *]
function gcd(a, b)
{
    [* Note that "a" refers to locally scoped variable *]
    repeat( a != b ; )
    {
        if( a > b)
        {
            a = a - b;
        }
        else
        {
            b = b - a;
        }
    }
    return a;
}

[* This question will ask the user to provide GCD of two numbers,
   it will calculate the GCD, provide the user with choices, and
   will grade the user's response *]
question q1
{
    var b = 18;
    var x = gcd(a, b);

    [* prompt, choice, and answer are built-in variables *]
```

```
    prompt = "What is the GCD of " ^ a ^ " and " ^ b ^ "?";
    choice = {"2", "4", x ^ "", "8"};
    answer = {x ^ ""};
}

[* This execute block will drive the order with which the user
   is prompted with questions *]
execute
{
    [* Continue to ask the user, until the correct answer
       is provided *]
    [* correct is a global built-in variable *]
    repeat(!correct ; )
    {
        -> q1;
    }
}
```

In the sample program above, the user will be continuously prompted to provide the GCD of two integers until the correct answer is provided. The comments in the program above (text between [* and *]) attempt to summarize the program features. For more detailed descriptions, see the Language Reference Manual.

3 Language Reference Manual

3.1 Lexical Conventions

3.1.1 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.

3.1.2 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.

3.1.3 Identifiers

An identifier is a sequence of letters, digits, and underscores that must begin with a letter. Identifiers are case sensitive.

3.1.4 Keywords

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

if	return	and
else if	question	true
else	execute	false
repeat	var	
function	or	

3.1.5 Constants

3.1.5.1 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.

3.1.5.2 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.

3.1.5.3 Boolean Constant

Boolean is defined as either a true or false constant.

3.1.5.4 Array Constant

An array constant begins with a left bracket, is followed by a comma-delimited list of expressions, and terminates with a right bracket.

3.2 Types

Four data types are defined: boolean, integer, array, and string. Since LEARN is dynamically typed, a type is not explicitly declared when a variable is defined. Instead, the LEARN compiler automatically determines a variable's type upon variable declaration. The LEARN bytecode executor, on the other hand, determines the type of a function's formal parameters and return value since these can depend on the run-time circumstances in which the function is called. Type conversion is not supported.

3.2.1 Boolean

There exist two types of boolean: true and false.

3.2.2 Integer

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

3.2.3 Array

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

3.2.4 String

The string data type describes a set of ASCII characters enclosed in double quotes.

3.3 Syntax

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

3.4 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.

3.4.1 Primary

Identifiers, literal integers, booleans, arrays, 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.

3.4.2 Variable

identifier

A bare identifier is used to reference and retrieve the value stored in a variable represented by the identifier.

3.4.3 Array References

identifier [***expr***]

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.

3.4.4 Function Call

identifier **(** ***expr-list***$_{opt}$ **)**

A function call consists of an identifier followed by open and close parenthesis which may optionally contain a comma-separated list of expressions. LEARN uses applicative order evaluation for function arguments. Arguments are evaluated from left to right.

3.4.5 Unary

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

3.4.5.1 Negation

- ***expr***

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

3.4.5.2 Logical Negation

! ***expr***

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

3.4.6 Arithmetic

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

3.4.6.1 Multiplicative

All multiplicative operators are grouped from left to right.

3.4.6.1.1 Multiplication

expr ***** ***expr***

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

3.4.6.1.2 Division

expr / expr

The division operator performs division of two operands. The / operator is used to indicate division. The operation is only 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.

3.4.6.1.3 Modulus

expr* % *expr

The modulus operator performs modular division of two operands. The % operator is used to indicate modular division. The operation is only defined for integer types. The resulting type is an integer. If both operands are positive, the result is positive. If either operand is negative, the sign of the result is the same as the sign of the left operand.

3.4.6.2 Additive

All additive operators are grouped from left to right.

3.4.6.2.1 Addition

expr* + *expr

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

3.4.6.2.2 Subtraction

expr* - *expr

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

3.4.7 String concatenation

expr* ^ *expr

The ^ operator is used to indicate string concatenation. The operation is only defined for strings, integers, and boolean data types. The resulting type is a string.

3.4.8 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.

3.4.8.1 Greater than

expr > expr

The > operator is used to indicate a greater than comparison. The operation is only 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.

3.4.8.2 Greater than or equal to

expr >= expr

The >= operator is used to indicate a greater than or equal to comparison. The operation is only 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.

3.4.8.3 Less than

expr < expr

The < operator is used to indicate a less than comparison. The operation is only 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.

3.4.8.4 Less than or equal to

expr <= expr

The <= operator is used to indicate a less than or equal to comparison. The operation is only 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.

3.4.9 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.

3.4.9.1 Equal to

expr == expr

The == operator is used to indicate an equal to comparison. The operation is only defined for integer, string, and boolean types. Integers must be compared with integers, strings must be compared with strings, and booleans must be compared with booleans. 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.

3.4.9.2 Not equal to

expr `!=` ***expr***

The != operator is used to indicate a not equal to comparison. The operation is only defined for integer, string, and boolean types. Integers must be compared with integers, strings must be compared with strings, and booleans must be compared with booleans. 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.

3.4.10 Logical comparison

Support for logical comparison is provided to evaluate the boolean return type of two operands. Logical comparison operators group from left to right.

3.4.10.1 Or

expr `or` ***expr***

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.

3.4.10.2 And

expr `and` ***expr***

The and operator is used to indicate a logical and comparison. This expression will return true if both of its operands evaluate to true.

3.4.11 Assignment

3.4.11.1Variable

identifier = expr

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.

3.4.11.2Array

identifier `[` ***expr*** `]` ***= expr***

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.

3.4.12 Question Execution

`->` ***identifier***

`->` { ***identifier-list*** }

The -> operator is used to have LEARN ask the user a pre-defined question. It may only be written within the execute block and is only defined for question types. The operation does not return anything but does have the side effect of setting the value of the built-in global variables *correct* to true or false and incrementing the *correctCount* depending on whether the previous question was answered correctly. The built-in global variable *askCount* is also incremented for every question that is asked. For example:

```
-> q1;
if(correct)
{
   -> {q2, q3};
}
```

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 questions 2 and 3. Otherwise, the user is not prompted with anymore questions. Afterwards, the program will exit.

3.5 Declarations

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

3.5.1 Variable

Variables must be initialized upon declaration. A variable may be initialized in the two ways described below.

3.5.1.1 Expression Initialization

`var` ***identifier = expr***

Initialization consists of an equal sign followed by an expression. The declaration explicitly describes the variable identifier, and the initialization implicitly describes the variable type (based on the return type of the expression), and defines the value stored in the variable.

3.5.1.2 Array Initialization

`var[`***expr***`]` ***identifier*** **=** ***expr***

Arrays are declared with the var keyword followed by a pair of square brackets followed by an identifier. An expression within the brackets must evaluate to an integer greater than zero that indicates the size of the array. The array declaration is completed by an equal sign followed by an expression. This expression is used to determine the value that each element in the array will be initialized with.

3.5.2 Function

`function (` ***id-list***$_{opt}$ `) {` ***var-decl-list***$_{opt}$ ***stmt-list*** `}`

Functions must be declared after global variable declarations. They must be declared with the "function" keyword, followed by open and close parentheses which may optionally contain within comma-separated identifiers to represent the function's formal arguments. Finally, open and close brackets are used to contain within optional variable declarations and one or more required statements.

LEARN uses applicative order evaluation. That is, a function's arguments are evaluated before they are called. Type checking is only performed when the function's formal arguments are used within the function's body. The function may also return one or more data types. As a result, a function's return type and the types of its formals are evaluated at run-time by the bytecode executor.

Consider the following function:

```
function foo(a, b){
    if(a>0) {
        return b+1;
    }
    else {
        return (b or true);
    }
}
```

The function receives two arguments and returns a value. The type of variable b and the type of the function's return value depend on a. If a is an integer greater than zero, then b is required to be an integer and the function will return an integer. However, if a is an integer less than or equal to zero, then b is required to be a boolean and the function will return a boolean.

3.5.3 Question

`question` ***identifier*** {***var-decl-list***$_{opt}$ ***stmt-list***}

Questions must be declared after function declarations, if any. Declaration of questions must begin with the `question` keyword followed by an identifier followed by variable declarations and statements within braces. The prompt and answer variables must be defined as part of the statements. Not doing so, however will not result in an error. The choice variable may optionally be defined. Other statements may optionally precede these keyword declarations.

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

LEARN to provide the user with a set of choices. The answer variable must be defined with an array of string and will be used by LEARN to check if the user's provided answer or answers are correct. 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 variable, as described in section 3.4.12, will be set to true. Otherwise, the value of the correct variable will be set to false.

3.5.4 Execution

`execute` {***var-decl-list*$_{opt}$ *stmt-list***}

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 an optional variable declaration and list of statements within braces. The list of statements must consist of question statements as described in section 3.4.12. Not doing so, however, will not result in an error but the program will terminate without providing the user with questions.

3.6 Statements

3.6.1 Expressions

***expr* ;**

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

3.6.2 Compound

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

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.

3.6.3 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(`***expr***`)`

stmt

`else if(`***expr***`)`

stmt

`else`

stmt

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.

3.6.4 Loops

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

`repeat(`***expr*** `;` ***expr***$_{opt}$`)`

stmt

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.

3.6.5 Return

`return` ***expr*** `;`

The return statement may only be used within a function block to terminate the function's execution and return control to the caller with a return value. A function may contain any number of return statements.

3.7 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 }).

3.8 LEARN Program

var-decl-list*$_{opt}$ *function-decl-list*$_{opt}$ *question-decl-list execute-decl

A LEARN program consists of global variable declarations, functions, questions, and an execute statement in that specific order. Global variable and function declarations are optional. Questions and the execute declaration are not optional.

3.9 Built-in Variables

LEARN defines the built-in variables described in this section.

3.9.1 Global Variables

The variables described in this sub-section are declared globally.

3.9.1.1 correct

The *correct* variable is of type integer. It is initialized to zero and is automatically incremented by one every time the user provides a correct answer to a question.

3.9.1.2 askCount

The *askCount* variable is of type integer. It is initialized to zero and is automatically incremented by one every time the LEARN program asks the user a question.

3.9.1.3 correctCount

The *correctCount* variable is of type integer. It is initialized to zero and is automatically incremented by one every time the user provides a correct answer to a question.

3.9.2 Local Variables

The variables described in this sub-section are declared locally within each question block.

3.9.2.1 prompt

The *prompt* variable is of type string. It is initialized to an empty string (ie. “”) and is used by LEARN to provide a question prompt to the user.

3.9.2.2 choice

The *choice* variable is of type array of string. It is initialized to an array of one element containing an empty string (ie. {“”}) and is used by LEARN to provide a user with a list of possible answers to a question.

3.9.2.3 answer

The *answer* variable is of type array of string. It is initialized to an array of one element containing an empty string (ie. {“”}) and is used by LEARN to grade a user's response to the question. If the user's response is equal to one of the elements in the *answer* array, then the answer is evaluated to be correct.

3.10 Example

3.10.1 Program

```
[* Create multiple-choice question with one correct answer *]
question q1
{
    prompt = "Convert 2/4 to decimal";
    choice = {"0.25", "0.50", "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 = 0;
    var c = 2;
    prompt = "Convert 0.50 to fraction.";
    choice = {a[b], "1/3" , "2/3", a[c]};
    answer = {a[b] , a[c]};
}

[* Create free-response question *]
question q3
{
    var a = 2;
    var b = 4;
    prompt = "Convert " ^ a ^ "/" ^ b ^ " to integer. Round to lowest integer.";
    answer = {(a/b) ^ ""};
}

[* Create flow of questions *]
execute
{
    -> q1;
    if(correct) {
        -> q2;
    }
    else {
        -> q3;
    }
}
```

3.10.2 Output

```
Convert 2/4 to decimal
Enter one of possible choices below:
0.25
```

```
0.50
0.75
1.00
> 0.50
Correct
Convert 0.50 to fraction.
Enter one of possible choices below:
1/2
1/3
2/3
4/8
> 1/2
Correct
Good bye!
2 out of 2 answered correctly.
```

3.11 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:

var-decl-list*$_{opt}$ *function-decl-list*$_{opt}$ *question-decl-list execute-decl

var-decl-list:

var-decl `;`

var-decl-list var-decl `;`

var-decl:

`var` ***identifier*** `=` ***expr***

`var[`***expr***`]` ***identifier*** `=` ***expr***

function-decl-list:

function-decl-list function-decl

function-decl:

`function (` ***id-list*$_{opt}$** `) {` ***var-decl-list*$_{opt}$ *stmt-list*** `}`

question-decl-list:

question-decl

question-decl-list question-decl

question-decl:

`question` ***identifier*** `{` ***var-decl-list***$_{opt}$ ***stmt-list*** `}`

execute-decl:

`execute` `{` ***var-decl-list***$_{opt}$ ***stmt-list*** `}`

stmt-list:

stmt

stmt-list stmt

stmt:

expr `;`

`{` ***stmt-list***$_{opt}$ `}`

conditional-stmt

loop-stmt

`return` ***expr*** `;`

conditional-stmt:

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

if-stmt:

`if` `(` ***expr*** `)` ***stmt***

else-if-stmt-list:

else-if-stmt

else-if-stmt-list else-if-stmt

else-if-stmt:

`else if (`***expr***`)` ***stmt***

else-stmt:

`else (`***expr***`)` ***stmt***

loop-stmt:

`repeat (`***expr*** `;` ***expr***$_{opt}$`)` ***stmt***

block-expr:

`[`***expr***`]`

identifier-list:

identifier

identifier-list `,` ***identifier***

expr-list:

expr

expr-list `,` ***expr***

expr:

question-transition-expr

question-transition-expr:

assignment-expr

`->` ***identifier***

`->` `{` ***identifier-list*** `}`

assignment-expr:

or-expr

identifier block-expr$_{opt}$ **=** ***assignment-expr***

var-assign-expr:

identifier = expr

array-assign-expr:

identifier block-expr = expr

or-expr:

and-expr

or-expr `or` ***and-expr***

and-expr:

equality-expr

and-expr `and` ***equality-expr***

equality-expr:

relational-expr

equality-expr == relational-expr

equality-expr `!=` ***relational-expr***

relational-expr:

string-concat-expr

relational-expr > string-concat-expr

relational-expr >= string-concat-expr

relational-expr < string-concat-expr

relational-expr <= string-concat-expr

string-concat-expr:

additive-expr

string-concat-expr ^ additive-expr

additive-expr:

multiplicative-expr

additive-expr* + *multiplicative-expr

additive-expr* – *multiplicative-expr

multiplicative-expr:

unary-expr

multiplicative-expr* * *unary-expr

multiplicative-expr* / *unary-expr

multiplicative-expr* % *unary-expr

unary-expr:

function-call-expr

\- ***unary-expr***

! ***unary-expr***

function-call-expr:

array-reference-expr

identifier (***expr-list***$_{opt}$)

array-reference-expr:

variable-expr

identifier block-expr

variable-expr:

primary-expr

identifier

primary-expr:

integer-literal

string-literal

boolean-literal

identifier

(***expr***)

{***expr-list***}

4 Project Plan

4.1 Roles

The LEARN programming language was designed,implemented, and tested by Jairo Pava.

4.2 Development Environment

The LEARN programming language was developed and tested on the Ubuntu 12.04 LTS operating system. The Eclipse Integrated Development Environment with the OcaIDE 1.2.17 plugin was used to write Objective Caml code. Version 3.12.1 of Objective Caml was used to compile and run all code written.

4.3 Time-line

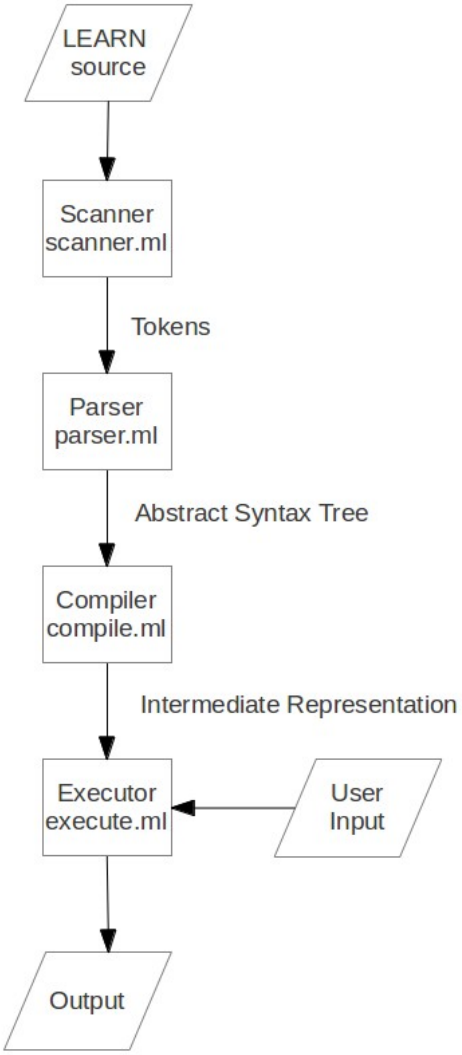
The following table describes the time-line followed during the entire project.

Task	Start Date	End Date
Development Environment Setup	05-30-2013	06-02-2013
Project Proposal Draft	06-03-2013	06-07-2013
Project Proposal Final Revision	06-13-2013	06-16-2013
First version of Scanner	06-17-2013	06-20-2013
First version of Parser	06-20-2013	06-23-2013
Language Reference Manual Draft	06-23-2013	06-28-2013
Language Reference Manual Final	07-07-2013	07-30-2013
Scanner	07-10-2013	07-11-2013
Parser	07-12-2013	07-15-2013
Bytecode defined	07-15-2013	07-20-2013
Compiler	07-20-2013	07-30-2013
Bytecode executor	07-30-2013	08-05-2013
Regression testing suite	08-05-2013	08-08-2013

Task	Start Date	End Date
Code Documentation	08-08-2013	08-12-2013
Refactoring	08-08-2013	08-12-2013
Final Report	08-12-2013	08-16-2013

5 Architectural Design

The figure below illustrates the architectural design of the LEARN project.



The LEARN source code, as described in the Language Reference Manual section of this document, is input to the scanner. The scanner, which is implemented using ocamllex reads the LEARN source code and produces a list of tokens based on a set of pre-defined syntax. If the source does not adhere to the syntax, the process terminates and an error is returned with the line number of where the error occurred. Otherwise, the list of tokens are passed to the parser.

The parser is implemented using ocamlyacc and reads the list of tokens to create an abstract syntax tree (AST). The AST is a tree representation of the abstract syntactic structure of the LEARN source code.

The AST is then walked by the compiler which performs type checking and raises failures when the types are not as expected. The compiler then generates an intermediate representation (IR) of the LEARN source code. This IR is a series of bytecode commands which are interpreted by the executor. The bytecode executor also performs type checking for function return types and function formal arguments since their types depends on the circumstances in which functions are called. In addition to the bytecode, the executor receives user input. User input is provided when the user is answering a question. The executor provides output in the form of question prompts, optional multiple choice answers, and answer feedback.

6 Test Plan

6.1 Goals

Testing throughout the project consisted of unit testing and automated regression testing. The goal of unit testing was to test individual units of each component as it was developed. The goal of automated regression testing was to end-to-end test each of the language's features as described in the Language Reference Manual.

6.2 Unit Testing

Unit testing consisted of loading Ocaml code into the Ocaml Toplevel, driving the code by invoking functions and providing stubs where necessary, and manually verifying the correctness of the output. For example, the compiler was tested by calling its translate function with multiple variations of an AST. This was typically done after a pattern matching case was introduced under the expr and stmt functions. Likewise, the bytecode executor was tested by providing its execute_prog function with multiple variations of bytecode and observing the contents of the stack and the output of the program.

Although this style of unit testing is less formal than traditional approaches, it worked well because it was fast, allowed early identification of subtle defects, and provided enough confidence to move onto development of other units.

6.3 Automated Regression Testing

Automated regression testing consisted of using a shell script to run a LEARN program, provide it with input from a file and compare its output to another file which contains the expected output. Any differences between the output and the expected output will result in an error message and a series of log files that may be used to identify the differences. A test was created in this manner for every

LEARN feature described in the Language Reference Manual. Some LEARN programs may contain more than one test case, but all test cases within a LEARN program consist of test variations of the same feature to keep the tests focused and make defect identification easier. The collection of these tests, or test suite, was run every time code was updated in any of the LEARN components. A successful run of the test suite would provide sufficient confidence that new code did not introduce defects.

7 Lessons Learned

This project was very interesting and fun for me because I was able to interactively learn about basic compilers through a project that at first seemed overwhelming but in the end became a satisfying effort. Additionally, I learned lessons that are applicable to other projects I will work on in the future and may be useful to students who take the PLT course. These lessons are listed below in no particular order.

7.1 Test early and often

I saved myself a couple of headaches by testing using the Ocaml Toplevel as soon as I would write a new unit of code. Many subtle defects were caught this way. Although testing often can be time consuming, I am certain I would have spent a lot more time and become more frustrated if I would have had to debug after the entire project had been written.

7.2 Start early

Start writing code early. It is easier to make design-level changes after the coding effort has begun if there is more time to work on the project. All other courses get busier towards the end of the semester anyway, and it will be nice to have one less thing to worry about.

7.3 Address a meaningful problem

This project may take a significant amount of time to complete. It will be less frustrating and even more exciting to focus the project on a programming language that addresses a personally meaningful problem. The more enjoyable the project, the more time that will be easily put into it.

7.4 Keep it simple

Begin the project by implementing simple features first. Over time it will be easier to think about how to implement more complicated features of the language and how to keep let them feel like natural components of the language.

8 Complete Listing

8.1 Scanner

scanner.mll

```
{ open Parser
   open Buffer
   open Lexing
}

rule token = parse
  [' ' '\t' '\r'] { token lexbuf } (* Whitespace *)
| ['\n']   { new_line lexbuf ; token lexbuf }
| "[*"     { comment lexbuf }          (* Comments *)
| '"'      { let buffer = Buffer.create 16 in
       STRING (stringr buffer lexbuf)} (* Quoted string *)
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| '['      { LBLOCK }
| ']'      { RBLOCK }
| ';'      { SEMI }
| ','      { COMMA }
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '%'    { MOD }
| '='      { ASSIGN }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
```

```
| '!'     { NEG }
| '^'     { CONCAT }
| "->"      { TRAN }
| "if"     { IF }
| "else if"{ ELSEIF }
| "else"   { ELSE }
| "repeat" { REPEAT }
| "var"     { VAR }
| "true"|"false" as lxm { BOOLEAN(bool_of_string lxm) }
| "and"     { AND }
| "or"   { OR }
| "question" { QUE }
| "execute"  { EXC }
| "function" { FUNC }
| "return"   { RET }
| ['0'-'9']+ as lxm { LITERAL(int_of_string lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

and comment = parse
  "*]" { token lexbuf }
| _    { comment lexbuf }

and stringr buffer = parse
  '"'       { Buffer.contents buffer }
| eof       { raise (Failure("end of file. end of string expected.")) }
| _ as char { Buffer.add_char buffer char; stringr buffer lexbuf }
```

8.2 Parser

parser.mly

```
%{  open Ast
      open Array
      open Lexing
      exception ParseFailure of string
      (* Obtain line number where parsing error occurred *)
```

```
      let string_of_position s =
            let line_str = string_of_int (s.pos_lnum) in
               "line:" ^ line_str
%}

%token SEMI LPAREN RPAREN LBRACE RBRACE
%token LBLOCK RBLOCK COMMA
%token PLUS MINUS TIMES DIVIDE MOD ASSIGN
%token EQ NEQ LT LEQ GT GEQ
%token NEG CONCAT TRAN VAR
%token IF ELSE ELSEIF REPEAT INT
%token AND OR QUE EXC FUNC RET
%token <int> LITERAL
%token <string> ID
%token <string> STRING
%token <bool> BOOLEAN
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%nonassoc ELSEIF
%right ASSIGN TRAN
%left OR AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left CONCAT
%left PLUS MINUS
%left TIMES DIVIDE MOD
%right NEG

%start learn_program
%type <Ast.learn_program> learn_program

%%

learn_program:
    globals_opt function_decl_list_opt question_decl_list execute_decl { $1, $2, $3, $4 }
 | error { raise (ParseFailure(string_of_position (symbol_start_pos ()))) }
```

```
globals_opt:
      var_decl_list_opt
         { { name = "";
            formals = [];
            decls = List.rev $1;
            body = [];
            t = Global } }

var_decl_list_opt:
   /* nothing */ { [] }
 |  var_decl_list_opt var_decl SEMI { $2 :: $1 }

var_decl:
   VAR ID ASSIGN expr { VarDecl($2, $4) }
 |  VAR block_expr ID ASSIGN expr { ArrDecl($2, $3, $5) }

function_decl_list_opt:
  /* nothing */ { [] }
 | function_decl_list_opt function_decl { $2 :: $1 }

function_decl:
  FUNC ID LPAREN id_list_opt RPAREN LBRACE var_decl_list_opt stmt_list RBRACE
     { {  name = $2;
          formals = List.rev $4;
          decls = List.rev $7;
          body = List.rev $8;
          t = Function } }

question_decl_list:
   question_decl { $1 :: [] }
 |  question_decl_list question_decl { $2 :: $1 }

question_decl:
   QUE ID LBRACE var_decl_list_opt stmt_list RBRACE
      { {    name = $2;
         formals = [];
         decls = List.rev $4;
```

```
        body = List.rev $5;
        t = Question } }

execute_decl:
   EXC LBRACE var_decl_list_opt stmt_list RBRACE
      { { name = "";
        formals = [];
        decls = List.rev $3;
        body = List.rev $4;
        t = Execute } }

stmt_list_opt:
   /* nothing */ { [] }
 | stmt_list { $1 }

stmt_list:
   stmt { $1 :: [] }
 | stmt_list stmt { $2 :: $1 }

elif_list:
     ELSEIF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
   | ELSEIF LPAREN expr RPAREN stmt ELSE stmt { If($3, $5, $7) }
   | ELSEIF LPAREN expr RPAREN stmt elif_list { If($3, $5, Block([$6])) }

stmt:
   expr SEMI { Expr($1) }
 | LBRACE stmt_list_opt RBRACE { Block(List.rev $2) }
 | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
 | IF LPAREN expr RPAREN stmt ELSE stmt { If($3, $5, $7) }
 | IF LPAREN expr RPAREN stmt elif_list { If($3, $5, $6) }
 | REPEAT LPAREN expr SEMI expr_opt RPAREN stmt { Repeat($3, $5, $7) }
 | RET expr SEMI { Return($2) }

block_expr_opt:
   /* nothing */ { Noexpr }
 | block_expr    { $1 }

block_expr:
```

```
   LBLOCK expr RBLOCK { $2 }

expr_opt:
   /* nothing */ { Noexpr }
 | expr        { $1 }

expr_list:
    expr { $1 :: [] }
 | expr_list COMMA expr { $3 :: $1 }

expr_list_opt:
    /* nothing */ { [] }
 | expr_list { $1 }

id_list:
    ID { $1 :: [] }
 | id_list COMMA ID { $3 :: $1 }

id_list_opt:
  /* nothing */ { [] }
 | id_list { $1 }

expr:
   LITERAL        { Literal($1) }
 | STRING         { SLiteral($1) }
 | BOOLEAN        { BLiteral($1) }
 | LBRACE expr_list RBRACE { ALiteral(List.rev $2) }
 | ID             { Id($1) }
 | ID block_expr    { ArrId($1, $2) }
 | MINUS  expr      { Binop(Literal(-1), Mult,  $2) }
 | NEG    expr      { Binop(Literal(-1), Neg,   $2) }
 | expr PLUS   expr { Binop($1, Add,   $3) }
 | expr MINUS  expr { Binop($1, Sub,   $3) }
 | expr TIMES  expr { Binop($1, Mult,  $3) }
 | expr DIVIDE expr { Binop($1, Div,   $3) }
 | expr MOD    expr { Binop($1, Mod,   $3) }
 | expr CONCAT expr { Binop($1, Concat, $3) }
 | expr EQ     expr { Binop($1, Equal, $3) }
```

```
 | expr NEQ    expr { Binop($1, Neq,   $3) }
 | expr LT     expr { Binop($1, Less,  $3) }
 | expr LEQ    expr { Binop($1, Leq,   $3) }
 | expr GT     expr { Binop($1, Greater, $3) }
 | expr GEQ    expr { Binop($1, Geq,   $3) }
 | expr OR     expr { Binop($1, Or,    $3) }
 | expr AND    expr { Binop($1, And,   $3) }
 | ID block_expr_opt ASSIGN expr { Assign($1, $2, $4) }
 | TRAN ID         { Que($2 :: []) }
 | TRAN LBRACE id_list RBRACE      { Que($3) }
 | ID LPAREN expr_list_opt RPAREN { Call($1, List.rev $3) }
 | LPAREN expr RPAREN { $2 }
```

8.3 Abstract Syntax Tree

ast.ml

```
type op =
   Add    | Sub | Mult    | Div | Equal | Neq
  | Less   | Leq | Greater | Geq | And   | Or
  | Concat | Mod | Neg

type expr =
   Literal of int
  | SLiteral of string
  | BLiteral of bool
  | ALiteral of expr list
  | ALitRpt of expr * expr
  | Id of string
  | ArrId of string * expr
  | Binop of expr * op * expr
  | Assign of string * expr * expr
  | Que of string list
  | Call of string * expr list
  | Noexpr

type var_decl =
   VarDecl of string * expr
```

```
  | ArrDecl of expr * string * expr

type stmt =
    Block of stmt list
  | Expr of expr
  | If of expr * stmt * stmt
  | Repeat of expr * expr * stmt
  | Decls of var_decl list
  | Return of expr

type t =
   Void
  | Int
  | Array of t
  | Boolean
  | String
  | Undefined

type named_block_type =
  | Global
  | Question
  | Execute
  | Function

type named_block = {
    name : string;
    formals : string list;
    decls : var_decl list;
    body : stmt list;
    t: named_block_type;
  }

type learn_program = named_block * named_block list * named_block list * named_block

(* string of type *)
let rec string_of_type t = match t with
  | Void -> "Void"
  | Int -> "Integer"
```

```
  | Array(t') -> "Array of " ^ string_of_type t'
  | Boolean -> "Boolean"
  | String -> "String"
  | Undefined -> "Undefined"

let rec string_of_expr = function
    Literal(l) -> string_of_int l
  | SLiteral(s) -> "\"" ^ s ^ "\""
  | BLiteral(b) -> string_of_bool b
  | ALiteral(a) ->
      "[" ^
      String.concat "," (List.map (fun e -> string_of_expr e) a)
      ^ "]"
  | ALitRpt(e1, e2) -> ""
  | Id(s) -> s
  | ArrId(s, e) -> s ^ "[" ^ string_of_expr e ^ "]"
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^
      (match o with
        Add -> "+"     | Sub -> "-"  | Mult -> "*" | Div -> "/"
      | Equal -> "==" | Neq -> "!=" | Less -> "<" | Leq -> "<="
      | Greater -> ">" | Geq -> ">=" | Neg -> "!"  | Mod -> "%"
      | Concat -> "^"  | Or -> "or"  | And -> "and") ^ " " ^
      string_of_expr e2
  | Assign(v, e1, e2) -> v ^ string_of_expr_opt "[" "]" e1 ^ "=" ^ string_of_expr e2
    | Que(qlist) ->
          "-> {" ^
          String.concat "," (List.map (fun id -> id) qlist)
          ^ "}"
    | Call(id, el) ->
          id ^ "(" ^
          String.concat "," (List.map string_of_expr el)
          ^ ")"
  | Noexpr -> ""

and string_of_expr_opt l r e = match e with
    Noexpr -> string_of_expr e
  | _ -> l ^ string_of_expr e ^ r
```

```
let rec string_of_stmt = function
    Block(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
  | Expr(expr) -> string_of_expr expr ^ ";\n";
  | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
     string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | Repeat(e1, e2, s1) ->
       "repeat (" ^ string_of_expr e1 ^ ";" ^ string_of_expr e2 ^ ")" ^ string_of_stmt s1
  | Decls dl ->
         let string_of_vdecl = function
            VarDecl(id, expr) -> "var " ^ id ^ " = " ^ string_of_expr expr ^ ";\n"
          | ArrDecl(expr1, id, expr2) ->
                "var [" ^ string_of_expr expr1 ^ "] " ^ id ^ " = " ^ string_of_expr expr2 ^
";\n" in
            String.concat "" (List.map string_of_vdecl dl)
  | Return expr -> "return " ^ string_of_expr expr ^ ";\n"

let string_of_named_block_type = function
  | Global -> ""
  | Question -> "question"
  | Execute -> "execute"
  | Function -> "function"

let string_of_named_block nb =
    string_of_named_block_type nb.t ^ " " ^ nb.name ^
    (match nb.t with
  | Global ->
            string_of_stmt (Decls(nb.decls))
  | Question | Execute ->
            "\n{\n" ^
            string_of_stmt (Decls(nb.decls)) ^
            String.concat "" (List.map string_of_stmt nb.body) ^
            "}\n"
  | Function ->
            "(" ^ String.concat "," nb.formals ^ ")" ^
            "\n{\n" ^
```

```
            string_of_stmt (Decls(nb.decls)) ^
            String.concat "" (List.map string_of_stmt nb.body) ^
            "}\n")

let string_of_learn_program (vars, func, ques, exec) =
    string_of_named_block vars ^ "\n" ^
    String.concat "" (List.map string_of_named_block func) ^
    String.concat "" (List.map string_of_named_block ques) ^
    string_of_named_block exec ^ "\n"
```

8.4 Bytecode

bytecode.ml

```
type bstmt =
    Lit of int          (* Push an integer literal *)
  | Slit of string      (* Push a string literal *)
  | Blit of bool        (* Push a boolean literal *)
  | Alit of int         (* Push an array literal. This will indicate size of array *)
  | Drp                 (* Discard a value *)
  | Arpt                (* Repeat *)
  | Bin of Ast.op       (* Perform arithmetic on top of stack *)
  | Loda of int         (* Fetch global array variable using index on top of stack *)
  | Lod of int          (* Fetch global variable *)
  | Str of int          (* Store global variable *)
  | Stra of int         (* Store global array variable using index on top of stack *)
  | Lfp of int          (* Load frame pointer relative *)
  | Lfpa of int         (* Load frame pointer relative with array index *)
  | Sfp of int          (* Store frame pointer relative *)
  | Sfpa of int         (* Store frame pointer relative with array index *)
  | Jsr of int          (* Call function by absolute address *)
  | Ent of int          (* Push FP, FP -> SP, SP += i *)
  | Rts of int          (* Restore FP, SP, consume formals, push result *)
  | Beq of int          (* Branch relative if top-of-stack is false *)
  | Bne of int          (* Branch relative if top-of-stack is true *)
  | Bra of int          (* Branch relative *)
  | Hlt                 (* Terminate *)
```

```
type prog = {
    num_globals : int;   (* Number of global variables *)
    text : bstmt array; (* Code for all the program *)
  }

let string_of_stmt = function
    Lit(i) -> "Lit " ^ string_of_int i
  | Slit(s) -> "Slit \"" ^ s ^ "\""
  | Blit(b) -> "Blit " ^ string_of_bool b
  | Alit(l) -> "Alit " ^ string_of_int l
  | Drp -> "Drp"
  | Arpt -> "Arpt"
  | Bin(Ast.Add) -> "Add"
  | Bin(Ast.Sub) -> "Sub"
  | Bin(Ast.Mult) -> "Mul"
  | Bin(Ast.Div) -> "Div"
  | Bin(Ast.Equal) -> "Eql"
  | Bin(Ast.Neq) -> "Neq"
  | Bin(Ast.Less) -> "Lt"
  | Bin(Ast.Leq) -> "Leq"
  | Bin(Ast.Greater) -> "Gt"
  | Bin(Ast.Geq) -> "Geq"
  | Bin(Ast.Neg) -> "Neg"
  | Bin(Ast.Mod) -> "Mod"
  | Bin(Ast.Concat) -> "Concat"
  | Bin(Ast.Or) -> "Or"
  | Bin(Ast.And) -> "And"
  | Lod(i) -> "Lod " ^ string_of_int i
  | Loda(i) -> "Loda " ^ string_of_int i
  | Str(i) -> "Str " ^ string_of_int i
  | Stra(i) -> "Stra " ^ string_of_int i
  | Lfp(i) -> "Lfp " ^ string_of_int i
  | Lfpa(i) -> "Lfpa " ^ string_of_int i
  | Sfp(i) -> "Sfp " ^ string_of_int i
  | Sfpa(i) -> "Sfpa " ^ string_of_int i
  | Jsr(i) -> "Jsr " ^ string_of_int i
  | Ent(i) -> "Ent " ^ string_of_int i
  | Rts(i) -> "Rts " ^ string_of_int i
```

```
  | Bne(i) -> "Bne " ^ string_of_int i
  | Beq(i) -> "Beq " ^ string_of_int i
  | Bra(i) -> "Bra " ^ string_of_int i
  | Hlt    -> "Hlt"

let string_of_prog p =
  string_of_int p.num_globals ^ " global variables\n" ^
  let funca = Array.mapi
      (fun i s -> string_of_int i ^ " " ^ string_of_stmt s) p.text
  in String.concat "\n" (Array.to_list funca)
```

8.5 Compiler

compile.ml

```
open Ast
open Bytecode

module StringMap = Map.Make(String)

(* Custom compilation exception *)
exception CompilationFailure of string

(* Symbol table: Information about all the names in scope *)
type env = {
    question_index : int StringMap.t; (* Index for each function *)
    mutable global_index   : (int * t) StringMap.t; (* "Address" for global variables *)
    function_index : (int * int * t) StringMap.t; (* "Address" for functions *)
    mutable local_index    : (int * t) StringMap.t; (* FP offset for args, locals *)
  }

type scope =
    | ScGlobal
    | ScLocal

(* Find variable and return 3-tuple with index, type, and scope where it was found *)
let find_variable n env =
    let ((i, t), s) = try
```

```
      (StringMap.find n env.local_index, ScLocal)
      with Not_found -> try
         (StringMap.find n env.global_index, ScGlobal)
         with Not_found -> raise (Failure ("undeclared variable " ^ n)) in
      (i, t, s)

(* Find question and return its index *)
let find_question n env =
   let i = try
      (StringMap.find n env.question_index)
      with Not_found -> raise (Failure ("undeclared question " ^ n)) in i

(* Find function and return its index *)
let find_function n env =
   let i = try
      (StringMap.find n env.function_index)
      with Not_found -> raise (Failure ("undeclared function " ^ n)) in i

(* Declare a variable type *)
let decl_variable_type s t env =
   let (i, st, sc) = find_variable s env in
      (match st with
         | Undefined ->
            (match sc with
               | ScLocal ->
                  env.local_index <- StringMap.add s (i, t) env.local_index
               | ScGlobal ->
                  env.global_index <- StringMap.add s (i, t) env.global_index)
         | _ ->
            raise (Failure ("cannot re-declare \"" ^ s ^ "\"")))

(* Create list of 3-tuple with address, identifier, and type *)
let rec enum stride n = function
   | [] -> []
   | hd::tl -> match hd with
                     | VarDecl(i,e) -> (n, i, Undefined) :: enum stride (n+stride) tl
                     | ArrDecl(e1, i, e2) -> (n, i, Undefined) :: enum stride (n+stride) tl
```

```
(* Create list of 2-tuple with address and identifier *)
let rec qenum stride n = function
    | [] -> []
    | hd::tl -> (n, hd) :: qenum stride (n+stride) tl

(* Create list of 4-tuple with address, identifier, number of formals, and type *)
let rec fenum stride n = function
    | [] -> []
    | (id, cnt)::tl -> (n, id, cnt, Undefined) :: fenum stride (n+stride) tl

(* Create a map where key is the identifier and value is address *)
let string_map_questions map pairs =
  List.fold_left (fun m (i, n) -> if StringMap.mem n m then
                    raise (Failure ("question already declared: " ^ n)) else
                    StringMap.add n i m) map pairs

(* Create a map where key is the identifier and value is a tuple of address and type *)
let string_map_variables map list =
  List.fold_left (fun m (i, n, t) -> if StringMap.mem n m then
                 raise (Failure ("variable already declared: " ^ n)) else
                 StringMap.add n (i, t) m) map list

(* Create a map where key is the identifier and value is a 3-tuple of addres, type, and number
of formals *)
let string_map_functions map list =
    List.fold_left (fun m (i, n, c, t) -> if StringMap.mem n m then
                    raise (Failure ("function already declared: " ^ n)) else
                    StringMap.add n (i, c, t) m) map list

(* Raise failure if types are not equal. *)
(* If type is undefined, type checking will be performed at run time. *)
let require_type t t' =
    if t <> Undefined && t' <> Undefined then
        if t <> t' then
            raise (Failure ("expected " ^ string_of_type t ^ " but received " ^ string_of_type t'))
        else
            ()
    else
```

```
    ()

(* Match argument with a comparable type, otherwise raise a failure. *)
(* If type is undefined, type checking will be performed at run time *)
let require_comparable c = match c with
   | Int | Boolean | String | Undefined -> ()
   | _ -> raise (Failure ("expected Int, Boolean, or String but received " ^ string_of_type c))

(* match argument with array and return array type, otherwise raise a failure *)
(* If type is undefined, type checking will be performed at run time *)
let get_array_type a = match a with
   | Array(t) -> t
   | Undefined -> Undefined
   | _ -> raise (Failure ("expected array but received " ^ string_of_type a))

(** Translate a program in AST form into a bytecode program.  Throw an
   exception if something is wrong, e.g., a reference to an unknown
   variable or question *)
let translate (globals, functions, questions, execute) =
   (* Allocate addresses for global variables *)
   let builtin_globals = [VarDecl("correct", BLiteral(false));
                          VarDecl("askCount", Literal(0));
                          VarDecl("correctCount", Literal(0))] in
   let globals = {globals with decls = builtin_globals @ globals.decls} in
   let global_index = string_map_variables StringMap.empty (enum 1 0 globals.decls) in
   let global_var_count = List.length globals.decls in

   (* Allocate address for functions *)
   let function_list = fenum 1 0 (List.map (fun f -> (f.name, List.length f.formals)) functions) in
   let function_index = string_map_functions StringMap.empty function_list in

   (* Allocate addresses for questions *)
   let question_index = string_map_questions StringMap.empty
      (qenum 1 (List.length function_list) (List.map (fun q -> q.name) questions)) in

   let translate env decl =
      (* Bookkeeping: FP offsets for locals *)
      let num_locals = List.length decl.decls
```

```
      and num_formals = List.length decl.formals
      and local_offsets = (enum 1 1 decl.decls)
      and formal_offsets = (enum (-1) (-2) (List.map (fun f -> VarDecl(f, Noexpr)) decl.formals)
) in
      env.local_index <- string_map_variables StringMap.empty (formal_offsets @
local_offsets);


      let rec expr = function
         | Literal i -> ([Lit i], Int)
         | SLiteral s -> ([Slit s], String)
         | BLiteral b -> ([Blit b], Boolean)
         | ALiteral e ->
               (match e with
                  | [] ->  raise (Failure ("array may not be empty"))
                  | hd :: tl ->
                        let rec evlarray ls t = function
                           | [] -> List.rev ls
                           | hd :: tl ->
                                 let (hd', hdt') = expr hd in
                                    if hdt' <> t then
                                       raise (Failure ("array may only contain elements of type "
^ string_of_type t))
                                    else
                                       evlarray ((List.rev hd') @ ls) t tl in

                        let (hd', hdt') = expr hd in
                           (hd' @ (evlarray [] hdt' tl) @ [ALit(List.length e)], Array(hdt')))
         | ALitRpt(e1, e2) ->
               let (e1', t1) = expr e1 in
                  require_type Int t1;
                  let (e2', t2) = expr e2 in
                     (e2' @ e1' @ [Arpt], Array(t2))
         | Id s ->
               let (i, t, sc) = find_variable s env in
                  (match sc with
                     | ScLocal  -> ([Lfp i], t)
                     | ScGlobal -> ([Lod i], t))
         | ArrId(s, e) ->
               let (e', et') = expr e in
```

```
                require_type Int et';
                let (i, st, sc) = find_variable s env in
                   let st' = get_array_type st in
                      (match sc with
                         | ScLocal  -> (e' @ [Lfpa i], st')
                         | ScGlobal -> (e' @ [Loda i], st'))
        | Binop (e1, op, e2) ->
              let (e1', t1) = expr e1 and (e2', t2) = expr e2 in
                let rt = match op with
                              | Add | Sub | Mult | Div | Mod ->
                                    require_type Int t1; require_type Int t2; Int
                              | Less | Leq | Greater | Geq ->
                                    require_type Int t1; require_type Int t2; Boolean
                              | Equal | Neq ->
                                    require_type t1 t2; require_comparable t1;
require_comparable t2; Boolean
                              | And | Or ->
                                    require_type Boolean t1; require_type Boolean t2; Boolean
                              | Concat ->
                                    require_comparable t1; require_comparable t2; String
                              | Neg ->
                                    require_type Boolean t2; Boolean in
                (e1' @ e2' @ [Bin op], rt)
        | Assign (s, e1, e2) ->
              let (e2', t2) = expr e2 in
                let(i, st, sc) = find_variable s env in
                (e2' @ (match e1 with
                  | Noexpr ->
                        if st = t2 || st = Undefined || t2 = Undefined then
                           (match sc with
                              | ScLocal  -> [Sfp i]
                              | ScGlobal -> [Str i])
                        else
                           raise (Failure ("cannot assign type " ^ string_of_type t2 ^ " to " ^
s ^ " of type " ^ string_of_type st))
                  | _ ->
                        let (e1', t1) = expr e1 in
                           require_type Int t1;
```

```
                        let st' = get_array_type st in
                           if st' = t2 || st = Undefined || t2 = Undefined  then
                              e1' @
                              (match sc with
                                 | ScLocal  -> [Sfpa i]
                                 | ScGlobal -> [Stra i])
                           else
                              raise (Failure ("cannot assign type " ^ string_of_type t2 ^ "
to " ^ s ^ " of type " ^ string_of_type st'))
                     ), Void)
         | Que(qlist) ->
               (match decl.t with
                  | Execute ->
                        let rec que ls = function
                           | [] -> ls
                           | hd :: tl ->
                                 let i = find_question hd env in
                                    que (Jsr(i) :: ls) tl in
                        (que [] qlist, Void)
                  | _ ->
                        raise (Failure ("cannot define question transition outside of execute
block")))
         | Call(id, el) ->
               let elc = List.length el in
                  let (i, c, t) = find_function id env in
                     if c = elc then
                        let rec push_args ls = function
                           | [] -> ls
                           | hd::tl ->
                                 let (e, _) = expr hd in
                                    push_args (e @ ls) tl in
                        (push_args [] el @ [Jsr i ], Undefined)
                     else
                        raise( Failure("call to " ^ id ^ " provides " ^
                           string_of_int elc ^ " but " ^ string_of_int c ^ " were expected"))

         | Noexpr -> ([], Void)

      in let rec stmt ps = try (match ps with
```

```
    | Block sl ->  List.concat (List.map stmt sl)
    | Expr e -> fst (expr e) @ [Drp]
    | If (p, t, f) -> let t' = stmt t and f' = stmt f in
        let (p', pt) = expr p in require_type Boolean pt;
          p' @ [Beq(2 + List.length t')] @
          t' @ [Bra(1 + List.length f')] @ f'
    | Repeat (e1, e2, b) ->
       let b' = stmt b and (e1', t1) = expr e1 and (e2', _) = expr e2 in
            require_type Boolean(t1); let be2' = b' @ e2' in
            [Bra (1 + List.length be2')] @ be2' @ e1' @
            [Bne (-(List.length be2' + List.length e1'))]
    | Decls dl ->
          let decl = function
            | VarDecl(s, e) ->
                  let (_, t1) = expr e in
                     decl_variable_type s t1 env;
                     fst (expr (Assign(s, Noexpr, e)))
            | ArrDecl(e1, s, e2) ->
                     let(e2', t2) = expr e2 in
                        decl_variable_type s (Array(t2)) env;
                        fst (expr (Assign(s, Noexpr, ALitRpt(e1, e2)))) in
          List.concat (List.map decl dl)
    | Return e ->
          (match decl.t with
            | Function ->  fst (expr e) @ [Rts num_formals]
            | _ -> raise ( Failure("cannot call return outside of function"))))
with Failure(s) -> raise(CompilationFailure("\n" ^ s ^ "\nin: " ^ Ast.string_of_stmt ps))


in match decl.t with
  | Question | Execute | Function ->
          let sd = stmt (Decls decl.decls) in
            let bd = stmt (Block decl.body) in
               [Ent num_locals] @ sd @ bd @ (match decl.t with
                  | Question -> [Jsr(-1); Rts 0]
                  | Execute -> [Jsr(-2)]
                  | Function -> [Rts num_formals]
                  | _ -> [])
```

```
        | Global -> stmt (Block decl.body)

    in let env = { question_index = question_index;
                   global_index = global_index;
                   function_index = function_index;
                   local_index = StringMap.empty } in

    (* Code to start the program *)

    (* First, initialize global variables *)
    let globals = {globals with body = [Decls globals.decls]; decls = [];} in
    let global_decls = translate env globals in

    (* Second, compile the functions *)
    let function_bodies = List.map (fun f -> translate env f) functions in

    (* Third, execute the Execute block*)
    let execute_body = (translate env execute) @ [Hlt] in

    (* Fourth, compile the questions *)
    let builtin_question_locals = [VarDecl("prompt", SLiteral(""));
                                   VarDecl("choice", ALiteral([SLiteral("")]));
                                   VarDecl("answer", ALiteral([SLiteral("")]))] in
    let question_bodies = List.map (fun q ->
                                       let q = {q with decls =
builtin_question_locals @ q.decls} in
                                          translate env q) questions in

    (* Calculate question and function entry points by adding their lengths *)
    let offset = List.length global_decls + List.length execute_body in
    let (que_func_offset_list, _) = List.fold_left
       (fun (l,i) f -> (i :: l, (i + List.length f))) ([],0) (function_bodies @ question_bodies) in
    let que_func_offset = Array.of_list (List.rev que_func_offset_list) in

    { num_globals = global_var_count;
    (* Concatenate the compiled questions and replace the question
    indexes in Jsr statements with PC values *)
    text = Array.of_list (List.map (function
```

```
            Jsr i when i > (-1) -> Jsr (que_func_offset.(i) + offset)
    | _ as s -> s) (global_decls @ execute_body @ (List.concat function_bodies) @ (List.concat
question_bodies)))
  }
```

8.6 Executor

execute.ml

```
open Ast
open Bytecode

(* Custom execute exception *)
exception ExecuteFailure of string

(* Data types that may be stored in the stack and globals *)
type data =
  | Integer of int
  | Boolean of bool
  | String  of string
  | Array   of data array
  | Undefined

(* return integer from data, otherwise raise failure *)
let int_of_data = function
  | Integer i -> i
  | _ -> raise(Failure("expected int"))

(* return boolean from data, otherwise raise failure *)
let bool_of_data = function
  | Boolean b -> b
  | _ -> raise(Failure("expected boolean"))

(* return string from data, otherwise raise failure *)
let sstring_of_data = function
  | String s -> s
  | _ -> raise(Failure("expected string"))

(* return array from data, otherwise raise failure *)
```

```
let array_of_data = function
   | Array a -> a
   | _ -> raise(Failure("expected array"))

(* require t1 and t2 to be of same data type. otherwise, raise failure *)
let require_same_type t1 t2 =
   (match t1 with
      | Integer x -> (match t2 with Integer y -> () | _ -> raise(Failure("expected int")))
      | Boolean x -> (match t2 with Boolean y -> () | _ -> raise(Failure("expected boolean")))
      | String x  -> (match t2 with String y  -> () | _ -> raise(Failure("expected string")))
      | Array x   -> (match t2 with Array y  ->  () | _ -> raise(Failure("expected array")))
      | Undefined -> ())

(* Data type to string *)
let rec string_of_data = function
   | Integer i -> string_of_int i
   | Boolean b -> string_of_bool b
   | String s  -> s
   | Array a   -> "{" ^ String.concat ";" (Array.to_list (Array.map string_of_data a)) ^ "}"
   | Undefined -> ""

(* Stack layout just after "Ent":

          <-- SP
  Local n
  ...
  Local 0
  Saved FP   <-- FP
  Saved PC
  Arg 0
  ...
  Arg n *)

let execute_prog prog =
  let stack = Array.make 1024 (Undefined)
  and globals = Array.make prog.num_globals (Undefined) in

  (* Set all elements in stack from f to l to Undefined *)
```

```
let clear f l =
  Array.fill stack f l Undefined in

let rec exec fp sp pc = try (match prog.text.(pc) with
    | Lit i  -> stack.(sp) <- Integer i ; exec fp (sp+1) (pc+1)
    | Blit b -> stack.(sp) <- Boolean b ; exec fp (sp+1) (pc+1)
    | Slit s -> stack.(sp) <- String s ; exec fp (sp+1) (pc+1)
    | Alit i -> stack.(sp-i) <- Array(Array.sub stack (sp - i) i) ; exec fp (sp-i+1) (pc+1)
    | Arpt   -> let op1 = stack.(sp-2) and op2 = int_of_data (stack.(sp-1)) in
                          stack.(sp-2) <- Array(Array.make op2 op1) ; exec fp (sp-1) (pc+1)
    | Drp    -> exec fp (sp-1) (pc+1)
    | Bin op -> let op1 = stack.(sp-2) and op2 = stack.(sp-1) in
       stack.(sp-2) <- (match (op1, op, op2) with
      | (Integer i1,  Add,     Integer i2)    -> Integer(i1 + i2)
      | (Integer i1,  Sub,     Integer i2)    -> Integer(i1 - i2)
      | (Integer i1,  Mult,    Integer i2)    -> Integer(i1 * i2)
      | (Integer i1,  Div,     Integer i2)    -> Integer(i1 / i2)
      | (Integer i1,  Mod,     Integer i2)    -> Integer(i1 mod i2)
      | (Integer i1,  Equal,   Integer i2)    -> Boolean(i1 = i2)
      | (String i1,   Equal,   String i2)     -> Boolean(i1 = i2)
      | (Boolean i1,  Equal,   Boolean i2)    -> Boolean(i1 = i2)
      | (Integer i1,  Neq,     Integer i2)    -> Boolean(i1 <> i2)
      | (String i1,   Neq,     String i2)     -> Boolean(i1 <> i2)
      | (Boolean i1,  Neq,     Boolean i2)    -> Boolean(i1 <> i2)
      | (Integer i1,  Less,    Integer i2)    -> Boolean(i1 <  i2)
      | (Integer i1,  Leq,     Integer i2)    -> Boolean(i1 <= i2)
      | (Integer i1,  Greater, Integer i2)    -> Boolean(i1 >  i2)
      | (Integer i1,  Geq,     Integer i2)    -> Boolean(i1 >= i2)
      | (Boolean b1,  Or,      Boolean b2)    -> Boolean(b1 || b2)
      | (Boolean b1,  And,     Boolean b2)    -> Boolean(b1 && b2)
      | (Integer i1,  Neg,     Boolean b2)    -> Boolean(not b2)
      | (_,          Concat,  _)       -> String(string_of_data op1 ^ string_of_data op2)
      | _                              -> raise (Failure
                      ("unexpected binop " ^ string_of_data op1 ^
                      " " ^ string_of_stmt (Bin(op)) ^ " " ^ string_of_data op2)));
       exec fp (sp-1) (pc+1)
    | Lod i   -> stack.(sp)   <- globals.(i)  ; exec fp (sp+1) (pc+1)
    | Loda i  -> let ai = int_of_data (stack.(sp-1)) in
```

```
                        let a = array_of_data (globals.(i)) in
                          stack.(sp-1) <- a.(ai) ; exec fp (sp) (pc+1)
    | Str i   -> require_same_type (globals.(i)) (stack.(sp-1)) ;
                     globals.(i)  <- stack.(sp-1) ; exec fp sp     (pc+1)
    | Stra i  -> let ai = int_of_data (stack.(sp-1)) in
                        let a = array_of_data (globals.(i)) in
                          let e = stack.(sp-2) in
                            require_same_type (a.(ai)) (e) ;
                            a.(ai) <- e ; exec fp (sp-1) (pc+1)
    | Lfp i   -> stack.(sp)   <- stack.(fp+i) ; exec fp (sp+1) (pc+1)
    | Lfpa i  -> let ai = int_of_data (stack.(sp-1)) in
                        let a = array_of_data (stack.(fp+i)) in
                          stack.(sp-1) <- a.(ai) ; exec fp (sp) (pc+1)
    | Sfp i   -> require_same_type (stack.(fp+i)) (stack.(sp-1)) ;
                     stack.(fp+i) <- stack.(sp-1) ; exec fp sp     (pc+1)
    | Sfpa i  -> let ai = int_of_data (stack.(sp-1)) in
                        let a = array_of_data (stack.(fp+i)) in
                          let e = stack.(sp-2) in
                            require_same_type (a.(ai)) (e) ;
                            a.(ai) <- e ; exec fp (sp-1) (pc+1)
    | Jsr(-1) -> let prompt = string_of_data (stack.(fp+1)) in
                        print_string (prompt ^ "\n") ; let choices = array_of_data(stack.
(fp+2)) in
                          (match Array.length choices with
                            | 1 ->
                                  if choices.(0) = String("") then
                                    print_string("Enter answer below:\n")
                                  else
                                    print_string("Enter one of possible choices below:\n"
^ sstring_of_data choices.(0) ^ "\n")
                            | _ ->
                                  print_string("Enter one of possible choices below:\n" ^
                                    String.concat "\n" (Array.to_list (Array.map
string_of_data choices)) ^ "\n")) ;
                          let answers = Array.to_list(array_of_data(stack.(fp+3))) in
                            let rsp = String(read_line ()) in
                              let rec is_correct ans = function
                                | [] -> false
                                | hd::tl -> if hd=ans then true else is_correct ans tl in
```

```
                                    let correct_ans = is_correct rsp answers in
                                       if correct_ans then
                                          (globals.(0) <- Boolean(true) ;
                                          globals.(2) <- Integer(int_of_data(globals.(2)) + 1) ;
                                          print_string("Correct\n"))

                                       else
                                          (globals.(0) <- Boolean(false) ;
                                          print_string("Not correct\n")) ;
                                       globals.(1) <- Integer(int_of_data(globals.(1)) + 1) ;

                                 exec fp sp (pc+1)
     | Jsr(-2) -> let correctCount = int_of_data(globals.(2)) in
                                 let askCount = int_of_data(globals.(1)) in
                                    print_string("Good bye!\n" ^ string_of_int correctCount ^
                                    " out of " ^ string_of_int askCount ^ " answered correctly.\n") ;
                                 exec fp sp (pc+1)
     | Jsr i   -> stack.(sp)   <- Integer(pc + 1)       ; exec fp (sp+1) i
     | Ent i   -> stack.(sp)   <- Integer(fp) ; clear (sp+1) (sp+i) ; exec sp (sp+i+1) (pc+1)
     | Rts i   -> let new_fp = int_of_data (stack.(fp)) and new_pc = int_of_data (stack.(fp-1)) in
                                    stack.(fp-i-1) <- stack.(sp-1) ; exec new_fp (fp-i) new_pc
     | Beq i   -> exec fp (sp-1) (pc + if stack.(sp-1) =  Boolean(false) then i else 1)
     | Bne i   -> exec fp (sp-1) (pc + if stack.(sp-1) <> Boolean(false) then i else 1)
     | Bra i   -> exec fp sp (pc+i)
     | Hlt     -> ())
    with Failure(s) -> raise(ExecuteFailure("\n" ^ s ^ "\nin: " ^ Bytecode.string_of_stmt
(prog.text.(pc))))


  in exec 0 0 0
```

8.7 Driver

learn.ml

```
type action = Ast | Bytecode | Compile

let _ =
  let (action, file_idx) = if Array.length Sys.argv > 2 then
    (List.assoc Sys.argv.(1) [ ("-a", Ast);
```

```
            ("-b", Bytecode);
            ("-c", Compile) ], 2)
else (Compile, 1) in
let lexbuf = Lexing.from_channel (open_in (Sys.argv.(file_idx))) in
let learn_program = Parser.learn_program Scanner.token lexbuf in
match action with
  Ast -> let listing = Ast.string_of_learn_program learn_program
        in print_string listing
| Bytecode -> let listing =
    Bytecode.string_of_prog (Compile.translate learn_program)
  in print_endline listing
| Compile -> Execute.execute_prog (Compile.translate learn_program)
```