Automata & Complexity

Jörg Endrullis

Vrije Universiteit Amsterdam

2018

Looking Back

Last Lecture:

Not all languages are regular. For example

$$L = \{ a^n b^n \mid n \geq 0 \}$$

Proof: use the **pumping lemma**!

Context-Free Languages

A grammar $G = (V, T, S, P)$ is called **context-free** if all production rules are of the form

$$A \to u$$

where $A \in V$ and $u \in (V \cup T)^*$.

That is, the left-hand side of all rules is a single variable.

A language L is **context-free** if there is a context-free grammar with $L = L(G)$.

Context-free grammars are used to

- describe the **syntax of programming languages**, and
- form the basis of **parsing algorithms**.

Exercises

Show that the following language is context-free:

$$\{ a^n b^m c^{2n} \mid n \geq 0,\ m \geq 0 \}$$

Show that the following language is context-free:

$$\{ x \in \Sigma^+ \mid \forall w \in \Sigma^+ .\ x \neq ww \}$$

where $\Sigma = \{a, b\}$.

Is the following language context-free?

$$\{ ww \mid w \in \Sigma^+ \}$$

where $\Sigma = \{a, b\}$.

Rightmost and Leftmost

Let G be a grammar, and consider a derivation $S \Rightarrow^* w$.

- If G is **(right) linear**, w contains **at most one variable**.
- If G is **context-free**, w can contain **multiple variables**.

Two strategies to choose which variable to expand:

- **leftmost**: always the leftmost variable
- **rightmost**: always the rightmost variable

$$S \to SaS \mid b$$

Two derivations of bab:

Leftmost: $S \Rightarrow SaS \Rightarrow baS \Rightarrow bab$

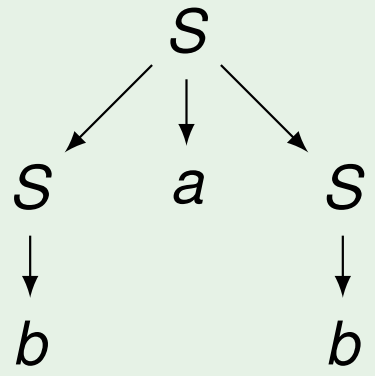
Rightmost: $S \Rightarrow SaS \Rightarrow Sab \Rightarrow bab$

Result depends not on the strategy, but the choice of the rules.

Derivation Trees

$$S \rightarrow SaS \mid b$$

A **derivation tree** for *bab* is:



Derivation Trees

A **derivation tree** for a context-free grammar $G = (V, T, S, P)$ has nodes with labels from $V \cup T \cup \{\lambda\}$.

- The **root** has the label S.
- If there is a production rule
  - $A \to x_1 \cdots x_n$ with $n \geq 1$, then
    a node labelled A can have children labelled $x_1, \ldots, x_n$.
  - $A \to \lambda$, then
    a node labelled A can have one child with label λ.
- Every node with a label from $T \cup \{\lambda\}$ is a **leave**.
  (That is, the node has no children).

The labels of the leaves of a derivation tree, read from left to right (skipping λ), form a word in $L(G)$.

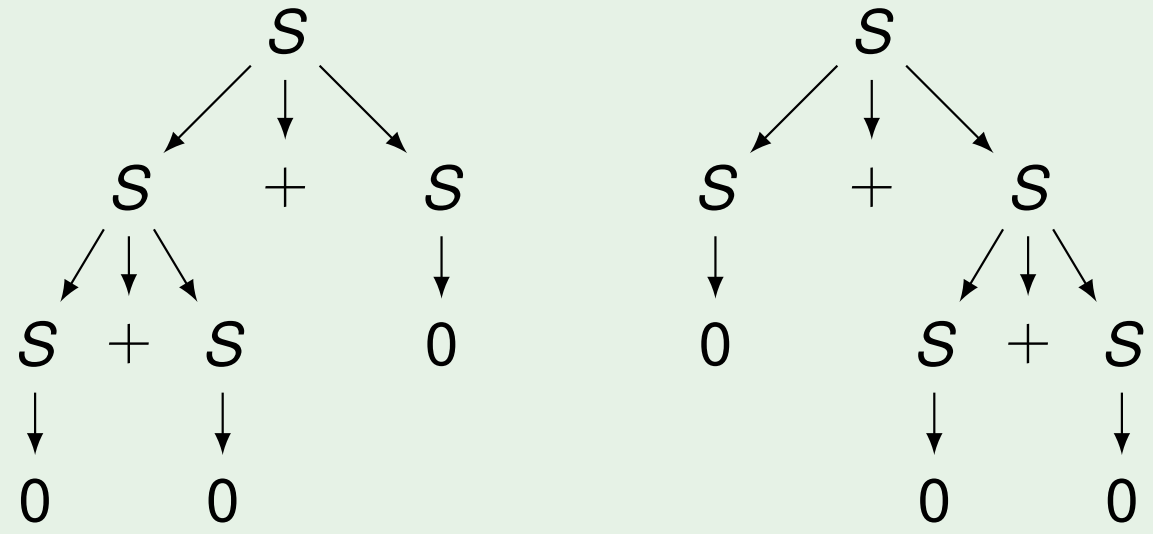
Ambiguous Grammars

A context-free grammar G is **ambiguous** if there exists a word $w \in L(G)$ for which there are multiple derivation trees.

Is the following grammar ambiguous?

$$S \to S + S \mid 0$$

Yes, there are two derivations trees for $0 + 0 + 0$:



Ambiguous Grammars

Is the following grammar ambiguous?

$$S \to S + 0 \mid 0$$

No, every word has precisely one derivation tree.

Note that both grammars

$$S \to S + S \mid 0 \qquad\qquad S \to S + 0 \mid 0$$

generate the same language:

$$\{\, 0(+0)^n \mid n \geq 0 \,\}$$

Ambiguity in Programming Languages

The following production rules (from ALGOL 60) became known as **dangling else problem**:

$$
\begin{array}{lcl}
\textit{Statement} & \rightarrow & \texttt{if}\ \textit{Condition}\ \texttt{then}\ \textit{Statement} \quad | \\
 & & \texttt{if}\ \textit{Condition}\ \texttt{then}\ \textit{Statement}\ \texttt{else}\ \textit{Statement} \quad | \\
 & & \ldots \\
\textit{Condition} & \rightarrow & \ldots
\end{array}
$$

There are two derivation trees for

```
if ... then if ... then ... else ...
```

The production rules are ambiguous.

Donald Knuth, *The Remaining Trouble Spots in ALGOL 60*, CACM, 1967

Ambiguity Typically Unwanted

Ambiguity is typically unwanted:

- derivation trees often used to assign meaning to words,
- multiple derivation tree may result in double meaning.

In practice, ambiguity is often resolved outside of the grammar.

For example, by a precedence on the rules:

- For example, $0 + 2 * 1$ is parsed as $0 + (2 * 1)$.

Ambiguity is undecidable. That is, there exists no algorithm that decides whether a context-free grammar is ambiguous.

Inherently Ambiguous Languages

A language L is **inherently ambiguous** if

- L is context-free, and
- **every** grammar G with $L(G) = L$ is ambiguous.

The following context-free language is inherently ambiguous

$$\{ a^n b^m c^\ell \mid n = m \vee m = \ell \}$$

It is generated by the following (ambiguous) grammar

$$S \to A \mid B \qquad A \to Ac \mid C \qquad B \to aB \mid D$$
$$C \to aCb \mid \lambda \qquad D \to bDc \mid \lambda$$

(Groups of two, 2 minutes)

Exercise

Find two derivation trees for the word abc.

Parsing

Parsing is the search for a derivation tree for a given word.

Theorem

For context-free grammars, parsing is possible in $O(|w|^3)$ time.
(Here $|w|$ is the length of the input word.)

For every context-free grammar G,
there exists an algorithm and $C \geq 0$ such that for every

- word w,

the algorithm determines in at most $C \cdot |w|^3$ steps

- whether $w \in L(G)$ (including a derivation tree if $w \in L(G)$).

Removal of λ-Rules

A production rule $A \to \lambda$ is called **λ-production rule**.

Theorem

For every context-free language L there exists a context-free grammar G without λ-rules such that $L(G) = L \setminus \{\lambda\}$.

Construction

Let G be a context-free grammar with $L(G) = L$.

Determine all variables A in G with $A \Rightarrow^* \lambda$:

- If $A \to \lambda$, then $A \Rightarrow^* \lambda$.
- If $A \to B_1 \cdots B_n$ and $B_1 \Rightarrow^* \lambda, \ldots, B_n \Rightarrow^* \lambda$, then $A \Rightarrow^* \lambda$.

For every rule $A \to xBy$ with $B \Rightarrow^* \lambda$, add a rule $A \to xy$.

Remove all λ-production rules.

The resulting grammar G has the property $L(G) = L \setminus \{\lambda\}$.

Exercise

Consider the following grammar

$$S \to ABaC \qquad A \to BC \qquad B \to b \mid \lambda \qquad D \to d$$
$$C \to D \mid \lambda$$

Which variables can produce λ?

- A, B and C

Construct the resulting grammar after removing all λ-rules:

$$S \to ABaC \mid BaC \mid AaC \mid ABa \mid aC \mid Ba \mid Aa \mid a$$
$$A \to BC \mid C \mid B$$
$$B \to b \qquad C \to D \qquad D \to d$$

Removal of Unit Production Rules

A rule $A \to B$ is called **unit production rule** (here $B \in V$).

Theorem

For every context-free language L there is a context-free grammar G without λ- and unit-productions with $L(G) = L \setminus \{\lambda\}$.

Construction

Let G be context-free, without λ-rules, and $L(G) = L \setminus \{\lambda\}$.

Determine all pairs $A \neq B$ with $A \Rightarrow^+ B$.

Whenever $A \Rightarrow^+ B$ and $B \to y$ is a rule, add a rule $A \to y$.

Remove all unit production rules.

The resulting grammar G has no λ- and unit-productions and it has the property $L(G) = L \setminus \{\lambda\}$.

Exercise

Remove all unit production rules from

$$S \to Aa \mid B \qquad A \to a \mid bc \mid B \qquad B \to A \mid bb$$

Note that there are no λ-productions.
(So no need to first remove λ-productions.)

We determine all pairs $A \neq B$ with $A \Rightarrow^+ B$:

$$S \Rightarrow^+ B \qquad A \Rightarrow^+ B \qquad B \Rightarrow^+ A \qquad S \Rightarrow^+ A$$

Thus we add the following rules:

$$S \to Aa \mid B \mid a \mid bc \mid A \mid bb$$
$$A \to a \mid bc \mid B \mid A \mid bb$$
$$B \to A \mid bb \mid a \mid bc \mid B$$

Removing all unit production rules yields the final result:

$$S \to a \mid bb \mid bc \mid Aa \qquad A \to a \mid bb \mid bc \qquad B \to a \mid bb \mid bc$$

Chomsky Normal Form

In a grammar in **Chomsky normal form** all rules have the form

$$A \to BC \quad \text{or} \quad A \to a$$

Note that a grammar in Chomsky normal form contains

- no λ-production rules,
- no unit production rules.

Theorem

For every context-free language L there is a grammar G in Chomsky normal form with $L(G) = L \setminus \{\lambda\}$.

Chomsky Normal Form

Construction

Let G be a context-free grammar without λ- and unit-productions and $L(G) = L \setminus \{\lambda\}$.

Introduce variables C_a and rules $C_a \to a$ for every $a \in T$.

Replace every rule $A \to x_1 \cdots x_n$ $(x_i \in V \cup T)$ with $n \geq 2$ by

$$A \to B_1 \cdots B_n \qquad \text{where} \qquad B_i = \begin{cases} x_i, & \text{if } x_i \in V \\ C_{x_i}, & \text{if } x_i \in T \end{cases}$$

Replace every $A \to B_1 \cdots B_n$ with $n \geq 3$ by

$$A \to B_1 \cdots B_{n-2} C \qquad C \to B_{n-1} B_n$$

where C is a fresh variable.

Repeat the last step until all rules are in Chomsky normal form.

Exercise

(Groups of two, 2 minutes)

Transform the following context-free grammar

$$\begin{aligned} S &\to aAbB \\ A &\to SbBa \mid a \\ B &\to AbBb \mid aba \end{aligned}$$

into Chomsky normal form.

Bottom-up Parsing

Bottom-up parsing applies rules backwards, it tries to construct the starting variable S from the input word.

Cocke-Younger-Kasami algoritme (1965)

The **CYK algorithm** is a bottom-up parsing technique for grammars in Chomsky normal form.

Cocke-Younger-Kasami algoritme (1965)

Let G be a grammar in Chomsky normal form.

Goal: determine whether word $a_1 \cdots a_n$ (with $n > 0$) is in $L(G)$.

Idea: compute sets V_{ij} of variables such that

$$V_{ij} = \{ A \in V \mid A \Rightarrow^+ a_i \cdots a_j \} \qquad (1 \leq i \leq j \leq n)$$

as follows:

- $V_{ii} = \{ A \in V \mid A \to a_i \in P \}$ for every $1 \leq i \leq n$.
- V_{ij} with $i < j$ is the set of all $A \in V$ such that

$$A \to BC \in P \qquad \text{with} \qquad B \in V_{ik} \text{ and } C \in V_{k+1j}$$

  for some k with $i \leq k < j$.

Finally, $a_1 \cdots a_n \in L(G) \Leftrightarrow S \in V_{1n}$.

The next slide presents the same algorithm but with better variable names.

Cocke-Younger-Kasami algoritme (1965)

Let G be a grammar in Chomsky normal form.

Goal: determine whether word $w \neq \lambda$ is in $L(G)$.

Idea: compute sets V_u of variables (u subword of w) such that

$$V_u = \{ A \in V \mid A \Rightarrow^+ u \}$$

as follows:

- if $|u| = 1$, then $V_u = \{ A \in V \mid A \to u \in P \}$
- if $|u| > 1$, then V_u is the set of all $A \in V$ such that
  - $u = u_1 u_2$ for some non-empty words u_1, u_2, and
  - $A \to BC \in P$ with $B \in V_{u_1}$ and $C \in V_{u_2}$.

Finally, $w \in L(G) \Leftrightarrow S \in V_w$.

Worst-case time complexity: $O(n^3)$
(There are $n(n+1)/2$ sets V_u, and computation of V_u is $O(n)$.)

Exercise

(Groups of two, 2 minutes)

Use the CYK algorithm to check whether *abbb* is generated by

$$S \to AB$$
$$A \to BB \mid a$$
$$B \to AB \mid b$$

Removal of Useless Variables

A variable A is **useless** for a context-free grammar if there exists no derivation of the form

$$S \Rightarrow^* uAv \Rightarrow^+ w \qquad \text{with } w \in T^*.$$

Removing production rules that contain a useless variable from a grammar does not change the generated language.

$$S \to aSb \mid BC \mid \lambda \qquad A \to Sb \qquad B \to a \qquad C \to C$$

Which variables are useless?

- A because there is no derivation $S \Rightarrow^* uAv$
- C because there is no derivation $C \Rightarrow^* w$ with $w \in T^*$
- B because B can be reached only together with C

The resulting grammar is $S \to aSb \mid \lambda$.

Removal of Useless Variables

Question

How to determine useless variables of a context-free grammar?

Construction

A variable A is called **productive** if $A \Rightarrow^+ w$ with $w \in T^*$.

We determine all productive variables:

- If $A \to y$ is a rule and all variables in y are productive, then A is productive.

Remove all rules that contain a non-productive variable.

We determine all **reachable** variables as follows:

- S is reachable.
- If $A \to y$ and A is reachable, then so are all variables in y.

Remove all rules that contain a non-reachable variable.

A variable is useless if it is not in one of the remaining rules.

Removal of Useless Variables

$$S \to aSb \mid BC \mid \lambda \qquad A \to Sb \qquad B \to a \qquad C \to C$$

Which variables are non-productive?

- C is not productive

We remove all rules containing non-productive variables:

$$S \to aSb \mid \lambda \qquad A \to Sb \qquad B \to a$$

Which variables are reachable from S?

- only S is reachable

We remove all rules containing non-reachable variables:

$$S \to aSb \mid \lambda$$

Hence only S is useful, the variables A, B, C are not useful.

Top-down Parsing

Top-down parsing tries to derive the input word from the starting variable S.

Simple leftmost strategy:

- Always expand the leftmost variable A.
  (Replace A by u if there is a production $A \to u$)
- Backtrack when a mismatch with the input string is found.
  Then try another production rule $A \to v$.

Disadvantage: backtracking is expensive and difficult.

LL Parsing

LL parsing

Parsing **top-down** with a **leftmost** strategy.

Backtracking is not allowed.

LL parsing does not work for every context-free grammar.

Starting point is a context-free grammar $G = (V, T, S, P)$:

- without useless variables
- λ-productions and unit productions are allowed (elimination often increases the size of the grammar)

Steps of LL parsing:

- Construct sets First(A) and Follow(A) for every variable A.
- Construct a parsing table.
- Parse the input word using the parsing table.

Erasable Variables

A variable A is called **erasable** if $A \Rightarrow^+ \lambda$.

The set of erasable variables is the smallest set such that:

- If $A \to \lambda$, then A is erasable.
- If $A \to B_1 \cdots B_n$ and $B_1, \ldots, B_n$ are erasable, then A is erasable.

Algorithm

The set of erasable variables can be constructed as follows:

1. Let $E = \varnothing$.
2. Repeat the following until there is nothing fresh to add:
   - If $A \to \lambda$, then add A to E.
   - If $A \to B_1 \cdots B_n$ and $B_1, \ldots, B_n$ are in E, then add A to E.
3. Finally, E is the set of erasable variables.

Erasable Variables

Consider the following grammar:

$$S \to AcB \qquad A \to CBC \qquad B \to abB \qquad C \to cCd$$
$$B \to \lambda \qquad C \to BB$$

We determine the set of erasable variables:

- B is erasable because of the rule $B \to \lambda$
- C is erasable because of $C \to BB$ and B is erasable
- A is erasable because of $A \to CBC$ and B, C are erasable

So the variables A, B, C are erasable.

First(A)

We consider the first terminal letters derivable from a word:

$$\text{First}(w) = \{ a \in T \mid w \Rightarrow^* a \ldots \} \cup \{ \lambda \mid w \Rightarrow^* \lambda \}$$

Algorithm

Let PreFirst(w) be the smallest set such that:

- $w \in \text{PreFirst}(w)$
- $a \in \text{PreFirst}(w)$ if $av \in \text{PreFirst}(w)$
- $B \in \text{PreFirst}(w)$ if $Bv \in \text{PreFirst}(w)$
- $v \in \text{PreFirst}(w)$ if $Bv \in \text{PreFirst}(w)$ and B erasable
- $v \in \text{PreFirst}(w)$ for every $A \in \text{PreFirst}(w)$ and rule $A \to v$

Then First(w) consists of

- all terminal letters $a \in T$ such that $a \in \text{PreFirst}(w)$, and
- λ if $w = A_1 A_2 \ldots A_n$ for erasable variables $A_1, \ldots, A_n$.

Exercise

$$S \to AAc \qquad A \to Ba \mid \lambda \qquad B \to Ab \mid d$$

The erasable variables ($V \Rightarrow^{+} \lambda$) are: A .

We determine PreFirst(A), PreFirst(B) and PreFirst(S):

$$\text{PreFirst}(A) = \{ A, \underbrace{Ba}_{\text{from } A}, \underbrace{\lambda}_{\text{from } A}, \underbrace{B}_{\text{from } Ba}, \underbrace{Ab}_{\text{from } B}, \underbrace{d}_{\text{from } B}, \underbrace{b}_{\text{from } Ab} \}$$

$$\text{PreFirst}(B) = \{ B, \underbrace{Ab}_{\text{from } B}, \underbrace{d}_{\text{from } B}, \underbrace{b}_{\text{from } Ab}, \underbrace{A}_{\text{from } Ab} \} \cup \text{PreFirst}(A)$$

$$= \{ A, Ba, \lambda, B, Ab, d, b \}$$

$$\text{PreFirst}(S) = \{ S, \underbrace{AAc}_{\text{from } S}, \underbrace{Ac}_{\text{from } AAc}, \underbrace{c}_{\text{from } Ac}, \underbrace{A}_{\text{from } AAc} \} \cup \text{PreFirst}(A)$$

$$= \{ S, AAc, Ac, c, A, Ba, \lambda, B, Ab, d, b \}$$

Thus we get

$$\text{First}(A) = \{ b, d, \lambda \} \quad \text{First}(B) = \{ b, d \} \quad \text{First}(S) = \{ b, c, d \}$$

Follow(A)

The sets First(A) are not yet sufficient for 'predictive' parsing, if there are derivations $A \Rightarrow^+ \lambda$.

We consider the terminal letters that can follow a variable:

$$\text{Follow}(A) = \{\, a \in T \mid S \Rightarrow^* \ldots Aa \ldots \}$$

Intuition: $a \in \text{Follow}(A)$ if A can be followed by a in a derivation.

We use $\$$ as a special '**end of word**' symbol.

Algorithm

- $\text{Follow}(S) \supseteq \{\$\}$
- $\text{Follow}(A) \supseteq \text{First}(w) \setminus \{\lambda\}$ for every rule $B \to vAw$
- $\text{Follow}(A) \supseteq \text{Follow}(B)$ for rules $B \to vAw$ with $\lambda \in \text{First}(w)$

Example

- $\text{Follow}(S) \supseteq \{\$\}$
- $\text{Follow}(A) \supseteq \text{First}(w) \setminus \{\lambda\}$ for every rule $B \to vAw$
- $\text{Follow}(A) \supseteq \text{Follow}(B)$ for rules $B \to vAw$ with $\lambda \in \text{First}(w)$

If $C \to AB$, then:

- $\text{First}(B) \subseteq \text{Follow}(A)$

  Example: $C \Rightarrow AB \Rightarrow^* Aaw$ if $B \to aw$

- $\text{Follow}(C) \subseteq \text{Follow}(B)$

  Example: $S \Rightarrow Ca \Rightarrow ABa$ if $S \to Ca$

- $\text{Follow}(C) \subseteq \text{Follow}(A)$ if $B \Rightarrow^* \lambda$

  Example: $S \Rightarrow Ca \Rightarrow ABa \Rightarrow Aa$ if $S \to Ca$ and $B \to \lambda$

Exercise

- $\text{Follow}(S) \supseteq \{\$\}$
- $\text{Follow}(A) \supseteq \text{First}(w) \setminus \{\lambda\}$ for every rule $B \to vAw$
- $\text{Follow}(A) \supseteq \text{Follow}(B)$ for rules $B \to vAw$ with $\lambda \in \text{First}(w)$

$$S \to Dc \qquad A \to Ba \mid \lambda$$
$$D \to AA \qquad B \to Ab \mid d$$

We have

$$\text{First}(S) = \{b, c, d\} \qquad \text{First}(A) = \{\lambda, b, d\}$$
$$\text{First}(D) = \{\lambda, b, d\} \qquad \text{First}(B) = \{b, d\}$$

Determine $\text{Follow}(S)$, $\text{Follow}(D)$, $\text{Follow}(A)$, $\text{Follow}(B)$:

$\text{Follow}(S) \supseteq \{\$\}$

$\text{Follow}(D) \supseteq \{c\}$

$\text{Follow}(A) \supseteq (\text{First}(A) \setminus \{\lambda\}) \cup \{b\} \cup \text{Follow}(D) \supseteq \{b, c, d\}$

$\text{Follow}(B) \supseteq \{a\}$

Parser Tables

The **parser table** for a context-free grammar is a table with

- columns indexed by terminals $T \cup \{\$\}$,
- rows indexed by variables V,

At place $[a \in T \cup \{\$\}, A \in V]$ it contains rules $A \to u$ for which

- $a \in \text{First}(u)$, or (never the case for $a = \$$)
- $\lambda \in \text{First}(u)$ and $a \in \text{Follow}(A)$.

$$S \to aSb \mid \lambda$$

We have

- $\text{First}(aSb) = \{a\}$, $\text{First}(\lambda) = \{\lambda\}$, $\text{First}(S) = \{\lambda, a\}$
- $\text{Follow}(S) = \{b, \$\}$,

Thus the parser table is:

	a	b	$\$$
S	$S \to aSb$	$S \to \lambda$	$S \to \lambda$

LL(1) Grammars and Parsing

A grammar is LL(1) if its parser table contains in ever cell $[a \in T \cup \{\$\}, A \in V]$ at most one production rule.

An LL(1) parser reads from Left to right, performs a Leftmost derivation, and looks always at 1 symbol of the input.

Given an LL(1)-grammar and parsing table.

To parse $a_1 \cdots a_n$, we start with $\langle S\$, a_1 \cdots a_n\$ \rangle$.

From a state $\langle v, w \rangle$ we can do the following steps:

- $\langle av', aw' \rangle$ becomes $\langle v', w' \rangle$
- $\langle Av', aw' \rangle$ becomes $\langle uv', aw' \rangle$ if $A \to u$ at position $[a, A]$
- $\langle Av', \$ \rangle$ becomes $\langle v', \$ \rangle$ if $A \to u$ at position $[\$, A]$
- $\langle \$, \$ \rangle$ results in **accept**
- In all other cases, $\langle v, w \rangle$ results in **reject**!

Example

$$S \to aSb \mid \lambda$$

The parser table is:

	a	b	$\$$
S	$S \to aSb$	$S \to \lambda$	$S \to \lambda$

$$\langle S\$, ab\$\rangle \to \langle aSb\$, ab\$\rangle \to \langle Sb\$, b\$\rangle \to \langle b\$, b\$\rangle \to \langle \$, \$\rangle \quad \textbf{accept}$$

$$\langle S\$, abb\$\rangle \to \langle aSb\$, abb\$\rangle \to \langle Sb\$, bb\$\rangle \to \langle b\$, bb\$\rangle \to \langle \$, b\$\rangle \quad \textbf{reject}$$

$$\langle S\$, aab\$\rangle \to \langle aSb\$, aab\$\rangle \to \langle Sb\$, ab\$\rangle \to \langle aSbb\$, ab\$\rangle \to \langle Sbb\$, b\$\rangle \to \langle bb\$, b\$\rangle \to \langle b\$, \$\rangle \quad \textbf{reject}$$

JavaCC (Java Compiler Compiler) automatically generates a parser from an LL(1) grammar.

Left Factorisation

Left factorisation: rewrite rules $A \to uv \mid uw$ ($u \neq \lambda$) into

$$A \to uB \qquad \text{and} \qquad B \to v \mid w$$

where B is a fresh variable.

The grammar $S \to ab \mid ac$ is **not** LL(1):

	a	b	c	$\$$
S	$S \to ab$			
	$S \to ac$			

After left factorisation we get: $S \to aA$ $A \to b \mid c$

The grammar $S \to aA$, $A \to b \mid c$ **is** LL(1):

	a	b	c	$\$$
S	$S \to aA$			
A		$A \to b$	$A \to c$	

LL(k) Grammars

The class of LL(1) grammars is often to restrictive in practice.

LL(1) parsers looks at 1 symbol to decide which rule to use.

An LL(k) parser looks k symbols ahead to choose the rule.

The parser table is constructed with k symbols look-ahead.

A grammar is LL(k) if this table has in every cell ≤ 1 rule.

LL(k) is strictly contained in LL($k+1$).

Disadvantage: size of the parser table grows exponential in k.

Exercises

Explain why these grammars cannot be LL(k) (for $k \geq 1$):

- ambiguous grammars
- $S \to aSa \mid \lambda$

Looking Back

- Context-free languages via context-free grammars
- Grammar simplifications & Chomsky normal form
- CYK parsing (bottom-up)
- LL and LL(1) parsing (top-down)

Looking Forward

Read:

- Linz 5.1–5.2, 6.1–6.3
- Grune & Jacobs H8 (LL parsing)

Do the following exercises:

- Linz 5.1: 2, 7b,e, 8b,h, 21, 22
- Linz 5.2: 14
- Linz 6.1: 6, 7, 9, 15, 16
- Linz 6.2: 3, 4
- Linz 6.3: 1, 2, 3
- Extra exercise about LL parsing

Following lecture:

- (Nondeterministic) pushdown automata