NTNU | Norwegian University of Science and Technology

Compiler Construction

Lecture 5: Introduction to Parsing

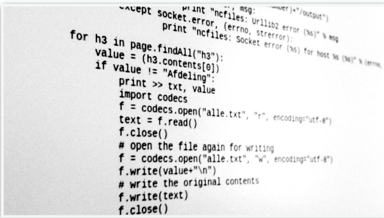
Michael Engel

Overview

- Compiler structure revisited
  - Interaction of scanner and parser
- Context-free languages
- Ambiguity of grammars
- BNF grammars
- Language classes and Chomsky hierarchy

Stages of a compiler (1)

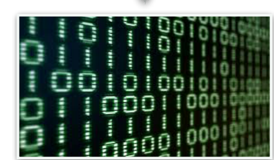
Source code



character stream

Lexical analysis → Syntax analysis → Semantic analysis → Code optimization → Code generation

token sequence



machine-level program

Lexical analysis (scanning):

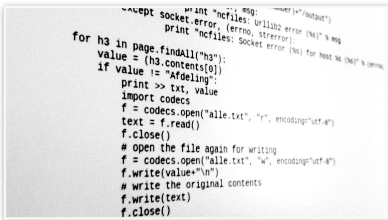
- Split source code into *lexical units*
- Recognize *tokens* (using regular expressions/automata)
- Token: character sequence relevant to source language grammar

x = y + 42 → id(x) op(=) id(y) op(+) number(42)

character stream *token sequence*

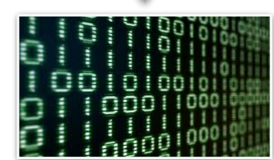
Stages of a compiler (2)

Source code



Lexical analysis → **Syntax analysis** → Semantic analysis → Code optimization → Code generation

token sequence *syntax tree*

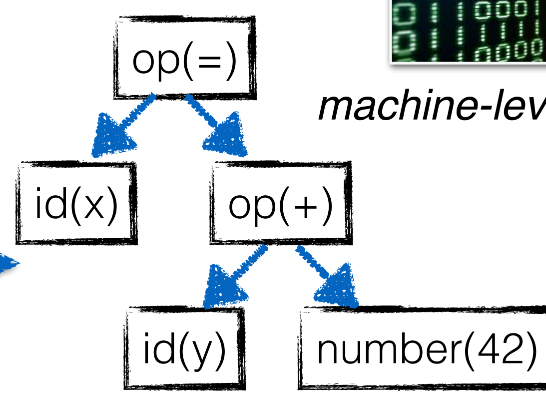


machine-level program

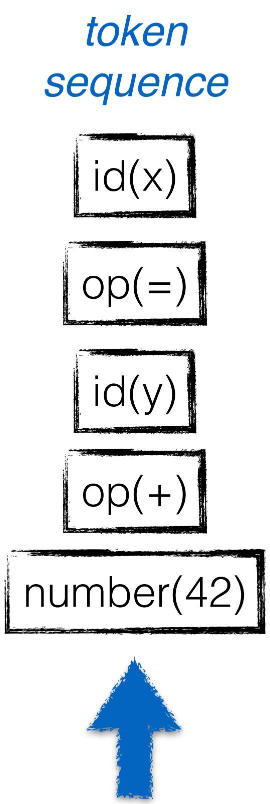
Syntax analysis (parsing)

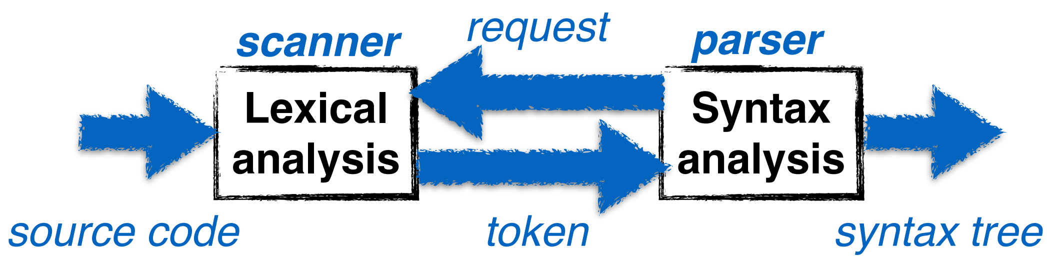
– Uses *grammar* of the source language
– Decides if input *token sequence* can be derived from the grammar

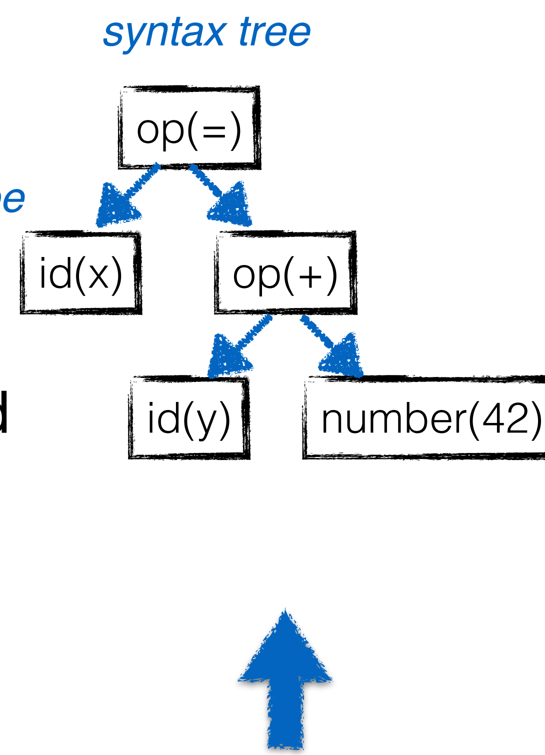
```
expression → term { (+|-) term }
term →       factor { (*|/) factor }
factor →     '(' expression ')'
             | id | number
```



Interaction of scanner and parser







Often, interaction between parser and scanner takes place

- e.g., parser requests next tokens from scanner

```
[0-9]+                 { return(NUMBER); }
[A-Za-z][A-Za-z0-9]*   { return(ID); }
=                      { return(OP); }
\+                     { return(OP); }
```

regular expressions/automaton

```
expression →  term { (+|-) term }
term →        factor { (*|/) factor }
factor →      '(' expression ')'
              | id | number
```

grammar

Parsing

Syntax analysis

- Parsing is the second stage of the compiler's front end
  - it works with program as transformed by the scanner
  - it sees a stream of words
    - each word is annotated with a syntactic category

syntactic category (returned token type) → number(42) ← *word (yytext)*

- Parser derives a syntactic structure for the program
  - it fits the words into a grammatical model of the source programming language
- Two possible outcomes:
  - ✔ input is valid program: builds a concrete model of the program for use by the later phases of compilation
  - ✘ input is not a valid program: report problem and diagnosis

Definition of parsing

- Task of the parser:
  - determining if the program being compiled is a valid sentence in the syntactic model of the programming language
- A bit more formal:
  - the syntactic model is expressed as ***formal grammar G***
  - *some string of words s is in the language defined by G we say that* ***G derives s***
  - *for a stream of words s and a grammar G, the parser tries to build a* ***constructive proof that s can be derived in G***

    — this is called parsing
- It's not as bad as it sounds…
  - we let the computer do (most of) the work!

Specifying language syntax

Syntax analysis

- We need…
  - a formal mechanism for specifying the syntax of the source language (grammar)
  - a systematic method of determining membership in this formally specified language (parsing)
- Let's make our lives a bit easier
  - we restrict the form of the source language to a set of languages called ***context-free languages***
  - typical parsers can efficiently answer the membership question for those
- Many different parsing algorithms exist, we will look at
  - ***top-down parsing:*** recursive descent and LL(1) parsers
  - ***bottom-up parsing:*** LR(1) parsers

Parsing approaches in general

- ***Top-down parsing:*** recursive descent and LL(1) parsers
  - Top-down parsers try to match the input stream against the productions of the grammar by predicting the next word (at each point)
  - For a limited class of grammars, such prediction can be both accurate and efficient
- ***Bottom-up parsing:*** LR(1) parsers
  - Bottom-up parsers work from low-level detail—the actual sequence of words—and accumulate context until the derivation is apparent
  - Again, there exists a restricted class of grammars for which we can generate efficient bottom-up parsers
- In practice, these restricted sets of grammars are large enough to encompass most features of interest in programming languages

Expressing syntax

Syntax analysis

- We already know a way to express syntax: ***regular expressions***
- Why are regexps not suitable for describing language syntax?

Example: recognizing algebraic expressions over variables and the operators +, -, ×, ÷

```
variable   = [a…z]( [a…z] | [0…9] )*
expression = [a…z]( [a…z] | [0…9] )* ( (+|-|×|÷) [a…z]( [a…z] | [ 0…9] )*)*
```

- This regexp matches e.g. "a+b×c" and "dee÷daa×doo"
- However, there is no way to express ***operator precedence***
  - should + or × be executed first in "a+b×c"?
  - standard rule from algebra suggests: "× and ÷ have precedence over + and -"

Expressing syntax: regexps?

Syntax analysis

```
variable   = [a…z]( [a…z] | [0…9] )*
expression = [a…z]( [a…z] | [0…9] )* ( (+|-|×|÷) [a…z]( [a…z] | [ 0…9] )*)*
```

- There is no way to express ***operator precedence***
  - to enforce evaluation order, algebraic notation uses parentheses
- Adding parentheses in regexps is tricky…
  - an expression can start with a "(", so we need the option for an initial "(". Similarly, we need the option for a final ")":

Literal parentheses are printed in red and enclosed in "": "("

```
("("|ε) [a…z]([a…z]|[0…9])* ((+|-|×|÷) [a…z] ([a…z]|[0…9])* )* (")"|ε)
```

- This regexp can produce an expression enclosed in parentheses, but ***not one with internal parentheses*** to denote precedence

Expressing syntax: regexps?

Syntax analysis

```
("("|ε) [a…z]([a…z]|[0…9])* ((+|-|×|÷) [a…z] ([a…z]|[0…9])* )* (")"|ε)
```

- This regexp can produce an expression enclosed in parentheses, but ***not one with internal parentheses*** to denote precedence
- Internal instances of "(" all occur before a variable
  - similarly, the internal instances of ")" all occur after a variable
  - so let's move the closing parenthesis inside the final *:

```
("("|ε) [a…z]([a…z]|[0…9])* ((+|-|×|÷) [a…z] ([a…z]|[0…9])* (")"|ε) )*
```

- This regexp matches both "a+b×c" and "(a+b)×c."
  - it will match ***any*** correctly parenthesized expression over variables and the four operators in the regexp
- Unfortunately, it also ***matches many syntactically incorrect expressions***
  - such as "a+(b×c" and "a+b)×c)."
- ***We cannot write a regexp matching all expressions with balanced parentheses:*** *"DFAs cannot count"*

Context-Free Grammars

- We need a more powerful notation than regular expressions
  - …that still leads to efficient recognizers
- Traditional solution: use a ***context-free grammar*** (CFG)
  - ***grammar*** G:
    set of rules that describe how to form sentences
  - ***language*** *L*(G) defined by G:
    collection of sentences that can be derived from G
- Example: consider the following grammar SN

```
SheepNoise → baa SheepNoise
           | baa
```

  - each line describes a ***rule*** or ***production*** of the grammar

Context-Free Grammars

Syntax analysis

```
SheepNoise → baa SheepNoise
           | baa
```

- The first rule *SheepNoise* → **baa** *SheepNoise* reads:
  "*SheepNoise* can derive the word baa followed by more *SheepNoise*"
- *SheepNoise* is a ***syntactic variable*** representing the set of strings that can be derived from the grammar

  written in *italics*

  - We call these syntactic variables "***nonterminal symbols***" **NT**
    Each word in the language defined by the grammar (**baa**) is a "***terminal symbol***"

    written in **bold letters**

"|" can be read as "OR": the parser can choose either the first or the second rule

- The second rule reads:
  "*SheepNoise* can also (|) derive the string baa"
  - The "|"-notation is a shorthand to avoid writing two separate rules:

```
SheepNoise → baa SheepNoise
SheepNoise → baa
```

Grammars and languages

Syntax analysis

```
SheepNoise → baa SheepNoise
           | baa
```

- Can we figure out which sentences can be derived from a grammar G?
  - i.e., what are valid sentences in the language *L*(G)?
- First, identify the ***goal symbol*** or ***start symbol*** of G
  - represents the *set of all strings* in *L*(G)
  - thus, it cannot be one of the words in the language
- Instead, it must be one of the nonterminal symbols introduced to add structure and abstraction to the language
  - Since our grammar SN has only one nonterminal, `SheepNoise` must be the start symbol
-

Grammars and languages

Syntax analysis

start here

```
1. SheepNoise → baa SheepNoise
2.            | baa
```

- Deriving a sentence:
  - start with a prototype string that contains just the start symbol, *SheepNoise*
  - pick a nonterminal symbol, α, in the prototype string
  - choose a grammar rule, $\alpha \rightarrow \beta$
  - and rewrite (replace) α with β
- Repeat until the prototype string contains no more nonterminals
  - the string then consists entirely of words (terminal symbols)

  ⇒ it is a sentence in the language

    - every version of the prototype string that can be derived is called a ***sentential form***

Grammars and languages

Syntax analysis

start here

```
1. SheepNoise → baa SheepNoise
2.             | baa
```

- Examples:

Rule	Sentential form
	SheepNoise
2	**baa**

Rewrite with rule 2

Rule	Sentential form
	SheepNoise
1	**baa** *SheepNoise*
2	**baa baa**

Rewrite with rule 1, then rule 2

- Rule 1 lengthens the string while rule 2 eliminates the NT *SheepNoise*
- The string can never contain more than one instance of *SheepNoise*
- All valid strings are derived by >= 0 applications of rule 1, followed by rule 2
- Applying rule 1 *k* times followed by rule 2 generates a string with *k*+1 **baa**s.

A more useful example…

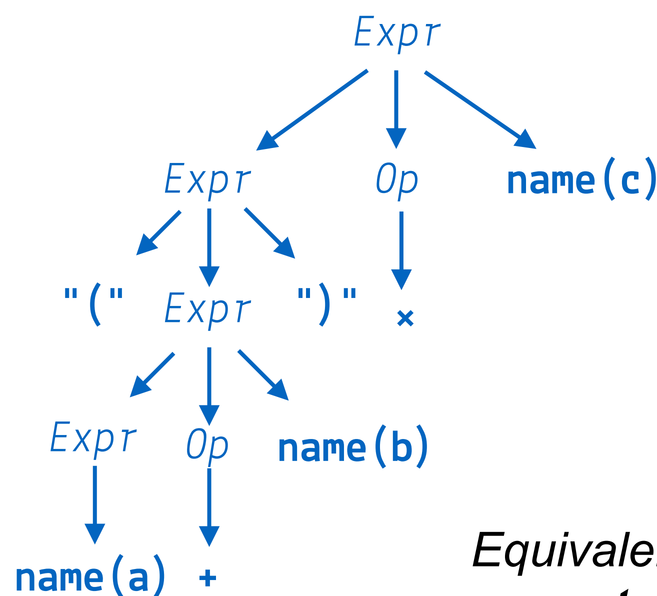
Syntax analysis

we added **rule numbers**, these are **not** part of the grammar

```
1. Expr → "(" Expr ")"
2.        | Expr Op name
3.        | name
4. Op   → +
5.        | -
6.        | ×
7.        | ÷
```

Rule	Sentential form
	Expr
2	*Expr Op* **name**
6	*Expr* × **name**
1	"(" *Expr* ")" × **name**
2	"(" *Expr Op* **name** ")" × **name**
4	"(" *Expr* + **name** ")" × **name**
3	"(" **name** + **name** ")" × **name**

Rightmost derivation of "(a + b) × c"



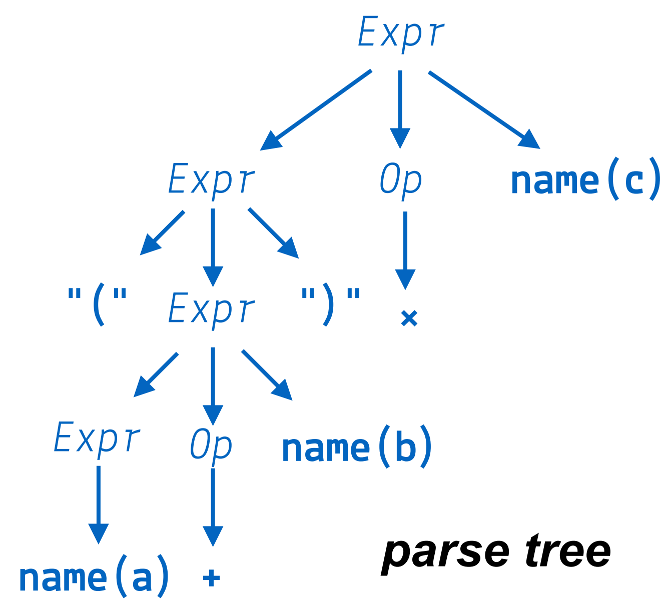
Equivalent ***parse tree***

A more useful example…

Syntax analysis

- This simple context-free grammar for expressions **cannot generate** a sentence with **unbalanced or improperly nested parentheses**
  - Only rule 1 can generate an open parenthesis; it also generates the matching close parenthesis
- Thus, it cannot generate strings such as “a+(b×c” or “a+b)×c)”
  - a parser built from the grammar will not accept such strings
- Context-free grammars allow to specify constructs that regexps do not

```
1. Expr → "(" Expr ")"
2.        | Expr Op name
3.        | name
4. Op   → +
5.        | -
6.        | ×
7.        | ÷
```



parse tree

Order of derivations

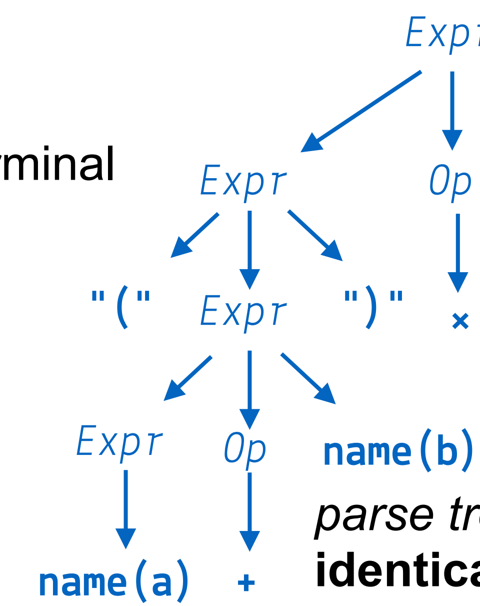
```
1. Expr → "(" Expr ")"
2.        | Expr Op name
3.        | name
4. Op    → +
5.        | -
6.        | ×
7.        | ÷
```

Rightmost:
rewrite, at each step, the rightmost nonterminal

Rule	Sentential form
	Expr
2	*Expr Op* **name**
6	*Expr* × **name**
1	"(" *Expr* ")" × **name**
2	"(" *Expr Op* **name** ")" × **name**
4	"(" *Expr* + **name** ")" × **name**
3	"(" **name** + **name** ")" × **name**

Leftmost: rewrite, at each step, the leftmost nonterminal

Rule	Sentential form
	Expr
2	*Expr Op* **name**
1	"(" *Expr* ")" *Op* **name**
2	"(" *Expr Op* **name** ")" *Op* **name**
3	"(" **name** *Op* **name** ")" *Op* **name**
4	"(" **name** + **name** ")" *Op* **name**
6	"(" **name** + **name** ")" × **name**



parse tree
identical for both!

Ambiguity of grammars

Syntax analysis

- For the compiler, it is important that each sentence in the language defined by a context-free grammar has a ***unique*** rightmost (or leftmost) ***derivation***
- A grammar in which multiple rightmost (or leftmost) derivations exist for a sentence is called an ***ambiguous grammar***
  - it can produce multiple derivations and multiple parse trees
- Multiple parse trees imply ***multiple possible meanings for a single program!*** ⚡

Ambiguity of grammars: example

Syntax analysis

"***dangling else***"-problem in ALGOL-like languages (e.g. PASCAL)

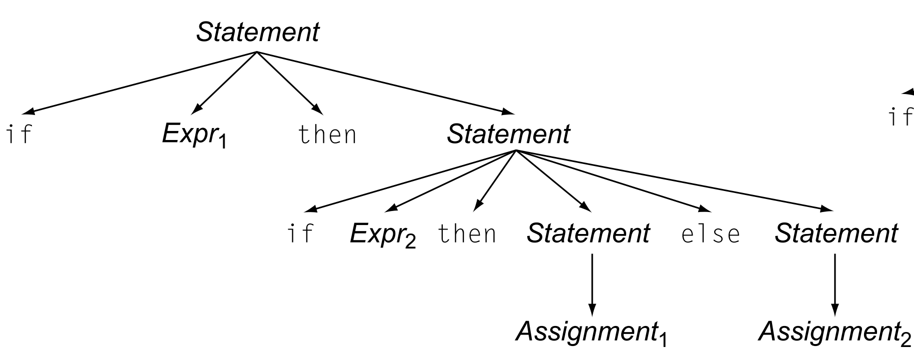
```
1.   Statement → if Expr then Statement else Statement
2.             | if Expr then Statement
3.             | Assignment
4.             | …other statements…
```

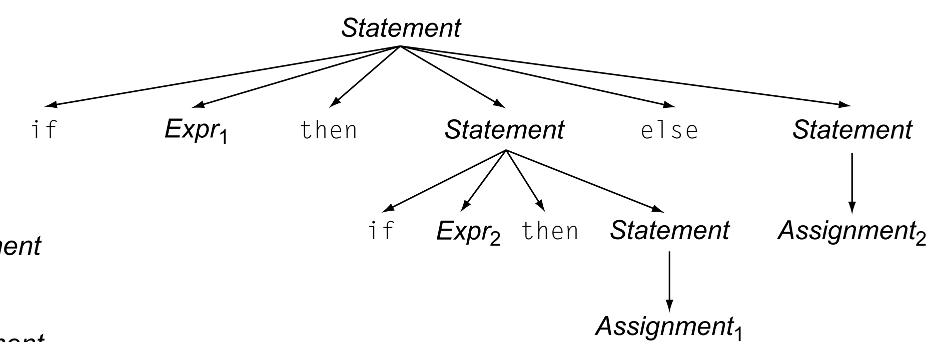
"else" part is optional

This statement

```
if Expr1 then if Expr2 then Assignment1 else Assignment2
```

has two distinct rightmost derivations with different behaviors:





Removing ambiguity

We can modify the grammar to include a rule that determined which **if** controls an **else**:

```
1.   Statement → if Expr then Statement
2.             | if Expr then WithElse else Statement
3.             | Assignment
4.   WithElse  → if Expr then WithElse else WithElse
5.             | Assignment
```

This solution restricts the set of statements that can occur in the **then** part of an **if-then-else** construct

- It ***accepts the same set of sentences*** as the original grammar
- but ensures that each else has an unambiguous match to a specific if

Removing ambiguity: example

Syntax analysis

The modified grammar has only one rightmost derivation for the example

```
1. Statement → if Expr then Statemen
2.           | if Expr then WithElse else Statement
3.           | Assignment
4. WithElse  → if Expr then WithElse else WithElse
5.           | Assignment
```

```
if Expr1 then if Expr2 then Assignment1 else Assignment2
```

Rule	Sentential form
	Statement
1	**if** *Expr* **then** *Statement*
2	**if** *Expr* **then** **if** *Expr* **then** *WithElse* **else** *Statement*
3	**if** *Expr* **then** **if** *Expr* **then** *WithElse* **else** *Assignment*
5	**if** *Expr* **then** **if** *Expr* **then** *Assignment* **else** *Assignment*

Addendum: Backus-Naur-Form

- The traditional notation to represent a context-free grammar is called ***Backus-Naur form*** (BNF) [1]
  - BNF denotes nonterminal symbols by wrapping them in angle brackets, like ⟨SheepNoise⟩
  - Terminal symbols are <u>underlined</u>.
  - The symbol ::= means "derives," and the symbol | means "also derives"
- In BNF, the sheep noise grammar becomes:

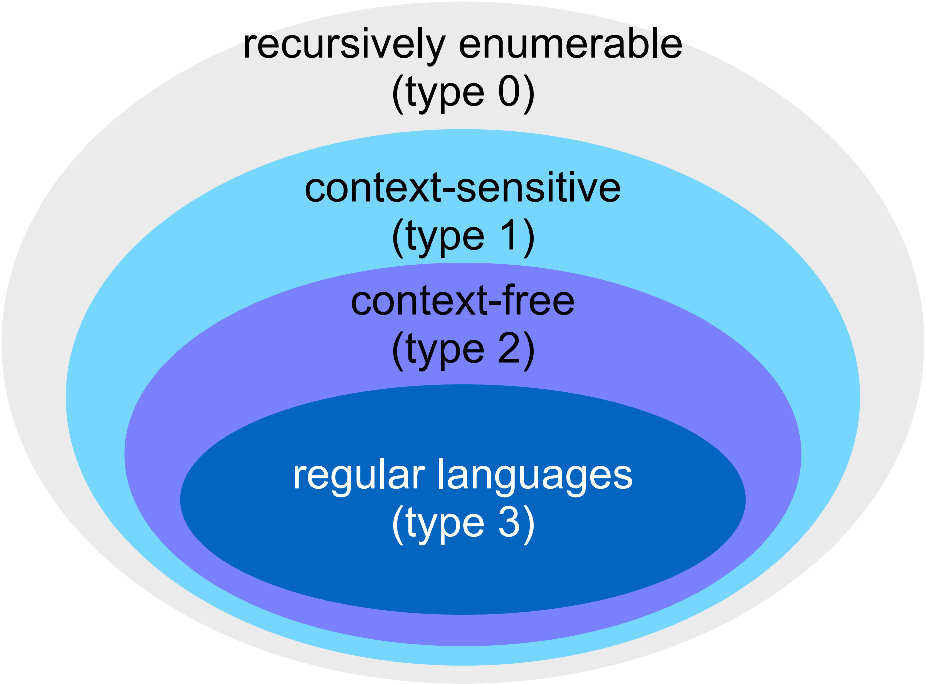
```
⟨SheepNoise⟩ ::= baa ⟨SheepNoise⟩
              |  baa
```

- This is equivalent to our grammar SN
  - …and was easier to typeset in the 1950’s 😉

Addendum: Types of languages

Syntax analysis

- Noam Chomsky (*1928):
  American linguist, philosopher, cognitive scientist, historian, social critic, and political activist
- The ***Chomsky hierarchy*** is a containment hierarchy of classes of formal grammars [2]
- Defines four types (0–3) of languages with increasing complexity from regular languages to recursively enumerable
- Accordingly, recognizing the language requires a successively more complex method



References

[1] P. Naur (Ed.), J.W. Backus, F.L. Bauer, J. Green, C. Katz, J. McCarthy, et al.:
Revised report on the algorithmic language Algol 60,
Commun. ACM 6 (1) (1963) 1–17

[2] Noam Chomsky, Marcel P. Schützenberger:
The algebraic theory of context free languages,
In Braffort, P.; Hirschberg, D. (eds.). Computer Programming and Formal Languages
Amsterdam: North Holland. pp. 118–161, 1963