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 program 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.

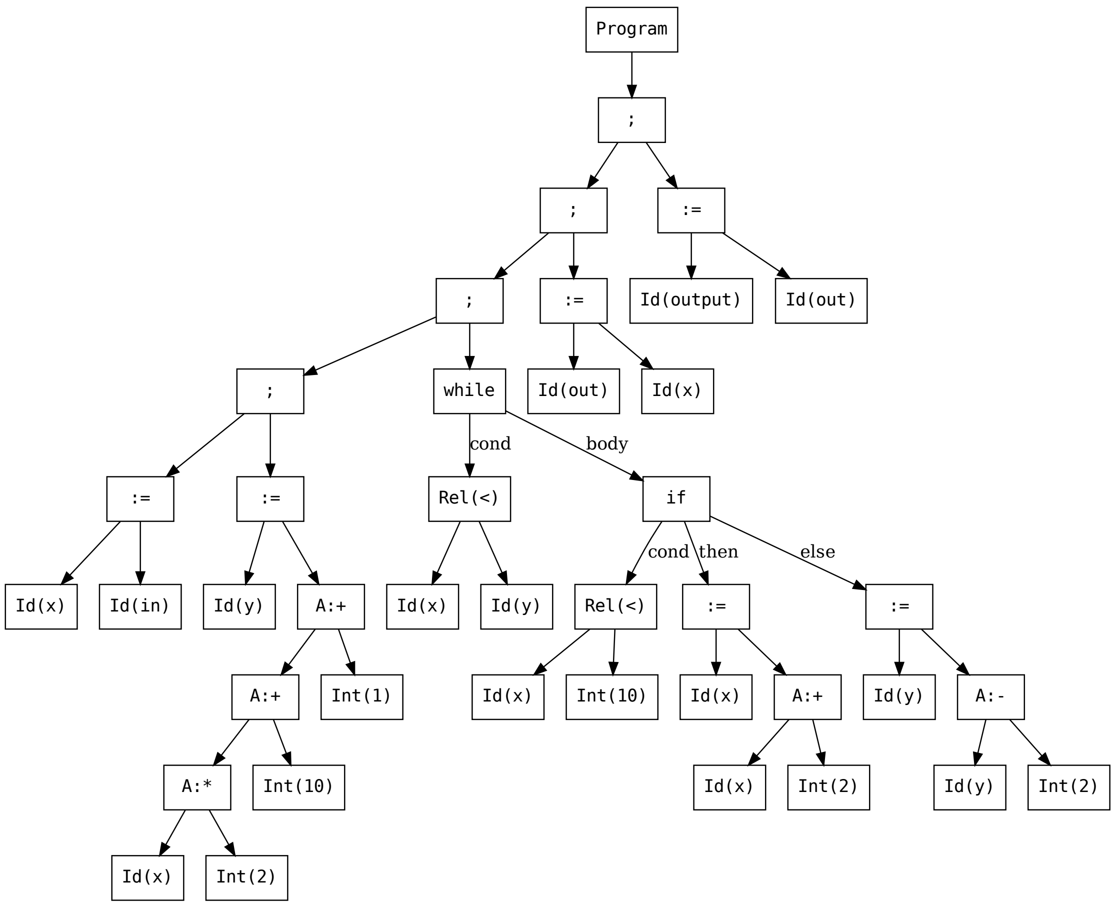
Parser and Abstract Syntax Tree (AST) The parser adopts a **top-down recursive descent** approach, following the grammar defined in Task 1.

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 Abstract Syntax Tree of Test File

The AST reflects operator precedence and associativity. At equal precedence, operations form a left-branching structure due to left associativity.



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 27 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 -}`	input file has no contents
Invalid Assignment	`x := 3.2;`	PARSE ERROR (line 1): expected token EOF, got INVALID (lexeme='.')

4 Contributions

4.1 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 two was implemented by Yue. Youssef wrote valid test `WHILE` programs and helped plan the scanner and debug the parser.

Task 3: 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.

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