SIMPLEX
LANGUAGE REFERENCE MANUAL

Steven Chen Gilbert Hom Kelvin Jiang Eric Zhang
{stc2104, gch2102, kxj1, ehz2101}@columbia.edu

INTRODUCTION

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 tokens 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;
```

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`. 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. Using any operator on a type without an implicit cast defined (i.e. multiplication on two strings) will result in a compile time error.

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 may be omitted if desired.

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.