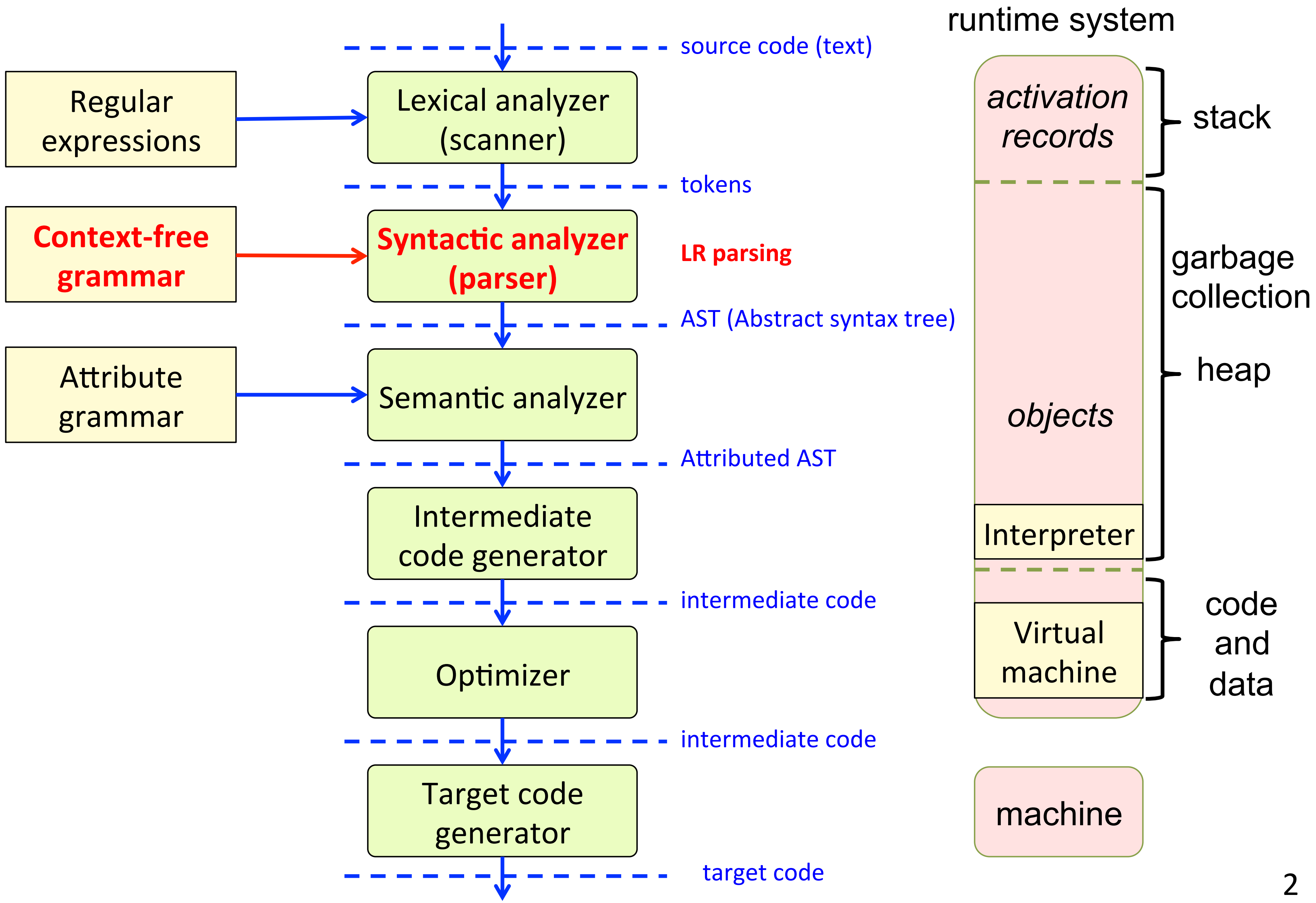
EDAN65: Compilers, Lecture 06 A

LR parsing

Görel Hedin

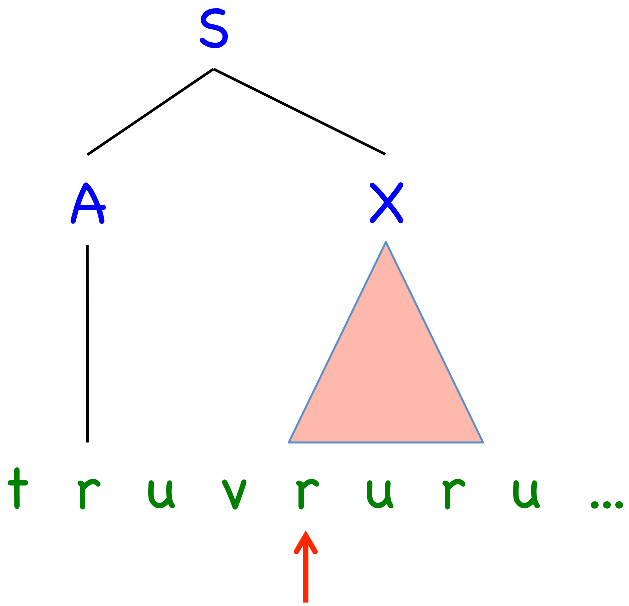
Revised: 2015-09-14

This lecture

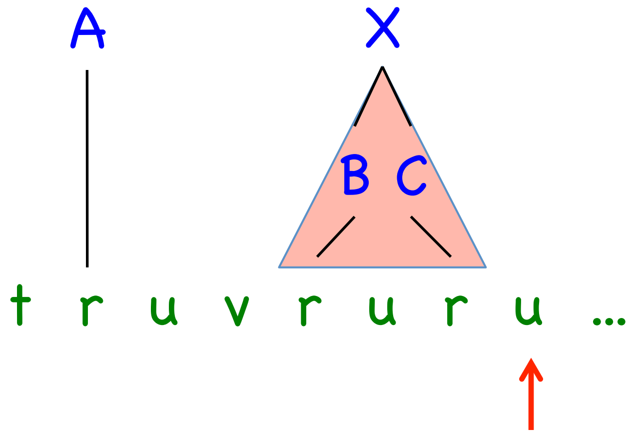


LR parsing

Recall main parsing ideas

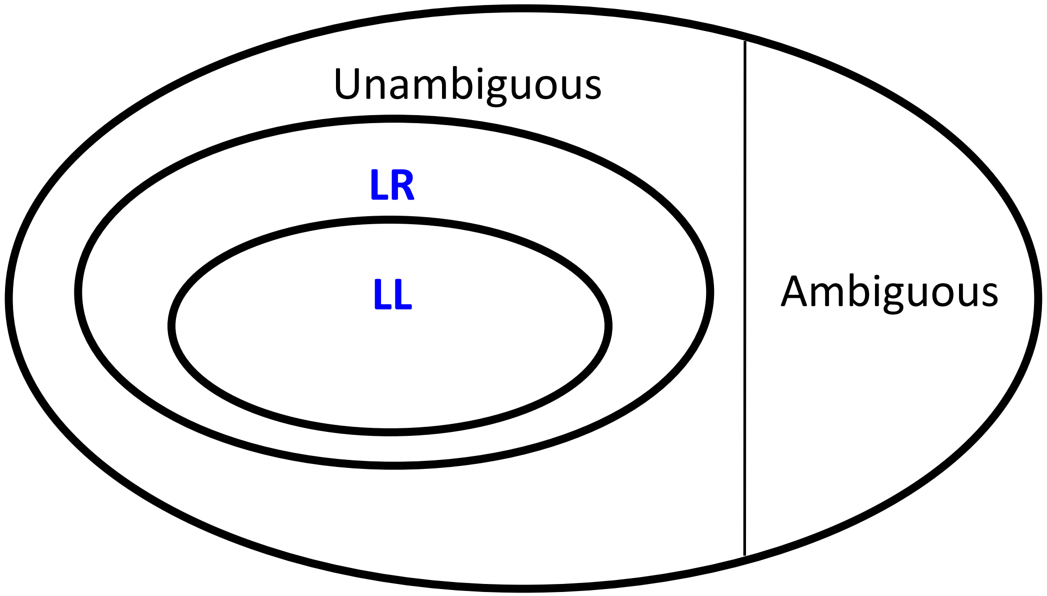


LL(1): decides to build Assign after seeing the first token of its subtree.
The tree is built top down.



LR(1): decides to build Assign after seeing the first token following its subtree.
The tree is built bottom up.

Recall different parsing algorithms



All context-free grammars

This lecture

LL:
Left-to-right scan
Leftmost derivation
Builds tree top-down
Simple to understand

LR:
Left-to-right scan
Rightmost derivation
Builds tree bottom-up
More powerful

Recall: LL(k) vs LR(k)

	LL(k)	**LR(k)**
Parses input	Left-to-right	
Derivation	Leftmost	Rightmost
Lookahead	k symbols	
Build the tree	top down	bottom up
Select rule	after seeing its first k tokens	after seeing all its tokens, and an additional k tokens
Left recursion	No	Yes
Unlimited common prefix	No	Yes
Resolve ambiguities through rule priority	Dangling else	Dangling else, associativity, priority
Error recovery	Trial-and-error	Good algorithms exist
Implement by hand?	Possible.	Too complicated. Use a generator.

LR parsing

Add the EOF token ($) and an extra start rule.

The parser uses a stack of symbols (terminals and nonterminals).

The parser looks at the current input token and decides to do one of the following actions:

- shift – Push the input token onto the stack. Read the next token.
- reduce –
  - Match the top symbols on the stack with a production right-hand side.
  - Pop those symbols and push the left-hand side nonterminal.
  - At the same time, build this part of the tree.
- accept – when the parser is about to shift $, the parse is complete.

The parses uses a finite state automaton (encoded in a table) to decide which action to take and which state to go to after each action.

LR parsing example

Grammar:

```
p0: S -> Stmt $
p1: Stmt -> ID ":=" Exp
p2: Exp -> ID
p3: Exp -> Exp "+" ID
```

Input:

```
ID := ID + ID $
```

Tree:	Stack:	Input:	Action:
		ID := ID + ID $	

shift: push input to stack, read next token **reduce**: pop production rhs, push lhs

LR parsing example

Grammar:

p0: S -> Stmt $
p1: Stmt -> ID ":=" Exp
p2: Exp -> ID
p3: Exp -> Exp "+" ID

Input:

ID := ID + ID $

Tree:	Stack:	Input:	Action:
		ID := ID + ID $	shift ID
ID	ID	:= ID + ID $	shift :=
:=	ID :=	ID + ID $	shift ID
ID	ID := ID	+ ID $	reduce Exp -> ID
Exp	ID := Exp	+ ID $	shift +
+	ID := Exp +	ID $	shift ID
ID	ID := Exp + ID	$	reduce Exp -> Exp "+" ID
Exp	ID := Exp	$	reduce Stmt -> ID ":=" Exp
Stmt	Stmt	$	accept

Follow the reduction steps in reverse order. They correspond to a rightmost derivation.

LR(1) items

The parser uses a DFA (a deterministic finite automaton) to decide whether to shift or reduce.

The states in the DFA are sets of *LR items*.

LR(1) item:

$X \text{ -> } \alpha \bullet \beta \qquad t,s$

An LR(1) item is a production extended with:

- A dot (•), corresponding to the position in the input sentence.
- One or more possible lookahead terminal symbols, t,s
  (we will use ? when the lookahead doesn't matter)

The LR(1) item corresponds to a state where:

- The topmost part of the stack is α.
- The first part of the remaining input is expected to match $\beta(t|s)$

Grammar:

```
p0: S -> E $
p1: E -> T "+" E
p2: E -> T
p3: T -> ID
```

Constructing state 1

```
S -> • E $          ?
```

First, take the start production and place the dot in the beginning...

```
S -> • E $          ?
E -> • T "+" E      $
E -> • T            $
```

Note that there is a nonterminal E right after the dot, and it is followed by a terminal $. Add the productions for E, with $ as the lookahead.

```
1   S -> • E $          ?
    E -> • T "+" E      $
    E -> • T            $
    T -> • ID           +,$
```

Note that there is a nonterminal T right after the dot, and which is followed by either "+" or $. Add the productions for T, with "+" and $ as the lookahead. (We write them on the same line as a shorthand.)

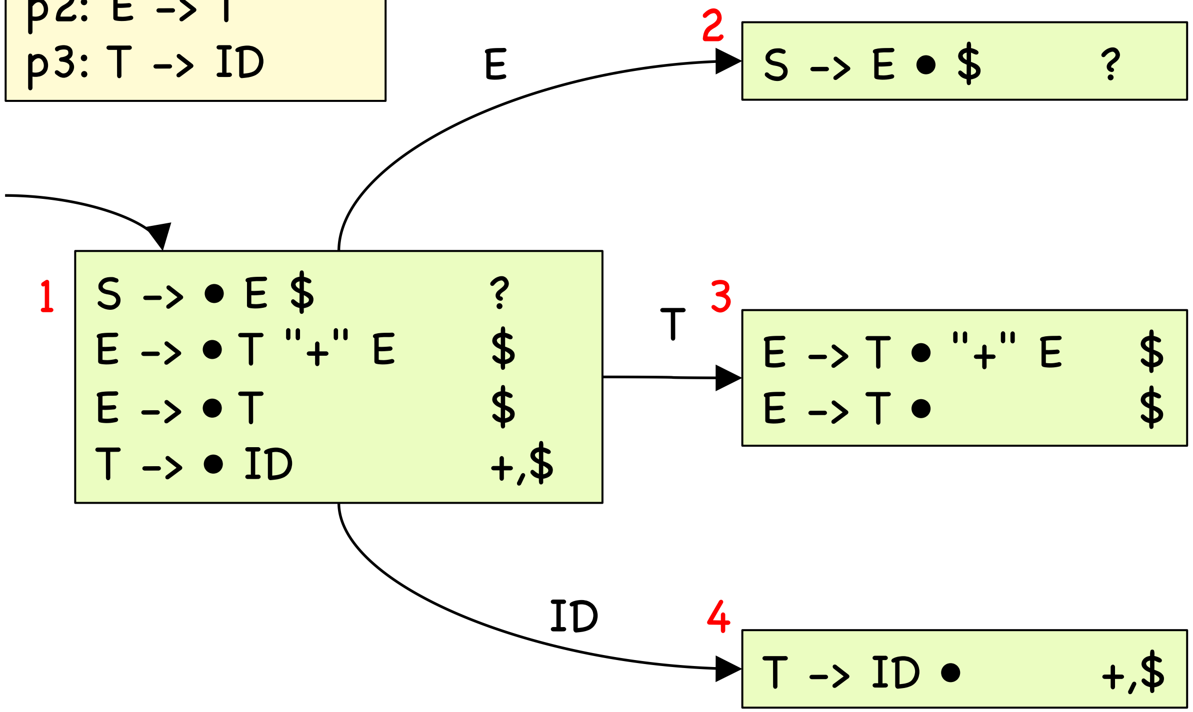
We have already added productions for all nonterminals that are right after the dot. Nothing more can be added. We are finished constructing state 1.

Adding new productions for nonterminals following the dot, until no more productions can be added, is called taking the closure of the LR item set.

Constructing the next states

Grammar:

```
p0: S -> E $
p1: E -> T "+" E
p2: E -> T
p3: T -> ID
```

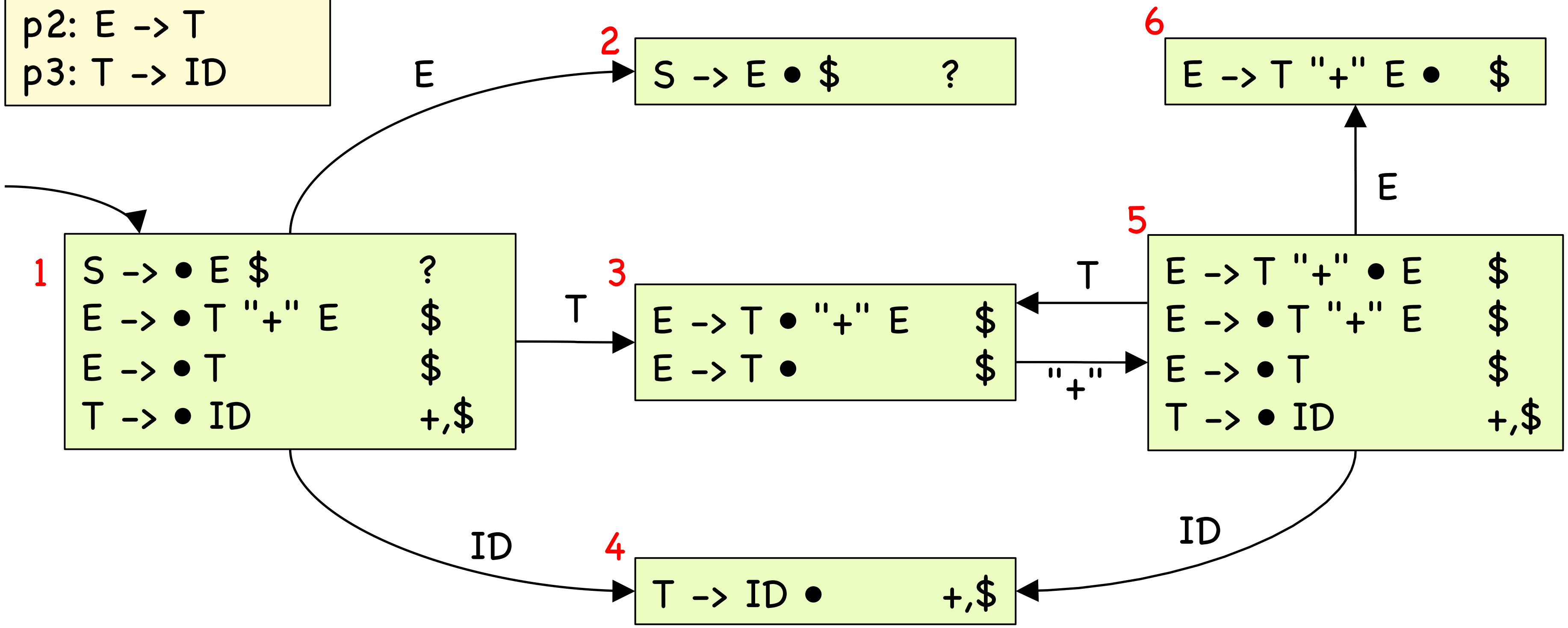


Note that the dot is followed by E, T, and ID in state 1. For each of these symbols, create a new set of LR items, by advancing the dot passed that symbol. Then complete the states by taking the closure. (Nothing had to be added for these states.)

Completing the LR DFA

Grammar:

p0: S -> E $
p1: E -> T "+" E
p2: E -> T
p3: T -> ID



Complete the DFA by advancing the dot, creating new states, completing them by taking the closure. If there is already a state with the same items, we use that state instead.

Constructing the LR table

- For each **token edge t**, from state j to state k, add a **shift action** "s k" (shift and goto state k) to table[j,t].
- For each **nonterminal edge X**, from state j to state k, add a **goto action** "g k" (goto state k) to table[j,X].
- For a state j containing an LR item with **the dot to the left of $**, add an **accept action** "a" to table[j,$]
- For each state j that contains an LR item **where the dot is at the end**, add a **reduce action** "r p" (reduce p) to table[j,t], where p is the production and t is the lookahead token.

state	"+"	ID	$		E	T
1						
2						
3						
4						
5						
6						

Constructing the LR table

- For each **token edge t**, from state j to state k, add a **shift action** "s k" (shift and goto state k) to table[j,t].
- For each **nonterminal edge X**, from state j to state k, add a **goto action** "g k" (goto state k) to table[j,X].
- For a state j containing an LR item with **the dot to the left of $**, add an **accept action** "a" to table[j,$]
- For each state j that contains an LR item **where the dot is at the end**, add a **reduce action** "r p" (reduce p) to table[j,t], where p is the production and t is the lookahead token.

state	"+"	ID	$		E	T
1		s 4			g 2	g 3
2			a			
3	s 5		r p2			
4	r p3		r p3			
5		s 4			g 6	g 3
6			r p1			

Using the LR table for parsing

- Use a symbol stack and a state stack
- The current state is the state stack top.
- Push state 1 to the state stack
- Perform an action for each token:
- Case Shift s:
  - Push the token to the symbol stack
  - Push s to the state stack
  - The current state is now s.
- Case Reduce p:
  - Pop symbols for the rhs of p
  - Push the lhs symbol X of p
  - Pop the same number of states
  - Let s1 = the top of the state stack
  - Let s2 = table[s1,X]
  - Push s2 to the state stack
  - The current state is now s2.
- Case Accept: Report successful parse

state	"+"	ID	$		E	T
1		s 4			g 2	g 3
2			a			
3	s 5		r p2			
4	r p3		r p3			
5		s 4			g 6	g 3
6			r p1			

Example of LR parsing

Grammar:

p0: S -> E $
p1: E -> T "+" E
p2: E -> T
p3: T -> ID

Parse table:

state	"+"	ID	$		E	T
1		s 4			g 2	g 3
2			a			
3	s 5		r p2			
4	r p3		r p3			
5		s 4			g 6	g 3
6			r p1			

Parsing ID + ID $

State stack	Symbol stack	Input	action
1		ID + ID $	

Example of LR parsing

Grammar:

p0: S -> E $
p1: E -> T "+" E
p2: E -> T
p3: T -> ID

Parse table:

state	"+"	ID	$		E	T
1		s 4			g 2	g 3
2			a			
3	s 5		r p2			
4	r p3		r p3			
5		s 4			g 6	g 3
6			r p1			

Parsing ID + ID $

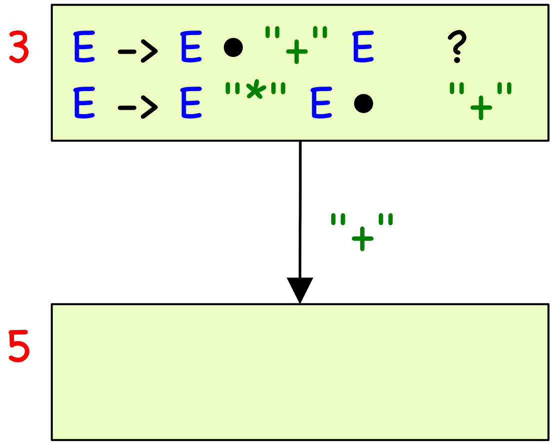
State stack	Symbol stack	Input	action
1		ID + ID $	shift 4
1 4	ID	+ ID $	reduce p3
1 3	T	+ ID $	shift 5
1 3 5	T +	ID $	shift 4
1 3 5 4	T + ID	$	reduce p3
1 3 5 3	T + T	$	reduce p2
1 3 5 6	T + E	$	reduce p1
1 2	E	$	accept

Conflict in an LR table

Grammar:

```
p0: S -> E $
p1: E -> E "+" E
p2: E -> E "*" E
p3: E -> ID
```

Parts of the DFA:



Parts of the parse table:

state	...	"+"	...		...	...
...						
3						
...						

Fill in the parse table.

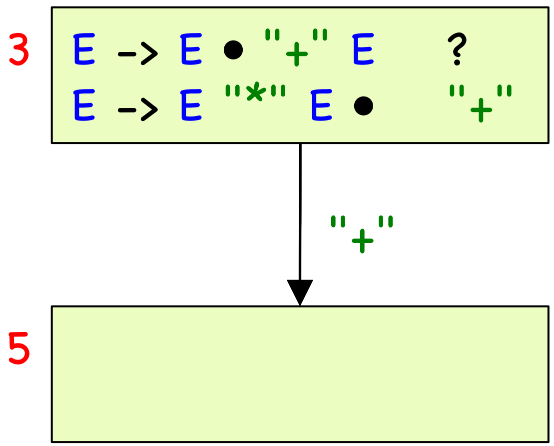
What is the problem?

Conflict in an LR table

Grammar:

p0: S -> E $
p1: E -> E "+" E
p2: E -> E "*" E
p3: E -> ID

Parts of the DFA:



Parts of the parse table:

state	...	"+"	...		...	...
...						
3		s 5, r p2				
...						

There is a shift-reduce conflict.
The grammar is ambiguous.
In this case, we can resolve the conflict by selecting one of the actions.

To understand which one, think about what the top of the stack looks like.
Think about what will happen later if we take the shift rule or the reduce rule.

Analyzing LR conflicts ...

Example output from parser generator (CUP):

```
...
*** Shift/Reduce conflict found in state #5
  between expr ::= expr PLUS expr (*)
  and       expr ::= expr (*) PLUS expr
  under symbol PLUS
  Resolved in favor of shifting.
...
```

Note! The dot is written as "(*)".

Note! The parser generator automatically resolves the conflict by shifting.

Is this what we want???

Line up the dots in the state:

```
expr -> expr PLUS expr •
expr ->           expr • PLUS expr
```

The top of stack and input may look like:

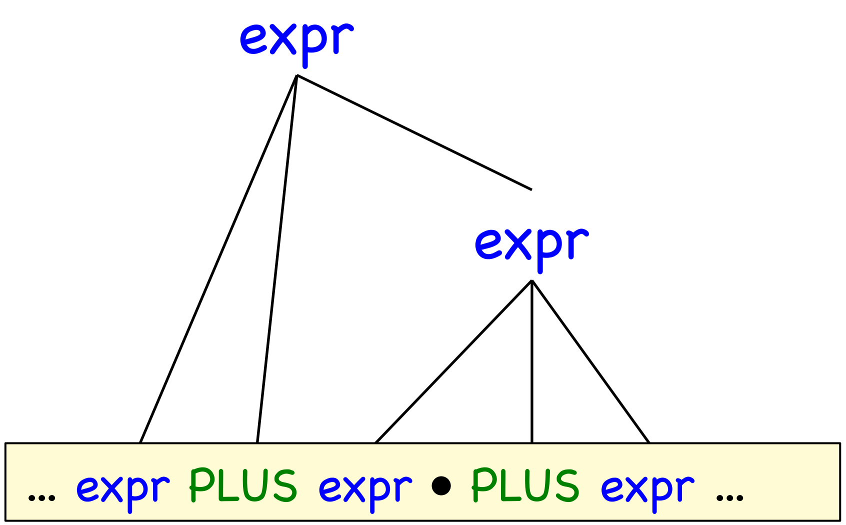
```
... expr PLUS expr • PLUS expr ...
```

top of stack   remaining input

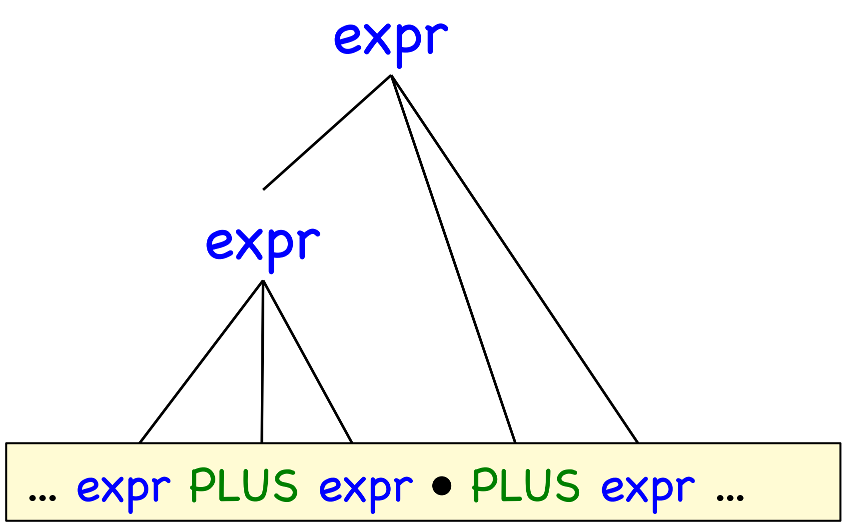
... Analyzing LR conflicts

Line up the dots in the state:

```
expr -> expr PLUS expr •                  PLUS
expr ->           expr • PLUS expr        ?
```



If we shift



If we reduce

Which rule should we choose?

Different kinds of conflicts

```
E -> E • "+" E     ?
E -> E "*" E •     "+"
```

A shift-reduce conflict.

```
A -> B C •         t
D -> C •           t
```

A reduce-reduce conflict.

Shift-reduce conflicts can sometimes be solved with *precedence rules*.
In particular for binary expressions with priority and associativity.

For other cases, you need to carefully analyze the shift-reduce conflicts to see if precedence rules are applicable, or if you need to change the grammar.

For reduce-reduce conflicts, it is advisable to think through the problems, and change the grammar.

Typical precedence rules for an LALR parser generator

```
E -> E "==" E
E -> E "**" E
E -> E "*" E
E -> E "/" E
E -> E "+" E
E -> E "-" E
E -> ID
E -> INT

precedence nonassoc EQ
precedence left PLUS, MINUS
precedence left TIMES, DIV
precedence right POWER
```

Shift-reduce conflicts are automatically resolved using the precedence rules.

Operators in the same rule have the same priority (e.g., PLUS, MINUS).

Operators in a later rule have higher priority (e.g. TIMES has higher prio than PLUS.)

How the precedence rules work

A rule is given the priority and associativity of its rightmost token.

For two conflicting rules with different priority, the rule with the highest priority is chosen:

```
E -> E • + E      ?
E -> E * E •      +
```

Reduce is chosen

```
E -> E • * E      ?
E -> E + E •      *
```

Shift is chosen

Two conflicting rules with the same priority have the same associativity.
Left-associativity favors reduce.
Right-associativity favors shift.
Non-associativity removes both rules from the table (input following that pattern will cause a parse error).

```
E -> E + E •      +
E -> E • + E      ?
```

Reduce is chosen

```
E -> E ** E •     **
E -> E • ** E     ?
```

Shift is chosen

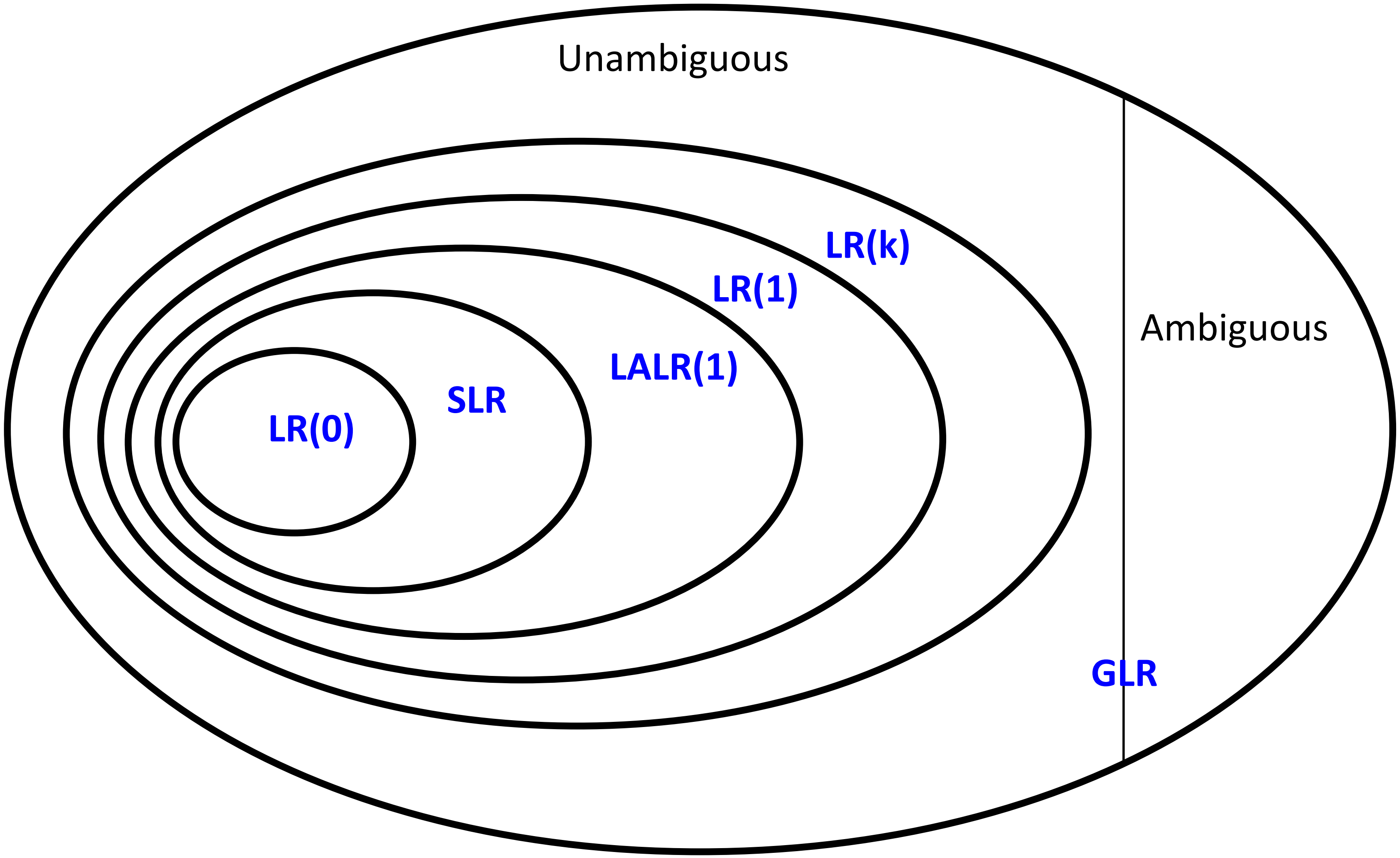
```
E -> E == E •     ==
E -> E • == E     ?
```

No rule is chosen

Different variants of LR(k) parsers

Type	Characteristics
LR(0)	LR items without lookahead. Not very useful in practice.
SLR Simple LR	Look at the FOLLOW set to decide where to put reduce actions. Can parse some useful grammars.
LALR(1) used in practice	Merges states that have the same LR items, but different lookaheads (LA). Leads to much smaller tables than LR(1). Used by most well known tools: Yacc, CUP, Beaver, SableCC, ... Sufficient for most practical parsing problems.
LR(1)	Slightly more powerful than LALR(1). Not used in practice – the tables become very large.
LR(k)	Much too large tables for k>1

Different variants of LR parsers



All context-free grammars

Universal parsing algorithms

GLR – Generalized LR

Can parse *any* context free grammar.

Including *ambiguous* grammars!

Returns a parse *forest* (all possible parse trees).
Additional mechanism needed to select which of the trees to use.

Can parse grammars with shift-reduce and reduce-reduce conflicts
(spawns parallel parsers).

Has cubic worst-case complexity (in the length of the input).

Is often much better than that in practice. But still slower than LALR.

Used in several research systems.

Some well-known parser generators

Name	Type, host language
JavaCC	LL, Java
ANTLR	LL, Java (also earlier versions for C, C#, ...)
yacc	LALR, C, "yet another compiler compiler" Developed for AT&T Unix in 1970.
bison	LALR, C++, GNU GPL
CUP	LALR, Java
beaver	LALR, Java
SDF/SGLR	Scannerless GLR, C, Java

For more examples, see
http://en.wikipedia.org/wiki/Comparison_of_parser_generators

Summary questions: LR parsing

- How does LR differ from LL parsers?
- What does it mean to shift?
- What does it mean to reduce?
- Explain how LR parsing works on an example.
- What is an LR item?
- What does an LR state consist of?
- What does it mean to take the closure of a set of LR items?
- What do the edges in an LR DFA represent?
- How can an LR table be constructed from an LR DFA?
- How is the LR table used for parsing?
- What is meant by a shift-reduce conflict and a reduce-reduce conflict?
- How can such a conflict be analyzed?
- How can precedence rules be used in an LR parser?
- What is LR(0) and SLR parsing?
- What is the difference between LALR(1) and LR(1)?
- Explain why the LALR(1) algorithm is most commonly used in parser generators.
- What is a GLR parser?