Top Down Parser for Simplified C Language

Schrodinger ZHU Yifan (118010469)

April 14, 2021

Contents

Chapter 1

Build and Run

1.1 Docker

We have already provided a pre-configured dockerfile and corresponding command line tools with the distributed source code.

- Dockerfile
- scc
- scc-graphical

The command line tools will automatically build the image for you. Therefore, the best way is **not to build the docker on yourself**. To get the text based compiler output, you should type the following:

```
./scc input.c output.txt
```

Or you can simply let the program output to stdout via:

```
./scc input.c output.txt
```

In the same time, if you want to get the graphical result in PNG format, you can use

```
./scc-graphical input.c output.png
```

Notice that, in this case you must provide the output path.

1.2 Build Environment

To build the project on yourself, you will need the following dependencies:

- Linux (perhaps > 5 is perferred)
- CMake ≥ 3.10
- GCC ≥ 10 (C++17 required)
- Ninja ≥ 1.10

1.3 Build Command

With the above equipment, you can then start the build with the following commands:

```
cd /path/to/project
mkdir build
cd build
cmake .. -DCMAKE_BUILD_TYPE=Release -GNinja
ninja
```

After the compilation is finished, you will see an executable named `scc` lies in the build directory.

1.4 Command Line Interface

The program provides a tiny CLI to facilitate you. to check the functions, invoke the tool with `-h` flag and see what happens:

```
$ ./scc -h
USAGE: scc [input file] [-o <output file>] [-g]
OPTIONS:
  -h,--help           print this help message and exit
  -o,--output <path>  set output file path
  -g,--graphviz       output graphviz format
NOTICES:
    if no input or output is set, the program will use standard
  io on default.
```

You can follow the above instructions to use the tool. However, notice that in this way, the output file for the graphical mode will be a dot file, which can be visualized by `graphviz`. Please convert the file to your favoured formats on your own.

Chapter 2

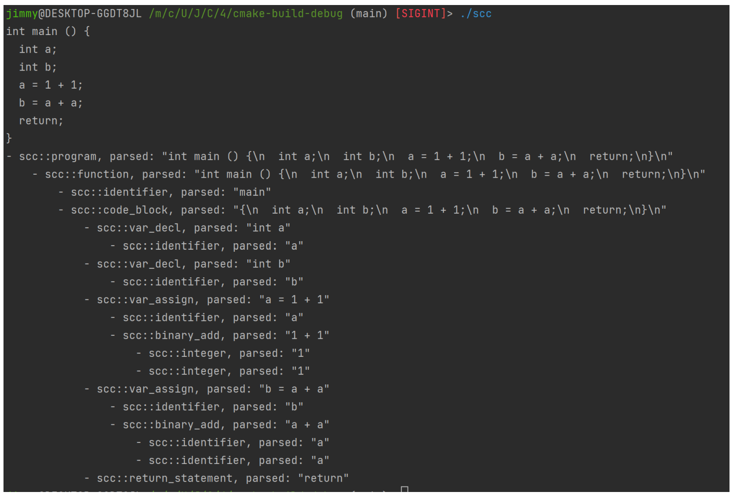
Results and Screenshots

2.1 Notice

The grammar used in this project is much more complex than the required grammar in the project instruction. Apart from the required contents, we also support:

- Unary expressions
- return statements with value
- Airbitary function call
- Multi-level nested blocks
- Flexible variable declarition
- Function and external symbol declarition

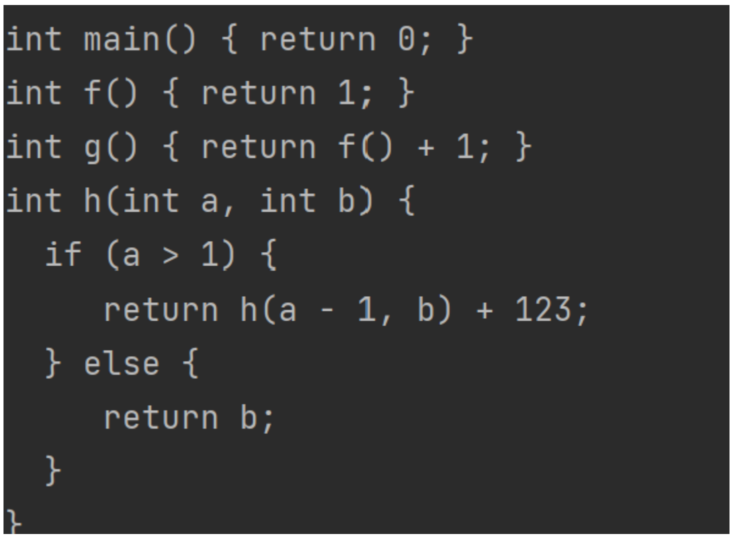
2.2 Text-based Outputs



```
jimmy@DESKTOP-GGDT8JL /m/c/U/J/C/4/cmake-build-debug (main) [SIGINT]> ./scc
int main () {
  int a;
  int b;
  a = 1 + 1;
  b = a + a;
  return;
}
- scc::program, parsed: "int main () {\n  int a;\n  int b;\n  a = 1 + 1;\n  b = a + a;\n  return;\n}\n"
    - scc::function, parsed: "int main () {\n  int a;\n  int b;\n  a = 1 + 1;\n  b = a + a;\n  return;\n}\n"
        - scc::identifier, parsed: "main"
        - scc::code_block, parsed: "{\n  int a;\n  int b;\n  a = 1 + 1;\n  b = a + a;\n  return;\n}\n"
            - scc::var_decl, parsed: "int a"
                - scc::identifier, parsed: "a"
            - scc::var_decl, parsed: "int b"
                - scc::identifier, parsed: "b"
            - scc::var_assign, parsed: "a = 1 + 1"
                - scc::identifier, parsed: "a"
                - scc::binary_add, parsed: "1 + 1"
                    - scc::integer, parsed: "1"
                    - scc::integer, parsed: "1"
            - scc::var_assign, parsed: "b = a + a"
                - scc::identifier, parsed: "b"
                - scc::binary_add, parsed: "a + a"
                    - scc::identifier, parsed: "a"
                    - scc::identifier, parsed: "a"
            - scc::return_statement, parsed: "return"
```

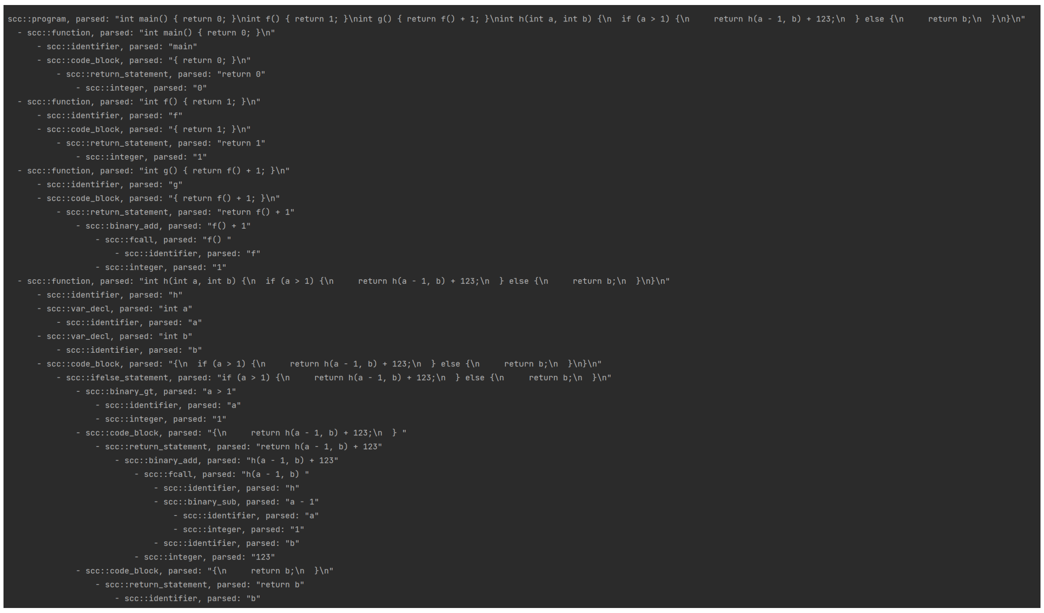
Figure 2.1: Text Based Output for a Simple Program

This example shows some simple results for expressions and empty return statements. Let us see another one with multiple function definitons, function calls and if-else blocks:



```
int main() { return 0; }
int f() { return 1; }
int g() { return f() + 1; }
int h(int a, int b) {
  if (a > 1) {
     return h(a - 1, b) + 123;
  } else {
     return b;
  }
}
```

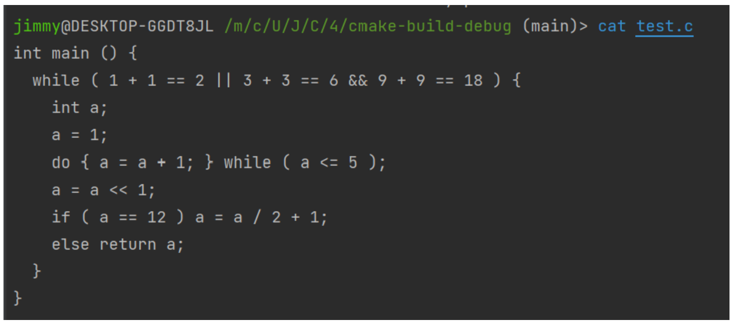
Figure 2.2: Text Based Output with Multiple Functions and Control Flows (Part 1)



```
scc::program, parsed: "int main() { return 0; }\nint f() { return 1; }\nint g() { return f() + 1; }\nint h(int a, int b) {\n  if (a > 1) {\n     return h(a - 1, b) + 123;\n  } else {\n     return b;\n  }\n}\n"
  - scc::function, parsed: "int main() { return 0; }\n"
      - scc::identifier, parsed: "main"
      - scc::code_block, parsed: "{ return 0; }\n"
          - scc::return_statement, parsed: "return 0"
              - scc::integer, parsed: "0"
  - scc::function, parsed: "int f() { return 1; }\n"
      - scc::identifier, parsed: "f"
      - scc::code_block, parsed: "{ return 1; }\n"
          - scc::return_statement, parsed: "return 1"
              - scc::integer, parsed: "1"
  - scc::function, parsed: "int g() { return f() + 1; }\n"
      - scc::identifier, parsed: "g"
      - scc::code_block, parsed: "{ return f() + 1; }\n"
          - scc::return_statement, parsed: "return f() + 1"
              - scc::binary_add, parsed: "f() + 1"
                  - scc::fcall, parsed: "f() "
                      - scc::identifier, parsed: "f"
                  - scc::integer, parsed: "1"
  - scc::function, parsed: "int h(int a, int b) {\n  if (a > 1) {\n     return h(a - 1, b) + 123;\n  } else {\n     return b;\n  }\n}\n"
      - scc::identifier, parsed: "h"
      - scc::var_decl, parsed: "int a"
          - scc::identifier, parsed: "a"
      - scc::var_decl, parsed: "int b"
          - scc::identifier, parsed: "b"
      - scc::code_block, parsed: "{\n  if (a > 1) {\n     return h(a - 1, b) + 123;\n  } else {\n     return b;\n  }\n}\n"
          - scc::ifelse_statement, parsed: "if (a > 1) {\n     return h(a - 1, b) + 123;\n  } else {\n     return b;\n  }\n"
              - scc::binary_gt, parsed: "a > 1"
                  - scc::identifier, parsed: "a"
                  - scc::integer, parsed: "1"
              - scc::code_block, parsed: "{\n     return h(a - 1, b) + 123;\n  } "
                  - scc::return_statement, parsed: "return h(a - 1, b) + 123"
                      - scc::binary_add, parsed: "h(a - 1, b) + 123"
                          - scc::fcall, parsed: "h(a - 1, b) "
                              - scc::identifier, parsed: "h"
                              - scc::binary_sub, parsed: "a - 1"
                                  - scc::identifier, parsed: "a"
                                  - scc::integer, parsed: "1"
                              - scc::identifier, parsed: "b"
                          - scc::integer, parsed: "123"
              - scc::code_block, parsed: "{\n     return b;\n  }\n"
                  - scc::return_statement, parsed: "return b"
                      - scc::identifier, parsed: "b"
```

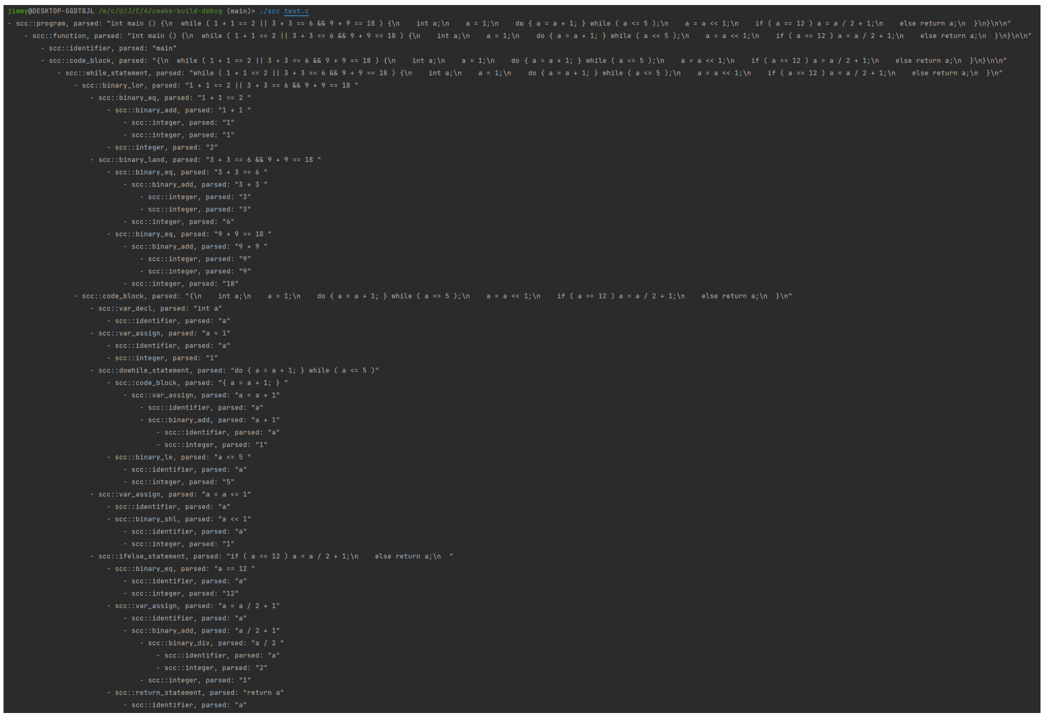
Figure 2.3: Text Based Output with Multiple Functions and Control Flows (Part 2)

Here is another example with complex and nested control flows:



```
jimmy@DESKTOP-GGDT8JL /m/c/U/J/C/4/cmake-build-debug (main)> cat test.c
int main () {
  while ( 1 + 1 == 2 || 3 + 3 == 6 && 9 + 9 == 18 ) {
    int a;
    a = 1;
    do { a = a + 1; } while ( a <= 5 );
    a = a << 1;
    if ( a == 12 ) a = a / 2 + 1;
    else return a;
  }
}
```

Figure 2.4: Text Based Output with Complex Control Flows (Part 1)



```
jimmy@DESKTOP-GGDT8JL /m/c/U/J/C/4/cmake-build-debug (main)> ./scc test.c
- scc::program, parsed: "int main () {\n  while ( 1 + 1 == 2 || 3 + 3 == 6 && 9 + 9 == 18 ) {\n    int a;\n    a = 1;\n    do { a = a + 1; } while ( a <= 5 );\n    a = a << 1;\n    if ( a == 12 ) a = a / 2 + 1;\n    else return a;\n  }\n}\n\n"
  - scc::function, parsed: "int main () {\n  while ( 1 + 1 == 2 || 3 + 3 == 6 && 9 + 9 == 18 ) {\n    int a;\n    a = 1;\n    do { a = a + 1; } while ( a <= 5 );\n    a = a << 1;\n    if ( a == 12 ) a = a / 2 + 1;\n    else return a;\n  }\n}\n\n"
    - scc::identifier, parsed: "main"
    - scc::code_block, parsed: "{\n  while ( 1 + 1 == 2 || 3 + 3 == 6 && 9 + 9 == 18 ) {\n    int a;\n    a = 1;\n    do { a = a + 1; } while ( a <= 5 );\n    a = a << 1;\n    if ( a == 12 ) a = a / 2 + 1;\n    else return a;\n  }\n}\n\n"
      - scc::while_statement, parsed: "while ( 1 + 1 == 2 || 3 + 3 == 6 && 9 + 9 == 18 ) {\n    int a;\n    a = 1;\n    do { a = a + 1; } while ( a <= 5 );\n    a = a << 1;\n    if ( a == 12 ) a = a / 2 + 1;\n    else return a;\n  }\n"
        - scc::binary_lor, parsed: "1 + 1 == 2 || 3 + 3 == 6 && 9 + 9 == 18 "
          - scc::binary_eq, parsed: "1 + 1 == 2 "
            - scc::binary_add, parsed: "1 + 1 "
              - scc::integer, parsed: "1"
              - scc::integer, parsed: "1"
            - scc::integer, parsed: "2"
          - scc::binary_land, parsed: "3 + 3 == 6 && 9 + 9 == 18 "
            - scc::binary_eq, parsed: "3 + 3 == 6 "
              - scc::binary_add, parsed: "3 + 3 "
                - scc::integer, parsed: "3"
                - scc::integer, parsed: "3"
              - scc::integer, parsed: "6"
            - scc::binary_eq, parsed: "9 + 9 == 18 "
              - scc::binary_add, parsed: "9 + 9 "
                - scc::integer, parsed: "9"
                - scc::integer, parsed: "9"
              - scc::integer, parsed: "18"
        - scc::code_block, parsed: "{\n    int a;\n    a = 1;\n    do { a = a + 1; } while ( a <= 5 );\n    a = a << 1;\n    if ( a == 12 ) a = a / 2 + 1;\n    else return a;\n  }\n"
          - scc::var_decl, parsed: "int a"
            - scc::identifier, parsed: "a"
          - scc::var_assign, parsed: "a = 1"
            - scc::identifier, parsed: "a"
            - scc::integer, parsed: "1"
          - scc::dowhile_statement, parsed: "do { a = a + 1; } while ( a <= 5 )"
            - scc::code_block, parsed: "{ a = a + 1; } "
              - scc::var_assign, parsed: "a = a + 1"
                - scc::identifier, parsed: "a"
                - scc::binary_add, parsed: "a + 1"
                  - scc::identifier, parsed: "a"
                  - scc::integer, parsed: "1"
            - scc::binary_le, parsed: "a <= 5 "
              - scc::identifier, parsed: "a"
              - scc::integer, parsed: "5"
          - scc::var_assign, parsed: "a = a << 1"
            - scc::identifier, parsed: "a"
            - scc::binary_shl, parsed: "a << 1"
              - scc::identifier, parsed: "a"
              - scc::integer, parsed: "1"
          - scc::ifelse_statement, parsed: "if ( a == 12 ) a = a / 2 + 1;\n    else return a;\n  "
            - scc::binary_eq, parsed: "a == 12 "
              - scc::identifier, parsed: "a"
              - scc::integer, parsed: "12"
            - scc::var_assign, parsed: "a = a / 2 + 1"
              - scc::identifier, parsed: "a"
              - scc::binary_add, parsed: "a / 2 + 1"
                - scc::binary_div, parsed: "a / 2 "
                  - scc::identifier, parsed: "a"
                  - scc::integer, parsed: "2"
                - scc::integer, parsed: "1"
            - scc::return_statement, parsed: "return a"
              - scc::identifier, parsed: "a"
```

Figure 2.5: Text Based Output with Complex Control Flows (Part 2)

Take a close look to the corresponding output for binary and logical expressions: the precedences are handled correctly.

2.3 Visualized Outputs

As a mentioned before, one can also get the graph of AST with the tool. The graph of the last example in the text-based part is shown below:

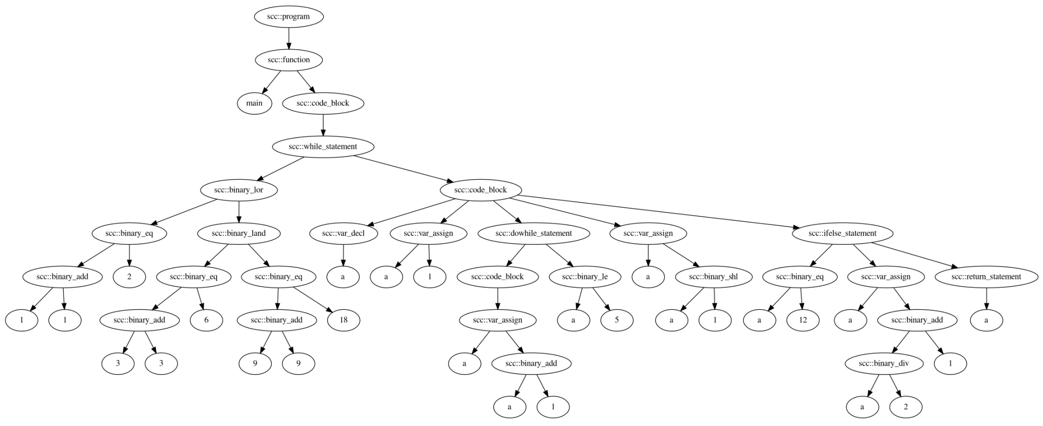


Figure 2.6: Text Based Output with Complex Control Flows (Part 2)

Chapter 3

Implementation Details

As many technical details have already been introduced in my previous reports, I will not expand them again here. I will just give some core insights of this assignment.

3.1 Handle of Precedence

We use grammar rule to handle precedence directly. Consider a simple program with addition and multiplcation, the grammar with precedence embedded can be expressed with:

```
primary = INTEGER | '(' addition ')'
multiplication = primary '*' multiplication | primary
addition = multiplication '+' addition | multiplication
```

That is, to represent the precedence to grammar, one can split the expression into multiple levels, each level use previous level (the level with higher priority) as operands and also refers back to that level. In this way, the parser directly give the correct order in the AST.

3.2 Memoization Speed Up

As grammar level increases, a trivial descent down parser will get into terriflying complexing, since the info in the lower levels is not reused, which results in repetitious re-trials of a large amount of grammar rules. This problem can be solved by introduce the following hash indices:

$$(position, grammar) \to failure|(success, length)$$

In this way, if a former rule is tested in a given position, the parser can directly return the result and thus save us a lot of time.

3.3 Visualization

Visualization works as a tiny code generator: just go down through the ast and dump all subtrees and then do the links. This process transform the program control flows into dot language.