SimpL: The Simple Programming Language ©

Getting Started

```
See docs/Simpl Setup.pdf
```

Language Overview

```
See docs/Simpl Overview.pdf
```

Project Structure

```
docs
src
    exceptions
    gen
    main
    utils
    SimpLCompiler.java
    SimpLMain.java
```

```
SimpL                   # project directory
    \lib                # third party library jars

    \examples           # SimpL programs examples

    \docs               # language overview, diagrams, etc

    \src                # source directory
        \exceptions     # collection of exceptions specific to SimpL
        \gen            # antlr generated listener/visitor/parser/tokens files
        \main           # core code including message handling, visitor, jasmin code emitter
        \utils          # various utilities such as files

    \tests              # all SimpL test files go here
        \basic          # small, basic tests
        \comprehensive  # larger, more comprehensive tests

    simplc.sh           # script to compile a source input (.simpl) file

    simplr.sh           # script to run a compiled source (.class) file
```

```
build.sh            # script to generate antlr sources, compile codebase, and run tests

...                 # readme/simpl scripts/etc files go here in outermost directory
```

SimpL: The Simple Programming Language ©

Setup

Building SimpL Source

To build SimpL, navigate to the SimpL directory and run:

```
./build.sh
```

The above will generate antlr sources, compile all java into an `out` folder, and run various tests. The project is built with the only dependencies being the JVM assembler `Jasmin` and the parser generator `Antlr` .

Compiling a SimpL program

To compile a `.simpl` file, navigate to the SimpL directory and run:

```
./simplc.sh <source_filepath>
```

The above will generate .j (JVM assembly) and a .class (JVM bytecode) files of the same name. The output location defaults to the parent directory of the given sourcefile, but can be specified with the command line option `-d` (to be added in the near future).

Executing a SimpL program

To execute a program, run it by specifying its compiled class filepath by using:

```
./simplr.sh <class_filepath>
```

The above will run the compiled SimpL program with any output printed to console.

Troubleshooting and Tips

If executing `java` or `javac` directly, ensure classpath is set correctly by using:

```
export CLASSPATH="out:<jasmin2.4-jar-path>:<antlr4.7-jar-path>:$CLASSPATH"
```

If a `permission denied` error occurred while running a script, grant access by using:

```
chmod +x ./<script_filepath>.sh
```

If newline issues occur after modifying the scripts on windows, remove excess new line characters by using:

```
sed -i 's/\r$//' ./<script_filepath>
```

Delete any generated jasmin and class files by using (optionally limit depth by adding `-maxdepth 1`):

```
find <output_directorypath> -regex ".*\.\(j\|class\)" -type f -delete
```

SimpL: The Simple Programming Language ©

Motivation and Summary

SimpL is a *simple* language for *simple* people, who are tired of overbearing syntax or are simply just learning to program. Elements of SimpL's syntax were influenced by Python, with all statements ending in a line break instead of the more traditional semicolon.

Unlike Python, however, SimpL is a statically typed, dynamically scoped language that compiles and runs on the JVM. It was built using `Jasmin` (JVM assembly) for intermediate code generation, and `ANTLR4` for parser source code generation.

SimpL programs are single files ending with a `.simpl` extension, that when compiled generates respective .j (Jasmin) and .class (bytecode) files.

General Syntactic Elements

Only *single programs* are supported, of which consistent of multiple statements, each of which are terminated with a line break. Any curly braces must be on their own separate lines -- no egyptian style braces, sorry!

Expressions

An `expression` is any mix of parenthetical expressions, datatype literals, identifiers, function calls, and operations.

Statements

A `name` is an underscore or letter followed by any combination of underscores, letters, or numbers. This applies to both function and variable names.

A `statement` is a declaration, assignment, standalone expression, function definition, conditional, and while loop. Of those, the last three are multiline statements with the following syntax:

```
<keyword> <expression>
{
    <0 or more statements>
    <optional return statement>
}
```

Note that the above braces and their enclosed statements form a block. Thus, blocks follow all function signatures, as well as all while, if, else if, else conditions.

Comments

Any `comment` is ignored by the SimpL parser, along with tabs and excess newlines.

Single line comments have the following syntax:

```
## <some single line comment>
```

Multiline comments have the following syntax:

```
##
<some multiline comment>
##
```

Datatypes

Number

An integer or decimal number (internally stored as a *double*), with the following syntax for literals:

```
<1 or more digits>
<1 or more digits>.<1 or more digits>
```

Text

A character sequence (internally stored as a *String*), with the following syntax for literals:

```
'<0 more characters>'
```

Note that quotes and slashes can be escaped with a slash (eg ' \).

Boolean

A true or false token (internally stored as a *boolean*), with the following syntax for literals:

```
True
False
```

Operators

Support for parenthetical, arithmetic, boolean, comparison operations, ordered from high to low precedence:

```
---------------- Operator Precedence ---------------
| order |  operator  |           meaning           |
|   0   |     ()     |         parenthesis         |
|   1   |     ^      |        exponentiation       |
|   2   |    * /     |     multiply and divide     |
|   3   |    + -     |       add and subtract      |
|   4   | < > <= >=  |          comparison         |
|   5   |   == !=    |   equality and inequality   |
|   6   |    not     |       logical negation      |
|   7   |    and     |     logical conjunction     |
|   8   |    or      |     logical disjunction     |
|   8   |    =       |          assignment         |
----------------------------------------------------
```

Equality, parenthesis, and assignment operators apply to *any* expression and *any* datatype. Comparison and arithmetic operators apply only to `Numbers`. Logical operators apply only to `Booleans`.

Variables, Declarations, and Assignments

Support for variables is restricted to datatypes `Number` and `Text`. Variables can declared with or without an initial value, as follows:

```
<datatype> <name> = <expression>
<datatype> <name>
```

Control Flow

Syntax for conditionals is as follows:

```
if <expression>
{
    <0 or more statements>
}
elif <expression>
{
    <0 or more statements>
}
else
{
    <0 or more statements>
}
```

Syntax for loops is as follows:

```
while <expression>
{
    <0 or more statements>
}
```

Functions

Function Definitions

A `function` definition consisting of a signature and a body, with the following syntax:

```
<void or datatype> <name>(<parameter list>)
{
    <0 or more statements>
}
```

If the return type is void, then nothing is returned within the function. Above, `<parameter list>` is comma separated list of one more `<datatype> <name>` .

Function Calls

A `function` defined within the current scope is invoked, with the following syntax:

```
<name>(<argument list>)
```

Above, `<argument list>` is defined a comma separated list of one or more `<name>` . When calling functions, arguments are passed by value, and must correspond to the parameters specified in the function definition.

Builtin Functions

Predefined functions `print` and `println` are available, and take any number of arguments. Just like in Java, `print` writes the expression to standard out,

Error Handling

Errors and exceptions encountered during compilation are written to standard error.

Type Checking

Errors are raised to ensure operators are between appropriate type(s), as specified in the *Operators* section from before.

Error Recovery

Like most compilers, `SimpL` will continue parsing the rest of the file even if an error occurs.