Goal Programming Language Documentation

Running a Goal Program

To run a Goal program you need to write and save your code in a .gol file then run the haskell script GoalCompiler.lhs. Then you simply need to use the “run” function giving the file location of your .gol file as an argument, as is shown below;

```
run "File_Location\\prog.gol"
```

Will run your Goal code.

File Layout

There is a very specific layout required when creating a new .gol file to create you code this is shown below;

```
GLOBAL VARIABLE ASSIGNMENTS

MAIN FUNCTION

FUNCTION DEFINITIONS
```

This is the order you must write your Goal programs in.

Syntax Rules

There are a few important syntax rules for Goal. The most important one is to make sure that each command you write in Goal is ended with a semi colon. This includes function declarations and if statements.

Overall Goal has a very simple syntax and is very easy to write code in, throughout this document are examples of Goals syntax.

Functions

In Goal you can have two types of functions. Functions that return a value and void functions. You must define functions after your main function. Function names can only contain letters and numbers. You can call any function as a void function it will just have no effect here are some example function declarations.

```
func square(n int) int {
      return n*n;
};

func printHello(j int) {
      for( j > 0){
            j--;
            Print("hello");
      };
};
```

It is also important to notice that like every command in Goal you are required to end each function declaration with a semi-colon.

Types

In Goal there are only 2 usable types booleans and integers. Even with this booleans are treated as integers in the virtual machine. You can also use strings but only in the Print(); command to output strings where needed.

Variables

Variables are used to hold values. Each variable has a scope, which once declared cannot be changed. Goal only allows a limited number of types, because of this declarations are not very strict and variables can be created on the fly. There is scope for this to change if new numerical types are added.

`j = 42;`
This is how you declare a new local variable. It is also how you assign and reassign a variable for instance if you were to write;

```
j = 42;
i = 3;
j = 10 + i;
Show(j);
```

Your output would be 13, as j has been declared as 42 then reassigned as 13 .

`global j = 42;`
This is how you declare a new global variable. This must be done before your main function. Global variables can be read and written too from anywhere though it is advised to use channels instead of global variables to communicate between concurrent processes.

`j += 2;`
Only for numerical types, will increment the variable by a set value (in this case 2).

`j++;`
Only for numerical types, will increment the variable by one.

`j -= 2;`
Only for numerical types, will deincrement the variable by a set value (in this case 2).

`j--;`
Only for numerical types, will deincrement the variable by one.

Arithmetic Expressions

These expressions are used for performing arithmetic operations between variables with the same numerical type.

`a + b`
This gives you the sum of a and b.

`a - b`
This returns the value of a minus b.

`a * b`
This returns the values of a multiplied by b.

`a / b`
This returns the value of a divided by b.

There is also an operator precedence in place, with division being done first and addition being done last for example the following expression;

`a + b - c *d / e` would be computed as; `a + (b - (c * (d / e)))`

You can use brackets to force a certain operation to precede another eg;

```
((a + b) - (c * d)) / e
```

Is acceptable Goal code.

Comparative Expressions

These are expressions that make a comparison between two values and return true or false.

a == b

True if a is equal to b, can be used with any type, although nested comparisons are not allowed for example ;

```
a == b == c or (a == b) == c
```

Is not valid Goal code.

a != b

True if a and b are not equal, can be used with any types. Again, as above, nested comparisons are not allowed

a < b

True if a is smaller than b, only for numerical types

a > b

True if a is greater than b, only for numerical types

a <= b

True if a is smaller or equal to b, only for numerical types

a >= b

True if a is greater or equal to b, only for numerical types

a && b

True if both a and b evaluate to true expressions

a || b

True if one or both of a and b, evaluate to both be true expressions.

If Statements

```
if ( a < b ) {
     return 5;
};
```

This is how you use if statements in Goal. Your comparative expression must be represented with in single brackets. You can also have single variables (either integers or booleans) or an arithmetic expression within the brackets where any positive expression is treated as true and

any value equal to or smaller than 0 is treated as false. eg the following code would output success;

```
if (4 - 6) {
      Print("fail");
};

if (5) {
      Print("success");
};
```

Also note how it is still necessary to end each if statement with a semicolon. This can be useful when dealing with if else statements and defining their scope.

```
if (a == b) {
      return 42;
} else {
      return 13;
};
```

This is how you use if else statements in goal.

```
if (a == b) {
      return 12;
} else if ( a < b ) {
      return 16;
} else if ( a > b ) {
      return 11;
} else {
      return 17;
};
```

This is how you use else if statements in Goal. You may have as many else if statements between your initial if statement and your final else statement but the first true condition it hits will be executed then exit the statement regardless of how many true conditions exist. For example the following code will output only “success”:

```
if (1 == 0){
            Print("fail");
} else if (1 == 1) {
            Print("success");
} else if (2 == 2) {
            Print("fail");
} else {
            Print("fail");
};
```

For Loops

For loops are dealt with the same way Go deals with for loops. There are two ways to use for loops. The traditional for loop with an incrementing or decrementing value and a condition, or you can use them like traditional while loops.

```
for(i = 0; i < 10; i++){
      CODE
};
```

This is an example of how you generate a traditional for loop in Goal.

```
for (j > 10) {
      CODE
};
```

This is an example of you create typical while loops in Goal, making use of the for keyword.

Subroutines

Subroutines are independent concurrent process that will start with the use of the “go” keyword. This is dealt with the same way Go deals with subroutines.

```
go func();
```

Will run the function func() as a independent concurrent process. Any function run this way may have arguments but cannot return a value.

```
Wait();
```

Calling this function will cause the program to wait here until all subroutines have finished executing. Will wait indefinitely if processes are in infinite loops.

```
WaitOn(PROCESS);
```

Calling this function you must give the name of a running concurrent process. Your code will will wait at this command until all instances of this process have stopped running

```
Kill();
```

This will immediately stop all currently running subroutines.

Channels

Channels are a way that sub routines running at the same time can communicate with each other and pass variables around. The behave like a simple stack where you push things onto them and then pop the off, with the most recently pushed value popped out first. The way channels are handled is almost exactly the same as how they are handled and used in Go.

```
var c chan = Make(chan int);
```
This is how you create a new channel. The arguments **chan int** state that the channel being created will only be able to handle integers.

```
c <- 10;
```
This is how you push something onto a channel.

```
i = <-c;
```
This is how you would pop something of a channel and assign it to a variable

```
WaitChan(CHANNEL);
```
You can use this command to avoid deadlock. Attempting to pop an empty channel will cause an error.Using the WaitChan() your code will wait on a certain channel until it is no longer empty.

Outputting

```
Show(n int);
```
This function will output an integer to the console on a new line.

```
Print(s string);
```
Calling this function will output a string to the console on a new line.