Project #1: Scanning and Parsing

CSCI 70 - Structures and Interpretation of Programming Languages
SY 2025-2026
Lecturer: Mon Espinosa

Introduction

You have previously worked on the Deterministic Finite Automata (DFA) for the simple calculator programming language **SimpCalc** in your homework. While the DFA is not a direct component of this project, it is an important foundation for the actual implementation. This project consists of two main components: a scanner (lexical analyzer) and a parser (syntax analyzer). You are free to use **any programming language** of your choice from the list that we used for the Scanner Lab.

Project Specifications

Program Structure

SimpCalc programs adhere to the following rules:

1. **Identifier**: An identifier begins with a letter or an underscore, followed by letters, digits, or underscores. Identifiers are case-sensitive.
2. **Number**: Numeric literals can be integers, floating-point numbers, or expressed in exponential notation.
3. **String**: Strings are enclosed in double quotes (`"`) and may contain any printable character. However, they cannot span multiple lines.
4. **Comments**: Comments begin with `//` and ignore all characters until the end of the line.
5. **Assignment Statements**: Assignment statements take the form `Identifier := <Expression>;`.
6. **Expressions**: Expressions can include arithmetic operators such as `+, -, *, /, **` (for exponentiation) and unary negation (`-`). Parentheses can be used for grouping, and the `SQRT` function is available for square roots.
7. **Operator Precedence**: Operations are evaluated in the following order: parentheses, negation, exponentiation, multiplication/division, and addition/subtraction. Within each level, operations are processed from left to right.
8. **Print Statements**: A print statement is written as `PRINT(<Argument>, ...);` and may accept one or more arguments.
9. **If Statements**: A condition compares two expressions using a relational operator, following the format `<Expression> <RelationalOperator> <Expression>`. A block consists of one or more valid SimpCalc statements, which can include assignments, print statements, or further nested if-else statements. If statements can be nested and may optionally include an `ELSE` block.
   - `IF <Condition>: <Block> ENDIF;`
   - `IF <Condition>: <Block> ELSE <Block> ENDIF;`

Lexical Analysis

Lexical analysis scans the source code, filters out white spaces and comments, identifies lexical errors, and converts the code into a stream of tokens. For SimpCalc, possible tokens include:

1. `Identifier`
2. `Number`
3. `String`
4. `Assign`: `:=`
5. `Semicolon`: `;`
6. `Colon`: `:`
7. `Comma`: `,`
8. `LeftParen`: `(`
9. `RightParen`: `)`
10. `Plus`: `+`
11. `Minus`: `-`
12. `Multiply`: `*`
13. `Divide`: `/`
14. `Raise`: `**`
15. `LessThan`: `<`
16. `Equal`: `=`
17. `GreaterThan`: `>`
18. `LTEqual`: `<=`
19. `GTEqual`: `>=`
20. `NotEqual`: `!=`
21. `EndofFile`

Additionally, it has the following case-sensitive keywords: `PRINT`, `IF`, `ELSE`, `ENDIF`, `SQRT`, `AND`, `OR`, `NOT`. These are treated as separate tokens.

Syntax Analysis

Syntax analysis ensures that the source code adheres to the correct grammatical structure. It verifies whether the token stream produced during lexical analysis is organized according to the rules of the language, reflecting the hierarchical structure of the program. The following grammar can be used to create recursive subroutines. Note that the curly braces indicate the valid set of all tokens that could initiate the corresponding production. If the token being read is not in any of the valid sets, use the production that does not precede a set of valid tokens (i.e. if **Blk** receives a String token, it performs **Blk** $\rightarrow \epsilon$.). If this fallback production is not available, like in the case of **Stm** and **Rel**, this will lead into an error, as detailed later.

Prg	→ Blk EndOfFile	
Blk	→ Stm Blk	{Identifier, PRINT, IF}
	→ ϵ	
Stm	→ Identifier := Exp ;	{Identifier}
	→ PRINT(Arg Argfollow) ;	{PRINT}
	→ IF Cnd : Blk Iffollow	{IF}
Argfollow	→, Arg Argfollow	{Comma}
	→ ϵ	
Arg	→ String	{String}
	→ Exp	
Iffollow	→ ENDIF ;	{ENDIF}
	→ ELSE Blk ENDIF ;	{ELSE}
Exp	→ Trm Trmfollow	
Trmfollow	→ + Trm Trmfollow	{Plus}
	→ − Trm Trmfollow	{Minus}
	→ ϵ	
Trm	→ Fac Facfollow	
Facfollow	→ * Fac Facfollow	{Multiply}
	→ / Fac Facfollow	{Divide}
	→ ϵ	
Fac	→ Lit Litfollow	
Litfollow	→ ** Lit Litfollow	{Raise}
	→ ϵ	
Lit	→ −Val	{Minus}
	→ Val	
Val	→ Identifier	{Identifier}
	→ number	{number}
	→ SQRT(Exp)	{SQRT}
	→ (Exp)	
Cnd	→ Exp Rel Exp	
Rel	→<	{LessThan}
	→=	{Equal}
	→>	{GreaterThan}
	→<=	{GTEqual}
	→! =	{NotEqual}
	→>=	{LTEqual}

As discussed in class, the productions in the grammar can be interpreted as procedures, with each non-terminal corresponding to the name of a procedure body. Execution begins with the procedure `Prg`, and the recursive calls from these procedures effectively trace out a traversal of the parse tree for an input program. Instead of constructing a parse tree, your parser program should print statements to verify the validity of the sequence of tokens.

- Print `Assignment Statement Recognized` when the procedure corresponding to `Stm → Identifier := Exp ;` is successful.
- Print `Print Statement Recognized` when the procedure corresponding to `Stm → PRINT(Arg Argfollow);` is successful.
- Print `If Statement Begins` at the start of the procedure corresponding to `Stm → IF Cnd : Blk Iffollow`.
- Print `If Statement Ends` at the end of the procedure corresponding to `Stm → IF Cnd : Blk Iffollow`.
- Print `Invalid Statement` if the `Stm` procedure fails.
- Print `Incomplete if Statement` if there is an error in the procedure corresponding to `Iffollow → ELSE Blk ENDIF ;`.
- Print `Missing relational operator` if there is an error in the procedure corresponding to the relational productions.
- Remember, there is a `match` function for terminal symbols in the productions which verifies that the token encountered is the one that was expected, as specified by the argument. Print `Symbol expected` if the token does not match.

SimpCalc Example

Below is a sample program written in the SimpCalc language:

```
// This program calculates the roots of the following quadratic equation:
// 5.5x^2 + 10x - 3
discriminant := 10**2 - (4*5.5*(-3));
IF discriminant >= 0:
    root1 := (-10 + SQRT(discriminant))/(2*5.5);
    root2 := (-10 - SQRT(discriminant))/(2*5.5);
    PRINT("roots are", root1, root2);
ELSE // discriminant is negative
    PRINT("no real roots");
ENDIF;
PRINT("end of program");
```

Sample Files and Expected Output

Sample files will be posted on Canvas. The expected output for the Scanner Tester will be a list of recognized tokens together with their corresponding lexemes. The text below shows the first few lines expected for the sample program shown at the beginning of this document:

```
Identifier         discriminant
Assign             :=
Number             10
Raise              **
Number             2
Minus              -
LeftParen          (
Number             4
Multiply           *
Number             5.5
Multiply           *
```

```
LeftParen         (
Minus             -
Number            3
RightParen        )
RightParen        )
Semicolon         ;
If                IF
Identifier        discriminant
GTEqual           >=
Number            0
Colon             :
Identifier        root1
Assign            :=
LeftParen         (
Minus             -
```

The Parser Tester will output indications that valid statements were recognized, and whether the source code is a valid SimpCalc program. For the sample program, the output is as follows:

```
Assignment Statement Recognized
If Statement Begins
Assignment Statement Recognized
Assignment Statement Recognized
Print Statement Recognized
Print Statement Recognized
If Statement Ends
Print Statement Recognized
sample1.txt is a valid SimpCalc program
```

Project Deliverables

1. **Scanner Module**: The scanner must recognize all SimpCalc tokens and pass them to a separate module one token at a time. If the input file is named `sample_input.txt`, then the output file for the scanner should be `sample_output_scan.txt`. This module should provide the following functions:

   - `gettoken()`: Returns the next token from the input.

   Note: If there are multiple input files present in the same directory, then the program must scan and parse all of the files.

2. **Parser Module**: You will implement a **recursive descent parser** based on the provided context-free grammar (CFG) for the SimpleCalc language. If the input file is named `sample_input.txt`, the output file for the parser should be `sample_output_parse.txt`. The parser must recognize all valid SimpCalc statements, including assignment, print, and if-else constructs.

   Note: The parser must take the tokens produced by the scanner module as its input. Ensure both modules are properly integrated. For comparison purposes, you may use the text files that are posted along with this file.

Submission Guidelines

Submit a single `zip` file containing the following:

- **Scanner and Parser Source Code**: Implement the scanner and parser in your chosen programming language.

- **Documentation**: Include comments and documentation explaining the functionality of your code.
- **Certificate of Authorship**: If working in a group (up to three members), the zip file should be named as `surname1_surname2_surname3_languageused_CSCI70_SCANNER_PARSER.zip`.
- **Instruction file**: Include a .txt file explaining how to run the program
- **Deadline**: Refer to the Canvas page for the deadline.

Grading Breakdown

- **Scanner (50%)**: Accurate recognition of tokens and proper implementation of the `gettoken()` and `getlinenum()` functions.
- **Parser (50%)**: Correct identification of valid statements and appropriate error handling for the SimpCalc language.
- **Bonus (+10%)**: (Optional) Whenever an error occurs, also include which line it is from, like you would see in an actual error message. This nets you 5% for the Scanner and 5% for the Parser.

Important Notes

- Since you have already submitted a DFA in a separate assignment, the DFA is no longer part of the project deliverables, but understanding DFA will assist with token recognition.
- You are free to use any programming language of your choice, as long as it is in the same list of allowed languages for the Scanner Lab.
- Ensure that your project runs correctly before submission.

Good luck.