University of Arizona, Department of Computer Science

CSc 453 — Assignment 2 — Due Fri Oct 2, 23.59 — 10%

Christian Collberg

September 26, 2009

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:

```
PROGRAM P;
BEGIN
   WRITE a+b;
END.
```

$\Longrightarrow$

```
<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"/>
            </VARREF>
         </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 <u>recursive descent</u> 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

⟨*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,` $-_{\mathtt{unary}}$	unary	right associative

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 **bold**, attributes are in a `typewriter font`, and node references (subtrees) in *italics*.

B.1 Declarations

(PROGRAM `Ident Pos` ***Decls Stats*****)**

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

```
PROGRAM P;
BEGIN
END.
```

$\Longrightarrow$

```
<block>
<PROGRAM name="P" pos="1">
   <DECLNULL pos="2"/>
   <STATNULL pos="3"/>
</PROGRAM>
</block>
```

(DECLS `Pos` ***Left Right*****)**

A list of declarations are linked together using **DECLS** nodes. *Left* points to the actual declaration, *Right* is the remaining subtree of declarations.

(DECLNULL `Pos`**)**

This node ends a sequence of declarations.

(PROCDECL `Ident Pos` ***Formals Decls Stats*****)**

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

(FORMALDECL `Ident TypeName Mode Pos`**)**

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

(VARDECL `Ident TypeName Pos`**)**

VARDECLs are linked together using **DECL** nodes to form lists of variable declarations. `Ident` is the name of the variable, `TypeName` its type.

(CONSTDECL `Ident TypeName Pos` ***Expr*****)**

`Ident` is the name of the constant, `TypeName` its type, and *Expr* the root of an expression tree.

(ARRAYDECL `Ident ElementType Pos` ***Count*****)**

ARRAYDECL nodes represent an array type declaration. `Ident` is the name of the type, *Count* is the root of an expression tree computing the number of elements in the array, and `ElementType` the type of the elements of the array.

(RECORDDECL `Ident Pos` ***Fields*****)**

RECORDDECL nodes represent a record type declaration. `Ident` is the name of the type, and *Fields* is a list of **FIELDDECL** nodes, linked together on **DECL**s.

(FIELDDECL `Ident TypeName Pos`**)**

FIELDDECL nodes represent a field in a record. They are linked together in a list of **DECL** nodes, terminated by a **DECLNULL**.

B.2 Statements

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

```
PROGRAM P;
BEGIN
   WRITELN;
   WRITELN;
   WRITELN;
END.
```

```
<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*****)**
: **VARREF**s 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.
```

$\Longrightarrow$

```
<block>
<PROGRAM name="P" pos="1">
   <DECLNULL pos="2"/>
   <STATS pos="3">
      <READ pos="3">
         <VARREF ident="a" pos="3">
            <FIELDREF ident="b" pos="3">
               <INDEX pos="3">
                  <VARREF ident="c" pos="3">
                     <DESNULL pos="3"/>
                  </VARREF>
                  <DESNULL pos="3"/>
               </INDEX>
            </FIELDREF>
         </VARREF>
      </READ>
      <STATNULL pos="4"/>
   </STATS>
</PROGRAM>
</block>
```

(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 `Ident` ***Next*** `Pos`**)**
FIELDREF nodes represent a record field reference. *Next* is the next element of the designator list.

(BINARY `Op` `Pos` ***Left Right*****)**
Left and *Right* are the expression trees for the left and right hand side of the operator. `"Op"` is the operator.

(UNARY `Op` ***Right Right*****)**
Right is the expression tree for the right hand side of the operator. `"Op"` is the operator.

C Examples

```
PROGRAM P;
BEGIN
   x := y;
END.
```

$\Longrightarrow$

```
<block>
<PROGRAM name="P" pos="1">
   <DECLNULL pos="2"/>
   <STATS pos="3">
      <ASSIGN pos="3">
         <VARREF ident="x" pos="3">
            <DESNULL pos="3"/>
         </VARREF>
         <VARREF ident="y" pos="3">
            <DESNULL pos="3"/>
         </VARREF>
      </ASSIGN>
      <STATNULL pos="4"/>
   </STATS>
</PROGRAM>
</block>
```

```
PROGRAM P;
BEGIN
   IF x THEN
      WRITELN;
   ENDIF;
END.
```

$\Longrightarrow$

```
<block>
<PROGRAM name="P" pos="1">
   <DECLNULL pos="2"/>
   <STATS pos="5">
      <IF1 pos="3">
         <VARREF ident="x" pos="3">
            <DESNULL pos="3"/>
         </VARREF>
         <STATS pos="4">
            <WRITELN pos="4"/>
            <STATNULL pos="5"/>
         </STATS>
      </IF1>
      <STATNULL pos="6"/>
   </STATS>
</PROGRAM>
</block>
```

```
PROGRAM P;
TYPE x = ARRAY a OF T;
BEGIN
END.
```

$\Longrightarrow$

```
<block>
<PROGRAM name="P" pos="1">
   <DECLS pos="2">
      <ARRAYDECL ident="x" elementTypeName="T" pos="2">
         <VARREF ident="a" pos="2">
            <DESNULL pos="2"/>
         </VARREF>
      </ARRAYDECL>
      <DECLNULL pos="3"/>
   </DECLS>
   <STATNULL pos="4"/>
</PROGRAM>
</block>
```

```
PROGRAM P;
BEGIN
   WRITE a*b/c;
END.
```

$\Longrightarrow$

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

```
PROGRAM P;
BEGIN
   x(y);
END.
```

$\Longrightarrow$

```
<block>
<PROGRAM name="P" pos="1">
   <DECLNULL pos="2"/>
   <STATS pos="3">
      <PROCCALL pos="3">
         <VARREF ident="x" pos="3">
            <DESNULL pos="3"/>
         </VARREF>
         <ACTUAL pos="3">
            <VARREF ident="y" pos="3">
               <DESNULL pos="3"/>
            </VARREF>
            <ACTUALNULL pos="3"/>
         </ACTUAL>
      </PROCCALL>
      <STATNULL pos="4"/>
   </STATS>
</PROGRAM>
</block>
```

```
PROGRAM P;
BEGIN
   READ a[3,4];
END.
```

⇒

```
<SYNTAX_ERROR pos="3"
   expected="]"
   found=","/>
```

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

⇒

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

```
PROGRAM P;
BEGIN
  ;
END.
```

⇒

```
<SYNTAX_ERROR pos="3"
   expected="END"
   found=";"/>
```

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

⇒

```
<SYNTAX_ERROR pos="3"
   expected="BEGIN"
   found="IF"/>
```

```
PROGRAM P;
BEGIN
   IF x THEN
      WRITELN;
END.
```

⇒

```
<SYNTAX_ERROR pos="5"
   expected="ENDIF"
   found="END"/>
```

```
PROGRAM P;
BEGIN
   WRITE a + ;
END.
```

⇒

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

```
PROGRAM P;
BEGIN
   WRITELN
   WRITELN;
END.
```

⇒

```
<SYNTAX_ERROR pos="4"
   expected=";"
   found="WRITELN"/>
```

```
PROGRAM P;
BEGIN
   WRITE (a + b;
END.
```

⇒

```
<SYNTAX_ERROR pos="3"
   expected=")"
   found=";"/>
```

```
PROGRAM P;
BEGIN
   WRITE ) ;
END.
```

⇒

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

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

⇒

```
<SYNTAX_ERROR pos="2"
   expected="ARRAY,RECORD"
   found="identifier"/>
```

```
PROGRAM P;
BEGIN
   WRITE (TRUNC FLOAT) -42;
END.
```

⇒

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

D For honor’s section only

These extensions are for the honor’s section only. Submit extensive test cases for each extension.

D.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

D.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]`.

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.