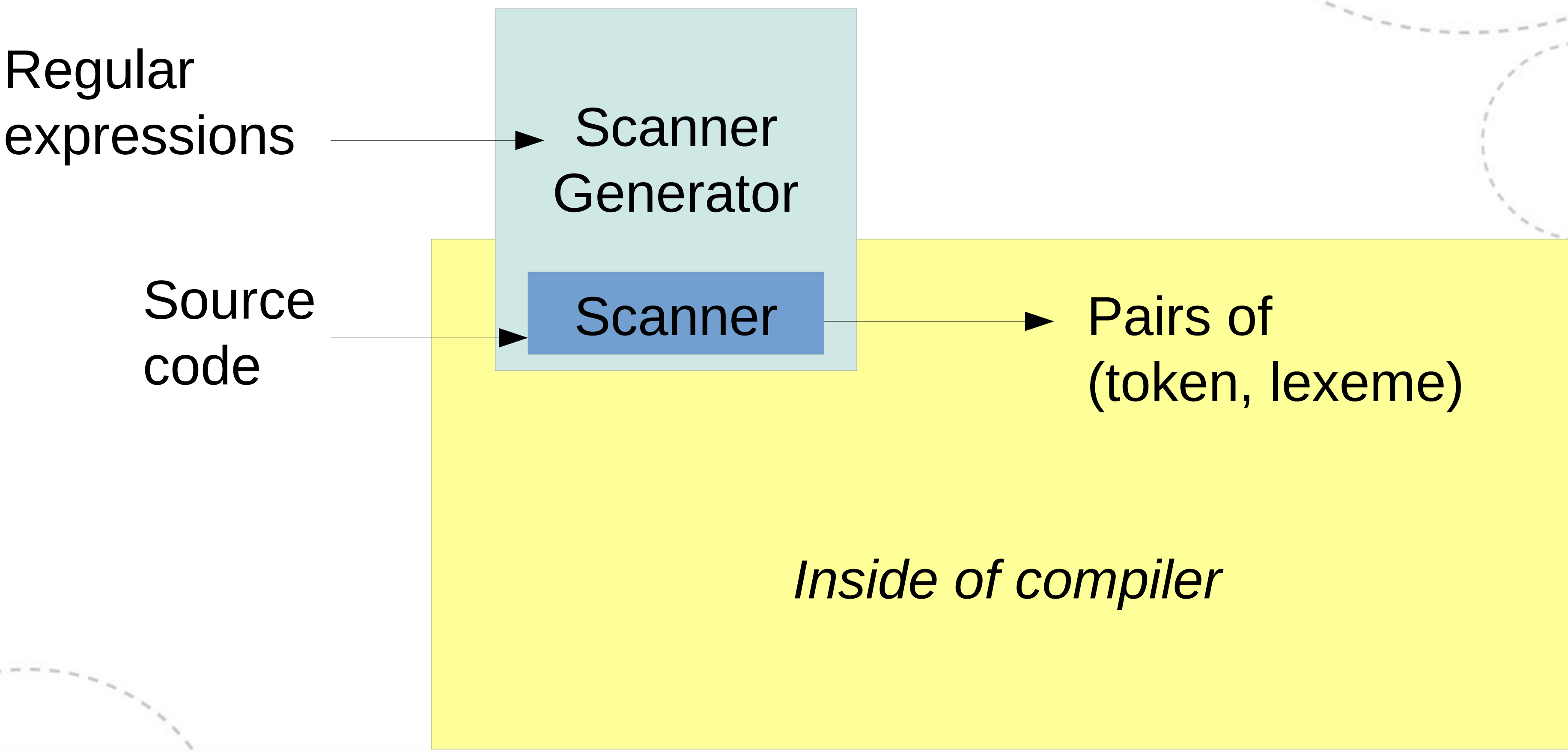
NTNU – Trondheim
Norwegian University of
Science and Technology

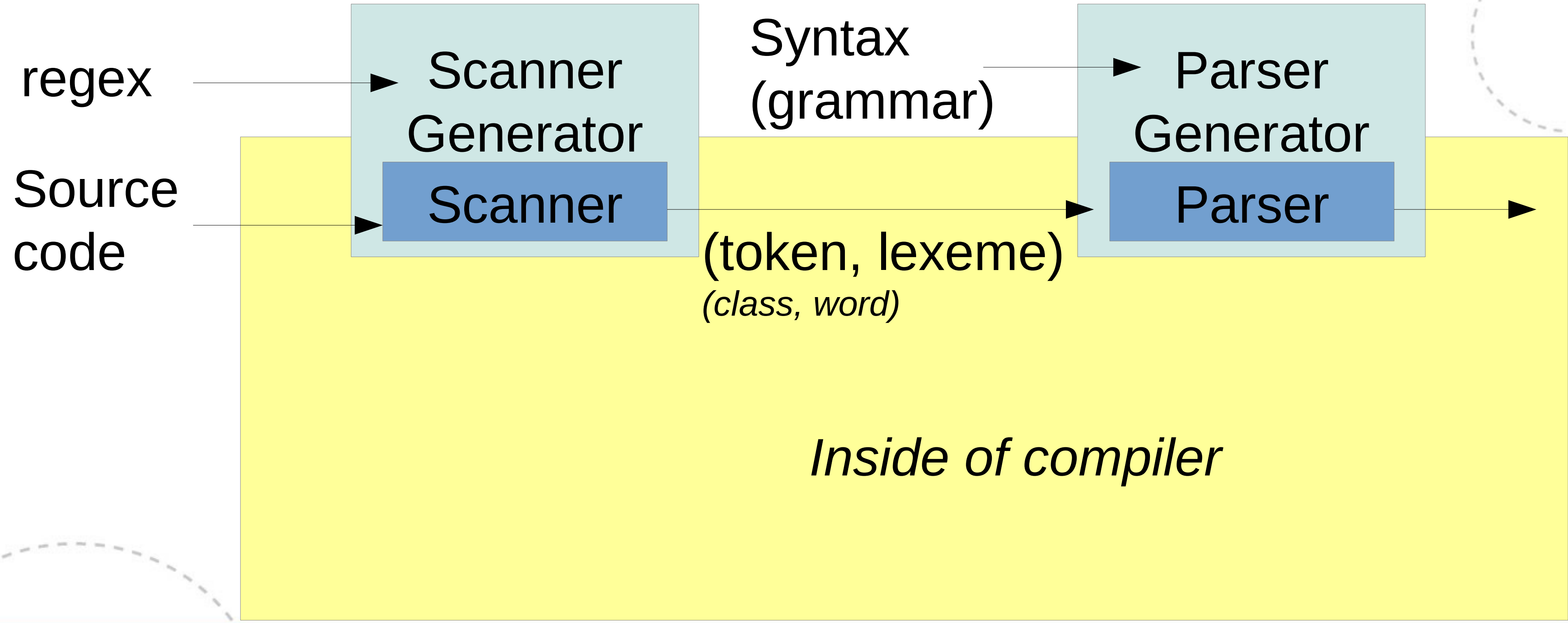
Context-Free Grammars

We’ve recognized the words



Next comes statements

- That is, *syntactic analysis*
  – Are words of the right kinds appearing in correct order?



Grammar, in writing

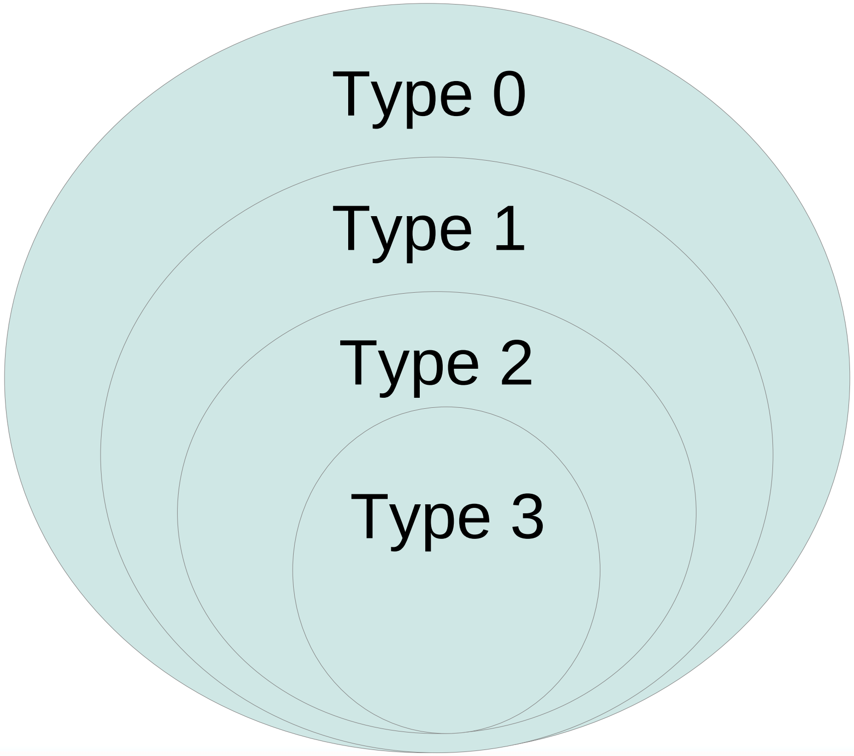
- In order to pull the same trick again, we need to write down syntax rules in a format that a generator can work with
- That is, we need a specification of what kinds of words can follow each other in a number of different orders
- Plain automata have trouble with the whole “a number of different orders” thing
  - They only remember what state they are in, and only implicitly represent what they have seen so far

That’s correct!

- Verifying what a “correct statement” is can be subject to a lot of different constraints
  - **“I came to work this morning, and sat down”** is an instance of *<u>pronoun verb preposition noun pronoun noun conjunction verb preposition</u>*
  - **“I came to work this morning, or sit into”** is the exact same pattern, but it is wrong because the verbs switch from past to infinitive, and the final preposition isn’t connected to a place
  - **“Colorless green ideas sleep furiously”** is a classic example that a *syntactically* correct statement can be without *semantic* meaning

How far we can take it

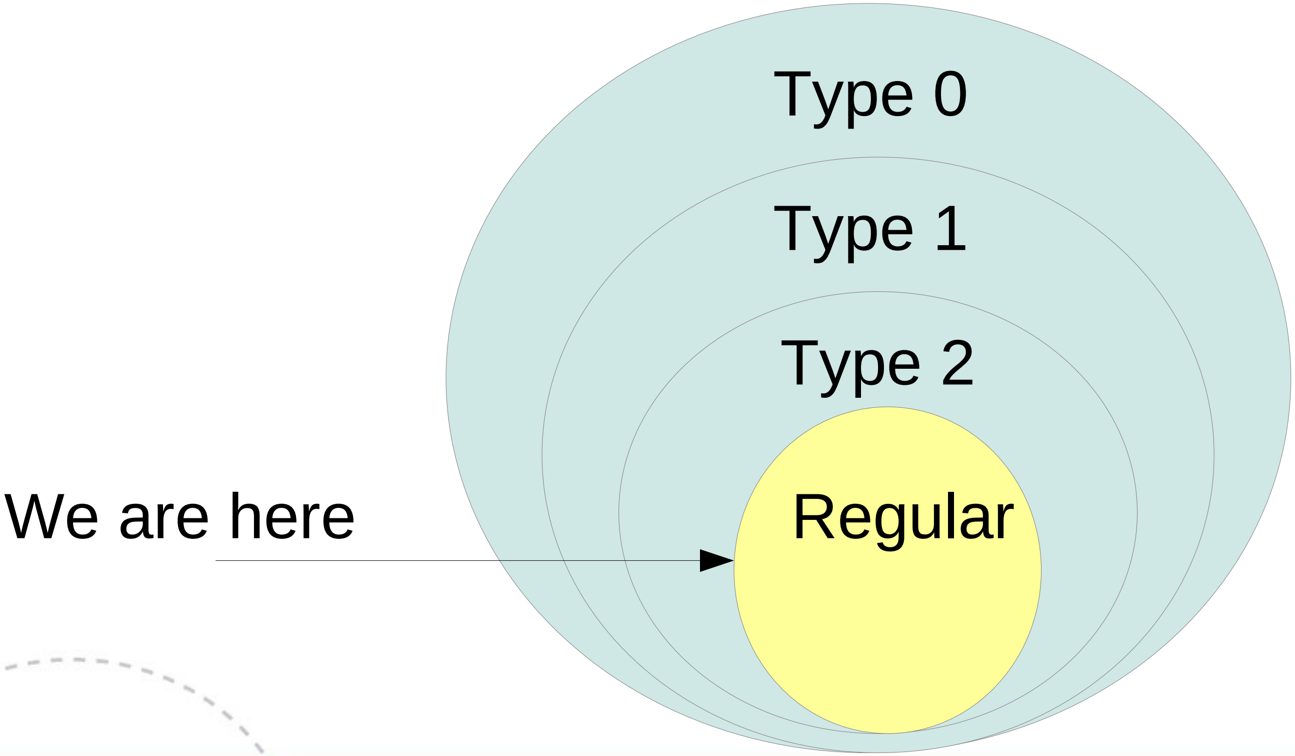
- This is the *Chomsky hierarchy,* which relates types of grammars to each other
  - Each successive type adds restrictions, making it a more specific sub-type



NTNU – Trondheim
Norwegian University of Science and Technology

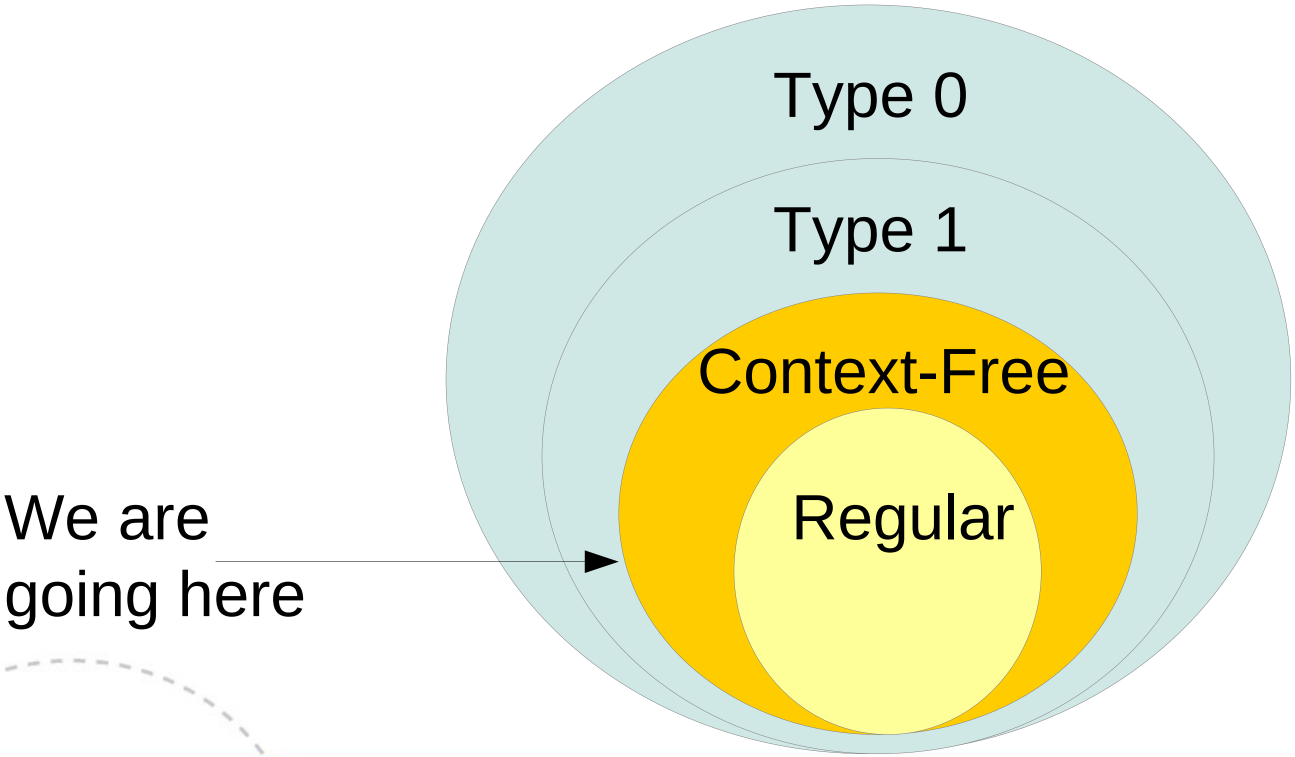
The most specific type

- Type 3 are the regular languages, recognizable by finite state automata



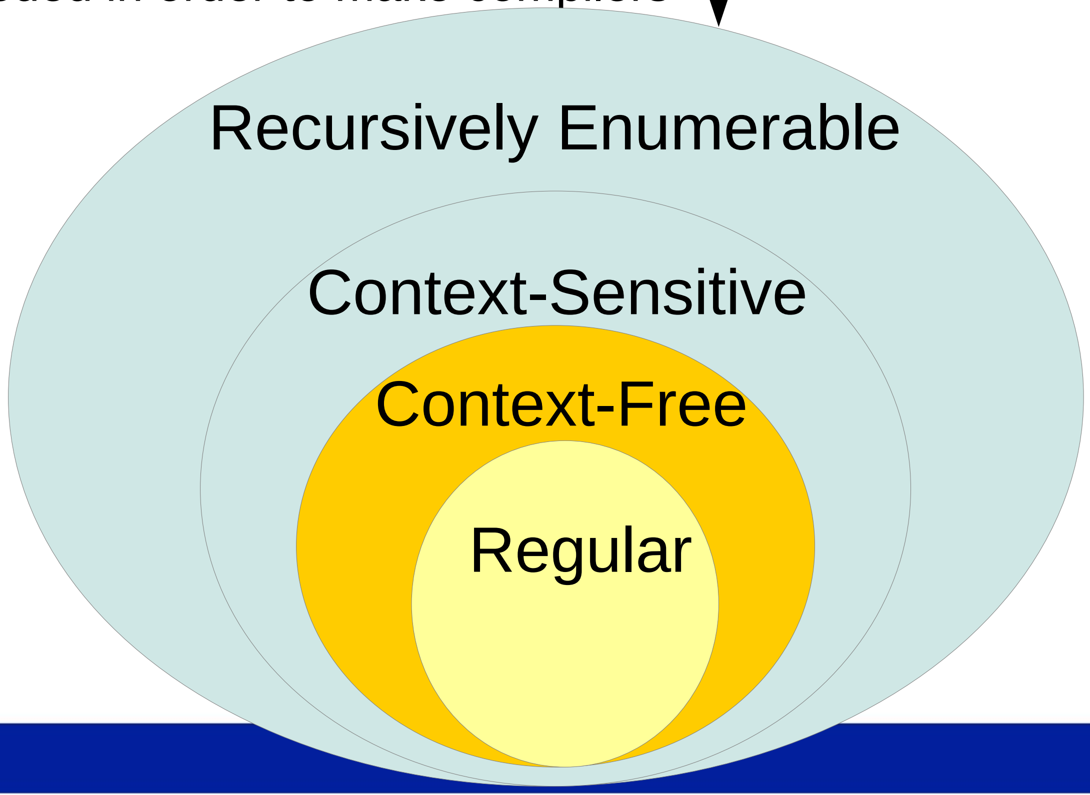
Slightly less specific

- Type 2 are the Context-Free grammars, recognizable by stack machines



All the way

- Curriculum-wise, we stop there and fix up contextual information later
  - I hope to say something about Type 0 on a rainy day, but it's not needed in order to make compilers



Production rules

- A production rule is an intermediate form of a statement, containing placeholders that must be substituted with words
- The rules
  1) $A \rightarrow w\ B\ z$
  2) $B \rightarrow x$
  3) $B \rightarrow y$

  describe the language of strings {“wxz”, “wyz”}

  $A \rightarrow w\ B\ z \rightarrow w\ x\ z$ (Rule 1, then rule 2)

  $A \rightarrow w\ B\ z \rightarrow w\ y\ z$ (Rule 1, then rule 3)

Terminals, non-terminals and derivations

- The placeholders are *non-terminals*
  - If there are any left in an intermediate statement, it's not yet a statement
  - They're usually capitalized
- The words are *terminals*
  - A source code can contain any string of terminals, whether or not they are a syntactically correct program
  - They're usually in lowercase
- The process of starting from grammar rules and constructing a string of terminals is a *derivation*
  - If there is a derivation that leads to a string of terminals matching the token stream from a source code, the program adheres to the grammar that derived it
  - That's how we do syntax analysis

More formally

- *Terminals* are the basic symbols that form strings
  - *cf*. “alphabet” from regex
- *Nonterminals* are syntactic variables that represent sets of strings
- One nonterminal is the *start symbol*
  - Derivations begin with it
  - If nothing else is stated, we take the first nonterminal listed
- *Productions* specify combinations of substitutions, and contain
  - A *head* nonterminal on the left hand side
  - An arrow ‘→’ *(or some other symbol to separate left from right)*
  - A *body* of terminals and/or nonterminals that describe how the head can be constructed

For brevity

- Beyond tiny and trivial ones, most grammars contain a great(-ish) number of productions

  Statement → If-Statement

  Statement → For-Statement

  Statement → Switch-Statement

  Statement → While-Statement

  Statement → Assignment-statement

  Statement → FunctionCall-Statement

  etc. etc.

- To save some ink,

  A → a

  A → b

  A → c

  abbreviates to

  A → a | b | c

  (but they are still 3 distinct productions)

NTNU – Trondheim
Norwegian University of
Science and Technology

Representative grammars

- Fragments of grammars can be used to study particular aspects of a language without recognizing the whole thing
- For this purpose, it’s nice to mock up tiny grammars where the nonterminals we’re not interested in just become a simple terminal that represent *‘something goes here, but we don’t care now’*
- It’s easier to manipulate grammars when you can prune away some of the many, many combinations of things they usually admit

E.g.: nested while statements

- For instance, somewhat realistic rules might say

  Statement → Assignment | Function | If-Statement | …

  Condition → Boolean-Expression

  Boolean-Expression → true | false | Expr BoolOperator Expr

  Statement → while Condition do Statement endwhile

- If we only care about the nesting of while statements, it's shorter to read

  S → w C d S e | s

  C → s

  so we can derive

  S → w C d S e → w C d w C d S e e → w c d w C d S e e → w c d w c d S e e → **w c d w c d s e e**

  for a once-nested construct, never mind what 'c' and 's' represent.

Shortening derivations

- These steps don't add much to the discussion either:

  S → w C d S e → **w C d w C d S e e → w c d w C d S e e** → w c d w c d S e e → w c d w c d s e e

  so we can write

  S → w C d S e →* w c d w c d S e e

  to get rid of the C-s in one go, and read

  – "w C d S e derives w c d w c d S e e in some number of steps"

- We could also assert

  S →* w c d w c d s e e

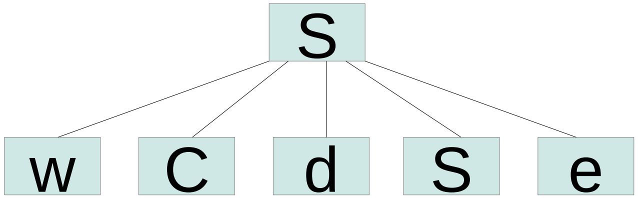
  to say that the statement is part of the language, but then we have omitted the whole derivation which proves it is really so

Syntax trees

- Nonterminals can be substituted in any order
  - The language contains all variations, except that we have to start from the start symbol
- The order we choose to substitute them in implies an ordered hierarchy of which ones we prioritize
  - Things that have an ordering can be drawn as graphs
- Taking the nested while grammar fragment,

  $S \rightarrow w\ C\ d\ S\ e$

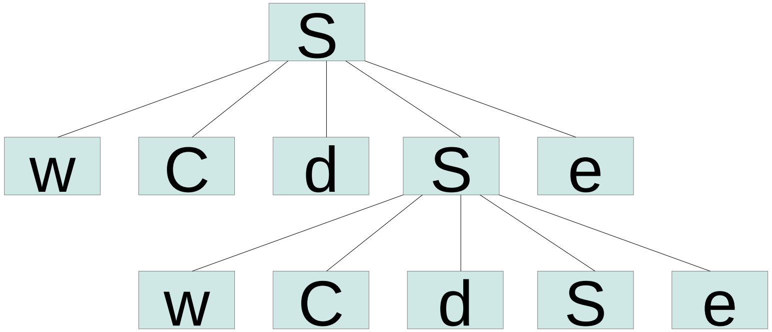
  means S is substituted first, so we get a tree like this



Moving on

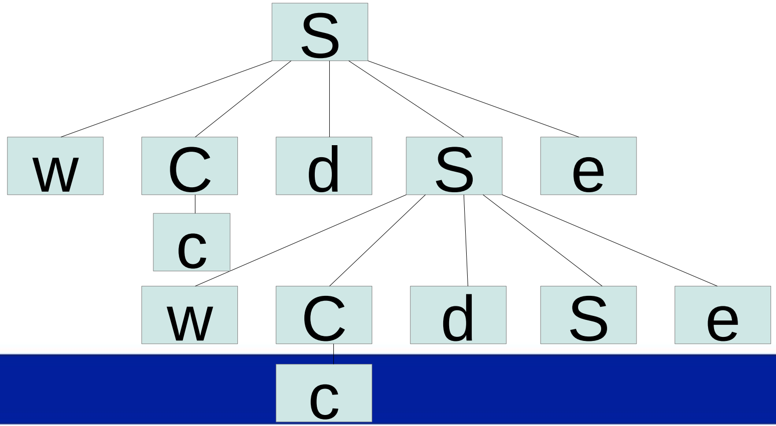
- Next, we can substitute the new S...

S → w C d S e → w C d <u>**w C d S e**</u> e



and get rid of the c-s

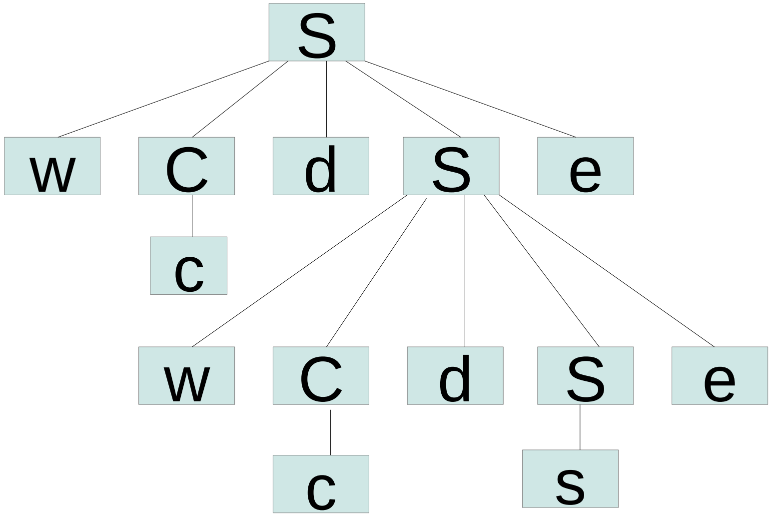
w C d w C d S e e →* w c d w c d S e e



NTNU – Trondheim
Norwegian University of
Science and Technology

and finally, the last S → s

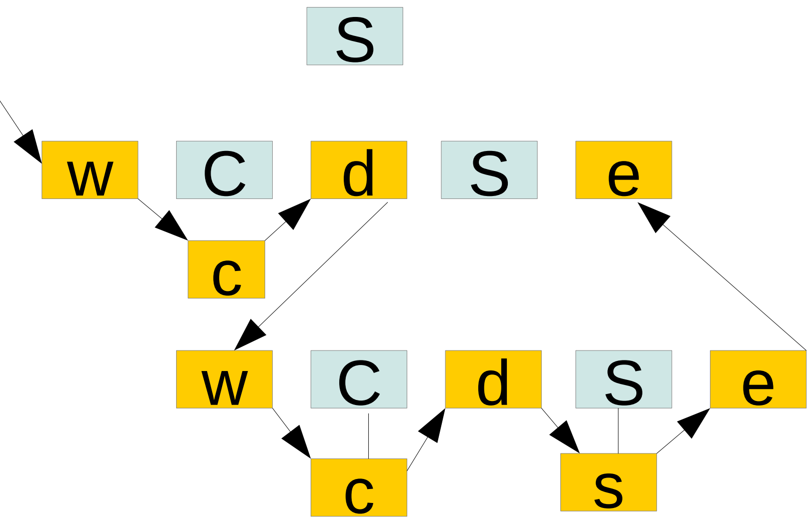
- That derivation gave us this *syntax tree*



- Graphs derived in this manner will always become trees, because every substitution only introduces nodes on the next level of the hierarchy

Notice how the leaves spell out the statement

- w c d w c d s e e



- It’s an observation we will make again

  Just sayin’

NTNU – Trondheim
Norwegian University of Science and Technology

Does the order really matter?

- Yup. Consider this grammar for if-statements:

  S → ictS | ictSeS | s

  Read right hand sides as

  “if condition then statement”,

  “if condition then statement else statement”,

  “statement”

  and derive

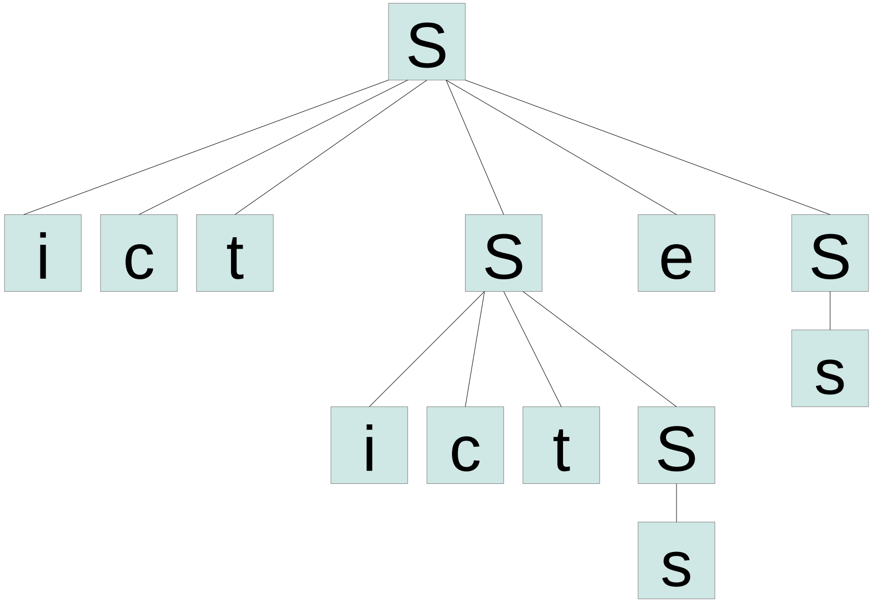
  S → ict **S** eS → ict **ictS** eS →* ict icts es (“ictictses” is ok)

  S → ict**S** → ict **ictSeS** → ict ictses (“ictictses” is ok)

Syntax tree for derivation #1

S → ict **S** eS → ict **ictS** eS →* ict icts es

gives us

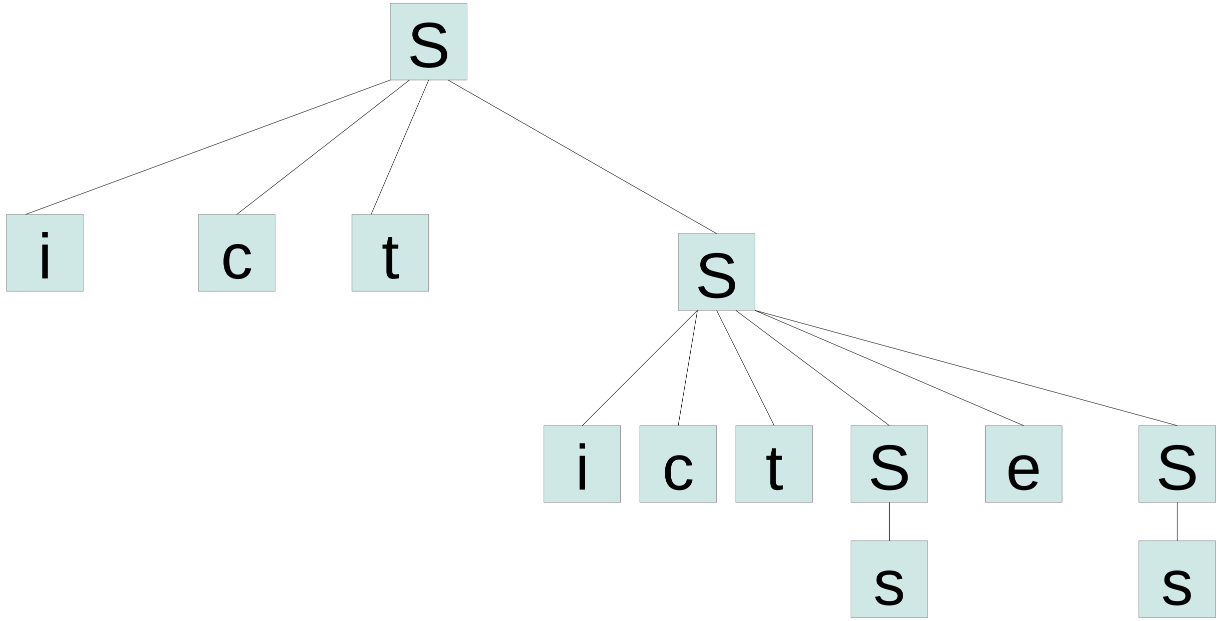


NTNU – Trondheim
Norwegian University of
Science and Technology

Syntax tree for derivation #2

S → ict**S** → ict **ictSeS** → ict ictses

gives us



Who cares?

- `if (x<10) then if (x>4) then “5-9” else “0-4”` can read

```
if (x<10) then
    if ( x>4 ) then “5-9”
    else “0-4”          /* Run when x is smaller than ten and not greater than 4 */
```

Tree #2

alternatively,

```
if (x<10) then
    if ( x>4) then “5-9”
else “0-4”          /* Run when x is not smaller than ten */
```

Tree #1

- Tree/derivation #1 is “wrong”, but syntactically, these are equally good

Ambiguous grammars

- A grammar is *ambiguous* when it admits several syntax trees for the same statement
- This was the “dangling-else ambiguity”
  - famous because if statements are such a basic part of a language
- These are of no use to us, they must be fixed
  - One way is to creatively re-write the grammar so that the problem disappears without altering the language
  - Another way is to assign priorities to the productions

(For the dangling else, and all its dangling head-reappears-at-the-end friends among productions, I personally like to introduce an “endif” delimiter)

Parsing

- There are two very intuitive ways to systematically select nonterminals for substitution
  - Take the leftmost one
  - Take the rightmost one
- Systematically deriving a statement (if it's valid) is what a syntax analyzer (parser) does
  - It's easiest to make one if you have simple rules like this to follow
  - Choosing a rule gives you only one syntax tree for any given statement
  - If we're going to say that the parser recognizes the language of the grammar, the one tree we get has to be the *only* tree

Left factoring

- Parsers, like scanners, can only see so far ahead
- If we have productions

  A → abcdef X gh | abcdef Y gh

  and the parser only has space to buffer one token, it can't choose between these two productions
- As with NFA→DFA conversion, if we can postpone the decision until it makes a difference, that works

  Rewriting the grammar as

  A → abcdef A'

  A' → X gh | Y gh

  preserves the language by adding 1 production to collect a common prefix shared by several other productions

Left recursion

- This could be a convenient grammar for a list of items

  A → A a | a

  it derives

  A → a

  A → A a → a a

  A → A a → A a a → a a a

  ...and so on…

- The production A → A a is *left recursive*, the head reappears on the left side of the body

Equivalently

- Another way to get lists of a-s could be

  A → a A’

  A’ → a A’ | ε

  (the empty string returns!)

  it derives

  A → a

  A → a A’ → a a A’ → a a

  A → a A’ → a a A’ → a a a A’ → a a a

  ...and so on...

Elimination of left recursion

- If a nonterminal has *m* productions that are left recursive and *n* productions that aren't

  $A \rightarrow A\ \alpha_1 \mid A\ \alpha_2 \mid A\ \alpha_3 \mid \ldots \mid A\ \alpha_m$

  $A \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots \mid \beta_n$

  (Greek letters symbolize any ol' combination of other [non-]terminals)

  introducing A' and rewriting it as

  $A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \ldots \mid \beta_n A'$

  $A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$

  preserves the language, and removes (immediate) left recursion

  "Immediate" because l.r. can also happen in several steps, like when productions

  $A \rightarrow B\ x$ and $B \rightarrow A\ y$

  gives $A \rightarrow B\ x \rightarrow A\ y\ x$

  so that A returns on the left of a derivation from A

In summary

- At this point, we've met
  - Context-Free Grammars, their derivations and syntax trees
  - Ambiguous grammars, and mentioned that there's no single, true way to disambiguate them (it depends on what we want them to represent)
  - Left factoring, which always shortens the distance to the next nonterminal
  - Left recursion elimination, which always shifts a nonterminal to the right

What lies ahead

- Left factoring and treating left recursion may not be obviously useful, but you might as well commit them to memory right away
- We will make use of these grammar-fixing rules next time, when we look at how to make parsers that derive by always picking nonterminals from the left