1 Introduction

Your task is to write a syntactic analyzer for the language LUCA. Your program should be named luca_parse. luca_parse reads a source program, writes syntactic error messages or (if there are no errors) an abstract syntax tree (in XML format) on standard output.

$ luca_parse expr1.luc > expr1.luc.out

The result is a tree in XML format:

```
<block>
  <PROGRAM name="P" pos="1">
    <DECLNULL pos="2"/>
    <STATS pos="3">
      <WRITE pos="3">
        <BINARY op="PLUS" pos="3">
          <VARREF ident="a" pos="3"/>
          <DESNULL pos="3"/>
        </VARREF>
        <VARREF ident="b" pos="3"/>
        <DESNULL pos="3"/>
      </BINARY>
    </WRITE>
    <STATNULL pos="4"/>
  </STATS>
  </PROGRAM>
</block>
```

For syntactically incorrect LUCA programs no abstract syntax tree is produced, just an error message.

Appendix A gives the complete syntax of LUCA. Appendix B gives the complete abstract syntax you should produce.

2 LUCA Syntactic Errors

Instead of printing error messages in human readable form, luca_parse generates errors in an XML format:

```
<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.

3 Implementation Notes

• This assignment can be coded in Java, C, or C++. If you want to use another language, ask me first.

• Make sure that your Makefile is working properly. The TA will do the following, and nothing else:

   $ make
   $ luca_parse test1.luc > test1.luc.out

   In other words, if you’re coding in Java you must provide a shell script called luca_parse that calls Java with the appropriate parameters.

• 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.

• See Sections 4.1.2 and 4.2.4 of the text-book for a description on how to adapt top-down parsers to build abstract syntax trees.

4 Submission and Assessment

• The deadline for this assignment is Fri Oct 2, 23.59. It is worth 10% of your final grade.

• You should submit the assignment electronically to d2l.arizona.edu.

• You can work alone or in teams of 2. You must submit a README file that lists the members of your team and how much each team member contributed to the assignment.

• If you work in a team you should only submit one copy of the assignment.

• You can download 58 syntactically corret test cases from the class website: http://www.cs.arizona.edu/~collberg/Teaching/453/2009/Assignments/index.html. Each will give you one point if you get it right and 0 points if you get it wrong. No partial credits. We won’t check for the correctness of line numbers.

• You can download an additional 11 test cases that should generate an error. You get 2 points if you get it right and 0 points if you get it wrong. No partial credits. We won’t check for the correctness of line numbers. We will ignore anything following the first error message.

• You can see some of the test cases in Appendix C. You can get an additional 20 points from more complicated “secret” test cases, for a total of 100 points.

• A large number of Java classes for building abstract syntax tress have been provided for you. Use them or not, it’s up to you.
• Your electronic submission must contain a working Makefile, and all the files necessary to build the lexer and parser. If your program does not compile “out of the box you will receive zero (0) points. The TA will not try to debug your program or your makefile for you!

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.

A Luca Syntax

\[ \langle \text{program} \rangle ::= \text{`PROGRAM'} \langle \text{ident} \rangle \text{`;'} \langle \text{decl_list} \rangle \langle \text{block} \rangle \text{`;'} \]

\[ \langle \text{block} \rangle ::= \text{`BEGIN'} \langle \text{stat_seq} \rangle \text{`END'} \]

\[ \langle \text{decl_list} \rangle ::= \{ \langle \text{declaration} \rangle \text{`;'} \} \]

\[ \langle \text{declaration} \rangle ::= \text{`CONST'} \langle \text{ident} \rangle \text{`;'} \langle \text{ident} \rangle \text{`='} \langle \text{expression} \rangle | \]

\[ \text{`VAR'} \langle \text{ident} \rangle \text{`;'} \langle \text{ident} \rangle \]

\[ \text{`TYPE'} \langle \text{ident} \rangle \text{`='} \text{`ARRAY'} \langle \text{expression} \rangle \text{`OF'} \langle \text{ident} \rangle | \]

\[ \text{`TYPE'} \langle \text{ident} \rangle \text{`=} \text{`RECORD'} [ \langle \text{field_list} \rangle ] \text{'} | \]

\[ \text{`PROCEDURE'} \langle \text{ident} \rangle \text{'} [ \langle \text{formal_list} \rangle ] \text{'} \text{`;'} \langle \text{decl_list} \rangle \langle \text{block} \rangle \]

\[ \langle \text{formal_list} \rangle ::= \langle \text{formal} \rangle \{ \langle \text{formal} \rangle \} \]

\[ \langle \text{field_list} \rangle ::= \langle \text{field} \rangle \{ \langle \text{field} \rangle \} \]

\[ \langle \text{formal} \rangle ::= [ \text{`VAR'} \langle \text{ident} \rangle \text{`;'} \langle \text{ident} \rangle \]

\[ \langle \text{stat_seq} \rangle ::= \{ \langle \text{statement} \rangle \text{`;' } \} \]

\[ \langle \text{statement} \rangle ::= \langle \text{designator} \rangle \text{`;'} \langle \text{expression} \rangle | \]

\[ \langle \text{designator} \rangle \text{'} [ \langle \text{actual_list} \rangle ] \text{'} \]

\[ \text{`IF'} \langle \text{expression} \rangle \text{`THEN'} \langle \text{stat_seq} \rangle \text{`ENDIF'} | \]

\[ \text{`IF'} \langle \text{expression} \rangle \text{`THEN'} \langle \text{stat_seq} \rangle \text{`ELSE'} \langle \text{stat_seq} \rangle \text{`ENDIF'} | \]

\[ \text{`WHILE'} \langle \text{expression} \rangle \text{`DO'} \langle \text{stat_seq} \rangle \text{`ENDDO'} | \]

\[ \text{`REPEAT'} \langle \text{stat_seq} \rangle \text{`UNTIL'} \langle \text{expression} \rangle | \]

\[ \text{`LOOP'} \langle \text{stat_seq} \rangle \text{`ENDLOOP'} | \]

\[ \text{`EXIT'} | \]

\[ \text{`WRITE'} \langle \text{expression} \rangle | \text{`WRITELN'} | \]

\[ \text{`READ'} \langle \text{designator} \rangle \]

\[ \langle \text{actual_list} \rangle ::= \langle \text{expression} \rangle \{ \langle \text{expression} \rangle \}

\[ \langle \text{expression} \rangle ::= \langle \text{expression} \rangle \langle \text{bin_operator} \rangle \langle \text{expression} \rangle | \]

\[ \langle \text{unary_operator} \rangle \langle \text{expression} \rangle | \]

\[ \langle \text{integer_literal} \rangle | \langle \text{char_literal} \rangle | \langle \text{real_literal} \rangle | \langle \text{string_literal} \rangle | \langle \text{designator} \rangle \]

\[ \langle \text{designator} \rangle ::= \langle \text{ident} \rangle \{ \langle \text{designator} \rangle \}

\[ \langle \text{designator} \rangle ::= \langle \text{ident} \rangle \langle \text{expression} \rangle [ \langle \text{designator} \rangle \}

\[ \langle \text{bin_operator} \rangle ::= \langle \text{`+'\rangle | \langle \text{`-'\rangle | \langle \text{`*\rangle | \langle \text{`/\rangle | \langle \text{`\%angle | \langle \text{`AND'} | \langle \text{`OR'} | \langle \text{`<'\rangle | \langle \text{`<='\rangle | \langle \text{`='\rangle | \langle \text{`#'} | \langle \text{`>'\rangle | \langle \text{`>'\rangle \}

\[ \langle \text{unary_operator} \rangle ::= \langle \text{`-\rangle | \langle \text{`NOT'} | \langle \text{`TRUNC'} | \langle \text{`FLOAT'} \]

This grammar is highly ambiguous. Here are the relevant operator precedence rules:
<table>
<thead>
<tr>
<th>precedence</th>
<th>operator</th>
<th>arity</th>
<th>associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>low</td>
<td>+, -</td>
<td>binary</td>
<td>left associative</td>
</tr>
<tr>
<td></td>
<td>*, /, %</td>
<td>binary</td>
<td>left associative</td>
</tr>
<tr>
<td></td>
<td>AND, OR</td>
<td>binary</td>
<td>left associative</td>
</tr>
<tr>
<td></td>
<td>&lt;, &lt;=, #, &gt;, &gt;=, =</td>
<td>binary</td>
<td>left associative</td>
</tr>
<tr>
<td>high</td>
<td>NOT, TRUNC, FLOAT, −unary</td>
<td>unary</td>
<td>right associative</td>
</tr>
</tbody>
</table>
B  Abstract Syntax

Below are the nodes in the Luca abstract syntax tree, in the format they are generated by luca\_parse. Node-names are in \textbf{bold}, attributes are in a \textit{typewriter font}, and node references (subtrees) in \textit{italics}.

B.1  Declarations

\texttt{(PROGRAM \textit{Ident} \textit{Pos} \textit{Decls} \textit{Stats})}

This is the topmost node of any AST. \textit{Ident} is the name of the program, \textit{Pos} the line number where \textit{Ident} occurs. \textit{Decls} and \textit{Stats} are the sub-trees for declarations and statements, respectively. Here's an example:

\[
\text{PROGRAM P; BEGIN END}.
\]

\[
\texttt{<block>}
\texttt{<PROGRAM name="P" pos="1">}
\texttt{<DECLNULL pos="2"/>}
\texttt{<STATNULL pos="3"/>}
\texttt{<PROGRAM>}
\texttt{</block>}
\]

\texttt{(DECLS \textit{Pos} \textit{Left} \textit{Right})}

A list of declarations are linked together using \texttt{DECLS} nodes. \textit{Left} points to the actual declaration, \textit{Right} is the remaining subtree of declarations.

\texttt{(DECLNULL \textit{Pos})}

This node ends a sequence of declarations.

\texttt{(PROCDECL \textit{Ident} \textit{Pos} \textit{Formals} \textit{Decls} \textit{Stats})}

This is the topmost node of a procedure declaration. \textit{Formals} is a list of \texttt{FORMALDECL} nodes. \textit{Decls} and \textit{Stats} are the sub-trees for declarations and statements, respectively.

\texttt{(FORMALDECL \textit{Ident} \textit{TypeName} \textit{Mode} \textit{Pos})}

\textit{Ident} is the name of a formal parameter, \textit{TypeName} its type. \textit{Mode} is \texttt{VAL} or \texttt{VAR}. \texttt{FORMALDECL}s are linked together using \texttt{DECL} nodes to form the lists of formal declarations used in \texttt{PROCDECL}s.

\texttt{(VARDECL \textit{Ident} \textit{TypeName} \textit{Pos})}

\texttt{VARDECL}s are linked together using \texttt{DECL} nodes to form lists of variable declarations. \textit{Ident} is the name of the variable, \textit{TypeName} its type.

\texttt{(CONSTDECL \textit{Ident} \textit{TypeName} \textit{Pos} \textit{Expr})}

\textit{Ident} is the name of the constant, \textit{TypeName} its type, and \textit{Expr} the root of an expression tree.

\texttt{(ARRAYDECL \textit{Ident} \textit{ElementType} \textit{Pos} \textit{Count})}

\texttt{ARRAYDECL} nodes represent an array type declaration. \textit{Ident} is the name of the type, \textit{Count} is the root of an expression tree computing the number of elements in the array, and \textit{ElementType} the type of the elements of the array.

\texttt{(RECORDDECL \textit{Ident} \textit{Pos} \textit{Fields})}

\texttt{RECORDDECL} nodes represent a record type declaration. \textit{Ident} is the name of the type, and \textit{Fields} is a list of \texttt{FIELDDECL} nodes, linked together on \texttt{DECL}s.

\texttt{(FIELDDECL \textit{Ident} \textit{TypeName} \textit{Pos})}

\texttt{FIELDDECL} nodes represent a field in a record. They are linked together in a list of \texttt{DECL} nodes, terminated by a \texttt{DECLNULL}.
B.2 Statements

(STATS Pos Left Right)

A list of statements are linked together using STATS nodes. Left points to the actual statement, Right is the remaining subtree of statements. Example:

```xml
<block>
  <PROGRAM name="P" pos="1">
    <DECLNULL pos="2"/>
    <STATS pos="3">
      <WRITELN pos="3"/>
      <STATS pos="4">
        <WRITELN pos="4"/>
        <STATS pos="5">
          <WRITELN pos="5"/>
          <STATNULL pos="6"/>
        </STATS>
      </STATS>
    </STATS>
  </PROGRAM>
</block>
```

(STATNULL Pos)

This node ends a sequence of statements.

(PROCCALL Pos Des Actuals)

PROCCALL nodes represent a procedure call statement. Des is a designator representing the procedure to be called (always just one VARREF for this version of LUCA), and Actuals is a tree of ACTUAL nodes, the actual arguments to the call.

(ACTUAL Pos Expr Next)

A tree of ACTUAL nodes are used to represent the argument list to a procedure call. Expr is the root of an expression tree, Next refers to another ACTUAL node or ACTUALNULL.

(ACTUALNULL Pos)

This node ends a sequence of expressions.

(WRITE Pos Expr)

`Expr` is the root of an expression tree whose value is to be written.

(WRITELN Pos)

A node with no children and no input attributes other than Pos.

(READ Pos Des)

`Read` is the root of a designator list representing the variable into which the value is supposed to be read.

(IF1 Pos Expr Then)

`Expr` is the root of an expression tree. `Then` is the body of the “then” part of the statement.

(IF2 Pos Expr Then Else)

`Expr` is the root of an expression tree. `Then` and `Else` are the bodies of the “then” and “else” parts of the statement.

(WHILE Pos Expr Stats)

`Expr` is the root of the expression tree for the loop condition. `Stats` is the body of the loop.

(REPEAT Pos Expr Stats)

`Expr` is the root of the expression tree for the loop condition. `Stats` is the body of the loop.
(LOOP Pos Stats)  
Stats is the body of the loop.

(EXIT Pos)  
This node represents the EXIT statement which can only occur within the body of a LOOP statement.

(ASSIGN Pos Des Expr)  
Expr is the expression tree for the right hand side of the statement, Des represents the designator for the left and side.

B.3 Expressions and Designators

(INTLIT Value Pos)  
Value is a decimal integer literal.

(CHARLIT "Value" Pos)  
Value is a one-character string.

(STRINGLIT "Value" Pos)  
Value is a string.

(REALLIT Value Pos)  
Value is a floating-point number.

(VARREF Ident Pos Next)  
VARREFs are the root of a designator list. This list begins with a VARREF, and ends with a DESNULL. Inbetween are INDEX and FIELDREF nodes representing record field and array references. Example:  

```
PROGRAM P;
BEGIN
  READ a.b[c];
END.
```

(DESNULL Pos)  
The last node of a designator list.

(INDEX Index Next Pos)  
INDEX nodes represent an array index reference. Index is the root of an expression tree. Next is the next element of the designator list.
FIELDREF nodes represent a record field reference. Next is the next element of the designator list.

(BINARY \texttt{op} \texttt{Pos} \texttt{Left} \texttt{Right})

Left and Right are the expression trees for the left and right hand side of the operator. "\texttt{op}" is the operator.

(UNARY \texttt{op} \texttt{Right} \texttt{Right})

Right is the expression tree for the right hand side of the operator. "\texttt{op}" is the operator.
PROGRAM P;
BEGIN
  x := y;
END.
⇒
<block>
  <PROGRAM name="P" pos="1"/>
  <DECLNULL pos="2"/>
  <STATS pos="3">
    <ASSIGN pos="3"/>
      <VARREF ident="x" pos="3"/>
        <DESNUL pos="3"/>
      </VARREF>
      <VARREF ident="y" pos="3"/>
        <DESNUL pos="3"/>
      </VARREF>
    </ASSIGN>
    <STATNULL pos="4"/>
  </STATS>
</PROGRAM>
</block>

PROGRAM P;
BEGIN
  IF x THEN
    WRITELN;
  ENDIF;
END.
⇒
<block>
  <PROGRAM name="P" pos="1"/>
  <DECLNULL pos="2"/>
  <STATS pos="5">
    <IF1 pos="3">
      <VARREF ident="x" pos="3"/>
        <DESNUL pos="3"/>
      </VARREF>
      <STATS pos="4">
        <WRITELN pos="4"/>
          <STATNULL pos="5"/>
        </WRITELN>
      </STATS>
    </IF1>
    <STATNULL pos="6"/>
  </STATS>
</PROGRAM>
</block>

PROGRAM P;
TYPE x = ARRAY a OF T;
BEGIN
END.
⇒
<block>
  <PROGRAM name="P" pos="1"/>
  <DECLS pos="2">
    <ARRAYDECL ident="x" elementType="T" pos="2">
      <VARREF ident="a" pos="2"/>
        <DESNUL pos="2"/>
      </VARREF>
    </ARRAYDECL>
    <DECLNULL pos="3"/>
  </DECLS>
  <STATNULL pos="4"/>
</PROGRAM>
</block>
PROGRAM P;
BEGIN
  WRITE a*b/c;
END.

PROGRAM P;
BEGIN
  x(y);
END.
PROGRAM P;
BEGIN
  READ a[3,4];
END.

PROGRAM P;
CONST a : b = VAR;
BEGIN
END.

PROGRAM P;
BEGIN
END.

PROGRAM P;
PROCEDURE x();
  IF x THEN WRITELN;
END;
BEGIN
END.

PROGRAM P;
BEGIN
  IF x THEN WRITELN;
END.

PROGRAM P;
BEGIN
  WRITE a + ;
END.

PROGRAM P;
TYPE x = RECORD [];
BEGIN
END.

PROGRAM P;
BEGIN
  WRITE (TRUNC FLOAT) -42;
END.
These extensions are for the honor's section only. Submit extensive test cases for each extension.

D.1 Grammar

Use this grammar instead:

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

D.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.