Sortle

A Programming Language Based on Insertion Sort

Sortle is a programming language based on expressions, lists, and sorting. A program is written as a list of named expressions. The expressions are evaluated in sorted order. An expression is defined like this:

name := expression

Each expression evaluates to a string. The string becomes the new name of the expression. If the string is “” (the null string), the expression is deleted.

Expression Names

An expression's name initially must consist only of the letters of the English alphabet. Case is significant. During runtime, an expression may rename itself to any sequence of bytes, provided the sequence does not contain a 0 byte. No limit is placed on the length of an expression name.

Expressions

Each expression has its own stack, which lasts only until the expression's evaluation is finished. The stack begins empty and must end with exactly one value on it. This final value is used as the new name for the expression. If it is a number, it will be converted to a string.

Expressions may in general contain terms, which either push literal, immediate values (numerical or string) onto the stack, or are operators, pulling values off the stack, performing operations on them, and pushing the result back on the stack. Formally, the terms supported by Sortle are *literal numbers*, *literal strings*, and *operators*.

Data Types

Any particular value on the stack can be a number or a string. Numbers are positive integers. All calculations regarding numbers, including conversions to or the importing of literal numbers during interpretation, are done modulo 4,294,967,296. Strings are arbitrarily long sequences of bytes. They may not contain a zero byte, but they are otherwise unrestricted.

Operators expect a particular one of these two types to work on, and they will convert as necessary. Numbers are represented in strings using the decimal system. Numbers converted to strings use as few digits as possible; thus, the number 0 becomes the null string.

Operators

Each operator requires two operands and produces one result. The operands we will call *op1* and *op2*. *op1* is the value pulled from the forefront of the stack, while *op2* can be considered to be “deeper down.”

The following operators each expect numbers and produce a number:

+ : Sums *op1* and *op2*.

* : Multiplies *op1* and *op2*.

/ : Divides *op1* into *op2*, producing a truncated (rounded down) result.

% : Divides *op1* into *op2*, producing the remainder of this operation.

The following operators each expect strings and produce a string:

^ : Compares *op1* and *op2* using methodology comparable to that of the C language's *strcmp* function, producing the string that compares as greater.

~ : Provides the result of concatenating *op1* onto the end of *op2*.

? : Performs *regex matching*, which will be described below.

$: Provides the null string if both operands are null strings, the operand that compares greater out of *op1* and *op2* if neither is a null string, or the operand that is not a null string if one operand is a null string.

Regex matching is to be explained in the following section.

Regex Matching

Regex matching involves comparing a series of strings against a pattern and producing either the first string that matches, or, depending on the pattern, a substring of the first match. Regex matching is done with the ? operator.

Patterns are strings, but with some enhancements. Instead of matching a byte, you can group several bytes and match an element. To do this, enclose the element in brackets. “[fun]d” looks for the element fun, followed by the element d. This could also be written “[fun][d]”. Element groups may not be nested.

This feature becomes more useful when elements are repeated. The ! modifier allows the element that preceded it to appear any number of times, provided it appears at least once. “[fun]!d” will match “funfund”, for instance. The @ modifier allows the element that preceded it to appear either once or not at all. Thus “[fun]@d” will match “fund” or “d”. Note that the regex matcher is generally lazy, and it will match elements as few times as it can get away with.

A dot “.” can be used to match any byte. Dots may be used within element groups.

Parentheses specify special groups that will be captured, rather than the full string being captured. For instance, if “[f.n]!d” matches “finfund”, the result produced will be “finfund”, but if “(f.n)!d” is used instead, the result will be “fin”. Parenthetical groups form elements just like bracketed groups, so they cannot be used within brackets.

The regex match operator ? uses *op2* as its pattern. If *op1* is the null string, every expression name other than that of the current expression is tested. Otherwise, every

substring of *op1* is tested, starting with one-byte substrings from left to right.

Control Flow

As has been hinted at, the expressions in a program are initially sorted on program load and are maintained in sorted order.

Execution begins at the then first expression in the list and continues downwards. Note, however, that an expression may, in renaming itself, cause its position in the list to be changed. Only after such a move is the next expression to evaluate decided upon.

If execution drops off the bottom of the list, it wraps around to the top and the first expression is evaluated again.

Deletion

An expression may choose to delete itself by renaming itself to the null string. When an expression does so, it may be said to have committed *suicide*. The expression is removed from the list and will no longer be evaluated.

After such an action, control flow passes to the expression below where the deleted expression was.

Clobbering

Another way to get rid of an expression is by *clobbering* it. If an expression evaluates to the name of another expression, the latter expression is *clobbered* and replaced by the former. The usual rules for control flow apply in this case.

Termination and Output

Sortle programs terminate when only one expression remains in the list. This may happen as the result of a suicide, a clobbering, or the loading of a source file that contains only one expression to begin with.

When a program terminates, the name of the only remaining expression is printed out to the user's terminal. This is the only method of output available to a Sortle programmer.

Source Code Format

Sortle source code files are text files containing expressions represented like this:

[name] := [expression]

As previously noted, the name must initially consist only of uppercase or lowercase letters of the English alphabet.

Expressions are represented by listing their terms in order from left to right, separated by spaces. Literal numbers are written in decimal. Literal strings are enclosed within double quotes.

Literal strings may contain *escape sequences*, backslashes (\) followed by two hexadecimal digits, specifying a byte. Use of escape sequences is required for newlines,

double quote marks, backslashes, and any other bytes that might confuse the parser.

Source code lines may contain comments. A comment begins with an octothorpe, which must not be inside a literal string, and ends at the end of the line. Comments are disregarded by the parser. Lines consisting of only whitespace and comments are disregarded as well.

By convention, Sortle source code files are named with a ".sort" extension, although the use of this extension is not generally required.

The comment mechanism makes Sortle an ideal language for scripting and automation of repetitive tasks. On a UNIX system, one need only place "#!/usr/bin/sortle" on the first line of the source file, and it will be immediately executable. Note that Sortle ignores the line, so its use is a special case of a polyglot.

Practical Programming

Since expressions in Sortle can only modify their own names and can only observe the names of other expressions, maintaining state requires at least two expressions. One approach might be to use an *observer*, an expression that waits for a certain condition, retaining its original name until the condition happens, and then deletes itself later when a second condition is met.

Anyone who has created a nontrivial program in Sortle is encouraged to contact the author with details.

Availability

A reference implementation of the Sortle interpreter in the C language may be downloaded from the website:

http://www.esolangs.org/

Although written by the author of this document, the implementation available there is not guaranteed to be free of deviations.

Author

The author is the language's designer and original implementor and may be contacted through electronic mail at <*graue@oceanbase.org*>.