Project 1: Terminators

Youssef Amin, Zhuo Li, Yue Hu

October 2025

1 Task 1

This task defines the formal grammar of the WHILE language using Backus–Naur Form (BNF) notation. The grammar specifies the syntactic rules for programs, commands, arithmetic and boolean expressions, identifiers, numbers, and comments. It serves as the foundation for constructing the scanner, lexer, and parser by outlining how valid WHILE programs are structured and how tokens are combined to form syntactically correct statements.

1.1 Formal Grammar of WHILE (Backus-Naur Form)

```
<program> ::= <command>
<command> ::= <command> ";" <SimpleCommand>
            | <SimpleCommand>
<SimpleCommand> ::= <id> ":=" <arithExpr>
            | "skip"
            | "if" <boolExpr> "then" <command> "else" <command> "fi"
            | while <boolExpr> "do" <command> "od"

<arithExpr> ::= <arithExpr> "+" <arithTerm>
            | <arithExpr> "-"  <arithTerm>
            | <arithTerm>
<arithTerm> ::= <arithTerm> "*" <arithfactor>
            | <arithfactor>
<arithfactor> ::= <num>
            | <id>
            | "(" <arithExpr> ")"
<boolExpr> ::= <boolExpr> "or" <boolTerm>
            | <boolTerm>
<boolTerm> ::= <boolTerm> "and" <boolFactor>
            | <boolFactor>
<boolFactor> ::= "true"
            | "false"
            | "[" <boolExpr> "]"
            | "not" <boolExpr>
            | <arithExpr> <bop> <arithExpr>
<bop> ::= "=" | "<" | "<=" | ">"| ">="
<num> ::= <Digits>
<Digits> ::= <Digit> <Digits>
           | <Digit>
<Digit>  ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
<id> ::= <Letter> <restid>
<restid> ::= <Digit><restid>
           |<Letter><restid>
           |"’"<restid>
           |"_"<restid>
           | EPSILON
<Letter> ::= "a" | "b" | ... | "z"
```

```
            | "A" | "B" | ... | "Z"
<comment> ::= "{-" <CommentText "-}"
             |"--" <CommentText> <Newline>
<Newline> ::= "\n" | "\r\n"
<CommentText> ::= <AnyChar><CommentText>
                | EPSILON
<AnyChar> ::= any character except the sequence "-}" or EOF
```

Since comments are difficult to define formally, and they do not affect the syntactic structure of the program, they will be handled at the lexical analysis level by the scanner. In other words, the comment text be completely ignored during lexical processing, and the parser will never see comment tokens.

2 Task 2

This task provides a collection of WHILE language test programs designed to evaluate the correctness and coverage of the scanner and parser. Each test exercises different language constructs such as assignments, arithmetic expressions, conditionals, loops, and nested boolean logic. The programs also include both styles of comments (- ... - and – ...), ensuring that the lexer properly handles comment parsing and whitespace skipping. Collectively, these examples verify that the grammar and parser can handle varied syntax patterns and operator precedence.

```
{-
This program tests the valid syntax for WHILE
includes arithmatic, while loop and if statment

Author: Youssef Amin
Input: inOutput: out
-}

-- inline comment wow :3

x := in;
y := ((x * 2) + 10) + 1;

while [x < y] do
    if [x < 10] then
        x := x + 2
    else
        y := y - 2
    fi
od;

out := x;
output := out
```

3 Task 3

This task is a C++ implementation of a hand written scanner, lexer, and recursive descent parser for the WHILE language. It defines a hierarchy of AST node structures representing arithmetic, boolean, and command constructs such as assignments, conditionals, and loops.

Scanner Specification The syntax of the scanner is defined as follows:

1. **Keywords:** `if`, `then`, `else`, `fi`, `while`, `do`, `od`, `skip`, `true`, `false`, `not`, `and`, `or`. When a keyword conflicts with an identifier (`ID`), it is recognized as a keyword in priority.

(a) Keywords

`if`	IF
`then`	THEN
`else`	ELSE
`fi`	FI
`while`	WHILE
`do`	DO
`od`	OD
`skip`	SKIP
`true`	TRUE
`false`	FALSE
`not`	NOT
`and`	AND
`or`	OR

(b) Special Symbols

`:=`	ASSIGN
`;`	SEMI
`( )`	LPAR/RPAR
`[ ]`	LBRACK/RBRACK
`+`	PLUS
`-`	MINUS
`*`	MULTI
`=`	EQ
`<`	LT
`<=`	LEQ
`>`	GT
`>=`	GEQ

(c) Identifiers and Numbers

identifier	ID
number	NUM

2. **Special symbols:** `:=`, `;`, `( )`, `[ ]`, `+`, `-`, `*`, `=`, `<`, `>`, `<=`, `>=`. Multi-character operators have higher priority than single-character ones.
3. **Other tokens:** variables (`ID`) and numbers (`NUM`). Identifiers may contain both uppercase and lowercase letters. They are defined by the following regular expressions:

$$\texttt{ID} ::= [A-Za-z][A-Za-z0-9\_']^*$$
$$\texttt{NUM} ::= [0-9]^+$$

4. **Whitespace:** consists of spaces, newlines, and tab characters, which are generally ignored. Carriage returns (\r) and comments are also skipped.
5. **Line tracking:** the scanner records the current line number to ensure that error reports indicate the precise location of a token.
6. **End of file:** finally, an `END_OF_FILE` token is returned.

Matching order:

1. Skip whitespace and comments.
2. Try multi-character operators (`:=`, `<=`, `>=`).
3. Then try single-character symbols (`; ( ) [ ] + - * = < >`).
4. If a digit is found, recognize a `NUM`.
5. If a letter is found, scan until the end of the identifier, then check the keyword table; if not found, classify as `ID`.
6. Otherwise, return an `INVALID` token or raise a lexical error.

Parser and Abstract Syntax Tree (AST) The parser adopts a **top-down recursive descent** approach, following the grammar defined in Task 1. And the AST reflects operator precedence and associativity. At equal precedence, operations form a left-branching structure due to left associativity.

Abstract Syntax Tree (AST) Design. The node types and their meanings (consistent with the implementation) are as follows:

1. **Program:** `Program`, `Seq(left, right)`, `Assign(name, expr)`, `Skip`, `If(cond, then, else)`, `While(cond, body)`.
2. **Arithmetic expressions:** `Int(val)`, `Var(name)`, `ABin(op, left, right)`, where `op` $\in \{+, -, *\}$.
3. **Boolean expressions:** `Bool(b)`, `Not(e)`, `BBin(op, left, right)`, `Rel(op, lhs, rhs)`, where `op` $\in \{=, <, <=, >, >=, \text{and}, \text{or}\}$.

3.1 Example of the lexer and parser

Input:

```
x := 1; while x < 5 do x := x + 1 od
```

Lexer output:

```
IDENT(x), ASSIGN(:=), INT(1), SEMI(;), WHILE, IDENT(x), LT(<), INT(5), DO, IDENT(x),
ASSIGN(:=), IDENT(x), PLUS(+), INT(1), OD
```

Parser output:

```
Program
 └── Seq
      ├── Assign(x, Int(1))
      └── While(
              Rel(<, Var(x), Int(5)),
              Assign(x, ABin(+, Var(x), Int(1)))
          )
```

3.2 Test Files With Syntax Errors

To verify the correctness and robustness of our implementation, we developed a set of invalid programs, each containing exactly one syntax or lexical error, designed to test specific error conditions. We systematically designed invalid programs to trigger different classes of errors:

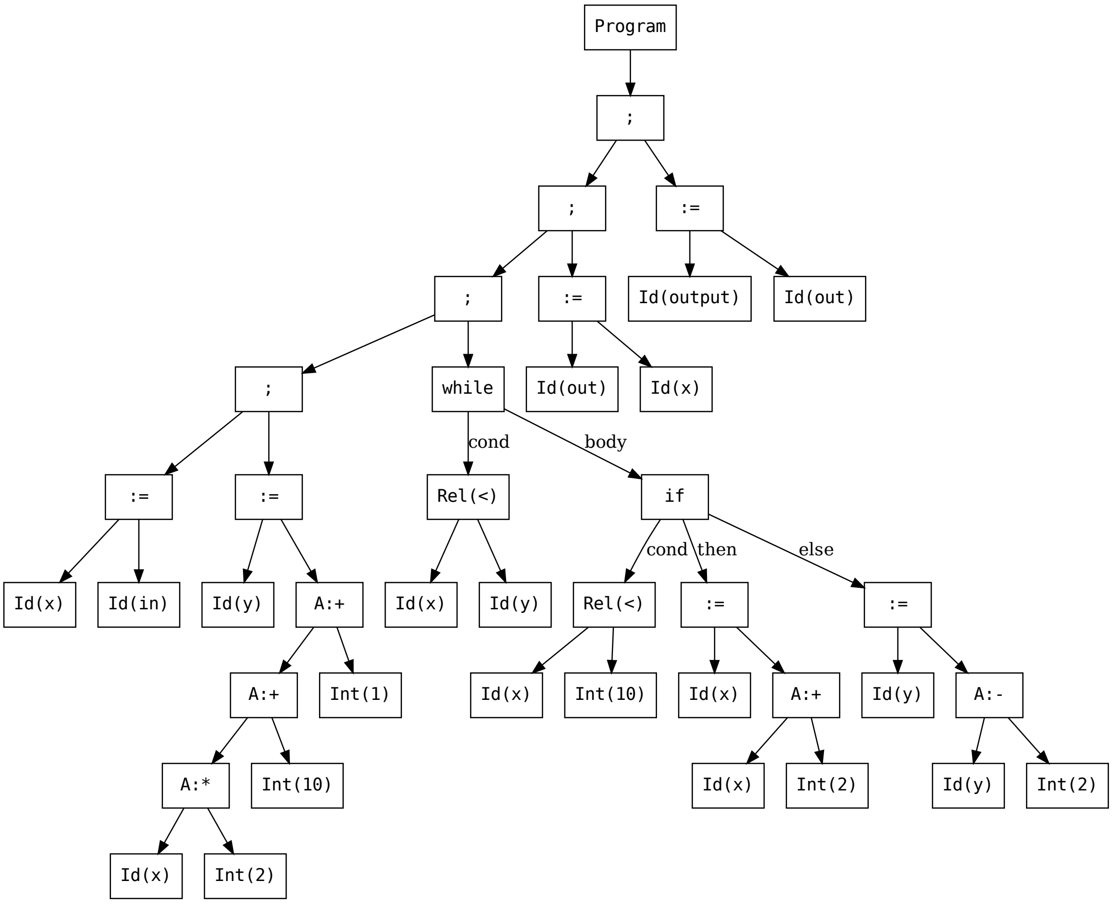
There are 28 test program developed to detect syntax errors, each invalid program triggered a meaningful syntax or lexical error message at the expected line. The parser correctly differentiates between various error types without crashing or misinterpretation, which confirms the correctness and robustness of our scanning and parsing stages.

Category	**Example**	**Expected Output**
Invalid Variable	`1var := 5;`	PARSE ERROR (line 4): expected SC (ID/if/while/skip) (lexeme='1')
Unclosed Comment	`{- comment not closed`	LEXICAL ERROR (line 1): comment needs closing bracket "-}"
Missing Operand	`x := * 3;`	PARSE ERROR (line 5): expected Atom '(' or ID or NUM (lexeme='*')
Missing Punctuation	`x := 3 y := 4`	PARSE ERROR (line 6): expected token EOF, got ID (lexeme='y')
Missing Keyword	`while [x < y] do x:=x + 1`	PARSE ERROR (line 7): expected token OD, got EOF (lexeme='')
Missing Parenthesis	`x := (3 + 4;`	PARSE ERROR (line 2): expected token RPAR, got SEMI (lexeme=';')
Empty Program	`nothing`	ERROR (input file has no contents)
Invalid Assignment	`x := 3.2;`	PARSE ERROR (line 1): expected token EOF, got INVALID (lexeme='.')
Infinite loop	`while 1=0 do n:=1 od`	PASS without any output

Table 1: Test Files with Expected Output

4 Task 4

This is the AST of Task2 program:



5 Task 5

This task implements the **code generation phase** of our WHILE compiler. The goal was to translate the abstract syntax tree (AST) into RISC-V assembly and produce a standalone executable by linking it with a generated `main.c` harness.

5.1 Implementation Overview

The code generator performs a recursive traversal of the AST, emitting RISC-V instructions for each construct in the language. Each variable is assigned a unique index through the `SymbolTable`, ensuring consistent addressing for load and store operations. The helper function `emitAddrOf()` computes the base address of a variable and offsets it by 64 bits:

```
li   t2, <slot>
slli t2, t2, 3
add  t2, a0, t2
```

Arithmetic and boolean expressions are compiled using registers `t0` and `t1`. The generator handles all binary operations defined in the grammar:

```
if (ab->op == "+")  out << "  add  t0, t1, t0\n";
else if (ab->op == "-") out << "  sub  t0, t1, t0\n";
else if (ab->op == "*") out << "  mul  t0, t1, t0\n";
```

and relational comparisons such as <, <=, >, and >= using RISC-V's `slt`, `xori`, and `seqz` instructions.

Control flow constructs are lowered into assembly through label based branching. Each `if` and `while` node generates a fresh set of labels:

```
beqz t0, L_else
...
j    L_end
L_else:
...
L_end:
```

These blocks ensure correct execution ordering for nested or sequential statements.

5.2 Program Generation and Testing

The `generate_riscv()` routine emits the final assembly section and program entry point:

```
.text
.globl program
program:
    ...
ret
```

A companion C harness initializes the variable array, calls the compiled procedure, and prints initial and final variable states.

I implemented the complete code generator, including expression evaluation, branching, and symbol management, and wrote several harnesses for early testing. Once stable, Zhuo integrated the module into our automated `GTest` based testing pipeline. The resulting compiler can generate and execute correct RISC-V assembly for all WHILE programs on the department's RISC-V host machine.

The testing pipeline is shown in Fig 1 below.

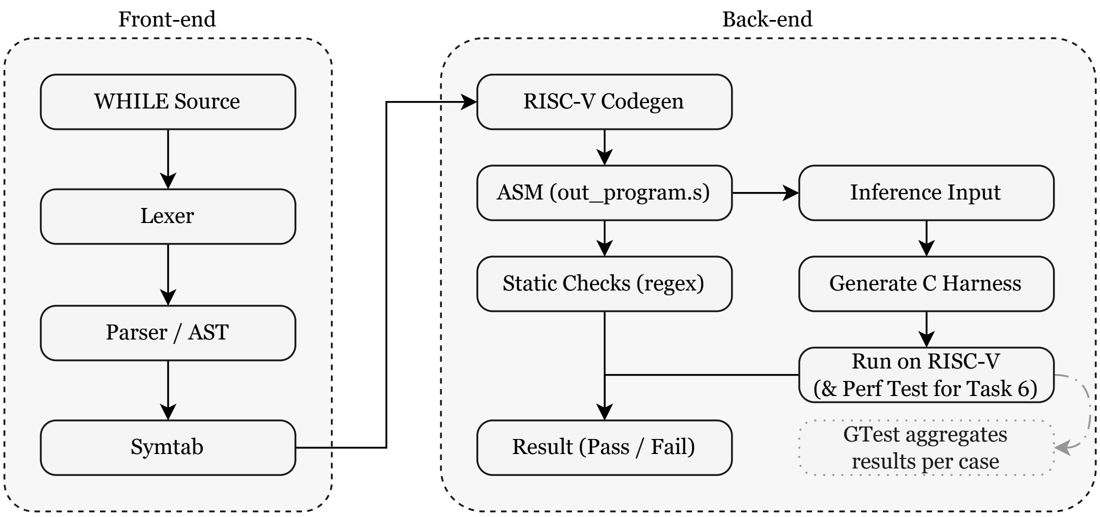


Figure 1: Testing pipeline of the compiler

5.3 Static Checks (Regex Assertions)

We assert 3 key structural patterns:

Address Computation: Use the sequence `li t2` → `slli t2, t2, 3` → `add a0, a0, t2`;

Memory Access: Use `ld`/`sd` with the computed base;

Control Flow: Emit conditional and unconditional branches when source has `if`/`while`.

Example Assertion for Address Computation:

```
std::vector<std::regex> addr_patterns = {
    std::regex(R"(\bli\s+t2,\s*\d+\b)"),         // li t2,<slot>
    std::regex(R"(\bslli\s+t2,\s*t2,\s*3\b)"),  // *8 for 64-bit
    std::regex(R"(\badd\s+t2,\s*(?:a0,\s*t2|t2,\s*a0)\b)")}; // base + offset

for (size_t i = 0; i < addr_patterns.size(); ++i)
{
    ASSERT_TRUE(std::regex_search(asm_text, addr_patterns[i]))
        << "Missing address calc pattern[" << i << "]";
}
```

5.4 Input Inference and Auto-Generated Harness

`analyze_inputs_from_asm` scans `out_program.s` for occurrences of `li t2,<slot>` and records the first memory operation on each slot. If a slot's first access is `ld` (read) rather than `sd` (write), than that slot is classified as a program input. Inputs are sorted by slot index and all non-input slots are initialized to 0 in the harness.

Then `write_harness_c_from_asm` generates a C harness that (1) declares `long int vars[]` and (2) initializes inputs from argv, (3) calls `program(vars)`, and (4) prints Initial/final states. The harness checks `argc` against the inferred input count and will report a usage message if there's a mismatch. We use the same harness both for correctness testing as well as for the performance evaluations in Task 6.

By inferring inputs, performance testing in task 6 would be simpler for the testers because they only provide the values for variables actually required by the test program.

6 Task 6

This task evaluates the runtime behavior of the generated executables on RISCV machine. The performance metric is based on the output of the `time` command.

6.1 Workloads and Results

The workload includes factorial by repeated addition, Euclidean GCD (mod & subtraction), prime couting with nested divisions and other tests provided on Canvas.

Table 2 summarizes the timing results on `risc-machine-2.cs.unm.edu`:

Workload	**Input**	**Wall-clock time**	**User CPU time**	**System CPU time**
Factorial (Addition)	12	0m2.960s	0m2.954s	0m0.004s
Factorial (Addition)	13	0m38.434s	0m38.426s	0m0.000s
GCD (Mod)	(102334155,102334156)	0m2.205s	0m0.745s	0m2.168s
GCD (Sub)	(1,100000000)	0m0.959s	0m0.949s	0m0.009s
Prime Count	10000	0m3.321s	0m3.309s	0m0.000s
Square (Nested loop)	100000	1m16.376s	1m16.347s	0m0.017s
collatz	77031	0m1.461s	0m1.456s	0m0.005s
madprime	524287	0m0.035s	0m0.026s	0m0.008s
sumfactors	1200	0m2.894s	0m2.888s	0m0.004s
countTriangularNumbers	(0,1000000)	0m11.726s	0m11.721s	0m0.005s

Table 2: Execution time for workloads on `risc-machine-2.cs.unm.edu`.

6.2 Thoughts and Possible Improvement

During the performance testing, we noticed a sharp jump when the input adds up to 13 for our "addition-based" factorial program. The reason is that we replaced the multiplication with repeated addition, and therefore, each outer iteration makes the next inner loop much longer than the previous one. By the time we move from `n=12` to `n=13`, the amount of work (branches, loads/stores, adds) grows roughly by 13 times, which is the current `n`.

This result gives us the thought that, if we can extend the code generator to recognize this "repeated addition" idiom, it can directly emit a multiply instruction. This could collapses thousands of additions into a single operation, and will dramatically improve the runtime.

7 Contributions

7.1 Stage 1

Task 1: All team member collaborated to discuss and define the grammar required for Task 1, including the formal BNF expressions for the `WHILE` language.

Task 2: Task 3 was implemented by Yue. Youssef wrote valid test `WHILE` programs and helped plan the scanner and debug the parser.

Task 3: Yue was responsible for the code for Lexer & Parser and writing. Zhuo was responsible for creating and testing `WHILE` programs containing syntax errors, ensuring that the scanner and parser correctly reported errors and ensuring the robustness of the program.

Task 4: Yue Hu

7.2 Stage 2

Task 5: Youssef Amin

Task 6: Zhuo Li

Task 7: The project report was revised and finalized by all team members.

The GitHub repository we worked together can be found here: Terminators.