NTNU – Trondheim
Norwegian University of
Science and Technology

Top-down parsing and LL(1) parser construction

Parsing by recursive descent

- Take this grammar, it models ifs and whiles:

    P → iCtSz | iCtSeSz | wCdSz
    C → c
    S → s

- Let’s parse the statement **‘ictsesz’**
- In top-down parsing, our starting point is the start symbol, we need to choose a production

P

- LL(1) parsing means
    - **L** eft-to-right scan
    - **L** eftmost derivation (*i.e. always expand leftmost nonterminal)*
    - **1** symbol of lookahead (this must be enough to select a production)

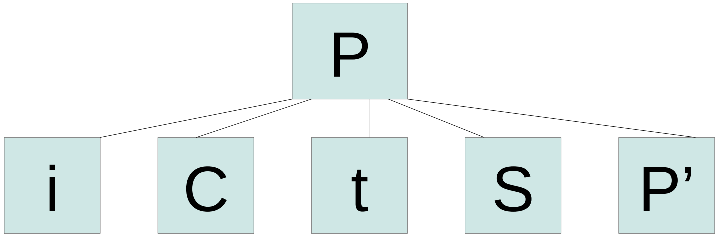
We can’t choose

- If we look ahead 1 token and find ‘i’, we get two productions to choose from
  - P → iCtSz
  - P → iCtSeSz
- There is no way to make this choice before seeing more of the token stream
- Left factoring (previous lecture) to the rescue!
- The grammar becomes
  - P → iCtSP’ | wCdSz
  - P’ → z | eSz
  - C → c
  - S → s

One step ahead

- Now that there's only one production which expands P on 'i', we can take it when we see 'i'

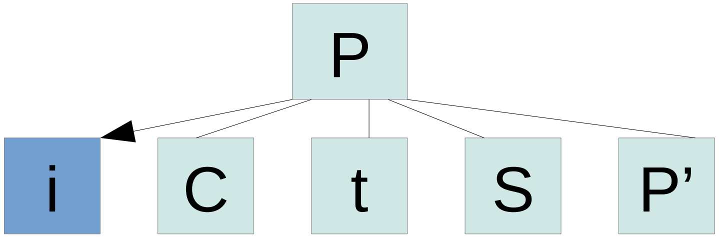
  P → iCtSP'



- ...and expand the parse tree according to the derivation

Moving along

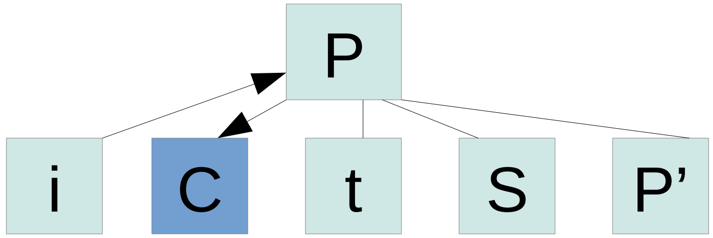
- *Recursive descent* means we follow the children of a tree node through to the bottom, where there must be a terminal.
  - The step we chose predicted that iCtSP’ is coming up, we’re looking at the ‘i’ in ‘ictsesz’
  - Following through to the first child...



...it’s an ‘i’! That matches, throw it away, we now have ‘ctsesz’ left to parse.

Backtrack, and repeat

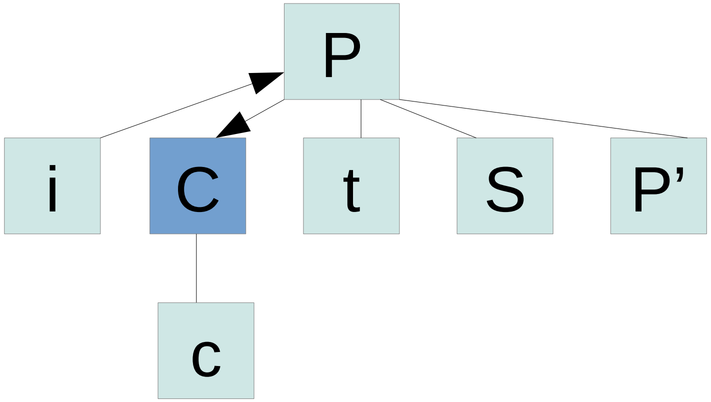
- Leaving that behind, the next child in the tree is a nonterminal
  - That can’t match any input, so we need to pick a production again



NTNU – Trondheim
Norwegian University of
Science and Technology

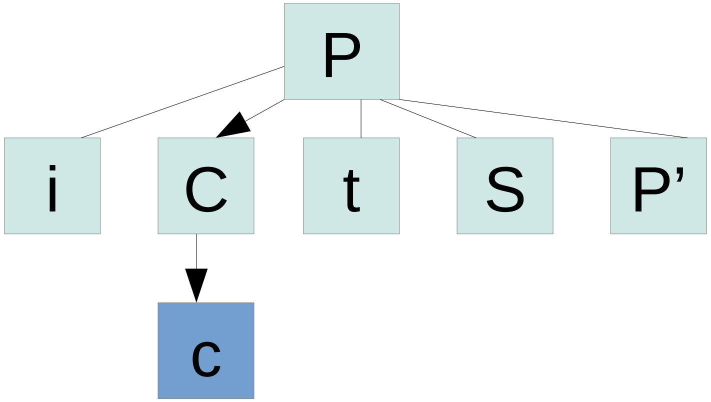
Pick the next production

- There's not a lot of choice on how to expand C, so it could be clear already
  - Nevertheless, look at the input ‘ctsesz’, lookahead is now ‘c’
  - Pick production C → c, and expand the tree accordingly



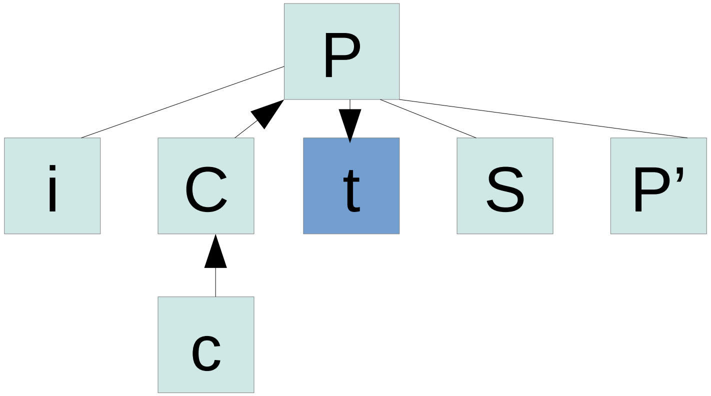
Verify another terminal

- We need to go all the way to the bottom before backtracking...
  - ...but we find the ‘c’ that was expected there
  - Away it goes, remaining input is **‘tsesz’**



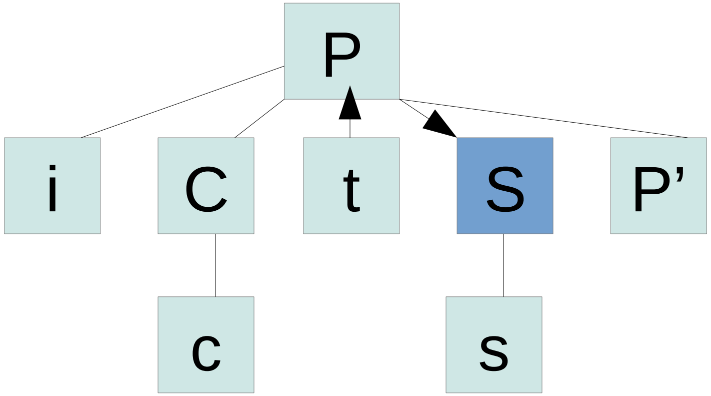
‘t’ disappears as well

- It was already predicted by the first production:
  - Toss it out, ‘sesz’ remains

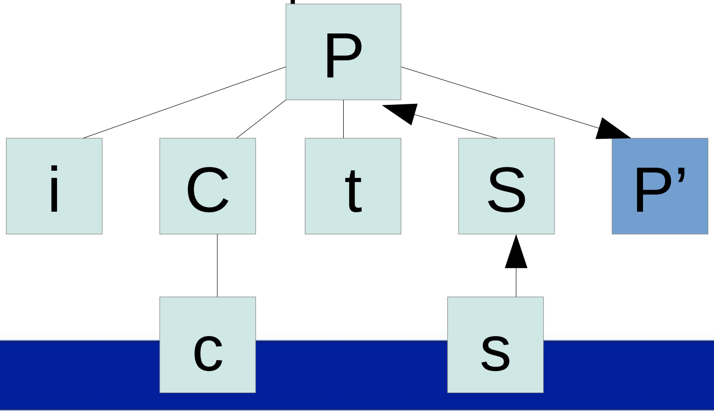


The next nonterminal is S

- Lookahead character ‘s’ drives the choice of S → s



  - Verify ‘s’, leave ‘esz’ and proceed to P’



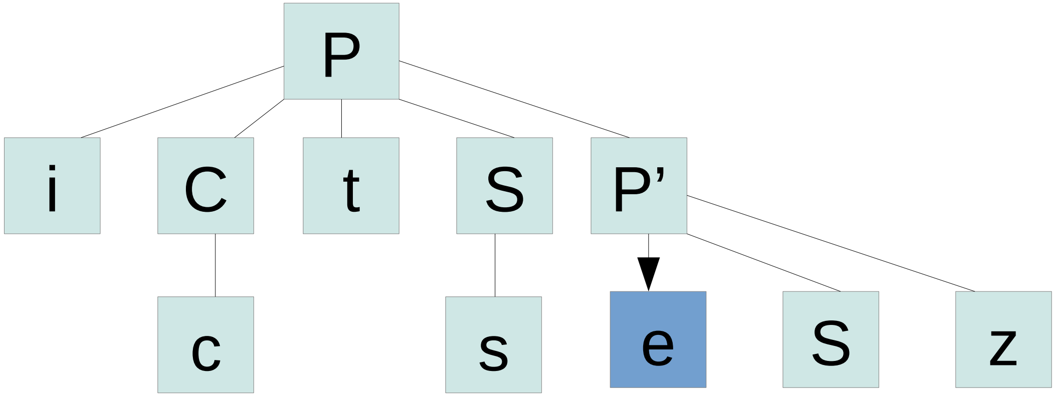
There is a choice here

- P’ expands in two ways

  P’ → z

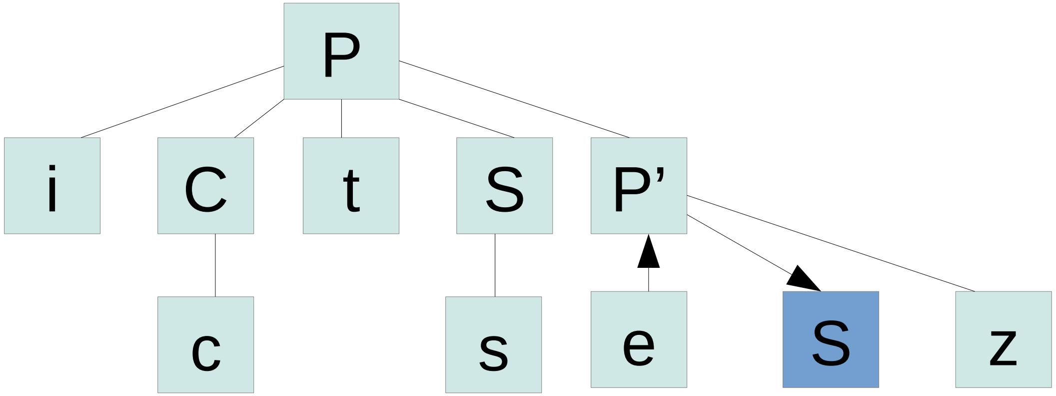
  P’ → eSz

  – This is our postponed selection, we can choose now because the lookahead symbol (‘e’ from remaining ‘esz’) tells us we need alternative #2:



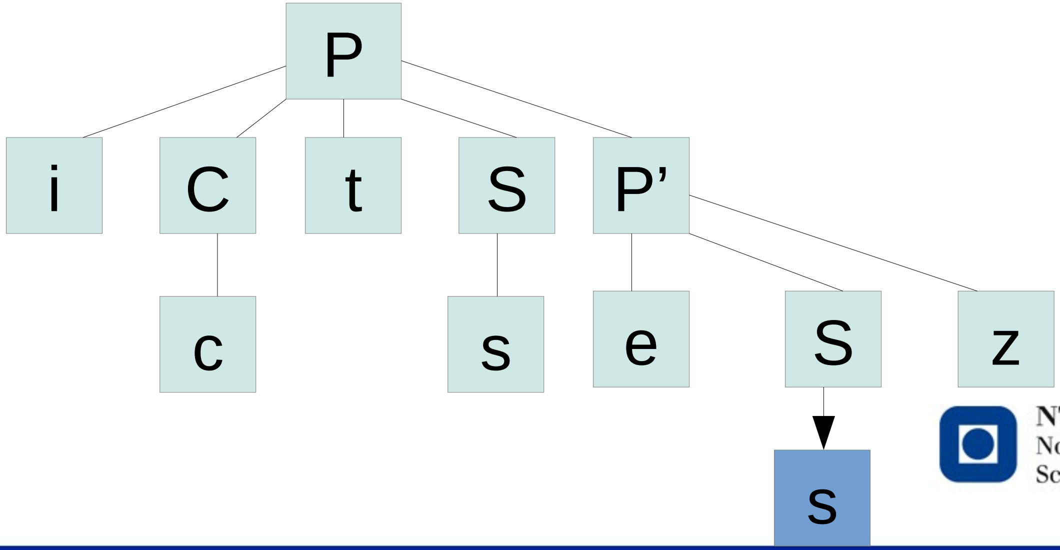
Continue in the same way

- You'll have to
  - Verify 'e', and backtrack (leaving 'sz' on input)



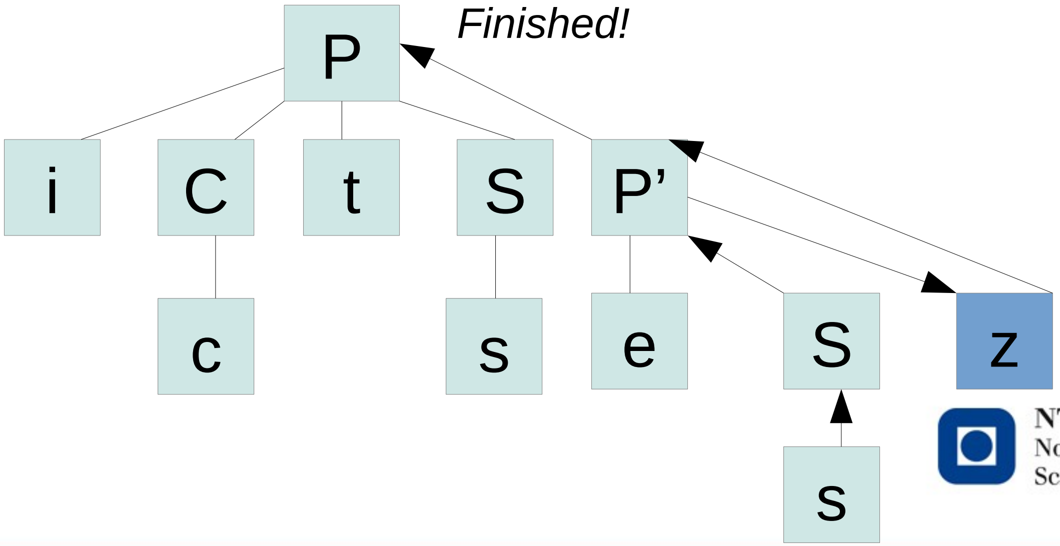
Continue in the same way

- You'll have to
  - Verify 'e', and backtrack (and leave 'sz' on input)
  - Expand another S → s, verify the terminal (leaving 'z' on input)



The statement is valid

- You’ll have to
  - Verify ‘e’, and backtrack (and leave ‘sz’ on input)
  - Expand another S → s, verify the terminal (leaving ‘z’ on input)
  - Verify the final ‘z’, and backtrack to find no further children
  - The parse tree is finished, and since that was all the input, it’s ok.



That is how it works

- Predictive parsing by recursive descent
  - Starts from the start symbol (top)
  - Verifies terminals
  - Picks a unique production for nonterminals based on the lookahead
  - Expands the syntax tree by productions, and recursively treats the new sub-tree in the same way
- This requires that the grammar is suitable, but we can adapt them somewhat
  - Left factor where a common lookahead prevents picking the right production
  - Eliminate left-recursive productions
  - We only saw left factoring in action so far, but let's examine another grammar

We’re aiming for a table

- As with DFA, an algorithm needs a table where it can make decisions based on indexing (nonterminal, terminal) pairs and find a single production
- To make that table, it’s a good idea to determine
  - What can the strings derived from a nonterminal begin with?
  - Which nonterminals can vanish, so that the lookahead symbol is actually part of the *next* production to choose?
  - What can come directly after a nonterminal that can vanish?

(where ‘vanish’ means that there’s a production $X \rightarrow \varepsilon$, so that nonterminal X disappears from the intermediate form in the derivation without consuming any characters from the input token stream)

Here’s another grammar

S → u B D z
B → B v | w
D → E F
E → y | ε
F → x | ε

- It doesn’t model anything in particular, it’s here to be short and sweet

FIRST

- The set FIRST(α) is the set of terminals that can appear to the left in α
  - α is really any ol' combination of terminals and nonterminals
- If we tabulate FIRST for all the heads in the grammar,

FIRST(S) = {u} (u begins the only production)

FIRST(B) = {w} (however many times B → Bv is taken, w appears on the left in the end)

FIRST(E) = {y} (only production that derives any terminal)

FIRST(F) = {x} (ditto)

and finally,

FIRST(D) = {y,x}

y because D → E F → y F

x because D → E F → F → x (E can disappear by E → ε)

nullablility

- A nonterminal is *nullable* if it can produce the empty string (in any number of steps)
  - The Dragon book denotes this by putting ε in the FIRST set
  - I denote it by keeping a separate record, because I like to
  - You can choose for yourself, we can read both notations
- In short order,

nullable (S) = no	(there are terminals in the only production)
nullable (B) = no	(there are terminals in both productions)
nullable (E) = yes	(it produces $E \to \varepsilon$)
nullable (F) = yes	(it produces $F \to \varepsilon$)
nullable (D) = yes	($D \to E\ F \to F \to \varepsilon$)

FOLLOW

- FOLLOW (N) for nonterm. N is the set of terminals that can appear directly to its right
  - In order to find these, you have to examine all the places N appears in production bodies, and find the terminals directly to its right
  - If it has a nonterminal on its right, you have to follow all its productions too, and find out what can come up instead of it
    - That will be its FIRST set
  - If it has a nonterminal that can vanish to its right, you have to look at what comes afterwards…
  - ...and in general, collect all the terminals that can appear to the right in one way or another
- This is a little trickier than FIRST, but it can be done if you concentrate
- If you don’t like to concentrate, you can also slavishly follow the rules beginning at the bottom of p. 221 in the book

For our grammar

- FOLLOW(S) = {$} (the end of input)
- FOLLOW(B) = {v,x,y,z} taken from the derivations

S → uBDz → u**Bv**Dz
S → uBDz → uBEFz → uBFz → u**Bx**z
S → uBDz → uBEFz → u**By**Fz
S → uBDz → uBEFz → uBFz → u**Bz**

- FOLLOW(D) = {z} (from S → uB**Dz**)
- FOLLOW(E) = {x,z} taken from the derivations

S → uBDz → uBEFz → uB**Ex**z
S → uBDz → uBEFz → uB**Ez**

- FOLLOW(F) = {z} (from S → uBDz → uBE**Fz**)

Two rules

- Armed with the FIRST, FOLLOW and nullable information, consider every production X → α in the grammar, and apply two rules:
  - Enter the production X → α at (X,t) where t is in FIRST(α)
  - When $\alpha \rightarrow^* \varepsilon$, enter the production X → α at (X,t) where t is in FOLLOW(X)

Trying out rule #1

- With the grammar that we have, the first rule gives the table

	u	w	v	x	y	z
S	S → uBDz					
B		B → w B → Bv				
D				D → EF	D → EF	
E					E → y	
F				F → x		

Houston, we have a... left recursion

- This will not do, expanding B on lookahead ‘w’ requires a choice we can’t make

	u	w	v	x	y	z
S	S → uBDz					
B		B → w B → Bv				
D				D → EF	D → EF	
E					E → y	
F				F → x		

Fix the grammar

- Eliminating left recursion gives us
  - S → uBDz
  - B → w B’
  - B’ → v B’ | ε
  - D → E F
  - E → y | ε
  - F → x | ε
- Update the FIRST, FOLLOW, nullable sets after the change:
  - FIRST(B) = {w}, FOLLOW(B) = {x,y,z} , nullable(B) = no
  - FIRST(B’) = {v}, FOLLOW(B’) = {x,y,z}, nullable(B’) = yes

Try rule #1 again

- This looks better:

	u	w	v	x	y	z
S	S → uBDz					
B		B → wB’				
B’			B’ → vB’			
D				D → EF	D→ EF	
E					E → y	
F				F → x		

Adding rule #2

- Where nonterms are nullable, insert at FOLLOW

	u	w	v	x	y	z
S	S → uBDz					
B		B → wB’				
B’			B’ → vB’	B’ → ε	B’ → ε	B’ → ε
D				D → EF	D→ EF	D→ EF
E				E → ε	E → y	E → ε
F				F → x		F → ε

Now we have an LL(1) parsing table

- There is only one rule to choose from any pair of (nonterminal, terminal), so the tree can be built deterministically by following the method from the first example
  - Pick productions for nonterminals by looking them up in the table
- Parse a sample statement like uwvvxz if you like
- Try to think of how you would structure a program that works the same way

Why we cover this

- Bottom-up parsers are a handful to construct, it’s a job best left for an automatic generator
- Top-down parsers work on a simple principle, those are doable by hand
  - At least as long as we stick to LL(1), longer lookaheads like LL(2) make for tables that have a column for every pair of terminals
- We’ll use a bottom-up generator in the practical work
- You should also know how to make a top-down one in the theoretical work
  - So as to make an informed choice if you need to parse things