CSc 453 — Assignment 3

Christian Collberg
University of Arizona, Department of Computer Science

February 11, 2026

This assignment has two parts. In the first part you will answer some theory questions, similar to those you saw in the in-class exercises. In the second part you will write a parser for Luca.

1 Theory Questions

Each question is worth 4 points.

Some of the problems require you to submit your answer in the form of a Graphviz file. You can learn about Graphviz by watching lecture **misc-1** in `https://ligerlabs.org/lectures.html`.

1.1 Problem 1

Starting with this ambiguous grammar:

```
E ::= E + E |
      E * E |
      E ^ E |
      ( E ) |
      number
```

Transform the grammar so that the following holds:

- "^", exponentiation, highest precedence, right associative
- "*", multiplication, medium precedence, left associative
- "+", addition, lowest precedence, left associative

Enter your answer in the file `problem1.json`.

1.2 Problem 2

Transform the grammar from Problem 1 so that all left recursion has been removed.

Enter your answer in the file `problem2.json`.

1.3 Problem 3

Compute FIRST and FOLLOW for all non-terminals:

```
S -> a S b | X
X -> c X b | b
X -> b  X Z
Z -> n
```

Enter your answer in the file `problem3.json`.

1.4 Problem 4

To implement the recursive descent Luca parser you will have to rewrite the grammar provided for you so that

1. it does not use any EBNF extensions
2. it is unambiguous
3. it does not have any left-recursive rules.

Enter your rewritten grammar in the file `problem4.json`.

1.5 Problem 5

Compute FIRST and FOLLOW for the non-terminals in the rewritten Luca grammar from problem 4.

Enter your answer in the file `problem5.json`.

2 A parser analyzer for Luca

Your task is to write a syntactic analyzer for the language LUCA. Your program should be named `luca_parse`. `luca_parse`

1. reads a source program,
2. uses your lexer from the last assignment,
3. parses the source using a recursive descent parser,
4. writes syntactic error messages,
5. and produces a trace of its actions.

```
$ luca_parse prog1.luc > prog1.luc.out
```

The next assignment will extend `luca_parse`'s parser to produce an Abstract Syntax Tree (AST).

Appendix A gives the complete concrete syntax of LUCA.

The output is a trace of the actions of the recursive descent parser in XML format:

```
<PROGRAM token="PROGRAM" line="1">
   <MATCH token="PROGRAM" line="1"/>
   <MATCH token="IDENT" line="1"/>
   <MATCH token="SEMICOLON" line="1"/>
   <decls token="BEGIN" line="2">
   </decls>
   <block token="BEGIN" line="2">
      <MATCH token="BEGIN" line="2"/>
      <stats token="WRITE" line="3">
         <write token="WRITE" line="3">
            <MATCH token="WRITE" line="3"/>
            <expr token="IDENT" line="3">
               <term token="IDENT" line="3">
                  <logical token="IDENT" line="3">
                     <relational token="IDENT" line="3">
                        <factor token="IDENT" line="3">
                           <designator token="IDENT" line="3">
                              <MATCH token="IDENT" line="3"/>
                              <designator1 token="PLUS" line="3">
                              </designator1>
                           </designator>
                        </factor>
                     </relational>
                  </logical>
               </term>
               <MATCH token="PLUS" line="3"/>
               <term token="IDENT" line="3">
                  <logical token="IDENT" line="3">
                     <relational token="IDENT" line="3">
                        <factor token="IDENT" line="3">
                           <designator token="IDENT" line="3">
                              <MATCH token="IDENT" line="3"/>
                              <designator1 token="SEMICOLON" line="3">
                              </designator1>
                           </designator>
                        </factor>
                     </relational>
                  </logical>
               </term>
            </expr>
         </write>
         <MATCH token="SEMICOLON" line="3"/>
         <stats token="END" line="4">
         </stats>
      </stats>
      <MATCH token="END" line="4"/>
   </block>
   <MATCH token="PERIOD" line="4"/>
   <MATCH token="EOF" line="5"/>
</PROGRAM>
```

2.1 Getting Started

This assignment can be done in teams of one or two.

To get started, follow these instructions:

1. Unzip `assignment-3.zip` which gives you these files:

```
assignment-3/assignment-3.pdf
assignment-3/validator.sh
assignment-3/firstname_lastname/lexer/TokenSet.java
assignment-3/firstname_lastname/parser/Parser.java
assignment-3/firstname_lastname/tests/*.luc
```

```
assignment-3/firstname_lastname/tests/*.xml
assignment-3/firstname_lastname/problem.json
assignment-3/firstname_lastname/collaboration.json
assignment-3/firstname_lastname/survey.json
```

2. Then, assuming your name is Pat Jones, execute these commands:

   ```
   > cd assignment-3
   > mv firstname_lastname pat_jones
   ```

   If you are working in a group of two, pick one of your names.

3. Now, go ahead and solve the problems, entering your answers in the template files in `pat_jones/*`. Your main work should be modifying the Java source files in `assignment-3/pat_jones/parser`.

4. Skeleton Java code has been provided for you in `assignment-3/firstname_lastname/parser/*.java`.

5. You should copy your code from the previous assignment into `assignment-3/firstname_lastname/lexer/`.

Don't show your code to anyone outside your team, don't read anyone else's code, don't discuss the details of your code with anyone. If you need help with the assignment see the TA or the instructor.

2.2 Grading

11 error test cases have been provided for you in `assignment-3/firstname_lastname/tests/`. For each test case that produces an appropriate error message you will get 3 points.

There are an additional 58 test cases in `assignment-3/firstname_lastname/tests/`. For each case that does not produce an error or crashes, but produces a good trace, you will get 0.5 points.

You can earn an additional 18 points from more complicated secret test cases.

2.3 Output Format

Use these methods from `Parse.java` to produce a parse trace:

```
void ENTER(String e) {}
void EXIT(String e) { }
void MATCH() {}
```

Since you may rewrite your grammar so that it is different from mine, we don't expect your trace to be exactly the same as mine. However, the trace should clearly indicate that you are doing a recursive descent parse! Also, the trace will be invaluable for you when you debug your parser.

3 Luca Syntactic Errors

Instead of printing error messages in human readable form, `lucac` generates errors in an XML format. There's really only one message:

```
<SYNTAX_ERROR pos="3" expected="LIST_OF_THE_EXPECTED_TOKENS" found="TOKEN_FOUND"/>
```

Here are some examples:

```
<SYNTAX_ERROR pos="3" expected="]" found=","/>
<SYNTAX_ERROR pos="2" expected="(,-,FLOAT,NOT,TRUNC,char,identifier,integer,real,string" found="VAR"/>
```

The list of expected tokens are printed in lexicographically sorted order!

You don't have to do any error recovery. We won't test any of your output that appears after the first syntactic error message.

Line numbers are allowed to be "approximately" correct, i.e. they should be ± 1 from the "correct" line.

4 Implementation Notes

- This assignment shall be coded in Java.
- Make sure that your `Makefile` is working properly.
- **You cannot use `yacc` or any other similar parser generator for this assignment, either directly or indirectly.** I expect you to construct the grammar by hand, compute `FIRST` sets by hand, and implement the parser by hand.
- **You should build your parser the way we've discussed in class, using a recursive descent technique.** If you try anything else you will get 0 points.
- Your program should follow normal software design practices. In other words, it should be well structured using reasonable identifiers, comments, etc.
- **You cannot use any AI resources for this assignment.**

5 Submission

To submit, follow these instructions:

1. Verify that your submission contains a working `Makefile` and *all* the files necessary to build the **lexer and parser**. Your program must compile out of the box. The graders will *not* try to debug your program or your makefile for you.
2. Fill out `collaboration.json` with information about your group.
3. Optionally fill out `survey.json` to give me feedback about the assignment.
4. Package up your answers and verify that they look OK:

```
> zip -r pat_jones.zip pat_jones
> assignment-3/validator.sh pat_jones.zip
==== This is the LigerLabs assignment validator ====
          ...         ...         ...         ...
==== 6: Checking that all problems have been answered.
==== 7: Checking that all answer files have valid structure.
```

5. Upload `pat_jones.zip` to d2l. If you are working in a group of two, you should only submit once.

A Luca Syntax

Below is an EBNF grammar for the Luca language.

⟨*program*⟩ ::= '**PROGRAM**' ⟨*ident*⟩ ';' ⟨*decl_list*⟩ ⟨*block*⟩ '.'

⟨*block*⟩ ::= '**BEGIN**' ⟨*stat_seq*⟩ '**END**'

⟨*decl_list*⟩ ::= { ⟨*declaration*⟩ ';' }

⟨*declaration*⟩ ::= ‘**CONST**’ ⟨*ident*⟩ ‘:’ ⟨*ident*⟩ ‘=’ ⟨*expression*⟩ |
    ‘**VAR**’ ⟨*ident*⟩ ‘:’ ⟨*ident*⟩ |
    ‘**TYPE**’ ⟨*ident*⟩ ‘=’ ‘**ARRAY**’ ⟨*expression*⟩ ‘**OF**’ ⟨*ident*⟩ |
    ‘**TYPE**’ ⟨*ident*⟩ ‘=’ ‘**RECORD**’ ‘[’ [⟨*field_list*⟩] ‘]’ |
    ‘**PROCEDURE**’ ⟨*ident*⟩ ‘(’ [⟨*formal_list*⟩] ‘)’ ‘;’ ⟨*decl_list*⟩ ⟨*block*⟩

⟨*formal_list*⟩ ::= ⟨*formal_param*⟩ { ‘;’ ⟨*formal_param*⟩ }

⟨*field_list*⟩ ::= ⟨*field*⟩ { ‘;’ ⟨*field*⟩ }

⟨*formal_param*⟩ ::= [‘**VAR**’] ⟨*ident*⟩ ‘:’ ⟨*ident*⟩

⟨*field*⟩ ::= ⟨*ident*⟩ ‘:’ ⟨*ident*⟩

⟨*stat_seq*⟩ ::= { ⟨*statement*⟩ ‘;’ }

⟨*statement*⟩ ::= ⟨*designator*⟩ ‘:=’ ⟨*expression*⟩ |
    ⟨*designator*⟩ ‘(’ [⟨*actual_list*⟩] ‘)’ |
    ‘**IF**’ ⟨*expression*⟩ ‘**THEN**’ ⟨*stat_seq*⟩ ‘**ENDIF**’ |
    ‘**IF**’ ⟨*expression*⟩ ‘**THEN**’ ⟨*stat_seq*⟩ ‘**ELSE**’ ⟨*stat_seq*⟩ ‘**ENDIF**’ |
    ‘**WHILE**’ ⟨*expression*⟩ ‘**DO**’ ⟨*stat_seq*⟩ ‘**ENDDO**’ |
    ‘**REPEAT**’ ⟨*stat_seq*⟩ ‘**UNTIL**’ ⟨*expression*⟩|
    ‘**LOOP**’ ⟨*stat_seq*⟩ ‘**ENDLOOP**’ |
    ‘**EXIT**’ |
    ‘**WRITE**’ ⟨*expression*⟩ | ‘**WRITELN**’ |
    ‘**READ**’ ⟨*designator*⟩

⟨*actual_list*⟩ ::= ⟨*expression*⟩ { ‘,’ ⟨*expression*⟩ }

⟨*expression*⟩ ::= ⟨*expression*⟩ ⟨*bin_operator*⟩ ⟨*expression*⟩ |
    ⟨*unary_operator*⟩ ⟨*expression*⟩ |
    ‘(’ ⟨*expression*⟩ ‘)’ |
    ⟨*integer_literal*⟩ | ⟨*char_literal*⟩ | ⟨*real_literal*⟩ | ⟨*string_literal*⟩ | ⟨*designator*⟩

⟨*designator*⟩ ::= ⟨*ident*⟩ { ⟨*designator'*⟩ }

⟨*designator'*⟩ ::= ‘[’ ⟨*expression*⟩ ‘]’ | ‘.’ ⟨*ident*⟩

⟨*bin_operator*⟩ ::= ‘+’ | ‘−’ | ‘*’ | ‘/’ | ‘%’ | ‘**AND**’ | ‘**OR**’ | ‘<’ | ‘<=’ | ‘=’ |‘#’ | ‘>=’ |‘>’

⟨*unary_operator*⟩ ::= ‘−’ | ‘**NOT**’ | ‘**TRUNC**’ | ‘**FLOAT**’

This grammar is highly ambiguous. Here are the relevant operator precedence rules:

precedence	**operator**	**arity**	**associativity**
low	`+, -`	binary	left associative
	`*, /, %`	binary	left associative
	`AND,OR`	binary	left associative
	`<, <=, #, >, >=, =`	binary	left associative
high	`NOT, TRUNC, FLOAT,` $-_{\texttt{unary}}$	unary	right associative

B Optional Extensions

If you want to challenge yourself some more add one or more of these extensions. Submit extensive test cases for each extension and a `README` file that describes what you did. There is no extra credit, just the satisfaction of a job well done!

B.1 Grammar

Use this grammar instead:

```
⟨program⟩ ::=
    'PROGRAM' ⟨ident⟩ ';' ⟨decl_list⟩ ⟨block⟩ '.'
⟨decl_list⟩ ::=
    { ⟨declaration⟩ ';' }
⟨declaration⟩ ::=
    'VAR' ⟨ident⟩ ':' ⟨ident⟩ |
    'TYPE' ⟨ident⟩ '=' 'ARRAY' ⟨expression⟩ 'OF' ⟨ident⟩ |
    'TYPE' ⟨ident⟩ '=' 'RECORD' '[' { ⟨field_list⟩ } ']' |
    'TYPE' ⟨ident⟩ '=' 'REF' ⟨ident⟩ |
    'CONST' ⟨ident⟩ ':' ⟨ident⟩ '=' ⟨expression⟩ |
    'TYPE' ⟨ident⟩ '=' 'CLASS' ['EXTENDS' ⟨ident⟩] '[' ⟨field_list⟩ ']' '[' ⟨method_list⟩ ']'|
    'PROCEDURE' ⟨ident⟩ '(' [⟨formal_list⟩] ')' ⟨decl_list⟩ ⟨block⟩ ';'
⟨field_list⟩ ::= ⟨field⟩ { ';' ⟨field⟩ }
⟨field⟩ ::= ⟨ident⟩ ':' ⟨ident⟩ ';'
⟨method_list⟩ ::= ⟨method_decl⟩ [';' ⟨method_list⟩]
⟨method_decl⟩ ::= 'METHOD' ⟨ident⟩ '(' [⟨formal_list⟩] ')' ';' ⟨decl_list⟩ ⟨block⟩
⟨formal_list⟩ ::=
    ⟨formal_param⟩ { ';' ⟨formal_param⟩ }
⟨formal_param⟩ ::=
    ['VAR'] ⟨ident⟩ ':' ⟨ident⟩
⟨actual_list⟩ ::=
    ⟨expression⟩ { ',' ⟨expression⟩ }
⟨block⟩ ::=
    'BEGIN' ⟨stat_seq⟩ 'END'
⟨stat_seq⟩ ::=
    { ⟨statement⟩ ';' }
⟨statement⟩ ::=
    ⟨designator⟩ ':=' ⟨expression⟩ |
    'WRITE' ⟨expression⟩ |
    'WRITELN' |
    ⟨designator⟩ '(' [ ⟨actual_list⟩ ] ')'
    'IF' ⟨expression⟩ 'THEN' ⟨stat_seq⟩ 'ENDIF'|
    'IF' ⟨expression⟩ 'THEN' ⟨stat_seq⟩ 'ELSE' ⟨stat_seq⟩ 'ENDIF' |
    'WHILE' ⟨expression⟩ 'DO' ⟨stat_seq⟩ 'ENDDO' |
    'REPEAT' ⟨stat_seq⟩ 'UNTIL' ⟨expression⟩ |
    'LOOP' ⟨stat_seq⟩ 'ENDLOOP' |
    'EXIT' |
    'READ' ⟨designator⟩
⟨expression⟩ ::=
    ⟨expression⟩ ⟨bin_operator⟩ ⟨expression⟩ |
    '(' ⟨expression⟩ ')' |
    ⟨unary_operator⟩ ⟨expression⟩ |
    ⟨real_literal⟩ |
    ⟨integer_literal⟩ |
    ⟨char_literal⟩ |
    ⟨designator⟩ | ⟨string_literal⟩
⟨designator⟩ ::=
    ⟨ident⟩ |
```

⟨*designator*⟩ ‘[’ ⟨*expression*⟩ ‘]’ |
⟨*designator*⟩ ’.’ ⟨*ident*⟩ |
⟨*designator*⟩ ‘@’ ⟨*ident*⟩ |
⟨*designator*⟩ ‘‘’ ⟨*ident*⟩ |
‘^’ ⟨*designator’*⟩

⟨*unary_operator*⟩ ::=
‘−’ | ‘**TRUNC**’ | ‘**FLOAT**’| ‘**NOT**’ | ‘**NEW**’ ⟨*ident*⟩

⟨*bin_operator*⟩ ::=
‘+’ | ‘−’ | ‘∗’ | ‘/’ | ‘%’ | ‘**ISA**’ | ‘**NARROW**’ | ‘<’ | ‘<=’ | ‘=’ | ‘#’ | ‘>=’ |‘>’ |‘`AND`’ | ‘`OR`’

precedence	**operator**	**arity**	**associativity**
low	`+, -`	binary	left associative
	`*, /, %`	binary	left associative
	`AND,OR`	binary	left associative
	`<, <=, #, >, >=, =`	binary	left associative
	`ISA, NARROW`	binary	left associative
high	`NOT, TRUNC, FLOAT,` $-_{\texttt{unary}}$	unary	right associative

B.2 Array references

Allow array references of the form `A[1,2,3]`. Generate the same abstract syntax as you would for the equivalent designator `A[1][2][3]`.

B.3 Error-recovery

Implement a fancy error-recovery scheme. You should not terminate execution on the first error but rather recover from the error, continue parsing, and possibly emit more errors. At the very least, implement this for the expression part of the grammar.