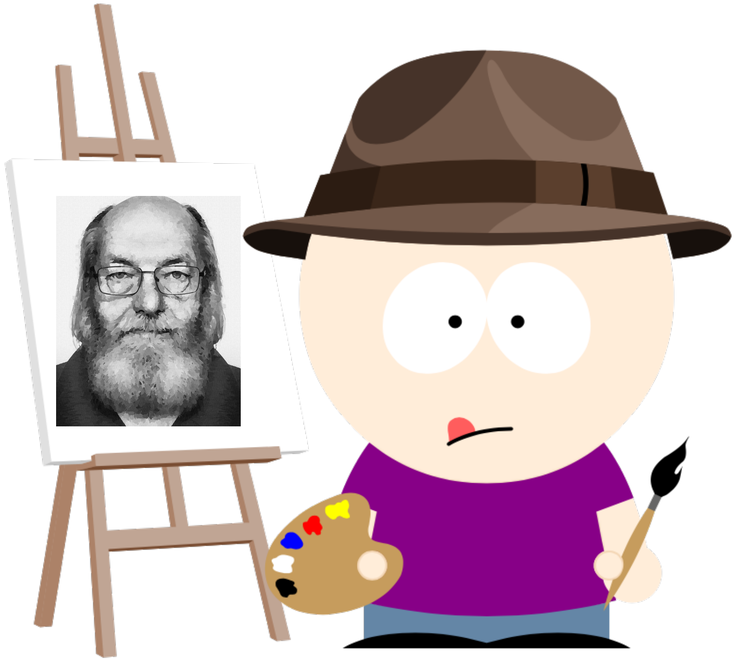


In-Class Exercises

Version: 2026-02-03

Formal Grammars

Download exercises

1. Go to

   https://ligerlabs.org/compilers.html

2. Download the file

   `grammars-1-exercises.zip`

3. Open up a terminal and Unzip the file

```
> unzip grammars-1-exercises.zip
> cd grammars-1-exercises
> ls
```

Task 1

- Given this grammar:

```
E ::= E + E
    | E - E
    | E * E
    | number
```

1. Show a leftmost and a rightmost derivation of the expression
   `"3 * 4 + 5 * 6"`
2. Show a leftmost and a rightmost derivation of the expression
   `"3 + 4 - 5"`

Task 2

- Consider the table of C operators below.

- The higher the precedence value, the tighter the operator binds. In other words, in C, a[k] (array indexing) binds the hardest and the comma operator (what does the comma operator do? Look it up!) has the lowest binding power.

- Based on the table, construct a C expression that a non-expert would find "unobvious".

- Switch with your partner and see if you were able to confuse them!

Task 3 (a)

- Consider the English grammar on the next page.
- Find a sentence (longer is better!) generated by this grammar.
- Switch with your partner.
- Show a derivation and a parse tree for each other's sentences!

Task 3 (b)

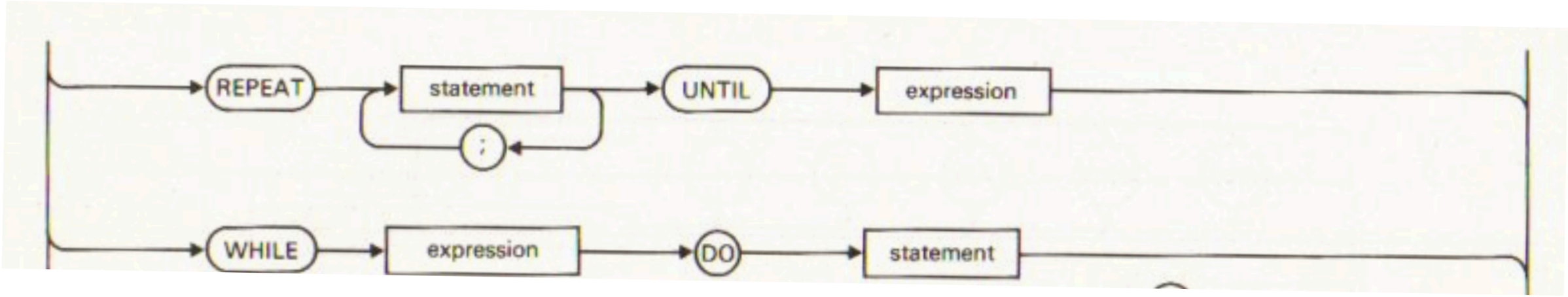
```
S   → NP VP
VP  → V  NP
VP  → V
NP  → N
NP  → Det N
```

```
N   → John
N   → Lisa
N   → house
V   → died
V   → saw
Det → the
Det → a
```

Task 4 (a)

- *Railroad diagrams* are a graphical way to describe the syntax of a programming language. Read about them here:
  - https://en.wikipedia.org/wiki/Syntax_diagram
  - https://www.ibm.com/docs/en/integration-bus/10.0.0?topic=diagrams-how-read-railroad
- On the next page you can see some diagrams for Pascal.
- Translate them into EBNF form!

Task 4 (b)

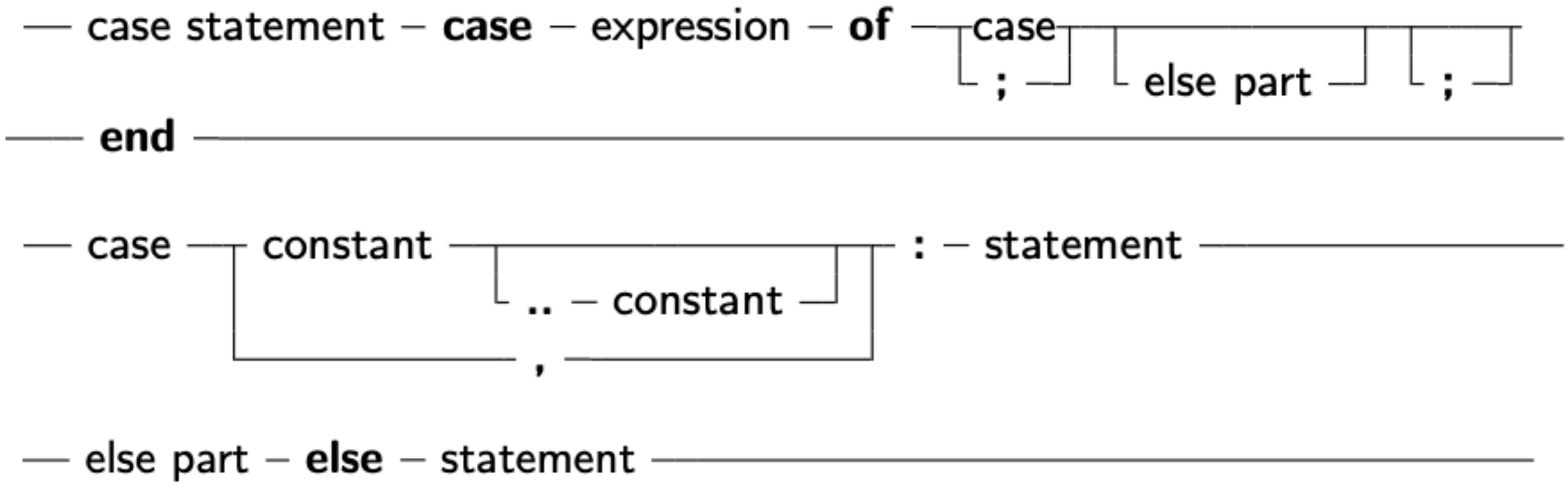


Peter Grogono's "Programming in PASCAL" Addison Wesley 1980

Task 5 (a)

- On the next page you can see the railroad diagrams for case-statements for the language FreePascal.
- In **Task 5 (c)** are two examples of case-statements.
- Pascal's case-statements are like C's switch-statements.
- Construct an *Abstract Grammar* for FreePascal's case-statement and draw the abstract syntax tree for the examples in **Task 5 (c)!**

Task 5 (b)



http://unix12.fzu.cz/localdoc/fpc/ref.pdf

Task 5 (c)

```
Var i : integer;
...
Case i of
 3 : DoSomething;
 1..5 : DoSomethingElse;
end;

Case C of
 'a','e','i','o','u' : WriteLn ('vowel pressed');
 'y' : WriteLn ('This one depends on the language');
else
  WriteLn ('Consonant pressed');
end;
```

http://unix12.fzu.cz/localdoc/fpc/ref.pdf

Task 6

- Classify the following grammars according to Chomsky's hierarchy.

$$S \to aSbc$$
$$S \to abc$$
$$cB \to Bc$$
$$bB \to Bb$$
$$bB \to bb$$

$$S \to aS$$
$$S \to a$$

$$S \to aSb$$
$$S \to ab$$

Task 7

- Construct an annotated abstract syntax tree for this C function.
- Make sure every relevant node has a "type" attribute, and that you assign it either the "int" or the "float" type.

C

```
void foo() {
 int x;
 float y;
 y = x*1.2 + 5;
 if (x) puts("!");
}
```

Task 8

- Modify the EBNF grammar for Luca below so that it no longer has any EBNF constructs (i.e. no curly braces for repetition, and no brackets for optional elements).

Task 9

- Construct an abstract syntax tree for this Luca program:

```
PROGRAM P;
BEGIN
   WHILE x DO
      REPEAT
         IF y THEN
            WRITELN;
         ENDIF;
      UNTIL z;
   ENDDO;
END.
```

Task 10

- Construct an abstract syntax tree for the Luca program below.
- You will have to invent nodes to handle sequence of variable declarations.

```
PROGRAM Euclid;
VAR a      : INTEGER;
VAR b      : INTEGER;
VAR t      : INTEGER;
VAR R      : INTEGER;
BEGIN
   a := 196418;
   b := 317811;
   WHILE b # 0 DO
      t := a;
      a := b;
      b := t % b;
   ENDDO;
   R := a;
   WRITE R-1;
   WRITELN;
END.
```

Definitions and Algorithms

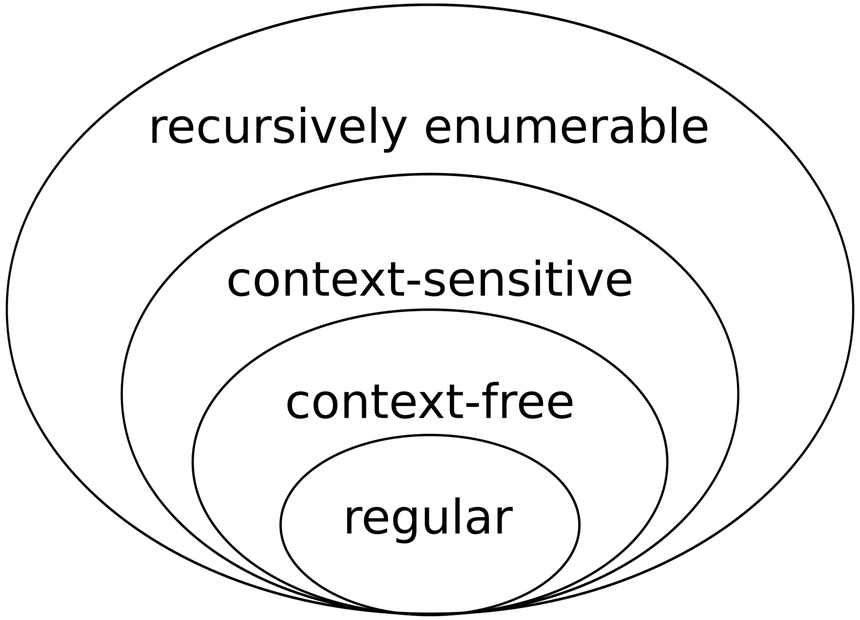
C Operator Precedence & Associativity

Operator	Kind	Prec	Assoc
a[k]	Primary	16	
f(· ··)	Primary	16	
.	Primary	16	
->	Primary	16	
a++, a--	Postfix	15	
++a, --a	Unary	14	
~	Unary	14	
!	Unary	14	
-	Unary	14	
&	Unary	14	
*	Unary	14	

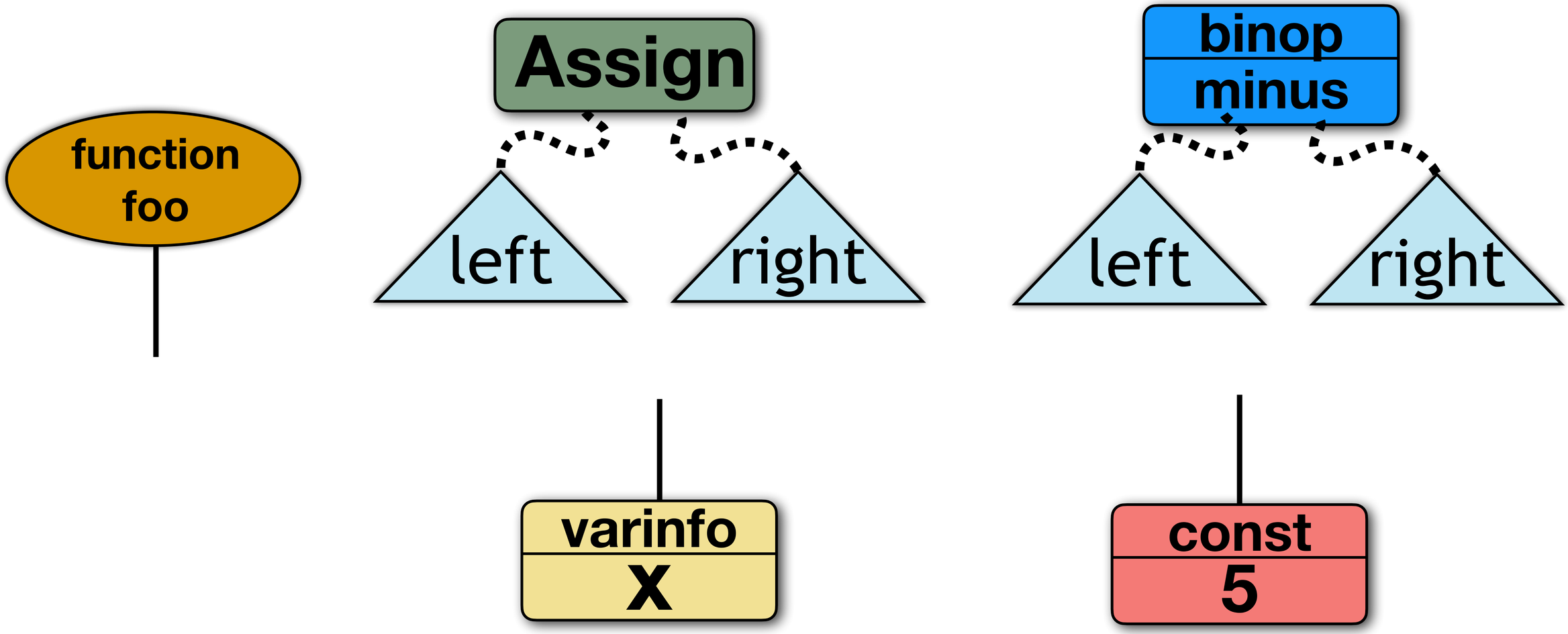
Operator	Kind	Prec	Assoc
*, /, %	Binary	13	Left
+, -	Binary	12	Left
<<, >>	Binary	11	Left
<, >, <=, >=	Binary	10	Left
== !=	Binary	9	Left
&	Binary	8	Left
^	Binary	7	Left
|	Binary	6	Left
&&	Binary	5	Left
||	Binary	4	Left
? :	Ternary	3	Right
=, +=, -=, *=,	Binary	2	Right
/=, %=, <<=,			
>>=, &=, ^=, |=			
,	Binary	1	Left

Chomsky's four types of grammars

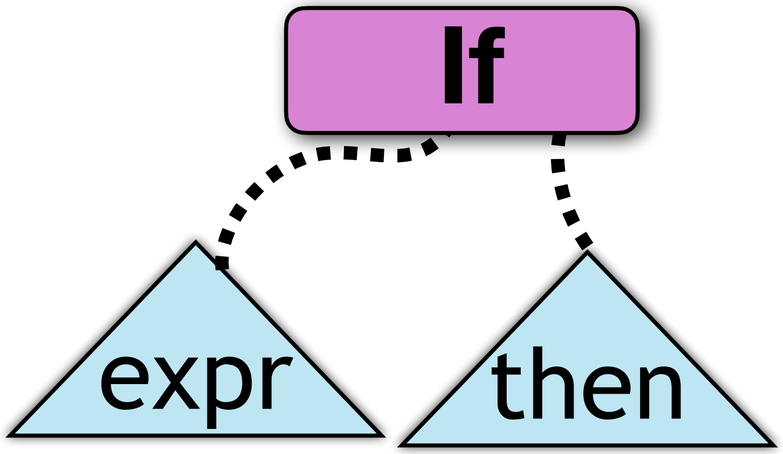
Type	Grammar	Rules
3	Regular	$A \rightarrow aB$ $A \rightarrow a$
2	Context-free	$A \rightarrow \beta$
1	Context-sensitive	$\alpha \rightarrow \beta$, $\lvert\alpha\rvert < \lvert\beta\rvert$
0	Unrestricted	$\alpha \rightarrow \beta$

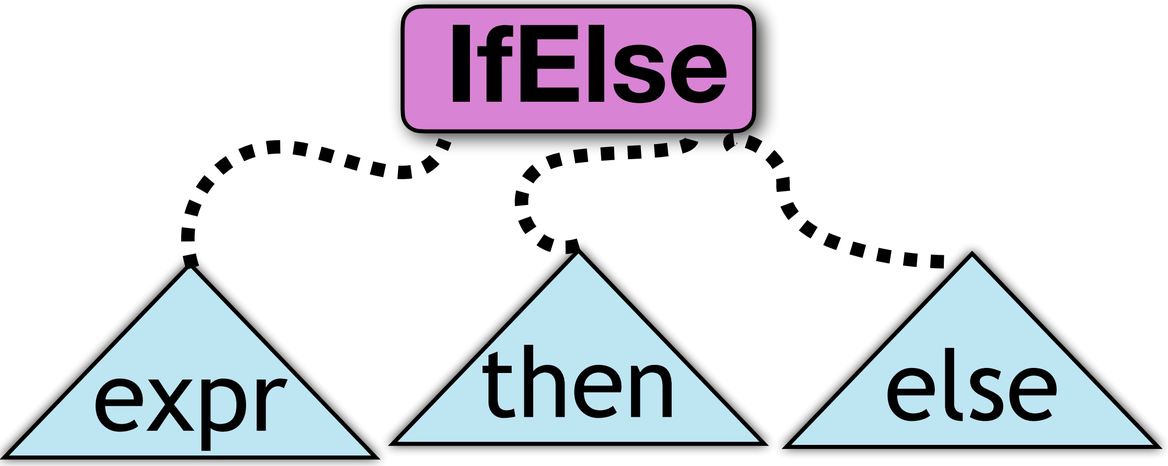


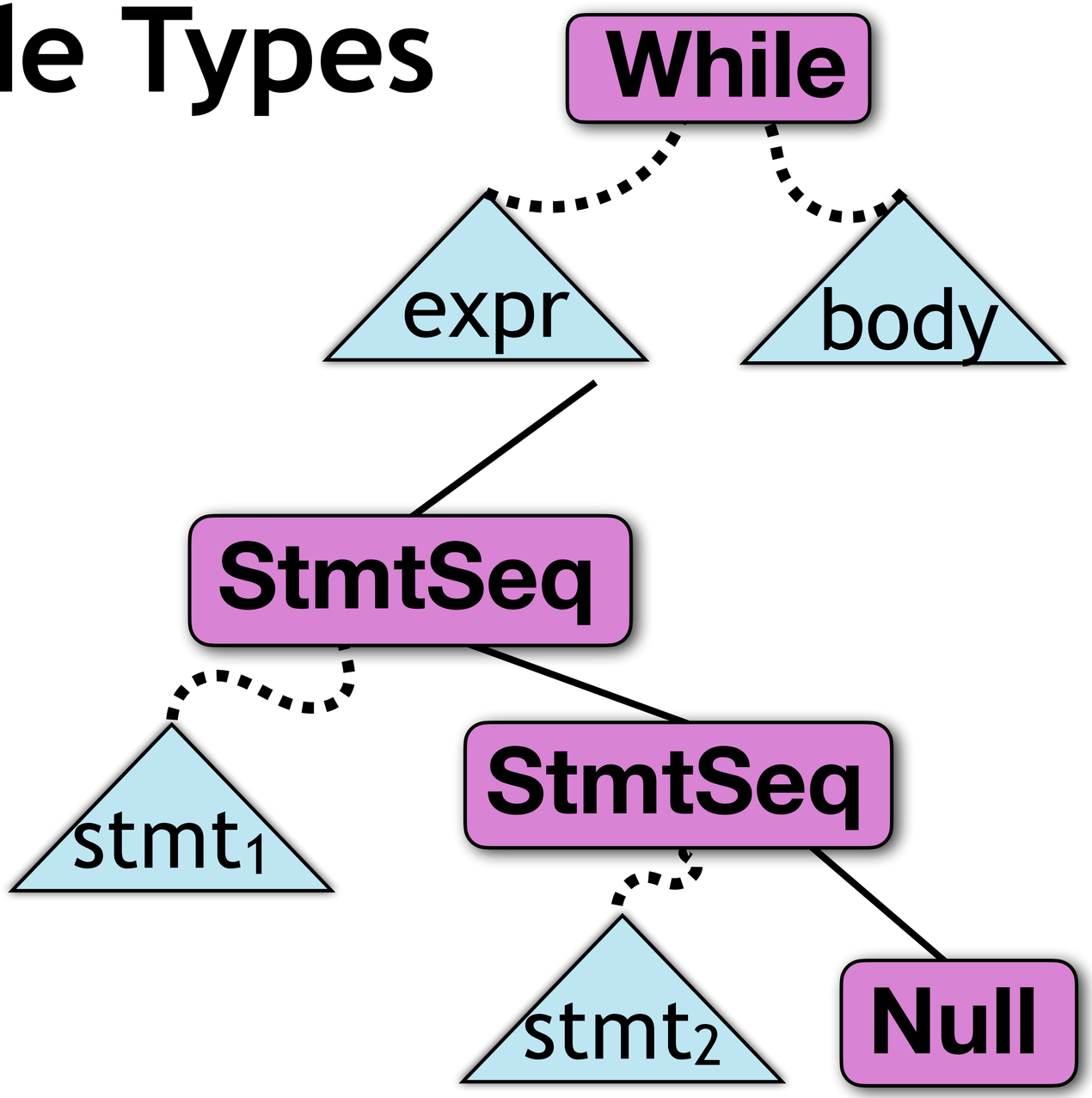
AST Node Types



AST Node Types







EBNF for Luca

```
program ::=
    PROGRAM ident ; decl list block .

block ::= BEGIN stat_seq END

stat_seq ::= { statement ; }
```

Luca Declarations

```
decl list ::=
    { declaration ; }
declaration ::=
    VAR ident : ident |
    TYPE ident = RECORD [ [ fields ] ] |
    TYPE ident = ARRAY expression OF ident |
    CONST ident : ident = expression |
    PROCEDURE ident ( [ formals ] ) decl list block
```

Luca Declarations...

```
field_list ::= field_decl {; field_decl }
field_decl ::= ident : ident
formal_list ::= formal_param {; formal_param }
formal_param ::= [VAR] ident : ident
actual_list ::= expression {, expression }
```

Luca Statements

```
statement ::=
    designator := expression |
    WRITE expression | WRITELN |
    READ designator |
    ident ( [ actual_list ] )
    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
```

Luca Expressions

```
expression ::=
    expression bin_operator expression |
    unary_operator expression |
    ( expression ) |
    real_literal | integer_literal |
    char_literal | string_literal |
    designator

designator ::=
    ident |
    designator [ expression ] |
    designator . ident

unary_operator ::= - | TRUNC | FLOAT| NOT

bin_operator ::= + |- |* |/ |% | < |<= |= | # | >= | > | AND | OR
```

collberg.cs.arizona.edu

ligerlabs.org

Supported by

NSF SATC/TTP-1525820

SATC/EDU-2029632

Christian Collberg

University of Arizona