Warm Up Problem

Is the following grammar LL(1)? Why or why not?

$$
\begin{array}{lll}
S' \to \vdash S \dashv & & (0) \\
S \to cT & & (1) \\
S \to abc & & (2) \\
T \to T + U & & (3) \\
T \to d & & (4) \\
U \to a & & (5)
\end{array}
$$

Warm Up Problem

Is the following grammar LL(1)? Why or why not?

$$
\begin{array}{llll}
S' & \rightarrow & \vdash \; S \; \dashv & (0) \\
S & \rightarrow & cT & (1) \\
S & \rightarrow & abc & (2) \\
T & \rightarrow & T + U & (3) \\
T & \rightarrow & d & (4) \\
U & \rightarrow & a & (5)
\end{array}
$$

How does one make it right recursive?

Warm Up Problem

Is the following grammar LL(1)? Why or why not?

$$
\begin{array}{lll}
S' & \rightarrow \vdash S \dashv & (0) \\
S & \rightarrow cT & (1) \\
S & \rightarrow abc & (2) \\
T & \rightarrow T + U & (3) \\
T & \rightarrow d & (4) \\
U & \rightarrow a & (5)
\end{array}
$$

How does one make it right recursive?

Ans: Change Rules (3) and (4) to $T \rightarrow dT'$ and $T' \rightarrow +UT' \mid \epsilon$.

CS 241 Lecture 14

Bottom Up Parsing Continued

With thanks to Brad Lushman, Troy Vasiga and Kevin Lanctot

Recap

We want grammars that we can efficiently parse. Knuth in his paper *On the Translation of Languages from Left to Right* (1965) accomplished a few major milestones:

- Defined LR(k) grammars; grammars that can always be deterministically parsed left to right with right most derivations and a k symbol lookahead in the processed string.
- Argued that for such languages, parsing can be done efficiently (roughly proportional to the length of string).
- Argued that a language generated by a LR(k) grammar can also be generated by some LR(1) grammar
- and finally...

Major Theorem

Theorem (Knuth 1965)

For any grammar G, the set of viable prefixes (stack configurations), namely

$$\begin{aligned}\{\alpha a : & \alpha \in V^* \text{ is a stack} \\ & a \in \Sigma' \text{ is the next character} \\ & \exists x \in (\Sigma')^* \text{ such that } S \Rightarrow^* \alpha a x\}\end{aligned}$$

is a regular language and the NFA accepting it corresponds to items of G! (Recall $V = (N' \cup \Sigma')$, $N' = N \cup \{S'\}$ and $\Sigma' = \Sigma \cup \{\vdash, \dashv\}$). Converting this NFA to a DFA gives a machine with states that are the set of valid items for a viable prefix!

We will show how to use this theorem to create a LR(0), SLR(1) and LR(1) automata to help us accept the words generated by a grammar.

Construction

- We are actually going to construct a deterministic pushdown automaton, that is, an automaton that is basically a DFA with a stack.
- It turns out that grammars accepted by such machines are called LR(0) grammars (and these are ones we want to consider!)
- We can use this machinery to also handle LR(1) grammars *since we added the $\dashv$ symbol to our language!* This is beyond what we need to know but does justify the $\dashv$ symbol (it turns out that the $\vdash$ has no real utility for us... the $\dashv$ however means no accepted word will ever be the proper prefix of another word which is partly why everything works).

An Example (Thanks to Brad and Kevin for this example)

Consider the following context-free grammar:

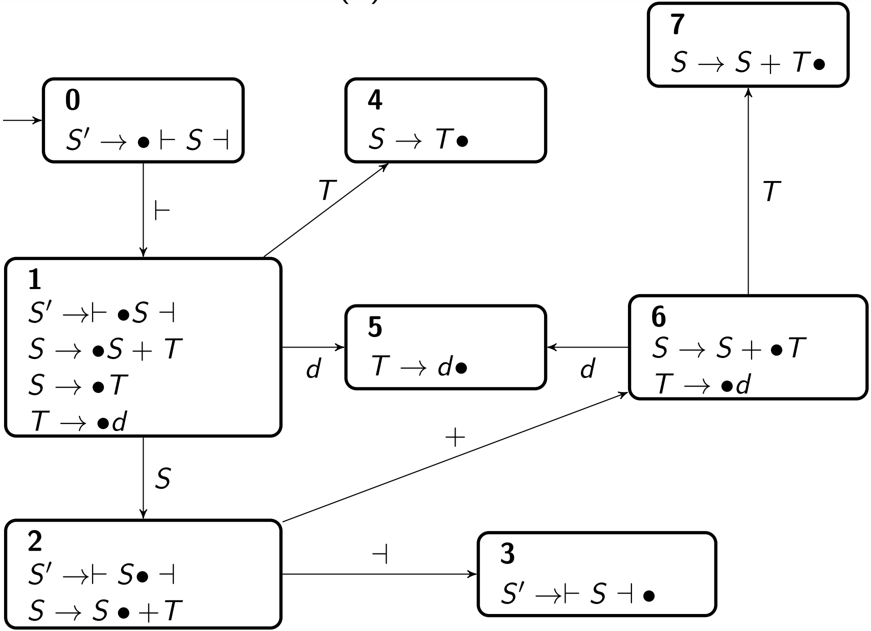
$$
\begin{array}{llll}
S' & \rightarrow & \vdash \; S \; \dashv & (0) \\
S & \rightarrow & S \; + \; T & (1) \\
S & \rightarrow & T & (2) \\
T & \rightarrow & d & (3)
\end{array}
$$

We construct the LR(0) automaton associated with this grammar.

LR(0) Construction

- From a state, for each rule in the state, move the dot forward by one character. The transition function is given by the symbol you jumped over.
- For example, with $S' \to \bullet \vdash S \dashv$, we move the $\bullet$ over $\vdash$. Thus, the transition function will consume the symbol $\vdash$.
- The state we end up in will contain the transition $S' \to \vdash \bullet S \dashv$. It also contains more!
- In the new state, if in the set of rules we have $\bullet A$ for some non-terminal A, we then add all rules with A in the left hand side of a production with a dot preceding the right hand side!
- In this case, this state will include the rules $S \to \bullet S + T$ and $S \to \bullet T$.
- Notice now we also have $\bullet T$ and so we also need to include the rules where T is the left hand side adding the rule $T \to \bullet d$.
- We continue on with these rules to get the final automaton.

New LR(0) Automaton



Using the Automaton

- Start in the start state with an empty stack.
- Shift:
  - Shift a character from input to the stack.
  - Follow any transition that follows with the character as a label.
  - If none, reduce or if not possible, give an error.
- Reduce:
  - Reduce states have only one item and the • is in the rightmost position.
  - Reduce the rule in the state: Pop the RHS off the stack and backtrack in your automaton the number of states corresponding to the number of elements in the RHS. Then follow the transition for the LHS and push the LHS on your stack.
- Accept if S' is on the stack and the input is empty.

NOTE: Because we need to backtrack, we also need to push the automaton's states on the stack as well! (Thus, accept when stack has the start state and S').

Example

Consider processing $\vdash d + d + d \dashv$:

Stack	Read	Processing	Action
0	ϵ	$\vdash d + d + d \dashv$	Shift $\vdash$, go to state 1
$0 \vdash 1$	$\vdash$	$d + d + d \dashv$	Shift d, go to state 5
$0 \vdash 1d5$	$\vdash d$	$+d + d \dashv$	Reduce $T \to d$. Pop one symbol and one state. Now in state 1. Push T go to state 4
$0 \vdash 1T4$	$\vdash d$	$+d + d \dashv$	Reduce $S \to T$. Pop one symbol and one state. Now in state 1. Push S go to state 2
$0 \vdash 1S2$	$\vdash d$	$+d + d \dashv$	Shift $+$, go to state 6
$0 \vdash 1S2 + 6$	$\vdash d+$	$d + d \dashv$	Shift d go to state 5
$0 \vdash 1S2 + 6d5$	$\vdash d + d$	$+d \dashv$	Reduce $T \to d$. Pop one symbol and one state. Now in state 6. Push T go to state 7
$0 \vdash 1S2 + 6T7$	$\vdash d + d$	$+d \dashv$	Reduce $S \to S + T$. Pop three symbols and three states. Now in state 1. Push S go to state 2
$0 \vdash 1S2$	$\vdash d + d$	$+d \dashv$	Shift $+$, go to state 6
$0 \vdash 1S2 + 6$	$\vdash d + d+$	$d \dashv$	Shift d go to state 5
$0 \vdash 1S2 + 6d5$	$\vdash d + d + d$	$\dashv$	Reduce $T \to d$. Pop one symbol and one state. Now in state 6. Push T go to state 7
$0 \vdash 1S2 + 6T7$	$\vdash d + d + d$	$\dashv$	Reduce $S \to S + T$. Pop three symbols and three states. Now in state 1. Push S go to state 2
$0 \vdash 1S2$	$\vdash d + d + d$	$\dashv$	Shift $\dashv$, go to state 3
$0 \vdash 1S2 \dashv 3$	$\vdash d + d + d \dashv$	ϵ	Reduce $S' \to \vdash S \dashv$. Pop three symbols and three states. Now in state 0. Push S' go to state 0
$0S'$	$\vdash d + d + d \dashv$	ϵ	Accept

Two Possible Issues

Issue one (Shift-Reduce):

- What if a state has two items of the form:

$$A \to \alpha \cdot a\beta$$
$$B \to \gamma\cdot$$

  where as always, $a \in \Sigma'$ and $\alpha, \beta, \gamma \in V^*$?

- Should we shift or reduce?

Issue two (Reduce-Reduce)

- What if a state has two items of the form:

$$A \to \alpha\cdot$$
$$B \to \gamma\cdot$$

- Which reduction should we do?

Two Possible Issues

Issue one (Shift-Reduce):

- What if a state has two items of the form:

$$A \to \alpha \cdot a\beta$$
$$B \to \gamma \cdot$$

  where as always, $a \in \Sigma'$ and $\alpha, \beta, \gamma \in V^*$?
- Should we shift or reduce?

Issue two (Reduce-Reduce)

- What if a state has two items of the form:

$$A \to \alpha \cdot$$
$$B \to \gamma \cdot$$

- Which reduction should we do?

Answer: Throw out all grammars that have these situations.

Definition

Definition

We say that a grammar is LR(0) if and only if after creating the automaton, no state has one of the aforementioned conflicts.

Practice: The example of bottom-up parsing form last lecture was LR(0)!

Question

Recall that LL(1) grammars were at odds with left recursive languages.

Question

Recall that LL(1) grammars were at odds with left recursive languages.

Are LR(0) grammars in conflict with a type of recursive language?

Question

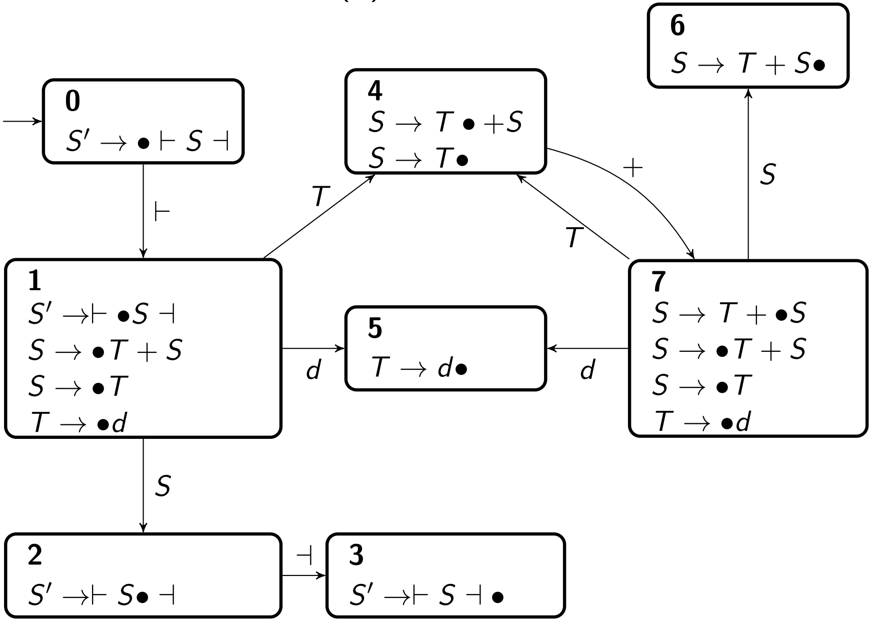
Recall that LL(1) grammars were at odds with left recursive languages.

Are LR(0) grammars in conflict with a type of recursive language?

Not in general! Bottom up parsing can support left and right recursive grammars! However, not all grammars are LR(0) grammars. Consider the following grammar (changed rule 1):

$$
\begin{array}{lllr}
S' & \to & \vdash S \dashv & (0) \\
S & \to & T + S & (1) \\
S & \to & T & (2) \\
T & \to & d & (3)
\end{array}
$$

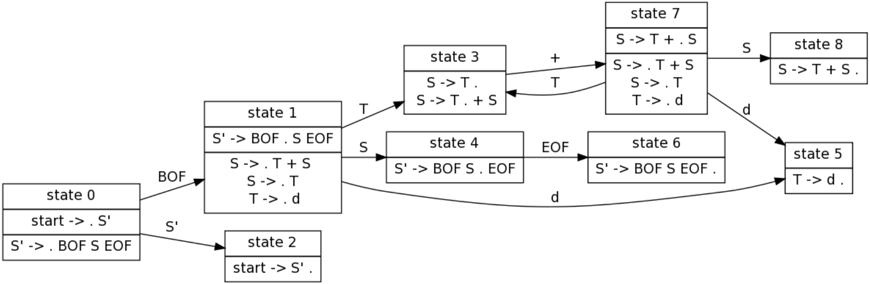
New LR(0) Automaton



Better Picture

Source: `http://smlweb.cpsc.ucalgary.ca/`

(Note: We don't have state 2 in our diagram).



Conflict

State 4 [state 3 in the second photo] has a shift-reduce conflict.

- Suppose the input began with $\vdash d$.

Conflict

State 4 [state 3 in the second photo] has a shift-reduce conflict.

- Suppose the input began with $\vdash d$.
- This gives a stack of $\vdash d$ and then we reduce in state 5 [state 5 in the second photo] so our stack changes to $\vdash T$ and we move to state 4 via state 1.

Conflict

State 4 [state 3 in the second photo] has a shift-reduce conflict.

- Suppose the input began with $\vdash d$.
- This gives a stack of $\vdash d$ and then we reduce in state 5 [state 5 in the second photo] so our stack changes to $\vdash T$ and we move to state 4 via state 1.
- Should we reduce $S \to T$?

Conflict

State 4 [state 3 in the second photo] has a shift-reduce conflict.

- Suppose the input began with $\vdash d$.
- This gives a stack of $\vdash d$ and then we reduce in state 5 [state 5 in the second photo] so our stack changes to $\vdash T$ and we move to state 4 via state 1.
- Should we reduce $S \to T$?
- It depends! If the input is $\vdash d \dashv$ then absolutely!
- If instead, the input was $\vdash d + \ldots$ then no!
- How do we fix this?

Lookahead!

We'll add a lookahead to the automaton to fix the conflict! For every $A \to \alpha\bullet$, attach Follow(A)! Recall:

$$\begin{array}{llll} S' & \to & \vdash\ S\ \dashv & (0) \\ S & \to & T\ +\ S & (1) \\ S & \to & T & (2) \\ T & \to & d & (3) \end{array}$$

What is Follow(S)? What about Follow(T)?

Follow Sets

Note that Follow(S) $= \{\dashv\}$ and Follow(T) $= \{+, \dashv\}$. So state 4 becomes

$$S \to T \bullet +S \qquad \text{and} \qquad S \to T \bullet \{\dashv\}$$

In other words, apply $S \to T \bullet +S$ if the next token is $+$ and apply $S \to T \bullet \{\dashv\}$ if the next token is $\dashv$.

We call these parsers SLR(1) parsers! (Simple LR with 1 character look ahead).

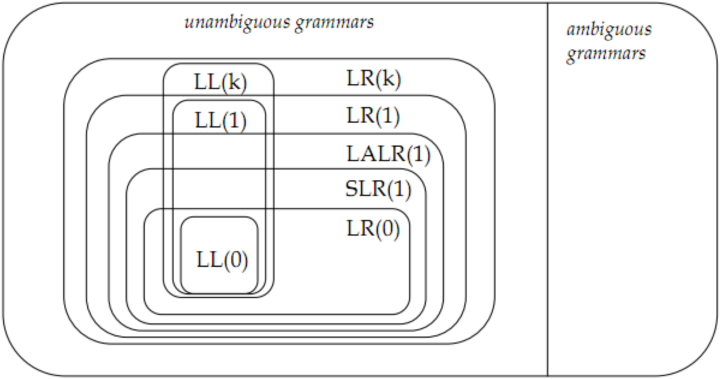
Wait... What is an LR(1) Parser?

- LR(1) parsing involves a more complicated procedure.
- Instead of adding all of Follow(S) to an item, you add only a subset of this set to each item.
- In this way, the number of states you get can blowup exponentially depending on your follow sets.
- However, the parsing mechanism is the same (just the automaton changes). Programming an SLR parser then swapping in an LR(1) automaton gives you an LR(1) parser.
- Yacc and Bison, for example, use LALR(1) (Lookahead LR) which lies somewhere between SLR(1) and LR(1).
- LR(1) parsers are extremely powerful; Knuth proved that if you have a language recognized by a LR(k) grammar for $k > 1$, then there is a LR(1) grammar recognizing the same language!

Grammar Picture

LL(1) versus LR(k)

A picture is worth a thousand words:



Source: `https://i.stack.imgur.com/TqAkP.png`

Recall: Every language accepted by a LR(k) grammar can be accepted by some LR(1) grammar!

Algorithm for LR(1) Automaton

Elements in an LR(1) automaton are items followed by a terminal symbol. Let S_M be the set of states of a automaton M

Algorithm 1 LR(1) Automaton Algorithm

Make the SLR(1) parser's automaton M
for each $A \to \alpha \bullet B\beta, \quad t$ line in a state s of S_M **do**
    for each $B \to \gamma$ in P **do**
        for each $b \in \text{First}(\beta t)$ **do**
            Add $B \to \bullet\gamma, b$ to s
        end for
    end for
end for
Repeat the above until no more changes have occurred.

Sample LR(1) Grammar That is Not SLR(1)

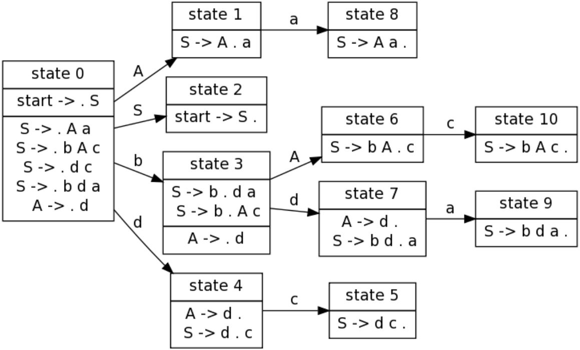
Omitting S' to be consistent with next pages notation.

$$
\begin{array}{rcll}
S & \rightarrow & Aa & (1) \\
S & \rightarrow & bAc & (2) \\
S & \rightarrow & dc & (3) \\
S & \rightarrow & bda & (4) \\
A & \rightarrow & d & (5)
\end{array}
$$

Source: `https://stackoverflow.com/questions/10505717/how-is-this-grammar-lr1-but-not-slr1`

SLR(1) Table

Source of next slides: `http://smlweb.cpsc.ucalgary.ca/`



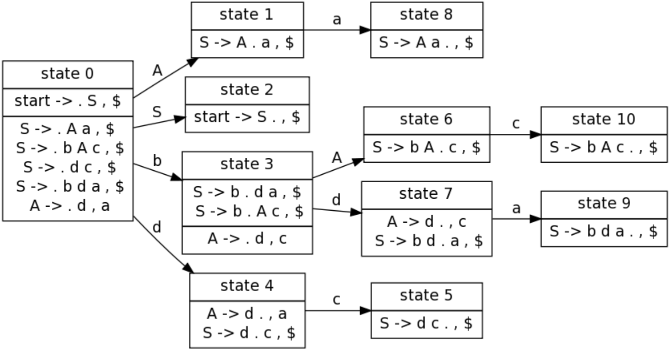
SLR(1) Table

SLR(1) Table

	$	d	a	b	c	S	A
0		s4		s3		s2	s1
1			s8				
2	acc						
3		s7					s6
4			r(A → d)		r(A → d)/s5		
5	r(S → d c)						
6					s10		
7			r(A → d)/s9		r(A → d)		
8	r(S → A a)						
9	r(S → b d a)						
10	r(S → b A c)						

LR(1) Table

Note $ means end of file.



LR(1) Table

LR(1) Table

	$	d	a	b	c	S	A
0		s4		s3		s2	s1
1			s8				
2	acc						
3		s7					s6
4			r(A → d)		s5		
5	r(S → d c)						
6					s10		
7			s9		r(A → d)		
8	r(S → A a)						
9	r(S → b d a)						
10	r(S → b A c)						

It is LR(1)