Final Report
ZX81 BASIC

Carlos G. Alustiza

Programming Languages and Translators

Columbia University – Fall 2006

Table of Contents

Introduction....4
Language Tutorial....6
- Program structure....6
- Variables....7
- Loops....7
- Jumps....8
- Comments....9
- Operators....10
- Strings and Alphanumeric Variables....10

Language Manual....11
- Lexical Conventions....11
- Tokens....11
- Comments....11
- Identifiers....12
- Keywords....13
- Constants....13
- Meaning of Identifiers....13
- Storage Class....14
- Types....14
- Conversions and Promotions....14
- Expressions....15
- Multiplicative Operators....15
- Additive Operators....15
- Equality Operator and Assignment Operator....16
- Declarations....16
- Type specifiers....17
- Statements....17
- Line Labeling....17
- Expression Statements....17
- Iteration Statements....18
- Jump Statements....18
- Program Termination....19

Project Plan....20
Architectural Design....22
Test Plan....25
Lessons Learned....27
Appendix....28
- BASICRuntime.java Implements the BASIC Runtime....28
- BASICVariable.java Implements a BASIC variable....43
- BASICLoop.java Implements a FOR...NEXT loop....44
- Main.java Implements the Boot component....46
- grammar8.g Implements the BASIC grammar....49

Introduction

This report describes an adaptation of the BASIC dialect used by the Sinclair ZX81 computer. No standard supports either this adaptation nor the original Sinclair BASIC dialect as, by the time the ZX81 computer was developed and sold (1980s), each manufacturer had the freedom to implement those portions of BASIC that best suited their interests and their machines' characteristics.

The motivation for this implementation of the ZX81 BASIC dialect is an article in Salon.com[1] by David Brin that describing the lack of a simple and fun to use programming language that can suit the needs of young children without forcing them to delve into language-specific or platform-specific complexities, but instead allows them to focus on the algorithms being implemented. Brin identifies BASIC as the perfect language for such application, but laments that BASIC implementations are hard to find, unless you buy an old 1980s home computer.

When I was a kid, the ZX81 BASIC was my first introduction to Computer Science and provided many hours of creative thinking in front of the TV set, home computers in those years used TVs as their display, and made me curious about computers. I think that the same opportunity should not be missed with today's kids, and hence I decided to develop a new version of the ZX81 BASIC dialect.

To develop a usable BASIC interpreter, I decided to eliminate certain instructions of the ZX81 dialect that had a direct reference to the ZX81 computer architecture:

Speed instructions (FAST, SLOW)
Memory instructions (PEEK, POKE)
Screen output (COL, TAB, AT)
Printer output (LPRINT, LLIST)
Interpreter commands (RUN, LIST)
Program saving / loading (SAVE, LOAD)

Also simplified variable manipulation by eliminating arrays and “slicing”, which was the way in which string variables could be used as arrays to access individual characters. These simplifications do not affect the essence of the BASIC language, but allow to create a working interpreter in the time allotted for this project.

The origins of the BASIC language can be traced back to 1963 when John George Kemeny and Thomas Eugene Kurtz at Dartmouth College designed the “Beginner's All-purpose Symbolic Instruction Code”, but it was not until the 1980s that BASIC became a very popular language with the revolution in home computers[2], of which the Sinclair ZX81 is a

1 David Brin, “Why Johnny can't code”. Salon.com (http://www.salon.com/tech/feature/2006/09/14/basic/index.html)

2 Source: Wikipedia (http://en.wikipedia.org/wiki/BASIC), 2006.

perfect example. A single reference work will be used as the guide to the ZX81 BASIC dialect; it is "Sinclair ZX81 Programming" by Steven Vickers, Second Edition, 1981.

Language Tutorial

Writing a BASIC program is not hard, however there are a couple of rules you have to understand in order to get the most from your BASIC interpreter.

Program structure

A BASIC program consists of statements on numbered lines, the line numbering is a requisite, as it is the way a programmer explains to the BASIC interpreter the basic order in which to process the statements. All statements occupy only one line and have to have a line number to the far left, separated from the rest of the statement by whitespace. Here's an example:

```
10 REM VOID PROGRAM
```

In the previous line, 10 is the line number and REM VOID PROGRAM constitutes a statement. There is no need to use all-caps for statements and you can mix upper and lower case letters in your statements without affecting their meaning. There is an important restriction about line numbers: two different lines cannot have the same line number. However, you can enter your lines in whatever order suits you and the interpreter will sort them in ascending order according to their line numbers.

The end of a BASIC program is signaled by the End Of File (EOF) you do not have to indicate the end of a program, you just write all the statements in numbered lines and the interpreter will figure it out.

Here is an example of the famous “Hello World” program in BASIC:

```
10 PRINT “Hello World!”
```

When you run the program you should see: Hello World! on your screen. The PRINT instruction indicates to the interpreter that whatever follows has to be shown on standard output. If you want to print text, you have to enclose it with “”. Otherwise it would return an error message. To get acquainted with the way the interpreter handles errors, let us repeat the previous program, but this time, without “” enclosing the text after PRINT:

```
10 PRINT Hello World!
```

After running this program, the interpreter gives us the following message:

```
Variable not found Hello Line: 11
```

What happened is that the interpreter tried to understand Hello as the name of a variable and since we did not specify Hello as a variable, it was not possible for the interpreter to print anything. This error leads us to the next important concept in BASIC, variables.

Variables

When we want to store values in memory to use them for calculations or printing, variables come in handy. Before we use a variable, we have to create it and assign an initial value to it. Both operations are carried out with the LET statement, as the following example shows:

```
10 LET a = 5
```

If we run this program, nothing happens, why? Because we are only storing the value 5 in variable a, but we are not instructing the computer to do anything else with it. Let us try a variation of the LET statement, that allows us to operate on the value of a variable.

```
10 LET a = 5
20 PRINT a
30 LET a = a * 2
40 PRINT a
```

We just used the recently learned statements, PRINT and LET to create a program that actually does create a variable, assign an initial value to it and display the initial value, modify that value and display the new value. After running the program we get the following output:

```
5.0
25.0
```

It means that variable a received the value 5 on creation and then its value was multiplied by 5 to obtain a new value (25).

Loops

There is a way to repeat instructions in a controlled way in our programs, the FOR ... NEXT loop. This statement is actually comprised of two different statements, FOR and NEXT. The FOR statement initializes a loop block while the NEXT statement signals the end of the loop block, whatever statements between FOR and NEXT will be repeated.

To initialize a loop, the FOR statement defines a control variable, that variable will hold a value to be tested in each iteration of the loop and while the control variable stays below the limit condition, the loop gets repeated. Let us illustrate the concept of loops with a small example.

```
10 let a = 1
20 for x = 1 to 10 step 1
30 print a
40 let a = a + 1
50 next x
```

The FOR statement defines x as a control variable, assigns it an initial value of 1 and sets the limit condition to 10 (the 'TO 10' portion) it also indicates that every time the loop is repeated, the value of the control variable gets incremented by 1 (the 'STEP 1' portion). After the FOR statement, lines 30 and 40 will be repeated until x becomes greater than 10. Please remember that the NEXT statement, signaling the end of the loop block has to have x as the control variable or you will get an error.

It is possible to next FOR...NEXT loops, consider the following program (indented to improve clarity):

```
10 let a = 1
15 let z = 1
20 for x = 1 to 10 step 1
        30 print a
        31 for b = 1 to 4 step 1
                32 print z
                33 let z = z * 3
        34 next b
        40 let a = a + 1
50 next x
```

Jumps

Sometimes, we do not want to execute our program in linear fashion, but we prefer to jump around. It is possible with the GOTO statement. Consider the following program:

```
10 print "Hello World"
20 goto 10
```

If you execute the program you will see that it simply prints “Hello World” constantly and

never stops, we just created an infinite loop because every time our program reaches line 20 it gets instructed to go back to line 10 and start running from there, which in turn will take it back to line 20 and successively. The GOTO statement is powerful, but should be used with caution.

Sometimes we need to jump to a different portion of the program to perform an action, but want to come back to where we were before the jump after executing some statements. In that case, the GOSUB RETURN statements come in handy. Consider the following program:

```
10 let a = 1
20 gosub 1000
30 print a
40 stop
1000 let a = a * 3
1010 print a
1020 return
```

When executed, the program assigns 1 to the variable a and jumps to line 1000. There, a gets a new value which equals its old value times 3 (remember to leave spaces on both sides of *), then the value of a is printed. On line 1020 we find RETURN and jump back to line 30, the line immediately after the GOSUB statement and PRINT again the value of variable a. The output of this program looks as follows:

```
3.0
3.0
```

The reason for the STOP statement on line 40 is to avoid having the program execute the lines 1000, 1010 and 1020 again. A STOP statement simply means “finish the program” and that is what we wanted to do after jumping with the GOSUB statement. Not separating the 'main body' from the 'subroutines' with STOP will provoke a nasty error message as the interpreter tries to execute the 'subroutine' but without a matching GOSUB statement.

Comments

So far, we did not explain the very first BASIC statement we came across:

```
10 REM void program
```

The REM statement indicates that the line constitutes a comment, something left by the programmer to document what the program does, but to the interpreter it means nothing and will simply skip it.

Operators

BASIC supports four mathematical operators:

"+" addition
"-" subtraction
"*" multiplication
"/" division

These operators can be used in binary operations (ie only two terms) and do not support more complex expressions. Thus LET a = a * 3 is valid, while LET a = a * 2 + 1 is not.

One operator, "+" can also be used with alphanumeric variables and will be introduced in the next section.

Strings and Alphanumeric Variables

It is possible to define an alphanumeric variable and store strings inside it, even better, it is possible to concatenate an alphanumeric variable with a string using the "+" operator, as the following program shows:

```
10 let a$ = "Hello"
20 print a$
30 gosub 1000
40 stop
1000 let a$ = a$ +" World"
1010 print a$
1020 return
```

You just have to remember that "+" used to concatenate a string with a variable does not accept a space between it and the string and it is not possible to do the inverse operation. Namely concatenate a string and a variable, as the LET statement would not recognize the string as a variable and would return a syntax error.

Language Manual

Nota Bene: for detailed syntax of rules, please refer to grammar8.g in the Appendix

Lexical Conventions

Tokens

There are 4 classes of tokens: identifiers, keywords, string literals, and operators. Horizontal and vertical tabs, newlines, form feeds and comments as described below are ignored except as they separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

Comments

The keyword **REM** at the beginning of a new line introduces a comment, which is terminated with CRLF (line terminator). Comments do not nest, do not extend more than a line and occupy a single line for themselves. Once a REM keyword is found at the beginning of a line, the interpreter ignores whatever tokens appear afterwards, up to and including the line terminator.

Syntax:

REM <characters, white space or digits>

Example:

REM This is a comment. Number of comment: 1
REM This is another comment. Number of comment: 2

Syntax errors:

REM This is a comment
this is the second line of the comment.

A REM keyword has to be found at the beginning of every comment.

LET a = 2 REM a simple addition

A comment cannot be placed beyond the beginning of a line, unless all characters to the left of the REM keyword consist of white space.

Identifiers

An identifier is a sequence of letters and digits that uniquely identifies a portion of storage reserved for the program, which allows us to say that an identifier is a synonym for a variable. Below is a list of characteristics of identifiers:

1. The first symbol of an identifier has to be a letter.
2. Uppercase and lowercase letters make no difference.
3. Identifiers can have any length, but cannot extend beyond the line terminator. An exception are control variables in FOR ... NEXT loops which have to be 1 character long.
4. Valid symbols for identifiers are [A-Z], [a-z], [0-9] defined as ranges within the ASCII character set. No white space is allowed in identifiers.[3]
5. The only special character allowed in identifiers is $ (dollar sign) and only for alphanumeric variables.

Identifiers have some restrictions depending of the data type they represent and their use throughout the program. Below is a list of restrictions for identifiers:

1. If the identifier represents a control variable in a FOR ... NEXT loop, its maximum length is 1 character and no digits are accepted.
2. If the identifier represents a string variable, its maximum length is 2 characters, but the rightmost character has to be a $.

Summarizing, identifiers for numeric values can be of any length, but identifiers for control variables in loops or identifiers for string variables have restrictions of their length.

<u>Example</u>:

A$ (string identifier or variable)
A or a (numeric variable or control variable, which is also numeric)
AABB11232 (numeric variable)

<u>Syntax errors</u>:

2$ (wrong, identifiers cannot start with numbers)
This is my identifier (wrong, no whitespace is allowed)

3 This is a deviation from the Sinclair BASIC dialect, as it allowed whitespace to be used in some identifiers, but ignored everything to the right of the first instance of white space. For example, “my identifier” would be known to the interpreter as “my” eliminating all white space and everything else after it.

ADG$ (wrong, string identifiers are allowed only one letter before the $ symbol)

Keywords

The following strings of characters are reserved as keywords, and may not be used otherwise:

ABS	EXP	NEXT	SQR
ACS	FOR	NOT	STEP
AND	GOSUB	OR	STOP
ASN	GOTO	PRINT	THEN
ATN	IF	REM	TO
CHR$	LET	RETURN	
COS	LN	SIN	

Constants

There are two types of constants and each has a different data type: numeric constants and string constants.

Numeric Constants

A number consists of a series of digits. No octal or hexadecimal representations of numbers are accepted as valid, but all numeric constants are in decimal base.

Numbers can be represented as having an integer part and a decimal part. The integer and decimal parts both consist of a sequence of digits. Either the integer or the decimal part (but not both) may be missing. The decimal point may be missing.

String Constants

A string constant is a sequence of one or more symbols from the ASCII character set enclosed in double quotes. A string constant does not contain the “ (double quotes) character or line terminators; to represent “ (double quotes) within a string constant, the sequence “”” (double quotes x3) is used.

Meaning of Identifiers

As defined above, identifiers are synonyms with variables, which in turn are simply an allocation of storage and their interpretation depends on two main attributes: the storage class

and the type. The storage class determines the lifetime of the storage associated with the identifier and the type determines the meaning of the values represented by the identifier. An identifier also has a scope, which is the program region in which is known.

Storage Class

There are two storage classes: automatic and static. The keyword **LET** specifies a static storage class, while only the control variables in a **FOR ... NEXT** loop have an automatic storage class, are discarded upon exit from the loop and their scope is the loop body, being possible to declare a variable with the same identifier as a control variable.

All variables are declared inside the single block that corresponds to a program. Declaring a variable inside a **FOR ... NEXT** loop makes it available immediately after its declaration and remains in scope until the program ends.

Types

There are only two types in the language: string and number. Variables declared as strings are large enough to store any member of the ASCII character set and their final size is only limited by the available memory. All symbols composing a string are stored as their ASCII codes, which are non-negative numbers.

Variables declared as numbers all can optionally represent decimal values.

There is a close relationship between identifiers, variables, and types in the language. Depending on how we specify the identifier, the resulting variable will be of type string or number. Therefore, an identifier not only provides a unique name for the variable, but also assigns a type to it.

Conversions and Promotions

No conversions are allowed between types nor promotions are available in the language. A small (in terms of value) number stored in a numeric variable is not altered even after being added a fractional or exponential part. The variable will remain of type number and the size of allocated storage does not change.

In every expression that can be defined in the language, both operands (lvalue and rvalue) have to be of the same type, or the interpreter will signal the expression as a syntax error.

Expressions

Three different types of expressions are defined for the language:

numeric-expression.
string-expression
relationship-expression

Numeric expressions are numeric identifiers or a combination of numeric identifiers and operators that yield a numeric value when evaluated by the interpreter. The fundamental characteristic of numeric expressions is that their value can be assigned to a numeric variable.

String expressions are string identifiers. The fundamental characteristic of string expressions is that their value can be assigned to a string variable.

Relationship expressions are the combination of identifiers and relational and logical operators and can only take two values: TRUE and FALSE. The fundamental characteristic of relationship expressions is that they require all operands to the of the same type and also require a relational operator to act on the operands, with the logical operators being optional. Besides these syntactic characteristics, relationship expressions cannot be assigned to a variable, but their value is used by the interpreter internally.

Multiplicative Operators

The multiplicative operators * and / require the left operand to be a numeric variable.

* denotes multiplication.
/ yields the quotient of operands, unless the second operand is zero, in which case a “division by zero” is returned by the interpreter.

Additive Operators

The additive operators can be used in three distinct cases:

numeric-variable + number Returns the arithmetic sum of the operands
numeric-variable – number Returns the arithmetic difference of the operands
string-variable + string-literal Returns the concatenation of the right operand after the left

operand

These are the only cases allowed, due to the lack of type conversions and the string difference not being defined.

Equality Operator and Assignment Operator

The operator = is considered the equality operator when used in String or numeric expressions and assigns to the left operand the value of the right operand, with the requirement that the left operand is an identifier, while the right operand can be a number or string literal, depending on the variable type (left operand).

Declarations

Declarations specify the interpretation given to an identifier and initialize it with a value, hence a declaration includes an assignment. Every declaration is prefixed with the keyword **LET** and follows a few simple rules:

1. The equality operator is used to separate the identifier (to the left) from the initialization value.
2. The operand to the left of the equality operator can only be an identifier.
3. The operand to the right of the equality operator can be the same identifier or number or String literal, depending on the type of the left-hand operand.
4. Both operands have to be of the same type.

A declaration assigns the value of the right operand to the left operand, if the left operand is a new identifier, then storage space is created for it before assigning it a value.

Syntax:

LET A$ = “Hello”
LET MYNUMBER = 2

A special case is when an identifier gets as a value an expression that involves its current value.

LET A = 1
LET A = A +1

In this case, we are incrementing the value of A by one, as a general rule, a declaration only

accepts a right operand that is an expression, if that expression involves the left operand, which is first evaluated by the interpreter, then used in the right operand's expression and then gets replaced by the value of the evaluation of the right operand.

Type specifiers

Type specifiers are simple, a $ (dollar sign) following a letter indicates a string identifier, while a longer identifier without the $ sign identifies a numeric identifier. While a numeric type does not impose conditions on the identifiers (with the exception that $ cannot be used), the string type rules the length and terminator for the identifier.

Statements

Statements cannot span multiple lines and multiple statements cannot be placed in a single line. There are four types of statements available, which will be defined in detail in following sections:

1. expression statement
2. iteration statement
3. jump statement

Line Labeling

All program lines have to be labeled with an integer number that is greater than zero, even comments are labeled with a line number. Two or more lines cannot share a line number. White space separates the line number from the statement that conforms the line.

Expression Statements

Most statements in the language are expression statements and most expression statements are assignments or declarations. The use of the keyword LET is mandatory.

Example:

```
LET A$ = “Hello”
LET B$ = “ World”
LET ROOT = SQR(9)
```

Iteration Statements

Are created using the **FOR TO STEP NEXT** keywords and take the following general syntax:

Syntax:

FOR control-variable **=** constant1 **TO** constant2 **STEP** constant3
<block of statements to be repeated in the loop>
NEXT control-variable

The control variable is created and receives the value of constant1, which has to be a numeric value. After the **TO** keyword, constant2 specifies the limit for the values of the control variable, meaning that the loop will be executed while control-variable <= constant2. **STEP** provides the increment for the control variable and it is applied to it using and arithmetic sum, which allows to indicate negative values to decrement the control variable or decimal values to make the control variable advance by fractional steps. All constants in the loop have to be numeric constants.

The **NEXT** keyword indicates the interpreter that the block of statements whose execution is controlled by the control variable has finished and the interpreter must go back to the **FOR** statement to evaluate the limit condition for the control variable and repeat the execution of the statement block or transfer the execution to the next line after the **NEXT** statement for the loop.

The loop body extends from the next line after the **FOR** statement to the previous line to the accompanying **NEXT** statement. Iteration statements can be nested and all variables, except the control variables, declared within the loop body are available from the moment of their declaration to the moment the program ends.

Jump Statements

Jump statements are implemented with the **GOTO** and **GOSUB ... RETURN** keywords. **GOTO** statements make the interpreter switch the program execution from the line where **GOTO** appears to the line whose numeric label appears after **GOTO**. If the line does not exist, the interpreter stops program execution and returns an error message.

GOSUB statements switch the interpreter to the line whose numeric label appears after the **GOSUB** keyword, but the interpreter keeps a reference to the line immediately after the **GOSUB** statement. Once the jump to the new line is performed, the apparition of a **RETURN** keyword will switch the interpreter back to the line whose reference is keeping.

It is possible to call **GOSUB** from a block that was accessed after another **GOSUB** call, the first **RETURN** statement, will simply return to the next line after the last **GOSUB**, as references are stored in a LIFO stack

All variables declared after a jump due to a **GOTO** or **GOSUB** are available from that moment on until the program ends.

Program Termination

The keyword STOP signals the program end or the interpreter will simply execute lines sequentially before and after jump statements and will end only when there are no more lines to execute in the sequence. It is up to the programmer to define a STOP statement to make the interpreter to stop execution at a predefined point.

Example:

```
10 LET A = 1
20 FOR C = 0 TO 10 STEP 1
30 LET A = A+1
40 NEXT C
```

The interpreter finishes execution after the loop exits, as there are no more lines to execute.

```
10 LET A = 1
20 FOR C = 0 TO 10 STEP 1
30 A = A+1
40 GOSUB 100
50 NEXT C
60 STOP
100 A = A + 1
110 RETURN
```

Here, the programmer added a **STOP** statement to ensure that after the loop is exited, the interpreter cannot keep executing the code in lines 100 and 110, as it would alter the value of variable A and would generate an error when the **RETURN** statement cannot find a return address in the **GOSUB** stack.

Project Plan

Specification Process

As the BASIC language specification being used was that of the ZX81 BASIC dialect. The specification process consisted of the following steps:

1. Identify those BASIC characteristics that are specific of the ZX81 machine
2. Identify those BASIC characteristics that would be too complex / time consuming to implement in the time available.
3. Purge the BASIC specification of elements identified on points (1) and (2)
4. Write language examples to explore the different ways in which language statements can be formulated
5. Write preliminary grammar rules that match the examples created in point (4)

The result of the specification process was a list of functionality that would not be included in the interpreter (memory manipulation, line printer commands, arrays and string slicing) and a set of grammatical rules for the language.

Planning Process

The planning process consisted of identifying the implementation tasks derived from the language specification and the tasks required to develop and test the interpreter:

1. Sort tasks by critical path. For example, implementing jumps in the interpreter requires the previous implementation of a program counter and a method to update it.
2. Sort tasks by level of complexity. Once the relationships and dependencies among tasks are known, sort them according to their complexity level.
3. Define alternative paths of implementation if possible to avoid stopping the process when a problem is found. For example, the development of assignments (LET statement) was an alternative identified (and used) in case the implementation of FOR...NEXT loops became too complicated. The idea behind this is to keep the project moving
4. Define a time for tests

The result looked like a laundry list with links between tasks, but served the purpose of organizing the project. While I did not have to account for team partners, I found that I grossly underestimated the time required to implement and test many tasks, which led to delays and corner-cutting.

Development and Testing Processes

The development process was of an iterative nature. From the specifications, code was generated (Java code and grammar code) then unit tested and then tested as part of the whole BASIC interpreter. For unit testing a single statement was enough, for system testing a small BASIC program was built.

Programming Style

I did not create a programming guide for Java because I did not have a team. However, I followed certain rules:

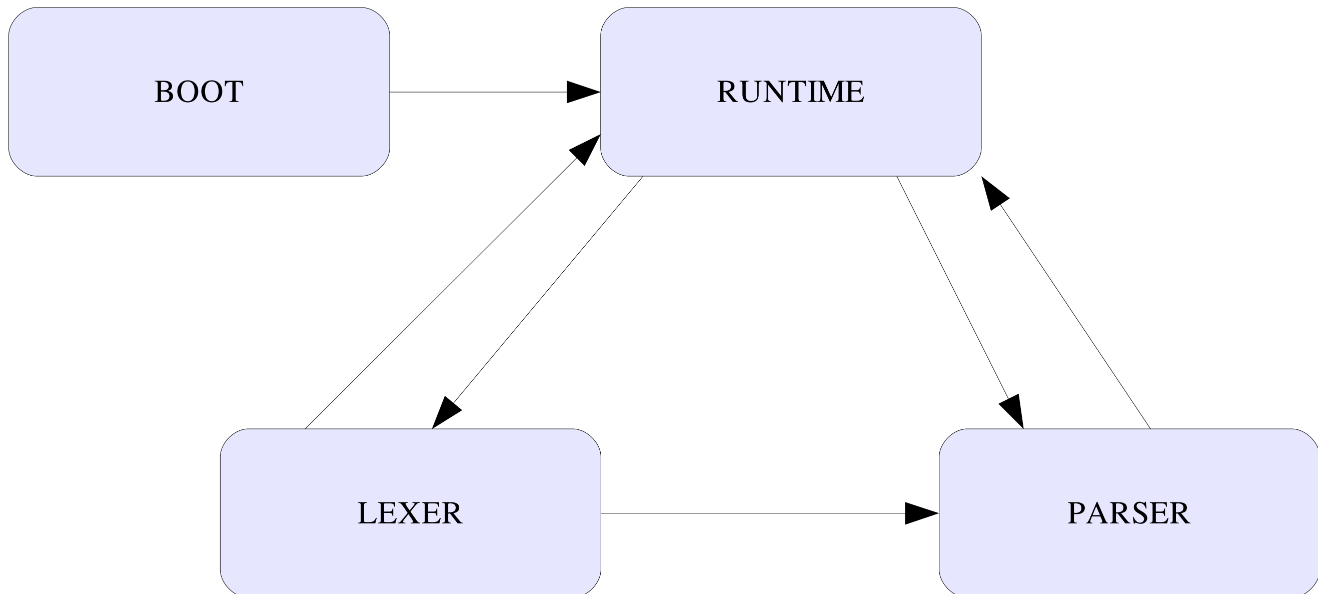
1. Variables that will be discarded when the method exits are prefixed with 'tmp' from temporal.
2. Variables that will be used inside a method only, but should be available in the whole method body are prefixed with 'my'.
3. All variable names should be descriptive
4. Break lines when necessary to improve readability.
5. Indent blocks of code to provide visual cues on what depends on what.
6. Use 'this' always, even when it is redundant, to avoid ambiguity when referencing a method or field.

Software Development Environment and Tools

I used NetBeans 5.0 exclusively for all Java needs.

Architectural Design

The next graphic identifies the main blocks in the architecture of the BASIC interpreter:



Boot

1. Reads input file (*.bas)
2. Sorts lines according to their user-assigned line number
3. Checks for duplicated line numbers
4. Initializes Runtime (by calling its run method)

The Boot block starts the whole process, but does not do much, simply reads the source BASIC file (*.bas), checks for duplicated lines and sorts lines by their line number. Once the sort is finished, it initializes the BASIC runtime and passes it the complete listing as a sorted array.

Runtime

The runtime is the heart of the BASIC interpreter. Its functions are:

1. Initializes internal DS (Data Structures), such as:
    a. Call stack
    b. Program counter
    c. Variable table
2. Initializes Lexer and Parser
3. Creates communication channel with Runtime

The initialization of the Lexer consists in assigning an input stream to it and the initialization of the Parser consists in passing the Lexer to the Parser constructor. Creating a communication channel is a bit more complex, but it is assigning a reference to the Runtime to both Lexer and Parser as the value of a protected field (myCallerLexer, myCallerParser) so it is possible to query the Runtime for information (Lexer) or execute methods in the Runtime to carry out actions (Parser).

The Runtime will provide the Lexer with the current BASIC statement to be analyzed and executed and will receive invocations (by the Parser) every time a valid statement needs to be executed.

After the initialization phase, the Runtime enters the main loop, which is executed as long as the end of the BASIC program has not been reached and the stop_condition has not been signaled by a STOP statement or an error. The body of this loop simply invokes the executeCurrentLine method in the Runtime.

Executing the current line (executeCurrentLine method) implies feeding the Lexer with the text of the current BASIC line to be interpreted an executed. Once the Lexer and Parser do their job, the Runtime expects to execute a BASIC statement through an invocation of a method.

Here's an example:

10 PRINT “Hello World”

The following tasks are performed when we execute this program:

1. Boot loads the file and checks it
2. Boot calls Runtime

3. Runtime initializes the Lexer and Parser
4. Runtime sets program counter to 1
5. Runtime enters main loop
6. Runtime asks the Parser to parse the line
7. Parser reads the line, considers it valid and invokes the method doPRINT("Hello World") in the Runtime
8. The doPRINT method prints the string passed as argument to standard output
9. The doPRINT method increments the program counter by 1 and exits
10. Runtime returns to the main loop, but program counter is greater than the number of lines in the program, therefore the Runtime exists.

Every method in the Runtime increments the program counter by 1; GOTO and GOSUB statements set the program counter to the index of the appropriate line. In our example, line 10 has index 1. The Runtime handles the translation between the original line number and an index. The index for each line is a result of the sorting based on line number performed by the Boot component.

Lexer and Parser

Both components are created by ANTLR from a grammar file. No modifications are performed on ANTLR-generated code.

Test Plan

The test plan was based on several BASIC source files (some examples below) and unit testing was performed by feeding statements to the Runtime by hand and setting breakpoints on the code. Debugging the grammar files was the most complex part as it required me to review a grammar file by hand, debug its associated Java code and debug my own code before figuring out where the error was.

The test plan is not sophisticated because I had a handful with design and coding.

CHARMAP.BAS

```
3000 rem prints the character map
4000 print "Character Map"
4010 for n = 0 to 255 step 1
4020 print chr$ (n)
4030 next n
```

GOSUB.BAS

```
11 print "Exploring gosubs and global scope"
12 print "The variable a is defined in the main body"
13 print "and modified in each gosub"
20 let a=0
30 gosub 1000
31 gosub 2000
32 gosub 3000
33 gosub 4000
40 stop
1000 print "First gosub"
1010 let a = a + 1
1020 print "Value of a is"
```

```
1030 print a
1040 return
2000 print "Second gosub"
2010 let a = a + 1
2020 print "Value of a is"
2030 print a
2040 return
3000 print "Third gosub"
3010 let a = a + 1
3020 print "Value of a is"
3030 print a
3040 return
4000 print "Fourth gosub"
4010 let a = a + 1
4020 print "Value of a is"
4030 print a
4040 return
```

COSINE.BAS

```
10 rem prints the cosine of the first 10 numbers
11 print "printing the cosine table (1 - 10)"
20 for x = 1 to 10 step 1
30 print cos(x)
40 next x
```

Lessons Learned

The main lessons learned are:

1. Keeping a simple design helps when tracking errors
2. What looks good on paper does not necessarily work on the first time on ANTLR

It cannot be overestimated how valuable is to have a simple design when you have to debug your code. By creating a simple Runtime with well-defined interfaces to the Lexer and Parser I was able to concentrate on the logic errors instead of running through spaghetti code that results from complex designs.

While I did most of my design on paper, I quickly came to regret it as I started to transcribe my design document into ANTLR grammar files. I found that it is paramount to get acquainted with ANTLR and its nooks and crannies as soon as possible to avoid frustrating moments when the solution to your problems is only an ANTLR option away. In many cases, what seemed to be erratic behavior was only my failure to realize the impact of ANTLR's options on the resulting Lexer and Parser.

Advice for New Teams

Get comfortable with ANTLR as soon as possible!

Appendix

BASICRuntime.java Implements the BASIC Runtime.

```
/*
 * BASICRuntime.java
 *
 * Created on December 3, 2006, 2:29 PM
 *
 * To change this template, choose Tools | Template Manager
 * and open the template in the editor.
 */

package basic;

import antlr.NoViableAltException;
import java.util.ArrayList;
import java.util.Stack;
import java.util.TreeMap;
import java.io.*;

/**
 *
 * @author Carlos G. Alustiza
 */
public class BASICRuntime {

    /** Creates a new instance of BASICRuntime */
    public BASICRuntime(int stackSize, String[][] lines) {

        //create a stack with the specified dimensions
        this.codeLines = lines;
        this.stackSize = stackSize-1;

    }
```

```
private String[] stack;
private int stackSize;
private String[][] codeLines;
protected int currLine = 1;
private Stack returnStack;
private TreeMap variableList;
private TreeMap loopStack;
protected String doNotExecBelongsTo = "";
protected boolean doNotExecute = false;

public void run() {

    //Here's where the action starts!

    //initialize the return stack
    this.returnStack = new Stack();

    //initialize the variable table
    this.variableList = new TreeMap();

    //initialize the loop table
    this.loopStack = new TreeMap();

    //initialize line counter
    this.currLine = 1;

    //initialize the parser
    String lineToRun = new String();

    //start processing lines, stop when reached end of stack or line = 0
    while (this.currLine != 0 && this.currLine <= this.stackSize)
        this.executeCurrentLine();
}

protected void doGOTO(int line){

    //check the global doNotExecute flag
```

```
    if (this.doNotExecute)
        return;

    if (line == 0){
        System.out.println("Program finished.");
        System.exit(0);
    }
    //move the program counter to the appropriate line

    //first of all find out if the line exists
    //Brute force approach
    boolean validJump = false;

    for (int i = 1; i<=this.stackSize; i++){

        String lineNo = this.codeLines[i][0];
        if (String.valueOf(line).equalsIgnoreCase(lineNo)){
            this.currLine = i;
            validJump = true;
            break;
        }

    }

    if (validJump == false){
        System.out.println("Jump to invalid line number. Last valid line: "
                +this.getOrigLineNumber(this.currLine));
        System.out.println("Program finished with errors.");
        System.exit(-1);
    }

}



protected void executeCurrentLine(){
    //very nasty method, but since it is not possible to modify the
    //underlying buffer of a ByteArrayInputStream or reattach the
```

```
//InputStream to the lexer / parser I had to do this
//There must be a better way, but could not find it / think of it

if (this.currLine != 0 & this.currLine <= this.stackSize){
    try{

        //get line to parse/execute
        this.lineToRun = this.codeLines[
                this.currLine][1].getBytes("UTF-8");

        //initialize lexer and parser
        this.myStream = new ByteArrayInputStream(this.lineToRun);
        this.inputStream = new DataInputStream(this.myStream);
        this.myLexer = new BASICLexer(this.inputStream);
        this.myLexer.myCallerLexer = this;
        this.myParser = new BASICParser(this.myLexer);
        this.myParser.myCaller = this;
        //parse the line

        this.myParser.exec_line();
        //clean up
        this.myParser = null;
        this.myLexer = null;
        this.inputStream = null;
        this.myStream = null;

    }catch(Exception e) {
        e.printStackTrace();
        System.out.println("RETURN without GOSUB in line: "
                +this.getOrigLineNumber(this.currLine));
        System.exit(-1);
    }

    //update the program counter
    //this.currLine++;
} else{
    System.out.println("Program finished.");
    System.exit(0);
```

```
    }
}
protected int getNextLine(int line) {

    int nextLine = 0; //default value, nothing else to execute

    if (line > 0){
        int tempCurLine = this.currLine;
        tempCurLine++;

        if (tempCurLine<=this.stackSize)
            nextLine = tempCurLine;
    }

    return nextLine;


}

protected void doGOSUB(int line){

    //check the global doNotExecute flag
    if (this.doNotExecute)
        return;

    //similar to the goto, but has to push the return address in the stack

    this.returnStack.push(
            getOrigLineNumber(this.getNextLine(this.currLine)));

    this.doGOTO(line);
}

protected void doRETURN()  {

    //check the global doNotExecute flag
    if (this.doNotExecute)
        return;
```

```
    //a simple goto, but popping the return address from the returnStack
    int returnLine;
    returnLine = (Integer)this.returnStack.pop();
    this.doGOTO(returnLine);
}

protected int getOrigLineNumber(int internalLineNo) {
    //given a line internal number (ie position in stack)
    //return the original line number (ie assigned by user)

    int originalLine = 0; //default value, means program finished
    if (internalLineNo != 0 & internalLineNo <= this.stackSize){
        originalLine = Integer.valueOf(this.codeLines[internalLineNo][0]);
    }

    return originalLine;


}



protected void doFOR(String ctrlVar, int initialVal, int endVal, int step) {

    //check the global doNotExecute flag
    if (this.doNotExecute)
        return;

    //first time we hit the for statement
    if (this.loopStack.containsKey(ctrlVar)==false){

        //create the BASICLoop object
        BASICLoop myLoop = new BASICLoop(ctrlVar, initialVal,endVal, step );
        myLoop.forAddress = this.currLine;

        //insert in the loop list
```

```
        this.loopStack.put(ctrlVar, myLoop);

        //in the cases the initial value and end values are not right
        if (myLoop.initialValue == myLoop.endValue){
            this.doNotExecute = true;
            this.doNotExecBelongsTo = ctrlVar;
        }

        if (myLoop.initialValue > myLoop.endValue & myLoop.step > 0){
            this.doNotExecute = true;
            this.doNotExecBelongsTo = ctrlVar;
        }

    }



    if (this.loopStack.containsKey(ctrlVar)==true) {
        BASICLoop myLoop = (BASICLoop)this.loopStack.get(ctrlVar);
        if (myLoop.currentValue > myLoop.endValue){
            int line = myLoop.nextAddress;

            //eliminate from stack, we do not need it around anymore
            this.loopStack.remove(ctrlVar);
            this.currLine = line;
            //jump to the line after the next
            //this.doGOTO(this.getOrigLineNumber(line));
        }else {
            //System.out.println("in loop");
            myLoop.currentValue+=myLoop.step;

            //advance the counter and execute
            this.currLine++;
            //this.executeCurrentLine();
        }

    }
}
```

```
protected void doNEXT(String ctrlVar) {


    //in case we got here because of a null initial FOR condition
    if (this.doNotExecute = false &
            this.doNotExecBelongsTo.equalsIgnoreCase(ctrlVar)){
        this.doNotExecBelongsTo = "";
        this.doNotExecute = false;
    }



    //the loop body has been executed
    if (this.loopStack.containsKey(ctrlVar)){

        //get the loop info
        BASICLoop tmpLoop = (BASICLoop)this.loopStack.get(ctrlVar);

        if (tmpLoop.nextAddress != 0 && tmpLoop.nextAddress ==
                this.getNextLine(this.currLine)){

            //this.doGOTO(this.getOrigLineNumber(tmpLoop.forAddress));
            this.currLine = tmpLoop.forAddress;
            return;
        }

        //have a FOR statement, but it is the first time we hit NEXT
        if (tmpLoop.nextAddress == 0){
            tmpLoop.nextAddress = this.getNextLine(this.currLine);
            this.loopStack.put(ctrlVar,tmpLoop);
            this.currLine = tmpLoop.forAddress;
            return;
            //this.doGOTO(this.getOrigLineNumber(tmpLoop.forAddress));
        }

        //have a for statement, but the next address is not the current
        //line, meaning duplicate control variables
```

```
        if (tmpLoop.nextAddress != this.getNextLine(this.currLine)){
            System.out.println("Unexpected NEXT statement. Line: " +
                    this.getOrigLineNumber(this.currLine));
            System.exit(-1);
        }

    }else { //found a NEXT statement, but no matching FOR statement
        System.out.println("NEXT statement not matched by FOR. Line: " +
                this.getOrigLineNumber(this.currLine));
        System.exit(-1);
    }

}

protected byte[] lineToRun = {' '};

protected BASICParser myParser;

protected BASICLexer myLexer;

protected ByteArrayInputStream myStream;

protected DataInputStream inputStream;

protected void doSTOP() {
    //simply stop the program
    this.currLine = 0;
    //this.executeCurrentLine();
}

protected void doREM() {

    this.currLine++;
}

protected void doAlphaAssign(String variable, String value) {
```

```
    //variable does not exist
    if (this.variableList.containsKey(variable)){
        BASICVariable tmpVar = (BASICVariable)
        this.variableList.get(variable);
        tmpVar.value = value;
        this.variableList.put(variable, tmpVar);
        this.currLine++;
        return;
    }else{
        BASICVariable newVar = new BASICVariable(variable, value);
        this.variableList.put(variable, newVar);
        this.currLine++;
        return;
    }

}

protected void doAlphaAssignConcat(String variable, String var2,
                                              String value) {



    //variable does not exist error!
    if (this.variableList.containsKey(variable)==false){
        System.out.println("Syntax error. Attempt to use value of undefined"
                +" variable. Line: " +
                this.getOrigLineNumber(this.currLine));

        this.currLine = 0;
        return;
    }else{
        if (variable.equalsIgnoreCase(var2)){
            BASICVariable tmpVar =
                    (BASICVariable) this.variableList.get(variable);

            String tmpValue = (String)tmpVar.value;
            tmpValue += value;
            tmpVar.value = tmpValue;
            this.variableList.put(variable, tmpVar);
```

```
            this.currLine++;
            return;
        }else {
            System.out.println("Syntax error inconsistent variables. Line: "
                    +this.getOrigLineNumber(this.currLine));

            this.currLine = 0;
            return;
        }
    }
}

protected void doNumAssign(String variable, double value) {



    //if variable exists
    if (this.variableList.containsKey(variable)){
        BASICVariable tmpVar =
                (BASICVariable) this.variableList.get(variable);
        tmpVar.value = value;
        this.variableList.put(variable, tmpVar);
        this.currLine++;
        return;
    }else {
        BASICVariable newVar = new BASICVariable(variable, value);
        this.variableList.put(variable, newVar);
        this.currLine++;
        return;
    }

}

protected void doPRINTAlphaVar(String variable){
    if (this.variableList.containsKey(variable)){
        BASICVariable tmpVar =
                (BASICVariable) this.variableList.get(variable);
        System.out.println(tmpVar.value);
        this.currLine++;
```

```
            return;
        }else{
            System.out.println("Variable not found " + variable +
                    " Line: "+this.getOrigLineNumber(this.currLine));

            this.currLine = 0;
            return;
        }

    }

    protected void doPRINTString(String theString) {
        System.out.println(theString);
        this.currLine++;
    }

    protected void doPRINTNumVar(String variable) {
        if (this.variableList.containsKey(variable)){
            BASICVariable tmpVar =
                    (BASICVariable) this.variableList.get(variable);

            System.out.println(tmpVar.value);
            this.currLine++;
            return;
        }else{
            System.out.println("Variable not found " + variable + " Line: "+
                    this.getOrigLineNumber(this.currLine));
            this.currLine = 0;
            return;
        }
    }

    protected void doNumAssignReplace(String var1, String var2,
            String operator, double value) {

        //variable does not exist error!
        if (this.variableList.containsKey(var1)==false){
            System.out.println("Syntax error. Attempt to use value of undefined"
```

```
            +" variable. Line: "+this.getOrigLineNumber(this.currLine));

        this.currLine = 0;
        return;
    }else{
        if (var1.equalsIgnoreCase(var2)){
            BASICVariable tmpVar =
                    (BASICVariable) this.variableList.get(var1);

            double tmpValue = Double.valueOf(tmpVar.value.toString());
            String op = operator.trim();
            switch(op.charAt(0)){

                case '+':
                {
                    tmpValue+=value;
                    break;
                }
                case '-':
                {
                    tmpValue-=value;
                    break;
                }
                case '*':
                {
                    tmpValue *=value;
                    break;
                }
                case '/':
                {
                    tmpValue /= value;
                    break;
                }
                default:
                {
                    System.out.println("Syntax error. Undefined operator."
                            + " Line: "+
                            this.getOrigLineNumber(this.currLine));
```

```
                }

            }

            tmpVar.value = tmpValue;
            this.variableList.put(var1, tmpVar);
            this.currLine++;
            return;
        }else {
            System.out.println("Syntax error inconsistent variables. Line: "
                    +this.getOrigLineNumber(this.currLine));

            this.currLine = 0;
            return;
        }
    }
}

protected double getVariableValueNum(String variable) {
    //variable does not exist error!
    if (this.variableList.containsKey(variable)==false){
        if (this.loopStack.containsKey(variable)==false){
            System.out.println("Syntax error. Attempt to use value of" +
                    " undefined" +
                    " variable. Line: "
                    + this.getOrigLineNumber(this.currLine));

            this.currLine = 0;
            return 0;
        }else{
            //it is a control var
            BASICLoop tmpVar = (BASICLoop)this.loopStack.get(variable);
            double value = Double.valueOf(tmpVar.currentValue);
            return value;
        }
    }else{
        BASICVariable tmpVar =
```

```
                (BASICVariable)this.variableList.get(variable);

            double value = Double.valueOf(tmpVar.value.toString());
            return value;
        }

    }


}
```

BASICVariable.java Implements a BASIC variable

```
/*
 * BASICVariable.java
 *
 * Created on December 1, 2006, 3:10 PM
 *
 * To change this template, choose Tools | Template Manager
 * and open the template in the editor.
 */

package basic;

/**
 *
 * @author Carlos G. Alustiza
 */
public class BASICVariable {

    /** Creates a new instance of BASICVariable */
    public BASICVariable(String name, Object value) {

        this.name = name;
        this.value = value;

    }



    protected String name;

    protected Object value;

}
```

BASICLoop.java Implements a FOR...NEXT loop

```
/*
 * BASICLoop.java
 *
 * Created on December 8, 2006, 12:52 AM
 *
 * To change this template, choose Tools | Template Manager
 * and open the template in the editor.
 */

package basic;

/**
 *
 * @author Carlos G. Alustiza
 */
public class BASICLoop {

    /** Creates a new instance of BASICLoop */
    public BASICLoop(String ctrlVar, int initialVal, int endVal, int step) {
        this.controlVar = ctrlVar;
        this.initialValue = initialVal;
        this.currentValue = initialVal;
        this.endValue = endVal;
        this.step = step;
    }

    public BASICLoop(String ctrlVar, int initialVal, int endVal) {
        this.controlVar = ctrlVar;
        this.initialValue = initialVal;
        this.currentValue = initialVal;
        this.endValue = endVal;
    }

    protected String controlVar;
```

```
    protected int initialValue;

    protected int endValue;

    protected int forAddress;

    protected int nextAddress = 0;

    protected int step = 1;

    protected int currentValue;

}
```

Main.java Implements the Boot component

```
/*
 * Main.java
 *
 * Created on December 1, 2006, 3:03 PM
 *
 * To change this template, choose Tools | Template Manager
 * and open the template in the editor.
 */

package basic;
import java.io.*;
import java.util.*;
/**
 *
 * @author Carlos G. Alustiza
 */
public class Main {

    /** Creates a new instance of Main */
    public Main() {
    }

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {


        // Get the name of the file from the command line
        String myFile = args[0];

        //Load the file, get the lines
        try{
            File myBasicFile = new File(myFile);
            BufferedReader myLines = new BufferedReader(new
```

```
                                        FileReader(myBasicFile));

codeStack = new TreeMap();
lastLine = 0;
String [] tmpLine;
String currentLine = null;
String myCurrentLine = null;

while ((currentLine=myLines.readLine())!=null){
    myCurrentLine = currentLine.trim();
    //Get the line number
    tmpLine = myCurrentLine.split(" ");

    if (tmpLine[0].equalsIgnoreCase(""))
        continue; //do not worry about empty lines

    Integer lineNo = new Integer(tmpLine[0]);
    if (lineNo > 0){
        //add line to map
        //update lastLine
        if (codeStack.put(lineNo,myCurrentLine.substring(
                                    tmpLine[0].length()))!=null){

            System.out.println("Duplicated lines. Last valid line: "
                    +lastLine);

            System.exit(-1);
        }
        lastLine = lineNo;
    }

}

//Create an array with the code lines and initialize the runtime

Set myKeys = codeStack.keySet();
String [][] lines = new String[myKeys.size()+1] [2];
Iterator myIter = myKeys.iterator();
```

```
            int iter = 1;
            while (myIter.hasNext()){
                Object key = myIter.next();
                lines [iter][0] = key.toString();
                lines [iter][1] = codeStack.get(key).toString();
                iter++;
            }

            //runtime initialization
            BASICRuntime runtime = new BASICRuntime(iter,lines);

            //start execution
            runtime.run();
        }catch (IOException IOex){
            System.out.println("Could not load BASIC file: "+args[0]);
            System.exit(-1);
        } catch (NumberFormatException NFex){
            System.out.println("Error while reading program. Line number not" +
                    " valid. Last valid line: "+lastLine);
        }

    }

    private static int lastLine;

    private static TreeMap codeStack;

}
```

grammar8.g Implements the BASIC grammar

```
//BASIC Grammar
//Author: Carlos G. Alustiza (cga2104)
//Programming Languages and Translators
//Columbia University - Fall 2006

header {
package basic;
}

class BASICParser extends Parser;
options {buildAST=true;k=3;}

{public BASICRuntime myCaller;}

//operators



logicalOperator:
        "and"
        |"or"
        |"not"
;

//print
print:
    print_string|print_num|print_alpha|print_func
;

print_alpha:
    (PRINT ALPHA_VAR) => PRINT a:ALPHA_VAR
{this.myCaller.doPRINTAlphaVar(a.getText());}
;

print_num:
    PRINT c:ID {this.myCaller.doPRINTNumVar(c.getText());}
```

```
;

print_string:
    PRINT b:STRING {this.myCaller.doPRINTString(b.getText());}
;

print_func:
PRINT b:NUM_FUNC {this.myCaller.doPRINTString(b.getText());}
;

exception // for print
    catch [RecognitionException ex] {
       System.out.println("Syntax Error. Print requires arguments." +
            this.myCaller.getOrigLineNumber(this.myCaller.currLine));
       System.exit(-1);
    }


//rules for lines
exec_line : (statement);
exception // for exec_line
    catch [RecognitionException ex] {
       System.out.println("Syntax Error.");
       System.exit(-1);
    }

statement:
        assignment
        |loop_next
        |loop_head
        |if_then
        |go_to
        |gosub
        |rtrn
        |comment
        |stop
        |print
;
```

```
//if I do not throw the exception here, the runtime goes nuts and signals errors
//that are not real (I went nuts before I found this out)

exception // for statement
    catch [RecognitionException ex] {
       System.out.println("Syntax Error. Statement cannot be recognized on line: "
+
                this.myCaller.getOrigLineNumber(this.myCaller.currLine));
       System.exit(-1);
    }


assignment: ASSIGNMENT (num_assignment
            |alpha_assignment)
;

num_assignment:
    (ID ASSN_OP NUMBER) => v:ID ASSN_OP n:NUMBER
{this.myCaller.doNumAssign (v.getText(), Double.valueOf(n.getText()));}
    | z:ID ASSN_OP f:NUM_FUNC
    {this.myCaller.doNumAssign (z.getText(), Double.valueOf(f.getText()));}
    |(ID ASSN_OP ID) => a:ID ASSN_OP b:ID o:OPERATOR c:NUMBER
{this.myCaller.doNumAssignReplace (a.getText(), b.getText(), o.getText(),
Double.valueOf(c.getText()));}
;

exception // for num_assignment
    catch [RecognitionException ex] {
       System.out.println("Syntax Error. Operator cannot be recognized on line: " +
                this.myCaller.getOrigLineNumber(this.myCaller.currLine));
       System.out.println ("Remember to separate operators (* + - /) from numbers
with spaces");
       System.exit(-1);
    }
```

```
alpha_assignment:
    (ALPHA_VAR ASSN_OP STRING) => v:ALPHA_VAR ASSN_OP s:STRING
{this.myCaller.doAlphaAssign (v.getText(), s.getText());}
    | x:ALPHA_VAR ASSN_OP y:ALPHA_VAR PLUS (WS)? z:STRING
{this.myCaller.doAlphaAssignConcat (x.getText(), y.getText(), z.getText());}
;



condition:
    (ID|ALPHA_VAR) (COMPARISON_OP|ASSN_OP) (NUMBER|STRING)
;

// if STEP is not added, then the rule is not recognized
loop_head:
    FOR_LOOP i:ID ASSN_OP iv:NUMBER TO_LOOP ev:NUMBER (STEP_LOOP sv:NUMBER)?
{this.myCaller.doFOR(i.getText(),
Integer.valueOf(iv.getText()),Integer.valueOf(ev.getText()),Integer.valueOf(sv.getT
ext()));}
;

loop_next:
    NEXT_LOOP v:ID
{this.myCaller.doNEXT(v.getText());}
;

if_then:
    IF condition (logicalOperator condition)* THEN statement
    {System.out.println("Found if_then: ");}
;

gosub:
    GOSUB v:NUMBER
    {this.myCaller.doGOSUB(Integer.valueOf(v.getText()));}
;

//It means return, but since it is a reserved word in Java, I had to modfy the name
rtrn:
    RET
```

```
{this.myCaller.doRETURN();}
;

go_to:
    GO_TO v:NUMBER
    {
        this.myCaller.doGOTO(Integer.valueOf(v.getText()));
}
;

stop:
    STOP
    {
        this.myCaller.doSTOP();
}
;

comment:
    REM
    {
        this.myCaller.doREM();
}
;


class BASICLexer extends Lexer;

options {k=3;filter=false;testLiterals=true;
caseSensitiveLiterals=false;}

{public BASICRuntime myCallerLexer;}

//define basic types and operators
protected NUMERIC : ('0'..'9');

protected LETTER: ('a'..'z'|'A'..'Z');

ID    options {testLiterals = true;}
```

```
        :       LETTER
                ( LETTER|NUMERIC )*
        ;

WS      : (' ')+
        {$setType(Token.SKIP);}
;

NEWLINE   :   "\r\n" // DOS
               | '\r'   // MAC
               | '\n'   // Unix
             { newline();
               $setType(Token.SKIP);
             }
          ;

//special symbols
LPAREN          : '(' ;
RPAREN          : ')' ;
STRING            : '"'!(~('"' | '\n')
            | ('"'! '"')
            )*
            '"'!
;

//TODO improve the NUMBER definition
NUMBER      :
           (MINUS)? INT ('.' INT)?
;

INT          :
       (NUMERIC)+
;

PLUS:
        '+'
;
```

```
MINUS:
        '-'
;

MUL   :
        '*'
;

DIV   :
        '/'
;

POW   :
    MUL MUL
;

OPERATOR: MATH_OP WS;

protected
MATH_OP :
        (PLUS {$setText("+");}
        |MINUS {$setText("-");}
        |MUL {$setText("*");}
        |DIV {$setText("/");}
)
;

//Alphanumeric variables only one character, no numbers or non-alphabetic chars.
End with $ sign
ALPHA_VAR   : LETTER "$";



//define language constructs
ASSIGNMENT
      : "let"
      ;

FOR_LOOP
        : "for"
```

```
        ;

TO_LOOP
        : "to"
        ;

STEP_LOOP
        : "step"
        ;

NEXT_LOOP
        : "next"
;

IF
        : "if"
;

THEN
        : "then"
;

REM   :
        "rem"
;

GOSUB   :
            "gosub"
;

RET     :
            "return"
;

GO_TO    :
            "goto"
;
```

```
STOP     :
            "stop"
;

ASSN_OP :'=';

RH_ASSN: (NUMERIC)+;

COMPARISON_OP    :
       "<="
      | "<"
      | ">="
      | ">"
      | "<>"
;

//functions

NUM_FUNC :
            ABS
            |ACS
            |ASN
            |ATN
            |CHR
            |COS
            |EXP
            |LN
            |SIN
            |SQR
;


protected
ABS :
    "abs" (WS)? LPAREN (WS)? v:NUMBER (WS)? RPAREN
    {   Double val = java.lang.Math.abs(Double.valueOf(v.getText()));
        setText(val.toString());}
;
```

```
protected
ACS :
    "acs" (WS)? LPAREN (WS)? v:NUMBER (WS)? RPAREN
    {   Double val = java.lang.Math.acos(Double.valueOf(v.getText()));
        setText(val.toString());}
;

protected
ASN :
    "asn" (WS)? LPAREN (WS)? v:NUMBER (WS)? RPAREN
    {   Double val = java.lang.Math.asin(Double.valueOf(v.getText()));
        setText(val.toString());}
;

protected
ATN :
    "atn" (WS)? LPAREN (WS)? v:NUMBER (WS)? RPAREN
    {   Double val = java.lang.Math.atan(Double.valueOf(v.getText()));
        setText(val.toString());}
;

protected
CHR :
    ("chr$" (WS)? LPAREN (WS)? NUMBER) => "chr$" (WS)? LPAREN (WS)? v:NUMBER (WS)?
RPAREN
    {   char [] val = java.lang.Character.toChars(Integer.valueOf(v.getText()));
        setText(String.valueOf(val[0]));}
   |"chr$" (WS)? LPAREN (WS)? a:ID (WS)? RPAREN
    {char [] val =
java.lang.Character.toChars((int)this.myCallerLexer.getVariableValueNum(a.getText()
));
        setText(String.valueOf(val[0]));}
;

protected
COS :
    ("cos" (WS)? LPAREN (WS)? NUMBER) => "cos" (WS)? LPAREN (WS)? v:NUMBER (WS)?
RPAREN
    {   Double val = java.lang.Math.cos(Double.valueOf(v.getText()));
```

```
        setText(val.toString());}
    |"cos" (WS)? LPAREN (WS)? a:ID (WS)? RPAREN
    {Double val =
java.lang.Math.cos(this.myCallerLexer.getVariableValueNum(a.getText()));
     setText(val.toString());}
;

protected
EXP :
    "exp" (WS)? LPAREN (WS)? v:NUMBER (WS)? RPAREN
    {   Double val = java.lang.Math.exp(Integer.valueOf(v.getText()));
        setText(val.toString());}
;

protected
LN :
    "ln" (WS)? LPAREN (WS)? v:NUMBER (WS)? RPAREN
    {   Double val = java.lang.Math.log(Integer.valueOf(v.getText()));
        setText(val.toString());}
;



protected
SIN :
    "sin" (WS)? LPAREN (WS)? v:NUMBER (WS)? RPAREN
    {   Double val = java.lang.Math.sin(Double.valueOf(v.getText()));
        setText(val.toString());}
;



protected
SQR :
    "sqr" (WS)? LPAREN (WS)? v:NUMBER (WS)? RPAREN
    {   Double val = java.lang.Math.sqrt(Double.valueOf(v.getText()));
        setText(val.toString());}
;

PRINT :
    "print"
```

```
;


//special case to exit during tests
EXIT    : '.' { System.exit(0); } ;
```