Project 1: Simple Code Generation

Raspberry Pi:
Jaime Gould, Qinghong Shao, Warren Craft

Dept. of Computer Science
University of New Mexico

28 October 2025

Introduction

Project 1 tasked us with the creation of a compiler to transform programs written in the simple declarative WHILE programming langauge, into 64-bit RISC-V code, and to create for each compiled WHILE program a corresponding C program from which to run and supply inputs to the resulting RISC-V program.

We implemented the compiler in the Python programming language [5]. A graphical overview of the compiler code structure appears in Fig. 1, with the organization of the compiler components unsurprisingly similar to typical textbook schematics [1, pg. 9]. A compiler module (`compiler.py`) served to coordinate scanner, parser, and codegen class modules, and also served to generate the eventual C program to control the generated RISC-V assembly program.

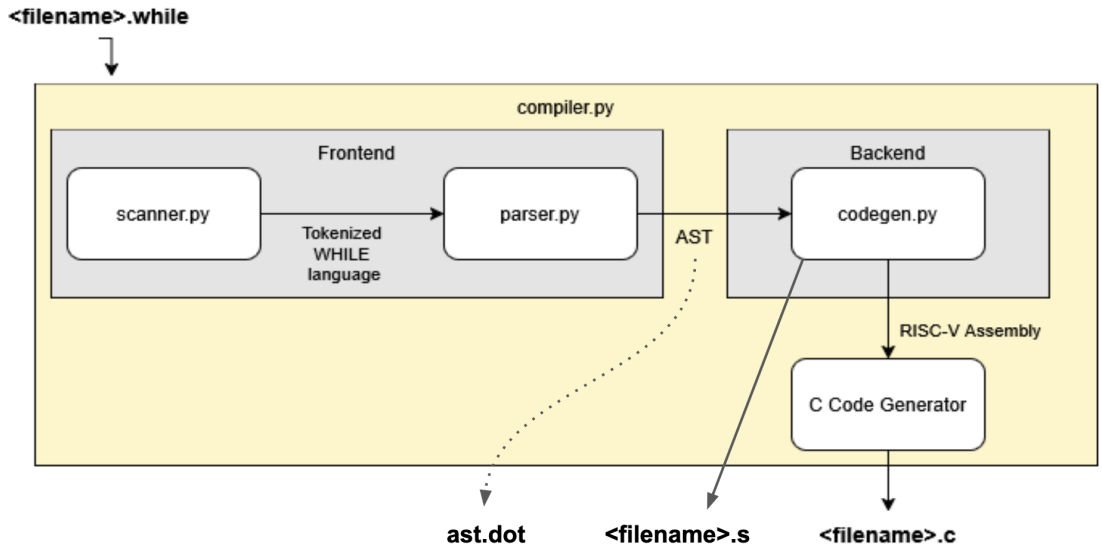

Figure 1: An overview of our compiler code structure.

Along the way, the compiler also coordinates the production of a DOT language version of both the parse tree and the abstract syntax tree (AST), producing `.dot` files that can then be viewed using a variety of graphical programs (such as Graphviz) or editor extensions (such as the Interactive Graphviz Dot Preview for Visual Studio Code [6]).

The four python code modules, `compiler.py`, `scanner.py`, `parser.py`, and `codegen.py` (hereafter referred to more simply as the compiler, scanner, parser, or codegen code) are all included in the submitted TAR file, as are numerous example WHILE program files. The example WHILE program files are divided into "good syntax" and "bad syntax" folders, according to whether or not they should successfully compile and run.

As a quick-start implementation detail to see the compiler perform its magic, the basic command would be:

```
python compiler.py <filename>
```

for any WHILE program `<filename>`. That will scan and parse the program, producing a parse tree and AST (each visualizable as described earlier) in the form of `parse_tree.dot` and `ast.dot` files in the same folder as `compiler.py`, as well as the RISC-V assembly language version as `<filename>.s` and the controller C program as `<filename>.c` If you are not already on a RISC-V machine, port the `.s` and `.c` files over to a RISC-V machine and compile and run with:

```
gcc -o <exec> <filename>.s <filename>.c
/.<exec> args
```

where `args` is a space-separated list of initial values for all variables (in alphabetical order) used in the original WHILE program. Without `args`, you will get a brief usage-error message. An example of the process starting at the `gcc` command is shown in Fig. 2.

```
wdcraft@risc-machine-2:~/CS554_2025Fall/PROJECT_01$ gcc -o fact example1-factorial.s example1-factorial.c
wdcraft@risc-machine-2:~/CS554_2025Fall/PROJECT_01$ ./fact
Executable requires 4 arguments.
Usage: <filename> output x y z
wdcraft@risc-machine-2:~/CS554_2025Fall/PROJECT_01$ ./fact 0 6 0 0

Initial variable values are:
output = 0
x = 6
y = 0
z = 0

Final variable values are:
output = 720
x = 6
y = 0
z = 720
```

Figure 2: An example of compiling the resulting `.s` and `.c` files and running the executable, both without and with arguments supplied, for the compiled `example1-factorial.while` program.

The rest of the report is organized roughly by Tasks explicitly set out in the project specifications, presenting and briefly discussing material on the following topics:

- Concrete Syntax (in which we also discuss the grammar);
- Test Programs;

- Scanner & Parser;
- The Abstract Syntax Tree (AST);
- RISC-V Code Generation;
- C Code Generation
- Performance Testing
- Effort & Division of Work

Concrete Syntax

$$
\begin{array}{rcll}
\mathbf{U} & \rightarrow & \mathrm{A} \mid \mathrm{B} \mid \mathrm{C} \mid \mathrm{D} \mid \mathrm{E} \mid \mathrm{F} \mid \mathrm{G} \mid \mathrm{H} \mid \mathrm{I} \mid \mathrm{J} \mid \mathrm{K} \mid \mathrm{L} \mid \mathrm{M} \mid \mathrm{N} \mid \mathrm{O} \mid \mathrm{P} \mid \mathrm{Q} \mid \mathrm{R} \mid \mathrm{S} \mid \mathrm{T} \mid \mathrm{U} \mid \mathrm{V} \mid \mathrm{W} \mid \mathrm{X} \mid \mathrm{Y} \mid \mathrm{Z} & \\
\mathbf{L} & \rightarrow & \mathrm{a} \mid \mathrm{b} \mid \mathrm{c} \mid \mathrm{d} \mid \mathrm{e} \mid \mathrm{f} \mid \mathrm{g} \mid \mathrm{h} \mid \mathrm{i} \mid \mathrm{j} \mid \mathrm{k} \mid \mathrm{l} \mid \mathrm{m} \mid \mathrm{n} \mid \mathrm{o} \mid \mathrm{p} \mid \mathrm{q} \mid \mathrm{r} \mid \mathrm{s} \mid \mathrm{t} \mid \mathrm{u} \mid \mathrm{v} \mid \mathrm{w} \mid \mathrm{x} \mid \mathrm{y} \mid \mathrm{z} & \\
\mathbf{D} & \rightarrow & 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 & \\
\mathbf{Z} & \rightarrow & 0 \mid \mathbf{D} & \\
x & \rightarrow & \mathbf{UX} \mid \mathbf{LX} & \text{scalar variable names} \\
\mathbf{X} & \rightarrow & \varepsilon \mid \mathbf{YX} & \\
\mathbf{Y} & \rightarrow & \mathbf{U} \mid \mathbf{L} \mid \mathbf{Z} \mid ' \mid - \mid \_ & \\
n & \rightarrow & 0 \mid \mathbf{DN} & \text{integer literals} \\
\mathbf{N} & \rightarrow & \varepsilon \mid \mathbf{ZN} & \\
o_a & \rightarrow & + \mid - \mid * & \text{binary arithmetic operators} \\
o_b & \rightarrow & \text{and} \mid \text{or} & \text{binary boolean operators} \\
o_r & \rightarrow & = \mid < \mid <= \mid > \mid >= & \text{binary relational operators} \\
a & \rightarrow & x \mid n \mid a\ o_a\ a \mid (a) & \text{arithmetic expressions} \\
b & \rightarrow & \text{true} \mid \text{false} \mid \text{not}[b] \mid b\ o_b\ b \mid a\ o_r\ a \mid [b] & \text{boolean expressions} \\
c & \rightarrow & x := a \mid \text{skip} \mid c; c \mid \text{if } b \text{ then } c \text{ else } c \text{ fi} & \text{commands} \\
 & & \mid \text{while } b \text{ do } c \text{ od} &
\end{array}
$$

We developed the concrete syntax for our language, beginning with our scalar variable names and literals. According to the initial specifications, variables names must begin with a letter (either upper or lower case), and can be composed of alphanumeric characters, the single quote character, dashes, and underscores. We define integer literals as numbers with no leading zeroes. The remaining terms are taken from Project 1's description. The basic syntax is elaborated into the conterxt-free grammar (CFG) presented in the next section.

Grammar

A context-free grammar (CFG) specification for the project version of the `while` language is shown below. For convenience, we let x stand in for any possible variable name, and n stand in for any integer literal. The details of the variable names and integer literals are shown in the previous section on concrete syntax. To make things a little more concise, we have also used the notation $(= \mid < \mid <= \mid > \mid >=)$ in the last line of the `bool_factor` production rule to cover all five possible relation cases. An empty program is not allowed, and an empty block of statements within the `while`, `if-then`, and `else` is also not allowed — to produce an `if-then` statement without an `else` component, for example, one would use a `skip` statement inside the `else` block. For clarity, the a, b, and c of the syntax presented in the previous section, are expanded to `arith_expr`, `bool_expr`, and `statement`, respectively.

```
        program ::= statement_list
statement_list ::= statement | statement;statement_list
     statement ::= x := expr
                 | skip
                 | if bool_expr then statement_list else statement_list fi
                 | while bool_expr do statement_list od
          expr ::= arith_expr | bool_expr
    arith_expr ::= arith_term | arith_term + arith_term
                 | arith_term - arith_term
    arith_term ::= arith_factor | arith_factor * arith_factor
  arith_factor ::= x | n | (arith_expr)
     bool_expr ::= bool_term | bool_term or bool_term
     bool_term ::= bool_factor | bool_factor and bool_factor
   bool_factor ::= x | true | false
                 | not[bool_expr]
                 | [bool_expr]
                 | arith_expr (=|<|<=|>|>=) arith_expr
```

Some details of the grammar specification are easier to understand if we note the following:

- The desired precedence among the logical `not`, `and`, and `or` operators can be preserved in the same way we preserve the desired precedence among the arithmetic operators `+ - *`. The logical `not`, `and`, and `or` are analogous to the "factor", "term", and "expr" productions for arithmetic expressions, respectively.
- The potential ambiguity of an `if-then-else` construct [1, pp. 93–95] is resolved by the inclusion of the closing symbol `fi` [7].
- The variable name x can represent a numerical value or a boolean value and thus appears in both the `arith_factor` and `bool_factor` production rules.

Test Programs

We wrote 43 simple programs in the WHILE language in order to test the robustness of our scanner and parser. 17 were good syntax, meaning our scanner and parser should accept them and output a correct AST. 26 of these were bad syntax, designed to identify syntax errors in the scanner (incorrect variable names, etc.) or parsing errors in the parser (e.g. incorrect ordering of tokens). These tests can be found in the supplementary material included with the project report and are named according to the same scheme as the initial provided test files (and stored in separate `good_syntax` *vs.* `bad_syntax` folders inside the `examples` folder).

Later we added the files provided by the instructor and prior students to our collection of programs.

One issue we discovered during testing was an error in the scanner that highlighted the need to identify the less than or equal to operator (<=) before the less than (<) and equal (=) operators, otherwise the tokenizer identifies them as separate tokens.

Another issue we identified was the need to order the "ignore" regular expression containing the two dashes before the regular expression identifying the minus (-) operator.

Scanner

To implement the scanning portion of our compiler, we employed the `re` python library to make use of regular expressions, allowing us to generate tokens from text files containing the WHILE language syntax.

The regular expressions in python are the following:

```
# Reserved Keywords
keywords = {"true", "false", "not", "skip", "if", "then", "else",
            "fi", "while", "do", "od", "and", "or"}
# Regular Expressions identifying Tokens in our Language
token_specification = [
    # ignore comments and white space
    ("ignore",     r"(--.*|\{-(.|\n|\r)*?-\})|\s+"),
    ("lpar",       r"\("),                 # Right parenthesis
    ("rpar",       r"\)"),                 # Left parenthesis
    ("lbrac",      r"\["),                 # Right bracket
    ("rbrac",      r"\]"),                 # Left bracket
    ("int",        r"0|([1-9])\d*"),       # Integer
    ("var",        r"[A-Za-z](\w|'|_)*"), # Variables
    ("assign",     r":="),                 # Assignment
    ("seq",        r";"),                  # Command sequencing
    ("op_a",       r'[+\-*]'),             # Arithmetic operators
    ("op_r",       r'<=|>=|=|<|>'),        # Binary relational operators
    ("newline",    r'\n'),                 # Line endings
    ("mismatch",   r'.'),                  # Any other character
]
```

We "or" all of these regular expressions together and call the `re` library's `finditer` function. We iterate through the input WHILE file's text to search for token matches. If any token results in a "mismatch" type, this identifies a syntax error in the input file.

As we were constructing our scanner, we noted that the order in which we identified these expressions was important. For instance, our "ignore" regular expression must come before our arithmetic operators expression, otherwise the scanner identifies the sequence of two dashes "– –" as two arithmetic minus operator tokens. Placing our "ignore" regular expression first in the `token_specification` list eliminates this error. Our binary relational operators must also be placed in a specific order so that less than or equal to "<=" is not identified as two separate tokens, "<" and "=" (the same for greater than or equal to ">=").

The reserved keywords in the WHILE language also technically fall within the variable names category. As we scan the input file, if we find a match with the "var" token we check that it is not one of our reserved keywords. If it is, we change the type of token to the reserved keyword.

The scanner returns a list of identified tokens consisting of their type (syntactic category) and their value, along with the line number and position number for the information in the original WHILE file, which is then passed to the parser. Below is a table of example token outputs (omitting the line and position numbers):

WHILE code	**Tokens**

while x > 1 do	`(type='while', value='while')`
	`(type='var', value='x')`
	`(type='op_r', value='>')`
	`(type='int', value=1)`
	`(type='do', value='do')`

x := x - 2	`(type='var', value='x')`
	`(type='assign', value=':=')`
	`(type='var', value='x')`
	`(type='op_a', value='-')`
	`(type='int', value=2)`

od	`(type='od', value='od')`

Parser

The parser is initialized with the list of tokens produced by the scanner, then employs top-down recursive descent (with a rarely-used look-ahead or "peek" option), to fit the token types to the grammar presented earlier and produce both a parse tree and abstract syntax tree (AST).

The Parser.parse() method is the entry point into the parsing process, and is shown in the listing below. Basically, `parse()` calls the recursive `statement_seq()`, that in turn calls "lower-level" methods to deal with successively more specific aspects of each statement being constructed.

```
    def parse(self):
        '''
        Top-level method to start the parsing process, assuming
        the Parser has been initialized with an appropriate list of
        tokens associated with a program consisting of a sequence
        of statements.
        '''

        pt_stmts, ast_stmts = self.statement_seq()
        self.program_pt = ('prog',) + (pt_stmts,)
        self.program_ast = ast_stmts

        if self.current_token_index < len(self.tokens) - 1:
            # parsing ended prematurely, possibly due to an
            # unexpected token or a missing sequencing token ';'
            _last_token = self.tokens[self.current_token_index]
            _value = _last_token.value
            _line = _last_token.line
            raise SyntaxError(
                    "Parsing ended prematurely, possibly due to an "
                    "unexpected token or missing seq token ';'. "
                    "Last token processed was "
                    f"'{_value}' on line {_line}.")

        return (self.program_pt, self.program_ast)
```

The `statement_seq()` method eventually calls `statement()` (see listing below), which then directs further nested processing according to the type of statement. Notice here in particular the recognition of the case for an `assignment` statement, signaled by the current token type `VAR` and subsequent token type `ASSIGN`:

```
    def statement(self):
        '''
        Pursue different parsing method(s) based on current statement.
        '''
        <portion omitted here>

        # assignment (e.g., x := 3 + 2 * y)
        if self.current_token.type == VAR and self.peek_ahead(ASSIGN):
            pt_result, ast_result = self.parse_assignment_stmt()
            pt_result = (STMT, pt_result)
            return (pt_result, ast_result)

        # skip (e.g., if x > 0 then x := x + 1 else skip)
```

```
        if self.peek(SKIP):
            pt_result, ast_result = self.parse_skip_stmt()
            pt_result = (STMT, pt_result)
            return (pt_result, ast_result)

        # if-then-else (if x < y then x := 0 else y := 0)
        if self.current_token.type == IF:
            pt_result, ast_result = self.parse_if_stmt()
            pt_result = (STMT, pt_result)
            return (pt_result, ast_result)

        # while-do (while x < 10 do x := x + 1)
        if self.peek(WHILE):
            pt_result, ast_result = self.parse_while_stmt()
            pt_result = (STMT, pt_result)
            return (pt_result, ast_result)
```

The `parse_assignment_stmt()` method is shown in the listing below, first processing the expression associated with the left-hand side of the assignment, then consuming the `ASSIGN` operator, and then processing the right-hand side of the assignment, and eventually returning the results back up to the `statement()` method, which eventually returns the result to the `statement_seq()` method, which eventually returns its result up to the `parse()` method.

```
    def parse_assignment_stmt(self):
        '''
        Parsing assignment statements of the form x := expr .
        Method assumes caller has already verified that the statement
        to be parsed is indeed an assignment statement.
        '''
        pt_left, ast_left = self.expr()
        self.consume(ASSIGN)
        pt_right, ast_right = self.expr()
        pt_result = (ASSIGN, pt_left, pt_right)
        ast_result = (ASSIGN, ast_left, ast_right)
        return (pt_result, ast_result)
```

Ultimately, the compiler module then hands the resulting AST off to the RISC-V Assembly Generator module.

Abstract Syntax Tree

During parsing, an abstract syntax tree (AST) is constructed alongside the detailed parse tree, capturing the essential characteristics of the syntactical structure. To visualize the resulting AST, the compiler automatically translates the AST into DOT language and saves the result in a `.dot` file, which can then be interpreted in the Graphviz visualization tool [2, 3] or interpreted in a text editor using one of several available editor extensions (for example, using the "Graphviz Interactive Preview" [6] extension in the Visual Studio Code editor [4]).

Examples of generated ASTs are shown in Figures 3, 4, and 5, resulting, respectively, from the scanning and parsing of example program files:

`example1-factorial.while`, `example6-collatz.while`, `example1-fibonacci.while`,

along with accompanying figure insets showing the actual content of the program files.

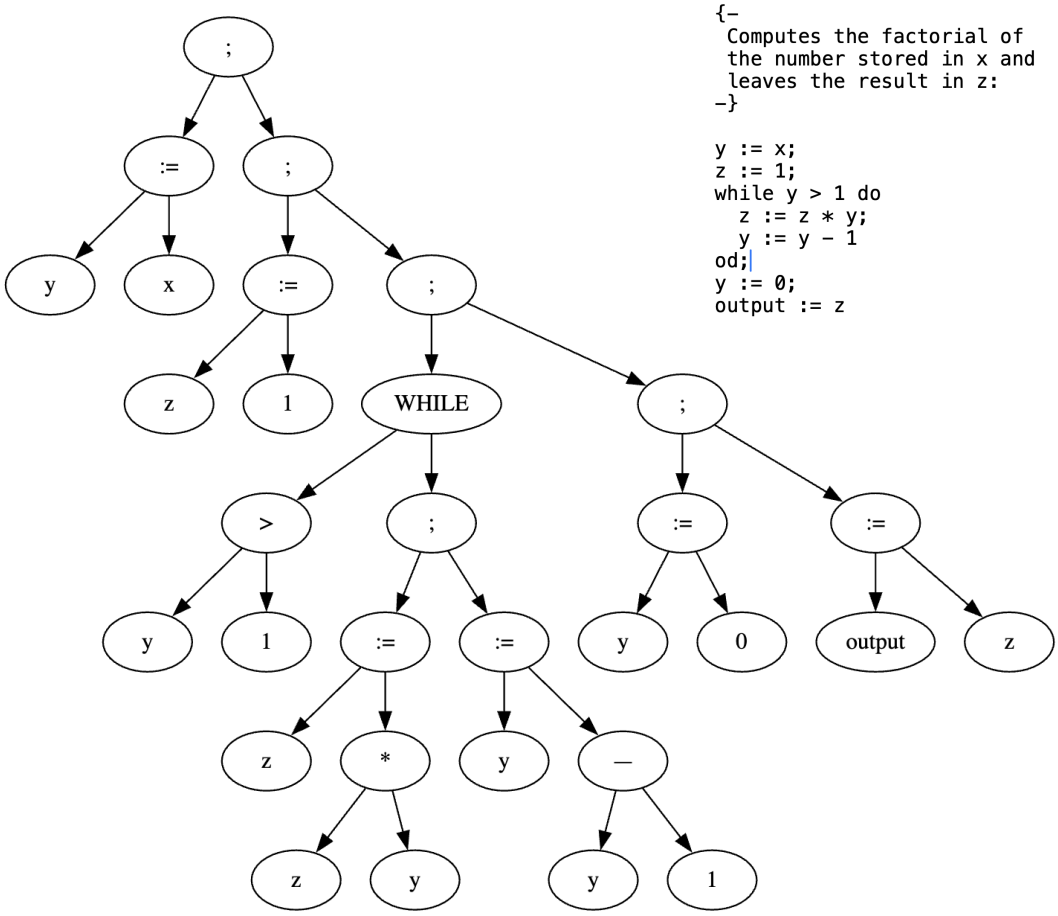

Figure 3: Example abstract syntax tree (AST) abstracted from the parse tree produced from the `example1-factorial.while` program (shown in inset). The string version of the AST was converted to DOT code and displayed using Graphviz. See other examples below in Figs 4 and 5.

Notable in the ASTs is a rightward skew as sequences of commands are nested in such a way as to produce exactly two children for each sequence ";" node, and extended sequences of com-

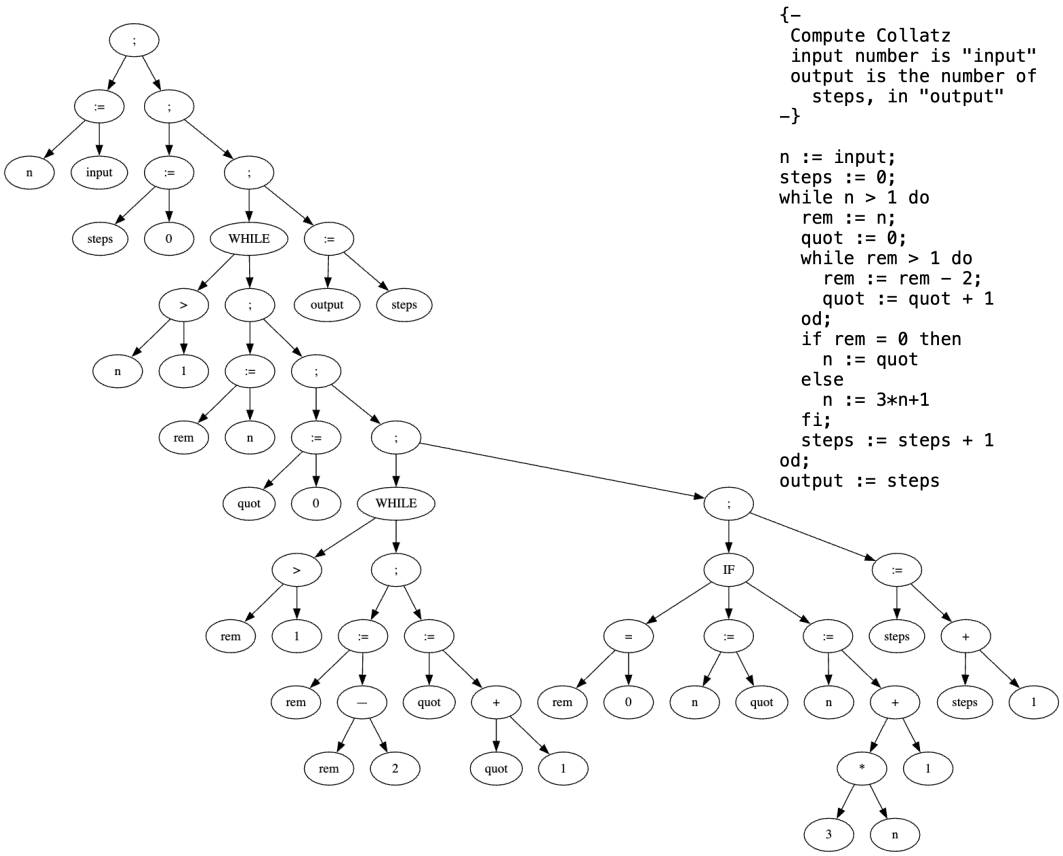

Figure 4: Example AST produced in parsing the `example6-collatz.while` program (shown in inset). See other examples in Figs 3 and 5.

mands expand rightward (through the right child) each time, in agreement with the grammar shown earlier. The resulting AST is then passed to the RISC-V Assembly Generator.

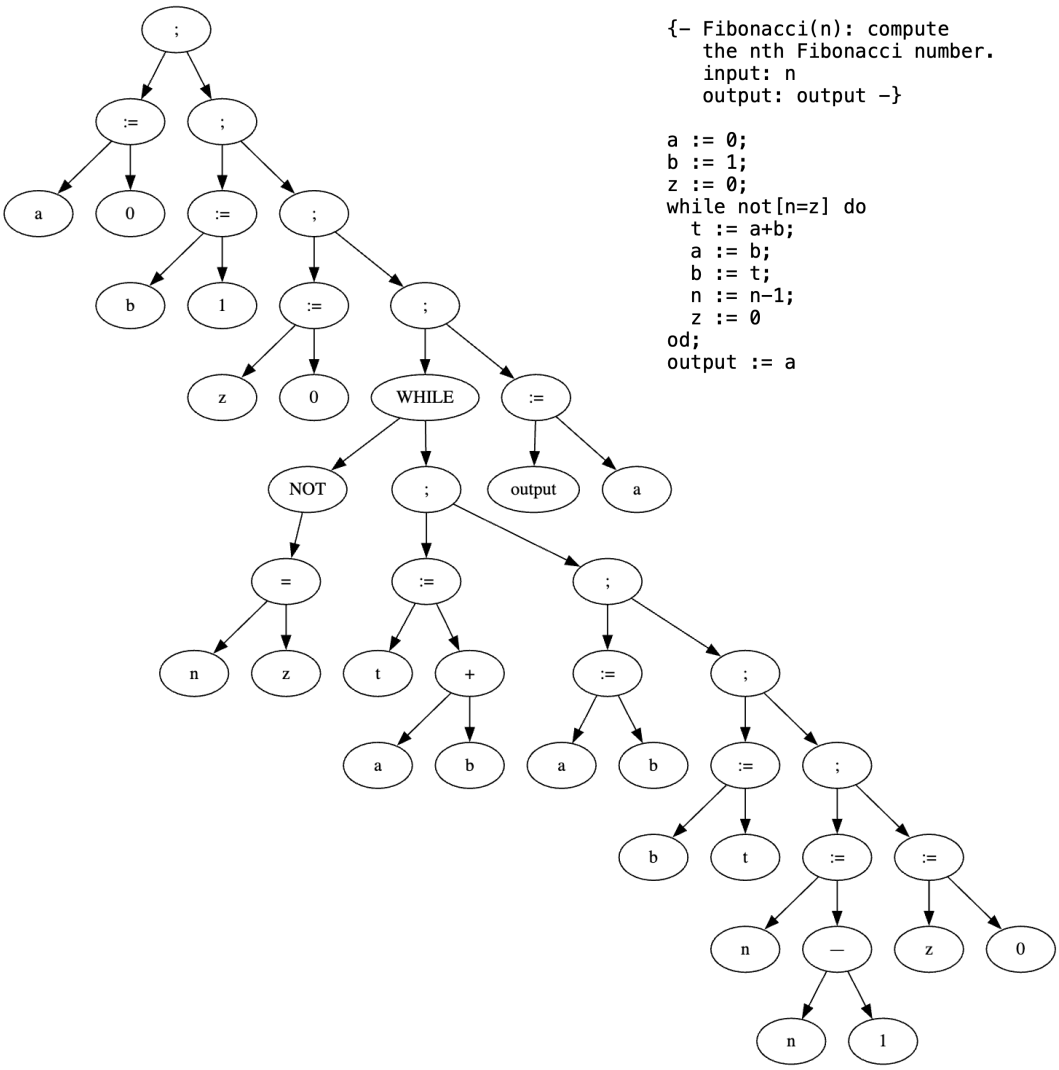

Figure 5: Example AST produced in parsing the `example13-fibonacci.while` program (shown in inset). See other examples in Figs 3 and 4

RISC-V Assembly Generation

We traverse the AST in order to generate the RISC-V assembly code. Upon receiving the AST, we do an initial traversal of the tree and collect all variables present in the program (as needed to comply with the project specifications). The pointer to the input list of arguments is located in register `a0`, where the alphabetically sorted variables can be accessed in memory like so: `0(a0), 8(a0), 16(a0),` ... . Each variable is allocated 64 bits.

Values

Literals are signed integers that can be represented in 64-bit registers. True and False in RISC-V are 1 and 0, respectively.

Basic Operations

We implemented the basic arithmetic and boolean operations in RISC-V as can be seen in the table below:

Operation	Symbol	Type	RISC-V Instruction	Info
addition	+	o_a	`add t0, t0, t1`	`t0 = t0 + t1`
subtraction	-	o_a	`sub t0, t0, t1`	`t0 = t0 - t1`
multiplication	*	o_a	`mul t0, t0, t1`	`t0 = t0 * t1`
and	and	o_b	`and t0, t0, t1`	`t0 = t0` $\wedge$ `t1`
or	or	o_b	`or t0, t0, t1`	`t0 = t0` $\vee$ `t1`

equals	=	o_r	`sub t0, t0, t1`
			`seqz t0, t0`
				`t0 = t0 - t1`
				`if t0 = 0: t0 = 1`
				`else: t0 = 0`

less than	<	o_r	`slt t0, t0, t1`	`if t0 < t1: t0 = 1` `else: t0 = 0`
greater than	>	o_r	`slt t0, t1, t0`	`if t1 < t0: t0 = 1` `else: t0 = 0`
less than or equal to	<=	o_r o_r	`slt t0, t1, t0` `xori t0, t0, 1`	`t0` $\leq$ `t1` $\equiv \neg$`(t1 < t0)`
greater than or equal to	>=	o_r o_r	`slt t0, t0, t1` `xori t0, t0, 1`	`t1` $\leq$ `t0` $\equiv \neg$`(t0 < t1)`
not	not	unary operator	`seqz t0, t0`	`t0` = $\neg$`t0`

We make use of RISC-V's built in instructions for arithmetic, boolean, and relational operations. Addition, subtraction, multiplication, and, or, less than, and greater than all require a single instruction. All other operations require two.

Our RISC-V generator currently uses two temp registers, `t0` and `t1` for all operations. We utilize the stack (pointer located in `sp`) in order to keep track of our computed values as we traverse the AST (more information in the Stack Implementation section).

While and If

We implement while and if using RISC-V's conditional jump instruction (`beqz`) and labels. The basic structure of an if statement in RISC-V is the following:

```
    # If Statement
    ########################################################
    # Boolean condition code is generated and inserted here
    ########################################################

    ld t0, 0(sp)            # Result is placed into t0
    beqz t0, else_label_1   # Conditional jump

    ########################################################
    # Then condition code is generated and inserted here
    ########################################################

    j end_label_1           # Jump

```

```
15 else_label_1:                # Else label
16     ########################################################
17     # Else condition code is generated and inserted here
18     ########################################################
19
20 end_label_1:                 # End label
```

We can see from the conditional jump that if the boolean expression evaluates to true (1), the code between lines 7 and 13 is executed. The command `j end_label_1` on line 13 then skips to line 20. If the boolean expression evaluates to false (0), the code between lines 15 and 20 is executed, which contains instructions generated from the "else" block.

The basic structure of a while statement in RISC-V is the following:

```
21     # While Statement
22 while_label_1:
23     # Condition
24     ########################################################
25     # Boolean condition code is generated and inserted here
26     ########################################################
27
28     ld t0, 0(sp)             # Result is placed into t0
29     beqz t0, end_label_2     # Conditional jump
30
31     # Do
32     ########################################################
33     # Body of while loop code is generated and inserted here
34     ########################################################
35     # Od
36     j while_label_1
37 end_label_2:
```

We can see that the condition is evaluated on each loop by placing the while label first. If the condition evaluates to false (0), the instruction on line 29 jumps to line 37, and the loop ends. Otherwise, it proceeds through the code generated for the body of the while loop between lines 29 and 36, then jumps back up to line 22.

Stack Implementation

We utilize the stack in order to keep track of our computed boolean and arithmetic expressions. In our first pass of RISC-V code generation, we grew and shrunk the stack dynamically. When pushing a value onto the stack, we decreased the stack pointer by 8 bytes and placed the value at `0(sp)`. When the value was popped, we moved the value located at the top of the stack (`0(sp)`) into a temporary register and increased the stack pointer by 8 bytes, deallocating the space.

However, we noted that this significantly increased the number of `addi` operations in the generated RISC-V code as a result of the stack pointer constantly being adjusted as we moved up and down the arithmetic/boolean expressions in the AST. We realized that the amount of space we needed in the stack corresponded to the maximum number of branches in our arithmetic and boolean expressions, so we could allocate the appropriate amount of space in our code prologue and deallocate it in our code epilogue before the generated function returns.

We keep track of the stack pointer in our code generator. While traversing a tree, if we run into an operator node we recursively call the left child, and then the right. The left child

receives the same branch value as its parent (with the default being 0), and the right child receives a branch value equal to its parent's branch value plus 1. The location it stores its computed value on the stack is then `{8*branch}(sp)`.

Figure 6 illustrates a simple example of how this process works.

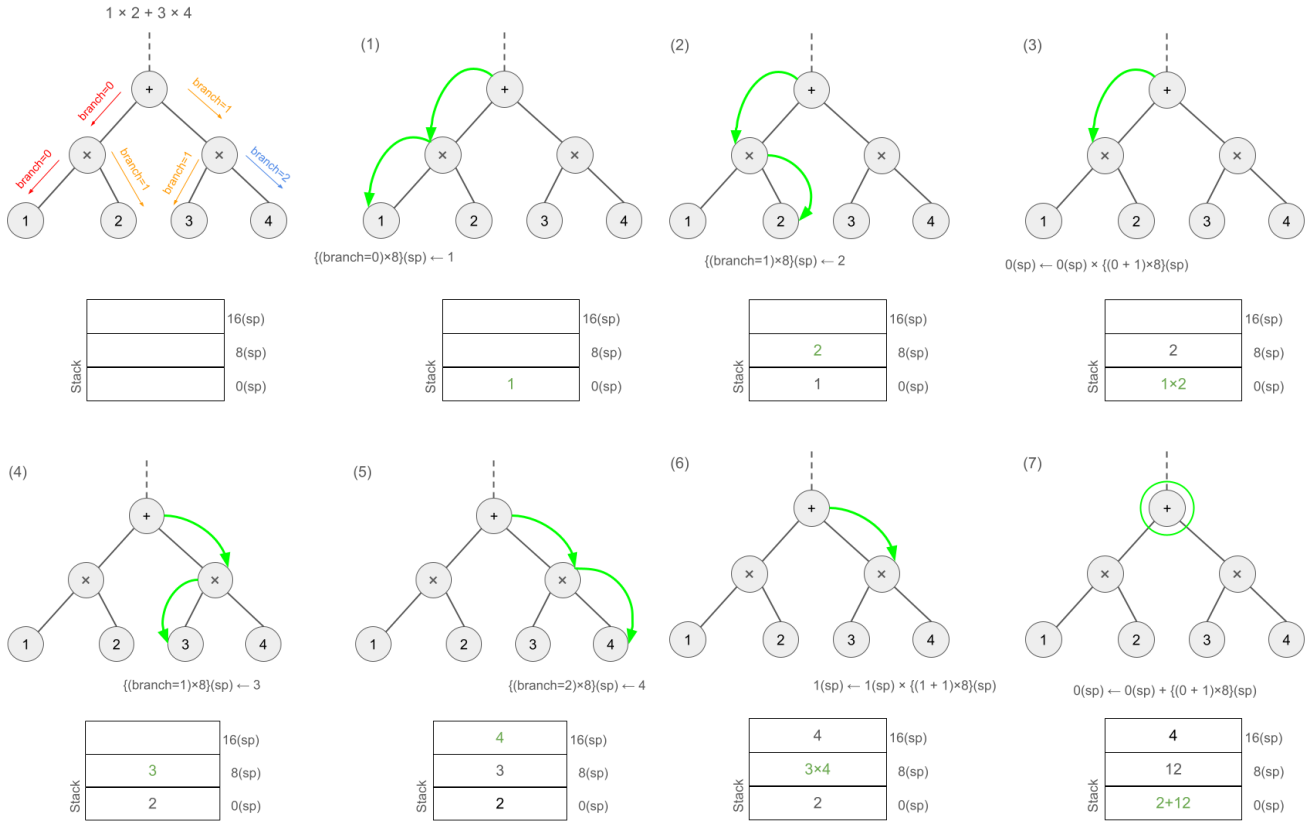

Figure 6: Given the AST representing the arithmetic expression $1 \times 2 + 3 \times 4$, the figure shows how the values will be computed and placed into the stack according to the branch and stack pointer.

Because the tree is evaluated depth-first left to right, values that need to be saved for parent operations are not overwritten until they are no longer needed. The maximum number of branches is computed in the initial tree traversal when we collect the number of variables.

C Code Generation

Back in the compiler module, we finish up the compilation process with the generation of the C code (and file) for eventually calling and supplying arguments for the RISC-V assembly code generated in the previous step. The Python code for this step is relatively short and simple, so we include its listing further below.

This step benefits from some processing undertaken in the RISC-V code generation step, accessing the list of variables found in the original WHILE program being compiled and generating the C code for printing the variables and their current values (see lines 2-9 below, for example). Python f strings make it relatively easy to dynamically create program-specific code and user output messages.

```
    # First, some helpful details and sub-strings
    num_vars = len(codegen.variables)
    printVars = " ".join(codegen.variables)
    printVals = ""
    for i in range(num_vars):
        printVals += (
                f"    printf(\"{codegen.variables[i]} = %lld \\n\", "
                f"(long long)var_array[{i}]);\n"
        )

    # Construct the C code as a Python string (to be printed to a file)
    c_code = (
              "#include <stdio.h>\n"
          +   "#include <stdlib.h>\n"
          +   "\n"
          +   f"extern void {codegen.name}(int64_t *var_arr);\n"
          +   "\n"
          +   "int main(int argc, char *argv[]) {\n"
          +   "\n"
          +   "    // Check if correct num of args provided\n"
          + f"    if(argc != {num_vars + 1}) " + "{\n"
          + f'        printf("Executable requires {num_vars} arguments.\\n");\n
    '
          + f'        printf("Usage: <filename> {printVars}\\n");\n'
          +   "        return EXIT_FAILURE;\n"
          +   "    }\n"
          +   "\n"
          + f"    // Establish array to store the {num_vars} int values\n"
          + f"    int64_t var_array[{num_vars}];\n"
          +   "\n"
          +   "    // Initialize the values\n"
          + f"    for (int i = 0; i < {num_vars}; i++) " + "{\n"
          +   "        var_array[i] = atoll(argv[i + 1]);\n"
          +   "    }\n"
          +   "\n"
          +   "    // Print initialized array values to verify:\n"
          +   '    printf("\\n");\n'
          +   '    printf("Initial variable values are: \\n");\n'
          + printVals
          +   "\n"
          +   f"    {codegen.name}(var_array);\n"
          +   "\n"
          +   "    // Print final array values:\n"
          +   '    printf("\\nFinal variable values are: \\n");\n'
          + printVals
          +   '    printf("\\n");\n'
          +   "\n"
          +   "    return EXIT_SUCCESS;\n"
          +   "}\n"
    )

    c_file_name = args.filename.replace('.while', '.c')
    with open(c_file_name, 'w') as f:
        f.write(c_code)
    print(f"\nC code saved to: {c_file_name}")
```

Performance Testing

On the `risc-v` machine, we were able to evaluate the performance of both the *compilation* of WHILE programs to RISC-V programs and the subsequent *running* of the eventual executables. We evaluated performance for a number of example WHILE programs, and showcase here the results for the following four example programs:

- `example1-factorial.while`
- `example6-collatz.while`
- `example13-fibonacci.while`
- `primescounter.while`
- `veryveryverysimpleloop.while`

These are all quite simple programs, of course, but it is gratifying to discover that they compile and run quite efficiently (in fact, at least one of us is amazed that they compile and run at all).

The execution results below come from running the eventual executable on the RISC-V machine with the full complement of args and prepending the command with the `time` command. The argument listed is for the primary "input", omitting the initialization values for the other (internal) variables; similarly, the "output" gives the primary output of interest — for example, in the case of the factorial, an input of 5 corresponds to an output of 5! = 120.

Compilation Results

`.while` File	Compile Time (sec)	C Size	Assembly Size	Instructions
`example1-factorial`	0.4439	1226 bytes	1382 bytes	50
`example6-collatz`	0.6784	1473 bytes	2854 bytes	107
`example13-fibonacci`	0.5159	1440 bytes	1734 bytes	66
`primescounter`	0.3470	1467 bytes	4158 bytes	155
`veryveryverysimpleloop`	0.3262	915 bytes	804 bytes	25

Execution Results

`.while` File	Argument (main)	Output (main)	Run Time (secs)
`example1-factorial`	5	120	0.004
`example1-factorial`	20	2432902008176640000	0.003
`example6-collatz`	1000	111	0.004
`example13-fibonacci`	90	2880067194370816120	0.004
`primescounter`	100	25	0.004
`primescounter`	1000	168	0.063
`primescounter`	10000	1229	13.22
`veryveryverysimpleloop`	1000000000	-1	17.186

Effort & Division of Work

Based on class presentations, it seems to have been common for project groups to explicitly divide up the workload by modules (or at least sub-modules), with someone explicitly assigned to deal primarily with the scanner, another person dealing with the parser, *etc.* The work flow in our group evolved a little differently, with stages of competitive problem-solving occurring first, in which each person worked on (say) the scanner, or the parser, and then at some point when some one of us was clearly making more progress than the others, we then adopted that person's approach and began more group-contributory efforts.

That being acknowledged, it has also been the case that each of us ended up being more explicitly responsible for certain components of the project than others. The group's adopted scanner module was largely due to initial work by Jaime, for example, with relatively minor contributions from the rest of the group. The parser code we eventually adopted was largely initiated by Hoss (with lots of subsequent contributions from everyone). And Qinghong was largely responsible for getting us going on the RISC-V code generation, the code for which was then greatly elaborated and refined by Jaime, with lots of questions and some more minor contributions by Hoss. Work on the formal project report was well-distributed among the group members, with Jaime taking the lead on substantial portions but Qinghong and Hoss contributing subtantially as well.

Hoss reports consuming some 20+ hrs per week on the project and related class work. Jaime and Qinghong also report 20+ hrs per week.

References

[1] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. 3rd ed. Cambridge, MA, USA: Morgan Kaufman Publishers (an imprint of Elsevier), 2023.

[2] John Ellson et al. "Graphviz—Open Source Graph Drawing Tools". In: *9th International Syposium on Graph Drawing (2001)*. Ed. by P. Mutzel, M. Jünger, and S. Leipert. Vol. 2265. Lecture Notes in Computer Science. Heidelberg: Springer, 2002. URL: https://link.springer.com/chapter/10.1007/3-540-45848-4_57.

[3] Graphviz community. *Graphviz - Graph Visualization Software*. Version 14.0.1. Dec. 14, 2019. URL: https://www.graphviz.org.

[4] Microsoft Corporation. *Visual Studio Code*. Version 1.105. Integrated Development Environment. 2025. URL: https://code.visualstudio.com/ (visited on 10/13/2025).

[5] Python Software Foundation. *Python Language Reference*. Version 3.8–3.12. Accessed on 2025-10-28. Python Software Foundation. 2023. URL: https://docs.python.org/3/reference/ (visited on 10/28/2025).

[6] tintinweb. *Interactive Graphviz Dot Preview for Visual Studio Code*. https://github.com/tintinweb/vscode-interactive-graphviz. Accessed: 11 October 2025. 2025.

[7] Wikipedia contributors. *Dangling else — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Dangling_else. [Online; last accessed 7-October-2025]. 2025.