**Brief description:**

This is a Hoc-like compiler based on the Hoc compiler, as seen in ‘The Unix Programming Environment’ by Kerninghan and Pike, written in C# using object-oriented programming.

**Manual:**

The specification of this compiler is similar to that of the Hoc compiler. The language is mostly expression based, with control-flow statements.

**This compiler supports:**

**Arithmetic and unary operators:**

* Addition +
* Subtraction -
* Multiplication \*
* Division /
* Logical and arithemetic negation !

**Logical and comparison operators:**

* Greater than >
* Greater or equal >=
* Less than <
* Less or equal <=
* Equal ==
* Not equal !=
* AND &&
* OR ||
* Assignment =

**Built in constants**

* DEG: 57.29577951308232087680
* E (Euler’s number): 2.71828182845904523536
* GAMMA: 0.57721566490153286060
* PHI: 1.61803398874989484820
* PI: 3.14159265358979323856

**Built in functions**

* Sin(x)
* Cos(x)
* Tan(x)
* Atan(x)
* Log(x)
* Log10(x)
* Exp(x)
* Sqrt(x)
* Abs(x)

**Constants:**

All numbers, including builtin constants, are decimals. For constants, input format is as follows: a signed or unsigned number must be present. This could be optionally followed by another number, a decimal point (.) and number, or signed/unsigned exponent. Since this hoc-like compiler also functions as a calculator, numerical constants and arithmetic operations are printed by default.

String constants are defined by enclosing characters or numbers in a double quotation mark.

**Identifiers:**

Identifiers can be user defined and must start with a letter, that can be optionally followed by a string of letters, numbers and/or underscore (\_).

Names of variables, functions and procedures are identifiers (please refer to below).

**Statement and Control Flow:**

If and while are parsed as statements. If is declared using the keyword ‘if’, followed by one or more comparison operators enclosed in brackets. The keyword ‘if’ must be on the same line as the comparison operators, else it will cause a syntax error. The body of the if statement can be a single statement with optional braces or a list of statement between braces. Declaration of while statement uses the same principles, with keyword ‘while’ instead of ‘if’.

After declaration of if statement, an optionally else statement can be declared. Likewise, the body of else statement can also be a single statement with optional braces or a list of statement between braces.

Assignment is parsed as a statement.

**Input and Output:**

Input is done via the read is a statement instead of a builtin function unlike in hoc. Read can only take a single variable. Unlike in hoc-compiler, in hoc-like compiler, the variable will have to be predefined then passed into the read function, after which a new value to the variable could be assigned during execution. This could be a decimal number, or if the input is enclosed in double quotation marks, it will be assigned as a string.

Print is a builtin function. Print can take a single expression or a list of expressions, separated by comma. Statements cannot be printed. Print must have at least 1 argument.

X = 0

read(x)

print x

^E

10

**Fig 1: Example usage of read and print**

**Functions and procedures:**

Functions are defined with the keyword ‘func’, followed by an identifier and enclosing brackets (e.g. func fib()). Procedures are defined in the same fashion, using the keyword ‘proc’ instead of ‘func (e.g. proc fib())’. During input, ‘func’ or ‘proc’, the identifier and brackets must be on the same line. A pair of braces is then required to define the body of the function or procedure. The body could be a single statement, or a list of statement. If the braces are left empty, it is a null function.

Both functions and procedure can be called using their identifier with enclosing brackets (e.g. fib()) and can be done so recursively.

Although functions and procedures utilises the same mechanisms during definition and calling, they are distinct. Function calls are parsed as expressions, and can return values whilst procedure calls are parsed as statements and cannot return values.

Single or multiple arguments can be passed into functions or procedures using ‘$’ and digits to denote the order of parameters. A comma is required to separate multiple arguments. For example, $1 refers to the first argument passed in and $2 is the second. $1, $2 denotes 2 parameters.

func fac()

{

if ($1 <= 0) return 1 else return $1 \* fac($1-1)

}

print fac(10)

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

func ack()

{

if ($1 == 0) return $2 + 1

if ($2 == 0) return ack($1-1, 1)

return ack($1-1, ack($1, $2-1))

}

print ack(3, 3)

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

proc PrintNumbers()

{

if ($1 == 0) return

PrintNumbers($1 - 1)

print $1

}

PrintNumbers(3)

**Fig 2: Examples of functions and procedures**

**Other functions:**

Verbose (Ctrl + X): Enables translated stack machine intermediate language (SMIC) to be printed.

Execute (Ctrl + E): Used to execute the code after console input.

Exit (Ctrl + Z): Exits the programme.

**Further Examples:**

func fac()

{

if ($1 <= 0) return 1 else return $1 \* fac($1-1)

}

func stirl()

{

return sqrt(2\*$1\*PI) \* ($1/E)^$1\*(1+1/(12\*$1))

}

i = 0

print "Enter a number: "

read(i)

i = i + 1

while (i <= 20)

{

print i, " ", fac(i)/stirl(i), “\n”

i = i + 1

}

**Fig 3a: Ratio of factorial to Stirling’s function**