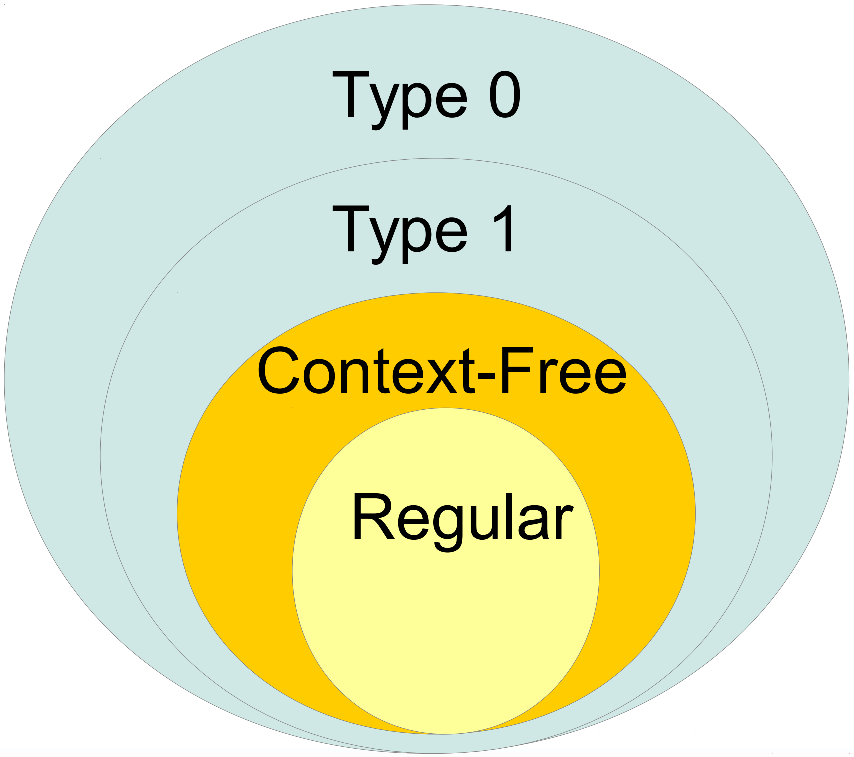
NTNU – Trondheim
Norwegian University of
Science and Technology

Bottom-up parsing

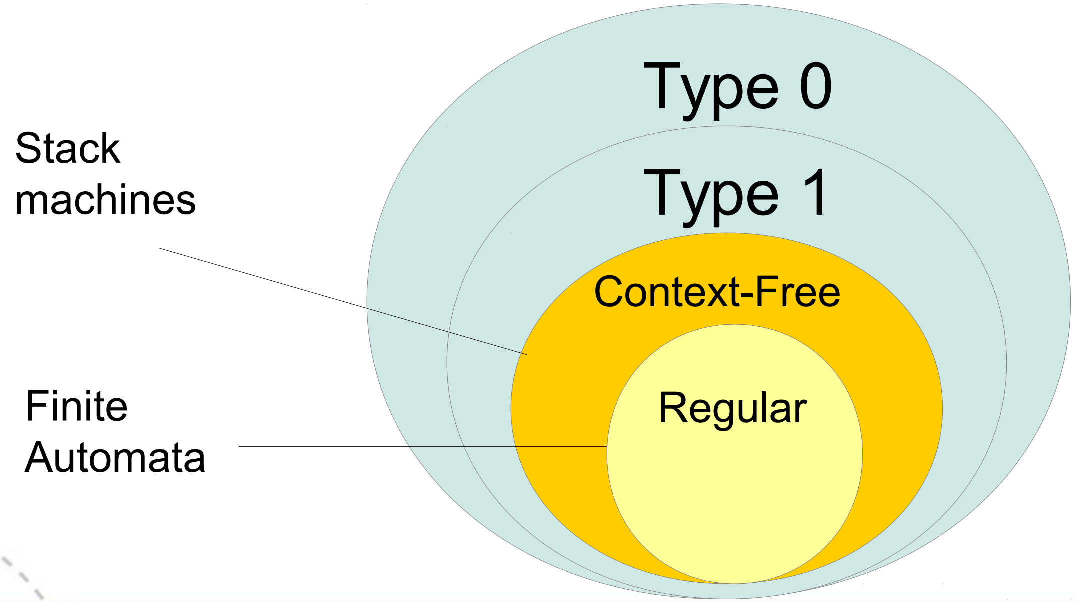
Where we are (again)

- Introducing C.F.Grammars, we said that they include regular languages, and then some more



Memories of past states

- These classes of languages are recognizable by (abstract) machines of differing power
  - We know the finite automata
  - Stack machines (or *pushdown automata*) are like F. A., but with added *push* and *pop* operations that let them trace the path they took to a state (and revert to where they've been)



What does a top-down parser look like?

- We looked at how to make an LL(1) parsing table, but not at how to turn it into a program
- Here's a grammar that's so simple that we can just knock the parsing table out by looking at it:

$A \rightarrow xB \mid yC$

$B \rightarrow xB \mid \varepsilon$

$C \rightarrow yC \mid \varepsilon$

	x	y	$
A	$A \rightarrow xB$	$A \rightarrow yC$	
B	$B \rightarrow xB$		$B \rightarrow \varepsilon$
C		$C \rightarrow yC$	$C \rightarrow \varepsilon$

In code

	x	y	$
A	A → xB	A → yC	
B	B → xB		B → ε
C		C → yC	C → ε

- One way to implement this is to write a function for each nonterminal, and make them mutually recursive according to the table

```
parse_A ():
    switch(symbol):
        case x:
            add_tree( x, B )
            match ( x )
            parse_B ( )
        case y:
            add_tree( y, C )
            match ( y )
            parse_C ( )
        case $:
            error()
    return
```

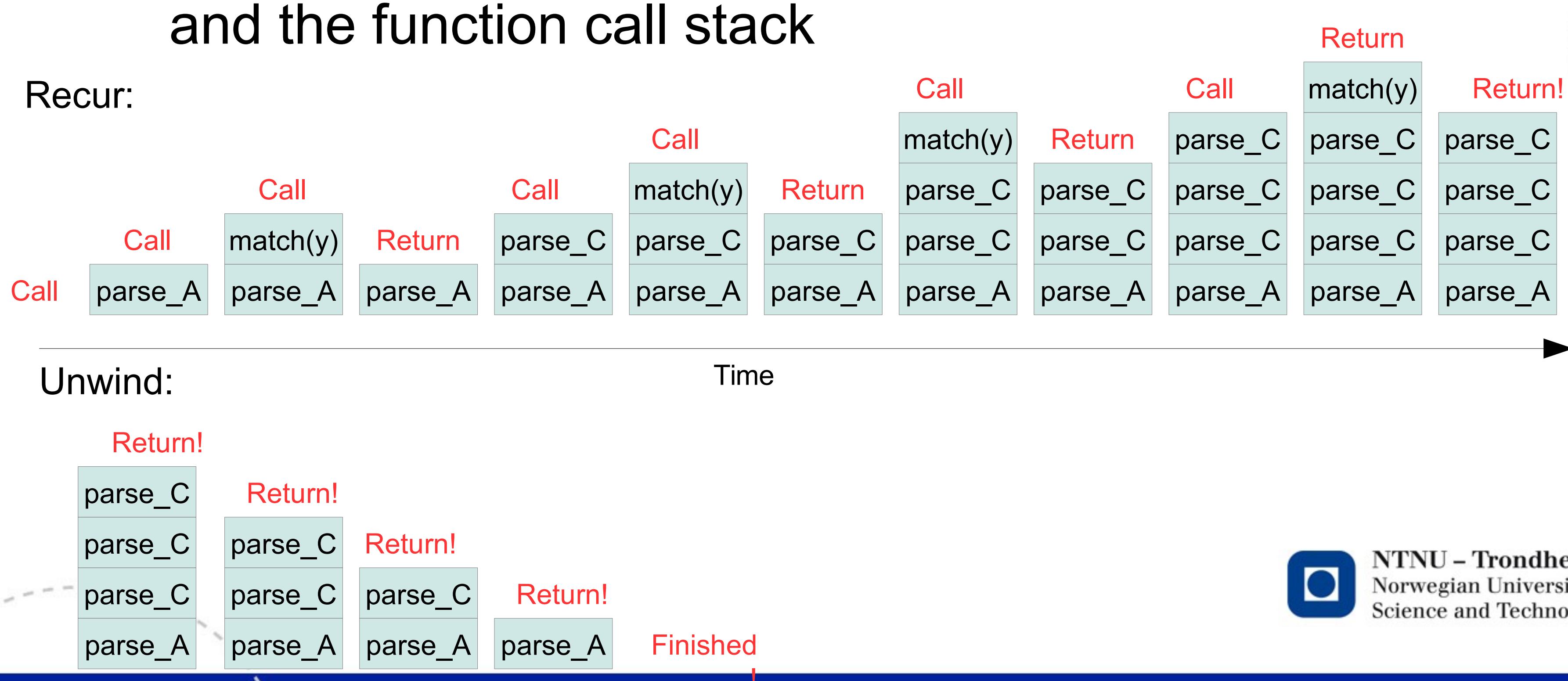
```
parse_B():
    switch(symbol):
        case x:
            add_tree(x,B)
            match(x)
            parse_B ( )
        case y:
            error()
        case $:
            return
    return
```

```
parse_C():
    switch(symbol):
        case x:
            error()
        case y:
            add_tree(y,C)
            match(y)
            parse_C ( )
        case $:
            return
    return
```

Function calls stack up

- Parsing ‘y y y’, we get
  - The derivation A → y C → y y C → y y y C → y y y

  and the function call stack



NTNU – Trondheim
Norwegian University of Science and Technology

Recursive descent vs. stack

- Recursive descent parsing uses the function call mechanism to implement its stack machine
  - It's hidden in the programming language, but it is there
- LL(1) can also be done with iterations
  - Provided that you're prepared to implement your own stack
- Generally, the need for a stack comes out of the need to match up beginnings and ends
  - Any construct of the sort <start> <thing> <end> where the <thing> can contain further <start> and <end>s, as in

    Expression → (expression)

    Statement → { statement }

    Comment → (* Comment *)

    (/ ML does this, C comments can't be nested */)*

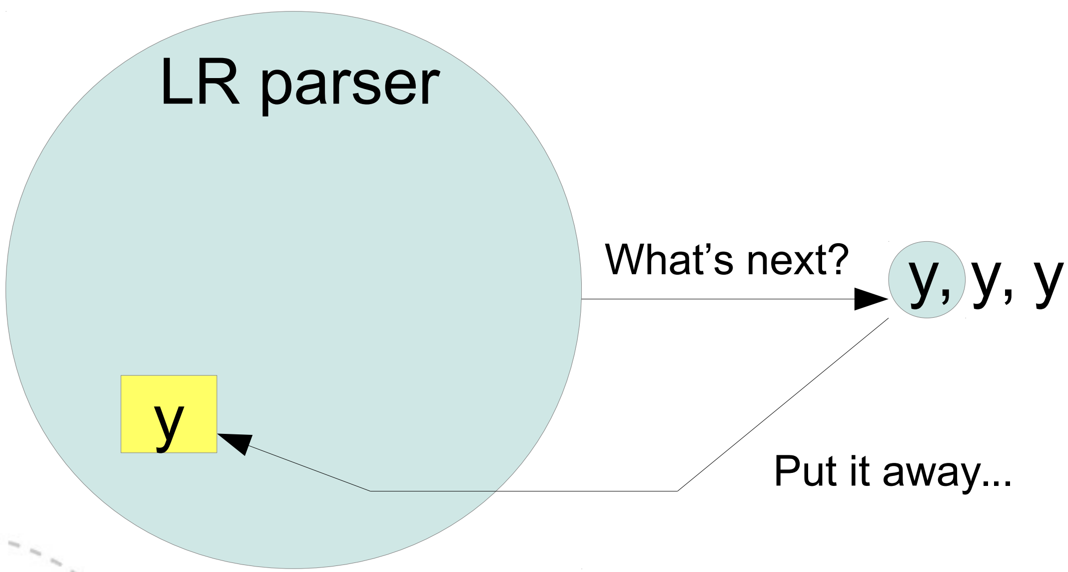
Another way to parse

- The "LL" in LL(1) is
  - Left-to-right scan
  - **Leftmost Derivation** (always expand the leftmost nonterminal)
- How can we go at it from the right?
  - i.e. get L**R** parsing, to obtain a Rightmost derivation?
- It will require looking deeper into the token stream before deciding on productions...

A → xB | yC
B → xB | ε
C → yC | ε

General operation

- Take the same, silly grammar again
- Instead of making a decision as soon as a terminal comes along, stack them up



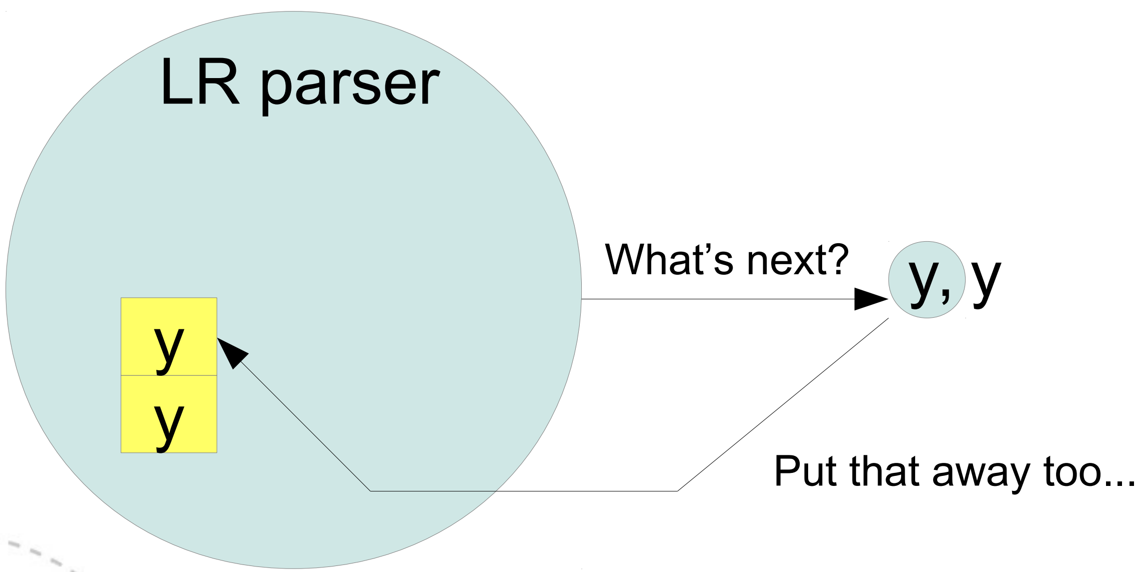
We might be making an A or a C here, hold on...

NTNU – Trondheim
Norwegian University of Science and Technology

$A \rightarrow xB \mid yC$
$B \rightarrow xB \mid \varepsilon$
$C \rightarrow yC \mid \varepsilon$

Keep stacking

- As the state of the internal stack grows, it identifies more and more of a single production rule

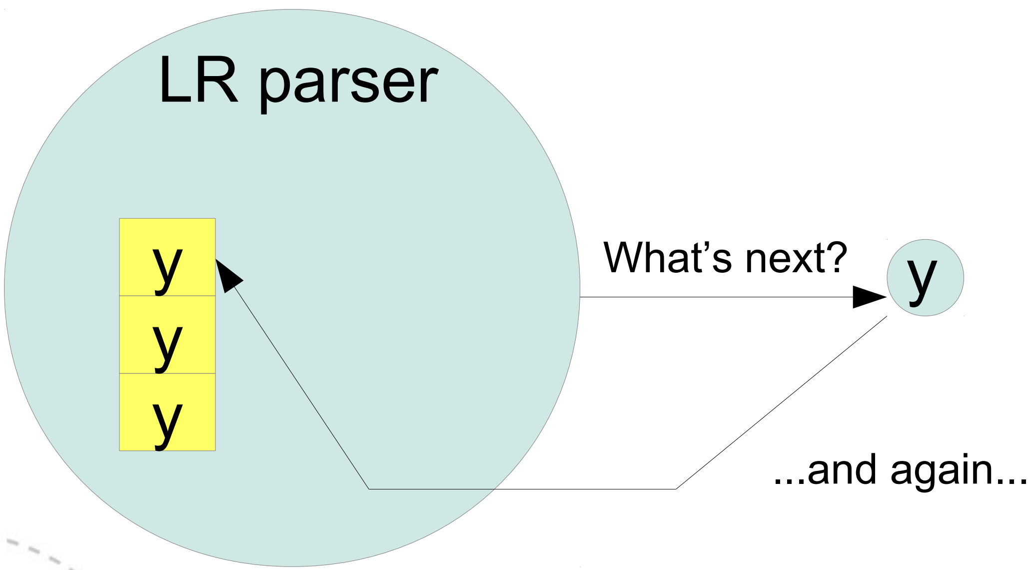


We're definitely working towards some C-s here, how many?

$A \rightarrow xB \mid yC$
$B \rightarrow xB \mid \varepsilon$
$C \rightarrow yC \mid \varepsilon$

Keep stacking

- As the state of the internal stack grows, it identifies more and more of a single production rule

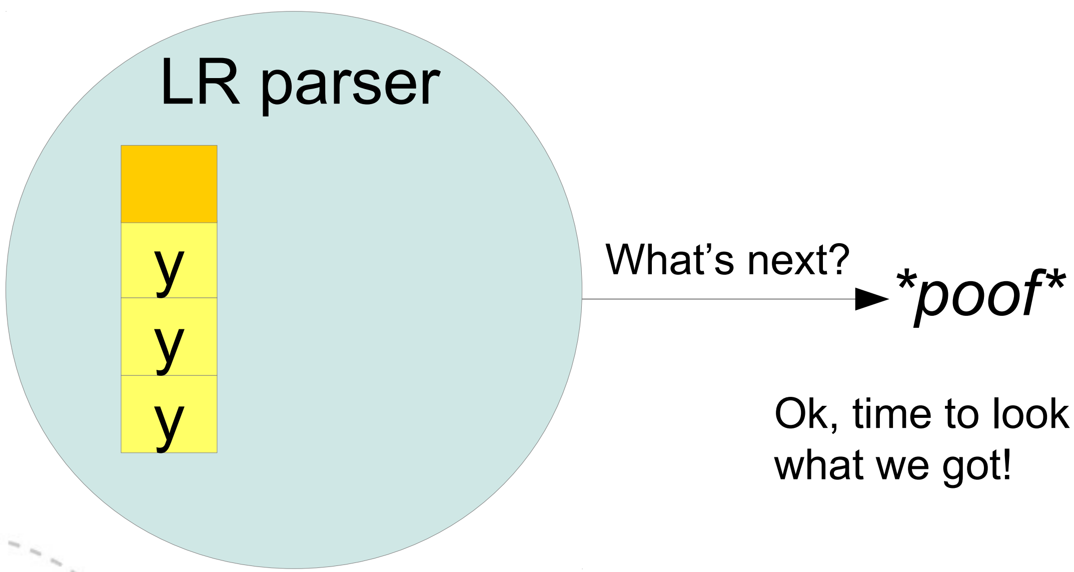


We're definitely working towards some C-s here, how many?

$A \rightarrow xB \mid yC$
$B \rightarrow xB \mid \varepsilon$
$C \rightarrow yC \mid \varepsilon$

Enough is enough

- For this grammar, the sequence ends when the input does

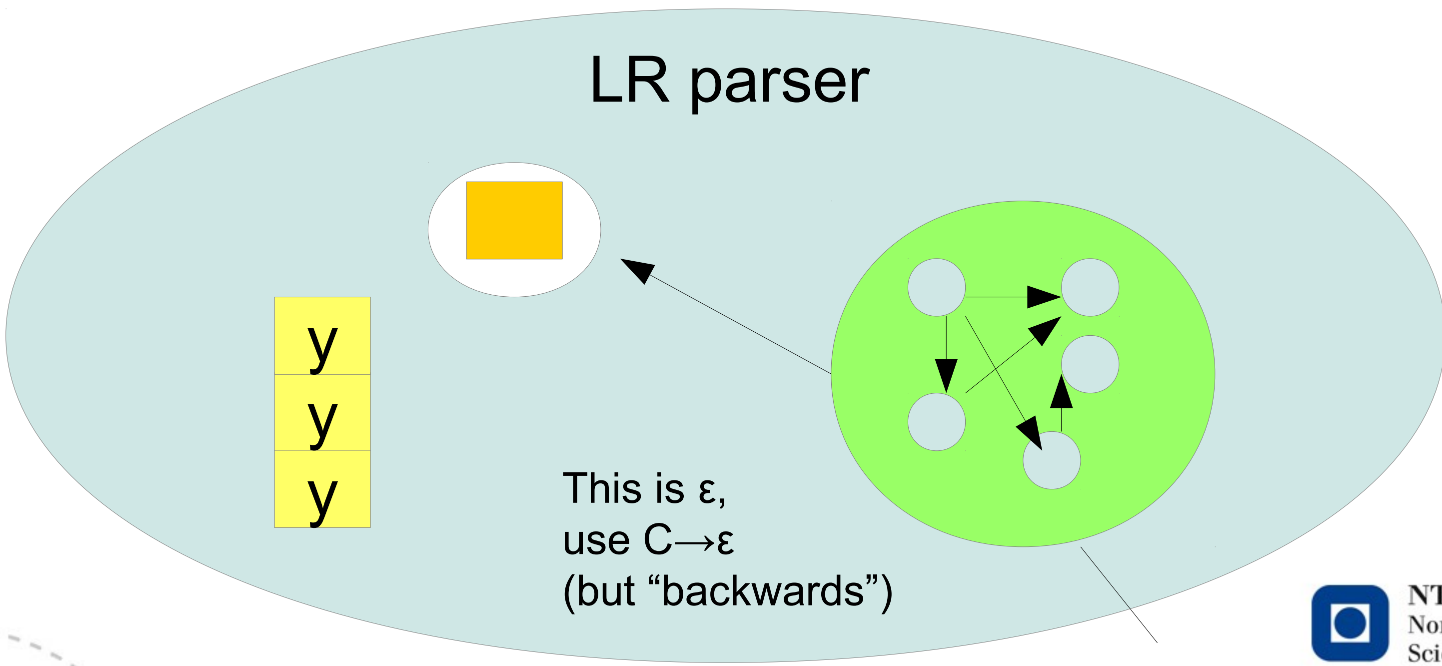


Ok, time to look at what we got!

$A \rightarrow xB \mid yC$
$B \rightarrow xB \mid \varepsilon$
$C \rightarrow yC \mid \varepsilon$

Bring out your states

- The stack extension is for memory, the production rules can be represented by a finite automaton
- It has been watching while we were stacking symbols, so it knows that we've taken a direction where there are no x-s or B-s

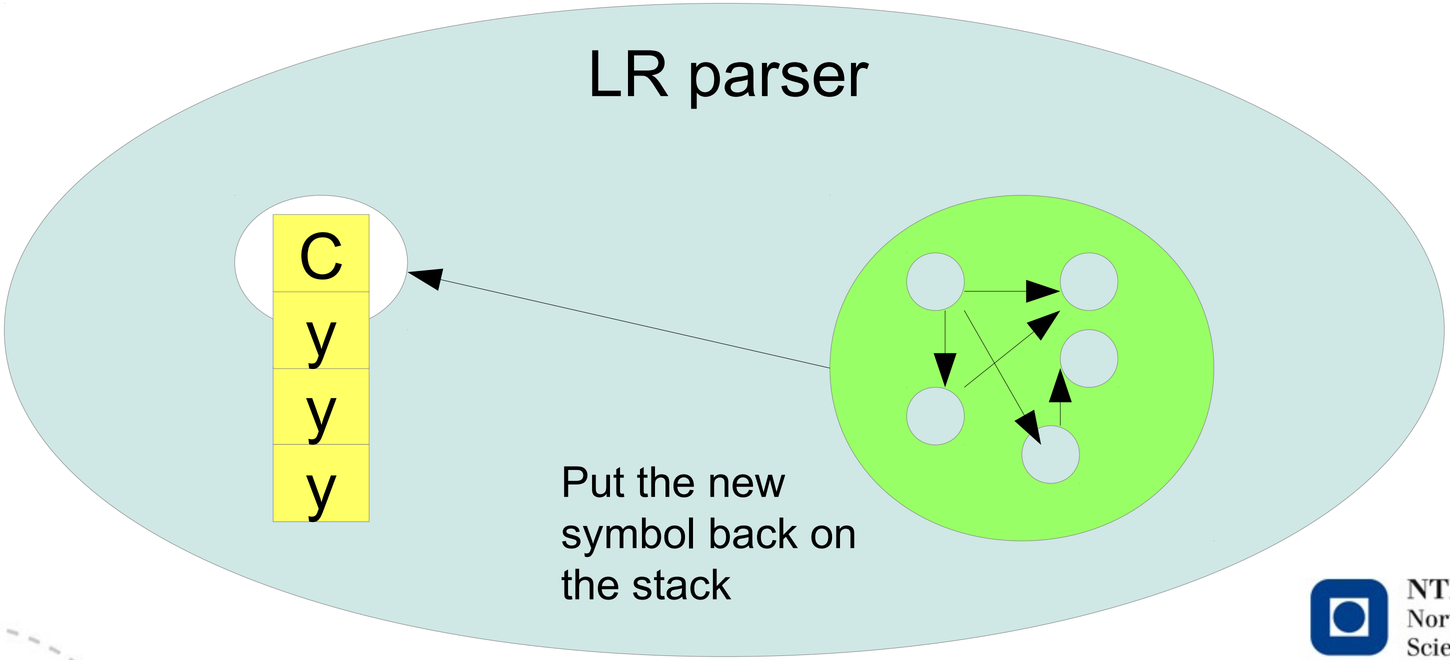


(purely for illustration...)

$A \longrightarrow xB \mid yC$
$B \longrightarrow xB \mid \varepsilon$
$C \longrightarrow yC \mid \varepsilon$

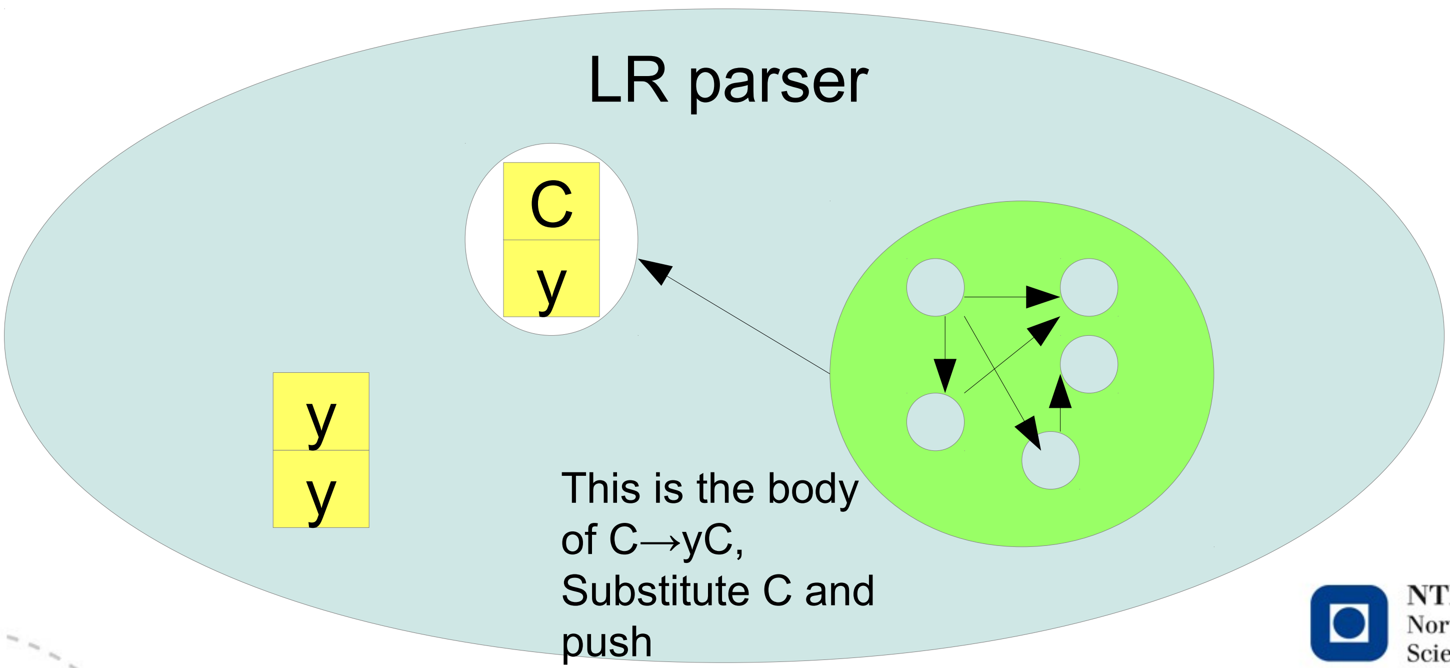
Reduce body to head

- We're at the end of the stream, so we're putting in the last (rightmost) C nonterminal
  - This works out the derivation in reverse order



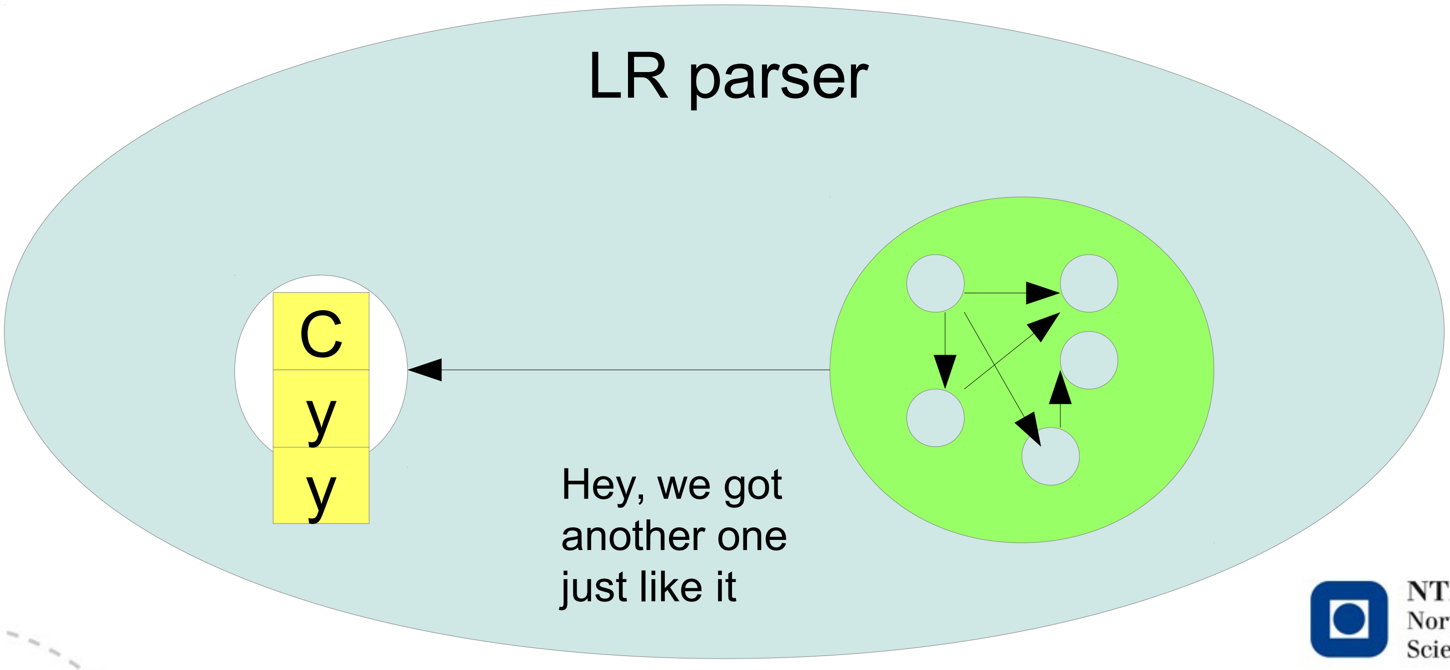
$A \rightarrow xB \mid yC$
$B \rightarrow xB \mid \varepsilon$
$C \rightarrow yC \mid \varepsilon$

Next move



$A \rightarrow xB \mid yC$
$B \rightarrow xB \mid \varepsilon$
$C \rightarrow yC \mid \varepsilon$

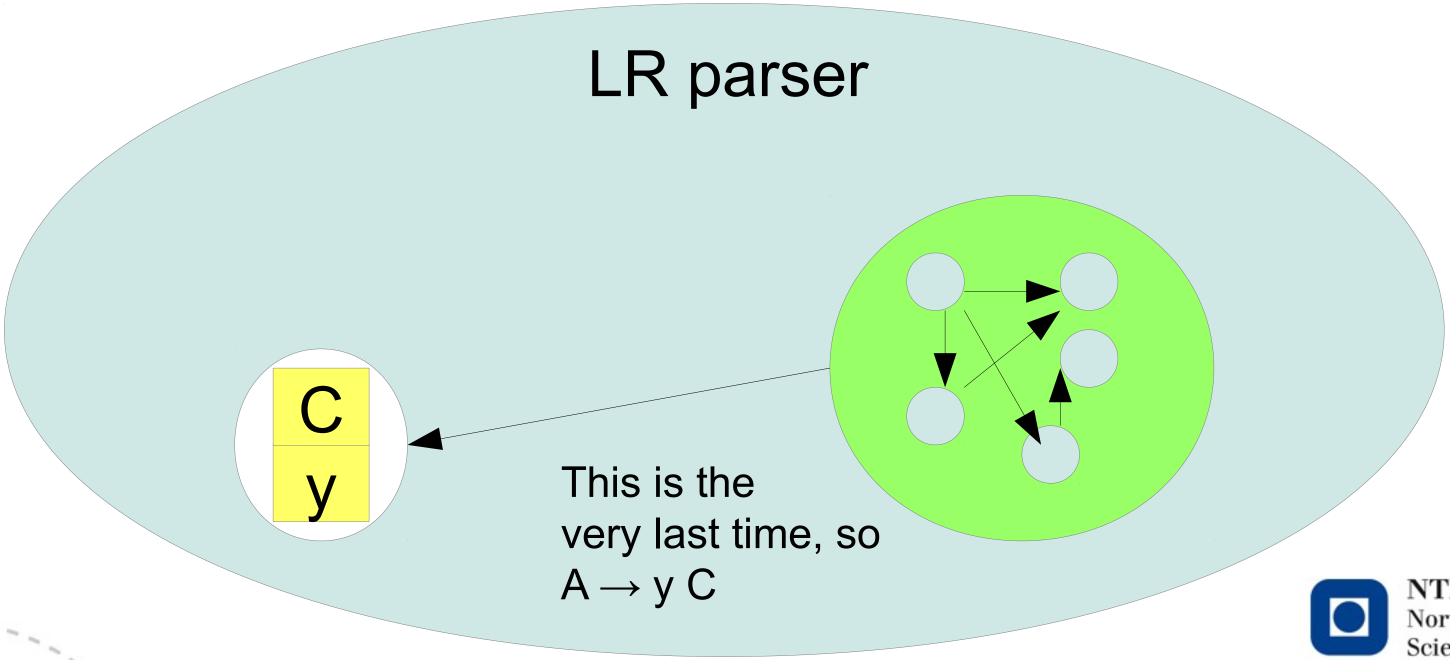
...and it repeats...



$A \to xB \mid yC$
$B \to xB \mid \varepsilon$
$C \to yC \mid \varepsilon$

...until...

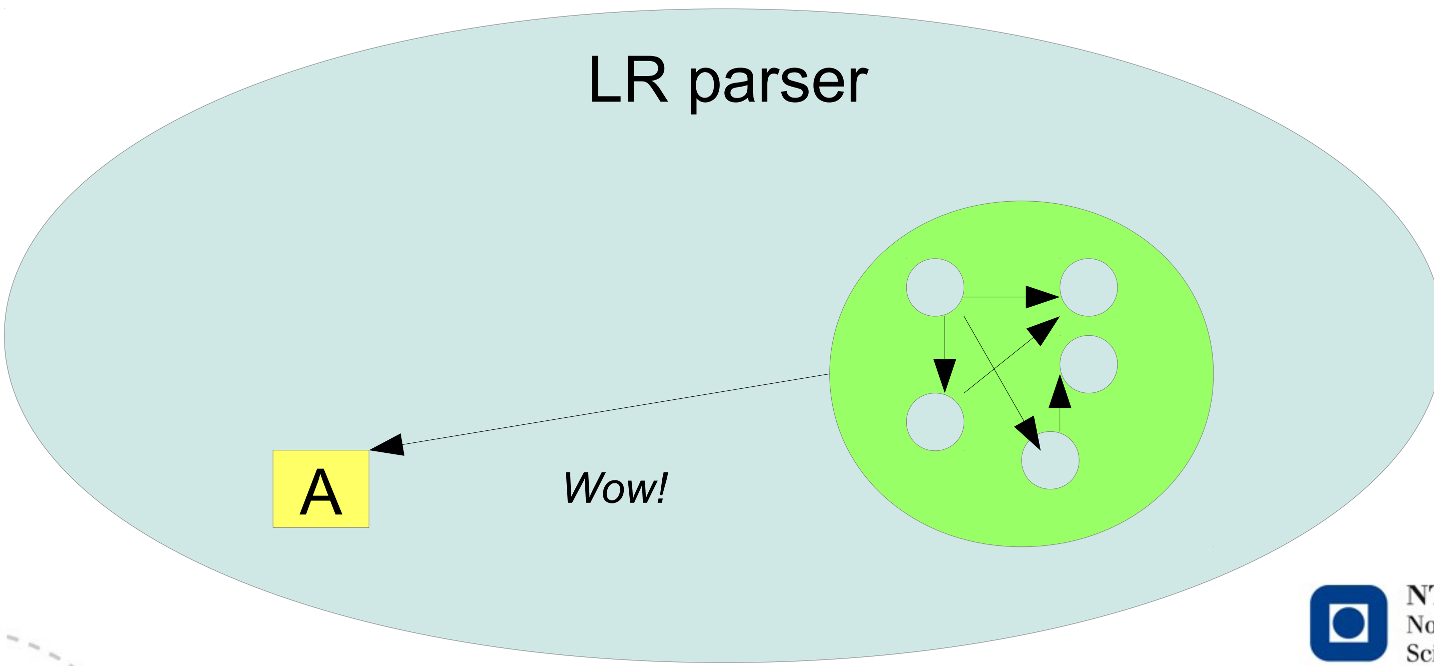
- The automaton built the stack
- The stack says how deeply into the grammar we've gone
- When the final body appears, we reduce the start symbol



$A \rightarrow xB \mid yC$
$B \rightarrow xB \mid \varepsilon$
$C \rightarrow yC \mid \varepsilon$

We're finished!

- Only the start symbol is left on stack, this says that the statement was syntactically correct



If you look for the derivation

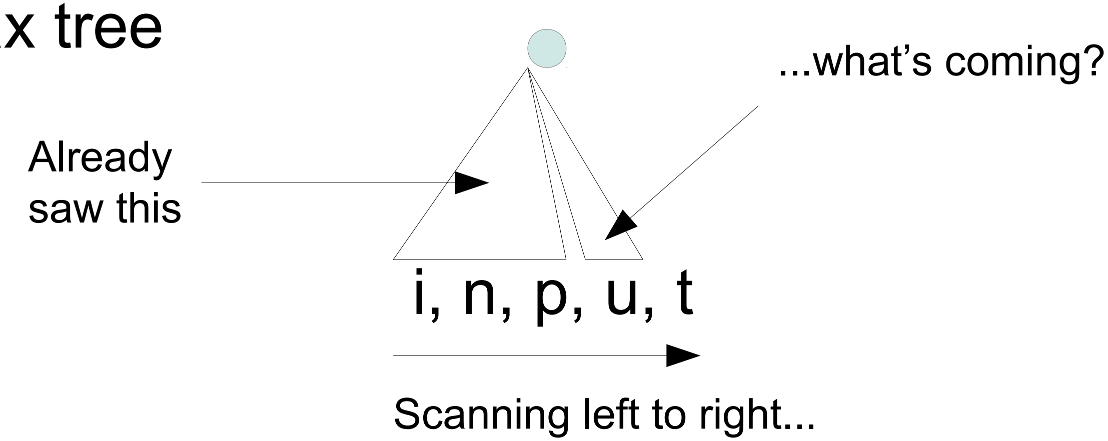
- Bending notation, space, and time a bit, we can illustrate it like this

Stack	Input	Action	
-	y,y,y	Shift	
y	y,y	Shift	
y,y	y	Shift	
y,y,y	-	Reduce $C \to \varepsilon$	(push C)
y,y,y,C	-	Reduce $C \to yC$	(pop y,C + push C)
y,y,C	-	Reduce $C \to yC$	(pop y,C + push C)
y,C	-	Reduce $A \to yC$	(pop y,C + push A)
A	-	Well done, cookies for everyone	

Here is our rightmost derivation, in reverse

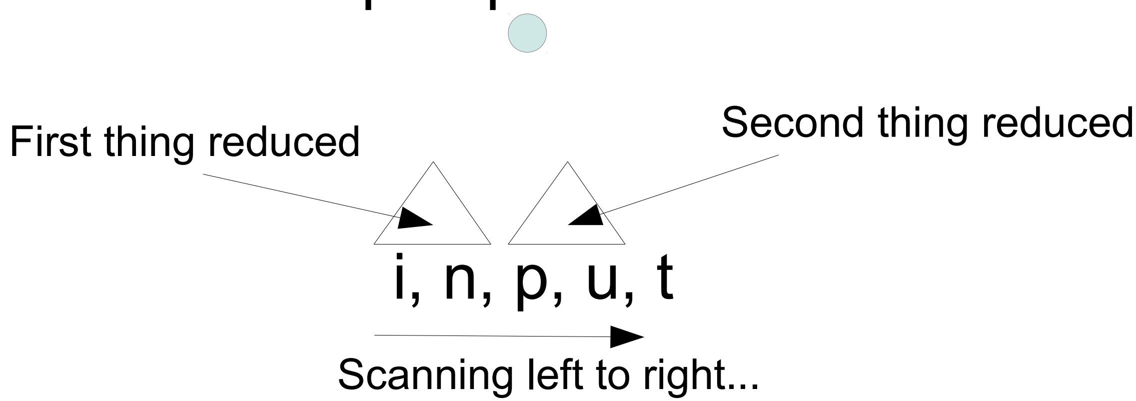
Things the example didn't show

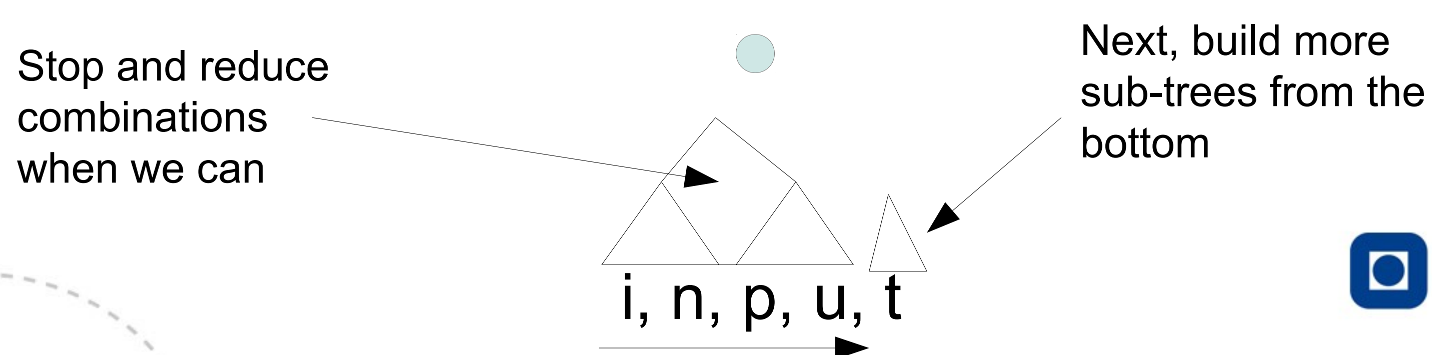
- Recognizing the body of a production doesn't have to wait until the very end
  - Only until it is uniquely determined
- Top-down parsing matches input to productions from above in the syntax tree



Things the example didn't show

- Bottom-up parsing buffers input until it can build productions on top of productions





Scanning left to right...

That’s the principle of it

- Key ingredients:
  - A stack to shift and reduce symbols on
  - An automaton that can use stacked history to backtrack its footsteps
  - A grammar with one and only one initial production
- The last point is easy, if you have a grammar like

  $S \rightarrow iCtSz \mid iCtSeSz$
  - It can (somewhat obviously) be *augmented* like so

    $S' \rightarrow S$

    $S \rightarrow iCtSz \mid iCtSeSz$

    without changing the language.
  - We’ll see the purpose of that shortly

Various schemes

- The LR(k) family of languages can all be parsed with some kind of *shift-reduce* parser like this
- The more elaborate your automaton, the more grammars it can handle
  - We're going to study a few variations of this theme:
    SLR, LALR, LR(1)
  - They're easier to understand if we start with one which is actually ~~blooming useless~~ somewhat restrictive, but demonstrates a lot of general principles
  - That is LR(0) automaton construction, up next.