The C-Sheep Programming Language

Introduction

This text is a reference for the C-Sheep programming language. C-Sheep is a member of the family of mini-languages[1] (*toy languages*) used for teaching programming and computer science principles. The command syntax of the C-Sheep programming language is a subset of the ANSI C programming language[2]. In its current version, C-Sheep only implements a basic set of control structures (*simple iteration, condition/alternative and sequence*) as well as the definition of sub-routines (*functions*).

The C programming language is an all purpose programming language which can be used to address any kind of programming problem. The core language itself has a very small set of instructions and commands, but it comes with a useful set of functions which are stored in so called libraries. These standard libraries contain functions for system commands, input/output and memory management.

Apart from being a tool for learning the basics of the C programming language, C-Sheep implements the C control structures that are required for teaching the basic computer science principles encountered in structured programming.

basic structure of a C(-*Sheep*) program

```
#include....           /* inclusion of libraries into the program code */
#define...            /* definition of symbolic constants             */

/*    declaration and initialisation (optional) of global variables */
<datatype> <variable1> [= <value1>[,...,<variablen> = <valuen>]];

/*     function prototyping (function forward declarations) */
<return datatype> <function name>(<parameter list>);[3]

/*    definition of the main function      */
<return datatype> main(<parameter list>)
{
  <declaration of local variables>
  <commands/instructions of the main function>
}

/*     definition of functions */[4]
<return datatype> <function name>(<parameter list>)
{
  <declaration of local variables>
  <commands/instructions of the function>
}
```

The entry point of a C-Sheep program from which it will begin program execution is always the “main” function. This function must be defined in every C-Sheep program.

---

[1] Brusilovsky, P., Calabrese, E., Hvorecky, J., Kouchnirenko, A., and Miller, P. (1997). Mini-languages: A Way to Learn Programming Principles. Education and Information Technologies 2 (1), pp. 65-83

[2] Kernighan, B.W. and Ritchie, D.M. (1988). The C Programming Language, Prentice-Hall

[3] in prototypes only the return datatype of the function (*if different from ”int*”) and the datatypes of parameters need to be stated - identifiers for the function parameters only have to be used in function definitions

```
/*     example function prototypes    */
int myfunction(int,int);
void myotherfunction(float f);
```

please note: while the current version of the C-Sheep compiler will recognize function prototypes it does not yet correctly generate code for the forward declaration of functions.

[4] the declaration and definition of a function can be carried out in a single step (as with the `main` function). however in that case the definition has to be made before the function is used for the first time

comments in C-Sheep programs

Comments can be considered a basic component of modern programming languages. They are used to make source code easier to read and easier to understand by allowing the programmer to annotate his work within the program source code itself. In C-Sheep comments are marked by two symbol sequences "**/*** " to open a comment and "***/**" to close a comment. The text encapsulated between these two symbol sequences is ignored by the compiler, i.e. during compilation the compiler will not evaluate the comments in the source code. It is important that every opening symbol of a comment has a matching closing symbol, as otherwise the compiler would be unable to determine what parts of the source code have to be compiled. C-Sheep comments can span across more than a single line of source code, i.e. if a comment is opened in one line of code and closed in another line of code, all text in between will be commented out.

```
syntax:     /* <comment text> */
examples:   /* a simple C-Sheep comment */
            /* a C-Sheep comment, spanning
               across three lines of
               source code         */
```

symbolic constants in C-Sheep

```
syntax:     #define <name>   <value>
examples:   #define  MAX       32000
```

By convention constants in the C-Sheep programming language are declared using capital letters. Constants and Macros defined with the **#define** preprocessor directive (*instruction*) are *typeless* command definitions. They are defined at the top of the program source code. While in the C programming language, the compiler's preprocessor only performs a simple text substitution of the constant value (allowing every possible character sequence to be used as the constant value), C-Sheep constants must be numeric values.

variables in C-Sheep

Every programming language needs variables, as otherwise it would be impossible to implement any kind of input or output functionality. In the C-Sheep programming language the memory for a program's variables has to be reserved at the start of the program. This is called variable declaration. A variable is declared by stating the variable datatype followed by the variable identifier (*name of the variable*). This identifier must not start with a digit and should not contain any special characters (*like '®' or '@'*) or characters from localized character sets (*like 'ä' or 'å'*) as they will not be recognized by the compiler.

```
syntax:     <variable_type> <variable_identifier>;
            <variable_type2> <variable_identifier2>,<variable_identifier3>;
```

Once a variable has been declared it can be used, i.e. data can be filled into the variable. For a newly declared variable this assignment of an initial value to the variable is called the variable definition.

```
syntax:     <variable_identifier> = <value>;
            <variable_identifier2> = <variable_identifier3>;
```

The contents of variables can also be defined (*initialised*) during variable declaration.

```
syntax:     <variable_type> <variable_identifier> = value;
            <variable_type2> <variable_identifier2> = <value2>,<variable_identifier3> = <value3>;
```

The C-Sheep programming language implements a number of predefined variable datatypes which are suitable for most common programming situations.

ordinal variable datatypes

Currently the only ordinal (*integer*) datatype in C-Sharp is the signed integer (*allowing negative as well as positive values*):

`int` This datatype holds integer numbers whose size (*memory requirements*) depends on the platform that the program is compiled on. **`int`** is probably the most commonly used datatype in C programs.

example: **`int`** integer = 12345**`;`**

floating point variable datatypes

Floating point numbers allow the use of real numbers in programs. The only floating point datatypes in C-Sheep is:

`float` This datatype holds single precision floating point numbers.

example: **`float`** f = 12.34**`;`**

control structures

Program flow in a computer program is linear and all statements are executed one after the other. To allow branching between different alternatives or repetition of statements, programming languages contain control structures that evaluate logical expressions and, depending on the outcome of these evaluations, execute different parts of the program code. These control structures are the (*unconditional*) sequence, conditional statements and loops[5]. The main control structure in the C-Sheep programming language, C-Sheep being a procedural programming language, is the function, which itself uses blocks to group commands/instructions.

statements

The simplest control structure is a single statement. This can be a declaration, function, expression or another control structure.

blocks

Blocks are used to group a sequence of statements (*functions, expressions or other control structures*). In C-Sheep blocks are encapsulated by braces: "**{**" to open a block and "**}**" to close a block.

syntax:
```
{
        <statement1>;
        <...>
        <statementn>;
}
```

conditional statements

The ***if statement*** is used to determine whether or not a particular function, expression or control structure is to be executed. To decide between alternatives, an ***else clause*** can be used in combination with the if.

syntax:
```
if(<expression>)
        <block or statement>
if(<expression>)
        <block or statement>
else
        <block or statement>
```

examples:
```
if(blocked(FRONT)==0)
        step();
```

[5] Böhm, C. and Jacopini, G. (1966). Flow diagrams, Turing Machines and Languages with only Two Formation Rules. Communications of the ACM 9(5), pp. 366-371

```
if(blocked(FRONT)==1)
        step();
else
        turn_left();
```

Within any expression in an if statement, a value of 0 (*zero*) will be interpreted as FALSE (*condition is not satisfied*), any other value will be interpreted as TRUE (*condition satisfied*).

loops

In the C-Sheep programming language two different types of loops (*iterations*) are available. Within any expression in a loop, a value of 0 (*zero*) will be interpreted as FALSE, any other value will be interpreted as TRUE.

The most simple to use loop its the head controlled ***while loop***.

syntax:
```
while(<expression>)
        <block or statement>
```

example:
```
while(blocked(FRONT)==0)
        step();
```

There is also the foot controlled ***do-while loop***.

syntax:
```
do
        <block or statement>
while(<expression>);
```

example:
```
do
        i++;
while( i<10 );
```

functions

Just as one can use predefined functions from the standard libraries that are provided with the C-Sheep programming language (*stdlib and sheep*), one can program one's own functions for use within C-Sheep programs. A function has a name, its identifier, a return datatype and (*optionally*) a list of parameters. As with all identifiers, the function identifier must not start with a digit.

Before a function is defined it has to be declared, i.e. the compiler has to be informed about return values and parameters of a function so it can detect possible errors or inconsistencies in the application of the function. These function declarations are called function prototypes.

syntax: <return datatype> <function name> **(**<datatype1>,⋯,<datatype***n***>**);**

example: `int` max**(**`int,int`**);**

It should be noted that if a function is declared without explicitly stating its return datatype, a return type of “int” will be assumed.

The scope of a function is anywhere below the declaration of the function within the C-Sheep source file in which the function has been declared. It is *impossible* to nest function definitions, i.e. functions must be defined globally.

When a function is defined, all the datatypes that have been given in the parameter list of the prototype have to be provided with an identifier.

syntax:
```
<return datatype> <function name> (<datatype1><argument1>,
                                        ⋯, <datatypen><argumentn>)
        <block or statement>
```

example:
```
int max(int value1,int value2)
{
        if( value1 > value2)
                return value1;
        else
                return value2;
}
```

The scope of local variables that are declared inside of a function is restricted to that function, i.e. they are known to the compiler exclusively within the function. Outside the function they are invalid. Local variables that have not been declared within the function's parameter list must be declared right after the opening braces of the function body. Functions can be jumped out of and values can be returned from within a function to the next higher level using the ***return statement***.

syntax:
```
return;
return <value or statement>;
```

Returned values or variables have to be of the same type as the return datatype of the function. Variables that receive a return value of a function have to be of the same datatype as the function. A return statement is not necessary in functions that do not return a value (*void functions*). These functions, which require the typeless datatype **`void`** as return datatype are called procedures.

operators

Like most programming languages the programming language C-Sheep has a wide range of operators. The most commonly used operators are for mathematical and logical operations.

arithmetical operators

The operators for arithmetical operations in C-Sheep are:

+ Operator for additions.
- Operator for subtractions.
***** Operator for multiplications.
/ Operator for divisions.
% Modulo operator.

logical operators

The logical operators in C-Sheep are:

&& Logical AND operator.
|| Logical OR operator.
! Logical NOT operator.

relational operators

The symbols for relational (*comparative*) operations in C-Sheep are:

== The '*is equal*' operator tests for equality.
!= The '*not equal*' operator tests for inequality.
< The '*less than*' operator tests if the expression left of the operator is smaller than the expression on the right of the operator.
<= The '*less than or equal*' operator tests if the expression left of the operator is smaller than or equal to the expression on the right of the operator.
> The '*greater than*' operator tests if the expression left of the operator has a bigger result than the expression on the right of the operator.
>= The '*greater than or equal*' operator tests if the expression left of the operator is greater than or equal to the expression on the right of the operator.

assignment operator

standard assignment operator

c=c+2; The expression left of the assignment operator is evaluated first, then the result is written into the variable on the right of the operator.

C-Sheep Standard Library Functions

The C-Sheep standard libraries provide a number of general purpose functions, maths functions and sheep-specific functions (*functions for controlling sheep in the virtual environment*).

general purpose functions

General purpose functions and associated symbolic constants are declared in the *stdlib.h* standard header file:

`void abort(void);`
This function terminates a running C-Sheep program (*abnormal program termination*).

`void exit(int v);`
This function terminates a running C-Sheep program (*normal program termination*). The parameter ***v*** contains the exit status of the program (*this should be 0 for a sucessful program run or any other value for program termination caused by an error*) . The pre-defined symbolic constants **EXIT_SUCCESS** (*value 0*) or **EXIT_FAILURE** (*value 1*) should be used.

`int rand(void);`
This function returns a random integer number between 0 and the value contained in the symbolic constant **RAND_MAX**.

maths functions

Maths functions and associated symbolic constants are declared in the *math.h* standard header file.

The current Version of C-Sheep does not yet have any maths functions implemented. A single symbolic constant **PI** containing the constant value π is defined.

sheep functions

The *sheep.h* standard header file declares functions for controlling sheep (*by exposing sensor information and instructions for the sheep actor*) in the virtual world:

`void pause(void);`
This function will pause a running C-Sheep program (*user-interaction will resume program execution*).

`void initialise(int x,int y);`
This function initialises the position of the sheep in the virtual world at the positions given by the parameters **x** and **y**. This function should only be called once (*and at the start of the program*).

`void step(void);`
This function causes the sheep to take a single step forward in the direction it is facing.

`void backstep(void);`
This function causes the sheep to take a single step backwards (*in the direction opposite to the one it is facing*).

`void turn_left(void);`
This function causes the sheep to turn left on the spot.

`void turn_right(void);`
This function causes the sheep to turn right on the spot.

`int blocked(int direction);`
This function returns the value **1** if the sheep's path is blocked in the direction that is passed in as a parameter or the value **0** if the sheep's path is free. The direction parameter (*relative to the direction the sheep is currently facing*) should be one of the symbolic constants **FRONT** (*value 1*), **RIGHT** (*value 2*) **BACK** (*value 3*) or **LEFT** (*value 4*).

`int found(int object);`

This function returns the number of objects of the type that is passed in as a parameter found on the current grid square in the virtual world or the value **0** if no objects of that type are found.
The object types supported by the current version of C-Sheep are:

- **BALL** (*value 1*) - a simple ball

`float query(int subject);`

This function returns a value related to the subject that is passed in as a parameter. The subject parameter can be one of the following symbolic constants:

- **WEATHER** (*value 1*) - returns one of **PERFECT** (*value 4.0*), **GOOD** (*value 3.0*), **CHANGING** (*value 2.0*), **BAD** (*value 1.0*) and **HORRIBLE** (*value 0.0*)
- **TIME** (*value 2*) - the current time of the day in the virtual world as a floating point value between 0 and 23.9. The symbolic constants **MIDNIGHT** (*value 0.0*) and **NOON** (*value 12.0*) can be used.

Appendix A – C-Sheep Overview

reserved (*key*-) words

The C-Sheep programming language contains a relatively small number of reserved words, which can only be used in their pre-defined context and which may not be used as identifiers. In the curent version of C-Sheep, these keywords are:

```
do      else    float   if      int   return    while     void
```

C-Sheep standard (*built-in*) datatypes

`int`	32 bit integer	-2147483648 to 2147483647
`float`	32 bit floating point number (24 bit base, 8 bit exponent)	

Variables can be declared without initialisation or with initialisation.

syntax: <datatype> <identifier1>**,** ..... **,** <identifier***n***>**;**
<datatype> <identifier1> = <value1>**,** ..... **,** <identifier***n***> = <value***n***>**;**
example: int x = 1, y = 2, a, b;

Once declared a variable can be used as an operand for an expression or as a parameter for a function.

syntax: `<variable identifier>` **<**operator**>** `<identifier or constant>;`
`<identifier or constant>` **<**operator**>** `<variable identifier>;`
example: x = 1;
x = y;

operators and operator precedence

operator	operation	example	priority[6]	grouping[7]
()	parentheses (*for operation grouping*)	a=(b+c)*d;	1	L -> R
- **+** **!**	minus (*unary*) plus (*unary*) negation (*logical*)	`if(a==-1) return 0;` `int u=+10;` `if(a) return !a;`	2	R -> L
***** **/** **%**	multiplication division modulus	`a=b*c;` `a=b/c;` `a=b%c;`	3	L -> R
+ **-**	addition subtraction	`a=b+c;` `a=b-c;`	4	L -> R
< **<=** **>** **>=**	less than (*relation*) less than or equal (*relation*) more than (*relation*) more than or equal (*relation*)	`if(a<b)  a++;` `if(a<=b) a++;` `if(a>b)  b++;` `if(a>=b) b++;`	5	L -> R
== **!=**	equal (*relation*) not equal (*relation*)	`if(a==b) b++;` `if(a!=b) a--;`	6	L -> R
&&	AND (*logical*)	`c=a&&b;`	7	L -> R
||	OR (*logical*)	`c=a||b;`	8	L -> R

[6] The priority (lowest number == highest priority) of each operator shows its precedence in compound operations, i.e. operations in which more than one operator is involved.

[7] The grouping specifies the sequence in which the operands of an operation are interpreted, i.e. "left to right" (L -> R) or "right to left" (R -> L).