gscc
A General Search and Compare Compiler

LANGUAGE
REFERENCE
MANUAL

gscc is a text manipulation language that rivals existing programmatic solutions. It is compact, intuitive and lightweight, giving programmers a means to quickly manipulate their text-based targets.

Eric [G]arrido
Russel [S]antillanes
Casey [C]allendrello
Ho Yin [C]heng

TABLE OF CONTENTS

1. Introduction

gscc uses simple English commands to manipulate text in complex ways. gscc will present a small number of commands to the programmer that will allow for huge possibilities of utility. The commands allow the programmer to perform the few basic text manipulations: insertion, deletion, appending, and searching. The programmer can perform these operations based on regular expressions and other logic-based decisions.

gscc iterates over each line and performs all defined commands on that line sequentially, only passing through each file once. As text processing requirements vary, the user will be able to specify domain-specific subroutines to acquire the desired result. The user can extend the built-in capabilities of the language by creating subroutines. These subroutines also allow for high amounts of code reuse and simplify potentially-complex code.

2. Lexical Conventions

2.1. Tokens

Tokens define identifiers, keywords, constants, string literals, line terminators, operators, and separators. The entirety of the program is made up of tokens separated by white space. All white space is ignored except when it is used to separate tokens.

2.2. Comments

Comments can be used through the program to leave notes and/or documentation. They begin with the characters %* and terminate with the characters *%. Alternatively, comments can begin with % and end with a line separator (see below definition). You cannot nest comments.

2.3. Identifiers

An identifier is a sequence of letters and digits; the underscore character '_' counts as a letter. Upper and lower case letters are different. The first character must be a type prefix of either # or $ Identifiers must have a length of at least 2 characters (including the type prefix). Identifiers cannot be a keyword or the null literal. Identifiers also include the special location variables @match and @line. All identifiers have a zero index.

2.4. Keywords

func	line	while
set	global	else
replace	insert	prerr
return	print	break
length	start	substr
delete	if	end
test	match	setchar

2.5. Constants

2.5.1. Integer Constants

An integer constant is a sequence of one or more decimal digits.

2.5.2. Character Constants

A character constant is a sequence of one or more characters enclosed in single quotes, such as ‘x’. Character constants do not contain the ‘ character or newlines; in order to represent these, and other characters, the following escape sequences may be used

Newline	NL(LF)	\n
Horizontal tab	HT	\t
carriage return	CR	\r
single quote	‘	\’
double quote	“	\”
backspace	BS	\b
formfeed	FF	\f
backslash	\	\\

2.6. White space

Some white space is required to separate adjacent identifiers, keywords, and constants. White space is defined as the ASCII space and horizontal tab characters, as well as line terminators.

2.7. String Literals

A string literal is a sequence of characters surrounded by double quotes. String Literals can contain newline or escaped double quotes.

2.8. Line Terminator

Lines are terminated by the ASCII characters CR, or LF, or CR LF. The two characters CR immediately followed by LF are counted as one line terminator, and not two.

2.9. Separators

The following ASCII characters are the separators

/	)
{	;
}	,
(	.

2.10. Operators

The following objects represent operators:

!	<=
*	>=
/	= =
+	! =
–	<
&&	>
||	

3. Identifiers

Similar to variables in other languages, identifiers are used to store information in gscc. There are only three types of identifiers in gscc: strings, integers, and locations. Identifiers do not need to be formally declared by the user. Instead, upon being called by a system command, such as ‘set’, the identifier will be implicitly declared. If an identifier is referenced and no value was previously defined to it by the ‘set’ system call, then it will return the null string.

The three types of identifiers can be broken down further into basic variables and system variables. Strings and integers fall under the basic variables category while locations are considered system variables. The difference between the two is that the former can have their initial value set by the user, whereas the latter has its value predefined by the interpreter. In addition to this, any number of basic variables can be created for use by the user while the number of system variables is predetermined.

3.1. Basic Variables

The two basic variables are strings, denoted by a $, and integers, denoted by a #. Strings will take on the string value of whatever expression is set to it, while integers take on the integer value of the expression set to it. With automatic conversion, strings will take on the string value of any number set to it, but for integers, the expression set to it must, overall, be an integer value or else an error is thrown.

Type: STRING
Syntax: $<token>
Examples:

```
%* this will store the value "Four score and seven years ago”
into the identifier $text *%
        set $text, "Four score and seven years ago";

%* this will automatically convert the resulting integer expression
into the string “12345” and store it into $count *%
        set $count, 12000 + 345;
```

Type: INTEGER
Syntax: #<token>
Examples:

```
% sets the value in #three to 3
        set #three, 3;

%* since the overall expression of #three + 4 is an integer,
this set is valid and will store 7 into #seven *%
        set #seven, #three + 4;
```

3.2. System Variables

All system variables are of a special type known as a location, starting with a given prefix @. Locations are set by the interpreter, and represent a location within a given input. They cannot be declared by the user and there are only two location variables: @match and @line.

Name: @match
Description: The string that matched the regular exp.

Name: @line
Description: The line containing that string.

Location attributes:
Locations have a few attributes of type integer:

```
@<loc_var>.line      % Line number
@<loc_var>.start     % starting column/character, inclusive
@<loc_var>.end       % ending column/character, inclusive
@<loc_var>.length    % length of the string value of location
```

Context: When a location is passed to an action or function that expects a string, it is converted to a string automatically by the parser.

4. Expressions

4.1. Definition

An expression (denoted by <expression>) is any group of identifiers separated by operators that are to be evaluated by the program and whose resulting value is to be used by a command, statement, or function. The overall value of an expression depends upon its expected value and also by its components. A combination of strings and/or integers will be evaluated to a string result, while a combination of all integers will be evaluated to either an integer result (if the call is expecting an integer) or a string result (if the call is expecting a string).

4.2. Evaluation Order

All the expressions are grouped and evaluated left to right.

4.3. Precedence

The precedence of the operators is as follows (highest precedence on left/top, lowest on right/bottom).

```
                    !
                  *   /
                  +   -
  <   <=   >   >=    ==   !=   &&   ||   ,
```

4.4. Unary Minus Operator

The operand of the unary – operator must be a number, and the result is the negative of the operand. The result of applying the unary minus operator to a signed operand is equivalent to the negative promoted type of the operand. Negative zero is zero.

4.5. Logical Not Operator

The logical not operator is denoted by the ! character. It negated the value of the <expression> that immediately follows it.

4.6. Multiplicative Operators

Syntax: <integer1> * <integer2>
Operator: *
Description: The * operator performs multiplication, resulting in the product of its operands. It multiplies the value of <integer1> by the value of <integer2>.

Syntax: <integer1> / <integer2>
Operator: /
Description: The / operator performs division, resulting in the quotient of its operands. The left-hand operand is the dividend, and the right-hand operand is the divisor. It divides the value of <integer1> by the value of <integer2>.

4.7. Additive Operators

Syntax: <integer1> + <integer2> | <string1> + <string2>
Operator: +
Description: The + operator performs addition when applied to two operands of numeric type, and concatenation when applied to two operands of type string. Numeric addition is commutative and associative.

Syntax: <integer1> - <integer2>

Operator: -
Description: The – operator performs subtraction, resulting in the difference of the two operands. This subtracts <integer2> from <integer1>.

4.8. Relational Operators

Syntax: <expression> < <expression>
Operator: <
Description: The less than relational operator yields 0 if the relation is false and 1 if it is true. The type of the result is number.

Syntax: <expression> > <expression>
Operator: >
Description: The greater than relational operator yields 0 if the relation is false and 1 if it is true. The type of the result is number.

Syntax: <expression> <= <expression>
Operator: <=
Description: The less than or equal relational operator yields 0 if the relation is false and 1 if it is true. The type of the result is number.

Syntax: <expression> >= <expression>
Operator: >=
Description: The greater than or equal relational operator yields 0 if the relation is false and 1 if it is true. The type of the result is number.

4.9. Equality Operators

Syntax: <expression> = = <expression>
Operator: = =
Description: The equals relational operator yields 0 if the relation is false and 1 if it is true. The type of the result is number.

Syntax: <expression> != <expression>
Operator: !=
Description: The not equals relational operator yields 0 if the relation is false and 1 if it is true. The type of the result is number.

4.10. Logical Operators

Syntax: <expression1> && <expression2>
Operator: &&
Description: The logical and operator yields 0 (false) if either of the Boolean values of <expression1> and <expression2> is false. It yields 1 (true) is both <expression1> and <expression2> yield Boolean values of true.

Syntax: <expression1> || <expression2>
Operator: ||
Description: The logical or operator yields 0 (false) if both <expression1> and <expression2> have Boolean values of false. It yields 1 (true) if either <expression1> or <expression2> are true.

4.11. Comma Operator

Syntax: <expression> , <expression>
Operator: ,
Description: The comma operator is a separator in lists of function arguments and therefore is required to be in a parenthetical grouping such as:

```
func #myhouse( #var, $string)
```

5. Syntax Notation

A gscc program is broken up into any number of global variable definitions, function blocks, and regular expression blocks. Global variable definitions are "set" system commands used to associate a value to a variable which will be referenced as a global variable. Function blocks (See Section 9) are blocks of commands and statements grouped together by the user for use throughout the program. Regular expression blocks (See Section 6) are the main way to execute commands and statements.

A block is a sequence of statements, system commands, and function calls contained within a set of open and close brackets. Blocks are used in the syntax of regular expressions, functions, and statements. Nesting of these various blocks are allowed and encouraged in order to create more robust programs. The structure of a gscc program is mainly determined by these blocks as defined below.

Program structure:
[global variable definitions]*
[function declarators/blocks]*
<regular expression block>*

6. Regular Expression Blocks

Regular expression blocks are the basic structure used in gscc to determine which commands are execute and when they are to be executed. Firstly, the 'global' or 'line' keyword determines whether the commands are called once per match, or once per match per line. This is similar to the global switch in SED.

The way the program works is that the interpreter has a queue of lines that is the input. A line is removed from the top, and then compared against each regular expression block. If the line is modified by any particular block, the modified line is presented to the subsequent block. The rules are parallelized while one pass through the file is made.

The line includes the newline character, so that entire lines may be deleted. If a particular block creates a line, the newly created line will be pushed on to the top of the incoming queue once that block finishes executing, while the remaining line will continue executing through the rest of the blocks.

6.1. regexp Blocks

Name: REGEXP
Type: GLOBAL
Syntax: /<regexp>/ global {
    [command | statement]*;
}
Description: For a global regular expression block, the commands and statements inside the brackets are executed once per match. In other words, if the regular expression occurs three times in one line, this block is executed three times for that line.
Example:
% will run through the commands for every “at” in the input
/at/ global {
    set $hi, “hi”;
    set #boo, “33”;
    print $hi + #boo + @match;
}

Name: REGEXP
Type: LINE
Syntax: /<regexp>/ line {
    [command | statement]*;
}
Description: Line regular expression blocks will execute once per every line that contains the regular expression. So, unlike the global type, if the

regular expression occurs three times in the same line, this block is only executed once.

Example:

```
%* will run through the commands once no matter how many
times "at" occurs in the same line *%
        /at/ line {
                set $hi, "hi";
                set #boo, "33";
                print $hi + #boo + @match;
        }
```

6.2. Regular Expression

Since this program is based on searching for the regular expression within the input, it is critical that we define a regular expression. The definition used is the same as the one presented in Michael Sipser's book: *Introduction to the Theory of Computation*.

Say that R is a regular expression (<regexp>) if R is:

1. *a* for some *a* in the alphabet $\sum$,
2. ε,
3. $\emptyset$,
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, or
6. (R_1^*), where R_1 is a regular expression.

7. Statements

There are only three available statement types for use in gscc: selection, iteration, and return. These statements are similar to those in C, C++ and Java in both syntax and usage. Few statements are provided, but due to their robustness, they are able to accomplish the same functionality of statements in most other languages.

7.1. Selection Statements

Name: IF

Syntax:

```
if(<expression>) {
        [command | statement]*;
}
```

Description: This statement performs a Boolean check on the given <expression>. If it is evaluated to be true, then the commands and statements within the open and closed brackets are executed.

Example:

```
%* if #three contain the value 3, then the set command
will be executed otherwise, this block is skipped *%
```

```
% will be executed
set #three, 3;
if (#three = = 3) {
        set $three, “yes”;
}
```

```
% will not be executed
set #three, 4;
if (#three = = 3) {
        set $three, “yes”;
}
```

Name: ELSE IF
Syntax:
```
<IF syntax>
else if (<expression>) {
        [command | statement]*;
}
```
Description: This statement is executed if the Boolean check on the preceding IF block results in a false value. It will then perform another Boolean check on the <expression> for the else if statement. If this is evaluated to be true, then the commands and statements within the open and closed brackets are executed.
Example:

```
%* if #three contain the value 3, then the set command
will be executed otherwise, this block is skipped.
the else if will be executed if #three contains the value 4 *%
```

```
% IF will be executed
set #three, 3;
if (#three = = 3) {
        set $three, “yes”;
}
else if (#three = = 4) {
        set $three, “no”;
}
```

```
% ELSE IF will be executed
set #three, 4;
if (#three = = 3) {
        set $three, “yes”;
}
else if (#three = = 4) {
        set $three, “no”;
}
```

Name: ELSE
Syntax:
```
<IF syntax>
[ELSE IF syntax]*
else {
        [command | statement]*;
}
```
Description: This statement is executed if the Boolean check on all of the preceding IF and ELSE IF block statements results in a false value. No check is needed for this statement; instead, if executed, it will execute all the commands and statements within the open and closed brackets.
Example: (continues to next page)

```
%* if #three contain the value 3, then the set command
will be executed otherwise, this block is skipped.
```

the else if will be executed if #three contains the value 4.
if neither are true, then the else is executed *%

```
% IF executed
set #three, 3;
if (#three = = 3) {
        set $three, “yes”;
}
else if (#three = = 4) {
        set $three, “no”;
}
else {
        set $three, “ack”;
}
```

```
% ELSE IF executed
set #three, 4;
if (#three = = 3) {
        set $three, “yes”;
}
else if (#three = = 4) {
        set $three, “no”;
}
else {
        set $three, “ack”;
}
```

```
%ELSE executed
set #three, 5;
if (#three = = 3) {
        set $three, “yes”;
}
else if (#three = = 4) {
        set $three, “no”;
}
else {
        set $three, “ack”;
}
```

7.2. Iteration Statements

Name: WHILE

Syntax:
```
while(<expression>) {
        [command | statement]*;
}
```

Description: This statement will execute the commands and statements inside the open and closed brackets while the <expression> evaluates to true under a Boolean check. The expression is checked when the statement is first encountered, and then checked again each time the program reaches the closed bracket. This loop will continue to repeat until the <expression> evaluates to be false; it will then skip over the block and precede normally with the program.

Example:

%* this will loop and keep performing the set commands until #boo is greater than 3 *%

```
% final value of $boo is 4
set #boo, 1;
while (#boo <= 3) {
        set #boo, #boo + 1;
        set $boo, #boo;
}
```

```
% skips the while entirely
set #boo, 5;
while (#boo <= 3) {
        set #boo, #boo + 1;
        set $boo, #boo;
}
```

7.3. Return Statement

Name: RETURN

Syntax: return <expression>;

Description: The return statement is used in a function to return the appropriate value back to where it was called. If expecting a string($)

return, then the <expression> returned will automatically be converted to a string. If an integer(#) is expected, then <expression> should evaluate to an integer value or an error will be thrown. When a return is encountered in a function block, it will immediately break out of the function after the return is executed.

Example:

```
% at the end of the function, the value of $hello is returned
    func $helloworld (#world) {
        set $hello, #world;
        return $hello;
    }
```

8. System Commands

System commands are the only ways to perform an operation on an identifier in gscc. All system commands are predefined by the compiler and cannot be created by the programmer. There are only two types of system commands: those that perform operations on identifiers and those that print expressions (either to stdout or stderr). They all take this general form:

<system command> <comma separated list of identifiers and/or expressions>

System commands do not return any value unlike functions (See Section 9). Operational types will perform an operation on the first argument based to the command called and the following parameters given to it. Printing types, on the other hand, consist of simply the command and the expression that is to be printed.

8.1. Operational System Commands

%* for all of the following definitions:
<string> refers to either a string identifier($) or a location(@)
<integer> refers to either an integer identifier(#) or an actual integer *%

System Command: SET
Syntax: set <identifier> <expression>
Parameters: <identifier> - can be any identifier
<expression> - this can be any expression

Description: This command will replace whatever value that is originally in <identifier> with the value given by <expression>. For locations(@), their string value will be replaced by the value specified in <expression>. Integers(#) must have a corresponding integer valued <expression>, while strings($) will have the <expression> value automatically converted.

Examples:

```
% would replace the location's string value with "abc"
    set @match, "abc";

% would replace the string $myvar with "1234"
```

```
set $myvar, "1234";

%* would replace the location's string value by the expression
"russssur" + $myvar, which if using the above set, would give the
overall string value "russssur1234" *%
    set @match, "russssur" + $myvar;

%* would replace the location's string value by the expression
@line + $var which takes the string converted value of @line
(let's say this is "4") and concats it to $myvar giving "41234" *%
    set @match, @line + $myvar;

% would replace the integer #mynum with 123
    set #mynum, 123;

%* would replace the integer #sub with the expression
#mynum – 23 which results in 100 *%
    set #sub, #mynum – 23;
```

System Command: SETCHAR
Syntax: setchar <string> <integer> <character>
Parameters: <string> - the string where the character will be placed
<integer> - the location where the character will be set
<character> - the character that will be set at <integer>
Description: This will set the value found at index <integer> in <string> to whatever value is specified in <character>
Examples:

```
%* would replace the character at index two of $var to 'g'
resulting in "logk at me" as the value for $var *%
set $var, "look at me";
setchar $var, 2, 'g';

%* would replace the location's string's third index value by 'e'
which in this case results in "lalelala" for @match *%
set #third, 3;
set @match, "lalalala";
setchar @match, #third, 'e';
```

System Command: DELETE
Syntax: delete <string> <integer> <integer>
Parameters: <string> - the string from which to delete
<integer> - starting index
<integer> - number of characters to delete
Description: This command will delete characters starting at the index given by the first <integer>. The number of characters deleted will be equal to the number given by the second <integer>.

Examples:

```
% deletes five characters from index 3 within @match
        delete @match, 3, 5;


% deletes 2 characters from index 1 within $hello leaving "hlo"
        set #one, 1;
        set #two, 2;
        set $hello, "hello";
        delete $hello, #one, #two;
```

System Command: INSERT
Syntax: insert <string> <integer> <expression>
Parameters: <string> - string where <expression> will be inserted
<integer> - index where insertion will be done
<expression> - the expression that is to be inserted
Description: This command will insert the string value of <expression> into the string <string> at index <integer>.
Examples:

```
%* inserts this string at index 3, if @match was "hihihi"
after insertion, it would contain "hihabcihi" *%
insert @match, 3, "abc";

%* inserted the expression @match + "121" into $test
at index 0, giving "hihabcihi121testing123" *%
set $test, "testing123";
set #zero, 0;
insert $test, #zero, @match + "121";
```

System Command: REPLACE
Syntax: replace <string> <integer> <expression>
Parameters: <string> - the string where replace will be done
<integer> - starting index
<expression> - expression to replace with
Description: Starting at index <integer>, this will replace in <string> the string converted value of <expression> expanding as necessary.
Examples:

```
% replaces $hi starting at index 0 with "blah" leaving "blahlalalala"
        set $hi, "lalalalalala";
        set #zero, 0;
        replace $hi, #zero, "blah";

%* given that @match was "hellooooo", this will begin replacement
at index 5 and expand as necessary leaving "helloabcabcab331" *%
        replace @match, 5, "abcabcab331";
```

%* does replacement at 2 with expression $hi + “11” in @match
resulting in “heblahlalalala31” *%
replace @match, 2, $hi + “11”;

8.2. Printing System Commands

System Command: PRINT
Syntax: print <expression>
Parameters: <expression> - expression to be printed to stdout
Description: This will print whatever value is in <expression> to stdout.
Examples:

% prints value in $hi to stdout. This is your basic hello world
set $hi, “hello world”;
print $hi”;

%* given that @match is “I am Sam” this prints
“I am Sam, hello world” to stdout *%
print @match + “, “ + $hi;

System Command: PRERR
Syntax: prerr <expression>
Parameters: <expression> - expression to be printed to stderr
Description: This will print out the string value of <expression> to stderr.
Examples:

% would print “you messed up” to stderr
prerr "you messed up";
%* will print the expression “you messed up” + $big to
stderr, which is “you messed up big time” *%
set $big, “ big time”;
prerr “you messed up” + $big;

9. Functions

Functions are named blocks of statements that return one of two types: string or integer. All functions must be declared before any regular expression blocks in the program. Arguments are passed to the function by value. To write a function, the following convention is used:

func <identifier>(<idenitfier list>) { }

1. The func keyword must precede the entire declaration.
2. This is followed by an appropriate identifier as per the function’s return type. $<string> identifiers signify functions that return strings while #<string> identifiers signify integer return types.

2a. The special identifier type @<match|line> cannot be used since function types are restricted to only string and integer types.

2b. The list of identifiers used for the function names must be unique. Although both #one( ) and $one( ) can co-exist, $two( ) and $two(#two) cannot. In other words, function overloading is not allowed

3. After the identifier is a comma separated identifier list enclosed by an open and closed parenthesizes. The identifier list can have either many or no parameters.
4. Finally, after the parameter list is all of the statements associated with the function enclosed in open and closed brackets.

The execution of the function beings with the first statement after the opening bracket. It will proceed statement by statement until either a return is executed, or it reaches the closing bracket of the function. If no return call is encountered before the end of the function, then the NULL value is returned by default. It is important to note that for these "pseudo-void" type functions, the user must still decide if it is a string or integer type function.

Example:

```
func $example($one, #two, $three) {
        if (#two > 2) {
                set $a, $one + $three;
                return $a; }
        set $b, "this makes no sense";
        return $b;
}
```

$example is an example of a string function that will concat the two strings if the parameter #two is greater than 2 and then return that value. However, if #two is not greater than 2, then it will set $b to "this makes no sense" and return that string to where it was called.

Example:

```
func #isTwo($two, #two) {
   if ($two = = "two") {
           set #add, #two + 2;
           return #add; }
   set #sub, #two – 2;
   return #sub;
}
```

Meanwhile, #isTwo is an integer type function that will set #add as the value of #two plus 2 and return that value if $two is the string "two". Otherwise, it will set #sub as the value of #two minus 2 and return that resulting value.

Example:

```
func $expA( ) {                          func $expB( ) {
```

```
    set #a, @line;
    if (#a = = 1) {
       return “one”; }
    else {
       set $b, "booooo";
       return $b; }
}
```

```
    set #a, @line;
    if (#a = = 1) {
       return “one”; }
    set $b, "booooo";
    return $b;
}
```

In the last example, both are string functions perform the same procedure: both will return the string “one” if the value in @line turns out to be 1 and return “booooo” otherwise. But upon compilation, the first function is not guaranteed to call a return, but as stated before, a return is not necessary; thus, both functions are still valid.

10. Built in functions

There are two built in functions pre-defined in the interpreter for use by the programmer. Since both names are keywords and function overloading is prohibited, these two functions cannot be overwritten.

Function Name: $substr
Syntax: $substr($variable, #start, #length)
Parameters: $variable – represents the string from which the substring will be taken from
#start – the starting index
#length – the ending index
Description: The function will return the substring represented by that location from indexes represented by #start to #length inclusive. Since both # and @ identifiers are automatically converted to string types, you can retrieve the substring of strings, numbers, and locations.
Example:

```
set $test, $substr(“abc123”, 2, 4);
%* This will set the value in $test to “c12” *%
```

Function Name: #length
Syntax: #length($string)
Parameters: $string – the string whose length is to be determined
Description: Will return the length of the given string.
Example:

```
set $hello, “hello”;
print #length($hello);
%* This will print “hello” to stdout *%
```

11. Scope

The scope of any identifier can be broken down into either global or local scope. Scope for any identifier is defined at the smallest containing block for that identifier. To clearly explain the scope of an identifier, we will describe lexical, stack based, and nested scope.

1. Lexical Scope

   The main concern for lexical scope is the indexing of string($) identifiers. If the programmer attempts to reference any index value outside of the maximum and minimum indexes of the string($), a runtime error will be thrown.

2. Stack Based

   Within a given function, there is no access to any identifiers/variables other than globally defined identifiers/variables. If one is referenced, a null value will be returned.

3. Nested Scope

   Any identifiers that are ‘set’ outside of all function and regexp blocks are considered to be global identifiers/variables. Identifiers that are defined within a function or regexp block are considered local identifiers/variables to that block. Any identifiers that are defined in a loop block are unique to that loop block and die when the loop block terminates.