Designing a Language for Introductory Computer Science Courses

Andrew Glaude

Abstract—Introduction to Computer Science courses are generally focused on teaching basic problem solving, algorithms, and data structures. Often these courses are bogged down in teaching the complexities of modern programming languages. We identified several pain points in introductory programming courses and desirable features in a learning language. In this paper we present a new language, The Beginner’s Instructional Little Language (BILL) and an open source interpreter written in C#, “BILL”. BILL is designed to minimize the amount of overhead associated with learning your first language. BILL provides a static type system and easy access to important library items like “printin,” list, and “input.” Using BILL in introductory programming courses will allow instructors to focus on the important aspects of Computer Science.

1 INTRODUCTION

The introduction to computer science course at RIT currently makes use of Python. After acting as a teaching assistant for this course I identified several pain points caused by using Python. One of the main issues with using Python was the lack of a static type system when teaching object oriented principles. To get around this issue Professor James Heliotis designed a system (“rit_lib”) [2] that introduced dynamic type checking to objects in Python that prevented student type errors. We found this solution to be not completely satisfactory as it did not present these errors until run time. We explored other popular languages like Java and C++ and found those options to create separate issues. For example, even a “Hello World!” program in Java requires use of access modifiers, the keyword static, function arguments, and the dot operator.

We worked with several faculty members and teaching assistants at RIT to identify ideal properties of a language for educating introductory programmers.

- a static type system
- reduced complexity
- user-defined structures
- easy-to-understand error messages

Although a static type system requires a bit more language syntax from the start, we believe a static type system will aid introductory programmers in writing programs that are bug-free. It is our belief that dynamic type systems allow for beginning programmers to make mistakes that are difficult to locate and identify, because the incorrectly-stored data value does not cause trouble until later in the program execution. In addition, lack of static typing encourages non-transferrable coding styles such as writing a function that returns several different types depending on the inputs.

While there exist many possible languages to use for an introductory course in computer science we are unaware of any that fit all the desires of this faculty and these students at RIT. We present in this paper a language designed to minimize required knowledge of syntax and semantics while using familiar language constructs and syntax so skills learned are easily transferable to other languages. The Beginner’s Instructional Little Language (BILL) uses familiar C syntax structures with a static type system so students...
can more readily identify errors in their code. Common structures like if statements and while loops have identical semantics to the C family of languages. BILL also has easy access to important library functions such as “print” and “input,” and the library class “List.” This reduces the amount of overhead in teaching important problem-solving principles of Computer Science.

The only feature of BILL that is completely different from other commonly used programming languages is the class declaration syntax. A class declaration includes a function parameter-like list that simultaneously declare the fields for this class and for its constructor. BILL automatically provides a method that converts its object to a string. Support for programmer-defined methods in classes is not included.

We believe that this language will reduce the overhead of learning programming syntax and structures and allow instructors to focus on Computer Science principles such as problem solving, data structures, and algorithms.

2 BILL

The Beginner’s Instructional Little Language was designed from the beginning to be used only as a learning and teaching language. BILL has a much reduced feature set compared to other modern programming languages. For example, there is no support for throwing or catching of exceptions, there are no access modifiers, all functions are global, and all fields in objects can be accessed and modified freely. There is no support for polymorphism, user-defined generic types, class methods, or inheritance. While this limits uses of BILL it is also one of the greatest strengths of BILL. With a reduced feature set there are far fewer things to learn as a first time programmer and therefore a student can focus on more basic and important language features such as conditionals, loops, function calls, recursion, and elementary data structures.

2.1 Syntax

The syntax for BILL is defined in an LALR(1) grammar (Appendix A). This grammar is based on the work done by Devin Cook[1] to convert the C# standard to an LALR-compliant grammar. Much of that original grammar has been modified by removing unnecessary productions for C# features that do not exist in Bill. A few productions in the grammar were left in but remain unimplemented to enable future work.

Much of the syntax of BILL is designed to mimic other languages such as C, Java, and C#. This is intentional so that instructors who elect to move to BILL can quickly understand most BILL programs with ease, and students who become familiar with BILL are able to transfer their knowledge and skills to more robust languages.

2.2 Classes

Classes in BILL are somewhat limited in their usage. This is because the design is based on the philosophy exemplified by the “rit_lib” architecture provided for use in Python in the current CS1 course at RIT. Classes in BILL are declared using the format:

```python
class name(type1 name1, type2 name2, ...){ }
```

Because user-defined methods are not currently supported the curly braces at the end currently must be empty.

A variable declaration list is included alongside the class name to define all available fields for this class as well as the parameters for the constructor. No additional constructor may be defined for a class. This follows the “rit_lib” architecture, which did not allow any method definitions. All fields of an object are accessible, for both reading and writing, to any function. There is no support for defining accessors.

Variables referring to user-defined objects and
Lists contain indirect references to those items. All other variables contain their values directly in BILL. This is important as BILL is a pass-by-value language for all function calls, thereby allowing functions to modify object fields, but not allowing them to assign an entirely new object.

2.3 Example Programs
Fibonacci Example Program 1:
The BILL program below calculates and displays the first 20 Fibonacci numbers using the recursive method defined above. Some areas to note include the call to println in the main function. Similar to Python’s print, this function must take a string so the int returned by fibNumber is converted using the built-in function toStr. The Java version of this program is almost identical with only some extra boilerplate code necessary.

```java
void main() {
    for(int i = 0; i < 20; i += 1)
    {
        println(toStr(fibNumber(i)));
    }
}

int fibNumber(int n)
{
    if (n == 0)
    {
        return 0;
    }
    if (n == 1)
    {
        return 1;
    }
    int fib1 = fibNumber(n - 1);
    int fib2 = fibNumber(n - 2);
    return fib1 + fib2;
}
```

Class Example Program 2:
BILL program two prints to the console “Enter a name: “ then waits for input from user. The input() function blocks and waits for a line of input to be entered. The input is then returned as a string with the trailing newline character removed. A Person object is then created invoking the constructor defined as part of the class declaration. Finally “Your name is “ is concatenated with the string entered as input and printed.

```java
class Person(String name){ }
void main() {
    print("Enter a name: ");
    String nm = input();
    Person p = new Person(nm);
    println("Your name is "+p.name);
}
```

2.4 Limitations
As BILL is intended to only be a teaching and learning language for a basic introduction to Computer Science it comes with a number of limitations. BILL supports no large library. The only data structure provided is an indexable list structure. BILL also only supports input and output via the console. There is no available graphic user interface (GUI) library. BILL also does not support inheritance or user-defined methods and therefore can not be used for teaching object-oriented principles like polymorphism or data encapsulation.

The authors believe that these limitations are allowable for the intended purpose of BILL. Based on the current RIT Computer Science 1 curriculum[5] nearly all topics can be properly covered with the current supported features of BILL. Some topics like Dictionaries are not available as libraries to the user, but could be implemented in BILL.

3 BILL INTERPRETER
The BILL Interpreter interprets Bill source files. It is a
C# application that can be run on the .NET runtime on Windows, or the Mono runtime on Unix / Linux systems. Overall the BILL Interpreter is designed to be easily accessible for modification and improvement.

3.1 Design

The BILL Interpreter has three main components: the parser, the type checker, and the interpreter. The parser is built on the GOLD Parsing engine[3]. This is an LALR(1) parser generator that when fed the BILL grammar generates a large skeleton C# file. This file is then modified to properly construct the abstract syntax tree. The abstract syntax tree is constructed from a variety of objects. Parsing errors are detected by the GOLD Parsing tool and are communicated to the user via the ErrorReporter class. ErrorReporter exists so that future iterations of this interpreter may alter how errors and warnings are displayed to the user, perhaps even over a network connection or directly to a log file. The ErrorReporter class is a concrete implementation of the IErrorReporter interface which defines all necessary functions an error reporter must have. This is done so the BILL Interpreter unit tests are able to access error and warning logs programmatically.

The type checker and interpreter both function using the Visitor design pattern[6]. The nodes of the abstract syntax tree all implement the visitable interface and visitable objects accept visitor objects by simply invoking the visitors visit function, passing themselves in. The visitor objects implement different visit functions for each possible visitable object. Following this pattern allows for strong separation of concerns for any class that would like to interact with the abstract syntax tree. This causes future work that involves traversing the tree to be nicely separated from the tree code itself. For example, a debugger visitor could be created that allows users to set breakpoints in their BILL source code.

The type checker, like any other visitor, has functionality for each possible node type of the abstract syntax tree, split into separate functions. It also maintains several state variables to remember its current context. Because of that state it may do things like reject break statements that occur outside of loops, and reject use of variables that are outside of the current scope. Scope is often handled by simply creating a new instance of the TypeValidatorVisitor which is then used to check specific pieces of a node in the tree. That new TypeValidatorVisitor will then have the result type stored in the resultType field which can be used to verify things such as if the conditional of an if statement is truly a boolean expression. The type checker does perform some operations that are required by the interpreter. When visiting a class declaration the type checker will generate a constructor for that class and place information about it into the symbol table.

The interpreter is structured similarly to the type checker. The interpreter does have additional state variables which keep track of program state. There are separate dictionaries that map the name of variables to their value for primitive values, objects, and lists. Currently the implementation for all of the built-in functions for BILL like println, input, and toStr are included in the visit function for function invocation. However as more are added this design will be untenable and the design will have to change. The interpreter also keeps track of a stack counter that is incremented each time a function is entered and decremented when leaving a function. BILL programs may not exceed 1,000 nested function calls currently, but this number may be modified in the future. This is to ensure that an interpreted BILL program does not cause a real stack overflow, which would show an unfamiliar and difficult to understand error message to
the first-time programmer. Instead, if the BILL program attempts to go beyond the enforced nested function limit the program will end and an error that accurately describes what occurred is displayed.

To reduce the number of errors in the BILL Interpreter itself the project makes use of NUnit for both unit and end to end testing. These tests are run manually after any bug fixes or feature additions to ensure no new bugs have been introduced. There are separate unit tests for the parser, type checker, and interpreter as well as end-to-end tests that verify small programs are able to be executed correctly. This section of the code also serves as a great place for contributors to the BILL Interpreter to get their feet wet in understanding how the project works and make meaningful contributions from the start.

4 Future Work

The Bill Interpreter is available on GitHub at https://github.com/ajgajg1134/MS_Project_Bill. Given the large size of this project and overall goal there are still a number of desirable features and changes to be made.

The highest priority future work is likely method support for classes, this will allow adoption of BILL by curricula that rely on the use of methods on objects and make BILL objects more truly objects than structs. There are a number of constructs in BILL that already use this syntax, namely Lists and Strings.

The next goal is an overall increase in stability and tests for the BILL Interpreter. There may certainly be more bugs still to be rooted out. The likelihood of a student being held back by bugs in the BILL Interpreter should be minimized as much as possible.

There is already a lengthy and expanding list of possible future features for BILL included in the GitHub repository for the BILL Interpreter, and they are also included below for the convenience of the reader. Some of these are very large undertakings and could serve as an entirely separate master’s capstone project. These include:

- A switch to force all objects to be immutable
- Port to a non .NET language
- Create development tools (IDE, debugger)
- A Read-Eval-Print-Loop for Experimentation
- Assertions or Contracts
- An interpreter that handles a large number of built-in functions and types better

5 Conclusion

In conclusion BILL can reduce the common language overhead of typical first-time programming languages. The chart below identifies features important to a language in CS 1 courses and how BILL stands against other popular options.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Python</th>
<th>Java</th>
<th>BILL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static compile-time type system</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Simple access to I/O</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>Objects</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Cross Platform</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Easy to learn syntax</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
</tr>
</tbody>
</table>

BILL allows instructors to focus on the principles of computer science and general problem solving ability which are often cited as difficulties in intro courses[4]. It is our hope that with a small amount of additional work BILL can be adopted and tested in classrooms.

Acknowledgement

The author wishes to thank James Heliotis, and Sean Strout for advising on this project.
Andrew Glaude: Designing a Language for Introductory Computer Science Courses

REFERENCES


Andrew Glaude B.S./M.S. Student at Rochester Institute of Technology (Graduation Spring 2017). Research interests include Programming Languages and Computer Science Education. Member of Computer Science House at RIT.
APPENDIX A - BILL GRAMMAR

This grammar uses the GOLD Parser meta-notation for regular expressions and comments.

"Start Symbol" = <Compilation Unit>

! ----------------------------------------------------------------------- Sets

{ID Head} = {Letter} + [_]
{ID Tail} = {AlphaNumeric} + [_]
{String Ch} = {Printable} - ['"]
{Char Ch} = {Printable} - ['']
{Hex Digit} = {Digit} + [abcdef] + [ABCDEF]

! ----------------------------------------------------------------------- Terminals

Identifier = {ID Head} {ID Tail}*
MemberName = '.' {ID Head} {ID Tail}*
DecLiteral = {Digit}+ ( [UuLl] | [Uu][Ll] | [Ll][Uu] )?
HexLiteral = '0'[xX]{Hex Digit}+ ( [UuLl] | [Uu][Ll] | [Ll][Uu] )?
RealLiteral = {Digit}*'.'{Digit}+
StringLiteral = ""( {String Ch} | '\'{Printable} )* ""
CharLiteral = ' ' ( {Char Ch} | '\'{Printable} )'

! ----------------------------------------------------------------------- Comments

Comment Line = '//'
Comment Start = '/*'
Comment End = '*/'

! Shared by multiple sections
! -----------------------------------------------------------------------

<Valid ID> ::= Identifier
       | this
       | <Type>

<Qualified ID> ::= <Valid ID> <Member List>

<Member List> ::= <Member List> MemberName
                | !Zero or more

----------------------

<Literal> ::= true
       | false
       | DecLiteral
       | HexLiteral
\[\begin{align*}
\text{RealLiteral} &\mid \text{CharLiteral} \\
\text{StringLiteral} &\mid \text{null} \\
\text{<Type>} &::= \text{int} \\
&\quad \mid \text{double} \\
&\quad \mid \text{char} \\
&\quad \mid \text{String} \\
&\quad \mid \text{void} \\
&\quad \mid \text{bool} \\
&\quad \mid \text{List} \ '<' \ <\text{Qualified ID}> '>
\end{align*}\]

--------------------------

Expressions

--------------------------

\[\begin{align*}
\text{<Expression Opt>} &::= \text{<Expression>} \\
&\quad \mid \text{!Nothing} \\
\text{<Expression List>} &::= \text{<Expression>} \\
&\quad \mid \text{<Expression>} ',' \ <\text{Expression List}>
\end{align*}\]

\[\begin{align*}
\text{<Expression>} &::= \text{<Conditional Exp>} '=' \ <\text{Expression}> \\
&\quad \mid \text{<Conditional Exp>} '+' \ <\text{Expression}> \\
&\quad \mid \text{<Conditional Exp>} '-' \ <\text{Expression}> \\
&\quad \mid \text{<Conditional Exp>} '*' \ <\text{Expression}> \\
&\quad \mid \text{<Conditional Exp>} '/' \ <\text{Expression}> \\
&\quad \mid \text{<Conditional Exp>}
\end{align*}\]

\[\begin{align*}
\text{<Conditional Exp>} &::= \text{<Or Exp>} '?' \ <\text{Or Exp}> ':' \ <\text{Conditional Exp}> \\
&\quad \mid \text{<Or Exp>}
\end{align*}\]

\[\begin{align*}
\text{<Or Exp>} &::= \text{<Or Exp>} '||' \ <\text{And Exp}> \\
&\quad \mid \ <\text{And Exp>}
\end{align*}\]

\[\begin{align*}
\text{<And Exp>} &::= \text{<And Exp>} '&&' \ <\text{Equality Exp}> \\
&\quad \mid \ <\text{Equality Exp>}
\end{align*}\]

\[\begin{align*}
\text{<Equality Exp>} &::= \text{<Equality Exp>} '==' \ <\text{Compare Exp}> \\
&\quad \mid \text{<Equality Exp>} '!=' \ <\text{Compare Exp}> \\
&\quad \mid \ <\text{Compare Exp>}
\end{align*}\]

\[\begin{align*}
\text{<Compare Exp>} &::= \text{<Compare Exp>} '<' \ <\text{Add Exp}> \\
&\quad \mid \text{<Compare Exp>} '>' \ <\text{Add Exp}> \\
&\quad \mid \text{<Compare Exp>} '<=' \ <\text{Add Exp}> \\
&\quad \mid \text{<Compare Exp>} '>=' \ <\text{Add Exp}> \\
&\quad \mid \ <\text{Add Exp>}
\end{align*}\]
\| \text{<Add Exp>}

\text{<Add Exp> ::= <Add Exp> \text{\texttt{+}} <Mult Exp>}
| \text{<Add Exp> \text{\texttt{-}} <Mult Exp>}
| \text{<Mult Exp>}

\text{<Mult Exp> ::= <Mult Exp> \text{\texttt{*}} <Unary Exp>}
| \text{<Mult Exp> \text{\texttt{/}} <Unary Exp>}
| \text{<Mult Exp> \text{\texttt{%}} <Unary Exp>}
| \text{<Unary Exp>}

\text{<Unary Exp> ::= \text{\texttt{!}} <Unary Exp>}
| \text{\texttt{-} <Unary Exp>}
| \text{\texttt{++} <Unary Exp> !Not currently supported in BILL}
| \text{\texttt{--} <Unary Exp> !Not currently supported in BILL}
| \text{<Object Exp>}

\text{<Object Exp> ::= <Method Exp>}

\text{<Method Exp> ::= <Method Exp> <Method>}
| \text{<Primary Exp>}

\text{<Primary Exp> ::= new <Valid ID> \text{\texttt{('}} \text{<Arg List Opt> \text{\texttt{)}}} !Object creation}
| \text{<Primary>}
| \text{\texttt{('} \text{<Expression> \text{\texttt{)}}}}

\text{<Primary> ::= <Valid ID>}
| \text{<Valid ID> \text{\texttt{('}} \text{<Arg List Opt> \text{\texttt{)}}} !Current object method}
| \text{<Literal>}

! ==------------------------------------------------------------------------==
! Arguments
! ==------------------------------------------------------------------------==

\text{<Arg List Opt> ::= <Arg List>}
| \text{Nothing}

\text{<Arg List> ::= <Arg List> \text{\texttt{,'}} <Argument>}
| \text{<Argument>}

\text{<Argument> ::= <Expression>
C.2.5 Statements

<Stm List> ::= <Stm List> <Statement> | <Statement>

<Statement> ::= <Local Var Decl> ';' | if '(' <Expression> ')' <Statement> | if '(' <Expression> ')' <Then Stm> else <Statement> | for '(' <For Init Opt> ';' <For Condition Opt> ';' <For Iterator Opt> ')' <Statement> | foreach '(' <Valid ID> Identifier in <Expression> ')' <Statement> !foreach not currently supported in BILL | while '(' <Expression> ')' <Statement> | <Normal Stm>

<Then Stm> ::= if '(' <Expression> ')' <Then Stm> else <Then Stm> | for '(' <For Init Opt> ';' <For Condition Opt> ';' <For Iterator Opt> ')' <Then Stm> | foreach '(' <Valid ID> Identifier in <Expression> ')' <Then Stm> !foreach not currently supported in BILL | while '(' <Expression> ')' <Then Stm> | <Normal Stm>

<Normal Stm> ::= break ';' | continue ';' | return <Expression Opt> ';' | <Statement Exp> ';' | ';' | <Block>

<Block> ::= '{' <Stm List> '}' | '{' '}'

<Variable Decs> ::= <Variable Declarator> | <Variable Decs> ',' <Variable Declarator>

<Variable Declarator> ::= Identifier | Identifier '=' <VariableInitializer>

<VariableInitializer> ::= <Expression> | <ArrayInitializer>

! Array Initializations

<ArrayInitializer> ::= '(' <VariableInitializer List Opt> ')' | '(' <VariableInitializer List> ',' ')'
<Variable Initializer List Opt>
   ::= <Variable Initializer List>
   | ! Nothing

<Variable Initializer List>
   ::= <Variable Initializer>
   | <Variable Initializer List> ',' <Variable Initializer>

!===========================================================================
! For Clauses
!===========================================================================

<For Init Opt>
   ::= <Local Var Decl>
   | <Statement Exp List>
   | !Nothing

<For Iterator Opt>
   ::= <Statement Exp List>
   | !Nothing

<For Condition Opt>
   ::= <Expression>
   | !Nothing

<Statement Exp List>
   ::= <Statement Exp List> ',' <Statement Exp>
   | <Statement Exp>

!===========================================================================
! Statement Expressions & Local Variable Declaration
!===========================================================================

<Local Var Decl>
   ::= <Qualified ID> <Variable Decs>

<Statement Exp>
   ::= <Qualified ID> '(' <Arg List Opt> ')' 
   | <Qualified ID> '(' <Arg List Opt> ')' <Methods Opt> <Assign Tail>
   | <Qualified ID> '[' <Expression List> ']' <Methods Opt> <Assign Tail>
   | <Qualified ID> '++'
   | <Qualified ID> '­­'
   | <Qualified ID> '
   | <Qualified ID>

<Assign Tail>
   ::= '++' !Not currently supported in BILL
   | '­­' !Not currently supported in BILL
   | '=' <Expression>
   | '+=' <Expression>
   | '-=' <Expression>
   | '*=' <Expression>
   | '/=' <Expression>

<Methods Opt>
   ::= <Methods Opt> <Method>
   | ! Nothing
<Method>
   ::= MemberName
      | MemberName '(' <Arg List Opt> ')' !Invocation
      | '[' <Expression List> ']
      | '++' !Not currently supported in BILL
      | '--' !Not currently supported in BILL

<Compilation Unit>
   ::= <Program Items>

<Program Items>
   ::= <Program Items> <Program Item>
      | ! Nothing

<Program Item>
   ::= <Method Dec>
      | <Class Decl>

! ============================================================================
! Methods
! ============================================================================

<Method Dec>
   ::= <Qualified ID> Identifier '(' <Formal Param List Opt> ')' <Block>

<Formal Param List Opt>
   ::= <Formal Param List>
      | ! Nothing

<Formal Param List>
   ::= <Formal Param>
      | <Formal Param List> ',' <Formal Param>

<Formal Param>
   ::= <Qualified ID> Identifier

! ============================================================================
! Class Declarations
! ============================================================================

<Class Decl>
   ::= class Identifier '(' <Formal Param List Opt> ')' '{' <Class Item Decs Opt> '}

<Class Item Decs Opt>
   ::= <Class Item Decs Opt> <Class Item>
      | ! Nothing

<Class Item>
   ::= <Method Dec> !Not currently supported in BILL
APPENDIX B - BILL SEMANTICS AND BUILT-INS

Types:

- bool - Boolean type, holds either true or false
- char - 16-bit type, holds one unicode character
- int - 32-bit signed integer type
- double - 64-bit floating point type
- String - A sequence of zero or more unicode characters
- List<type> - A variable sized array holding type type

Operators:

- + - Addition for integer and double types, concatenation for strings
- / - Integer division for two integers, floating point division otherwise
- -, *, % - Subtract (or negate), Multiply, Modulus
- <, <=, >=, > - Less than, Less than or Equal to, Greater than or equal to, Greater than
- ==, != - Equal to, Not equal to
- &&, ||, ! - Boolean AND, boolean OR, boolean NOT

Declarations:

- ret_type name(arg_type arg_name, ...) { block }
  - Defines a function with identifier name, with a defined return type of ret_type, that takes parameters of type arg_type to be called arg_name in block.

- type name = exp;
  - Defines a variable with type type and with identifier name, is assigned the result of evaluating expression exp.

- class name(field_type field_name, ...) { }
  - Defines a class with identifier name, with fields of type field_type and identifier field_name. The order of these fields also defines the constructor for this class.

- class_name name = new class_name(arg_exp, ...);
  - Defines an object of type obj_type and with identifier name, calling the constructor with the result of evaluating arg_exp.

Statements:

- if (cond) { block } - If and only if the expression cond evaluates to True, block is executed.

- if (cond) { block_then } else { block_else }
  - If and only if the expression cond evaluates to True, block_then is executed, otherwise block_else is executed.

- while (cond) { block }
  - While the expression cond evaluates to True block is executed. Cond is evaluated before each execution of block.

- for (for_init; for_cond; for_iter) { block }
  - The statement for_init is executed and any state it creates is kept locally. While for_cond evaluates to true, block is executed, then for_iter is executed before re-evaluating for_cond.

- break;
  - Immediately leave the closest loop (while, or for).

- continue;
  - Immediately move to the end of the body of the closest loop.

- return exp;
- Immediately leave this function, returning the result of evaluating \( \text{exp} \).

\[ \text{name}(\arg_{\text{exp}}, \ldots); \]

- Invoke the function with identifier \text{name} passing in parameters from the result of evaluating \arg_{\text{exp}}. The result of this statement is the result of evaluating the function.

**Built-In Functions:**

- \text{String toStr(int i)};
- \text{String toStr(double d)};
- \text{String toStr(char c)};
  - Takes an argument and returns it as that value’s string representation.

- \text{void println(String s)};
  - Takes a string and prints it to the console with an appended new line.

- \text{String input();}
  - Returns the next line of user input (strips the entered new line). This will block until a user presses enter.

- \text{int toInt(String s)};
  - Converts a string to an integer. If the string is not a valid integer an error message is displayed and the program exits.

- \text{double toDouble(String s)};
  - Converts a string to a double. If the string is not a valid integer an error message is displayed and the program exits.

**List Methods:**

- \text{int size();}
  - Returns the number of elements in this list

- \text{void add(T item)};
  - Adds an item to this list. Adds to the end of the list.