CS 331 - Assignment 1

Martin Mueller

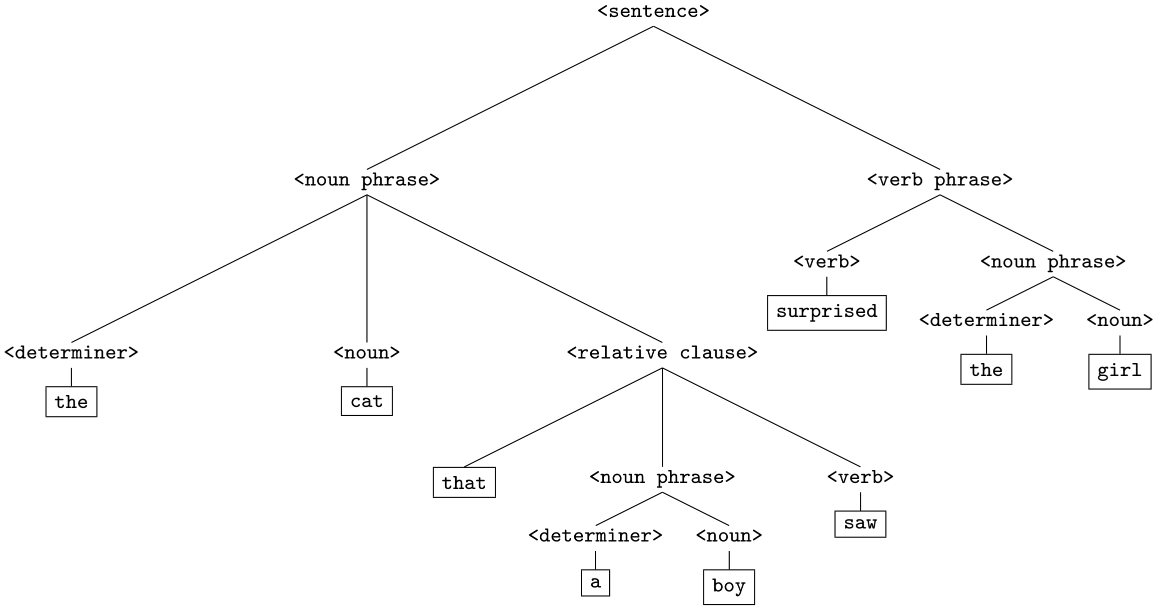
Due: Fri., Feb. 14^{th}, 2020 @ 08:30

1. (15 points) Consider the following BNF grammar G_1:

```
<sentence> ::= <noun phrase> <verb phrase>
<noun phrase> ::= <determiner> <noun> | <determiner> <noun> <relative clause>
<verb phrase> ::= <verb> | <verb> <noun phrase>
<relative clause> ::= that <noun phrase> <verb>
<noun> ::= boy | girl | cat | telescope | song | feather
<determiner> ::= a | the
<verb> ::= saw | touched | surprised | sang
```

Prove with a parse tree, or disprove using the simplest, most CONCISE irrefutable logical argument, that each one of the following `<sentence>`s can be parsed using G_1:

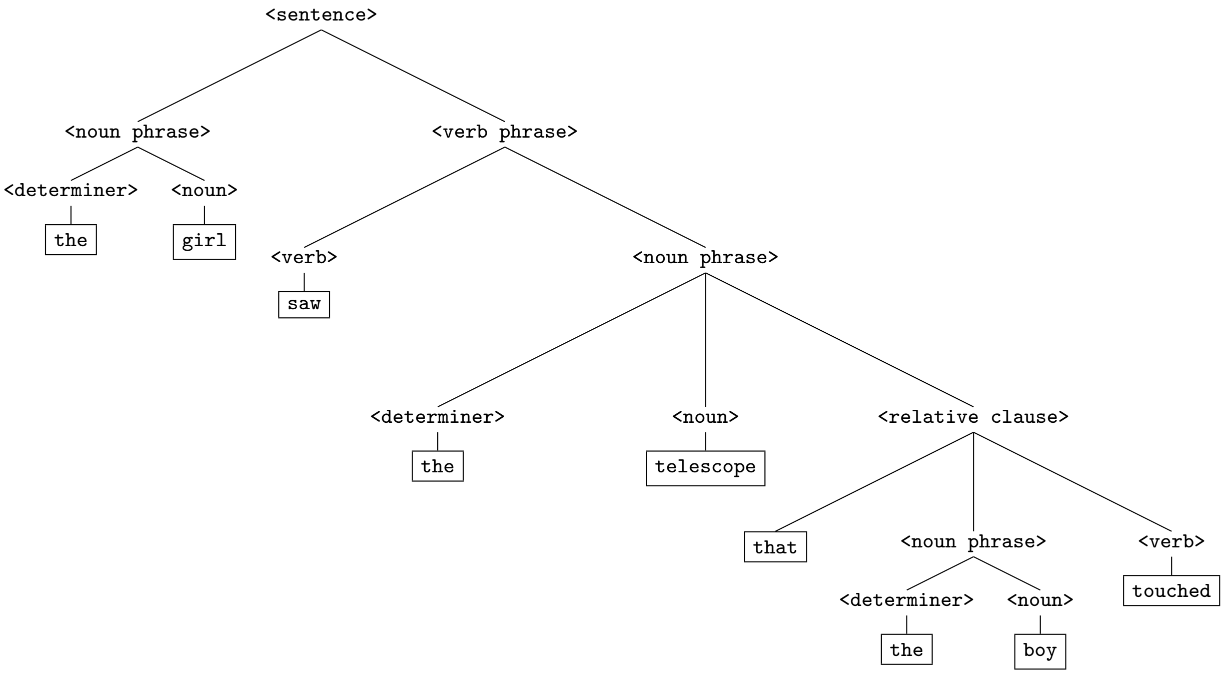
- (3 points) the cat that a boy saw surprised the girl.



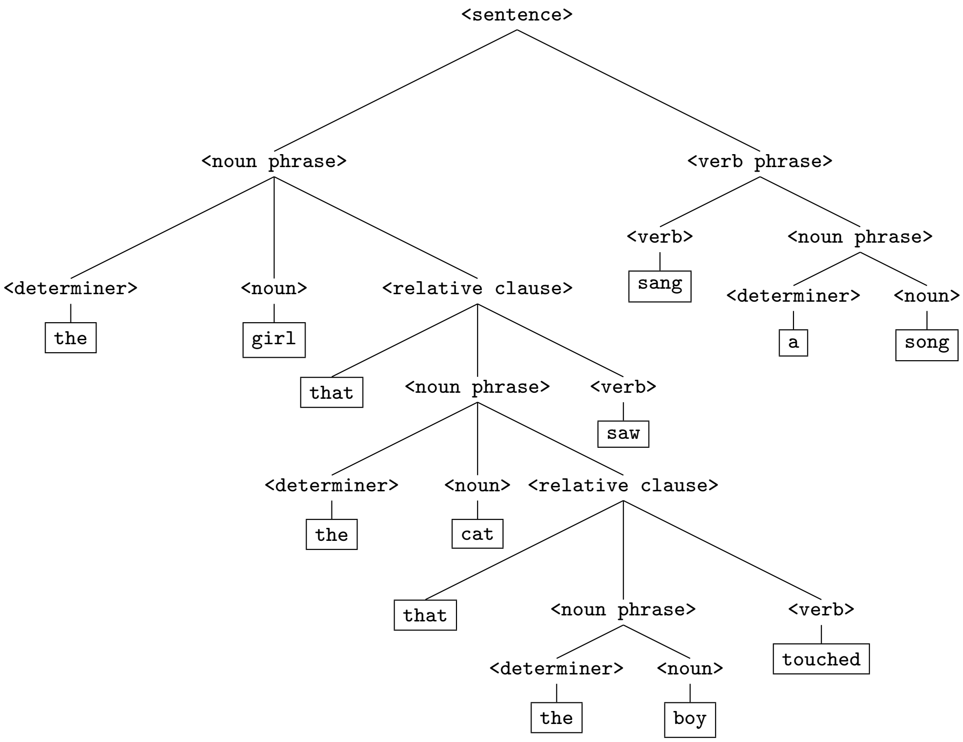
- (3 points) a boy the girl saw sang.

  Not possible. Every sentence is made of a `<noun phrase>` followed by a `<verb phrase>`. Since the word ”that” is absent from the sentence, the first production defining a `<noun phrase>` would fit here to construct ”a boy”. Due to the nature of `<verb phrase>`s, a `<verb>` would need to either stand alone or precede a `<noun phrase>`. Since the next word following ”boy” is an article (or `<determiner>` as it's called here) and not a `<verb>`, this sentence cannot be parsed from G_1.

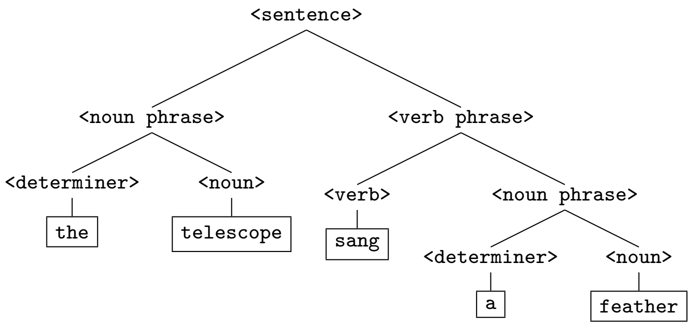
- (3 points) the girl saw the telescope that that boy touched.



- (3 points) the girl that the cat that the boy touched saw sang a song.



- (3 points) the telescope sang a feather.



2. (10 points) Write a BNF grammar (with `<s>` as its start symbol) for the set of ALL (and ONLY) the strings that are made of 0 or more occurrences, in any order, of the letters a and/or b in which the total number of occurrences of the letter a is even.

```
<s>  ::=  <t><a><a> | <a><t><a> | <a><a><t> | <t>
<t>  ::=  <s>b | b<s> | <s> | ε
```

3. (10 points) Write an EBNF grammar for the same language as in the previous problem. For full credit, your answer must use the EBNF extensions WHENEVER possible in order to minimize the number of non-terminals and productions.

```
<s> ::= ((b)*a(b)*a(b)*)*
```

4. (5 points) Consider the following BNF grammar G_4 that defines `expressions` (that is, `<expression>` is the start symbol of the grammar):

```
<expression> ::= <thing> | <thing> op1 <expression>
<object> ::= <element> | <element> op2 <object>
<thing> ::= <object> | <thing> op3 <object>
<element> ::= a | b | c | ( <object> )
```

- (2 points) List the three operators in G_4 from left to right in order of precedence, from highest to lowest.

  op2, op3, op1

- (3 points) For each operator in G_4, state if it is left- or right-associative.

  op1 = right-associative
  op2 = right-associative
  op3 = left-associative

5. (10 points) Consider a fictitious programming language containing only two numerical types (integer and real), in which arbitrarily long (possibly empty) 1-dimensional arrays of numbers are declared, instantiated, and initialized in a single statement, two of which are given below:

```
integer[] numbers = { 1, 5, 7, 9, 10 };
real[] reals = { 2.0, 3.35, 1.24, 54.145, -4.9 };
```

Write an EBNF grammar with `<a>` as its start symbol that can generate **all** and **only** the syntactically correct array declarations of this form. Instead of creating rules to generate the set of all possible identifiers (variable names), integer constants, and floating point constants, your grammar must refer to the non-terminals `<id>`, `<int>`, and `<real>`, which can be assumed to be already defined elsewhere. (To think about: Why does it make sense for these non terminals to be defined in another grammar, or in a distinct part of the grammar?)

For full credit, you MUST use EBNF extensions WHENEVER possible to minimize the number of productions in your grammar.

```
<s> ::= integer[] <id> = { (<int>(, <int>)*)?  };
              | real[] <id> = { (<real>(, <real>)*)?  };
```

6. (10 points) Consider the following two alternative BNF grammar fragments for defining conditional statements:

```
<statement> ::= <assignment stmt> | <cond stmt1> | <while stmt> | ...
<cond stmt1> ::= if <boolean expr> then <statement> endif
<cond stmt1> ::= if <boolean expr> then <statement> else <statement> endif

<statement> ::= <assignment stmt> | <cond stmt2> | <while stmt> | ...
<cond stmt2> ::= if <boolean expr> then <statement>
<cond stmt2> ::= if <boolean expr> then <statement> else <statement>
```

Are these two grammars equivalent, or is one better than the other in some way? Justify your answer precisely. Hint: Think about the important properties that grammars (NOT the language) may have and argue that these two grammars share exactly the same important properties, or that one has an important property that the other one does not have.

These grammars are nearly the same, but there is one key difference. The first grammar solves the dangling else problem. The first grammar removes any ambiguity relating to the placement of `else` in nested `if` statements since each `if` statement always ends with an `endif`. In contrast, the second grammar would have an unresolved logical issue with nested `if` statements. If some nested `if` statements contained `else`s and others did not, then it may not be clear which `else`s go along with which `if`s without some special rule.

7. (5 points) Is the following grammar ambiguous? Remember to prove your answer as concisely as possible.

```
<game> ::= <one-player> | <two-player>
<one-player> ::= solitaire | minesweeper | ϵ
<two-player> ::= chess | backgammon | ϵ
```

This grammar is ambiguous because it has two different parse trees that produce the same terminals: