Formal Languages

Matthias Braun

Today's Plan

- What's a formal language?

Today's Plan

- What's a formal language?
- How can we create our own formal language?

Formal Languages

The fuzzy explanation

- A language is a system to express ideas

Formal Languages

The fuzzy explanation

- A language is a system to express ideas
- Formal languages are a generalization of programming languages

Formal Languages

The fuzzy explanation

- A language is a system to express ideas
- Formal languages are a generalization of programming languages
- A formal language has rules that define which sentences (think programs) are allowed

Formal Languages

The fuzzy explanation

- A language is a system to express ideas
- Formal languages are a generalization of programming languages
- A formal language has rules that define which sentences (think programs) are allowed
- This allows us to create (programming) languages!

Alphabet

- A formal language has an alphabet

Alphabet

- A formal language has an alphabet
- An alphabet consists of symbols, for example:

Alphabet

- A formal language has an alphabet
- An alphabet consists of symbols, for example:
  - A binary alphabet: $\{0, 1\}$

Alphabet

- A formal language has an alphabet
- An alphabet consists of symbols, for example:
  - A binary alphabet: $\{0, 1\}$
  - Three-letter alphabet: $\{a, b, c\}$

Alphabet

- A formal language has an alphabet
- An alphabet consists of symbols, for example:
  - A binary alphabet: $\{0, 1\}$
  - Three-letter alphabet: $\{a, b, c\}$
  - Alphabet for a game: $\{\clubsuit, \diamondsuit, \heartsuit, \spadesuit\}$

Alphabet

- A formal language has an alphabet
- An alphabet consists of symbols, for example:
  - A binary alphabet: $\{0, 1\}$
  - Three-letter alphabet: $\{a, b, c\}$
  - Alphabet for a game: $\{\clubsuit,\diamondsuit,\heartsuit,\spadesuit\}$
  - Alphabet for DNA: {G,A,T,C}

Alphabet

- A formal language has an alphabet
- An alphabet consists of symbols, for example:
  - A binary alphabet: $\{0, 1\}$
  - Three-letter alphabet: $\{a, b, c\}$
  - Alphabet for a game: $\{\clubsuit,\diamondsuit,\heartsuit,\spadesuit\}$
  - Alphabet for DNA: {G,A,T,C}
- We construct **strings** from the alphabet's symbols:

Alphabet

- A formal language has an alphabet
- An alphabet consists of symbols, for example:
  - A binary alphabet: $\{0, 1\}$
  - Three-letter alphabet: $\{a, b, c\}$
  - Alphabet for a game: $\{\clubsuit,\diamondsuit,\heartsuit,\spadesuit\}$
  - Alphabet for DNA: {G,A,T,C}
- We construct **strings** from the alphabet's symbols:
  - 11000101011001

Alphabet

- A formal language has an alphabet
- An alphabet consists of symbols, for example:
  - A binary alphabet: $\{0, 1\}$
  - Three-letter alphabet: $\{a, b, c\}$
  - Alphabet for a game: $\{\clubsuit,\diamondsuit,\heartsuit,\spadesuit\}$
  - Alphabet for DNA: {G,A,T,C}
- We construct **strings** from the alphabet's symbols:
  - 11000101011001
  - bcababbaac

Alphabet

- A formal language has an alphabet
- An alphabet consists of symbols, for example:
  - A binary alphabet: $\{0, 1\}$
  - Three-letter alphabet: $\{a, b, c\}$
  - Alphabet for a game: {♣,♢,♡,♠}
  - Alphabet for DNA: {G,A,T,C}
- We construct **strings** from the alphabet's symbols:
  - 11000101011001
  - bcababbaac
  - ♢♠♡♡♣♠

Alphabet

- A formal language has an alphabet
- An alphabet consists of symbols, for example:
  - A binary alphabet: $\{0, 1\}$
  - Three-letter alphabet: $\{a, b, c\}$
  - Alphabet for a game: {♣,♢,♡,♠}
  - Alphabet for DNA: {G,A,T,C}
- We construct **strings** from the alphabet's symbols:
  - 11000101011001
  - bcababbaac
  - ♢♠♡♡♣♠
  - ATAAAAAACTAG

Languages are built from Alphabets

- A language built from the alphabet $\{a, b, c\}$:
  $\{aa, cb, baba, cabab\}$

Languages are built from Alphabets

- A language built from the alphabet $\{a, b, c\}$:
  $\{aa, cb, baba, cabab\}$
- Another language from the same alphabet:
  $\{ccccc\}$

Languages are built from Alphabets

- A language built from the alphabet $\{a, b, c\}$:
  $\{aa, cb, baba, cabab\}$
- Another language from the same alphabet:
  $\{ccccc\}$
- Let's concatenate those two languages:
  $\{aa, cb, baba, cabab, ccccc\}$

Languages are built from Alphabets

- A language built from the alphabet $\{a, b, c\}$:
  $\{aa, cb, baba, cabab\}$
- Another language from the same alphabet:
  $\{ccccc\}$
- Let's concatenate those two languages:
  $\{aa, cb, baba, cabab, ccccc\}$
- Languages are sets of strings

Grammar

- A grammar for English tells us whether a sentence is well formed or not:
  - “I went skiing”
  - “went skiing I”

Grammar

- A grammar for English tells us whether a sentence is well formed or not:
  - "I went skiing"
  - "went skiing I"
- Using a grammar, we can also *generate* sentences of a language

Grammar

- A grammar for English tells us whether a sentence is well formed or not:
  - "I went skiing"
  - "went skiing I"
- Using a grammar, we can also *generate* sentences of a language
- An example grammar:

```
a = "a";
b = "b";
c = "c";
sentence = c, a, b;
```

Grammar

- A grammar for English tells us whether a sentence is well formed or not:
  - “I went skiing”
  - “went skiing I”
- Using a grammar, we can also *generate* sentences of a language
- An example grammar:

```
a = "a";
b = "b";
c = "c";
sentence = c, a, b;
```

- Comma means concatenation

Grammar

- A grammar for English tells us whether a sentence is well formed or not:
  - "I went skiing"
  - "went skiing I"
- Using a grammar, we can also *generate* sentences of a language
- An example grammar:

```
a = "a";
b = "b";
c = "c";
sentence = c, a, b;
```

- Comma means concatenation
- Generated language: $\{cab\}$

More Grammars

- Another grammar:

More Grammars

- Another grammar:

```
subject = "I" | "You" | "We";
```

More Grammars

- Another grammar:

```
subject = "I" | "You" | "We";
verb = "went" | "love" | "hate";
```

More Grammars

- Another grammar:

```
subject = "I" | "You" | "We";
verb = "went" | "love" | "hate";
object = "skiing" | "playing chess" | "Mark
Ruffalo";
```

More Grammars

- Another grammar:

```
subject = "I" | "You" | "We";
verb = "went" | "love" | "hate";
object = "skiing" | "playing chess" | "Mark
Ruffalo";
sentence = subject, " ", verb, " ", object;
```

More Grammars

- Another grammar:

```
subject = "I" | "You" | "We";
verb = "went" | "love" | "hate";
object = "skiing" | "playing chess" | "Mark
Ruffalo";
sentence = subject, " ", verb, " ", object;
```

- Some strings of the language generated from this grammar:

More Grammars

- Another grammar:

```
subject = "I" | "You" | "We";
verb = "went" | "love" | "hate";
object = "skiing" | "playing chess" | "Mark
Ruffalo";
sentence = subject, " ", verb, " ", object;
```

- Some strings of the language generated from this grammar:
  - “I love skiing”

More Grammars

- Another grammar:

```
subject = "I" | "You" | "We";
verb = "went" | "love" | "hate";
object = "skiing" | "playing chess" | "Mark
Ruffalo";
sentence = subject, " ", verb, " ", object;
```

- Some strings of the language generated from this grammar:
  - “I love skiing”
  - “You went Mark Ruffalo”

More Grammars

- Another grammar:

```
subject = "I" | "You" | "We";
verb = "went" | "love" | "hate";
object = "skiing" | "playing chess" | "Mark
Ruffalo";
sentence = subject, " ", verb, " ", object;
```

- Some strings of the language generated from this grammar:
  - “I love skiing”
  - “You went Mark Ruffalo”
  - “I hate playing chess”

More Grammars

- Another grammar:

```
subject = "I" | "You" | "We";
verb = "went" | "love" | "hate";
object = "skiing" | "playing chess" | "Mark
Ruffalo";
sentence = subject, " ", verb, " ", object;
```

- Some strings of the language generated from this grammar:
  - "I love skiing"
  - "You went Mark Ruffalo"
  - "I hate playing chess"
  - "I love playing chess"

More Grammars

- Another grammar:

```
subject = "I" | "You" | "We";
verb = "went" | "love" | "hate";
object = "skiing" | "playing chess" | "Mark
Ruffalo";
sentence = subject, " ", verb, " ", object;
```

- Some strings of the language generated from this grammar:
  - “I love skiing”
  - “You went Mark Ruffalo”
  - “I hate playing chess”
  - “I love playing chess”
- Vertical bar | means choice (”or”)

More Grammars

- Another grammar:

```
subject = "I" | "You" | "We";
verb = "went" | "love" | "hate";
object = "skiing" | "playing chess" | "Mark
Ruffalo";
sentence = subject, " ", verb, " ", object;
```

- Some strings of the language generated from this grammar:
  - "I love skiing"
  - "You went Mark Ruffalo"
  - "I hate playing chess"
  - "I love playing chess"
- Vertical bar | means choice ("or")
- How many different strings (sentences) does the language generated from this grammar have?

Grammars, formalized

- A grammar consists of **productions rules**:
  rule1 = "I" | "You" | "We";
  rule2 = "went" | "love" | "hate";
  rule3 = "skiing" | "playing chess" | "Mark Ruffalo";
  rule4 = rule1, " ", rule2, " ", rule3;

Grammars, formalized

- A grammar consists of **productions rules**:
  rule1 = "I" | "You" | "We";
  rule2 = "went" | "love" | "hate";
  rule3 = "skiing" | "playing chess" | "Mark Ruffalo";
  rule4 = rule1, " ", rule2, " ", rule3;
- Production rules define how to generate strings: rule 4 is replaced with rule 1, 2, and 3 (separated by spaces)

Grammars, formalized

- A grammar consists of **productions rules**:
  rule1 = "I" | "You" | "We";
  rule2 = "went" | "love" | "hate";
  rule3 = "skiing" | "playing chess" | "Mark Ruffalo";
  rule4 = rule1, " ", rule2, " ", rule3;
- Production rules define how to generate strings: rule 4 is replaced with rule 1, 2, and 3 (separated by spaces)
- A rule is also called a **non-terminal** since we haven't finished/terminated replacing them to produce symbols

Grammars, formalized

- A grammar consists of **productions rules**:
  rule1 = "I" | "You" | "We";
  rule2 = "went" | "love" | "hate";
  rule3 = "skiing" | "playing chess" | "Mark Ruffalo";
  rule4 = rule1, " ", rule2, " ", rule3;
- Production rules define how to generate strings: rule 4 is replaced with rule 1, 2, and 3 (separated by spaces)
- A rule is also called a **non-terminal** since we haven't finished/terminated replacing them to produce symbols
- A rule is replaced by other rules or a **terminal** symbol from the languages alphabet

Grammars, formalized

- A grammar consists of **productions rules**:
  rule1 = "I" | "You" | "We";
  rule2 = "went" | "love" | "hate";
  rule3 = "skiing" | "playing chess" | "Mark Ruffalo";
  rule4 = rule1, " ", rule2, " ", rule3;
- Production rules define how to generate strings: rule 4 is replaced with rule 1, 2, and 3 (separated by spaces)
- A rule is also called a **non-terminal** since we haven't finished/terminated replacing them to produce symbols
- A rule is replaced by other rules or a **terminal** symbol from the languages alphabet
- Terminals can't be replaced by anything else, they are quoted

Grammars, formalized

- A grammar consists of **productions rules**:
  rule1 = "I" | "You" | "We";
  rule2 = "went" | "love" | "hate";
  rule3 = "skiing" | "playing chess" | "Mark Ruffalo";
  rule4 = rule1, " ", rule2, " ", rule3;
- Production rules define how to generate strings: rule 4 is replaced with rule 1, 2, and 3 (separated by spaces)
- A rule is also called a **non-terminal** since we haven't finished/terminated replacing them to produce symbols
- A rule is replaced by other rules or a **terminal** symbol from the languages alphabet
- Terminals can't be replaced by anything else, they are quoted
- The notation we use for our grammars is called **extended Backus-Naur form** (EBNF)

Formal Languages

The formal explanation

- A formal language is defined by an alphabet and a grammar
- A formal language is the set of all strings generated from the grammar

Test your knowledge

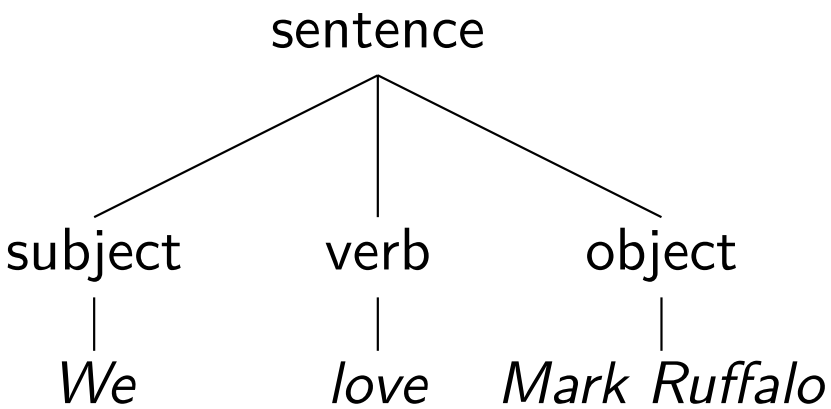
- List all the terminals from the previous grammar
- How many production rules does the grammar have?
- In EBNF, what's the difference between the comma and the vertical bar?

Parse Trees

- Example string: “We love Mark Ruffalo”

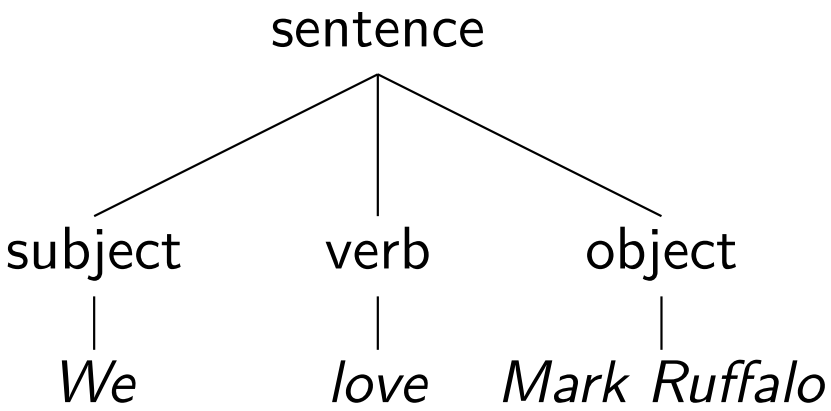
Parse Trees

- Example string: "We love Mark Ruffalo"
- Parse tree:



Parse Trees

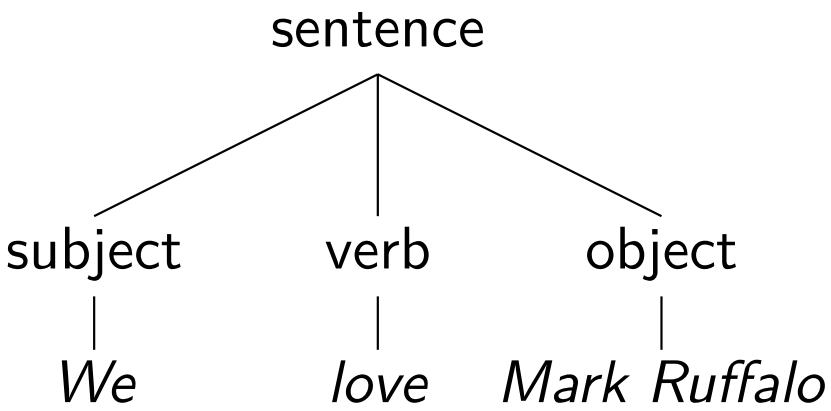
- Example string: "We love Mark Ruffalo"
- Parse tree:



- Each leaf is a terminal symbol

Parse Trees

- Example string: "We love Mark Ruffalo"
- Parse tree:



- Each leaf is a terminal symbol
- Each interior node is a non-terminal production rule

Recursive Grammars

- All the grammars we've seen so far generate a finite amount of strings

Recursive Grammars

- All the grammars we've seen so far generate a finite amount of strings
- What about this one:

```
bit = "0" | "1";
sentence = bit, sentence;
```

Recursive Grammars

- All the grammars we've seen so far generate a finite amount of strings
- What about this one:

```
bit = "0" | "1";
sentence = bit, sentence;
```

- How do we make it stop?

Recursive Grammars

- All the grammars we've seen so far generate a finite amount of strings
- What about this one:

```
bit = "0" | "1";
sentence = bit, sentence;
```

- How do we make it stop?
- `sentence = bit, (sentence | "");`

Your turn

- Write a grammar for a language with *exactly* these three strings:
  {abc, cba, bac}
- A grammar for a language with an *infinite* number of strings like these:
  {a, aa, aaa, aaaa, aaaaa, ...}
- A grammar for a language with an *infinite* number of strings like these:
  {a, b, aba, ababa, aaaa, ...}

Grammar for Palindromes

Reads the same from left to right and right to left

- We want to create bit palindromes:
  - "0110"
  - "11"
  - "10101"
  - "0"
  - ""

Grammar for Palindromes

Reads the same from left to right and right to left

- We want to create bit palindromes:
  - "0110"
  - "11"
  - "10101"
  - "0"
  - ""
- Start with the non-recursive base cases:

Grammar for Palindromes

Reads the same from left to right and right to left

- We want to create bit palindromes:
  - "0110"
  - "11"
  - "10101"
  - "0"
  - ""
- Start with the non-recursive base cases:

```
1 pal = "" | "1" | "0";
```

Grammar for Palindromes

Reads the same from left to right and right to left

- We want to create bit palindromes:
  - "0110"
  - "11"
  - "10101"
  - "0"
  - ""
- Start with the non-recursive base cases:

```
1 pal = "" | "1" | "0";
```

- Now the recursive cases:

Grammar for Palindromes

Reads the same from left to right and right to left

- We want to create bit palindromes:
  - "0110"
  - "11"
  - "10101"
  - "0"
  - ""
- Start with the non-recursive base cases:

```
1 pal = "" | "1" | "0";
```

- Now the recursive cases:

```
2 pal = "0", pal, "0";
```

Grammar for Palindromes

Reads the same from left to right and right to left

- We want to create bit palindromes:
  - "0110"
  - "11"
  - "10101"
  - "0"
  - ""
- Start with the non-recursive base cases:

```
1 pal = "" | "1" | "0";
```

- Now the recursive cases:

```
2 pal = "0", pal, "0";
3 pal = "1", pal, "1";
```

Grammar for Palindromes

Reads the same from left to right and right to left

- We want to create bit palindromes:
  - "0110"
  - "11"
  - "10101"
  - "0"
  - ""
- Start with the non-recursive base cases:

```
1 pal = "" | "1" | "0";
```

- Now the recursive cases:

```
2 pal = "0", pal, "0";
3 pal = "1", pal, "1";
```

- Let's generate the example palindromes above by replacing `pal` step-by-step

Grammar for Addition and Subtraction

- Some valid expressions:
  - “1 + 2”
  - “5”
  - “3 - 1 + 5”
  - “3 - 1 + 5 + 8”

Grammar for Addition and Subtraction

- Some valid expressions:
  - “1 + 2”
  - “5”
  - “3 - 1 + 5”
  - “3 - 1 + 5 + 8”
- Production rules for the base cases:

```
1 expr = digit;
2 digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" |
  "7" | "8" | "9";
```

Grammar for Addition and Subtraction

- Some valid expressions:
  - “1 + 2”
  - “5”
  - “3 - 1 + 5”
  - “3 - 1 + 5 + 8”
- Production rules for the base cases:

```
1 expr = digit;
2 digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" |
  "7" | "8" | "9";
```

- Production rules for the recursive cases:

Grammar for Addition and Subtraction

- Some valid expressions:
  - “1 + 2”
  - “5”
  - “3 - 1 + 5”
  - “3 - 1 + 5 + 8”
- Production rules for the base cases:

```
1 expr = digit;
2 digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" |
  "7" | "8" | "9";
```

- Production rules for the recursive cases:

```
3 expr = expr, " + ", digit;
```

Grammar for Addition and Subtraction

- Some valid expressions:
  - “1 + 2”
  - “5”
  - “3 - 1 + 5”
  - “3 - 1 + 5 + 8”
- Production rules for the base cases:

```
1 expr = digit;
2 digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" |
  "7" | "8" | "9";
```

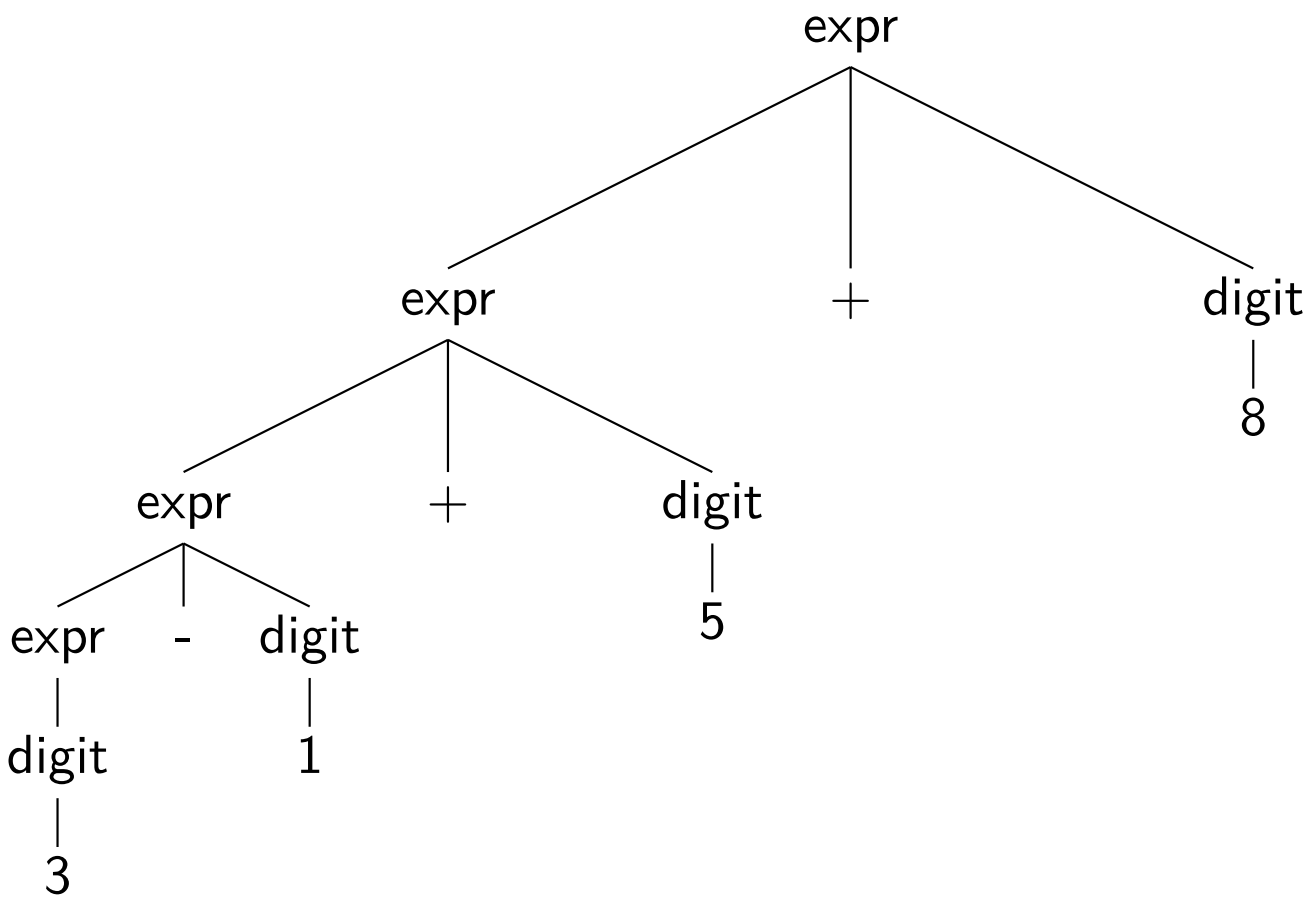
- Production rules for the recursive cases:

```
3 expr = expr, " + ", digit;
4 expr = expr, " - ", digit;
```

Grammar for Addition and Subtraction

Parse Tree for "3 - 1 + 5 + 8"

Notice how this tree grows down towards the *left* because the recursive part, `expr`, is on the left-hand side of `expr, " + ", digit;` and `expr, " - ", digit;`



Your turn

- Generate the following strings from the previous grammar and draw the parse tree. Make a note of the production rule you use in each step:
  - "1"
  - "1 + 2"
  - "8 + 9 - 0"
- Write down a grammar for the language of propositional logic (you know: $\vee$, $\wedge$, $\neg$, parentheses). You can limit the number of different variables in this language to a, b, and c. The language should contain strings like these:
  - "a"
  - "$a \wedge b$"
  - "$a \vee \neg b$"
  - "$a \vee \neg(b \wedge c)$"
  - "$a \vee \neg a$"
- Hint: You can do this with *two* production rules

Your Turn

- Create a grammar that generates integer numbers larger than 99.
- Create a grammar that generates constants in a programming language. The naming convention for constants is that a constant:
  - starts with an upper-case letter
  - only contains upper-case letters and underscores (no digits)
  - has at least one letter
- Let's say that a password should contain at least one lower-case letter and a digit. The minimum password length is two, the maximum is infinity. Create a grammar that generates such passwords. Some valid example passwords:
  - "a0"
  - "9z"
  - "z12ab90zkl"
  - "hunter2"