ecc - A C compiler By Elliot Carter

User Manual

Contents

What is ecc?

‘ecc’ is a C compiler which currently compiles a small subset of the C language into mips assembly.

What do you need to run ecc?

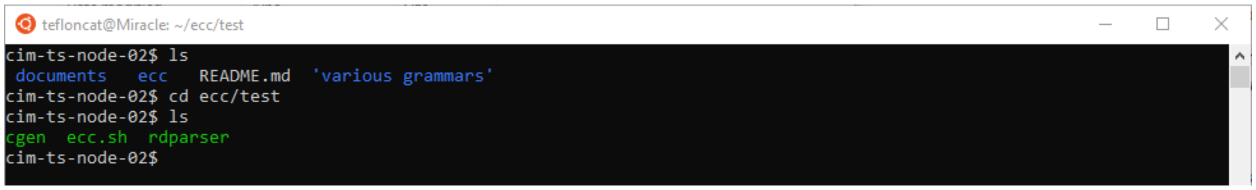
The compiler itself requires no dependencies in order to run. You can simply write your C program and use the provided shell script to run the compiler. It is however necessary to use SPIM in order to run the compiled program.

Using ecc

ecc is used via the linux command line similar to how conventional C compilers such as gcc and clang work.

From the root folder of ecc, navigate to the ‘test’ folder via the following:

```
cd ecc/test
```



teflоncat@Miracle: ~/ecc/test

```
cim-ts-node-02$ ls
 documents   ecc   README.md  'various grammars'
cim-ts-node-02$ cd ecc/test
cim-ts-node-02$ ls
cgen  ecc.sh  rdparser
cim-ts-node-02$
```

Once inside the test folder, you will find two programs and a shell script.

<u>`rdparser`</u>

`A parser which converts a C source file into a form cgen can recognise.`

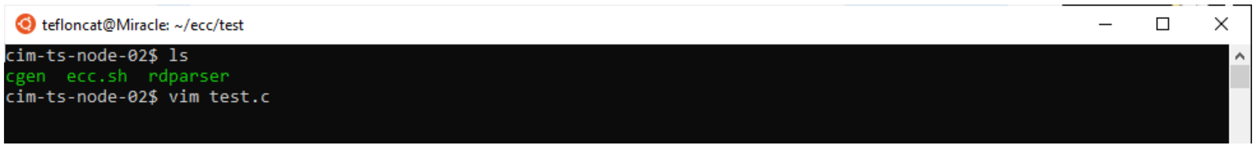
<u>`cgen`</u>

`A code generator which converts the output of the parser into mips assembly`

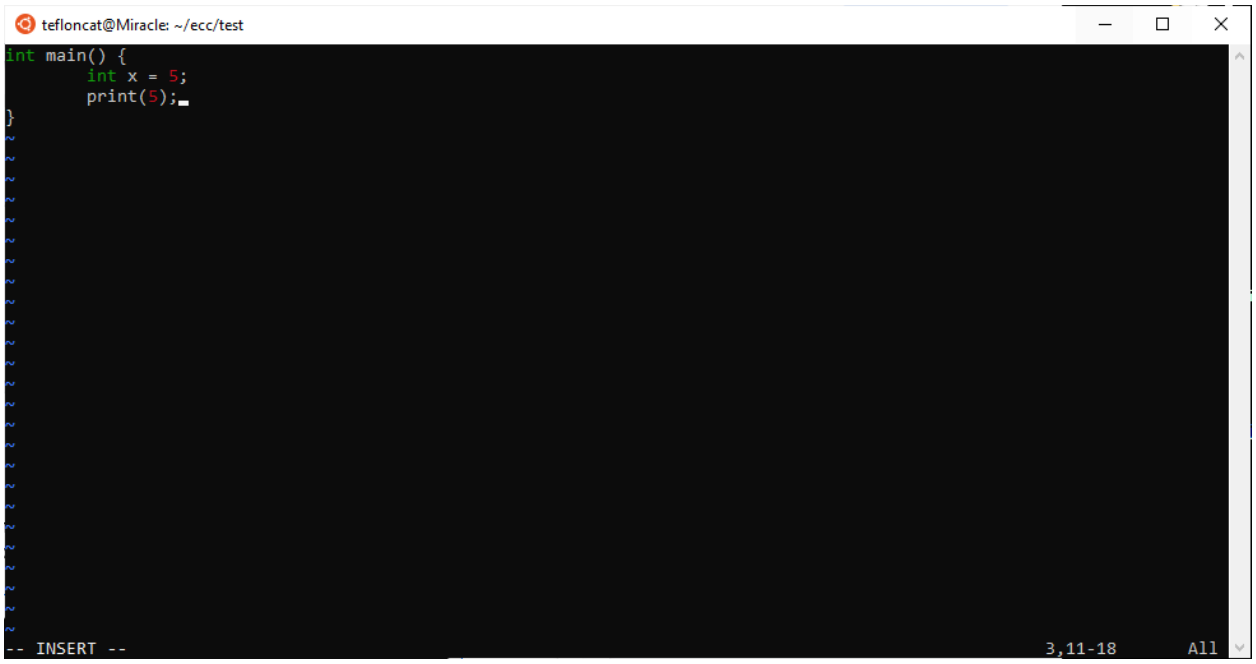
<u>`ecc.sh`</u>

`A shell script which runs the rdparser followed by cgen automatically to improve convenience for the user.`

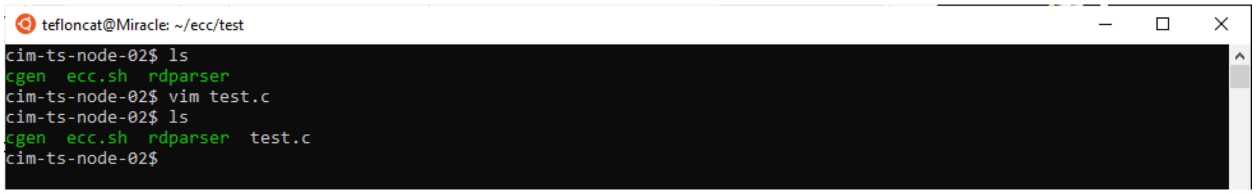
In order to begin using ecc, you will first need to write a c program. Feel free to use any text editor of your choosing to write your program.



```
cim-ts-node-02$ ls
cgen  ecc.sh  rdparser
cim-ts-node-02$ vim test.c
```



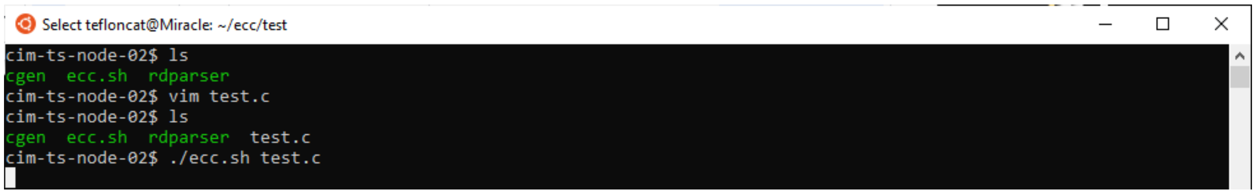
```
int main() {
        int x = 5;
        print(5);
}
-- INSERT --                                                  3,11-18       All
```



```
cim-ts-node-02$ ls
cgen  ecc.sh  rdparser
cim-ts-node-02$ vim test.c
cim-ts-node-02$ ls
cgen  ecc.sh  rdparser  test.c
cim-ts-node-02$
```

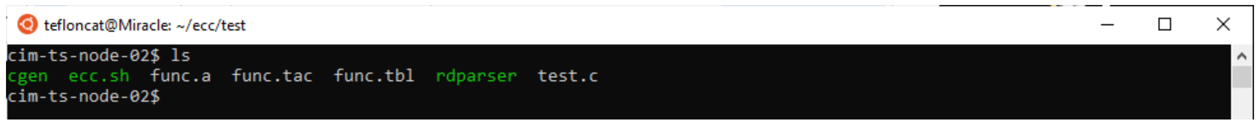
Once you have written and saved the program, you can proceed to use ‘ecc.sh’ to compile it. To do this, simply execute the script with the name of your source file as the only argument.

```
./ecc.sh ‘your source file’
```



```
cim-ts-node-02$ ls
cgen  ecc.sh  rdparser
cim-ts-node-02$ vim test.c
cim-ts-node-02$ ls
cgen  ecc.sh  rdparser  test.c
cim-ts-node-02$ ./ecc.sh test.c
```

Once you have done this, the folder will contain three new files.


teflonCat@Miracle: ~/ecc/test

```
cim-ts-node-02$ ls
cgen  ecc.sh  func.a  func.tac  func.tbl  rdparser  test.c
cim-ts-node-02$
```

These files are as follows:

`func.tac`

`Contains your program represented as “Three address code”, and intermediate code form which cgen uses to generate the final assembly program.`

`func.tbl`

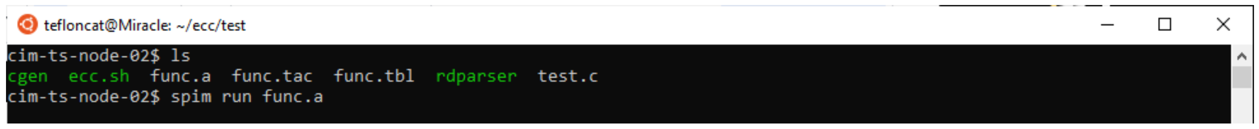
`Contains information from the symbol table which cgen also requires to build the final assembly program. This will contain the names of global variables and functions from your program, alongside some additional information, such as the number of local variables in each function and the values each global variable was initialized with.`

`func.a`

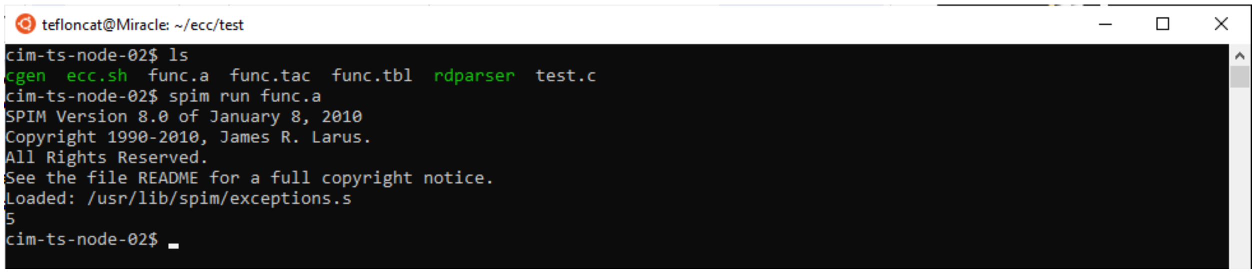
`This contains your compiled program. The contents of this folden is a program written in mips assembly which can be executed to perform the tasks of your original C program.`

As mentioned above `‘func.a’` is your compiled program. In order to run your program you will need to use SPIM. SPIM is a self-contained simulator that runs MIPS32 programs. Once you have spim installed simply do the following:

```
spim run func.a
```


teflonCat@Miracle: ~/ecc/test

```
cim-ts-node-02$ ls
cgen  ecc.sh  func.a  func.tac  func.tbl  rdparser  test.c
cim-ts-node-02$ spim run func.a
```



```
cim-ts-node-02$ ls
cgen  ecc.sh  func.a  func.tac  func.tbl  rdparser  test.c
cim-ts-node-02$ spim run func.a
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
5
cim-ts-node-02$
```

Spim will then proceed to execute your program. Above you can see the expected print output of ‘5’ from the test program used in this demonstration.

Language Features

ecc Supports the following features:

Variables

ecc currently only support integers. Variables can be declared in the following way:

```
int x; // no initializer
int y = 5; // initialized
```

ecc does **not** support declaration lists.

```
int x, y, z; // not supported
```

Variables can be declared in both the local and global scope. ecc supports shadowing, so variables declared with the same name across multiple scopes is also valid.

```
int x; // global x
int main() {
    int x; // local x
}
```

Assignment & Expressions

ecc supports the following numerical operations:

```
Basic Arithmetic: +, - , *, /, %, =
Boolean Arithmetic :  ==, !=, >, <, >=, <=
```

Expressions can be used to assign values to variables. This can be performed both when the variable is declared and later as an assignment operation. Existing variables can also be used in expressions.

```
int x;
int y = 3 + 5 * 2;
x = y / 4;
```

Expressions follow the conventional order of operations. Bracketed expressions are also supported.

```
x = 2 + 3 * 2; // result: 8
x = (2 + 3) * 2; // result: 10
x = 2 + 2 == 1 + 4; // result: 0
x = 2 + 2 == 1 + 3; // result: 1
```

ecc does **not** support negation of values:

```
x = -1; // not supported
```

Arrays

Arrays can be declared in the following manner:

```
int ar[5]; // explicit size allocation
int ar[] = {4, 5, 6}; // implicit size allocation using an initializer list
int ar[5] = {1, 2}; // explicit size allocation alongside an initializer list
```

Arrays can be sized and initialized using expressions.

```
int ar[1 + 2] = {5 * 2, x + y, 3 < 4};
```

ecc does **not** support variable length arrays. This means arrays can only be sized using integer literals.

```
int x = 5;
int ar[x]; // not supported
```

Arrays follow the same scoping rules as stated for variables, and can be declared in both the global and local scope.

Flow Control

ecc supports the following flow control statements:

```
if
elif (replacement for ‘else if’)
else
while
```

Switch case statements and for loops are **not** supported.

Any expression can be used as the condition for a flow control statement. ecc follows standard C semantics, where 0 is false, and any non zero value is true.

```
if(y == x)
while(1)
while(x)
elif(x + 2 * z)
```

All flow control statements must be followed by a compound statement

```
if (x) {
   // do something
} elif (y) {
   // do something else
} else {
   //do whatever there is left to do
}

while(1) {
    // do this forever
}
```

This means inline if statements and while loops are **not** supported.

```
while(x) x = x - 1; // no supported

// all invalid
if(x) x = x + 1;
elif (y) y = y + 1;
else x = x + y;
```

Ecc also supports the use of the **break** and **continue** statements inside of while loops.

Functions

Functions can be declared in the global scope of any C program.

```
func(int x) {
    return x * 2;
}

int main() {
}
```

Functions can be declared with no parameters or multiple.

```
func()

func(int x, int y, int z)
```

Functions can be called from within the scope of other functions.

```
int fn(itn x) {
    return x + 1;
}

int fx(int x) {
    return fn(x);
}

int main() {
    if (1) {
        fx(5);
    }
}
```

Functions can also be called recursively.

```
int fib(int x) {
      if (x == 0) { return 1; }
      elif (x == 1) { return 1; }
      else { return fib(x - 1) + fib(x - 2); }
}
```

As can be seen from the examples above, function calls can be used in expressions.

The use of function calls as arguments to function is permitted by the parser, however this will result in undefined behaviour.

```
fn(fx()); // undefined
```

It is not currently possible to pass arrays as arguments to functions, however it is possible to pass elements of arrays.

```
int main() {
    int  arr[5];
    func(arr) // not supported
}

int main() {
    int  arr[] = {1, 2, 3};
    func(arr[2]) // supported!
}
```

Printing

A print statement has been provided in ecc as a way to provide output. The print statement displays the result of a single expression.

```
int x = 4;
print(x * 2); // will display ‘8’
```

The print statement includes an implicit trailing newline.

Example Programs

I have provided a small set of example programs in the following directory:

`test/examples`

The example programs are:

`Bubsort - A program which sorts an array of integers`

`Fib - A program which uses a recursive function to calculate and output the first 10 fibonacci numbers.`

`Prime - A program which attempts to calculate and output every prime number between 1 and  to` 2^{31} `- 1.`

```
Sum - A program which calculates the sum of integers from 1 to 20 and outputs
the result.
```

You can compile the C source files yourself, or simply use SPIM to run the provided assembly programs.

Considerations

- ecc has been successfully tested on both my physical installation of Ubuntu 20.04 LTS and on the rhul linux cim server. I was however unable to get the parser to run in wsl.

Troubleshooting

After pulling the repository from git, it is likely that the executes in the test and bin directories won't have the correct permissions. To solve this use `chmod 777` on the following files

```
cgen
rdparser
ecc.sh
```