CS 554: Compiler Construction
Graduate Exercise: README

Warren D. (Hoss) Craft

Due: Mon 12/01/2025 (11:59 pm)
Submitted: Mon 12/01/2025 (5 pm)

This is the README file for the "Graduate Exercise," described as Exercise 7: for students enrolled in CS554: rendering to other pro- gramming languages — assigned 6 November — due on Monday 1 December (40pts). This exercise is attached to Project 2.

This README file is accompanied by code and example translations. The primary code can be found in the TAR file in the folder `Project_2_grad_exercise` as:

- `c_codegen.py`: The primary code for generating C code from the AST or CFG of a source WHILE language program.
- `c_compiler.py`: The over-arching code, coordinating the scanning and parsing of a WHILE program, the production of the associated AST and CFG, and the eventual production of the C code.
- `cfg.py`: The primary code for deriving a WHILE program's CFG from the corresponding AST.
- `parser.py`: The primary code for parsing a WHILE program.
- `scanner.py` The primary code for scanning and tokenizing a WHILE program.
- `trees.py` Supporting code for AST structural nodes.

Example WHILE programs are included in the `/examples` folder.

A typical compilation proceeds like this (explicit examples are shown further below):

```
> python c_compiler.py examples/closest_prime.while
```

Such a command produces both the AST-based C code and the CFG-based C code, along with their gcc-compiled executables, saved in separate files in the folder `/examples/compiled`. The original WHILE code and both C code translations are also echoed to the terminal.

Note regarding the `countTriangularNumbers.while` program: The terse description included in the program is incorrect or at least misleading. Instead of producing a count of the triangular numbers from 0 to n, the progam appears to compute the number of triangular numbers from argument i to argument `goUpToNumber`.

7.1 AST as the starting point (20pts)

Here we look at one explicit example of a typical trace of the compiling and execution process, with the trace below slightly edited for brevity. This example shows the invocation of `c_compiler.py` on the `example1-factorial.while` program, resulting in an echoing of the original WHILE code program and the C code versions produced from both the AST and CFG.

```
MacBook-Pro-160:Project_2_grad_exercise warrencraft$ python c_compiler.py examples/example1-
    factorial.while

Source WHILE code:
----------------------------------------------------------------------
{-
Computes the factorial of the number stored in x and leaves the result in z:
-}

y := x;
z := 1;
while y > 1 do
  z := z * y;
  y := y - 1
od;
y := 0;
output := z

----------------------------------------------------------------------

C Code generated from AST:
----------------------------------------------------------------------
/**
 * filename: examples/compiled/example1-factorial_ast.c
 * created:  2025-12-01
 * descr:    C program produced from the AST of
 *           examples/example1-factorial.while.
*/

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {

    // Check if correct num of args provided
    if (argc != 5 ){
        printf("Executable requires 4 integer arguments.\n");
        printf("Usage: <filename> output x y z\n");
        return EXIT_FAILURE;
    }

    // Establish array to store the 4 int values
    int var_values[4];

    // Store the user-supplied initial values.
    for (int i = 0; i < 4; i++) {
        var_values[i] = atoi(argv[i + 1]);
    }

    // Declare & initialize the variables.
    int output = var_values[0];
    int x = var_values[1];
    int y = var_values[2];
    int z = var_values[3];

    printf("\nInitial variable values are:\n");

    // Print initialized variable values to verify:
```

```
    printf("\n");
    printf("output = %d \n", var_values[0]);
    printf("x = %d \n", var_values[1]);
    printf("y = %d \n", var_values[2]);
    printf("z = %d \n", var_values[3]);

    y = x;
    z = 1;
    while (y > 1) {
        z = z * y;
        y = y - 1;
    }
    y = 0;
    output = z;

    // Print final array values:
    printf("\nFinal variable values are: \n");
    printf("\n");
    printf("output = %d \n", output);
    printf("x = %d \n", x);
    printf("y = %d \n", y);
    printf("z = %d \n", z);

    printf("\n");

    return EXIT_SUCCESS;

}
----------------------------------------------------------------------

C code translation saved to: examples/compiled/example1-factorial_ast.c
Successfully compiled examples/compiled/example1-factorial_ast.c
Executable saved to: examples/compiled/example1-factorial_ast


C Code generated from CFG:
----------------------------------------------------------------------
/**
 * filename: examples/compiled/example1-factorial_cfg.c
 * created:  2025-12-01
 * descr:    C program produced from the CFG of
 *           examples/example1-factorial.while.
*/

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {

    // Check if correct num of args provided
    if (argc != 5 ){
        printf("Executable requires 4 integer arguments.\n");
        printf("Usage: <filename> output x y z\n");
        return EXIT_FAILURE;
    }

    // Establish array to store the 4 int values
    int var_values[4];

    // Store the user-supplied initial values.
    for (int i = 0; i < 4; i++) {
        var_values[i] = atoi(argv[i + 1]);
    }

    // Declare & initialize the variables.
    int output = var_values[0];
    int x = var_values[1];
    int y = var_values[2];
    int z = var_values[3];

    printf("\nInitial variable values are:\n");

    // Print initialized variable values to verify:
    printf("\n");
    printf("output = %d \n", var_values[0]);
    printf("x = %d \n", var_values[1]);
    printf("y = %d \n", var_values[2]);
    printf("z = %d \n", var_values[3]);

    y = x;
    z = 1;
    while (y > 1) {
        z = z * y;
        y = y - 1;
    }
    y = 0;
    output = z;

    // Print final array values:
    printf("\nFinal variable values are: \n");
    printf("\n");
    printf("output = %d \n", output);
    printf("x = %d \n", x);
    printf("y = %d \n", y);
    printf("z = %d \n", z);

    printf("\n");

    return EXIT_SUCCESS;

}
------------------------------------------------------------------------

C code translation saved to: examples/compiled/example1-factorial_cfg.c
Successfully compiled examples/compiled/example1-factorial_cfg.c
Executable saved to: examples/compiled/example1-factorial_cfg
```

And here is an example of a subsequent execution of the resulting C executable (already compiled automatically by `c_compiler.py`). The trace shows an initial call of the executable without supplied arguments to obtain a reminder about what arguments are needed. In this case, the only relevant argument is the second one for x. The output gives $6! = 720$.

```
MacBook-Pro-160:Project_2_grad_exercise warrencraft$ ./examples/compiled/example1-factorial_ast
Executable requires 4 integer arguments.
Usage: <filename> output x y z
MacBook-Pro-160:Project_2_grad_exercise warrencraft$ ./examples/compiled/example1-factorial_ast 1
     6 3 4

Initial variable values are:

output = 1
x = 6
y = 3
z = 4

Final variable values are:

output = 720
x = 6
y = 0
z = 720
```

The AST-based C code file in this case, `examples/compiled/example1-factorial_ast.c` is shown below:

```c
/**
 * filename: examples/compiled/example1-factorial_ast.c
 * created:  2025-12-01
 * descr:    C program produced from the AST of
 *           examples/example1-factorial.while.
*/

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {

    // Check if correct num of args provided
    if (argc != 5 ){
        printf("Executable requires 4 integer arguments.\n");
        printf("Usage: <filename> output x y z\n");
        return EXIT_FAILURE;
    }

    // Establish array to store the 4 int values
    int var_values[4];

    // Store the user-supplied initial values.
    for (int i = 0; i < 4; i++) {
        var_values[i] = atoi(argv[i + 1]);
    }

    // Declare & initialize the variables.
    int output = var_values[0];
    int x = var_values[1];
    int y = var_values[2];
    int z = var_values[3];

    printf("\nInitial variable values are:\n");

    // Print initialized variable values to verify:
    printf("\n");
    printf("output = %d \n", var_values[0]);
    printf("x = %d \n", var_values[1]);
    printf("y = %d \n", var_values[2]);
    printf("z = %d \n", var_values[3]);

    y = x;
    z = 1;
    while (y > 1) {
        z = z * y;
        y = y - 1;
    }
    y = 0;
    output = z;

    // Print final array values:
    printf("\nFinal variable values are: \n");
    printf("\n");
    printf("output = %d \n", output);
    printf("x = %d \n", x);
    printf("y = %d \n", y);
    printf("z = %d \n", z);

    printf("\n");

    return EXIT_SUCCESS;

}
```

Other examples of C code files produced from WHILE programs appear in the

`/examples/compiled` folder (along with their associated executables) as:

- `closest_prime_ast.c`
- `countTriangularNumbers_ast.c`
- `example1-factorial_ast.c`
- `example6-collatz_ast.c`
- `example13-fibonacci_ast.c`
- `primescounter_ast.c`

7.2 CFG as the starting point (20pts)

Each invocation of the `c_compiler.py` on a WHILE program produces both a AST-based C program and a CFG-based C program, so we have already seen this process above in section 7.1.

Instead, here we show another explicit example of the generated C code, now based on the CFG for the `closest_prime.while` program.

```c
/**
 * filename: examples/compiled/closest_prime_cfg.c
 * created:  2025-12-01
 * descr:    C program produced from the CFG of
 *           examples/closest_prime.while.
*/

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {

    // Check if correct num of args provided
    if (argc != 10 ){
        printf("Executable requires 9 integer arguments.\n");
        printf("Usage: <filename> closestprime i input j mod output sqrt stop stop1\n");
        return EXIT_FAILURE;
    }

    // Establish array to store the 9 int values
    int var_values[9];

    // Store the user-supplied initial values.
    for (int i = 0; i < 9; i++) {
        var_values[i] = atoi(argv[i + 1]);
    }

    // Declare & initialize the variables.
    int closestprime = var_values[0];
    int i = var_values[1];
    int input = var_values[2];
    int j = var_values[3];
    int mod = var_values[4];
    int output = var_values[5];
    int sqrt = var_values[6];
    int stop = var_values[7];
    int stop1 = var_values[8];

    printf("\nInitial variable values are:\n");

    // Print initialized variable values to verify:
    printf("\n");
    printf("closestprime = %d \n", var_values[0]);
    printf("i = %d \n", var_values[1]);
    printf("input = %d \n", var_values[2]);
    printf("j = %d \n", var_values[3]);
    printf("mod = %d \n", var_values[4]);
    printf("output = %d \n", var_values[5]);
    printf("sqrt = %d \n", var_values[6]);
    printf("stop = %d \n", var_values[7]);
    printf("stop1 = %d \n", var_values[8]);

    mod = input;
    while (!(mod == 1 || mod == 0)) {
        mod = mod - 2;
    }
    if (mod == 1) {
        input = input - 2;
    } else {
        input = input - 1;
    }
    i = input;
    stop1 = 0;
    while (i >= 2 && stop1 == 0) {
        sqrt = 1;
        while (sqrt * sqrt <= i) {
            sqrt = sqrt + 1;
        }
        j = 3;
        stop = 0;
        while (j <= sqrt && stop == 0) {
            mod = i;
            while (mod > 0) {
                mod = mod - j;
            }
            if (mod == 0) {
                stop = 1;
            } else {
                j = j + 2;
            }
        }
        if (j > sqrt) {
            stop1 = 1;
        } else {
            i = i - 2;
        }
    }
    closestprime = i;
    output = closestprime;

    // Print final array values:
    printf("\nFinal variable values are: \n");
    printf("\n");
    printf("closestprime = %d \n", closestprime);
    printf("i = %d \n", i);
    printf("input = %d \n", input);
    printf("j = %d \n", j);
    printf("mod = %d \n", mod);
    printf("output = %d \n", output);
    printf("sqrt = %d \n", sqrt);
    printf("stop = %d \n", stop);
    printf("stop1 = %d \n", stop1);

    printf("\n");

    return EXIT_SUCCESS;

}
```

And here is an example of a subsequent execution of the resulting C executable (already compiled automatically by `c_compiler.py`). The trace shows an initial call of the executable without supplied arguments to obtain a reminder about what arguments are needed. In this case, the only relevant argument is the third one for '`input`'. The output gives 97 as the closest prime less than 100.

```
MacBook-Pro-160:Project_2_grad_exercise warrencraft$ ./examples/compiled/closest_prime_cfg
Executable requires 9 integer arguments.
Usage: <filename> closestprime i input j mod output sqrt stop stop1
MacBook-Pro-160:Project_2_grad_exercise warrencraft$ ./examples/compiled/closest_prime_cfg 1 2
    100 4 5 6 7 8 9

Initial variable values are:

closestprime = 1
i = 2
input = 100
j = 4
mod = 5
output = 6
sqrt = 7
stop = 8
stop1 = 9

Final variable values are:

closestprime = 97
i = 97
input = 99
j = 11
mod = -2
output = 97
sqrt = 10
stop = 0
stop1 = 1
```

Other examples of CFG-based C code files produced from WHILE programs appear in the `/examples/compiled` folder (along with their associated executables) as:

- `closest_prime_cfg.c`
- `countTriangularNumbers_cfg.c`
- `example1-factorial_cfg.c`
- `example6-collatz_cfg.c`
- `example13-fibonacci_cfg.c`
- `primescounter_cfg.c`