SIMPLEX

SYNTAX FOR INTERNATIONAL MONETARY,
PROPERTY, AND LIQUIDITY EXCHANGE

FINAL REPORT

December 19, 2006

stc2104 | Steven Chen
gch2102 | Gilbert Hom
kxj1| Kelvin Jiang
ehz2101 | Eric Zhang

TABLE OF CONTENTS

INTRODUCTION

The financial services industry is dynamic and extremely competitive. From making investment choices to providing advice to Fortune 500 corporations, one's reputation depends on the ability to make wise, precise and timely decisions. SIMPLEX is designed to assist financial professionals quickly produce accurate and precise financial analysis under intense deadline pressure.

SIMPLEX allows financial professionals to focus on analysis rather than on intricacies in programming languages such as Java and C++. The language is syntactically intuitive and uses terminology and conventions familiar to professionals in the industry. Furthermore, users of SIMPLEX will appreciate the built-in features which again, allow the user to focus on the analysis and not on formatting output, memorizing formulas, determining data types or learning a significant amount of syntax. Overall, SIMPLEX is a flexible and intuitive programming language which can be used across many platforms and serves as a powerful tool for financial professionals.

BACKGROUND

Financial models serve as the foundation upon which financial analysis occurs. Financial models are used to calculate and/or estimate financial figures. They can be used to predict future earnings of a company, the result of a potential merger/acquisition, and the amount of debt a company can take on, among other things. Good models are designed in such a fashion that, should a user change an initial assumption, the change in the assumption should flow through the model without having to update anything else. Furthermore, good models can be easily modified and expanded to provide greater functionality and flexibility.

MOTIVATING SCENARIOS

An investment banking analyst has been asked to determine the pro forma effect of the merger between two large media companies. A homeowner wants to determine the best mortgage that fits his/her financial situation. A student wants to keep track of her student loans and the payments needed to repay those loans. An entrepreneur needs to estimate the growth of the sales of his company for a presentation to potential investors. An economist wants to determine the effect of a rise in interest rates by the Fed. A foreign exchange trader wants exploit currency arbitrage situations. These are all scenarios in which SIMPLEX will be able to provide a path to an answer or a solution.

While Microsoft Excel is powerful and considered the industry standard when it comes to financial analysis, it is fairly expensive. For students and young entrepreneurs who want to perform their own financial analysis, Microsoft Excel is most likely unaffordable. SIMPLEX will give its users the power to create everything from expansive financial models to quick financial calculations at virtually no cost. Overall, SIMPLEX serves as an economical alternative to Microsoft Excel without compromising any features required to generate meaningful financial analysis.

FUNDAMENTAL FEATURES

Data Types

A significant feature of SIMPLEX is the use of different currencies as data types. For example, US Dollars, Euros, Japanese Yen and other currencies will be set as data types. The reason motivating this feature is to ensure that different currencies cannot be mathematically manipulated without first applying the appropriate currency exchange rate. This ensures apples-to-apples comparisons. In addition SIMPLEX will include a data type called 'rate.' Users will program rates—such as interest rates—as 7.50 rather than 0.075. It is more intuitive to think of rates in terms of their actual number (7.50%) rather than their decimal equivalent (0.075).

In order to keep track of time, SIMPLEX will support 'years,' 'months' and 'days' data types. Lastly, for all other purposes, the programming language will have a generic string and number data type.

Currency Manipulation

SIMPLEX will automatically import currency exchange rate data in order to perform manipulations between currencies. This is a valuable feature because it prevents the user from having to enter all applicable exchange rate ratios into his/her program. A unique feature of SIMPLEX will include support to quickly typecast currencies. For example, if it was necessary to convert a value from US Dollars to Euros, the user could simply type `(Euros)` in front of a US Dollar variable name. This automatically applies the appropriate exchange rate to the value of the variable being converted.

Appropriately Formatted Output

In order to differentiate between currencies when values are displayed, the accompanying symbol (e.g. $, ¥, €, £, etc.) is also printed. Rather than having a minus sign in front of negative values, which are often small and hard to see, negative values will be shown with parenthesis around them. Also, values being printed will always have the appropriate amount of significant digits shown. For example, US Dollars are represented as having 2 decimal places and never more. This feature ensures that no superfluous data is shown.

Built-In Functions

Due to the mathematical complexity of some formulas, SIMPLEX will contain several frequently used functions including payment, present value, future value and internal rate of return functions.

Mathematical Expressions

SIMPLEX offers the convenience of being able to perform and evaluate mathematical expressions through the use of operators, not procedures and functions. This mimics the format of evaluating mathematical expressions in graphical calculators and Microsoft Excel. Having this feature focuses the user's attention to the program and not syntax. For example **`Math.pow(3,2)`** is the Java operation to raise 3 to the second power. The syntax to perform the same operation in SIMPLEX is **`2^3`**.

User-Defined Functions

A rudimentary equations solver will be incorporated into the compiler. Users can use this solver to define functions which take parameters of linear equations and solve for the missing variable. This way, long and common strings of arithmetic can be condensed into a user defined function. The user can then call the function in the program, leaving one of the arguments undefined, and the function will return the value of the missing variable.

Portability

An essential component of SIMPLEX is its ability to be run on several computing platforms. To achieve this feature, SIMPLEX code is translated into Java, which is then able to run on computers with the Java Virtual Machine installed.

LANGUAGE TUTORIAL

INTRODUCTION

Here is how the "Hello World" program is formulated in SIMPLEX.

```
void main()
{
      print ("Hello World");
}
```

The main portion of all SIMPLEX programs must be contained in the `void main()` method. The `print` command allows you to print items to screen. Any user defined functions must occur prior to the `main` method.

VARIABLE DECLARATION AND INITIALIZATION

As will be mentioned in the Language Reference Manual later on, there are data types that can be used in SIMPLEX. Variables must be declared first before they can be used. However, declaration and initiation cannot occur in the same statement.

There are several currencies available to the user in SIMPLEX. Please refer to the Language Reference Manual for the slate of options. The following is an example of how to declare a variable of YEN type called "myAmount" and how to initialize it to 100:

```
YEN myAmount;
myAmount = 100;
```

The declaration of the generic number data type follows similarly. For example:

```
number thisNumber;
thisNumber = 3.14;
```

Again, the declaration of the string data type follows similarly. For example:

```
string myName;
myName = "Steven Chen";
```

The rate data type is fairly unique to SIMPLEX. Rates are initialized using the actually percentage amount. For example, if one were to desire a variable containing 5% (0.05 in real number terms), this would be the code:

```
rate myInterestRate;
myInterestRate = 5;     // refers to 5% (or 0.05)
```

ITERATIVE STATEMENTS

The `while` and `for` loop statements are available for creating iterative statements. The syntax to these statements is similar to that of C, C++ and Java. Below are two examples illustrating these statements.

```
number a;
a = 0;
while (a > 5)
{
```

```
        a++;
}


number a;
for (a = 0; a < 5; a++)
{
        print “Hi!”;
}
```

CONDITIONAL STATEMENTS

No different from most other languages, if and else statements are available to the user when conditions need to be evaluated. Conditional statements can be nested, thus allowing the user to make multiple conditional statements. Conditional statements are illustrated in the example below:

```
if (a > 5)
{
        print (“a is greater than 5”);
}
else
{
        print (“a is less than or equal to 5”);
}
```

USER INPUT

The `input` function is a pre-defined function in SIMPLEX that allows the user to quickly and easily obtain input from the user from the command line. The argument of the `input` function is the variable in which the data the user enters in the command line is to be stored. For example, if the program were to obtain user input on a number, the following would be an example of how to achieve this:

```
number userInput;

print (“User: Please input a number”);
input(userInput);
```

PROGRAM OUTPUT

The `print` function is a pre-defined function in SIMPLEX that allows the user to quickly output information to the command line. The `print` function will automatically format currencies to show the appropriate symbol or abbreviation along with 2 decimal points. The `print` function is capable of handling all data types in SIMPLEX. The following is an example of the `print` function printing out an entire line with several data types

```
number a;
USD b;
string c;
rate d;

a = 4.1;
b = 9.3311;
c = “steve”;
d = 3;

print ( a + “ “ + b + “ “ + c + “ “ + d);
```

The output to this print statement is

```
4.1 $9.34 steve 3.0%
```

USER-DEFINED FUNCTIONS

SIMPLEX gives the user the ability to create user-defined functions. As mentioned previously, these functions must occur prior to the `main` method being called. Functions must have associated with them a name (identifier), argument(s) and a return type. The following is an example of a function:

```
USD myFunction (USD amt, rate r)
{
      USD temp;
      temp = amt * r;
      return temp;
}
```

CASTING CURRENCIES

Another fairly unique aspect to SIMPLEX is the ability to 'cast' currencies, rather than apply a conversion factor. In order to convert between variables of different currency types, place the new currency type in brackets prior to the variable name. For example, to convert a variable of USD to YEN:

```
USD a;
YEN b;

a = 5.50;          // $5.50
b = (YEN)a;        // converts $5.50 into Yen equivalent and stores in b
```

COMMENTS

As with nearly every programming language, the user has the option to place 'comments' in code. Comments are ignored by the compiler and can be single line or multiple lines. Single line comments start with two slashes: `//`. Multiple line comment lines begin with `/*` and end with a `*/`.

LANGUAGE MANUAL

OVERVIEW

SIMPLEX (Syntax for International Monetary, Property, and Liquidity Exchange) is a light-weight, C-like language that greatly simplifies the development of financial applications. It provides special money, date, and percentage types that are required by almost every financial system, and also makes software development more accessible to financial analysts through more intuitive expression and operator syntax. It is portable, compact, and easy to learn. SIMPLEX programs are case sensitive and can be written easily using any ASCII text editor.

LEXICAL CONVENTIONS

There are 5 types of tokens in SIMPLEX. They are: identifiers, keywords, constants, operators and separators. Identifiers must be separated by whitespace and are greedy, meaning they consume as much input as they can match. The details of each type of token are discussed below.

Whitespace

Whitespace is ignored by the SIMPLEX compiler. The purpose of whitespace is to separate out different tokens and allow users to follow the code in an easier fashion. Whitespace includes indentations, tabs, spaces and line terminators (including support for DOS, UNIX and MAC standards).

Comments

Comments are also ignored by the SIMPLEX compiler. Single-line comments begin with `//` and end at the end of the line. Multiple-line comments begin with `/*` and continue until `*/` is reached.

Identifiers

Identifiers in SIMPLEX are constructed using any alphanumeric combination of characters and the underscore character but cannot begin with a numerical digit. Furthermore, identifiers cannot take the same name as keywords. As mentioned earlier, SIMPLEX is case sensitive; thus, upper and lower case characters are different from each other.

Keywords

There is a small set of words reserved in SIMPLEX for essential functions. These keywords cannot be used as names for identifiers and include the following:

```
if          for         while       else        number
rate        USD         EUR         CAD         GBP
AUD         CHF         CNY         MXN         SOS
YEN         year        month       day         void
continue    break       return      string      main
```

Please note that this list of keywords includes 10 standard currencies described later.

Number Constants

A number is constructed with digits and can be followed, optionally, by a decimal point and more digits. Scientific notation is not accepted. Unlike many other programming languages

such as Java and C++, the user does not have to differentiate between integers and floating point numbers.

String Constants

A string is constructed with any combination of ASCII characters enclosed by double quotes.
For example, `"Steven Chen"` is considered to be a string constant since it is a series of ASCII characters surrounded by double quotes on both the left and right side. Certain characters must be 'escaped' meaning that a backslash must be placed immediately to the left of the character. Characters that must be escaped include double quotes ("), newlines, backslash characters and tabs.

Separators

The following characters are used in SIMPLEX as separators:

`{}` – Code block separator
`()` – Grouping separator and parameter list separator
`;` – Statement Delimiter
`,` – List Delimiter

DATA TYPES

SIMPLEX requires explicit type specification— meaning that identifier types must be declared prior to the compilation of the program. The variable is declared in the following manner:

```
data-type identifier;
```

The variable can also be initialized with a constant value on the same line as the declaration:

```
data-type identifier = value;
```

There are five 'classes' of data types: currencies, numbers, dates, rates, and strings. Within each class are the actual data types. The programmer may only declare variables with specific data types; however, the result of operations on these variables and constants are defined by the data type classes present in the operation rather than the data type itself. This will be explained in further detail in the 'Expressions' section, below.
If a variable is cast to a data type within the same class, the appropriate conversion is applied. For example, if USD are cast to CNY, the result is a CNY value with a value that is 8 times that of the original. If the variable is cast to a data type in a different class, the value is first cast to a number and then from a number to the desired data type. Since the conversion factor to and from a number is 1 to 1, this directly converts a numerical value of any type to the same value of another type.

Currencies

One of the unique aspects of SIMPLEX is the use of currencies as data types. The following 10 currencies are the standard currencies included with SIMPLEX. Later versions of SIMPLEX will include additional currencies.

```
USD   YEN   EUR   CAD   GBP
AUD   CHF   CNY   MXN   SOS
```

Currency values are constructed the same way as number constants (described in the Lexical Conventions section). The user does not have to include the currency symbol (e.g. $, ¥, €, £, etc.) since the compiler will take care of this.

Number

Numbers consist of signed 64-bit floating point numbers. The data type of a number variable is `number`.

Dates

There are three Date data types: `year`, `month` and `day`. They can be initialized the same way as a number and have built in casts defined: a year casts to 12 months and 365 days. A month casts to 30 days.

Rate

A fairly unique feature to SIMPLEX is support for a 'rate' data type. For example, 5% is typically entered into programming languages as 0.05. However, SIMPLEX allows the user to simply enter the rate as `5%` (The percent operator divides a number by 100, and the percentage is still stored internally as 0.05). Rate values are constructed the same way as number constants (described in the Lexical Conventions section). The Rate data type is identified by `rate`.

String

The `string` data type allows the programmer to store arbitrary sequences of characters in a variable. String variables are initialized with string constants (described in the Lexical Conventions section).

EXPRESSIONS

The following section discusses expressions used in SIMPLEX. Expression operators are listed in order of highest to lowest level of precedence. Operators in the same section are considered to have equal precedence and evaluate from left to right. Note that the operators can not be applied to arbitrary data types: for example, it is not clear what the result of adding a date and a currency should be. To resolve this problem, the language will have a compatibility table that tells the compiler which operations are legal. Operations on data type combinations not in the compatibility table are allowed but produce a warning. When this happens, the values are both first converted to numbers, and the operation is performed.

Primary Expressions

A primary expression is the simplest form of an expression and is typically used to represent a single value. Primary expressions include identifiers, constants and procedure calls. Furthermore, parenthesized expressions are considered to be primary expressions. Primary expressions are of highest precedence in SIMPLEX.

Unary Expressions

The next level of operator precedence in SIMPLEX consists of unary expressions. Unary expressions consist of one operator and an identifier. Operators include: `!`, `++`, `--`, `%`, `unary +`, `unary -`, and `(type)`. These operators are described below.

`!expression`

The result of this unary expression is the logical negation of the expression. Thus, the negation returns a one when the expression returns a zero and returns a zero when the expression is non-zero.

`expression++`

The result of this unary expression is addition of 1 to the expression. This operator can only be applied to number, currency, rate and date data types. This operator mimics the functionality of the post-increment operator in C. A pre-increment operator is not available in SIMPLEX.

`expression--`

The result of this unary expression is subtraction of 1 to the expression. This operator can only be applied to number, currency, rate and date data types. Again, a pre-increment version of this operator is not defined.

`expression%`

Unlike most programming languages where the `%` symbol is used for the modulus operation, SIMPLEX reserves the `%` symbol as denoting the division of the expression by 100. This expedites the conversion of numerical percentages to decimal values, a very common operation in financial applications.

`+expression`

The result of the unary + operator is the expression itself.

`-expression`

The result of the unary - operator is the negation of the expression.

`(type)expression`

The result of the cast operation is the conversion of the data type of the expression into the data type specified in the parenthesis. Casting allows for conversions between data types of the same class. For example, an expression of type month can be cast into a day since both month and day are in the date data type class. When the programmer asks for an undefined cast (casting a month into USD, for example) the value is simply cast into a number type, then to the desired data type. The compiler may produce a warning if this occurs.

Multiplicative Operators

The three multiplicative operators, `*` for multiplication, `/` for division, and `^` for exponentiation, can be used to perform multiplicative operations. They are grouped left to right and can be applied to numbers, rates and currency data types.

Additive Operators

The two additive operators, `+` for addition and `-` for subtraction, can be used to perform additive operations. They are grouped left to right and can be applied to numbers, rates and currency data types. The addition operator can also be used to concatenate strings.

Assignment Operators

The assignment operator, "=", stores the value returned by the expression on its right side into the identifier on its left side. It has the lowest precedence, and is associative from right to left. This allows multiple assignments to be made on the same line, in the following manner:

```
identifier1 = identifier2 = identifier3 = expression;
```

In this statement, the low precedence and right to left associatively of the assignment operator ensures that the entire expression is evaluated first. Next, the value of the expression is assigned to identifier3, then identifier2, then identifier1.

Relational and Equality Expressions

The following table summarizes the 6 relational operators available.

Operator	Description
>	Greater Than
<	Less Than
>=	Greater Than or Equal To
<=	Less Than or Equal To
==	Equal To
!=	Not Equal To

Relational and equality expressions are constructed as follows:

```
expression operator expression
```

Relational expressions return either zero or one, indicating whether the expression was false or true, respectively.

Logical Expressions

There are two logical operators available in SIMPLEX. The logical AND operator, denoted by `&&` takes higher precedence than the logical inclusive OR operator, which is denoted by `||`. These operators are generally applied to relational and equality expressions, and other logical expressions. The values are evaluated, and the logical expression returns true or false (1 or 0, respectively). Logical expressions may be grouped with parenthesis like any other binary operation. The exclusive OR operator is not defined. Logical expressions treat zero as false, and non-zero values as true.

Operator Shorthand

All additive and multiplicative operators have a shorthand form that allows for easy manipulation of the result variable in the expression. For example, to add 5 to the variable *x*, one could write `x = x+5`. The shorthand notation allows the user to write `x += 5` instead. This shorthand works with `+`, `-`, `*`, `/`, and `^`.

OPERATOR PRECEDENCE

The following table shows the precedence of operators in SIMPLEX from highest to lowest:

Operators	Associativity
`(`*`expression`*`)`	Left to Right
`!, ++, --, %, unary +, unary -, (`*`type`*`)`	Right to Left
`^, *, /`	Left to Right
`+, -`	Left to Right
`<, <=, >=, >, ==, !=`	Left to Right
`&&`	Left to Right
`||`	Left to Right
`=, +=, -=, *=, /=`	Right to Left

STATEMENTS

A statement is a line of code which is terminated by a semicolon and represents an executable instruction to the compiler. There are several types of statements, including assignment, jump, procedure call, and return statements. Compound statements—blocks of multiple statements, are also possible.

Assignment Statements

The assignment statement is simply an assignment expression followed by a semicolon. For an example, see the 'Assignment Expression' section above.

Jump Statements

The `break` statement is used to break the inner most loop during execution. The break statement can be used in both `while` and `for` loops. The statement is invoked by simple typing:

```
break;
```

The `continue` statement is used to exit out of the current iteration of a loop. The break statement can be used in both `while` and `for` loops. The statement is invoked by simply typing

```
continue;
```

Subprocedure Call Statements

Subprocedures, both user-defined and built-in, can be called by typing the name of the function, an open parenthesis, a list of parameters, and a close parenthesis.

Return Statements

The `return` statement is used to return a value to the caller of a subprocedure. The statement takes the following form:

```
return (expression);
```

The value of the expression is returned to the caller of the function. If the function has a return type, then it *must* have at least one return statement in its top level code block.

Statement Blocks

Statement blocks are used when more than one statement or compound statement should be considered a single statement overall. A left brace bracket is used to denote the beginning of a block and a right brace bracket used to denote the end of a block.

```
{
        statement1
        statement2
        .
        .
        .
}
```

In the following compound statements, 'statement' can be either a single statement, or a statement block.

Conditional Statements

Conditional statements are used to control the flow of a program. A simple conditional statement takes the following form:

```
if (expression) statement
```

or

```
if (expression) statement1 else statement2
```

In the first form, if the expression returns a non-zero value, then the statement is executed. The second form adds one more component to the conditional. Like before, if the expression evaluates to a non-zero value, `statement1` is executed; however, `statement2` is executed otherwise. Conditional statements can be nested, allowing for special syntax like 'else if'.

Iterative Statements

Iterative statements are used for executing a certain statement or group of statements for multiple iterations known as loops.

`while` loops are used to execute statements based on a condition. They take the following form:

```
while (expression) statement
```

As long as the expression evaluates to non-zero value, the statement is repeated until the condition returns a zero value. Infinite loops occur when the condition never evaluates to false.

`for` loops are used to execute based on number of iterations. `For` loops take the following form:

```
for (expression1 ; expression2 ; expression3) statement
```

The first expression, `expression1` is used to initialize the loop. The second expression, `expression2` is used to specify a testing condition to exit the loop. The third expression, `expression3` is used to increment the variable initialized in `expr1`, which is performed after every iteration of the `for` loop. All three expressions must be present.

PROCEDURES

Procedures take the following form in SIMPLEX:

```
type-specifier procedure-identifier (identifier-list)
{
   statement
}
```

The type-specifier is the data type that the procedure should return. The procedure-identifier is an identifier that names the procedure. The identifier-list is an optional list of arguments (separated by commas) that are passed by value.

PROGRAM STRUCTURE

SIMPLEX programs can be simply written in ASCII text files. The structure of a SIMPLEX program is as follows: global variables should be declared at the beginning of the file. Followed by the global variables should be procedures. Lastly, a procedure called `main` serves as the starting point for program execution.

PROJECT PLAN

PROCESS OVERVIEW

The initial planning of the project was fairly smooth. We all pitched to each other a potential idea for a programming language and ultimately settled on one that we felt would be interesting and fairly unique, as there are were no other financially-focused programming languages popular on the market.

The specification, development and testing stages were much different. Very early on, we all realized the strengths and weaknesses that made us unique and valuable to the team. Some of us have worked on large software engineering projects in the past, some of us had strong Java and programming skills and some of us were well organized and brought other intangibles. Because of each person's unique skill set, we quickly picked up our own 'specialties' and would be known as that 'go-to guy' when it came to that aspect of the project.

Thus, during the specification, development and testing stages, one person would usually dictate what he wanted, send the group an email detailing his idea and then if anyone had an issue with that idea, a discussion would then form. Discussions usually took place during our weekly meetings (typically Monday night) or right after class on Tuesdays and Thursdays during lunch.

PROGRAMMING STYLE

ANTLR	
Convention	**Description**
Names in Lexer	Names are to be entirely capitalized (e.g. ASTERISK, COLON)
Underscores in Lexer Names	Names in Lexer that are conceptually more than one word will be joined by underscores (e.g. L_PAREN, G_THAN)
Names in Parser of CamelCase Format	Names are to be completely lower case unless name is conceptually more than one word, where a capitalized letter is allowed (e.g. functionDef, argumentList)
Parser Rules	Attempt to place each component of each rule on a separate line for readbility

JAVA	
Convention	**Description**
Commenting	Use comments to separate sections of code, embed to do lists, explain code, illustrate what does not work, show who worked on code
Left Braces '{'	Occur on same line as if, while, for, etc. statements
Right Braces '}'	Occur on own line
Operator / Operand Spacing	Place spaces between operators and operands
Standardized Abbreviations	If something is to be abbreviated, maintain the fact that it is abbreviated and use the same abbreviation throughout (e.g. expr, argList)
Names in CamelCase Format	Variable and method names should follow standard CamelCase Format
Multiple Lines	In the case of a long expression, place on multiple lines to prevent need to scroll right
Import Statements	Place in first lines of source file
Write Methods as Appropriate	Writing methods allows for the re-using of fuctions, allowing for less confusion and less duplication of code
Method Headers	Method Headers take form: `<access> returnType methodName (argList)`
Line Lengths	Avoid line lengths greater than 90

PROJECT TIMELINE

Date	Task
9/7/2006	Form Group
9/11/2006	Form Idea for Language
9/26/2006	Whitepaper
10/7/2006	ANTLR Lexer Completion
10/19/2006	Language Reference Manual
10/20/2006	Begin Work on Tree Walker and IR Classes
10/24/2006	ANTLR Parser Completion
11/8/2006	Begin Regression Test Suite Evaluation
12/15/2006	Total Project Completion
12/18/2006	Final Presentation
12/19/2006	Final Report

ROLES AND RESPONSIBILITIES

Team Member	Primary Responsibilities
Steven Chen	Whitepaper, LRM, ANTLR Lexer and Parser, Final Report
Gilbert Hom	Testing, Final Presentation
Kelvin Jiang	Testing, Intermediate Representation Classes
Eric Zhang	Tree Walker, Testing, Intermediate Representation Classes,

SOFTWARE DEVELOPMENT TOOLS

Eclipse (with ANTLR Plugin)

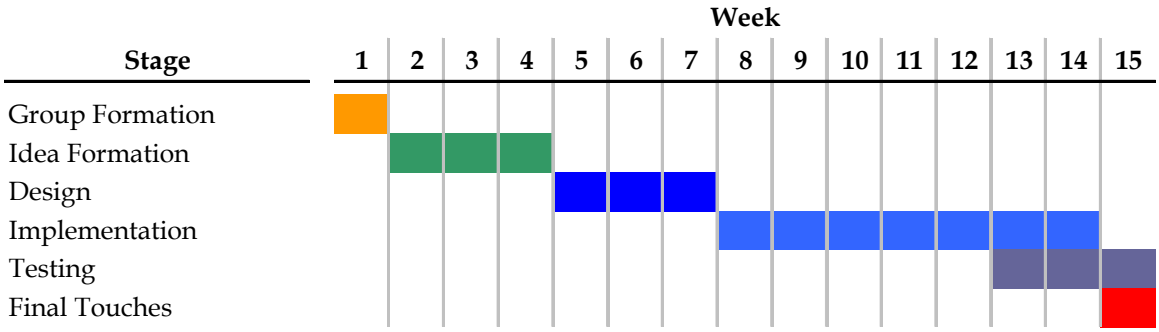
We chose to use the Eclipse IDE (**http://www.eclipse.org/**) because it is a free, easy to use tool with an ANTLR plugin (**http://antlreclipse.sourceforge.net/**) available. Along with the typical IDE features (e.g. syntax highlighting, automatic indenting, etc.), we were able to manage the many Java classes that made up our project. The debugger in Eclipse also served as an invaluable resource.

Subversion

Subversion is a version controlling system similar to CVS. Subclipse (**http://subclipse.tigris.org/**) is an Eclipse plugin that integrates Subversion with the Eclipse IDE. Subclipse allowed us to check out different files at the same time to ensure that everyone could actively work on a separate part of the project without issue.

PROJECT LOG

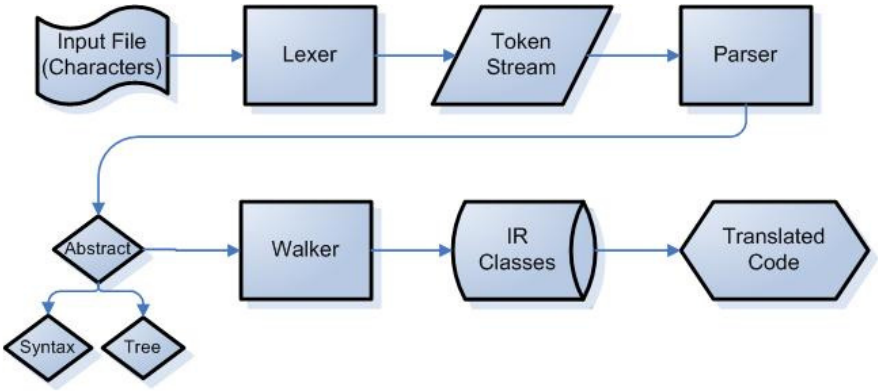
	Week	Phase	Tasks In Progress
1	9/5 - 9/10	Group Formation	Group Formed
2	9/11 - 9/17	Idea Formation	Decide on a language and its specialty
3	9/18 - 9/24	Idea Formation	Whitepaper Write-Up
4	9/25 - 10/1	Design	Whitepaper Write-Up, Language Design
5	10/2 - 10/8	Design	Language Reference Manual Write-Up; ANTLR Lexer
6	10/9 - 10/15	Design	Language Reference Manual Write-Up; ANTLR Lexer/Parser
7	10/16 - 10/22	Design	Language Reference Manual Completed; Parser/Tree Walker
8	10/23 - 10/29	Implementation	Tree Walker/Java IR Classes
9	10/30 - 11/5	Implementation	Java IR Classes
10	11/6 - 11/12	Implementation	Java IR Classes (*Elections Weekend*)
11	11/13 - 11/19	Implementation	Java IR Classes
12	11/20 - 11/26	Implementation	Java IR Classes (*Thanksgiving Weekend)*
13	11/27 - 12/3	Implementation/Testing	Java IR Refinements, Error Checking
14	12/4 - 12/10	Implementation/Testing	Java IR Refinements, In-Depth Testing
15	12/11 - 12/19	Testing/Report Completion	More Testing, Final Report Write-Up, Presentation Creation



ARCHITECTURAL DESIGN

OVERVIEW

The SIMPLEX compiler has four major components—the Lexer, the Parser, the Tree Walker, and the intermediate representation (IR) classes. Their interactions are shown in the diagram below:



The Lexer is the first component to see the input file. Its role is to convert the string of characters in the file into a series of tokens. It eliminates whitespace and comments, and sends the resulting tokens to the parser.

The parser then reads in the series of tokens and generates an abstract syntax tree representation of the file. In the tree, the unnecessary punctuation is removed and the statements and expressions are connected in the proper order. It is in this step that syntax rules like left associativity and operator precedence are handled. Once the expressions and statements have been parsed into the AST, the ambiguity in certain language constructs is resolved.

The next step is walking the tree, which is done by the SIMPLEX tree walker. The walker takes an abstract syntax tree as input and constructs an intermediate representation with the IR classes. It is in this stage that the semantics of the language constructs are resolved—variable declarations nodes on the tree form 'Decl' objects in the IR, expressions become 'Expr' objects, and so on. Semantic errors such as the comparison of incompatible types are handled at this stage.

The intermediate representation classes can each generate their corresponding Java constructs. When the entire program has been converted into the IR, the generate function is called on the top level statement which subsequently calls each of the generate functions recursively, and the entirety of the java code is output.

SIMPLEX LEXER

To a compiler, the initial input file is nothing more than a long string of characters without any apparent syntax or semantic meaning. The Lexer is a simple regular expression matcher that gives names to certain strings of characters in a file. The Lexer uses a system of rules and a simple state machine system to define what character strings become which tokens. For example,

```
OPEQUALS:          ("+=" | "-=" | "/=" | "*=" | "^=");
```

defines a rule that matches any of the string between double quotes, and names the token 'OPEQUALS' for use later on with the parser and tree walker.

The Lexer can also help the compiler eliminate input that will not be useful in later stages such as whitespace. This is done with a rule that looks like:

```
WHITESPACE
      :     (' ' | '\t' | '\f')+
            { $setType(Token.SKIP); }
      ;
```

The rule matches tabs, spaces, and form feed characters, and the code within the braces tells the Lexer to omit those characters from the output token stream.

SIMPLEX PARSER

The Parser, like the Lexer, matches patterns in its input. However, unlike the Lexer, it does not match string patterns. Rather, it matches patterns between tokens rather than characters, and outputs an abstract syntax tree. Again, this matching is defined by a set of rules; however, there are a few extra operators that help define the resulting AST:

```
functionBody:           L_BRACE^ (statement)* R_BRACE! ;
```

The rule above, for example, matches blocks of code between braces. A block of code is defined as zero or more statements between a left brace and a right brace. Statements are matched by the statement subrule, and 'zero or more' is specified with the Kleene star operator (*). The structure of the resulting AST tree is also specified within the rule: the carat operator (^) defines the left brace token as the root of the functionBody subtree; the statements become the root node's children, and the exclamation point (!) tells the parser to match the right brace but not include it in the AST.

SIMPLEX TREE WALKER

The tree walker is another pattern matcher, but matches subtrees in the AST rather than tokens or characters. Because the tree walker creates Java IR objects to represent the nodes on the AST, there is significantly more Java code embedded in the ANTLR rules. Again, the syntax of the walker is different from that of the previous steps. An example rule is below:

```
decl returns [Decl d]
{ d = null; Type t; }
      :     #(DECL t=type IDENTIFIER
                  {
                        if (stack.get(#IDENTIFIER.getText()) != null) {
                              System.err.println("Variable already
                                    declared: " + #IDENTIFIER.getText());
                        }
                        stack.put(#IDENTIFIER.getText(), t);
                        d = new Decl(t, #IDENTIFIER.getText());
                  }
            )
      ;
```

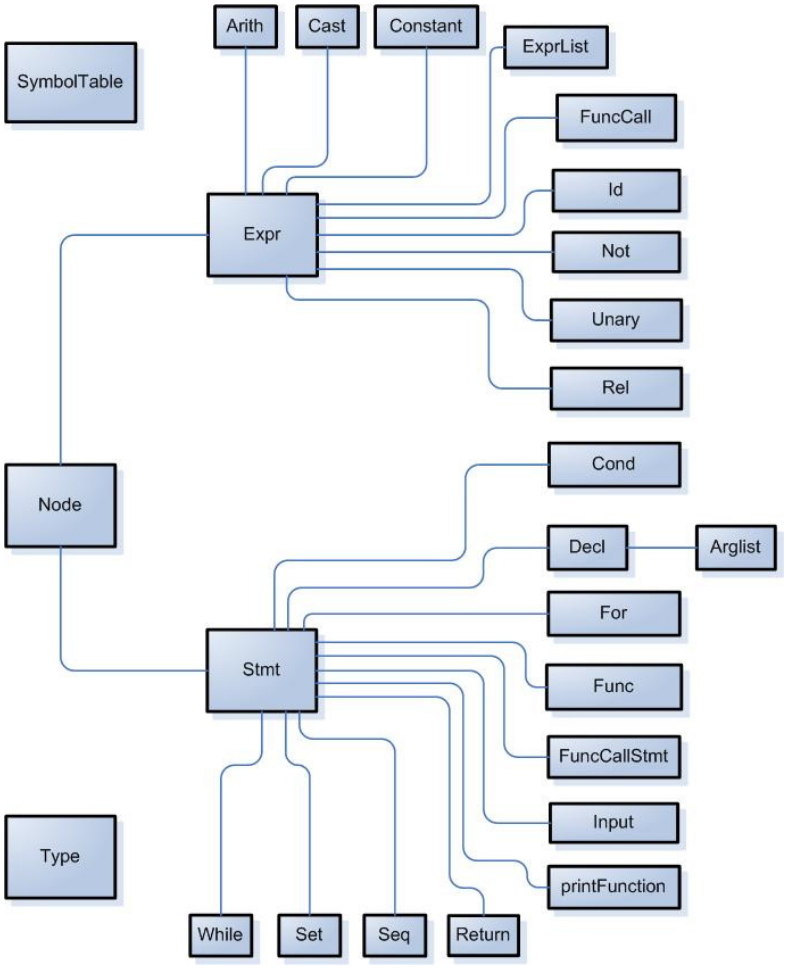
This rule, called 'decl', passes a Decl object to the rule that calls it. The code within the first set of braces initializes the Decl object, d, to null, and creates a Type object, t, in the same scope.

It then matches a tree with a DECL node as the root and a type node and an IDENTIFIER node as children. Once it has matched these, it executes the Java code within the second set of braces. This code first checks the global symbol table, 'stack' to see if the identifier is declared, and if not, it puts the identifier on the stack and initializes the Decl object to return. If the variable is already declared, it generates an error message.

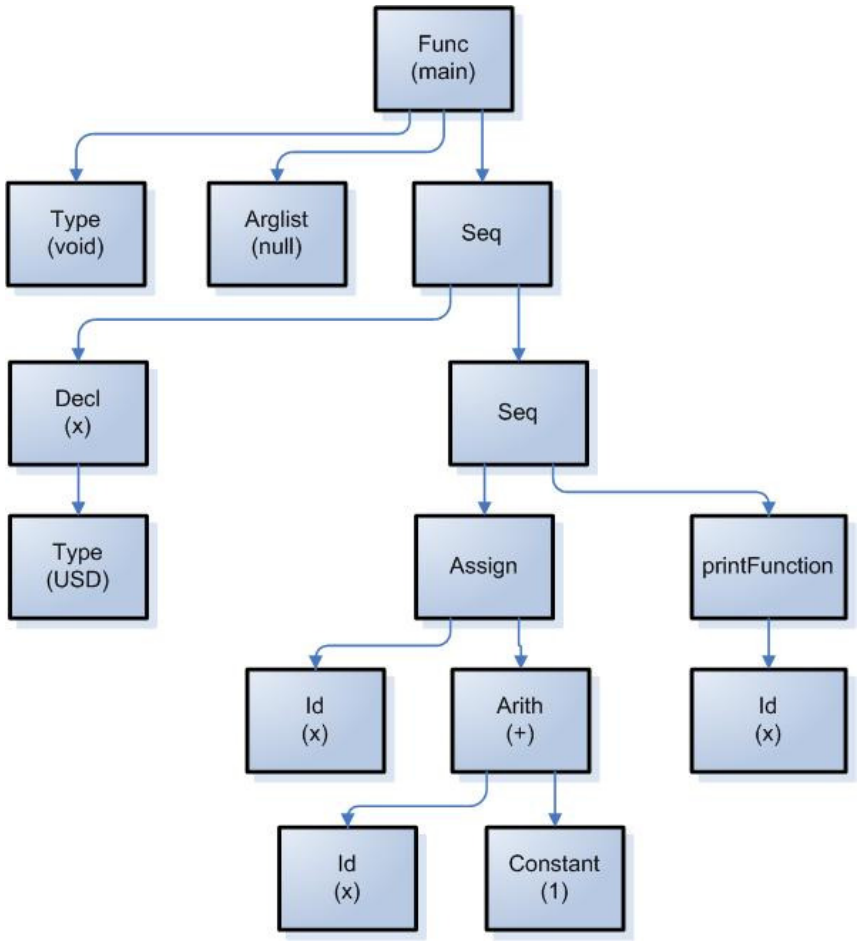
The Decl and Type objects are just two of the intermediate representation classes that eventually generate the translated code. The Decl object inherits from the more general Stmt object, which, through the magic of polymorphism, will eventually represent the entire program.

SIMPLEX INTERMEDIATE REPRESENTATION CLASSES

There is one Java IR class for every language construct in SIMPLEX. Almost all of the classes inherit from the Node object, which contains a method to print error messages to the console. The two subclasses, Expr and Stmt, represent expressions and statements respectively. Expressions consist of the class of language constructs that return values—arithmetic expressions, boolean expression, function calls, etc. Statements are assignments, conditionals, loops, and the like. Each possible expression or statement is represented by its own IR class, which inherits from their respective subclasses (Expr or Stmt). The entire class hierarchy is shown below:



There are several special classes in the hierarchy—Seq, Arglist, and ExprList. These do not represent specific statements or expressions; rather, they represent groups of statements and expressions. Seq, for example, contains two Stmt objects. Since Seq itself inherits from Stmt, it can contain instances of itself. By including Seq objects in other Seq objects, large numbers of statements can be grouped together in a tree structure. Each subclass also defines a 'gen()' method which overrides the superclass's 'gen()', and generates the translated Java code for the corresponding construct. By creating a single Seq object and adding other statements to it, the gen() function can be called on each of the statements recursively when all the IR classes are created, and the translated program will be output. For example, the following IR would represent the SIMPLEX program below it:



SIMPLEX program:

```
void main() {
      USD x;
      x = x + 1;
      print(x);
}
```

The Func object's gen() function will then call the gen() function of Seq, which then calls the gen() function of its children, Decl and Seq. By recursively calling gen() in this way, the tree

walker can return a single object whose base type is Stmt and generate the translated Java code from that object.

TEST PLAN

TEST METHODS

For testing, the team developed a testing framework consisting of two modules: the console and the regression testing suite. The console is a Java based program that provides a command line environment for compiling and running .spx files. The user can input "compile <filename>" to translate a .spx file into a .java file and compile it into a .class file. Upon successful completion, the user can then input "run <programname>" to execute a program. The second module is the regression testing suite, which is a Java based program that automates the compilation and execution of a batch of .spx files. These .spx files are located in the <classpath>/tests directory. The regression test suite does this by first automatically generate a list of .spx files in the /tests directory, then tries to compile and run each of the .spx file. If the compilation is unsuccessful, then the test suite will flag that file with erroneous syntax. If the compilation is successful, then the program will be executed and any output will be printed to the console. The console and the regression test suite together form the testing framework for Simplex.

THE CONSOLE

As mentioned above, the console provides a command line environment for Simplex developers to compile and run a .spx program. It is Java based and its class and method signatures are as follows:

```
public class Console {

	// prints the commands available in the console
	public static void printUsage(boolean err) {
		...
	}

	// parses the .spx file name from the path provided by the user
	public static String getFilename(String path) {
		...
	}

	// checks whether a file is a valid .spx file
	public static boolean checkFilename(String path, boolean checkSpx) {
		...
	}

	// executes a windows shell command
	public static Process cmdExec(String cmd) throws IOException,
			InterruptedException {
		...
	}

	// runs a compiled .spx program
	public static void run(String file) throws IOException,
			InterruptedException {
		...
	}

	// compiles a .spx file
```

```
	public static void compile(String path) throws
	FileNotFoundException, IOException, InterruptedException,
	RecognitionException, TokenStreamException {
		...
	}

	// main program
	public static void main(String[] args) {
		...
	}

}
```

For the console's full source code, please see Appendix I. Depending on the command inputted by the user, the console compiles or runs a .spx program with exceptions handling. A sample console usage would be:

```
> help
Commands:
compile <filename>
run <programname>
exit
<filename> must a string of alphabets followed by the extension .spx

> compile /tests/arith.spx
arith.spx successfully compiled.

> run arith
48.0

> exit
```

Therefore the console provides the user an environment to compile and execute .spx programs.

REGRESSION TEST SUITE

The regression test suite builds on top of the console. It uses the static methods such as cmdExec(), run(), and compile() to automatically compile and run the list of .spx files in the /tests directory. To include a .spx in the regression test, simply include it in the /test directory. The regression test suite has the following pseudo code:

```
public class TestSuite {

	public static void main(String[] args) {


		Use cmdExec() to get a list of .spx files in the /test directory.
		Save the files in a String array.

		For each file in the String array:
			Compile(file)
			If(Compile is successful)
				Run(file)

		Testing completed.
```

```
	}

}
```

For its complete source code, please view Appendix II. The group developed simple test cases first to test fundamental aspects of the language such as variable declaration, types, print functionalities, arithmetics, function declarations and usage, logicals, branches, etc. The following are some of the test cases we have developed:

```
print.spx:
void main() {
	print("Hello world");
}
Output: Hello world

declaration.spx:
void main() {
	number num;
	string str;
	USD u;
	print(num);
}
Output: 0.0

arithmetics.spx:
void main() {
	EUR a;
	day b;
	string x;
	USD y;
	number neg;
	neg++;
	// a = (b / a) * (365);
	y=1;
	a = a * (2 + 4 + a) / y;
	a = 2 * 4 + 8 * 5;
	print(a);
}
Output: 48.0

for.spx:
void main() {
	number i;
	number j;

	for (i = 1; i < 11; i = i + 1)
		j++;
	print(j);
}
Output: 10.0

if.spx:
void main() {
	YEN x;
	x = 0;
	if (x < 10)
		if (x > 11)
			print(x);
		else
			x = 3;
	else
```

```
            x = 11;
}
Output: 0

while.spx:
void main() {
      number x;
      number y;
      while (x < 10) {
            x = x + 1;
            x = y + x;
      }
      print(x);
}
Output: 10

func.spx:
YEN test(rate testrate, USD x) {
      print("testing!");
      EUR y;
      testrate = 1;
      return 2;
}

void main() {
      CNY x;
      test(1,x);
      x = -x;
      return;
}
Output: testing!

greedy.spx:
void main ()
{
      number a;
      a = 4;
      if (a == 3) {
            if (a == 4) {
                  print ("I got inside!");
            }
      }
      else {
            print ("I am in the else statement");
      }
}
Output: I am in the else statement
```

With the language's fundamental elements properly tested, we developed more sophisticated programs both to test several elements simultaneously as well as to demonstrate the type of programs typically developed with Simplex.

TEST APPLICATIONS

Our first test application is a typical financial application – the mortgage payment and amortization schedule. This is the schedule of payments a homeowner would make on their mortgage. In the test application, we allow the user to input the total mortgage amount, the number of payments per year, the number of years and the interest rate. The program then outputs a period by period payment schedule. The test application makes strong use of mathematical functions, loops, our input and print functions, and rate data type. Below is the code for this application:

```
// author: Steven Chen

void main() {
   USD totalMortgage;
   number numYears;
   number paymentsPerYear;
   rate interestRate;
   USD payment;

   print("Enter Total Mortgage amount: ");
   input(totalMortgage);
   print("Enter interest rate: ");
   input(interestRate);
   print("Enter Payments Per Year: ");
   input(paymentsPerYear);
   print("Enter Number of years: ");
   input(numYears);

   rate interestPerPeriod;
   interestPerPeriod = interestRate/paymentsPerYear;

   // Calculate Payment manually
   payment = totalMortgage * (interestPerPeriod);
   payment = payment * ((1+interestPerPeriod)^(numYears*paymentsPerYear));
   payment = payment / ((1+interestPerPeriod)^(numYears*paymentsPerYear)-1);


   // calculate and output amortization schedule
   number currentPeriod;
   currentPeriod = 0;
   USD beginningBalance;
   beginningBalance = totalMortgage;
   USD endingBalance;
   USD interestPayment;
   USD amort;

   print ("PERIOD\tBEGINNING BALANCE\tINTEREST PAYMENT\tAMORTIZATION\tENDING BALANCE");
   print ("------\t-----------------\t----------------\t------------\t--------------");

   while (currentPeriod < ((number)numYears*paymentsPerYear))
   {
      interestPayment = (beginningBalance * (interestPerPeriod));
      amort = (payment - interestPayment);
      endingBalance = (beginningBalance - amort);

      print (currentPeriod + "\t\t" + beginningBalance + "\t     " + interestPayment + "\t" +
amort + "\t" + endingBalance);

      beginningBalance = endingBalance;
      currentPeriod++;
   }

}
```

The second test application is one that involves heavy use of type (currency) casting in addition to the print function. The test application assumes that a US-based landlord owns buildings in several countries (Canada, Japan and somewhere in the E.U.). The landlord would like to know his/her cash inflow from rents for the next 4 years in U.S. Dollars. The landlord would also like to know the growth rate year of year of rents.

```
// author: Steven Chen

USD calculateYearlyRent(USD a, EUR b, CAD c, YEN d)
{
   USD yearly;
   yearly = ( (USD)a + (USD)b + (USD)c + (USD)d );
   return yearly;

}

void main()
{
   USD rent_2007;
   USD rent_2008;
   USD rent_2009;
   USD rent_2010;

   rate growth_07_08;
   rate growth_08_09;
   rate growth_09_10;

   rent_2007 = calculateYearlyRent(1000000, 300000, 123222, 233332322);
   rent_2008 = calculateYearlyRent(1030000, 330000, 121990, 235665645);
   rent_2009 = calculateYearlyRent(1060900, 363000, 120770, 238022302);
   rent_2010 = calculateYearlyRent(1092727, 399300, 119562, 240402525);

   growth_07_08 = (rent_2008 - rent_2007)/rent_2007;
   growth_08_09 = (rent_2009 - rent_2008)/rent_2008;
   growth_09_10 = (rent_2010 - rent_2009)/rent_2009;

   print ("Total 2007 Rent: " + rent_2007);

   print ("Total 2008 Rent: " + rent_2008);
   print ("\t\tYear over Year growth: " + growth_07_08);

   print ("Total 2009 Rent: " + rent_2009);
   print ("\t\tYear over Year growth: " + growth_08_09);

   print ("Total 2010 Rent: " + rent_2010);
   print ("\t\tYear over Year growth: " + growth_09_10);

}
```

LESSONS LEARNED

STEVEN CHEN

As an engineer, working on a large group project is nothing new to me. However, this project was much different. There was no usual document with a step by step procedure. There were no major restricting guidelines. There was certainly no hand-holding. The task at had was to design our own language, implement it and put it under a rigorous testing procedure.

The beginning of the project proved to be a struggle for me. I spent a significant portion of time trying to understand ANTLR and what was required to even begin programming. I spent time looking at online tutorials, class handouts and asking my teammates and classmates for help. An integral part to being successful is to try and absorb as much information earlier on so that the latter parts of the projects will require less learning and more 'doing.'

Another key to being successful is to recognize that everyone has a different way of doing things and that everyone works on a different schedule and to take advantage of this fact. As a group, we were not afraid to let a teammate 'run' with an idea and let him work ahead and catch the team up later. If we had all waited until everyone was free, we may not have began the project until late November/early December. Thus, at least some portion of the project was being looked at or worked on constantly since mid September, albeit not with the every single team member.

Lastly, as I fell into a leadership position towards the beginning of the project, I began setting fairly aggressive deadlines for the group as a whole. While not every deadline was met, setting an aggressive schedule constantly forced us to show a sense of urgency to prevent falling too far behind. Milestones (e.g. finishing the Lexer or Tree Walker) were established in order to focus the group and work towards a common goal under deadline pressure. This was effective since unlike some groups in the past, we did not end up struggling for time towards the end of this process.

GILBERT HOM

Creating a language like SIMPLEX initially looked like a daunting task. I didn't have any prior experience or idea on how to handle such a task but after working on it throughout the semester, not only did it become very feasible, I've learned quite a bit about the programming development phases.

Planning – Experimenting with many ideas for a new type of language, working out syntax on paper helped *a lot* in the development phase. Fleshing out more than ~85% of our ideas onto paper and into our Language Reference Manual was essential to the growth of our language. Working out all the foundations and fundamental blocks first allowed for versatility and growth. Advanced planning on our walker helped a lot while parsing and organizing the way SIMPLEX handled equations. If it weren't for organized planning this task would've been impossible.

Organization – While planning is essential, organization was just as important. Having a method of syncing everyone's ideas and code was crucial in our development phase. Eclipse's SVN helped a great deal throughout this whole process. It prevented group members from stepping on each other toes during implementation.

Testing – While implementing all aspects of a language, testing to make sure they are all functioning properly isn't the only thing that is needed to be tested. Testing for exceptions and

testing thoroughly is just as important. Lack of testing will just bring up issues when users attempt to utilize the language incorrectly.

KELVIN JIANG

The project was a very useful experience. I learned a lot about the fundamental building blocks of developing a language, but more importantly, I learned a lot about software engineering and software development cycles. Here are four major lessons I've gained from this project, which also serve as advice for future projects:

- Plan a lot and plan early. A good project should consist of at least 50% planning effort before implementation, especially for a highly complex undertaking such as this. Know exactly what the goal is, how to get there, and the effort needed to make it successful.
- Keep it simple. This is especially true for writing a language. Don't bloat the language with libraries and functions; instead, write small features and building blocks that will allow users to develop powerful apps on top.
- Have a good working platform. Before actually starting the project, make sure the necessary tools such as CVS/SVN, Eclipse/ANTLR Studios, etc, are in place.
- Test thoroughly and incrementally. If you don't test every single aspect of your language, then your users sure will. However, writing a language is complex and it's highly unlikely for a group to cover all the bugs before release. Therefore, remember to test the language incrementally and build complex tests on top of the simple ones.

ERIC ZHANG

This project has been a fantastic learning experience. Not only did we have the opportunity to learn how a compiler a built from the ground up, we had the chance to build a working compiler ourselves and experience firsthand how all of the different parts interact and the kinds of problems compiler writers run into on a daily basis.

One of the major lessons I've learned during this project is the importance of teamwork. In a software engineering endeavor of this size, cooperation, coordination, and collaboration are essential to the project's timely completion. Unlike an individual project where each person can work at his or her own pace, the team project requires that each contributor sticks to the deliverables schedule or the progress of the entire project will stall. Coordinating using a versioning system like SVN helped immensely.

However, even the most seamless of teams cannot complete a project without a plan. This project required that we each fully understand the entire compilation process before a single line of code was even written. Writing code blindly without understanding the interactions between parts is a recipe for disaster. We needed to first come up with a development plan and a good idea of what the code would look like and how classes would interact before the coding could begin.

Understanding the importance of planning has given me a new perspective on how a piece of software should be written. Instead of diving directly into coding at the start, I have found that thinking about the structure of the program and understanding how each requirement is fulfilled saves an enormous amount of time later on, since the codebase will be much more flexible and resilient to changes in the later phases of development.

APPENDIX

grammar.g
author: Steven Chen, Eric Zhang

```
class SimplexLexer extends Lexer;
options
{
   charVocabulary = '\0'..'\377';
   testLiterals = false;       // don't check every rule vs. keywords
   k = 2;                  // character lookahead to 2
   exportVocab = SIMPLEX;      // export for tree walker
}


// Math Related
POINT:          '.';
ASTERISK:       '*';
SLASH:          '/';
PLUS:          '+';
MINUS:          '-';
CARAT:          '^';
PERCENT:       '%';
DBL_OP:          ("++" | "--");
EQUALS:          '=';
OPEQUALS:       ("+=" | "-=" | "/=" | "*=" | "^=");

// Logic Related
OR:             "||";
AND:           "&&";
NOT:          "!";

// Relational / Equality
G_THAN:          ">";
L_THAN:          "<";
EQ_TO:          "==";
NOT_EQ_TO:       "!=";
GT_EQ_TO:       ">=";
LT_EQ_TO:       "<=";

// Brackets
L_PAREN:       '(';
R_PAREN:       ')';
L_BRACE:       '{';
R_BRACE:       '}';
L_S_BRACKET:    '[';
R_S_BRACKET:    ']';

// Others
COLON:          ':';
SEMICOLON:       ';';
COMMA:          ',';

protected DIGIT:    '0'..'9';
protected LETTER:    ('A'..'Z' | 'a'..'z');

IDENTIFIER options {testLiterals = true;}
   :   ('_' | LETTER) ('_' | LETTER | DIGIT)*
   ;

NUMBER
   :   ('0'..'9')+ ('.' ('0'..'9')*)?
   ;

STRING
   :   '"' (ESCAPECHAR|~'"')* '"'
   ;
```

```
protected ESCAPECHAR
   : '\\' ( 'n' | 'r' | 't' | 'b' | 'f' | '"' | '\'' | '\\' )
   ;

UNIX_NEWLINE
   :   '\n'
      { $setType(Token.SKIP); }
   ;

MAC_NEWLINE
   :   '\r'
      { $setType(Token.SKIP); }
   ;

WHITESPACE
   :   (' ' | '\t' | '\f')+
      { $setType(Token.SKIP); }
   ;

COMMENT
   :    (
         "/*"
            (   // multi-line comments
               options {greedy=false;}
               : ('\n' | '\r')
               | ~( '\n' | '\r' )
            )*
         "*/"
      |
         "//"    // single line comments
            (~( '\n' | '\r' ))* ('\n' | '\r')
      )

      { $setType(Token.SKIP); }
   ;


////////////////////////////////////////////////////////////

class SimplexParser extends Parser;
options {    buildAST = true;
         k = 3;
         exportVocab = SIMPLEX;
      }
tokens { DECL; FUNC; NEG; CAST; }

type
   :
      "number" | "string" | "rate" | "day" | "month" | "year" |
      "USD" | "YEN" | "CNY" | "EUR" | "ILS" | "AUD" | "CAD" | "GBP" | "MXN" | "SOS"
   ;

start
   :
      (variableDeclaration SEMICOLON!)*
      (functionDef)*
      "void"! "main"^ L_PAREN! R_PAREN! functionBody
   ;

//////////////FUNCTION RELATED////////////////////////
functionDef
   :
      ("void" | type) IDENTIFIER argumentList functionBody
      { #functionDef = #([FUNC, "FUNC"], #functionDef); }
   ;

argumentList
   :
      L_PAREN! (variableDeclaration (COMMA! variableDeclaration)*)? R_PAREN!
   ;
```

```
functionBody
   :         // body is just a series of statements in braces
      L_BRACE^
      (statement)*
      R_BRACE!
   ;

printFunction
   :
      "print"^
      L_PAREN!
      expression
      R_PAREN!
   ;

inputFunction
   :
      "input"^
      L_PAREN!
      IDENTIFIER
      R_PAREN!
   ;

functionCall
   :
      IDENTIFIER L_PAREN! variableList R_PAREN!
      { #functionCall = #([FUNC, "FUNC"], #functionCall); }
   ;

variableList
   :   expression (COMMA! expression)*
   |   //nothing
   ;

////////////////////////////////////////////////////////

variableDeclaration
   :
      type IDENTIFIER
      {#variableDeclaration = #([DECL, "DECL"], #variableDeclaration); }
   ;

statement
   :   variableDeclaration SEMICOLON!
   |    assignmentStatement SEMICOLON!
   |   printFunction SEMICOLON!
   |   inputFunction SEMICOLON!
   |   functionCall SEMICOLON!
   |   returnStatement SEMICOLON!
   |   ifStatement
   |   whileStatement
   |   forStatement
   |   jumpStatement
   |   SEMICOLON!
   ;

assignmentStatement
   :
      IDENTIFIER EQUALS^ expression
   |   IDENTIFIER DBL_OP^
   |   IDENTIFIER OPEQUALS^ expression
   ;

expression
   :
      logical (OR^ logical)*
   ;

logical
   :
```

```
      relation (AND^ relation)*
   ;

relation
   :
      addition
      (
         (G_THAN^ | L_THAN^ | EQ_TO^ | NOT_EQ_TO^ | GT_EQ_TO^ | LT_EQ_TO^)
         addition
      )?
   ;

addition
   :
      multiply
      (
         (PLUS^ | MINUS^)
         multiply
      )*
   ;

multiply
   :
      unary
      (   (
            (ASTERISK^ | SLASH^ | CARAT^)
            unary
         )*
      |   PERCENT^
      )
   ;

unary
   :
      rValue
   |   MINUS! unary { #unary = #([NEG, "NEG"], #unary); }
   |   L_PAREN! type R_PAREN! unary { #unary = #([CAST, "CAST"], #unary); }
   |   NOT^ unary
   ;

rValue
   :
      NUMBER
   |   STRING
   |   ("true" | "false")
   |   IDENTIFIER
   |   L_PAREN! expression R_PAREN!   // expr in parens
   |   functionCall
   ;

returnStatement
   :   "return"^ (expression)?
   ;

ifStatement
   : "if"^ L_PAREN! expression R_PAREN! (statement | functionBody)
      (
         options {greedy = true;}:
         "else"! (statement | functionBody)
      )?
   ;

whileStatement
   :   "while"^ L_PAREN! expression R_PAREN! (statement | functionBody)
   ;

forStatement
   :   "for"^ L_PAREN! assignmentStatement SEMICOLON! expression SEMICOLON! assignmentStatement
R_PAREN! (statement | functionBody)
   ;
```

```
jumpStatement
   :   "break"   | "continue"
   ;

class simplexWalker extends TreeParser;
{
   SymbolTable stack = null; int LoopLevel = 0;
}

main returns [Stmt s]
{ s = null; Stmt s1 = null; Type t; stack = new SymbolTable(stack, Type.Void); }
   :   #("main"
         (s1=globals)?
         s=functionbody
         {
            Arglist args = new Arglist();
            args.add(new Decl(Type.String, "args[]"));
            s = new Func(new Type("public static void", -1, 1, ""), "main", args, s);
            if (s1 != null) s = new Seq(s1, s);
         }
      )
   ;

globals returns [Stmt s]
{ s = null; Stmt s1; }
   :   s=globaldecl ( s1=globals { s = new Seq(s, s1); } )?
   ;

globaldecl returns [Stmt s]
{ s = null; Type t; }
   :   s=decl
   |   s=function
   ;

function returns [Stmt s]
{ s = null; Type t; Stmt s1; Arglist a = new Arglist(); }
   :   #(FUNC (t=type | "void" {t=Type.Void;}) IDENTIFIER
         {
            //Create a new stack for the function
            SymbolTable oldstack = stack; stack = new SymbolTable(stack, t);
         }
         //Get the arguments (args puts arg variables onto new stack)
         (a=args)?
         {
            //Put this function name onto the outer stack so other functions can access it
            oldstack.put(#IDENTIFIER.getText(), t, a);
         }
         //Get the rest of the function body
         s1=functionbody
         {
            if (t != Type.Void) {
               if (!s1.hasReturn) {
                  System.err.println("Error: Function " + #IDENTIFIER.getText() + " may not
return a value");
                  System.exit(0);
               }
            }
            s = new Func(t, #IDENTIFIER.getText(), a, s1);
            stack = oldstack;
         }

      )
   ;

args returns [Arglist a]
{ a = new Arglist(); Decl a1; Arglist a2; }
   :   a1=decl
      {
         a.add(a1);
      }
      (a2=args { a.copy(a2); })?
```

```
   ;

functionbody returns [Stmt s]
{ s = null; }
   :   #(L_BRACE
         (s=stmts)?
       )
   ;

stmts returns [Stmt s]
{ s = null; Stmt s1; }
   :   s=stmt ( s1=stmts { s = new Seq(s, s1); } )?
   ;

assign returns [Set s]
{ s = null; Expr e; }
   :   #(EQUALS IDENTIFIER e=expr
         {
            Id var = stack.get(#IDENTIFIER.getText());
            if (var == null) {
               System.err.println("Error: Undeclared Identifier: " + #IDENTIFIER.getText());
               System.exit(0);
            }
            else s = new Set(#EQUALS.getText(), var, e);
         }
       )
   |   #(OPEQUALS IDENTIFIER e=expr
         {
            Id var = stack.get(#IDENTIFIER.getText());
            if (var == null) {
               System.err.println("Error: Undeclared Identifier: " + #IDENTIFIER.getText());
               System.exit(0);
            }
            else s = new Set(#OPEQUALS.getText(), var, e);
         }
       )
   |   #(DBL_OP IDENTIFIER
         {
            Id var = stack.get(#IDENTIFIER.getText());
            if (var == null) {
               System.err.println("Error: Undeclared Identifier: " + #IDENTIFIER.getText());
               System.exit(0);
            }
            else s = new Set(#DBL_OP.getText(), var, null);
         }
       )
   ;

stmt returns [Stmt s]
{ s = null; Stmt s1 = null; Set f1, f2; Type t; Expr e = null; ExprList args = new ExprList();}
   :   s=decl
   |   #("print" e=expr
         {
            s = new printFunction(e);
         }
       )
   |   #("input" IDENTIFIER
         {
            Id var = stack.get(#IDENTIFIER.getText());
            if (var == null) {
               System.err.println("Error: Undeclared Identifier: " + #IDENTIFIER.getText());
               System.exit(0);
            }
            else s = new Input(var);
         }
       )
   |   s=assign
   |   #("if" e=expr (s=stmt | s=functionbody) (s1=stmt | s1=functionbody)?
         {
            s = new Cond(e, s, s1);
         }
```

```
      )
   |   #("while" e=expr
         {
            LoopLevel++;
         }
         (s=stmt | s=functionbody)
         {
            s = new While(e, s);
            LoopLevel--;
         }
      )
   |   #("for" f1=assign e=expr f2=assign
         {
            LoopLevel++;
         }
         (s=stmt | s=functionbody)
         {
            s = new For(f1, e, f2, s);
            LoopLevel--;
         }
      )
   |   e=funccall
         {
            s = new FuncCallStmt((FuncCall)e);
         }
   |   #("return" (e=expr)?
         {
            s = new Return(stack.getReturnType(), e);
         }
      )
   |   #("break"
         {
            if (LoopLevel > 0) {
               s = new Stmt("break");
            }
            else {
               System.err.println("Error: Cannot use BREAK statement outside a For or a While
loop.");
            }
         }
      )
   |   #("continue"
         {
            if (LoopLevel > 0) {
               s = new Stmt("continue");
            }
            else {
               System.err.println("Error: Cannot use CONTINUE statement outside a For or a While
loop.");
            }
         }
      )
   ;

funccall returns [Expr e]
{ e = null; ExprList args = new ExprList(); }
   :   #(FUNC IDENTIFIER (args=exprs)?
         {
            Id func = stack.get(#IDENTIFIER.getText());
            if (func == null) {
               System.err.println("Error: Undeclared Identifier: " + #IDENTIFIER.getText());
               System.exit(0);
            }
            else {
               e = new FuncCall(func, args);
            }
         }
      )
   ;

decl returns [Decl d]
```

```
{ d = null; Type t; }
   :   #(DECL t=type IDENTIFIER
         {
            if (stack.get(#IDENTIFIER.getText()) != null) {
               System.err.println("Variable already declared: " + #IDENTIFIER.getText());
               System.exit(0);
            }
            stack.put(#IDENTIFIER.getText(), t);
            d = new Decl(t, #IDENTIFIER.getText());
         }
       )
   ;

//TODO: add boolean
type returns [Type t]
{ t = null; }
   :   ("number"   { t = Type.Number; }
       |"rate"     { t = Type.Rate; }
       |"day"      { t = Type.Day; }
       |"month"    { t = Type.Month; }
       |"year"     { t = Type.Year; }
       |"string"   { t = Type.String; }
       |"USD"      { t = Type.USD; }
       |"EUR"      { t = Type.EUR; }
       |"CAD"      { t = Type.CAD; }
       |"GBP"      { t = Type.GBP; }
       |"AUD"      { t = Type.AUD; }
       |"ILS"      { t = Type.ILS; }
       |"CNY"      { t = Type.CNY; }
       |"MXN"      { t = Type.MXN; }
       |"SOS"      { t = Type.SOS; }
       |"YEN"      { t = Type.YEN; }
       )
   ;

exprs returns [ExprList e]
{ e = new ExprList(); Expr e1; ExprList e2; }
   :   e1=expr
       {
          e.add(e1);
       }
       (e2=exprs { e.copy(e2); })?
   ;

expr returns [Expr e]
{ Expr a, b; e = null; Type t;}
    :
       #(OR a=expr b=expr { e = new Rel("||", a, b); } )
    |    #(AND a=expr b=expr { e = new Rel("&&", a, b); } )
    |   #(EQ_TO a=expr b=expr { e = new Rel("==", a, b); } )
    |    #(NOT_EQ_TO a=expr b=expr { e = new Rel("!=", a, b); } )
    |    #(L_THAN a=expr b=expr { e = new Rel("<", a, b); } )
    |    #(LT_EQ_TO a=expr b=expr { e = new Rel("<=", a, b); } )
    |    #(G_THAN a=expr b=expr { e = new Rel(">", a, b); } )
    |    #(GT_EQ_TO a=expr b=expr { e = new Rel(">=", a, b); } )
    |   e=funccall
    |    #(PLUS a=expr b=expr { e = new Arith("+", a, b); } )
    |    #(MINUS a=expr b=expr { e = new Arith("-", a, b); } )
    |    #(ASTERISK a=expr b=expr { e = new Arith("*", a, b); } )
    |    #(SLASH a=expr b=expr { e = new Arith("/", a, b); } )
    |   #(CARAT a=expr b=expr { e = new Arith(",", a, b); } )
    |   #(PERCENT a=expr { e = new Arith("/", a, new Constant(100)); } )
    |    #(NOT a=expr { e = new Not(a); } )
    |    #(NEG a=expr { e = new Unary("-", a); } )
    |   #(CAST t=type a=expr { e = new Cast(t, a); } )
    |   #(IDENTIFIER
          {
             e = stack.get(#IDENTIFIER.getText());
             if (e == null) {
                 System.err.println("Error: Undeclared Identifier: " + #IDENTIFIER.getText());
             System.exit(0);
```

```
          }
        }
    )
|     NUMBER { e = new Constant(#NUMBER.getText(), Type.Number); }
|     "true" { e = Constant.True; }
|     "false" { e = Constant.False; }
|    STRING   { e = new Constant(#STRING.getText()); }
;
```

Main.java
author: Eric Zhang, Kelvin Jiang

```
import java.io.*;
import antlr.CommonAST;
//import antlr.collections.AST;
import antlr.debug.misc.ASTFrame;

class Main {
	public static void main(String[] args) {
		try {
			File f;
			String filename = "  ";
			while (true) {
				do {
					System.out.print("File Name: ");
					BufferedReader in = new BufferedReader(
							new InputStreamReader(System.in));
					filename = in.readLine();
					f = new File(filename);
					if (filename.equals("q")) {
						System.out.println("Quit.");
						return;
					}
				} while (!f.exists());

				SimplexLexer lexer = new SimplexLexer(new DataInputStream(
							new FileInputStream(filename)));
				SimplexParser parser = new SimplexParser(lexer);
				parser.start();

				// Get the AST from the parser
				CommonAST parseTree = (CommonAST) parser.getAST();
				// Print the AST in a human-readable format
				System.out.println(parseTree.toStringList());
				// Open a window in which the AST is displayed graphically
				ASTFrame frame = new ASTFrame("AST from the Simp parser",
							parseTree);
				frame.setVisible(true);

				simplexWalker walker = new simplexWalker();
				Stmt s = walker.main(parseTree);
				//TODO: add these to the console class, etc.
				System.out.println("import java.util.*;");
				System.out.println("import java.io.*;");
				System.out.println("public class Main {");
				System.out.println("public static BufferedReader stdin = new
BufferedReader(new InputStreamReader(System.in));");
				System.out.println("public static double readDouble() {");
				System.out.println("while (true) {");
				System.out.println("try { return
Double.parseDouble(stdin.readLine()); }");
				System.out.println("catch (NumberFormatException e) {
System.out.println(\"That is not a valid number, please reenter.\"); }");
				System.out.println("catch (IOException e) {
System.out.println(\"I/O Error, terminating program.\"); System.exit(0); }");
				System.out.println("}}");
				s.gen();
				System.out.println("}");

			}

		} catch (Exception e) {
			System.err.println("exception: " + e);
		}
	}
}
```

Arglist.java

```
author: Eric Zhang

import java.util.*;

public class Arglist extends Decl {
        public Vector decls;

        public Arglist() {
                decls = new Vector();
        }

        public void add(Decl d) {
                decls.add(d);
        }

        public void copy(Arglist a1) {
                for (int i = 0; i < a1.decls.size(); i++) {
                        decls.add(a1.decls.get(i));
                }
        }

        public int size() {
                return decls.size();
        }

        public Type getType(int i) {
                return ((Decl)(decls.get(i))).type;
        }

        public void gen() {
                if (decls.size() > 0) {
                        ((Decl)(decls.get(0))).gen2();
                        for (int i = 1; i < decls.size(); i++) {
                                System.out.print(", ");
                                ((Decl)(decls.get(i))).gen2();
                        }
                }
        }

}
```

Arith.java
author: Kelvin Jiang

```
public class Arith extends Expr {
	public Expr expr1, expr2;

	public Arith(String op, Expr x1, Expr x2) {
		super(op, null);
		expr1 = x1;
		expr2 = x2;

		//Set the type of this expression
		// If the expressions are of different types
		if (expr1.type.classId != expr2.type.classId) {
			//First check for string concatenation
			if (expr1.type == Type.String) {
				if (op != "+") error("Error: Cannot use string expression in
arithmetic operation.");
				else if (expr2.type.isNumeric()){
					type = Type.String;
					expr2 = new Cast(Type.String, expr2);
				}
			}
			else if (expr2.type == Type.String) {
				if (op != "+") error("Error: Cannot use string expression in
arithmetic operation.");
				else if (expr1.type.isNumeric()){
					type = Type.String;
					expr1 = new Cast(Type.String, expr1);
				}
			}
			//If no concatentation, then you can't do arithmetic ops on booleans
			else if (expr1.type == Type.Boolean || expr2.type == Type.Boolean) {
				error("Error: Cannot use boolean expression in arithmetic
operation.");
			}
			//Check for number or rates
			// Always convert currencies and dates to USD and Days to maintain
consistency
			else if (expr1.type.classId == Type.NUMBER || expr1.type.classId ==
Type.RATE) {
				if (expr2.type.classId == Type.CURRENCY) type = Type.USD;
				else if (expr2.type.classId == Type.DATE) type = Type.Day;
				else if (expr1.type == Type.Number) type = expr2.type;
				else type = expr1.type;
			}
			else if (expr2.type.classId == Type.NUMBER || expr2.type.classId ==
Type.RATE) {
				if (expr1.type.classId == Type.CURRENCY) type = Type.USD;
				else if (expr1.type.classId == Type.DATE) type = Type.Day;
				else if (expr2.type == Type.Number) type = expr1.type;
				else type = expr2.type;
			}
			// If here, then either:
			//  expr1 = currency && expr2 = date
			//  or
			//  expr1 = date && expr2 = currency
			else if (expr1.type.classId == Type.CURRENCY) {
				if (op == "*" || op == "/") type = Type.USD;
				else type = Type.Number;
			}
			else if (expr1.type.classId == Type.DATE) {
				if (op == "*") type = Type.USD;
				else if (op == "/") type = Type.Day;
				else type = Type.Number;
			}
			else {
				type = Type.Number;
			}
		}
		// If both expressions are of the same type
```

```
                else {
                        if (expr1.type.classId == Type.CURRENCY) {
                                if (op == "+" || op == "-") type = Type.USD;
                                else if (op == "/") type = Type.Rate;
                                else type = Type.Number;
                        }
                        else if (expr1.type.classId == Type.DATE) {
                                if (op == "+" || op == "-") type = Type.Day;
                                else if (op == "/") type = Type.Rate;
                                else type = Type.Number;
                        }
                        else if (expr1.type.classId == Type.BOOLEAN) {
                                error("Error: Cannot use boolean expression in arithmetic
operation.");
                        }
                        else if (expr1.type.classId == Type.STRING) {
                                if (op != "+") error("Error: Cannot use string expression in
arithmetic operation.");
                                else type = Type.String;
                        }
                        else {
                                type = expr1.type;
                        }
                }

        }

        public Expr calculate() {
                if (expr1 instanceof Constant && expr2 instanceof Constant) {
                        Constant c = new Constant(0);
                        //Do type checks!
                        switch(s.charAt(0)) {
                        case '+':
                                c = new Constant(Double.parseDouble(expr1.s) +
Double.parseDouble(expr2.s));
                                break;
                        case '-':
                                c = new Constant(Double.parseDouble(expr1.s) -
Double.parseDouble(expr2.s));
                                break;
                        case '*':
                                c = new Constant(Double.parseDouble(expr1.s) *
Double.parseDouble(expr2.s));
                                break;
                        case '/':
                                c = new Constant(Double.parseDouble(expr1.s) /
Double.parseDouble(expr2.s));
                                break;
                        case ',':
                                c = new Constant(Math.pow(Double.parseDouble(expr1.s),
Double.parseDouble(expr2.s)));
                                break;
                        }
                        return c;
                }
                return this;
        }

        public Expr reduce() {
                 return (new Arith(s, expr1.reduce(), expr2.reduce())).calculate();
        }

        public String toString() {
                String result = "(";
                if (s == ",") {
                        result = "Math.pow(";
                }
                if (expr1.round) {
                        result += "(Math.round(" + expr1.toString() + "*100)/100.0)";
                }
                else {
```

```
                        result += expr1.toString();
                }
                result += " " + s + " ";
                if (expr2.round) {
                        result += "(Math.round(" + expr2.toString() + "*100)/100.0)";
                }
                else {
                        result += expr2.toString();
                }
                result += ")";
                return result;
//              return "(" + expr1.toString() + " " + s + " " + expr2.toString() + ")";
        }

        public void gen() {
//              System.out.print(type.toString());
                System.out.print(reduce().toString());
        }
}
```

Cast.java
author: Eric Zhang

```
public class Cast extends Expr {
	//Because Java is pass by reference, there is only
	// one copy of the underlying expressions (identifiers, for example)
	Expr expr;
//	Expr outerexpr;

	public Cast(Type t, Expr e) {
		super("CAST", t);
		if (t == Type.String) {
			if (!e.type.isNumeric()) {
				expr = e;
			}
			else {
				Type temp = e.type;
				if (temp.classId == Type.CURRENCY) {
					e.type = Type.String;
					e.round = true;
					expr = new Arith("+", new Constant("\"" + temp.symbol +
"\""), e);
				}
				else if (temp.classId == Type.DATE) {
					e.type = Type.String;
					e.round = true;
					expr = new Arith("+", e, new Constant("\"" + temp.symbol +
"\""));
				}
				else if (temp.classId == Type.RATE) {
					e.type = Type.Number;
					Arith convert = new Arith("*", e, new Constant(100));
					convert.type = Type.String;
					expr = new Arith("+", convert, new Constant("\"%\""));
				}
				else {
					expr = e;
				}
			}
		}
		else if (!(t.isNumeric() && e.type.isNumeric())) {
			error("Error: No valid cast defined between " + t + " and " + e.type);
		}
		else if (t.classId == e.type.classId) {
			if (!(t == Type.USD || t == Type.Day)) {
				expr = new Arith("*", new Constant(1/t.conversion), e);
				expr.type = t;
			}
			else {
				expr = e;
//				expr.type = t;
			}
		}
		else {
			e.type = Type.Number;
			expr = e;
			expr.type = t;
		}
	}

	public Expr reduce() {
		return expr.reduce();
	}

	public String toString() {
		return expr.toString();
	}

	public void gen() {
		expr.gen();
	}
```

```
}
```

Cond.java

```
author: Eric Zhang

public class Cond extends Stmt {
	Expr expr;
	Stmt ifstmt, elsestmt;

	public Cond(Expr e, Stmt s1, Stmt s2) {
		if (e.type != Type.Boolean) {
			error("Error: Argument to if statment must be a boolean expression");
		}
		expr = e;
		ifstmt = s1;
		elsestmt = s2;
	}

	public void gen() {
		System.out.print("if (");
		expr.gen();
		System.out.println(") {");
		if (ifstmt != null) ifstmt.gen();
		System.out.println("}");
		if (elsestmt != null) {
			System.out.println("else {");
			if (elsestmt != null) elsestmt.gen();
			System.out.println("}");
		}
	}
}
```

Console.java
author: Kelvin Jiang

```
import java.io.*;
import java.util.*;
import java.text.*;
import antlr.CommonAST;
import antlr.collections.AST;
import antlr.debug.misc.ASTFrame;
import antlr.*;

public class Console {

	public static void printUsage(boolean err) {
		if (err) {
			System.out.println("You did not enter a valid command.");
			System.out.println("Type help to list all available commands.");
		} else {
			System.out.println("Commands:");
			System.out.println("compile <filename>");
			System.out.println("run <filename>");
			System.out.println("exit");
			System.out
					.println("<filename> must a string of alphabets followed by
the extension .spx");
		}
	}

	public static String getFilename(String path) {
		String[] s = path.split("/");
		return s[s.length - 1];
	}

	public static boolean checkFilename(String path, boolean checkSpx) {
		if (checkSpx) {
			String file = getFilename(path);
			StringTokenizer st = new StringTokenizer(file, ".");
			if (st.countTokens() != 2)
				return false;
			char c[] = st.nextToken().toLowerCase().toCharArray();
			for (int i = 0; i < c.length; i++)
				if (c[i] < 'a' || c[i] > 'z')
					return false;
			return true;
		} else {
			char c[] = path.toLowerCase().toCharArray();
			for (int i = 0; i < c.length; i++)
				if (c[i] < 'a' || c[i] > 'z')
					return false;
			return true;
		}
	}

	public static Process cmdExec(String cmd) throws IOException,
			InterruptedException {
		Process p = null;
		p = Runtime.getRuntime().exec(cmd);
		p.waitFor();
		return p;
	}

	public static void run(String file) throws IOException,
			InterruptedException {

		File f = new File(file + ".class");
		if (!f.exists()) {
			System.out.println(file
					+ ".spx has not been compiled. Please compile it first.");
			return;
		}
```

```
                String str;
                BufferedReader br = new BufferedReader(new InputStreamReader(cmdExec(
                                "cmd /c start cmd /k\"java -classpath . " +
file+"\"").getInputStream()));
                while ((str = br.readLine()) != null)
                        System.out.println(str);

        }

        public static void compile(String path) throws FileNotFoundException,
                        IOException, InterruptedException, RecognitionException,
                        TokenStreamException {
                String file = getFilename(path);
                StringTokenizer st = new StringTokenizer(file, ".");
                String t = st.nextToken();

                // redirect stdout to file
                File f = new File(t + ".java");
                PrintStream printStream = new PrintStream(new BufferedOutputStream(
                                new FileOutputStream(f)), true);
                // System.setErr(printStream);
                System.setOut(printStream);

                // translate .spx to .java
                // filename = args[0];
                SimplexLexer lexer = new SimplexLexer(new DataInputStream(
                                new FileInputStream(path)));
                SimplexParser parser = new SimplexParser(lexer);
                parser.start();
                CommonAST parseTree = (CommonAST) parser.getAST();
                simplexWalker walker = new simplexWalker();
                Stmt s = walker.main(parseTree);
                System.out.println("import java.util.*;");
                System.out.println("import java.io.*;");
                System.out.println("public class " + t + " {");
                System.out
                                .println("public static BufferedReader stdin = new
BufferedReader(new InputStreamReader(System.in));");
                System.out.println("public static double readDouble() {");
                System.out.println("while (true) {");
                System.out
                                .println("try { return Double.parseDouble(stdin.readLine()); }");
                System.out
                                .println("catch (NumberFormatException e) {
System.out.println(\"That is not a valid number, please reenter.\"); }");
                System.out
                                .println("catch (IOException e) {
System.out.println(\"IOException\"); }");
                System.out.println("}}");
                s.gen();
                System.out.println("}");

                // redirect stdout to console
                System.setOut(new PrintStream(new BufferedOutputStream(
                                new FileOutputStream(java.io.FileDescriptor.out), 128), true));

                cmdExec("cmd /c javac " + t + ".java");
                File classCheck = new File(t + ".class");
                if (classCheck.exists())
                        System.out.println(file + " successfully compiled.");
                else
                        System.out
                                        .println(file
                                                        + " did not compile successfully. Please
check your syntax.");

        }

        public static void main(String[] args) {

                String cmd;
```

```
		try {

			System.out.print("> ");
			BufferedReader in = new BufferedReader(new InputStreamReader(
					System.in));
			cmd = in.readLine();
			StringTokenizer st = new StringTokenizer(cmd);
			if (st.countTokens() == 2) {
				String t = st.nextToken();
				if (t.equals("compile")) {
					String t2 = st.nextToken();
					if (checkFilename(t2, true)) {
						compile(t2);
					} else {
						printUsage(true);
					}
				} else if (t.equals("run")) {
					String t2 = st.nextToken();
					if (checkFilename(t2, false)) {
						run(t2);
					} else {
						printUsage(true);
					}
				}

			} else if (st.countTokens() == 1) {
				String t = st.nextToken();
				if (t.equals("exit")) {
					System.out.println("Goodbye!");
					return;
				} else if (t.equals("help")) {
					printUsage(false);
				} else {
					printUsage(true);
				}
			} else {
				printUsage(true);
			}

		} catch (FileNotFoundException e) {
			System.setOut(new PrintStream(new BufferedOutputStream(
					new FileOutputStream(java.io.FileDescriptor.out), 128),
					true));
			System.err.println("This file does not exist.");
		}

		catch (Exception e) {
			System.out.println("hi");
			System.err.println("exception: " + e);
		}
	}

}
```

Constant.java
author: Eric Zhang

```
public class Constant extends Expr {

	public Constant(String tok, Type p) {
		super(tok, p);
	}

	public Constant(double i) {
		super(String.valueOf(i), Type.Number);
	}

	public Constant(String s) {
		super(s, Type.String);
	}

	public static final Constant True = new Constant("true", Type.Boolean),
			False = new Constant("false", Type.Boolean);

	public void jumping(int t, int f) {
		if (this == True && t != 0)
			emit("goto L" + t);
		else if (this == False && f != 0)
			emit("goto L" + f);
	}
}
```

Decl.java
author: Eric Zhang

```
public class Decl extends Stmt {
	public String ident;
	public Type type;
	String javatype;

	//Need to define an empty constructor since Arglist extends this
	public Decl() {}

	public Decl(Type t, String id) {
		ident = id;
		type = t;
		if (t.classId == Type.BOOLEAN) {
			javatype = "boolean";
		} else if (t.classId == Type.CURRENCY || t.classId == Type.DATE
				|| t.classId == Type.CURRENCY || t.classId == Type.RATE
				|| t.classId == Type.NUMBER) {
			javatype = "double";
		} else if (t.classId == Type.STRING) {
			javatype = "String";
		}
	}

	public void gen() {
		System.out.print(javatype + " " + ident);
		if (type.isNumeric()) {
			System.out.println(" = 0;");
		}
		else if (type == Type.String) {
			System.out.println(" = \"\";");
		}
		else if (type == Type.Boolean) {
			System.out.println(" = false;");
		}
		else {
			System.out.println(";");
		}

	}

	//This is used only in 'Arglist' to not add the semicolon
	public void gen2() {
		System.out.print(javatype + " " + ident);
	}
}
```

Expr.java
author: Eric Zhang

```
public class Expr extends Node {
	public String s;
	public Type type;
	public boolean round = false;

	//Empty constructor to allow ExprList to extend
	Expr() {}

	Expr(String token, Type p) {
		s = token;
		type = p;
	}

	public void gen() {
		System.out.print(reduce().toString());
	}

	public Expr reduce() {
		return this;
	}

	public void jumping(int t, int f) {
		emitjumps(toString(), t, f);
	}

	public void emitjumps(String test, int t, int f) {
		if (t != 0 && f != 0) {
			emit("if " + test + " goto L" + t);
			emit("goto L" + f);
		} else if (t != 0)
			emit("if " + test + " goto L" + t);
		else if (f != 0)
			emit("iffalse " + test + " goto L" + f);
	}

	public String toString() {
		if ((type.classId == Type.CURRENCY || type.classId == Type.DATE) && !(type ==
Type.USD || type == Type.Day)) {
			return "(" + type.conversion + "*" + s + ")";
		}
		else {
			return s;
		}
	}
}
```

ExprList.java
author: Eric Zhang

```
import java.util.*;

public class ExprList extends Expr {
	public Vector exprs;
	public Vector args;

	public ExprList() {
		exprs = new Vector();
	}

	public void add(Expr e) {
		exprs.add(e);
	}

	public void copy(ExprList e) {
		for (int i = 0; i < e.exprs.size(); i++) {
			exprs.add(e.exprs.get(i));
		}
	}

	public int size() {
		return exprs.size();
	}

	public Type getType(int i) {
		return ((Expr)(exprs.get(i))).type;
	}

	public void genExpr(int i) {
		Expr temp = (Expr)(exprs.get(i));
		Decl dest = (Decl)(args.get(i));
		if (temp.type.classId == dest.type.classId && temp.type != dest.type
				&& (dest.type != Type.USD && dest.type != Type.Day)) {
			System.out.print(1/(dest.type.conversion) + "*");
		}
		((Expr)(exprs.get(i))).gen();
	}

	public void gen() {
		if (exprs.size() > 0) {
			genExpr(0);
			for (int i = 1; i < exprs.size(); i++) {
				System.out.print(", ");
				genExpr(i);
			}
		}
	}

}
```

For.java
author: Eric Zhang

```
public class For extends Stmt {
        Set init, incr;
        Expr expr;
        Stmt body;

        public For(Set s1, Expr e, Set s2, Stmt b) {
                init = s1;
                incr = s2;
                if (e.type != Type.Boolean) {
                        error("Error: Second argument of for loop must be a boolean expression");
                }
                expr = e;
                body = b;
        }

        public void gen() {
                System.out.print("for(");
                init.gen2();
                System.out.print("; ");
                expr.gen();
                System.out.print("; ");
                incr.gen2();
                System.out.println(") {");
                if (body != null) body.gen();
                System.out.println("}");
        }
}
```

Func.java
author: Eric Zhang

```
public class Func extends Stmt {
	String ident;
	String type;
	Arglist args;
	Stmt body;
	int classId;

	public Func(Type t, String id, Arglist a, Stmt b) {
		ident = id;
		if (t.classId == Type.BOOLEAN) {
			type = "boolean";
		} else if (t.classId == Type.CURRENCY || t.classId == Type.DATE
				|| t.classId == Type.CURRENCY || t.classId == Type.RATE
				|| t.classId == Type.NUMBER) {
			type = "double";
		} else if (t.classId == Type.STRING) {
			type = "String";
		} else if (t.classId == Type.CUSTOM) {
			type = t.name;
		}
		classId = t.classId;
		args = a;
		body = b;
	}

	public void gen() {
		if (type.indexOf("static") < 0) {
			System.out.print("static ");
		}
		System.out.print(type + " " + ident + "(");
		if (args.size() != 0) args.gen();
		System.out.println(") {");
		if (body != null) body.gen();
		if (type == "double") System.out.println("return 0;");
		else if (type == "boolean") System.out.println("return true;");
		else if (type == "String") System.out.println("return \"\";");
		else System.out.println("return;");
		System.out.println("}");
	}
}
```

FuncCall.java
author: Eric Zhang

```
public class FuncCall extends Expr {
	String ident;
	String javatype;
	ExprList args;

	public FuncCall(Id func, ExprList a) {
		ident = func.s;
		type = func.type;
		if (type.classId == Type.BOOLEAN) {
			javatype = "boolean";
		} else if (type.classId == Type.CURRENCY || type.classId == Type.DATE
				|| type.classId == Type.CURRENCY || type.classId == Type.RATE
				|| type.classId == Type.NUMBER) {
			javatype = "double";
		} else if (type.classId == Type.STRING) {
			javatype = "String";
		} else if (type.classId == Type.CUSTOM) {
			javatype = type.name;
		}
		args = a;
		if (args.size() != func.args.size()) {
			error("Error: Wrong number of arguments to function " + func);
		}
		for (int i = 0; i < args.size(); i++) {
			if (args.getType(i) == Type.Void) {
				error("Error: Passing void type as function argument");
			}
			if (args.getType(i).isNumeric() != func.args.getType(i).isNumeric() &&
func.args.getType(i) != Type.String) {
				error("Error: Passing incompatible type to function " + func);
			}
		}
		//Send the decls vector to the ExprList for generation
		// This is so args.gen() produces the correct currency conversion factors
		args.args = func.args.decls;
	}

	public void gen() {
		if ((type.classId == Type.CURRENCY || type.classId == Type.DATE) && !(type ==
Type.USD || type == Type.Day)) {
			System.out.print(type.conversion + "*");
		}
		gen2();
	}

	public void gen2() {
		System.out.print(ident + "(");
		if (args.size() != 0)
			args.gen();
		System.out.print(")");
	}
}
```

FuncCallStmt.java

```
author: Eric Zhang

public class FuncCallStmt extends Stmt {
	FuncCall func;

	public FuncCallStmt(FuncCall f) {
		func = f;
	}

	public FuncCallStmt(Id f, ExprList a) {
		func = new FuncCall(f, a);
	}

	public void gen() {
		func.gen2();
		System.out.println(";");
	}
}
```

Id.java
author: Eric Zhang

```
public class Id extends Expr {
	public Arglist args = new Arglist();

	public Id(String id, Type p) {
		super(id, p);
	}

	public Id(String id, Type p, Arglist a) {
		super(id, p);
		args = a;
	}

	public void gen() {
		if ((type.classId == Type.CURRENCY || type.classId == Type.DATE) && !(type ==
Type.USD || type == Type.Day)) {
			System.out.print(type.conversion + "*");
		}
		System.out.print(s);
	}

	//Used for left side of assignment
	public void gen2() {
		System.out.print(s);
	}
}
```

Input.java
author: Eric Zhang

```
public class Input extends Stmt {
	Id var;

	public Input(Id v) {
		var = v;
		if (var.type == Type.Boolean) {
			error("Error: Boolean variables cannot be read from the console.");
		}
	}

	public void gen() {
		var.gen2();
		if (var.type == Type.String) {
			System.out.println(" = stdin.readLine();");
		}
		else if (var.type == Type.Rate){
			System.out.println(" = readDouble()/100.0;");
		}
		else {
			System.out.println(" = readDouble();");
		}
	}
}
```

Node.java
author: Eric Zhang

```
public class Node {
	void error(String s) {
		System.err.println(s);
		System.exit(0);
	}

	public static void emit(String s) {
		System.out.println("\t" + s);
	}
}
```

Not.java
author: Eric Zhang

```
public class Not extends Expr {
	Expr expr;

	public Not(Expr e) {
		super("!", Type.Boolean);
		if (e.type != Type.Boolean) {
			error("Error: Argument to Not operator must be a boolean expression");
		}
		expr = e;
	}

	public void gen() {
		System.out.print("!(");
		expr.gen();
		System.out.print(")");
	}
}
```

printFunction.java
author: Eric Zhang

```
public class printFunction extends Stmt {
	Expr line;

	public printFunction(Expr e) {
		line = new Cast(Type.String, e);
	}

	public void gen() {
		System.out.print("System.out.println(");
		System.out.println(line + ");");
	}
}
```

Rel.java
author: Eric Zhang

```
public class Rel extends Expr {
	Expr expr1, expr2;

	public Rel(String op, Expr e1, Expr e2) {
		super(op, Type.Boolean);
		//If operator is AND or OR, only boolean expressions are ok
		if (op == "&&" || op == "||") {
			if (!(e1.type == Type.Boolean && e2.type == Type.Boolean)) {
				error("Error: && and || operators must take boolean expressions as
operands");
			}
		}
		//If both are numerics, all operations are ok
		//Otherwise:
		if (!(e1.type.isNumeric() && e2.type.isNumeric())) {
			if (!(e1.type == e2.type && op == "==")) {
				error("Error: Incompatible types in comparison");
			}
		}
		expr1 = e1;
		expr2 = e2;
		if (expr1.type.classId != expr2.type.classId) {
			if (expr1.type == Type.Number) {
				expr1.type = expr2.type;
			}
			else if (expr2.type == Type.Number) {
				expr2.type = expr1.type;
			}
		}
	}

	public void gen() {
		//If either expression is a currency, do the rounding
		// round to three decimal places, and then compare.
		boolean round = false;
		if (expr1.type.classId == Type.CURRENCY || expr2.type.classId == Type.CURRENCY) {
			round = true;
			System.out.print("Math.round((");
		}
		expr1.gen();
		if (round) {
			System.out.print(")*1000)");
		}
		System.out.print(" " + s + " ");
		if (round) {
			System.out.print("Math.round((");
		}
		expr2.gen();
		if (round) {
			System.out.print(")*1000)");
		}
	}
}
```

Return.java
author: Eric Zhang

```
public class Return extends Stmt {
	Expr returnValue;
	Type returnType;

	public Return(Type t, Expr e) {
		//Check to make sure return value is consistent with function type
		if (t == Type.Void) {
			if (e != null) {
				error("Error: Void function cannot return value");
			}
		}
		else {
			if (e == null) {
				error("Error: Null value in return statement of function of type "
+ t);
			}
			else if (t != Type.String) {
				if (t.isNumeric() != e.type.isNumeric()) {
					error("Error: Return type not consistent with function
type");
				}
			}
		}
		returnType = t;
		returnValue = e;
		hasReturn = true;
	}

	public void gen() {
		System.out.print("if (true) return");
		if (returnValue != null) {
			if (returnType.classId == returnValue.type.classId && returnType !=
returnValue.type) {
				System.out.print(1/(returnType.conversion) + "*");
			}
			System.out.print(" ");
			returnValue.gen();
		}
		System.out.println(";");
	}
}
```

Seq.java
author: Eric Zhang

```
public class Seq extends Stmt {
	Stmt stmt1;

	Stmt stmt2;

	public Seq(Stmt s1, Stmt s2) {
		stmt1 = s1;
		stmt2 = s2;
		if (s1.hasReturn || s2.hasReturn) hasReturn = true;
	}

	public void gen() {
		stmt1.gen();
		stmt2.gen();
	}
}
```

Set.java
author: Eric Zhang

```
public class Set extends Stmt {
	public Id var;
	public Expr e;
	public String op;

	public Set(String o, Id lhs, Expr rhs) {
		op = o;
		var = lhs;
		e = rhs;
		if (e == null) {
			if (op.charAt(1) == '+') {
				op = "+=";
				e = new Constant(1);
			}
			else if (op.charAt(1) == '-') {
				op = "-=";
				e = new Constant(1);
			}
		}
		if (op.charAt(0) == '^') {
			e = new Arith(",", var, e);
			op = "=";
		}
		if (var.type.isNumeric() != e.type.isNumeric() && var.type != Type.String) {
			error("Error: Assignment of incompatible types");
		}
	}

	public void gen() {
		gen2();
		System.out.println(";");
	}

	//Used for For loop to not output semicolon
	public void gen2() {
		var.gen2();
		System.out.print(" " + op + " ");
		//Add conversion factor if needed
		if (var.type.classId == e.type.classId && var.type != e.type
				&& (var.type != Type.USD && var.type != Type.Day)) {
			System.out.print(1/(var.type.conversion) + "*");
		}
		e.gen();
	}
}
```

Stmt.java
author: Eric Zhang

```
public class Stmt extends Node {
        public boolean hasReturn = false;
        String str;

        public Stmt() {
        }

        public Stmt(String s) {
                str = s;
        }

        public void gen() {
                System.out.println(str + ";");
        }
}
```

SymbolTable.java

```
author: Eric Zhang, Gilbert Hom

import java.util.*;

public class SymbolTable {
	private Hashtable table;

	protected SymbolTable outer;
	public Type returnType = null;

	public SymbolTable(SymbolTable st) {
		table = new Hashtable();
		outer = st;
	}

	public SymbolTable(SymbolTable st, Type t) {
		table = new Hashtable();
		outer = st;
		returnType = t;
	}

	public void put(String token, Type t) {
		if (get(token) != null) {
			System.out.println("Variable already declared: " + token);
			System.exit(0);
		}
		table.put(token, new Id(token, t));
	}

	public void put(String token, Type t, Arglist a) {
		table.put(token, new Id(token, t, a));
	}

	public Id get(String token) {
		for (SymbolTable thisTable = this; thisTable != null; thisTable = thisTable.outer)
{
			Id id = (Id) (thisTable.table.get(token));
			if (id != null) {
				//create a new Id in the return so that each Id in the IR is a
separate instance
				Id result = new Id(id.s, id.type);
				result.args = id.args;
				return result;
			}
		}
		return null;
	}

	public Type getReturnType() {
		SymbolTable temp = this;
		while (temp.returnType == null) {
			temp = this.outer;
		}
		return temp.returnType;
	}

}
```

TestSuite.java
author: Kelvin Jiang

```
import java.io.*;
import java.util.*;

public class TestSuite {

	public static void main(String[] args) {
		try {
			Vector tests = new Vector();
			Process p = Console.cmdExec("cmd /c dir /b tests");

			BufferedReader br = new BufferedReader(new
InputStreamReader(p.getInputStream()));
			String str;
			while ((str = br.readLine()) != null)
				tests.add((String) str);

			for(int i=0;i<tests.size();i++) {
				String file = (String)tests.elementAt(i);
				Console.compile("tests/"+file);
				StringTokenizer st =new StringTokenizer(file,".");
				file = st.nextToken();
				Console.run(file);
				Console.cmdExec("cmd /c del "+file+".java "+file+".class");
			}
			System.out.println("Testing Completed.");
		} catch (Exception e) {

		}
	}

}
```

Type.java
author: Eric Zhang

```
public class Type {
	public String name = "";
	public int classId = 0;
	public double conversion = 1;
	public String symbol = "";

	public Type(String s, int cls, double conv, String symb) {
		name = s;
		classId = cls;
		conversion = conv;
		symbol = symb;
	}

	// Type Classes
	public static final int CUSTOM = -1;

	public static final int NUMBER = 0;

	public static final int RATE = 1;

	public static final int DATE = 2;

	public static final int CURRENCY = 3;

	public static final int BOOLEAN = 4;

	public static final int STRING = 5;

	// Types
	public static final Type
			Void = new Type("void", CUSTOM, 1, ""),
			Number = new Type("number", NUMBER, 1, ""),
			Rate = new Type("rate", RATE, 1, ""),
			Day = new Type("day", DATE, 1, "d"),
			Month = new Type("month", DATE, 30, "m"),
			Year = new Type("year", DATE, 365, "y"),
			USD = new Type("USD", CURRENCY, 1, "$"),
			EUR = new Type("EUR", CURRENCY, 1.3252, "\u20AC"),
			CAD = new Type("CAD", CURRENCY, 0.867302689, "CAD"),
			GBP = new Type("GBP", CURRENCY, 1.9692, "\u00A3"),
			AUD = new Type("AUD", CURRENCY, 0.7876, "AUD"),
			ILS = new Type("ILS", CURRENCY, 0.238766057, "ILS"),
			CNY = new Type("CNY", CURRENCY, 0.127785729, "CNY"),
			MXN = new Type("MXN", CURRENCY, 0.0922653922, "MXN"),
			SOS = new Type("SOS", CURRENCY, 0.0008037, "SOS"),
			YEN = new Type("YEN", CURRENCY, 0.00853679358, "\u00A5"),
			Boolean = new Type("boolean", BOOLEAN, 1, ""),
			String = new Type("string", STRING, 1, "");

	public boolean isNumeric() {
		return classId == Type.NUMBER || classId == Type.RATE
				|| classId == Type.DATE || classId == Type.CURRENCY;
	}

	public static Type max(Type p1, Type p2) {
		if (p1 == Type.String || p2 == Type.String)
			return Type.String;
		else if (p1 == p2)
			return p1;
		else if (p1.isNumeric() && p2.isNumeric())
			return Type.Number;
		else
			return null;
	}

	public String toString() {
		return name;
	}
```

```
}
```

Unary.java
author: Eric Zhang

```
public class Unary extends Expr {
	Expr expr;

	public Unary(String op, Expr e) {
		super(op, null);
		if (e.type.classId == Type.CURRENCY) {
			type = Type.USD;
		}
		else if (e.type.classId == Type.DATE) {
			type = Type.Day;
		}
		else {
			type = e.type;
		}

		if (!e.type.isNumeric()) {
			error("Error: Argument to negation operator must be a numeric
expression");
		}
		expr = e;
	}

	public void gen() {
		System.out.print("(");
		if (s == "-") System.out.print(s);
		System.out.print("(");
		expr.gen();
		System.out.print(")");
		if (s == "++" || s == "--") System.out.print(s);
		System.out.print(")");
	}
}
```

While.java
author: Eric Zhang

```
public class While extends Stmt {
	Expr expr;
	Stmt body;

	public While(Expr e, Stmt s) {
		if (e.type != Type.Boolean) {
			error("Error: Argument to if statment must be a boolean expression");
		}
		expr = e;
		body = s;
	}

	public void gen() {
		System.out.print("while (");
		expr.gen();
		System.out.println(") {");
		if (body != null) body.gen();
		System.out.println("}");
	}
}
```