NTNU – Trondheim
Norwegian University of
Science and Technology

LR(0) parser construction

Bottom-up parsing

- Recapping the last lecture, what we need for a bottom-up parsing scheme is
  - An internal stack to shift and reduce symbols on
  - An automaton that tell us what to do when, and use stacked history to backtrack its footsteps
  - A grammar with one and only one initial production

The LR(0) automaton

- The overall construction is similar to the NFA→DFA idea, in that
  - We're tracking all the different things that can happen throughout a derivation (there will be closures of related things)
  - We're introducing states whenever it is necessary to cover something new
  - The states represent all the different paths that may have led to them
  - Some states are *reducing*, that means
    - Pop body, push head (but we won't draw the stack in, just remember that this happens)
    - Revert to where we started recognizing the present production

      ...and the production which led to that one, if it is now finished...

      … and the production which led to that other one…

      … until the start symbol is all that's left
  - Transitions *shift* symbols
    - They are what moves us ahead while working toward a reduction

LR(0) *items*

- Since we have to track how far we have come along in a production's body, the notation for productions extends to a bunch of *LR(0) items*
  - All this means is that we add a marker ('.') to denote it
  - It's not punctuation in the language, just a position-tracking dot
  - The production

    $S \rightarrow aBc$

    gives us the four items

$S \rightarrow . a B c$	(I'm just starting on an S)
$S \rightarrow a . B c$	(I'm working on S, already saw an a)
$S \rightarrow a B . c$	(I'm working on S, have found a and B)
$S \rightarrow a B c .$	(We have seen a whole S at this point, toss its history and put S)
  - We won't need every item all the time, but they are the things we will combine into states, so this is what they mean

NTNU – Trondheim
Norwegian University of
Science and Technology

(Augmenting the grammar with) A Place to Begin

- The point of rewriting the grammar

  S → iCtSz | iCtSeSz

  into

  S' → S

  S → iCtSz | iCtSeSz

  is that when we're looking at items,

  S' → . S

  uniquely says that we're just starting out, and

  S' → S .

  uniquely says that we're finished.
- It's just a convenience, to avoid the corner cases of

  "Begin the construction with every item from the start symbol", and

  "Accept if any of the start symbol's productions reduce"

  – This way, the rule can just be "start at the beginning, stop at the end", so to speak

NTNU – Trondheim
Norwegian University of Science and Technology

(Let's keep the grammar handy, for reference)

$D \rightarrow E\ F$
$E \rightarrow y \mid F$
$F \rightarrow x$

Closures of items

- Closures of items are the sets of all items that can be obtained from the grammar without moving the position marker
  - Consider this grammar

    $D \rightarrow E\ F$

    $E \rightarrow y \mid F$

    $F \rightarrow x$

    and start from the item $D \rightarrow .\ E\ F$

$D \longrightarrow E\ F$
$E \longrightarrow y \mid F$
$F \longrightarrow x$

What can happen here?

- Without moving the marker, applying productions can give us
  - $D \rightarrow .\ E\ F$
  - $E \rightarrow .\ y$
  - $E \rightarrow .\ F$
    - (those were the E-productions, we've made an F in the process)
  - $F \rightarrow .\ x$
    - (this comes from repeating the procedure with the new items we found)
- What this closure represents, is that derivations from D can begin as
  - $D \rightarrow .\ E\ F$
  - $D \rightarrow .\ E\ F \rightarrow .\ y\ F$
  - $D \rightarrow .\ E\ F \rightarrow .\ F\ F \rightarrow .\ x\ F$

  without moving the marker past any symbols.
- The notation implies all this (transitively)

$D \rightarrow E\ F$
$E \rightarrow y \mid F$
$F \rightarrow x$

Closures are states

– The closure gives us an automaton state, labeled with what it represents

$D \rightarrow .\ E\ F$
$E \rightarrow .\ y$
$E \rightarrow .\ F$
$F \rightarrow .\ x$

– The transitions out of this state represent all the different ways we can advance the input marker:

F

What happens at
$E \rightarrow y\ .$

y

$D \rightarrow .\ E\ F$
$E \rightarrow .\ y$
$E \rightarrow .\ F$
$F \rightarrow .\ x$

E

What happens at
$D \rightarrow E\ .\ F$

What happens at
$F \rightarrow x\ .$

$D \rightarrow E\ F$
$E \rightarrow y \mid F$
$F \rightarrow x$

Advancing input
(*Shift* actions)

- *Selecting* one of these transitions gives us a new item to find the closure of, and hence, the destination state

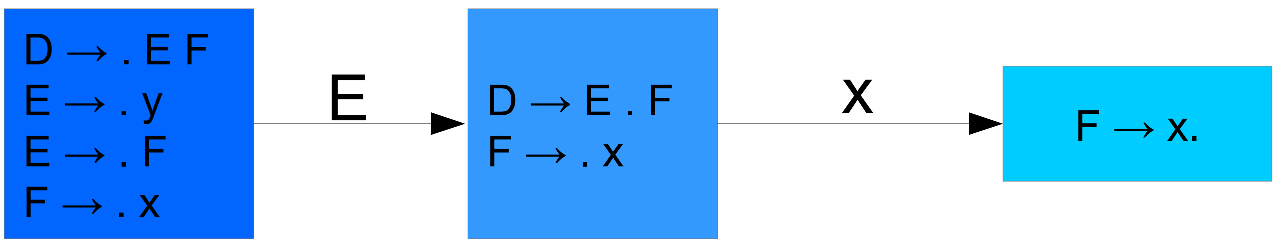
$$\boxed{\begin{array}{l} D \rightarrow .\ E\ F \\ E \rightarrow .\ y \\ E \rightarrow .\ F \\ F \rightarrow .\ x \end{array}} \xrightarrow{\quad E \quad} \boxed{\begin{array}{l} D \rightarrow E\ .\ F \\ F \rightarrow .\ x \end{array}}$$

NTNU – Trondheim
Norwegian University of
Science and Technology

$D \rightarrow E\ F$
$E \rightarrow y \mid F$
$F \rightarrow x$

Matching productions
(*Reduce* actions)

- When we reach an item that has the marker at the end, we've gone through states that encode a sequence that can appear in a derivation

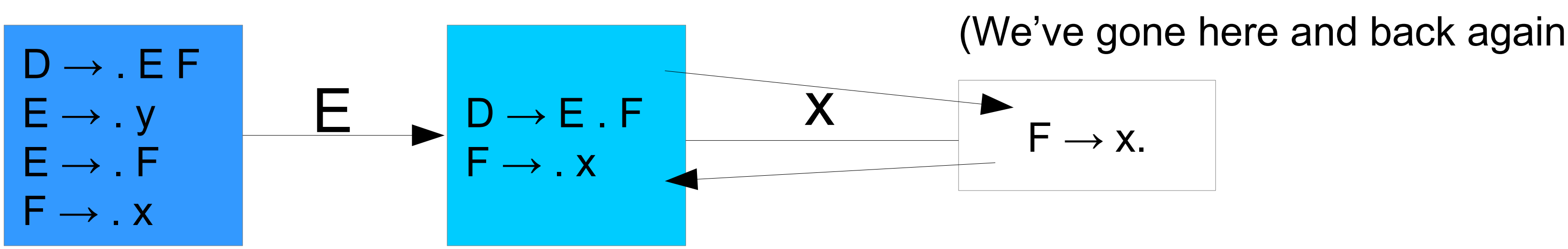


- D → . E F *(is) E . F* → E . x *(is)* E x .
  - As far as the grammar is concerned, D → E F → E x
  - We just decorated it with a position in our stack-based reasoning

$D \rightarrow E\ F$
$E \rightarrow y \mid F$
$F \rightarrow x$

This is where we backtrack

- Reducing at state $F \rightarrow x\ .$ means
  - We have a stack with an x on top of it
  - Remove it, and replace it with an F
- That returns us to the stage where we were about to shift one

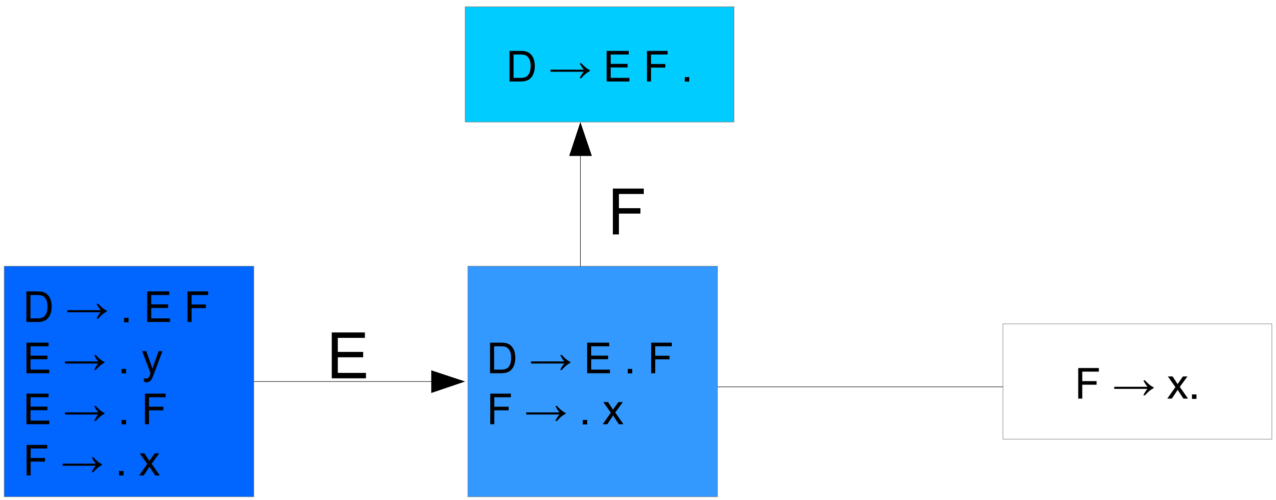


  - Next thing on stack is the F we just created a moment ago

$D \to E\ F$
$E \to y \mid F$
$F \to x$

...and again

- Below the F is the E that brought us here…

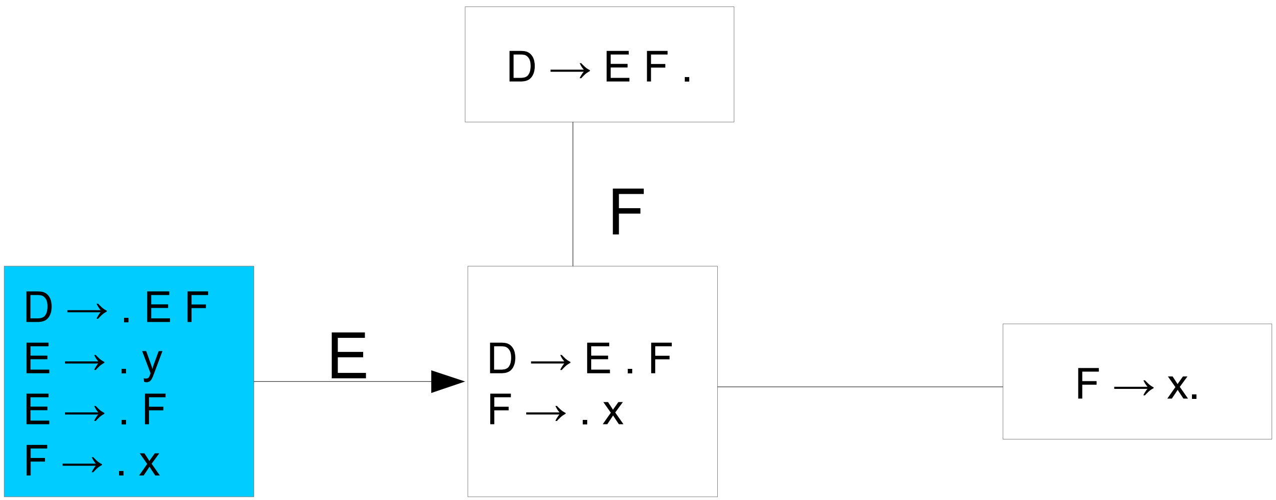


- D → E F . is another reducing state, take the E, F off stack, put in a D instead, and return to before the E was shifted...

D → E F
E → y | F
F → x

That was one traversal

- We've been through D → E F → E x



- We started by shifting an E
  - That one must have been produced by a similar traversal elsewhere in the automaton, which shifted y, reduced E → y, and thus gave us an E to shift

Making the LR(0) automaton

Start with the designated start item

- The one that looks like X’ → . X

1) Find its closure, make a state
2) Follow all the transitions
3) Repeat from 1

until you reach the reduction X’ → X at the other end.

(...and don’t duplicate states when you get an item you’ve already made a state from)

NTNU – Trondheim
Norwegian University of
Science and Technology

(Going by hand)

- If you don't respect some strict depth-first traversal ordering, the final reduction can be the first thing you find
  - It gets hard to remember where you were after a few branches and backtracks
- In this case, it is necessary to stare at the automaton for a bit, to convince yourself that you've visited it everywhere
  - That's not so mathematically rigorous, but it's OK for what we're after
- We're doing this to understand how it works
  - Homespun LR(0) parsing is a waste of time, there are splendid generator programs

A simple grammar to try it on

This one models nested*, comma-separated lists:

$S \rightarrow (L) \mid x$

$L \rightarrow S \mid L, S$

i.e. statements like

(x,x,(x,x))

S → (L) → (L,S) → (L,S,S) → (S,S,S) → (x,x,S) → (x,x,(L)) → (x,x,(L,S))*

→ (x,x,(x,x))*

or

(x,(x,x),x,(x,x))

and similar

**(Confer w. regex versus nested parentheses - Context-Free Grammars are more powerful...)*

NTNU – Trondheim
Norwegian University of
Science and Technology

Blackboard time

- There's a summary of the whole development on the next few slides, but I think it's worth going through at a more leisurely pace, so I'll draw it step by step.
- Follow as it fits yourself, you can review the slides.

From the beginning

- Augmenting the grammar with a production that has a single rule, we get somewhere to start:

  S’ → S

- Taking that as the basis for a first state, its closure is

  S’ → .S

  S → .(L)

  S → .x

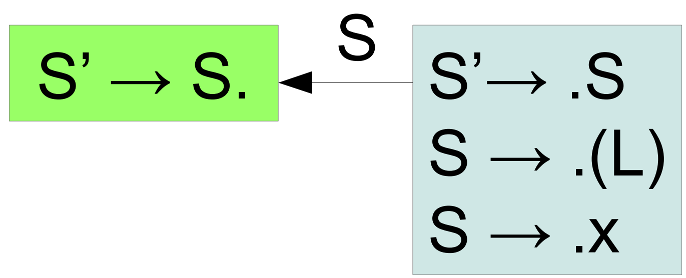
  so we make an automaton state out of that

S’→ .S S → .(L) S → .x

S’ → S
S → (L)
S → x
L → S
L → L , S

From the end

- The last thing that will happen is that we have parsed an S, and can shift it to complete parsing
- That sends us to a state where we can reduce our artificial start symbol, and declare victory

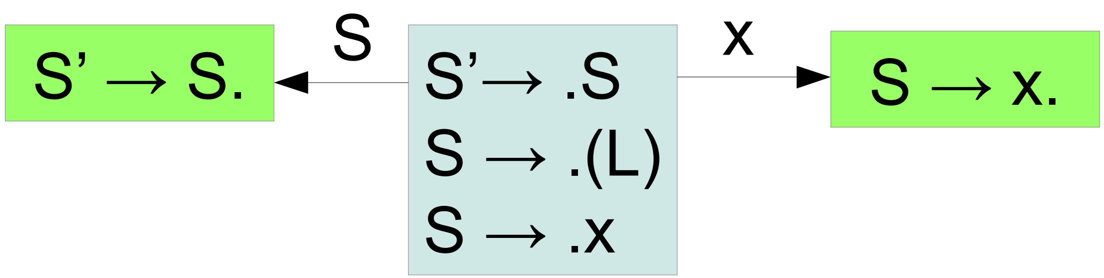


NTNU – Trondheim
Norwegian University of Science and Technology

S’ → S
S → (L)
S → x
L → S
L → L , S

Another thing can happen

- We might shift an ‘x’ terminal

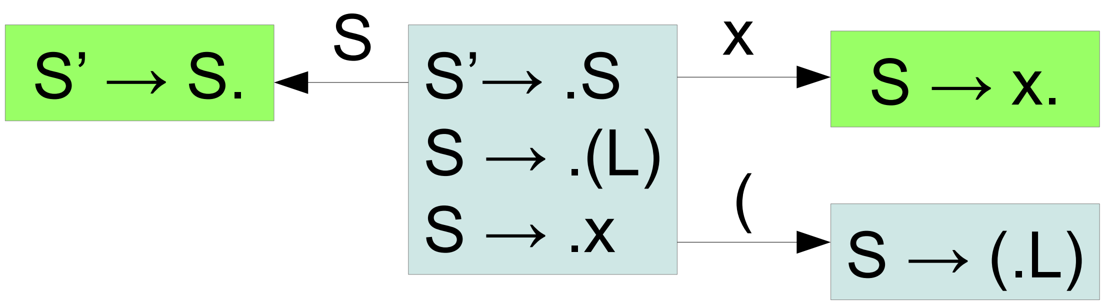


- This completes another production (. is at the end), so it is also a state where we have found a reduction

S' → S
S → (L)
S → x
L → S
L → L , S

The final thing that can happen

- We might shift a '(' terminal



- This doesn't complete any productions, so we'll have to build more states
- Start over, with the closure at the destination state

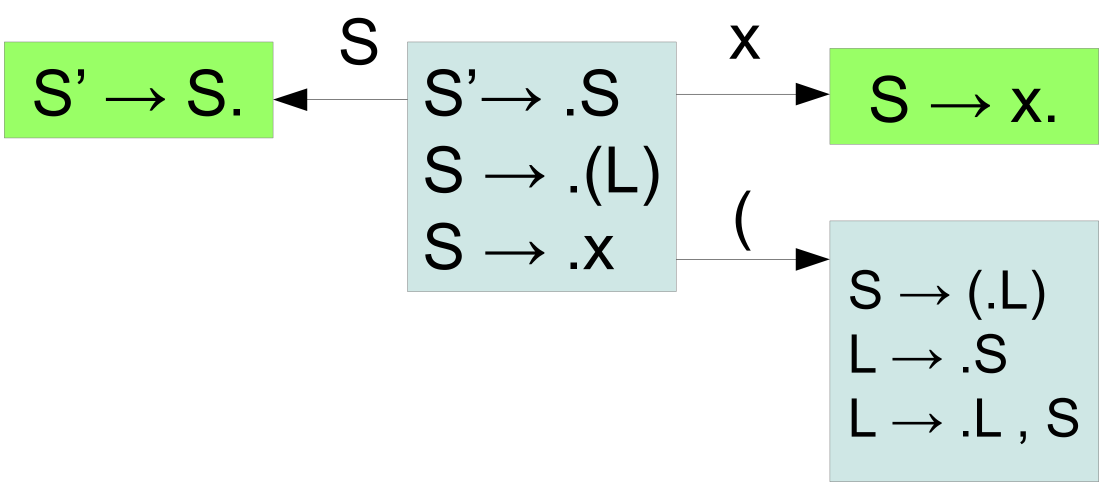
$S' \rightarrow S$
$S \rightarrow (L)$
$S \rightarrow x$
$L \rightarrow S$
$L \rightarrow L , S$

The closure at S→(.L)

- A non-terminal follows the position marker
  - That can expand into

    $L \rightarrow S$

    $L \rightarrow L, S$



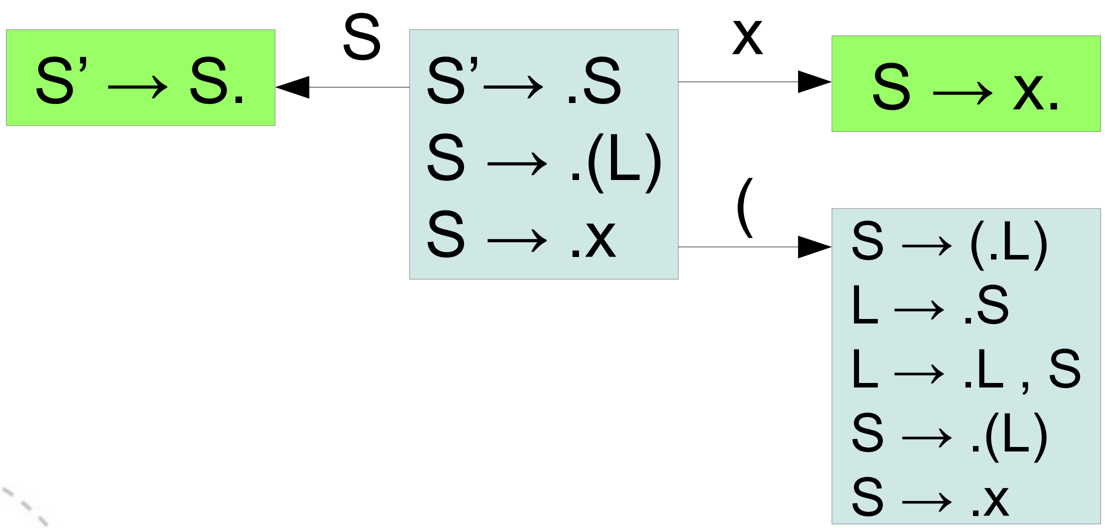
S' → S
S → (L)
S → x
L → S
L → L , S

We're not done yet

- The expansion put the position marker ahead of another nonterminal (that is, S)
  - S can expand into

    S → (L)

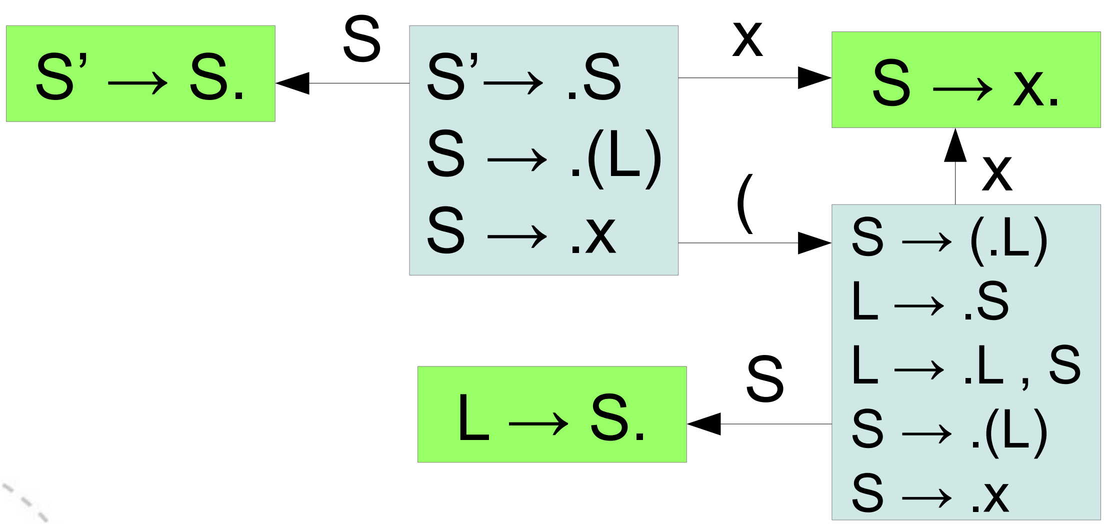
    S → x



$S' \to S$
$S \to (L)$
$S \to x$
$L \to S$
$L \to L , S$

Time to shift more symbols

- Reducing states are easy to find, shifting S or x completes a production
  - We already have a state for S → x.
  - L → S. is the other reducing state we can reach in one shift

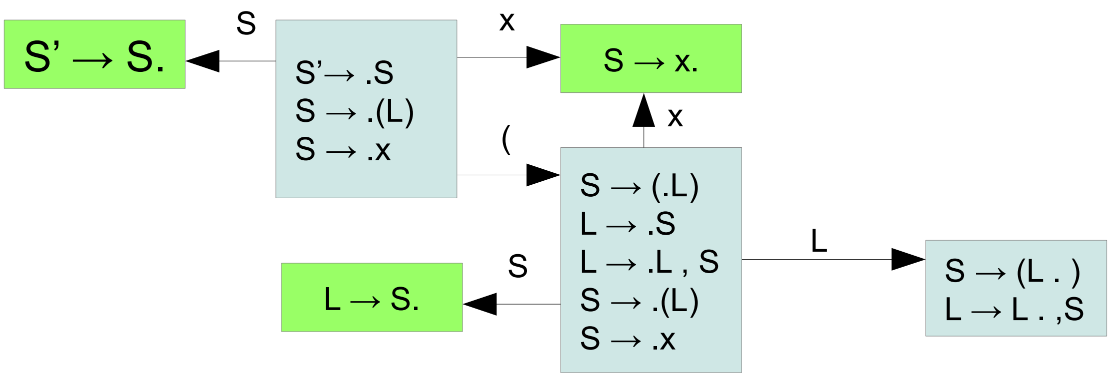


NTNU – Trondheim
Norwegian University of Science and Technology

S’ → S
S → (L)
S → x
L → S
L → L , S

More states, same work

- The remaining two items both suggest shifting an L
- That’s one transition, but both items can be a reason to take it

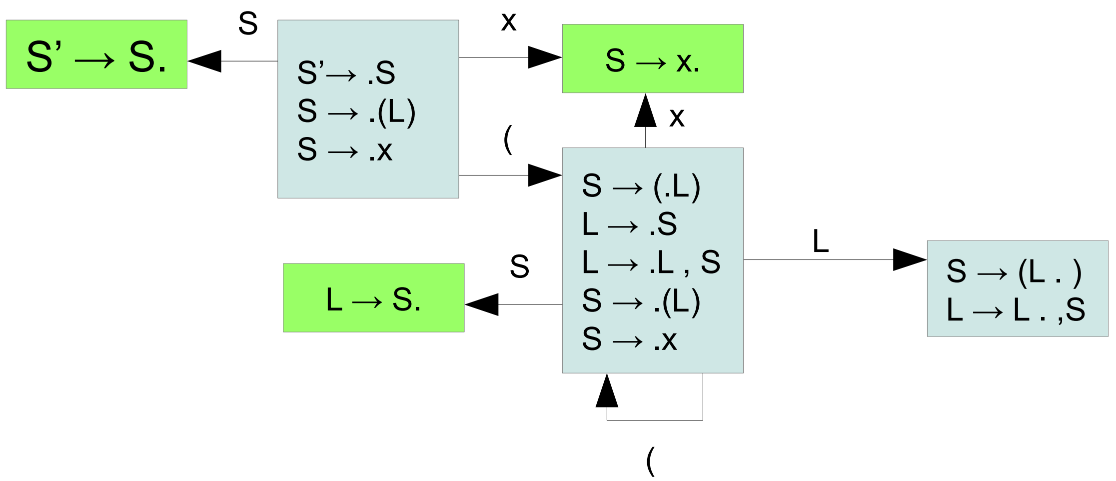


- The position marker doesn’t precede any nonterminals, so this is all the closure we need

S' → S
S → (L)
S → x
L → S
L → L , S

The nested parentheses

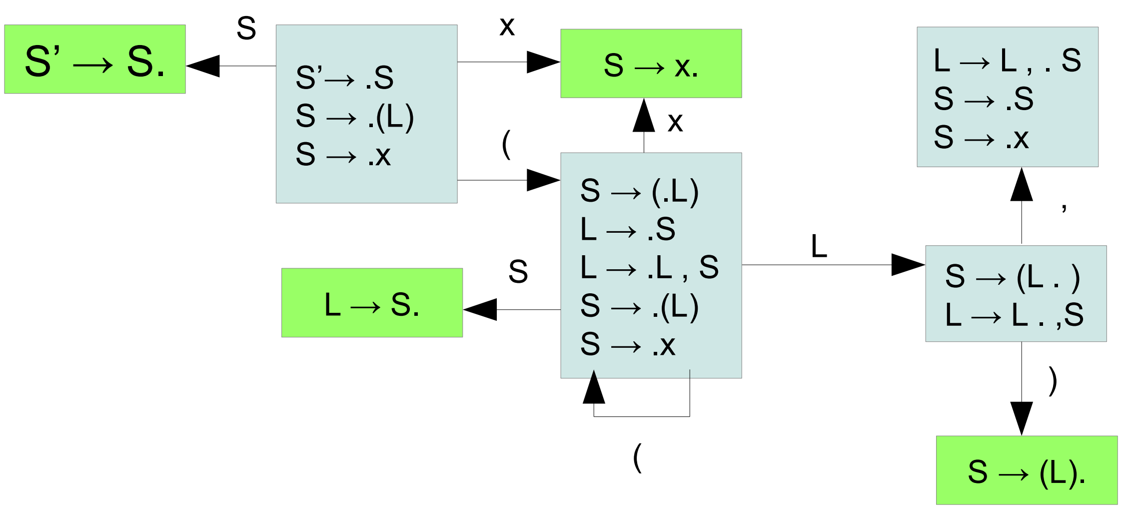
- Shifting for the item S → .(L) leads to S → (.L)
  - That's the item we created this state from
  - Thus, the transition must lead to the same state



S' → S
S → (L)
S → x
L → S
L → L , S

Two alternatives

- Shift a ‘)’ terminal, or a ‘,’ terminal
  - The ‘)’ leads to a reducing state
  - ‘,’ leads to an item we haven’t treated yet, so make a state, and find the closure it represents

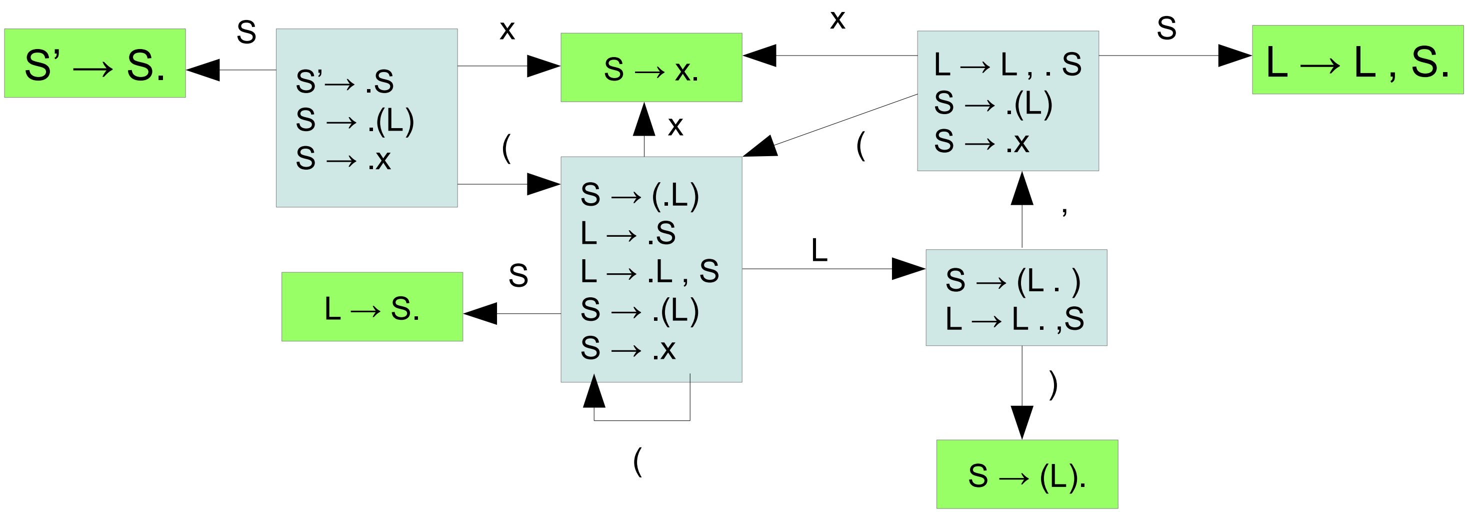


NTNU – Trondheim
Norwegian University of
Science and Technology

S' → S
S → (L)
S → x
L → S
L → L , S

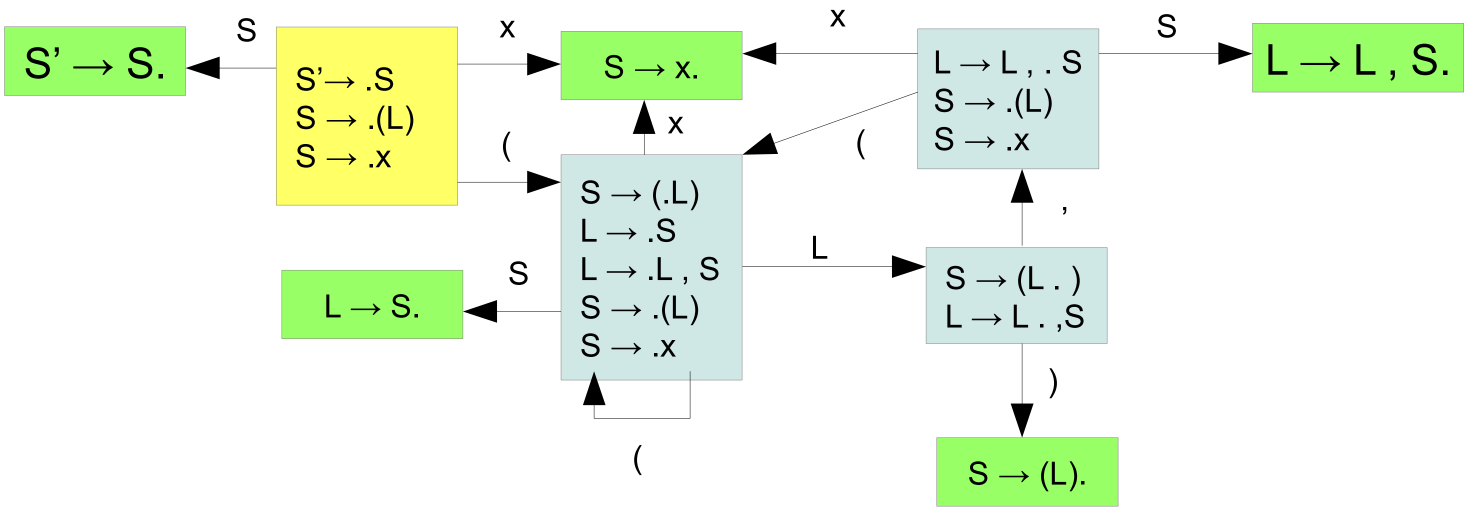
There's one more state

- The only new thing here is that we can finalize parsing of the production L → L , S
  - 'x' and '(' lead to states we have already created



That’s our LR(0) automaton

- There is a bit of window-dressing to transform it into a table, we can look at that next time



Just a final footnote

- As you may have noticed
  - Wrt. the Kleene closure of regex (r*), we saw that it's an infinite set
  - The epsilon closures have the added constraint that no character should be matched, so they become finite sets of states
  - Today's closures of items have the same constraint, for much the same reason
- There is a pattern here
  - The way we find FOLLOW sets in top-down parsing is the exact same principle at work too, I just didn't say it out loud at the time

The General Approach

(in a very pseudo-code way)

1) Initialize some number of sets
2) Update them so that they satisfy all constraints
3) Record whether any of them changed because of step 2
4) If any did, repeat from step 2
5) If none did, declare victory

- This is *“iteration to a fixed point”* (victory is the “fixed point”)
  - Calling it by that name foreshadows something deeper, every program can be rewritten as a constraint problem
  - We’re moving outside compiler construction here, so never mind
- I mention it still

  ...because recognizing this pattern on sight might make it easier to remember all our different variants.

NTNU – Trondheim
Norwegian University of Science and Technology