Unit home
Project 1
help2002
Lecture & Workshop recordings on LMS
Schedule
FAQ
Unit outline
C textbooks
OS textbooks
Information resources
Extra reading
Past projects
Recent feedback
Working effectively
Look after yourself!

Project 1 2024 - see also project clarifications (updated 5pm 28th Aug) and marking rubric

The goal of this project is to implement a C11 program to translate programs written in a small mini-language to C11, and to then compile and execute that program.

Successful completion of the project will enhance your understanding of core features of the C11 programming language, functions from the C11 standard library, your operating system's system-calls, the creation and management of operating system processes, and reward familiarisation with online systems' documentation.

The project is due **11:59pm Friday 13th September (end of week 7)**.

(The wording on this page may change to improve grammar and to make things clearer)

Project Description

We're all very familiar with higher-level programming languages, such as Python, Java, and C. They have many well-defined features, and are supported by standard libraries and modules. We'll term these *large-languages*.

At the other end of the scale are *mini-languages*, that are often embedded in other programs that don't require the support of a full programming language, or are invoked from the command-line or in a shellscript. You could consider the use of macros in MS-Excel, or the Unix command-line program *bc* (see *man bc*), as examples of a mini-languages. Chapter 8 of The Art of Unix Programming provides an overview of some (older) mini-languages [not required reading].

This project requires you write a C11 program to compile and execute a program written in a mini-language that we'll name *ml*. Note that there already exists a very successful programming language named *ML* (for Meta Language), but our mini-language is unrelated to ML (or to Machine Learning).

Writing a compiler for any programming language is an enormous task, clearly not one suited for this project. However, what many have recognised, is that C is an excellent language to support other languages, and that C has an extensive *toolchain* supporting compilation and linking. The strategy is to first translate programs written in other languages, such as our *ml*, to C, to compile that translated C code using a standard C compiler, and to finally execute the resultant program. This sequence is often termed *transpiling*, the 'joining' of the words translating and compiling. In this role, C is often described as a high-level assembly language, sometimes a 'wallpaper language'.

The goal of this project is to implement a C11 program, named *runml*, which accepts a ~~single~~ command-line argument providing the pathname of a text file containing a program written in *ml*, *[added 28/8] and any optional command-line arguments to be passed the transpiled program when it is executed* . Successful execution of *runml* will involve checking the syntax of the *ml* program, translating that valid *ml* program to a C11 program, compilation of the resultant C program and, finally, execution of the compiled program.

Our *ml* language

0. the syntax of *ml* programs
   sample01.ml, sample02.ml, sample03.ml, sample04.ml, sample05.ml, sample06.ml, sample07.ml, sample08.ml

1. programs are written in text files whose names end in *.ml*
2. statements are written one-per-line (with no terminating semi-colon)
3. the character '#' appearing anywhere on a line introduces a comment which extends until the end of that line

4. only a single datatype is supported - real numbers, such as *2.71828*
5. identifiers (variable and function names) consist of 1..12 lowercase alphabetic characters, such as *budgie*
6. there will be at most 50 unique identifiers appearing in any program
7. variables do not need to be defined before being used in an expression, and are automatically initialised to the (real) value *0.0*
8. the variables *arg0*, *arg1*, and so on, provide access to the program's command-line arguments which provide real-valued numbers

9. a function must have been defined before it is called in an expression
10. each statement in a function's body (one-per-line) is indented with a *tab* character
11. functions may have zero-or-more formal parameters
12. a function's parameters and any other identifiers used in a function body are *local* to that function, and become unavailable when the function's execution completes

13. programs execute their statements from top-to-bottom and function calls are the only form of control-flow *(yes, the language would be more useful with loops and conditions, but this project is not about designing programming languages - future work for those interested)*

The steps to compile and execute an *ml* program

1. Edit a text file named, for example, *program.ml*

2. Pass *program.ml* as a command-line argument to your *runml* program
3. *runml* validates the *ml* program, reporting any errors
4. *runml* generates C11 code in a file named, for example, *ml-12345.c* (where 12345 could be a process-ID)
5. *runml* uses your system's C11 compiler to compile *ml-12345.c*
6. *runml* executes the compiled C11 program *ml-12345*, passing any optional command-line arguments (real numbers)
7. *runml* removes any files that it created

Project requirements

1. Your project **must** be written in the C11 programming language, in a single source-code file named **runml.c**

2. Your project **must** perform as a standard utility program - checking its command-line arguments, displaying a *usage* message on error, printing 'normal' output to *stdout* and error messages to *stderr*, terminate with a exit status reflecting its execution success.

3. Your project **must not** depend upon any libraries (such as from 3rd-party, downloaded from the internet) other than the system-provided libraries (providing OS, C11, and POSIX functions).

4. All syntax errors detected in invalid *ml* programs **must** be reported via *stderr* on a line commencing with the '!' character. Your *runml* program **must** be able to detect all invalid *ml* programs - EXCEPT that your program **will not** be tested with any invalid *expressions*, so you do not need to validate the syntax of expressions.

5. The only 'true' output produced by your translated and compiled program (when running) is the result of executing *ml*'s **print** statement. Any 'debug' printing should appear on a line commencing with the '@' character.

6. When printed, numbers that are exact integers **must** be printed without any decimal places; other numbers **must** be printed with exactly 6 decimal places.

7. The project *can* be successfully completed without using any dynamic memory allocation in C (such as with *malloc()*). You may choose to use dynamic memory allocation, but will not receive additional marks for doing so.

Assessment

The project is due **11:59pm Friday 13th September (end of week 7)**.

- The project is worth **20% of your final mark** for CITS2002. It will be marked out of 40.
- If your submitted source code in **runml.c** does not compile successfully using the command

  ```
  cc -std=c11 -Wall -Werror -o runml runml.c
  ```

  (if it produces any errors or warnings) you will receive **zero** for the project.
- The project may be completed **individually or in teams of two** (but not teams of three). The choice of project partners is up to you - you will not be automatically assigned a project partner.
- If the project is undertaken by a team of two, both students will receive the same mark for the project.
- Special Consideration (including requests for extensions) will not be granted for students working in a team of two. This also applies if any member of a team of two has a registered University Academic Adjustment Plan (UAAP).
- Your submission will be examined, compiled, and run on a contemporary **Ubuntu Linux** or **macOS** platform. Your project only needs to execute successfully on one of these platforms. Excuses such as *"it worked on my laptop, just not when you tested it!"* will not be accepted.
- This project is subject to UWA's Policy on Assessment - particularly §5.3 *Principles of submission and penalty for late submission*, and Policy on Academic Conduct. In accordance with these policies, you may *discuss* with other students the general principles required to understand this project, but the work you submit **must** be the result of your own team's efforts. All projects will be compared using software that detects significant similarities between source code files. Students suspected of plagiarism will be interviewed and will be required to demonstrate their full understanding of their project submission.
- 20 of the possible 40 marks will come from assessing your design and programming style, including your use of meaningful comments; well chosen identifier names; appropriate choice of basic data-structures, data-types and functions; and appropriate choice of control-flow constructs (**manual marking**).
- 20 of the possible 40 marks will come from the correctness of your solution (**automated marking**), which will assessed by having the correct output (measurements line) when tested with a number of *sysconfig* and *command* input files..

During the marking, attention will obviously be given to the correctness of your solution. However, a correct and efficient solution should not be considered as the perfect, nor necessarily desirable, form of solution. Preference will be given to well presented, well documented solutions that use the appropriate features of the language to complete tasks in an easy to understand and easy to follow manner. That is, do not expect to receive full marks for your project simply because it works correctly. Remember, a computer program should not only convey a message to the computer, but also to other human programmers.

Submission requirements

1. You **must** submit your project electronically using *cssubmit*.

   The *cssubmit* program will display a receipt of your submission. You should print and retain this receipt in case of any dispute. Note also that *cssubmit* does not archive submissions and will simply overwrite any previous submission with your latest submission.

2. Submit *only* a single C11 source-code file named **runml.c**
   Do not submit any header-files, documentation, instructions, ...

3. If working as a team of two, only one team member should make the team's submission.

4. Your submission's C11 source file **must** contain the C11 block comment:

```
//  CITS2002 Project 1 2024
//  Student1:   STUDENT-NUMBER1   NAME-1
//  Student2:   STUDENT-NUMBER2   NAME-2
//  Platform:   Linux  (or Apple)
```

Final information

- You are **strongly** advised to work with another student who is around the same level of understanding and motivation as yourself. This will enable you to discuss your initial design together, and to assist each other to develop and debug your joint solution. Work together - do not attempt to split the project into two equal parts, and then plan to meet near the deadline to join your parts together.

- Please post requests for clarification about any aspect of the project to *help2002* so that all students may remain equally informed.
  Significant clarifications (corrections) will be also added to the **project clarifications** page.

- Workshop #5, at 9am Friday 30th August, will be devoted to answering *your* questions about the project.

- This project spans UWA's non-teaching week. Please note that during this week there will be no lectures, scheduled laboratory sessions, or workshops. However, teaching staff will be reading and responding to *help2002* (just a bit more slowly).

Good luck!

Chris McDonald.
August 2024.

The University of Western Australia

Computer Science and Software Engineering

CRICOS Code: 00126G

Presented by Chris.McDonald@uwa.edu.au