A CS1 Oriented Programming Language
MS Project Milestone 1

Andrew Glaude
Advisors: James Heliotis & Sean Strout

1.0 Introduction

The following represents a semi-formal specification for the language I am designing for my MS Project. The grammar is given for use in an LALR parser, I have been using the GOLD Parsing System[1]. The Meta-language to describe this grammar follows standard Backus-Naur Form, and supports use of regular expressions and set notation with terminals. The exclamation mark is a special character that indicates the rest of that line is a comment.

The syntax of my language will be extremely familiar to any programmer with experience in high level languages such as Java, C#, or C++. Thanks to this, much of the grammar specification below is heavily based on the formal C# specification and an LALR grammar defined for it by Devin Cook. Several important changes have been made including a severe reduction in the complexity of the language, note that there is no support for switch statements, exception handling of any kind, and the available types are fairly restricted. The only concept in this language that is likely to be unfamiliar is the lack of constructors in the creation of classes, for an example and explanation of this see section 4.2.

2.0 Example Programs

2.1 Program #1:

```
1:    void main()
2:    {
3:          println(“Hello World!”);
4:    }
```

Notes:

This is the typical “Hello World” program often used for a student’s first assignment. Although this program is extremely simple there are several important features of the language to note here. Namely, the declaration of main serves as an entry point to the program, no access modifier is provided as they are not included in BILL, and if command line arguments aren’t desired then no formal arguments need be specified. Note also that the print function is simply `println` which is by default accessible in the language.

2.2 Program #2:

```
1:    void main()
2:    {
3:          List<int> nums = new List<int>();
4:          println("Enter integers (-1 to quit)");
5:          int num = toInt(input());
6:          while(num != -1)
7:          {
8:                nums.add(num);
9:                num = toInt(input());
10:         }
```

[1] http://goldparser.org/

```
11:         println("You entered " + toStr(nums.size()) + " numbers.");
12:   }
```

Notes:

Here we see use of the List class which is provided by default as a feature of the language. The syntax for List is extremely similar to Java and C# and will have the necessary methods to interact with it including insertion, access, deletion, and size. BILL also provides access to a function named `input` which simply provides the next line of input as a string. To accompany this several explicit conversion functions are provided such as `toInt`, and `toStr` which are shown above.

2.3 Program #3:

```
1:    class Student(double gpa, String name){ }
2:
3:    void main(){
4:          Student a = new Student(3.8, "Andrew G");
5:          println(a.toString());
6:    }
```

Notes:

This program shows what I've begun to refer to as class default field construction syntax. On line one you will notice that in the definition of the class what looks like a formal parameter list is included after the class identifier. This parameter list serves to define the constructor of the class as well as any desired fields the class should contain. This is designed to mimic the behavior students at RIT in CS1 currently face using the "rit_lib" in Python. To use this constructor simply use the keyword `new` then provide the name of the class and the parameters in the order they were originally given in the class declaration.

3.0 Formal Grammar

```
!------------------
! Much of this grammar is heavily based on the C# Specification
! and an LALR grammar for it written by Devin Cook
!------------------

"Name"     = 'TBD'
"Version"  = '1.0'
"Author"   = 'Andrew Glaude'

"About"    = 'TBD was created as an ideal CS1 language at RIT for an MS Project.'
           | 'Based heavily on other OOP languages like C# and Java.'

"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}*        !The @ is an override char

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
        |  RealLiteral
```

```
        |  CharLiteral
        |  StringLiteral
        |  null

<Type>
       ::= int
        |  double
        |  char
        |  String
        |  void
        |  bool
        |  List '<' <Qualified ID> '>'

!------------------
! Expressions
!------------------

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

<Expression List>
       ::= <Expression>
        |  <Expression> ',' <Expression List>


<Expression>
       ::= <Conditional Exp> '='   <Expression>
        |  <Conditional Exp> '+='  <Expression>
        |  <Conditional Exp> '-='  <Expression>
        |  <Conditional Exp> '*='  <Expression>
        |  <Conditional Exp> '/='  <Expression>
        |  <Conditional Exp>

<Conditional Exp>
       ::= <Or Exp> '?' <Or Exp> ':' <Conditional Exp>
        |  <Or Exp>

<Or Exp>
       ::= <Or Exp> '||' <And Exp>
        |  <And Exp>

<And Exp>
       ::= <And Exp> '&&' <Equality Exp>
        |  <Equality Exp>

<Equality Exp>
       ::= <Equality Exp> '==' <Compare Exp>
        |  <Equality Exp> '!=' <Compare Exp>
        |  <Compare Exp>
```

```
<Compare Exp>
       ::= <Compare Exp> '<'  <Add Exp>
        |  <Compare Exp> '>'  <Add Exp>
        |  <Compare Exp> '<=' <Add Exp>
        |  <Compare Exp> '>=' <Add Exp>
        |  <Add Exp>

<Add Exp>
       ::= <Add Exp> '+' <Mult Exp>
        |  <Add Exp> '-' <Mult Exp>
        |  <Mult Exp>

<Mult Exp>
       ::= <Mult Exp> '*' <Unary Exp>
        |  <Mult Exp> '/' <Unary Exp>
        |  <Mult Exp> '%' <Unary Exp>
        |  <Unary Exp>

<Unary Exp>
       ::= '!'  <Unary Exp>
        |  '-'  <Unary Exp>
        |  '++' <Unary Exp>
        |  '--' <Unary Exp>
!------ |  '(' <Expression> ')' <Object Exp>     !Cast "expression" (Maybe optional)
        |  <Object Exp>

<Object Exp>
       ::= <Method Exp>

<Method Exp>
       ::= <Method Exp> <Method>
        |  <Primary Exp>

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

<Primary>
       ::= <Valid ID>
        |  <Valid ID> '(' <Arg List Opt> ')'    !Current object method
        |  <Literal>

! ========================================================================
! Arguments
! ========================================================================

<Arg List Opt>
       ::= <Arg List>
```

```
        |  !Nothing

<Arg List>
       ::= <Arg List> ',' <Argument>
        |  <Argument>

<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>
        |  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>
        |  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 '=' <Variable Initializer>

<Variable Initializer>
        ::= <Expression>
         |  <Array Initializer>

! ----- Array Initializations

<Array Initializer>
       ::= '{' <Variable Initializer List Opt> '}'
        |  '{' <Variable Initializer 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> '++'                         <Methods Opt> <Assign Tail>
        |  <Qualified ID> '--'                         <Methods Opt> <Assign Tail>
        |  <Qualified ID>                                            <Assign Tail>

<Assign Tail>
       ::= '++'
        |  '--'
        |  '='   <Expression>
        |  '+='  <Expression>
        |  '-='  <Expression>
        |  '*='  <Expression>
        |  '/='  <Expression>

<Methods Opt>
        ::= <Methods Opt> <Method>
         |  ! Nothing

<Method>
        ::= MemberName
         |  MemberName '(' <Arg List Opt> ')'    !Invocation
         |  '++'
         |  '--'

<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>
```

4.0 Semantics

Thanks to the similarities of this language to other OOP languages much of the semantics of BILL are identical to Java or C#, at some points exactly so. Expressions (as defined by `<Expression>` and its sub-rules) have order of operations as created by the formal grammar in section 3.0. The reader will also note that expressions are not valid statements unlike a language like C. Unfortunately Section C.2.5 is somewhat complicated due to use of an LALR parser, in order to handle the dangling-else case and prevent ambiguity some of the grammar is repetitious here.

Much of the semantics of this language rely on the modification of an environment which holds references to variables, classes, and functions. Upon entering new scope a new copy of the environment is made and used to enforce scope and prevent local variables in one function from being accessed in another. Function and Class declarations all affect global scope and are therefore accessible from all points in the program, they need not be declared above usages. It is my goal for the language to support function overloading allowing multiple functions with the same name, but different parameter lists to co-exist in the global environment.