LL grammars

recursive descent parsers

LL parsers

LL grammars are those than can be parsed using a LL parser ⇒ parses by scanning the input from **Left** to right and building a **Leftmost** derivation

They can be parsed by a (top-down) *recursive descent parser* ; LL(k) grammars correspond to parser that take their decision based on a look-ahead of k symbols (and without backtracking)

We look at an example of LL(1) grammars next

LL parsers: example

1. $S \rightarrow a\, S\, b\, T$
2. $S \rightarrow c\, T$
3. $S \rightarrow d$
4. $T \rightarrow a\, T$
5. $T \rightarrow b\, S$
6. $T \rightarrow c$

$$w = a\ c\ c\ b\ b\ a\ d\ b\ c$$

$$S \rightarrow a \circ S\ b\ T \qquad (1)$$

LL parsers

1. $S \rightarrow a\, S\, b\, T$
2. $S \rightarrow c\, T$
3. $S \rightarrow d$
4. $T \rightarrow a\, T$
5. $T \rightarrow b\, S$
6. $T \rightarrow c$

$w = a\ c\ c\ b\ b\ a\ d\ b\ c$

$$
\begin{aligned}
S &\rightarrow a \circ S\, b\, T && (1)\\
&\rightarrow a\, c \circ T\, b\, T && (2)\\
&\rightarrow a\, c\, c \circ b\, T && (6)\\
&\rightarrow a\, c\, c\ b \circ T && \\
&\rightarrow a\, c\, c\ b\, b \circ S && (5)\\
&\rightarrow a\, c\, c\ b\, b\, a \circ S\, b\, T && (1)\\
&\rightarrow a\, c\, c\ b\, b\, a\, d \circ b\, T && (3)\\
&\rightarrow a\, c\, c\ b\, b\, a\, d\, b \circ T && \\
&\rightarrow a\, c\, c\ b\, b\, a\, d\, b\, c \circ \epsilon && (6)
\end{aligned}
$$

Questions

- What is a suitable *accepting device* for this example ?
- How can I check that my grammar is LL(1) ?
- If it is not, is there a way to repair it ?

LL Parser

At each step we have a derivation of the form $u \circ \alpha$ where u is a prefix of w of length i ($u = w[:i]$)

$\Rightarrow$ we match the suffix $w[i:]$ with α

We decide what rule to match by looking at the next symbol (say $w[i+1] = a$)

$\Rightarrow$ the choice should be unique, depending only on the top symbol ($w[i]$) and the start of α

$\Rightarrow$ we could encode this “ function” in a table

LL Parser

At each step, we try to match a suffix, $w[i:]$, with a pattern α

(SHIFT) $\alpha = b\,\gamma$ and $w[i] = b$

we try to match word $w[i+1:]$ with pattern γ

(REDUCE) $\alpha = X\,\gamma$

we need to match symbol a with $X \to \beta$

we continue with $w[i:]$ and the pattern $\beta\,\gamma$

(STOP) we matched the whole word and $\alpha = \epsilon$,

or when we have no rules to match (ERROR)

LL Parser: amelioration

To make sure we match the axiom, S, we add a new symbol, $\$$, and a new top-level axiom rule $S' \to S\ \$$

$\Rightarrow$ the initial pattern is $S\ \$$

Possible cases for errors are:

- we “shift” a bad symbol: $\alpha = b\ \gamma$ and $w[i] \neq b$
- we reach the end of the input ($\$$) and $\alpha \neq \$$
- we reach the end of the pattern $\alpha = \$$ and $w[i] \neq \$$

Parsing Table

$$S' \rightarrow S\,\$$$
$$S \rightarrow a\,S\,b\,T \mid c\,T \mid d$$
$$T \rightarrow a\,T \mid b\,S \mid c$$

To match a symbol a, and a non-terminal, X, to a rule, $X \rightarrow \beta$, we assume that we computed a parsing table

	a	b	c	d
S	$a\,S\,b\,T$		$c\,T$	d
T	$a\,T$	$b\,S$	c	

	a	b	c	d
S	$a\,S\,b\,T$		$c\,T$	d
T	$a\,T$	$b\,S$	c	

$$S' \rightarrow S\,\$$$
$$S \rightarrow a\,S\,b\,T \mid c\,T \mid d$$
$$T \rightarrow a\,T \mid b\,S \mid c$$

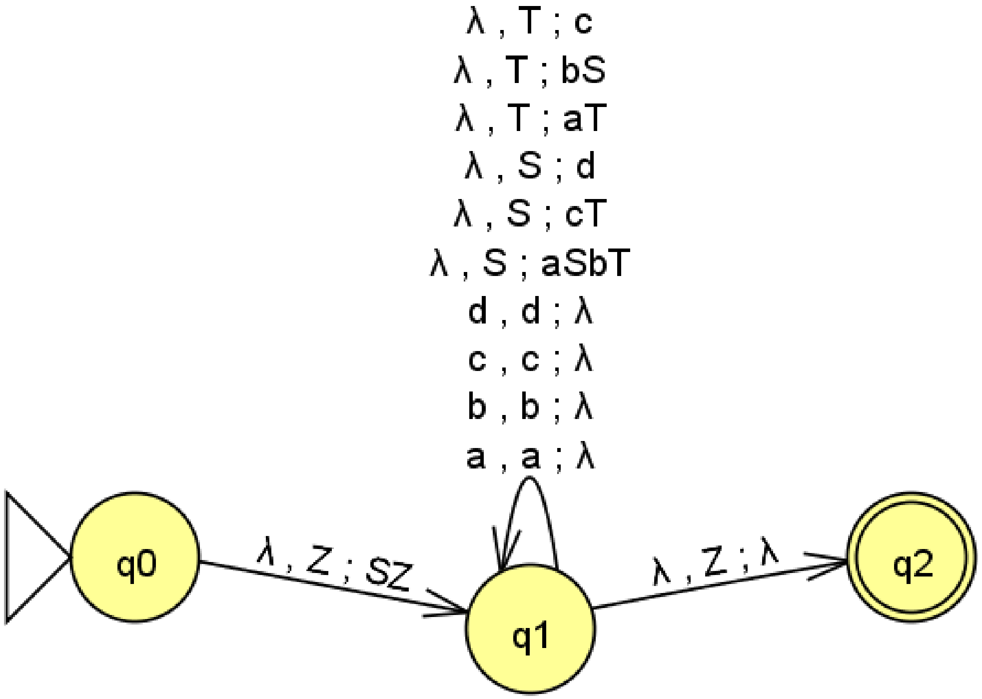
$a\ c\ c\ b\ a\ c\ \$$	parse successful
$a\ a\ b\ c\ d\ d\ d\ \$$	illegal input
$a\ a\ d\ c\ a\ a\ c\ c\ \$$	illegal input

Recursive descent parser

```
func member(u []byte) bool {
    st, i := stack("S", "$"), 0
    for {
      if i == len(u) || len(stack) == 0 {
        return false
      }
      α := stack.pop()
      switch {
        case α == "$" && u[i] == "$":  // accept
            return true
        case α == u[i]: i++              // shift
        case α.(nterm):                  // reduce
            stack := stack.push(reduce(α, u[i]))
        default : return false           // error
} } }
```

Recursive parser

We shall see that *(deterministic) Pushdown Automata* provide an adequate notion of accepting devices for LL grammars



	a	b	c	d
S	$a\ S\ b\ T$		$c\ T$	d
T	$a\ T$	$b\ S$	c	

LL grammars

building a parsing table

Questions

- What is a suitable *accepting device* for this example ?

→ How can I check that my grammar is LL(1) ?

- If it is not, is there a way to repair it ?

Building the parsing table

A grammar is LL

$\Longleftrightarrow$ we can build a LL parser from it

$\Longleftrightarrow$ we can build a (LL) parsing table

Next we show how to build this table by computing three different relations: FIRST, NULL and FOLLOW

LL Parser: FIRST

To build a table, we need to know: what symbols can "appear first", at the beginning of a non-terminal X and to which production $X \to \alpha$ it belongs

E.g. we want to match $a\ w$ with pattern $X\ \gamma$ and we have a rule $X \to a\ Y$

Also, we should not have $X \to a\ Y$ and $X \to a\ Z$

LL Parser: NULL

Therefore we should also know when a non-terminal X is *nullable*, that is $X \Rightarrow^{\star} \epsilon$

E.g. we want to match a with pattern $X\ \gamma$ and we are in a situation where $X \Rightarrow^{\star} \epsilon$

Also, we should not match symbol a with Z when $Z \rightarrow X\ Y$ with $X \rightarrow a\ \gamma \mid \Lambda$ and $Y \rightarrow a\ \gamma'$

LL Parser: FOLLOW

Meaning, we should know the symbols that can follow a non-terminal X.

E.g. when we want to match symbol a with pattern $X\ Y$, a possible solution is that $X \Rightarrow^{\star} \epsilon$ and $Y \Rightarrow^{\star}\ a\ \gamma$

NULL, FIRST and FOLLOW

We have FIRST(α) $= \{ b \in \Sigma \mid \alpha \Rightarrow^\star b\,\gamma \}$

We say that null(α) when $\alpha \Rightarrow^\star \epsilon$

this is decidable

We have FOLLOW(X) $= \{ a \in \Sigma \mid S \Rightarrow^\star \beta\, X\, a\, \gamma \}$

Ambiguity $\Rightarrow$ we should not find two rules
$X \rightarrow \alpha$ and $X \rightarrow \beta$ such that
FIRST(α) $\cap$ FIRST(β) $\neq \emptyset$

a FIRST-FIRST conflict

Ambiguity revisited

Actually, we can prove that the grammar is LL(1) when, for every non-terminal X with productions $X \ \rightarrow \alpha_1 \mid \ \ldots \ \mid \alpha_n$, we have that:

For every pair $X \ \rightarrow \alpha$ and $X \ \rightarrow \beta$ we have FIRST$(\alpha) \cap$ FIRST$(\beta) = \emptyset$

no FIRST-FIRST conflicts

if NULL(X) then FIRST$(\alpha_i) \cap$ FOLLOW$(X) = \emptyset$

no FIRST-FOLLOW conflicts

LL Parser: FIRST

We have FIRST$(\alpha) = \{ b \in \Sigma \mid \alpha \Rightarrow^\star \; b\,\gamma \}$

$$\text{FIRST}(\epsilon) = \emptyset$$

$$\text{FIRST}(a) = \{ a \}$$

$$\text{FIRST}(\alpha_1 \; \ldots \alpha_n) = \; \bigcup_{i \in 1..n} \{ \text{FIRST}(\alpha_i) \mid \text{null}(\alpha_j), j < i \}$$

Equivalently: FIRST is the smallest relation such that $X \; \to Y_1 \; \ldots Y_n \, Z \, \beta$ implies FIRST$(Z) \subseteq$ FIRST(X) when $Y_1, \ldots, Y_n$ are all nullable.

LL Parser: FOLLOW

We have $\text{FOLLOW}(X) = \{\, a \in \Sigma \mid S \Rightarrow^{\star} \beta\ X\ a\ \gamma \,\}$

(and we assume $\text{FOLLOW}(S) \supseteq \{\ \$\ \}$)

FOLLOW is the smallest relation such that:

$A \rightarrow \alpha\ X\ Y_1\ \dots Y_n\ Z\ \beta$ implies
$\text{FIRST}(Z) \subseteq \text{FOLLOW}(X)$ when $Y_1, \dots, Y_n$ nullable.

$A \rightarrow \alpha\ X\ Y_1\ \dots Y_n\ Z$ implies
$\text{FOLLOW}(A) \subseteq \text{FOLLOW}(X)$ when $Y_1, \dots, Y_n$ nullable.

Symboles Directeurs (SD)

Dans les notations utilisées à l'ENSEEIHT, on fait usage de la notion de *symbole directeur* pour une production $X \ \rightarrow \alpha$.

$\text{SD}(X \ \rightarrow \alpha) = \text{FIRST}(\alpha)$ si $\alpha \neq \Lambda$

$\text{SD}(X \ \rightarrow \Lambda) = \text{FOLLOW}(X)$

conflits LL $\equiv$ le même symbole dans deux règles

$\text{SD}(X \ \rightarrow \alpha)$ et $\text{SD}(X \ \rightarrow \beta)$

Symboles Directeurs (SD)

Avantage 1: un critère unique pour reconnaître l'ambiguité d'une grammaire.

Avantage 2: si on veut matcher $w[i:]$ (avec symbole de tête b) contre le non-terminal X; il suffit de choisir l'unique production $X \to \alpha$ telle que $b \in \text{SD}(X \to \alpha)$

		b	c	…
X		α	$\emptyset$	…
…		…	…	…

$b \in \text{SD}(X \to \alpha)$

Example

$$S \rightarrow A\ B\ S \mid d$$
$$A \rightarrow B \mid a$$
$$B \rightarrow c \mid \Lambda$$

	NULL	FIRST	FOLLOW
S	no	$\{ a, c, d \}$	$\{ \$ \}$
A	yes	$\{ a, c \}$	$\{ a, c, d \}$
B	yes	$\{ c \}$	$\{ a, c, d \}$

Example: FIRST(S)

$S \rightarrow A\ B\ S \mid d$ ← FIRTS(S) $\supseteq$ FIRST(A) $\cup \{ d \}$

$A \rightarrow B \mid a$

$B \rightarrow c \mid \Lambda$

	NULL	FIRST	FOLLOW
S	no	$\{ a, c, d \}$	$\{ \$ \}$
A	yes	$\{ a, c \}$	$\{ a, c, d \}$
B	yes	$\{ c \}$	$\{ a, c, d \}$

Example

$S \rightarrow A\ B\ S \mid d$ ⟵ FIRTS(S) $\supseteq$ FIRST$(A) \cup \{\ d\ \}$

$A \rightarrow B \mid a$ ⟵ FIRTS(S) $\supseteq$ FIRST$(B) \cup \{\ a, d\ \}$

$B \rightarrow c \mid \Lambda$

	NULL	FIRST	FOLLOW
S	no	$\{\ a, c, d\ \}$	$\{\ \$\ \}$
A	yes	$\{\ a, c\ \}$	$\{\ a, c, d\ \}$
B	yes	$\{\ c\ \}$	$\{\ a, c, d\ \}$

Example

$S \rightarrow A\ B\ S \mid d$ ⟵ FIRTS(S) $\supseteq$ FIRST$(A) \cup \{ d \}$

$A \rightarrow B \mid a$ ⟵ FIRTS(S) $\supseteq$ FIRST$(B) \cup \{ a, d \}$

$B \rightarrow c \mid \Lambda$ ⟵ FIRTS(S) $\supseteq$ FOLLOW$(B) \cup \{ a, d, c \}$

	NULL	FIRST	FOLLOW
S	no	$\{ a, c, d \}$	$\{ \$ \}$
A	yes	$\{ a, c \}$	$\{ a, c, d \}$
B	yes	$\{ c \}$	$\{ a, c, d \}$

Example: FOLLOW(A)

$S \rightarrow A\ B\ S \mid d$ ⟵ FOLLOW(A) $\supseteq$ FIRST(B)

$A \rightarrow B \mid a$ NULL(B) $\Rightarrow$ FOLLOW(A) $\supseteq$ FIRST(S)

$B \rightarrow c \mid \Lambda$ $\neg$ NULL(S) $\Rightarrow$ \$ $\notin$ FOLLOW(A)

	NULL	FIRST	FOLLOW
S	no	$\{a, c, d\}$	$\{\$\}$
A	yes	$\{a, c\}$	$\{a, c, d\}$
B	yes	$\{c\}$	$\{a, c, d\}$

Example

$S \rightarrow A\ B\ S \mid d$

$A \rightarrow B \mid a$

$B \rightarrow c \mid \Lambda$

SD($S \rightarrow A\ B\ S$) ∩
SD($S \rightarrow d$) $= \{ d \}$

FIRST-FIRST conflict

	NULL	FIRST	FOLLOW
S	no	$\{ a, c, d \}$	$\{ \$ \}$
A	yes	$\{ a, c \}$	$\{ a, c, d \}$
B	yes	$\{ c \}$	$\{ a, c, d \}$

SD($B \rightarrow c$) ∩
SD($B \rightarrow \Lambda$) $= \{ c \}$

FIRST-FOLLOW conflict

Example

$S \rightarrow i\ E\ t\ S\ e\ S \mid c$

$E \rightarrow b$

	NULL	FIRST	FOLLOW
S	no	$\{ i, c \}$	$\{ e, \$ \}$
E	no	$\{ b \}$	$\{ t \}$

Example

$S' \to \$$

$S \to i\ E\ t\ S\ e\ S \mid c$

$E \to b$

$\mathrm{SD}(S \to i\ E\ \ldots) = \{\ i\ \}$

$\mathrm{SD}(S \to c) = \{\ c\ \}$

$\mathrm{SD}(E \to b) = \{\ b\ \}$

	NULL	FIRST	FOLLOW
S	no	$\{\ i, c\ \}$	$\{\ e, \$\ \}$
E	no	$\{\ b\ \}$	$\{\ t\ \}$

Example

$S' \rightarrow \$$

$S \rightarrow i\ E\ t\ S\ e\ S \mid c$

$E \rightarrow b$

$\mathrm{SD}(S \rightarrow i\ E\ \ldots) = \{\, i \,\}$

$\mathrm{SD}(S \rightarrow c) = \{\, c \,\}$

$\mathrm{SD}(E \rightarrow b) = \{\, b \,\}$

$w = i\ b\ t\ c\ e\ c\ \$$

$\gamma_0 = S\ \$$

	i	t	e	c	b
S	$i\ E\ t\ S\ e\ S$			c	
E					b

Example

$$X \rightarrow Y\ c \mid a$$

$$Y \rightarrow b\ Z \mid \Lambda$$

$$Z \rightarrow \Lambda$$

	NULL	FIRST	FOLLOW
X			
Y			
Z			

Example

$X \rightarrow Y\ c \mid a$

$Y \rightarrow b\ Z \mid \Lambda$

$Z \rightarrow \Lambda$

$\text{SD}(X \rightarrow Y\ c) = \{\ b, c\ \}$

$\text{SD}(X \rightarrow a) = \{\ a\ \}$

$\text{SD}(Y \rightarrow b\ Z) = \{\ b\ \}$

$\text{SD}(Y \rightarrow \Lambda) = \{\ c\ \}$

$\text{SD}(Z \rightarrow \Lambda) = \{\ c\ \}$

	NULL	FIRST	FOLLOW
X	no	$\{\ a, b\ c\ \}$	$\{\ \$\ \}$
Y	yes	$\{\ b\ \}$	$\{\ c\ \}$
Z	yes	$\emptyset$	$\{\ c\ \}$

Example

$$X \to Y\ c \mid a$$
$$Y \to b\ Z \mid \Lambda$$
$$Z \to \Lambda$$

$$\text{SD}(X \to Y\ c) = \{\ b, c\ \}$$
$$\text{SD}(X \to a) = \{\ a\ \}$$
$$\text{SD}(Y \to b\ Z) = \{\ b\ \}$$
$$\text{SD}(Y \to \Lambda) = \{\ c\ \}$$
$$\text{SD}(Z \to \Lambda) = \{\ c\ \}$$

	a	b	c
X	a	$Y\ c$	$Y\ c$
Y		$b\ Z$	Λ
Z			Λ

Eliminating conflicts

It is not always possible to eliminate ambiguities in a grammar (hint: undecidability!)

But we can always try to use substitution; elimination and left-recursion elimination

Example:

$$S \rightarrow A\,S \mid b$$
$$A \rightarrow A\,a \mid b$$

RecElim(A) + SUBST(A) + FACT + SUBST

Another example

$S' \to S\ \$$

$S \to A \mid B \mid \Lambda$ $\quad (A \vee B \vee \{\ \epsilon\ \})$

$A \to a\ A\ b \mid \Lambda$ $\quad (a^n b^n)$

$B \to b\ B\ a \mid \Lambda$ $\quad (b^n a^n)$

Exercise: show that this grammar is LL(1)

Another example

$S' \to S\ \$$

$S \to A \mid B \qquad (A \vee B)$

$A \to a\ A\ b \mid 0 \qquad (a^n 0 b^n)$

$B \to a\ B\ b\ b \mid 1 \qquad (a^n 1 b^{2n})$

Exercise: can you think of a reason why this grammar is not LL(k) ; can you think of a program to test if a word is accepted by this grammar

Yet Another Example

$$E \rightarrow E + T \mid E - T \mid T$$

$$T \rightarrow T * F \mid T / F \mid F$$

$$F \rightarrow id \mid (E)$$

Exercise: eliminate the left-recursion (on E and T) and show that the resulting grammar is LL(1). Write a recursive descent parser for this simple "expression languages" using your programming language of choice.

Yet Another Example

$E \rightarrow E + T \mid E - T \mid T$

$T \rightarrow T * F \mid T \, / \, F \mid F$

$F \rightarrow id \mid (\, E \,)$

Exercise: give the derivations for the words,

$id \; * \, (\, id + id \,) \, \$$

$id \; id \; \$$

$id \;) \; \$$

Pushdown Automata

automates à piles [FR]

Pushdown automata (PDA)

- A PDA is a finite state automata that can use a *stack* to keep a list of symbols
- We extends FSA with:
  - an alphabet for symbols in the stack (Γ)
  - an initial stack symbol $Z \in \Gamma$
- We extend the transition function, δ, so that we can *read, test* (pop) and *write* (push) to the stack

$$\delta(q, a, S) \to (q', \beta)$$

meaning $(q, a\, w, S\, \gamma) \Rightarrow (q', w, \beta\, \gamma)$

Pushdown automata (PDA)

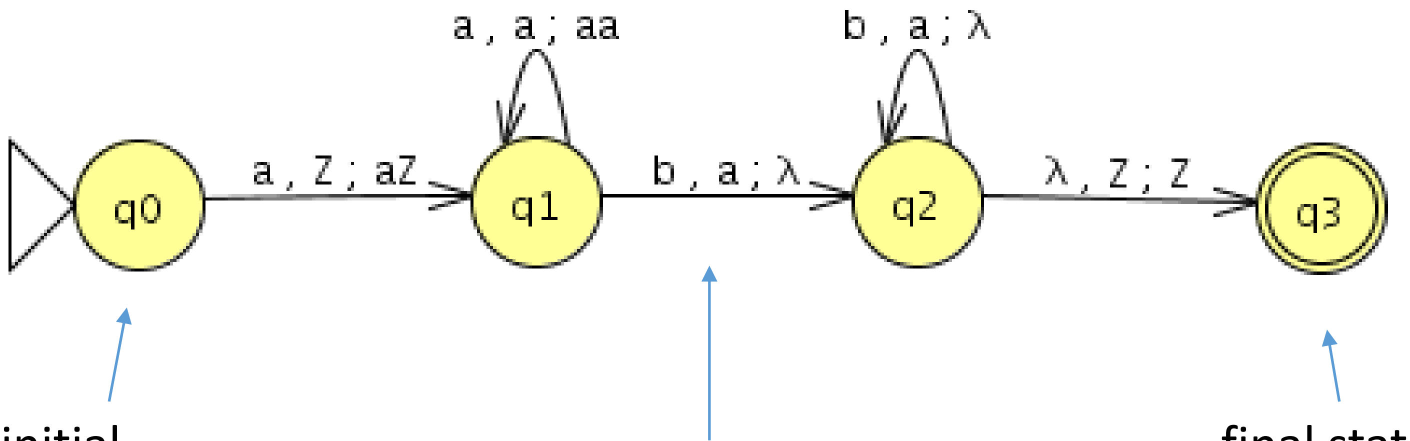
We extend the transition function, δ, so that we can *read, test* and *write* to the stack

$\delta(q, a, S) = (q', \beta)$ means than, in state q, with symbol $S \in \Gamma$ at top of the stack, when reading symbol $a \in \Sigma \cup \{\epsilon\}$, we transition to state q', pop S and replace the top of the stack with $\beta \in \Gamma^\star$.

This defines a transition relation of the form:

$$(q, w, \alpha) \Rightarrow^\star (q', w', \beta)$$

PDA: graphical representation



initial state

$\delta(q_1, b, a) \to (q_2, \Lambda)$

final states

We use label a , S ; β to represent transition $\delta(q, a, S) \to (q', \beta)$. Other notation: a , S / β

Figure obtained using JFLAP

Pushdown automata (PDA)

There are four classes of *configurations*:

1. $(q, a\,w, S\,\gamma) \Rightarrow (q', w, \beta\,\gamma)$ shift + reduce
   $\delta(q, a, S) \to (q', \beta)$
2. $(q, w, S\,\gamma) \Rightarrow (q', w, \beta\,\gamma)$ $(a = \epsilon)$ reduce
   $\delta(q, \epsilon, S) \to (q', \beta)$
3. $(q, a\,w, S\,\gamma) \Rightarrow (q', w,\ S\,\gamma)$ $(\beta = S)$ shift
   $\delta(q, a, S) \to (q', S)$
4. $(q, w, S\,\gamma) \Rightarrow (q', w, \gamma)$ pop
   $\delta(q, \epsilon, S) \to (q', \Lambda)$

Pushdown automata (PDA)

We can choose among many different (but equivalent) accepting conditions:

- $(q_I, w, Z) \Rightarrow^\star \ (q_f, \epsilon, \ \beta)$ with $q_f \in F$

  end in final state (arbitrary stack)
- $(q_I, w, Z) \Rightarrow^\star \ (q', \epsilon, \epsilon)$

  end with empty stack (arbitrary state)
- $(q_I, w, Z) \Rightarrow^\star \ (q_f, \epsilon, \ \epsilon)$ with $q_f \in F$

  end with final state + empty stack

in each case we must entirely read the input word, w

Pushdown automata: example

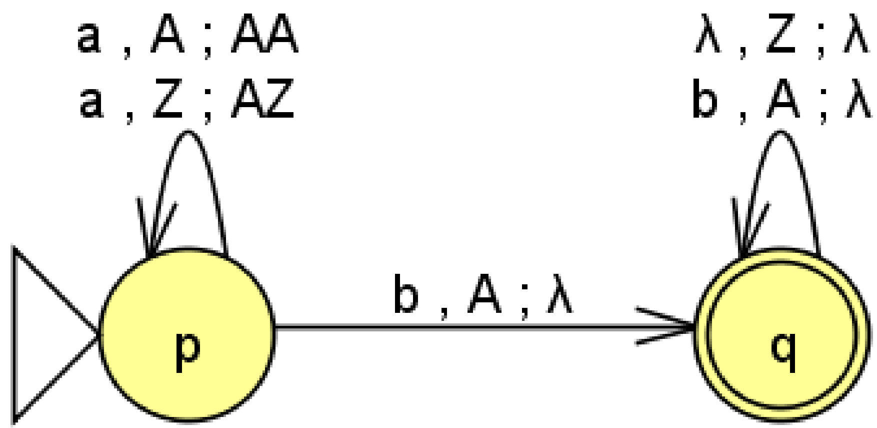
$$\delta(p, a, \mathrm{Z}) \rightarrow (p, AZ)$$

$$\delta(p, a, \mathrm{A}) \rightarrow (p, AA)$$

$$\delta(p, b, A) \rightarrow (q, \Lambda)$$

$$\delta(q, b, A) \rightarrow (q, \Lambda)$$

$$\delta(q, \epsilon, Z) \rightarrow (q, \Lambda)$$



here we assume acceptance with empty stack

Pushdown automata: example

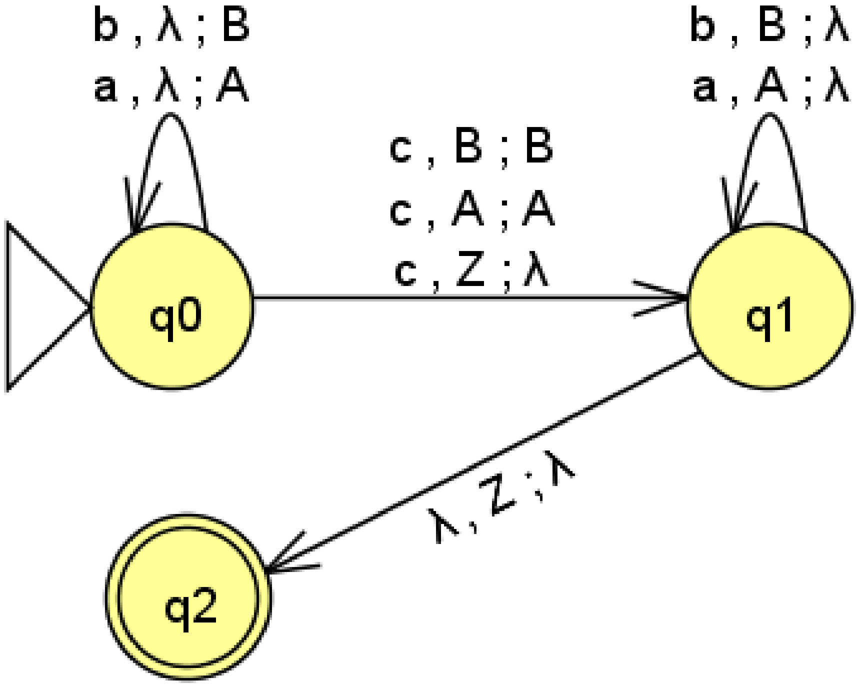
$\delta(q_0, a, \Lambda) \rightarrow (q_0, A)$

$\delta(q_0, c, \mathrm{A}) \rightarrow (q_1, A)$

$\delta(q_1, a, A) \rightarrow (q_1, \Lambda)$

$\delta(q_1, \epsilon, \mathrm{Z}) \rightarrow (q_2, \Lambda)$

accepts: $a\ b\ b\ c\ b\ b\ a$



Notation: $\widetilde{w}$ is the mirror image of w

Here we assume acceptance with empty stack

Equivalence PDA $\leftrightarrow$ AG

A language $\mathcal{L}$ is *algebraic* iff there is a PDA $\mathcal{A}$ such that $\mathcal{L} = \mathcal{L}(\mathcal{A})$. Call ALG this class of languages.

We have REG $\subseteq$ ALG (easy)

We have CFG $\subseteq$ ALG (build a PDA from a grammar)

We have ALG closed by $\cup$, $\star$ and $\cdot$ ($\approx$ automata)

But ALG is not closed by $\cap$ and complement; while it is closed by $\cap$ with regular languages.

CFL ⊆ ALG

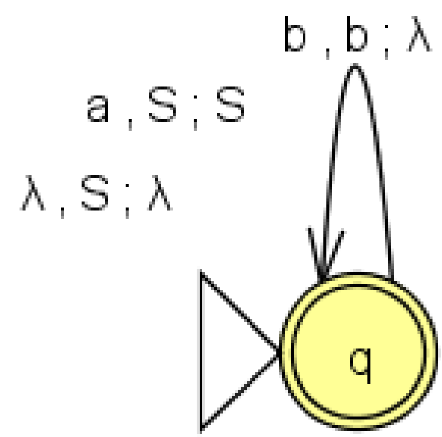
Take a grammar with production $X \rightarrow \alpha$ and axiom S

(We can always assume $\alpha = b\,\gamma$ or $\alpha = \epsilon\,\gamma$)

We can build a (non-deterministic) PDA, with stack symbol S and a single state, q, that accepts (empty stack) the same language very easily

Just take: $\delta(q, b, X) \rightarrow (q, \alpha)$

$$S \rightarrow \epsilon\,\Lambda$$
$$S \rightarrow a\,S$$
$$S \rightarrow b\,\Lambda$$



Complexity of CFL problems

Many problems are undecidable for CFL

- Universality
- Language inclusion, equality
- Given a CFL, is there an equivalent Type-3 grammar

On the other hand, checking emptiness ($\mathcal{L} =^? \emptyset$) is decidable for CFL, whereas it is not the case with more complex models (e.g. context-sensitive languages)

Deterministic PDA

Like with DFA, we can very much accept to have many transitions for the same “input” (q, a, S); meaning that δ is a function in $Q \times \Sigma^{\perp} \times \Gamma \rightarrow Q \times 2^{\Gamma^{\star}}$

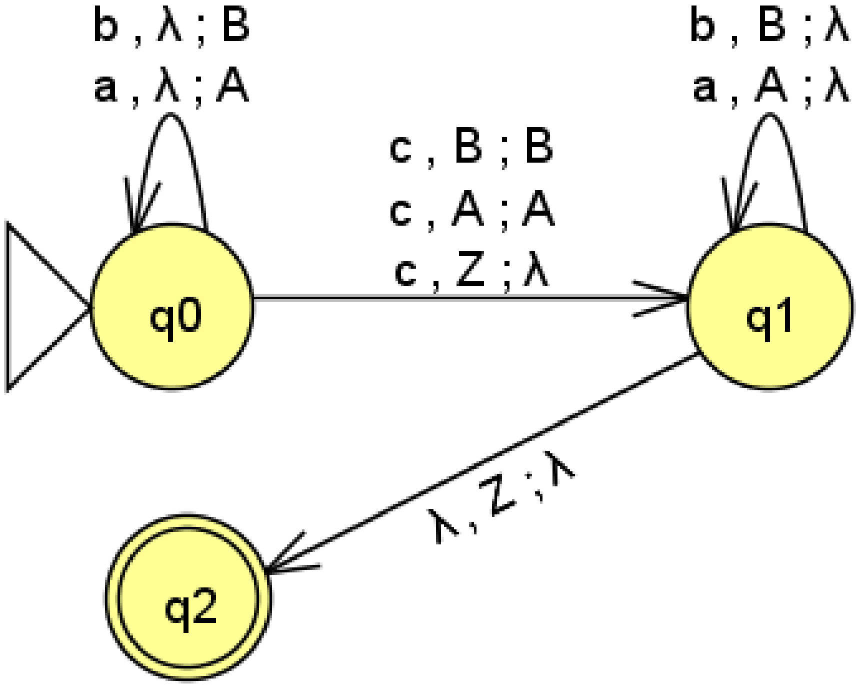
A PDA is *deterministic* when, for every $q \in Q, a \in \Sigma^{\perp}, S \in \Gamma$ we have:

1. $|\delta(q, a, S)| = 1$
2. if $\delta(q, \epsilon, S) \neq \emptyset$ then $\delta(q, b, S) = \emptyset$ for all $b \in \Sigma$

DCFL languages are “accepted” by DPDA

DPDA

Our previous example is a Deterministic-PDA



Limitations of DPDA

DCFL is an interesting class; in particular it includes LL(1) grammars.

There are some CFL which are not DCFL

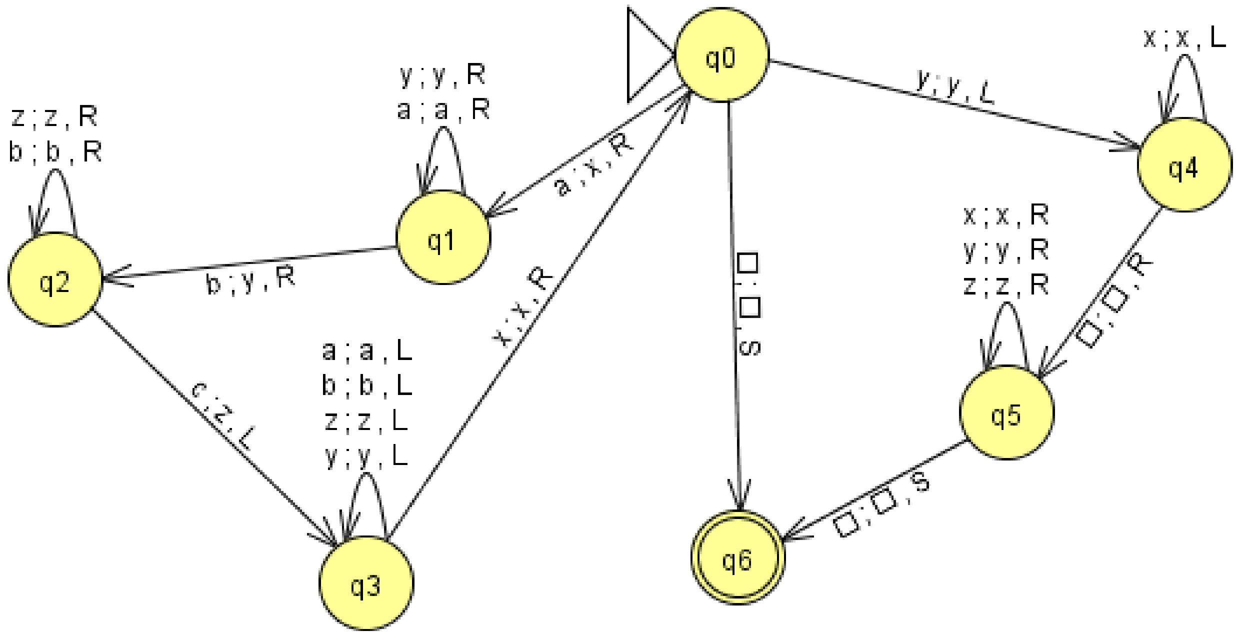
$\Rightarrow$ DCFL $\neq$ CFL

Idea: take words of the form $w\ \widetilde{w}$ (compare that with the words $w\ c\ \widetilde{w}$)

Also: DCFL are not closed by $\cup$ (idea?), but they are closed by complement (hard!).

More General Computation Models

It is easy to extend PDA with an ∞ *tape,* prefilled with *blank symbols* □, and with special actions (LEFT, RIGHT and STAY moves) ≡ Turing Machines



TM for $a^n b^n c^n$

More General Computation Models

It is easy to extend PDA so that they can use n (≥ 2) stacks

0-PDA are automata

2-PDA stacks are more powerful than 1-PDA … and actually are universal

0-PDA ⊂ 1-PDA ⊂ 2-PDA $=$ n-PDA = TM

Post Correspondence Problem

It may be hard to believe that problems become (that much) complex with the introduction of a stack

Problem: you are given two lists (equal length) of words $u_1, \ldots, u_n$ and $w_1, \ldots, w_n$. Decide whether there is a sequence of indices $i_1, \ldots, i_k$ in $1..n$ such that:

$$u_{i_1} \ldots \ u_{i_k} = w_{i_1} \ldots \ w_{i_k}$$

PCP is undecidable

This is "almost" like dominoes: $\frac{u_1}{w_1}$ $\frac{u_2}{w_2}$ $\frac{u_n}{w_n}$

$$\frac{10111}{10}\ \frac{1}{111}\ \frac{1}{111}\ \frac{10}{0} = \frac{10111.1.1.10}{10.111.111.0}$$

LR grammars

LR parsers

LR grammars are those than can be parsed using a LR parser ⇒ parses by scanning the input from **Left** to right and building a **Rightmost** derivation (in reverse)

rightmost ⇒ replaces the right-hand side of production rules (the α in $X \rightarrow \alpha$) with their left-hand side

We also use PDA and tables, but they are different.

LR parsers are nice

- LR parsers can handle a large class of CFG; and more languages than LL grammars
- LR parser can detect syntax errors “as soon as they occur”
- LR parsing is the most general, non-back tracking, shift-reduce parsing method

LR parser have drawbacks

- It may be complex to build an unambiguous version of a grammar
- Once you have a suitable grammar, it is too complex to build a parser by hand $\Rightarrow$ need a tool to generate it

LR parser: example

We build a table with 4 kinds of actions

- [SHIFT n] transfer look-ahead to the stack and move to state n
- [REDUCE k] replace α with X on the stack using rule number k
- [ACCEPT] terminate and answer OK
- [ERROR] terminate and answer KO

LR parser: example

Take the grammar $S \rightarrow a\ S\ b \mid b$

The LR(1) table obtained from this grammar is

	a	b	$\$$	S
0	$s2$	$s3$		1
1			OK	
2	$s2$	$s3$		4
3		$r2$	$r2$	
4		$s5$		
5		$r1$	$r1$	

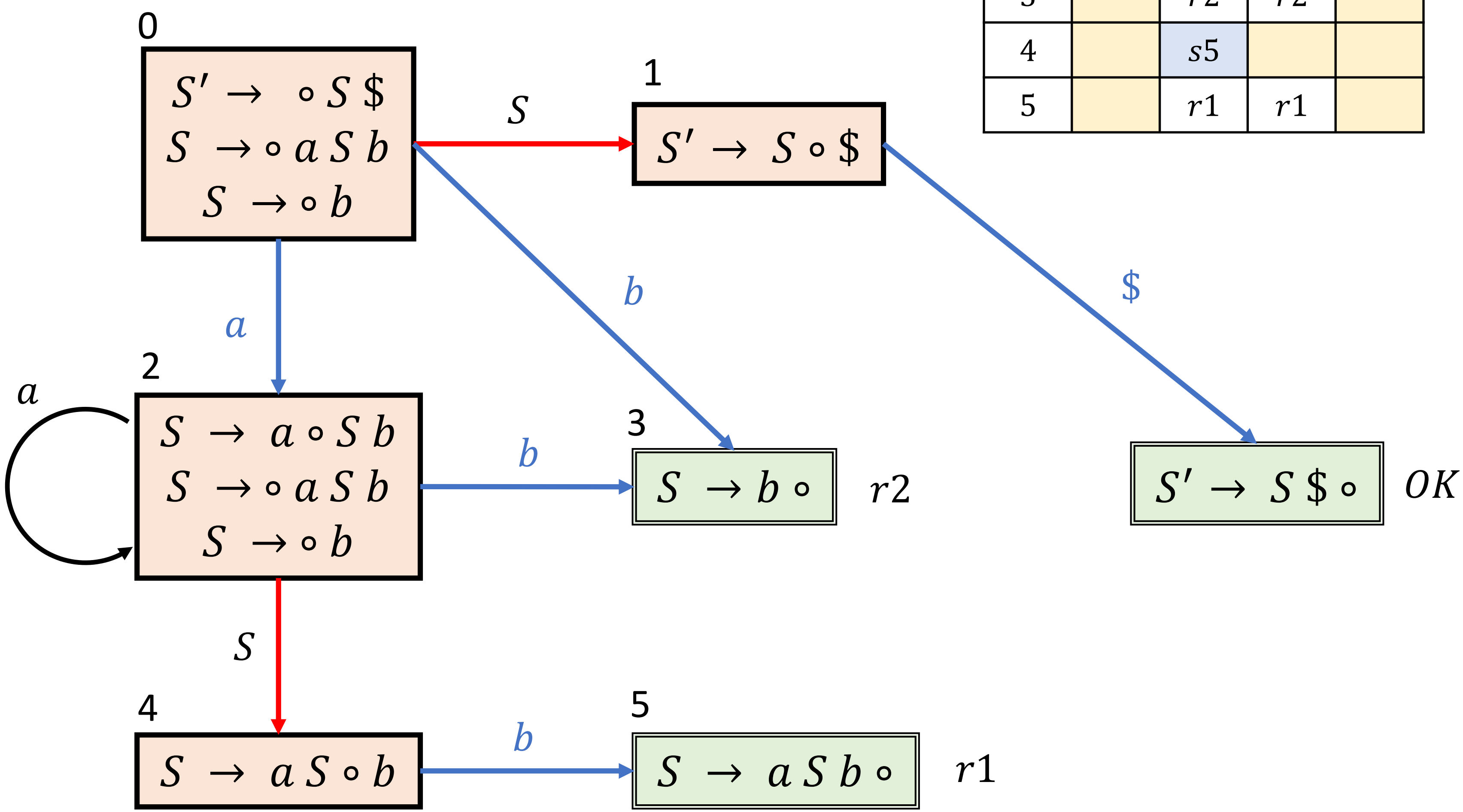
Header row ← $\Gamma \cup \{ \$ \}$

derivations # → (row numbers)

Row 2, column S (4) ← move to row 4

LR parser: table

	a	b	$\$$	S
0	$s2$	$s3$		1
1			OK	
2	$s2$	$s3$		4
3		$r2$	$r2$	
4		$s5$		
5		$r1$	$r1$	



LR parser: $a\ a\ b\ b\ b$

	a	b	$\$$	S
0	$s2$	$s3$		1
1			OK	
2	$s2$	$s3$		4
3		$r2$	$r2$	
4		$s5$		
5		$r1$	$r1$	

init $(a_0, 0, \$_0)$

We are in position 0 of the word, with look-ahead a

We start in state (row) 0; the stack contains state 0 and symbol $

The stack is a sequence of pairs (state i) × symbol, which we write symbol $_i$

LR parser: $a\ a\ b\ b\ b$

	a	b	$\$$	S
0	$s2$	$s3$		1
1			OK	
2	$s2$	$s3$		4
3		$r2$	$r2$	
4		$s5$		
5		$r1$	$r1$	

init $(a_0, 0, \$_0)$

T[0, a] = $s2$, the first action is a shift to state 2

- the new state is 2
- we push the symbol and state, a_2, in the stack
- we read the next symbol

LR parser: $a\ a\ b\ b\ b$

	a	b	$\$$	S
0	$s2$	$s3$		1
1			OK	
2	$s2$	$s3$		4
3		$r2$	$r2$	
4		$s5$		
5		$r1$	$r1$	

init $(a_0, 0, \$_0)$

$s2$ $\rightarrow (a_1, 2, a_2\ \$_0)$

LR parser: $a\ a\ b\ b\ b$

	a	b	$\$$	S
0	$s2$	$s3$		1
1			OK	
2	$s2$	$s3$		4
3		$r2$	$r2$	
4		$s5$		
5		$r1$	$r1$	

init $(a_0, 0, \$_0)$

$s2 \rightarrow (a_1, 2, a_2\ \$_0)$

$s2 \rightarrow (b_2, 2, a_2\ a_2\ \$_0)$

$s3 \rightarrow (b_3, 3, b_3\ a_2\ a_2\ \$_0)$

T[3, b] = $r2$, the next action is a shift for rule 2, $S \rightarrow b$

- we pop b from the stack, it is at state 2
- we push S with state T[2, S] $= 4$
- and move to state 4

LR parser: $a\ a\ b\ b\ b$

	a	b	$\$$	S
0	$s2$	$s3$		1
1			OK	
2	$s2$	$s3$		4
3		$r2$	$r2$	
4		$s5$		
5		$r1$	$r1$	

init $(a_0, 0, \$_0)$

$s2 \quad \rightarrow (a_1, 2, a_2\ \$_0)$

$s2 \quad \rightarrow (b_2, 2, a_2\ a_2\ \$_0)$

$s3 \quad \rightarrow (b_3, 3, b_3\ a_2\ a_2\ \$_0)$

$r2 \quad \rightarrow (b_3, 4, S_4\ a_2\ a_2\ \$_0)$

LR parser: $a\ a\ b\ b\ b$

	a	b	$	S
0	$s2$	$s3$		1
1			OK	
2	$s2$	$s3$		4
3		$r2$	$r2$	
4		$s5$		
5		$r1$	$r1$	

init $(a_0, 0, \$_0)$

$s2 \quad \rightarrow (a_1, 2, a_2\ \$_0)$

$s2 \quad \rightarrow (b_2, 2, a_2\ a_2\ \$_0)$

$s3 \quad \rightarrow (b_3, 3, b_3\ a_2\ a_2\ \$_0)$

$r2 \quad \rightarrow (b_3, 4, S_4\ a_2\ a_2\ \$_0) \qquad S \rightarrow b$

$s5 \quad \rightarrow (b_4, 5, b_5\ S_4\ a_2\ a_2\ \$_0)$

$r1 \quad \rightarrow (b_4, 4, S_4\ a_2\ \$_0) \qquad S \rightarrow a\ S\ b$

LR parser: $a\ a\ b\ b\ b$

	a	b	$\$$	S
0	$s2$	$s3$		1
1			OK	
2	$s2$	$s3$		4
3		$r2$	$r2$	
4		$s5$		
5		$r1$	$r1$	

init $(a_0, 0, \$_0)$

$s2 \rightarrow (a_1, 2, a_2\ \$_0)$

$s2 \rightarrow (b_2, 2, a_2\ a_2\ \$_0)$

$s3 \rightarrow (b_3, 3, b_3\ a_2\ a_2\ \$_0)$

$r2 \rightarrow (b_3, 4, S_4\ a_2\ a_2\ \$_0)$ $S \rightarrow b$

$s5 \rightarrow (b_4, 5, b_5\ S_4\ a_2\ a_2\ \$_0)$

$r1 \rightarrow (b_4, 4, S_4\ a_2\ \$_0)$ $S \rightarrow a\ S\ b$

$s5 \rightarrow (\$_5, 5, b_5\ S_4\ a_2\ \$_0)$

$r1 \rightarrow (\$_5, 1, S_1\ \$_0)$ $S \rightarrow a\ S\ b$

OK

LR parser: $a\ a\ b\ b\ b$

	a	b	$\$$	S
0	$s2$	$s3$		1
1			OK	
2	$s2$	$s3$		4
3		$r2$	$r2$	
4		$s5$		
5		$r1$	$r1$	

init $(a_0, \$)$

$s2$ $\rightarrow (a_1, a\ \$)$

$s2$ $\rightarrow (b_2, a\ a\ \$)$

$s3$ $\rightarrow (b_3, b\ a\ a\ \$)$

$r2$ $\rightarrow (b_3, S\ a\ a\ \$)$ because $S \rightarrow b$

$s5$ $\rightarrow (b_4, b\ S\ a\ a\ \$)$

$r1$ $\rightarrow (b_4, S\ a\ \$)$ because $S \rightarrow a\ S\ b$

$s5$ $\rightarrow (\$_5, b\ S\ a\ \$)$

$r1$ $\rightarrow (\$_5, S\ \$)$ because $S \rightarrow a\ S\ b$

OK

LR parser

We have not discussed:

- how to check whether the grammar is LR (finding conflicts between rules)
- what are the possible kind of conflicts
- how to solve conflicts (when possible)