Compilers (Oversættere): Syntax Analysis

Jost Berthold
berthold@diku.dk
Department of Computer Science
Syntax Analysis (Parsing)

Program text
↓
Lexical analysis
↓
Token sequence
↓
Syntax analysis
↓
Syntax tree
↓
Typecheck
↓
Syntax tree
↓
Intermediate code generation

Binary machine code
↑
Assembly and linking
↑
Ditto with named registers
↑
Register allocation
↑
Symbolic machine code
↑
Machine code generation
↑
Intermediate code
Syntax Analysis

syntax analysis covers lecture
This.
Syntax Analysis

syntax analysis covers lecture
This.

Syntax error!

• Words (tokens) need to appear in the right order to form correct sentences (programs)
Syntax Analysis

syntax analysis covers lecture
This.

Syntax error!

• Words (tokens) need to appear in the right order to form correct sentences (programs) (not necessarily meaningful\(^1\)).

\(^1\)Traditional example (Noam Chomsky 1957): *Colorless green ideas sleep furiously.*
Syntax Analysis

syntax analysis covers lecture This.

Syntax error!

This analysis covers lecture syntax.

(semantic error)

• Words (tokens) need to appear in the right order to form correct sentences (programs) (not necessarily meaningful\(^1\)).

• Syntax analyser, commonly called parser,

• ... analyses token sequence to build program structure.

• Essential tool and theory used here: Context-free languages.

\(^1\) Traditional example (Noam Chomsky 1957): *Colorless green ideas sleep furiously.*
Contents and Goals of this Part

1. Context-Free Grammars and Languages

2. Top-Down Parsing, LL(1)
   - Recursive Parsing Functions (Recursive-descent)
   - First- and Follow-Sets
   - Look-Ahead Sets and LL(1) Parsing

3. Bottom-Up Parsing, SLR
   - Parser Generator Yacc
   - Shift-Reduce Parsing

4. Precedence and Associativity
Contents and Goals of this Part

1. Context-Free Grammars and Languages

2. Top-Down Parsing, LL(1)
   - Recursive Parsing Functions (Recursive-descent)
   - First- and Follow-Sets
   - Look-Ahead Sets and LL(1) Parsing

3. Bottom-Up Parsing, SLR
   - Parser Generator Yacc
   - Shift-Reduce Parsing

4. Precedence and Associativity

Goals:

- Use suitable context-free grammars to describe syntactic structure (especially for programming languages);
- Use parser generators and explain their inner workings;
- Know and use recursive-descent (top-down) parsing;
- Understand concepts and limitations of context-free parsing.
Context-Free Grammars

Definition (Context-Free Grammar)

A context-free grammar is given by

- a set of terminals $\Sigma$ (the alphabet of the resulting language),
- a set of nonterminals $N$,
- a start symbol $S \in N$
- a set $P$ of productions $X \rightarrow \alpha$ with a single nonterminal $X \in N$ on the left and a (possibly empty) right-hand side $\alpha \in (\Sigma \cup N)^*$ of terminals and nonterminals.

$$G : S \rightarrow aSB$$

$$S \rightarrow \varepsilon$$

$$S \rightarrow B$$

$$B \rightarrow Bb$$

$$B \rightarrow b$$
Definition (Context-Free Grammar)

A context-free grammar is given by

- a set of terminals $\Sigma$ (the alphabet of the resulting language),
- a set of nonterminals $N$,
- a start symbol $S \in N$,
- a set $P$ of productions $X \rightarrow \alpha$ with a single nonterminal $X \in N$ on the left and a (possibly empty) right-hand side $\alpha \in (\Sigma \cup N)^*$ of terminals and nonterminals.

Context-free grammars describe (context-free) languages over their terminal alphabet $\Sigma$.

Each nonterminal describes a set of words.

Nonterminals recursively refer to each other.

(cannot do that with regular expressions)
Context-Free Grammars

Definition (Context-Free Grammar)

A context-free grammar is given by

- a set of terminals \( \Sigma \) (the alphabet of the resulting language),
- a set of nonterminals \( N \),
- a start symbol \( S \in N \),
- a set \( P \) of productions \( X \rightarrow \alpha \) with a single nonterminal \( X \in N \) on the left and a (possibly empty) right-hand side \( \alpha \in (\Sigma \cup N)^* \) of terminals and nonterminals.

\[
\begin{align*}
G : & S \rightarrow aSB \\
    & S \rightarrow \varepsilon \\
    & S \rightarrow B \\
    & B \rightarrow Bb | b
\end{align*}
\]

- Context-free grammars describe (context-free) languages over their terminal alphabet \( \Sigma \).
- Each nonterminal describes a set of words.
- Nonterminals recursively refer to each other.
  (cannot do that with regular expressions)
Example, Derivation of Words

\[ G : S \rightarrow aSB \quad (1) \]
\[ S \rightarrow \varepsilon \quad (2) \]
\[ S \rightarrow B \quad (3) \]
\[ B \rightarrow Bb \quad (4) \]
\[ B \rightarrow b \quad (5) \]

Intuitive: Nonterminals describing sets

\[ S = \{ a \cdot x \cdot y \mid x \in S, y \in B \} \cup \{ \varepsilon \} \cup B \quad (1) \]
\[ B = \{ x \cdot b \mid x \in B \} \cup \{ b \} \quad (2) \]

Starting from the start symbol \( S \), words of the language can be derived by successively replacing nonterminals with right-hand sides.
Example, Derivation of Words

\[
G : S \rightarrow aSB \quad (1) \\
S \rightarrow \varepsilon \quad (2) \\
S \rightarrow B \quad (3) \\
B \rightarrow Bb \quad (4) \\
B \rightarrow b \quad (5)
\]

Intuitive: Nonterminals describing sets

\[
S = \{a \cdot x \cdot y \mid x \in S, y \in B\} \cup \{\varepsilon\} \cup B \\
B = \{x \cdot b \mid x \in B\} \cup \{b\}
\]

- Starting from the start symbol \(S\), ... 
- words of the language can be derived... 
- by successively replacing nonterminals with right-hand sides.

\[
S \xrightarrow{1} aSB \xrightarrow{1} aaSBB
\]
Example, Derivation of Words

\[ G : S \rightarrow aSB \quad (1) \]

\[ S \rightarrow \epsilon \quad (2) \]

\[ S \rightarrow B \quad (3) \]

\[ B \rightarrow Bb \quad (4) \]

\[ B \rightarrow b \quad (5) \]

Intuitive: Nonterminals describing sets

\[ S = \{ a \cdot x \cdot y \mid x \in S, y \in B \} \cup \{ \epsilon \} \cup \{b\} \]

\[ B = \{ x \cdot b \mid x \in B \} \cup \{b\} \]

- Starting from the start symbol \( S \),... 
- words of the language can be derived... 
- by successively replacing nonterminals with right-hand sides.

\[ S \xrightarrow{1} aSB \xrightarrow{1} aaSBB \xrightarrow{5} aaSbB \]
Example, Derivation of Words

\[ G : S \rightarrow aSB \quad (1) \]
\[ S \rightarrow \varepsilon \quad (2) \]
\[ S \rightarrow B \quad (3) \]
\[ B \rightarrow Bb \quad (4) \]
\[ B \rightarrow b \quad (5) \]

Intuitive: Nonterminals describing sets

\[ S = \{ a \cdot x \cdot y \mid x \in S, y \in B \} \cup \{ \varepsilon \} \cup B \quad (1) \]
\[ B = \{ x \cdot b \mid x \in B \} \cup \{ b \} \quad (2) \]

- Starting from the start symbol \( S \),... 
- words of the language can be derived... 
- by successively replacing nonterminals with right-hand sides.

\[ S \xrightarrow{1} aSB \xrightarrow{1} aaSBB \xrightarrow{5} aaSbB \xrightarrow{1} aaaSBbB \]
Example, Derivation of Words

**G**: $S \rightarrow aSB$ (1)

$S \rightarrow \varepsilon$ (2)

$S \rightarrow B$ (3)

$B \rightarrow Bb$ (4)

$B \rightarrow b$ (5)

Intuitive: Nonterminals describing sets

$$S = \{a \cdot x \cdot y \mid x \in S, y \in B\} \cup \{\varepsilon\} \cup B$$

$$B = \{x \cdot b \mid x \in B\} \cup \{b\}$$

• Starting from the start symbol $S$, ...  
  • words of the language can be derived...  
  • by successively replacing nonterminals with right-hand sides.

$$S \xrightarrow{1} aSB \xrightarrow{1} aaSBB \xrightarrow{5} aaSbB \xrightarrow{1} aaaSBbB$$

$$\xrightarrow{2} aaa_BbB$$
Example, Derivation of Words

\[ G : S \rightarrow aSB \] (1)

\[ S \rightarrow \varepsilon \] (2)

\[ S \rightarrow B \] (3)

\[ B \rightarrow Bb \] (4)

\[ B \rightarrow b \] (5)

Intuitive: Nonterminals describing sets

\[ S = \{ a \cdot x \cdot y \mid x \in S, y \in B \} \cup \{ \varepsilon \} \cup B \] (1)

\[ B = \{ x \cdot b \mid x \in B \} \cup \{ b \} \] (2)

Starting from the start symbol \( S \),...

words of the language can be derived...

by successively replacing nonterminals with right-hand sides.

\[ S \stackrel{1}{\Rightarrow} aSB \stackrel{1}{\Rightarrow} aaSBB \stackrel{5}{\Rightarrow} aaSbB \stackrel{1}{\Rightarrow} aaaSBbB \]

\[ \Rightarrow^2 \text{aaa}_BbB \stackrel{4}{\Rightarrow} \text{aaaBbbB} \]
Example, Derivation of Words

\[ G : S \rightarrow aSB \quad (1) \]

\[ S \rightarrow \varepsilon \quad (2) \]

\[ S \rightarrow B \quad (3) \]

\[ B \rightarrow Bb \quad (4) \]

\[ B \rightarrow b \quad (5) \]

Intuitive: Nonterminals describing sets

\[ \mathcal{S} = \{ a \cdot x \cdot y | x \in \mathcal{S}, y \in \mathcal{B} \} \cup \{ \varepsilon \} \cup \mathcal{B} \quad (1) \]

\[ \mathcal{B} = \{ x \cdot b | x \in \mathcal{B} \} \cup \{ b \} \quad (2, 3, 4, 5) \]

• Starting from the start symbol \( S \), ...
• words of the language can be derived...
• by successively replacing nonterminals with right-hand sides.

\[ S \xrightarrow{1} aSB \xrightarrow{1} aaSBB \xrightarrow{5} aaSbB \xrightarrow{1} aaaSBBbB \]

\[ \xrightarrow{2} aaa_BbB \xrightarrow{4} aaaSBBbB \xrightarrow{5} aaaSBBBbB \xrightarrow{5} aaabbbb \]
Derivation Relation

Definition (Derivation ⇒)

Let $G = (\Sigma, N, S, P)$ be a grammar.

The derivation relation $\Rightarrow$ on $(\Sigma \cup N)^*$ is defined as follows:

- For an $X \in N$ and a production $(X \rightarrow \beta) \in P$ of the grammar, $\alpha_1 X \alpha_2 \Rightarrow \alpha_1 \beta \alpha_2$ for all $\alpha_1, \alpha_2 \in (\Sigma \cup N)^*$.

- Describes one derivation step using one of the productions.
- Can indicate used production by a number ($\Rightarrow_k$).
- Can indicate left-most (or right-most) derivation ($\Rightarrow_l, \Rightarrow_r$).
Derivation Relation

**Definition (Derivation ⇒)**

Let \( G = (\Sigma, N, S, P) \) be a grammar. The **derivation relation** \( \Rightarrow \) on \((\Sigma \cup N)^*\) is defined as follows:

- For an \( X \in N \) and a production \((X \rightarrow \beta) \in P\) of the grammar, \( \alpha_1X\alpha_2 \Rightarrow \alpha_1\beta\alpha_2 \) for all \( \alpha_1, \alpha_2 \in (\Sigma \cup N)^* \).

- Describes one derivation step using one of the productions.
- Can indicate **used production** by a number \((\Rightarrow^k)\).
- Can indicate **left-most** (or **right-most**) derivation \((\Rightarrow^l, \Rightarrow^r)\).

\[ G : S \rightarrow aSB \quad (1) \]
\[ S \rightarrow \varepsilon \quad (2) \]
\[ S \rightarrow B \quad (3) \]
\[ B \rightarrow Bb \quad (4) \]
\[ B \rightarrow b \quad (5) \]

\[ S \xrightarrow{1} aSB \xrightarrow{1} aaSBB \xrightarrow{2} aa\_BB \]
Derivation Relation

Definition (Derivation ⇒)

Let \( G = (\Sigma, N, S, P) \) be a grammar.
The derivation relation ⇒ on \((\Sigma \cup N)^*\) is defined as follows:

- For an \( X \in N \) and a production \((X \rightarrow \beta) \in P\) of the grammar, \( \alpha_1 X \alpha_2 \Rightarrow \alpha_1 \beta \alpha_2 \) for all \( \alpha_1, \alpha_2 \in (\Sigma \cup N)^* \).

- Describes one derivation step using one of the productions.
- Can indicate used production by a number (⇒\(k\)).
- Can indicate left-most (or right-most) derivation (⇒\(l\), ⇒\(r\)).

\[ G : S \rightarrow aSB \quad (1) \]
\[ S \rightarrow \varepsilon \quad (2) \]
\[ S \rightarrow B \quad (3) \]
\[ B \rightarrow Bb \quad (4) \]
\[ B \rightarrow b \quad (5) \]

\[ S \Rightarrow^1 aSB \Rightarrow^1 aaSBB \Rightarrow^2 aaBB \]
\[ \Rightarrow^4 aaBBbB \Rightarrow^5 aabbB \Rightarrow^5 aabbb \]
Extended Derivation Relation (Transitive Closure)

Definition (Transitive Derivation Relation $\Rightarrow^*$)

Let $G = (\Sigma, N, S, P)$ be a grammar and $\Rightarrow$ its derivation relation. The **transitive derivation relation** of $G$ is defined as:

- $\alpha \Rightarrow^* \alpha$ for all $\alpha \in (\Sigma \cup N)^*$ (derived in 0 steps).
- For $\alpha, \beta \in (\Sigma \cup N)^*$, $\alpha \Rightarrow^* \beta$ if there exists a $\gamma \in (\Sigma \cup N)^*$ such that $\alpha \Rightarrow \gamma$ and $\gamma \Rightarrow^* \beta$ (derived in at least one step).

More generally, this is known as the **transitive closure** of a relation.
Extended Derivation Relation (Transitive Closure)

**Definition (Transitive Derivation Relation $$\Rightarrow^*$$)**

Let $$G = (\Sigma, N, S, P)$$ be a grammar and $$\Rightarrow$$ its derivation relation. The **transitive derivation relation** of $$G$$ is defined as:

- $$\alpha \Rightarrow^* \alpha$$ for all $$\alpha \in (\Sigma \cup N)^*$$ (derived in 0 steps).
- For $$\alpha, \beta \in (\Sigma \cup N)^*$$, $$\alpha \Rightarrow^* \beta$$ if there exists a $$\gamma \in (\Sigma \cup N)^*$$ such that $$\alpha \Rightarrow \gamma$$ and $$\gamma \Rightarrow^* \beta$$ (derived in at least one step).

More generally, this is known as the **transitive closure** of a relation. In our previous examples, we saw $$S \Rightarrow^* aaabbbb$$ and $$S \Rightarrow^* aabbb$$. That means, both words are in the language of $$G$$.

**Definition (Language of a Grammar)**

Let $$G = (\Sigma, N, S, P)$$ be a grammar and $$\Rightarrow$$ its derivation relation. The language of the grammar is $$L(G) = \{ w \in \Sigma^* \mid S \Rightarrow^* w \}$$.
Syntax Tree and Directed Derivation

\[ G : S \rightarrow aSB \] (1)
\[ S \rightarrow \varepsilon \] (2)
\[ S \rightarrow B \] (3)
\[ B \rightarrow Bb \] (4)
\[ B \rightarrow b \] (5)

- Syntax trees describe the derivation independent of the direction.

\[ a \quad S \quad B \]
\[
\begin{array}{c}
S \\
\text{a} \\
S \\
\text{a} \\
\text{S} \\
\text{B} \\
\text{b} \\
\varepsilon \\
B \\
\text{b} \\
b \\
\end{array}
\]
Syntax Tree and Directed Derivation

\[ G : S \rightarrow aSB \quad (1) \]
\[ S \rightarrow \varepsilon \quad (2) \]
\[ S \rightarrow B \quad (3) \]
\[ B \rightarrow Bb \quad (4) \]
\[ B \rightarrow b \quad (5) \]

- Syntax trees describe the derivation independent of the direction.
- Left-most derivation: depth-first left-to-right tree traversal.

\[ S \xrightarrow{1} aSB \xrightarrow{1} aaSBB \xrightarrow{2} aaBB \xrightarrow{4} aaBbB \xrightarrow{5} aabbB \xrightarrow{5} aabbb \]
Syntax Tree and Directed Derivation

\[ G : S \rightarrow aSB \quad (1) \]
\[ S \rightarrow \varepsilon \quad (2) \]
\[ S \rightarrow B \quad (3) \]
\[ B \rightarrow Bb \quad (4) \]
\[ B \rightarrow b \quad (5) \]

- Syntax trees describe the derivation independent of the direction.
- Left-most derivation: depth-first left-to-right tree traversal.
- \[ S \Rightarrow aSB \Rightarrow aaSB \Rightarrow aaBB \Rightarrow aabB \Rightarrow aabbbb \]

Nevertheless: \( S \Rightarrow^* aabbbb \) can be derived in two ways.
- \[ S \Rightarrow aSB \Rightarrow aaSB \Rightarrow aaBB \Rightarrow aabB \Rightarrow aabbbb \]

The grammar G is said to be ambiguous.
Avoiding Ambiguity (Changed Grammar)

\[
G : S \rightarrow aSB \\
S \rightarrow \varepsilon \\
S \rightarrow B \\
B \rightarrow Bb \\
B \rightarrow b
\]

Your grammar here

Modify the grammar to make it non-ambiguous. (describing the same language), give a syntax tree for \texttt{aabbb}.

- Idea: generate extra bs separately
Avoiding Ambiguity (Changed Grammar)

\begin{align*}
G : & \quad S \rightarrow aSB \\
G' : & \quad S \rightarrow \text{AB} \quad (1) \\
& \quad S \rightarrow \varepsilon \\
& \quad S \rightarrow B \\
& \quad B \rightarrow Bb \\
& \quad B \rightarrow b
\end{align*}

\begin{align*}
& \quad A \rightarrow aAb \quad (2) \\
& \quad A \rightarrow \varepsilon \quad (3) \\
& \quad B \rightarrow bB \quad (4) \\
& \quad B \rightarrow \varepsilon \quad (5)
\end{align*}

Modify the grammar to make it non-ambiguous. (describing the same language), give a syntax tree for $aabbb$.

- Idea: generate extra bs separately by new start production
- Avoiding left-recursion (explained later)
Avoiding Ambiguity (Changed Grammar)

\[ G : S \rightarrow aSB \quad G' : S \rightarrow AB \quad (1) \]

\[ S \rightarrow \varepsilon \quad A \rightarrow aAb \quad (2) \]

\[ S \rightarrow B \quad A \rightarrow \varepsilon \quad (3) \]

\[ B \rightarrow Bb \quad B \rightarrow bB \quad (4) \]

\[ B \rightarrow b \quad B \rightarrow \varepsilon \quad (5) \]

Modify the grammar to make it non-ambiguous. (describing the same language), give a syntax tree for \texttt{aabbb}.

- Idea: generate extra bs separately by new start production
- Avoiding left-recursion (explained later)
- Left-most derivation: (1 2 2 3 4 5)

\[ S \xrightarrow{1} AB \xrightarrow{2} aAbB \xrightarrow{2} aaAbB \xrightarrow{3} aa_bbB \xrightarrow{4} aabbbB \xrightarrow{5} aabbb \]
Parsing

- Producing a syntax tree from a token sequence.
- Representation of the tree: left-most or right-most derivation
Parsing

Token sequence → Syntax analysis → Syntax tree

- Producing a syntax tree from a token sequence.
- Representation of the tree: left-most or right-most derivation

Two approaches

- **Top-Down Parsing:** Builds syntax tree from the root. Builds a left-most derivation sequence
- **Bottom-Up Parsing:** Builds syntax tree from the leaves. Builds a reversed right-most derivation sequence

- Both: use stack to keep track of derivation.
Idea of Top-Down Parsing

- Recursive functions modelling the productions ("recursive-descent")

```
fun parseS () = print "parsing_S: prod_1"
  (* one production S -> A B *)
parseA (); parseB (); match EOF
```

```
and parseA () =
  (* choose A -> a A b or A -> <epsilon > *)
  if should_use_production_2
    then print "parsing_A: prod_2"
    match #"a" ; parseA (); match #"b"
  else print "parsing_A: prod_3" ;()

and parseB () =
  (* choose B -> b B or B -> <epsilon > *)
  if should_use_production_4
    then print "parsing_B: prod_4"
    match #"b" ; parseB ()
  else print "parsing_B: prod_5" ;()
```
Idea of Top-Down Parsing

- Recursive functions modelling the productions ("recursive-descent")

```
fun parseS () = print "parsing\$S:prod\$1";
  (* one production S -> A B *)
  parseA (); parseB (); match EOF

and parseA () =
  (* choose A -> a A b or A -> \epsilon *)
  if should_use_production_2
    then print "parsing\$A:prod\$2";
        match #"a"; parseA (); match #"b"
    else print "parsing\$A:prod\$3"();
```
Idea of Top-Down Parsing

- Recursive functions modelling the productions ("recursive-descent")

```haskell
fun parseS () =
  print "parsing\nS: prod 1";
  (* one production S -> A B *)
  parseA (); parseB (); match EOF

and parseA () =
  (* choose A -> a A b or A -> \epsilon *)
  if should_use_production_2
  then print "parsing\nA: prod 2";
      match #"a"; parseA (); match #"b"
  else print "parsing\nA: prod 3"();
```

```haskell
6
HHH
A
@@
A
a
A
@@
aabb
```
Idea of Top-Down Parsing

- Recursive functions modelling the productions ("recursive-descent")

```
fun parseS () = print "parsing\nS: prod 1";
   (* one production S -> A B *)
   parseA (); parseB (); match EOF

and parseA () =
   (* choose A -> a A b or A -> \epsilon *)
   if should_use_production_2
      then print "parsing\nA: prod 2";
          match #"a"; parseA (); match #"b"
      else print "parsing\nA: prod 3";()
```

```
Idea of Top-Down Parsing

- Recursive functions modelling the productions ("recursive-descent")

```haskell
fun parseS () = print "parsing S: prod 1";
    (* one production S -> A B *)
    parseA (); parseB (); match EOF

and parseA () =
    (* choose A -> a A b or A -> <epsilon> *)
    if should_use_production_2
        then print "parsing A: prod 2";
            match #"a"; parseA (); match #"b"
        else print "parsing A: prod 3";
            ()
```

How can we decide which production to use?
Idea of Top-Down Parsing

- Recursive functions modelling the productions ("recursive-descent")

```ml
fun parseS () = print "parsing S: prod 1";
    (* one production S -> A B *)
    parseA (); parseB (); match EOF

and parseA () =
    (* choose A -> a A b or A -> <epsilon> *)
    if should_use_production_2
        then print "parsing A: prod 2";
            match #"a"; parseA (); match #"b"
        else print "parsing A: prod 3"();

and parseB () =
    (* choose B -> b B or B -> <epsilon> *)
    if should_use_production_4
        then print "parsing B: prod 4";
            match #"b"; parseB()
        else print "parsing B: prod 5"();
```
Idea of Top-Down Parsing

- Recursive functions modelling the productions ("recursive-descent")

```
fun parseS () = print "parsing S: prod 1";
(* one production S -> A B *)
parseA (); parseB (); match EOF

and parseA () =
(* choose A -> a A b or A -> <epsilon> *)
if should_use_production_2
  then print "parsing A: prod 2";
       match #"a"; parseA (); match #"b"
  else print "parsing A: prod 3"();

and parseB () =
(* choose B -> b B or B -> <epsilon> *)
if should_use_production_4
  then print "parsing B: prod 4";
       match #"b"; parseB()
  else print "parsing B: prod 5"();
```
Top-Down Parsing (LL(1) Parsing)

- Producing a left-most derivation from a token sequence.
- Uses a stack (maybe the function call stack) to keep track of derivation.

- Called predictive parsing: needs to “guess” used productions.
Top-Down Parsing (LL(1) Parsing)

- Producing a left-most derivation from a token sequence.
- Uses a stack (maybe the function call stack) to keep track of derivation.

- Called predictive parsing: needs to “guess” used productions.
- Information to choose the right production (look-ahead):
  - For each right-hand side: What input token can come first?
  - Special attention to empty right-hand sides. What can follow?
Top-Down Parsing (LL(1) Parsing)

- Producing a left-most derivation from a token sequence.
- Uses a stack (maybe the function call stack) to keep track of derivation.

- Called **predictive parsing**: needs to “guess” used productions.

- Information to choose the right production (**look-ahead**):
  - For each right-hand side: What input token can come first?
  - Special attention to empty right-hand sides. What can follow?

- A production $A \rightarrow \alpha$ is chosen
  - if look-ahead $c$ and $\alpha \Rightarrow^* c\beta$ (can start with $c$).
  - or if look-ahead $c$, $\alpha \Rightarrow^* \varepsilon$, and $c$ can follow $A$. 
First Sets and Property Nullable

Definition (FIRST set and NULLABLE)

Let $G = (\Sigma, N, S, P)$ a grammar and $\Rightarrow$ its derivation relation. For all sequences of grammar symbols $\alpha \in (\Sigma \cup N)^*$, define

- $\text{FIRST}(\alpha) = \{ c \in \Sigma \mid \exists \beta \in (\Sigma \cup N)^* : \alpha \Rightarrow^* c\beta \}$
  (all terminals at the start of what can be derived from $\alpha$)

- $\text{Nullable}(\alpha) = \begin{cases} 
  \text{true} & \text{if } \alpha \Rightarrow^* \varepsilon \\
  \text{false} & \text{otherwise}
\end{cases}$
**First Sets and Property Nullable**

**Definition (First set and Nullable)**

Let $G = (\Sigma, N, S, P)$ a grammar and $\Rightarrow$ its derivation relation. For all sequences of grammar symbols $\alpha \in (\Sigma \cup N)^*$, define

- $\text{First}(\alpha) = \{ c \in \Sigma \mid \exists \beta \in (\Sigma \cup N)^* : \alpha \Rightarrow^* c\beta \}$
  (all terminals at the start of what can be derived from $\alpha$)

- $\text{Nullable}(\alpha) = \begin{cases} true, & \text{if } \alpha \Rightarrow^* \varepsilon \\ false, & \text{otherwise} \end{cases}$

Computing Nullable and First for right-hand sides:

- Set equations recursively use results for nonterminals.
- Smallest solution found by computing a smallest fixed-point.
- Solved simultaneously for all right-hand sides of the productions.
Computing **Nullable** by Set Equations

\[
\begin{align*}
\text{Nullable}(\varepsilon) &= \text{true} \\
\text{Nullable}(a) &= \text{false} \text{ for } a \in \Sigma \\
\text{Nullable}(\alpha\beta) &= \text{Nullable}(\alpha) \land \text{Nullable}(\beta) \text{ for } \alpha, \beta \in (\Sigma \cup \mathbb{N})^* \\
\text{Nullable}(A) &= \text{Nullable}(\alpha_1) \lor \ldots \lor \text{Nullable}(\alpha_n), \text{ using all productions for } A, A \rightarrow \alpha_i \ (i \in \{1..n\})
\end{align*}
\]
Computing Nullable by Set Equations

\[
\begin{align*}
\text{Nullable}(\varepsilon) &= \text{true} \\
\text{Nullable}(a) &= \text{false} \text{ for } a \in \Sigma \\
\text{Nullable}(\alpha\beta) &= \text{Nullable}(\alpha) \land \text{Nullable}(\beta) \text{ for } \alpha, \beta \in (\Sigma \cup N)^* \\
\text{Nullable}(A) &= \text{Nullable}(\alpha_1) \lor \ldots \lor \text{Nullable}(\alpha_n), \\
&\quad \text{using all productions for } A, A \rightarrow \alpha_i \ (i \in \{1..n\})
\end{align*}
\]

- Equations for nonterminals of the grammar:

\[
G' : S \rightarrow AB \\
A \rightarrow aAb | \varepsilon \\
B \rightarrow bB | \varepsilon
\]

\[
\begin{align*}
\text{Nullable}(S) &= \text{Nullable}(AB) \\
\text{Nullable}(A) &= \text{Nullable}(aAb) \lor \text{Nullable}(\varepsilon) \\
\text{Nullable}(B) &= \text{Nullable}(bB) \lor \text{Nullable}(\varepsilon)
\end{align*}
\]
Computing **Nullable** by Set Equations

\[
\begin{align*}
\text{Nullable}(\varepsilon) &= \text{true} \\
\text{Nullable}(a) &= \text{false} \text{ for } a \in \Sigma \\
\text{Nullable}(\alpha\beta) &= \text{Nullable}(\alpha) \land \text{Nullable}(\beta) \text{ for } \alpha, \beta \in (\Sigma \cup N)^* \\
\text{Nullable}(A) &= \text{Nullable}(\alpha_1) \lor \ldots \lor \text{Nullable}(\alpha_n), \\
&\quad \text{using all productions for } A, A \rightarrow \alpha_i \ (i \in \{1..n\})
\end{align*}
\]

- **Equations for nonterminals of the grammar:**
  \[
  G' : \begin{array}{c}
  S \rightarrow AB \\
  A \rightarrow aAb | \varepsilon \\
  B \rightarrow bB | \varepsilon
  \end{array}
  \begin{align*}
  \text{Nullable}(S) &= \text{Nullable}(AB) \\
  \text{Nullable}(A) &= \text{Nullable}(aAb) \lor \text{Nullable}(\varepsilon) \\
  \text{Nullable}(B) &= \text{Nullable}(bB) \lor \text{Nullable}(\varepsilon)
  \end{align*}
  \]
  
  - **Equations for the right-hand side**
    \[
    \begin{align*}
    \text{Nullable}(AB) &= \text{Nullable}(A) \land \text{Nullable}(B) \\
    \text{Nullable}(aAb) &= \text{Nullable}(a) \land \text{Nullable}(A) \land \text{Nullable}(b) \\
    \text{Nullable}(bB) &= \text{Nullable}(b) \land \text{Nullable}(B) \\
    \text{Nullable}(\varepsilon) &= \text{true}
    \end{align*}
    \]

Compute smallest solution of system, starting by \textit{false} for all.
Computing Nullable by Set Equations

- Nullable of ε = true
- Nullable of a = false for a ∈ Σ
- Nullable of αβ = Nullable(α) ∧ Nullable(β) for α, β ∈ (Σ ∪ N)*
- Nullable of A = Nullable(α₁) ∨ ... ∨ Nullable(αₙ),
  using all productions for A, A → αᵢ (i ∈ {1..n})

- Equations for nonterminals of the grammar:
  \[ G' : S \rightarrow AB \]
  \[ A \rightarrow aAb | \varepsilon \]
  \[ B \rightarrow bB | \varepsilon \]
  \[ \text{Nullable}(S) = \text{Nullable}(AB) \]
  \[ \text{Nullable}(A) = \text{Nullable}(aAb) \lor \text{Nullable}(\varepsilon) \]
  \[ \text{Nullable}(B) = \text{Nullable}(bB) \lor \text{Nullable}(\varepsilon) \]

- Equations for the right-hand side
  \[ \text{Nullable}(AB) = \text{Nullable}(A) \land \text{Nullable}(B) \]
  \[ \text{Nullable}(aAb) = \text{Nullable}(a) \land \text{Nullable}(A) \land \text{Nullable}(b) = \text{false} \]
  \[ \text{Nullable}(bB) = \text{Nullable}(b) \land \text{Nullable}(B) = \text{false} \]
  \[ \text{Nullable}(\varepsilon) = \text{true} \]

Compute smallest solution of system, starting by false for all.
Computing **Nullable** by Set Equations

\[
\begin{align*}

\text{Nullable}(\varepsilon) &= \text{true} \\
\text{Nullable}(a) &= \text{false} \text{ for } a \in \Sigma \\
\text{Nullable}(\alpha \beta) &= \text{Nullable}(\alpha) \land \text{Nullable}(\beta) \text{ for } \alpha, \beta \in (\Sigma \cup N)^* \\
\text{Nullable}(A) &= \text{Nullable}(\alpha_1) \lor \ldots \lor \text{Nullable}(\alpha_n), \\
&\text{using all productions for } A, A \rightarrow \alpha_i \ (i \in \{1..n\})
\end{align*}
\]

- Equations for nonterminals of the grammar:

\[
\begin{align*}
G' : S &\rightarrow AB & \text{Nullable}(S) &= \text{Nullable}(AB) = \text{true} \\
A &\rightarrow aAb | \varepsilon & \text{Nullable}(A) &= \text{Nullable}(aAb) \lor \text{Nullable}(\varepsilon) = \text{true} \\
B &\rightarrow bB | \varepsilon & \text{Nullable}(B) &= \text{Nullable}(bB) \lor \text{Nullable}(\varepsilon) = \text{true}
\end{align*}
\]

- Equations for the right-hand side

\[
\begin{align*}
\text{Nullable}(AB) &= \text{Nullable}(A) \land \text{Nullable}(B) \\
\text{Nullable}(aAb) &= \text{Nullable}(a) \land \text{Nullable}(A) \land \text{Nullable}(b) = \text{false} \\
\text{Nullable}(bB) &= \text{Nullable}(b) \land \text{Nullable}(B) = \text{false} \\
\text{Nullable}(\varepsilon) &= \text{true}
\end{align*}
\]

Compute smallest solution of system, starting by **false** for all.
Computing FIRST by Set Equations

\[
\begin{align*}
\text{FIRST}(\varepsilon) & = \emptyset \\
\text{FIRST}(a) & = a \text{ for } a \in \Sigma \\
\text{FIRST}(\alpha\beta) & = \begin{cases} 
\text{FIRST}(\alpha) \cup \text{FIRST}(\beta), & \text{if NULLABLE}(\alpha) \\
\text{FIRST}(\alpha), & \text{otherwise}
\end{cases} \\
\text{FIRST}(A) & = \text{FIRST}(\alpha_1) \cup \ldots \cup \text{FIRST}(\alpha_n), \\
& \quad \text{using all productions for } A, \ A \to \alpha_i \ (i \in \{1..n\})
\end{align*}
\]
Computing \textsc{First} by Set Equations

\begin{align*}
\text{First}(\varepsilon) &= \emptyset \\
\text{First}(a) &= a \text{ for } a \in \Sigma \\
\text{First}(\alpha\beta) &= \begin{cases} 
\text{First}(\alpha) \cup \text{First}(\beta), & \text{if Nullable}(\alpha) \\
\text{First}(\alpha), & \text{otherwise}
\end{cases} \\
\text{First}(A) &= \text{First}(\alpha_1) \cup \ldots \cup \text{First}(\alpha_n), \\
& \text{using all productions for } A, A \rightarrow \alpha_i \ (i \in \{1..n\})
\end{align*}

- Equations for nonterminals of the grammar:

\begin{align*}
G' : S & \rightarrow AB & \text{First}(S) &= \text{First}(AB) \\
A & \rightarrow aAb | \varepsilon & \text{First}(A) &= \text{First}(aAb) \cup \text{First}(\varepsilon) \\
B & \rightarrow bB | \varepsilon & \text{First}(B) &= \text{First}(bB) \cup \text{First}(\varepsilon)
\end{align*}
Computing \textsc{First} by Set Equations

\begin{align*}
\text{First}(\varepsilon) &= \emptyset \\
\text{First}(a) &= a \text{ for } a \in \Sigma \\
\text{First}(\alpha\beta) &= \begin{cases} 
\text{First}(\alpha) \cup \text{First}(\beta), & \text{if } \text{Nullable}(\alpha) \\
\text{First}(\alpha), & \text{otherwise}
\end{cases} \\
\text{First}(A) &= \text{First}(\alpha_1) \cup \ldots \cup \text{First}(\alpha_n), \\
&\text{using all productions for } A, A \rightarrow \alpha_i (i \in \{1..n\})
\end{align*}

- Equations for nonterminals of the grammar:

\begin{align*}
G' : S &\rightarrow AB & \text{First}(S) &= \text{First}(AB) = \text{First}(A) \cup \text{First}(B) \\
A &\rightarrow aAb | \varepsilon & \text{First}(A) &= \text{First}(aAb) \cup \text{First}(\varepsilon) \\
B &\rightarrow bB | \varepsilon & \text{First}(B) &= \text{First}(bB) \cup \text{First}(\varepsilon)
\end{align*}

- Equations for the right-hand side

\begin{align*}
\text{First}(aAB) &= \text{First}(a) \\
\text{First}(bB) &= \text{First}(b) \\
\text{First}(\varepsilon) &= \emptyset
\end{align*}
Computing \textit{FIRST} by Set Equations

\begin{itemize}
  \item Equations for nonterminals of the grammar:

\begin{align*}
  G' : S &\rightarrow AB & \text{FIRST}(S) &= \text{FIRST}(AB) = \text{FIRST}(A) \cup \text{FIRST}(B) \\
  A &\rightarrow aAb | \varepsilon & \text{FIRST}(A) &= \text{FIRST}(aAb) \cup \text{FIRST}(\varepsilon) \\
  B &\rightarrow bB | \varepsilon & \text{FIRST}(B) &= \text{FIRST}(bB) \cup \text{FIRST}(\varepsilon)
\end{align*}

\item Equations for the right-hand side

\begin{align*}
  \text{FIRST}(aAB) &= \text{FIRST}(a) = \{ a \} \\
  \text{FIRST}(bB) &= \text{FIRST}(b) = \{ b \} \\
  \text{FIRST}(\varepsilon) &= \emptyset
\end{align*}

\end{itemize}

Compute smallest solution of system, starting by \( \emptyset \) for all sets.
Computing \textsc{First} by Set Equations

\begin{align*}
\text{First}(\varepsilon) & = \emptyset \\
\text{First}(a) & = a \text{ for } a \in \Sigma \\
\text{First}(\alpha\beta) & = \begin{cases} 
\text{First}(\alpha) \cup \text{First}(\beta), & \text{if Nullable}(\alpha) \\
\text{First}(\alpha), & \text{otherwise}
\end{cases} \\
\text{First}(A) & = \text{First}(\alpha_1) \cup \ldots \cup \text{First}(\alpha_n), \\
& \text{using all productions for } A, A \rightarrow \alpha_i \ (i \in \{1..n\})
\end{align*}

• Equations for nonterminals of the grammar:

\begin{align*}
G' : S & \rightarrow AB & \text{First}(S) & = \text{First}(AB) = \text{First}(A) \cup \text{First}(B) = \{a, b\} \\
A & \rightarrow aAb | \varepsilon & \text{First}(A) & = \text{First}(aAb) \cup \text{First}(\varepsilon) = \{a\} \\
B & \rightarrow bB | \varepsilon & \text{First}(B) & = \text{First}(bB) \cup \text{First}(\varepsilon) = \{b\}
\end{align*}

• Equations for the right-hand side

\begin{align*}
\text{First}(aAB) & = \text{First}(a) = \{a\} \\
\text{First}(BB) & = \text{First}(b) = \{b\} \\
\text{First}(\varepsilon) & = \emptyset
\end{align*}

Compute smallest solution of system, starting by \(\emptyset\) for all sets.
**Follow Sets for Nonterminals**

**First** Sets are often not enough. In production $X \rightarrow \alpha$, if $\text{NULLABLE}(\alpha)$, we need to know what can follow $X$ (First set of $\alpha$ cannot provide this information).

**Definition (Follow Set of a Nonterminal)**

Let $G = (\Sigma, N, S, P)$ a grammar and $\Rightarrow$ its derivation relation. For each nonterminal $X \in N$, define

- $\text{Follow}(X) = \{ c \in \Sigma | \exists \alpha, \beta \in (\Sigma \cup N)^* : S \Rightarrow^* \alpha X \beta \}$ (all input tokens that follow $X$ in sequences derivable from $S$)

To recognise the end of the input:

- add a new character $\$ to the alphabet
- add start production $S' \rightarrow S$ to the grammar.

Thereby, we always have $\$ \in \text{Follow}(S)$. 
Follow Sets for Nonterminals

First Sets are often not enough. In production $X \rightarrow \alpha$, if $\text{Nullable}(\alpha)$, we need to know what can follow $X$ ($\text{FIRST}$ set of $\alpha$ cannot provide this information).

**Definition (Follow Set of a Nonterminal)**

Let $G = (\Sigma, N, S, P)$ a grammar and $\Rightarrow$ its derivation relation. For each nonterminal $X \in N$, define

- $\text{Follow}(X) = \{ c \in \Sigma \mid \exists \alpha, \beta \in (\Sigma \cup N)^* : S \Rightarrow^* \alpha X c \beta \}$

(all input tokens that follow $X$ in sequences derivable from $S$)

To recognise the end of the input

- add a new character $\$$ to the alphabet
- add start production $S' \rightarrow S\$$ to the grammar.

Thereby, we always have $\$$ \in \text{Follow}(S)$. 
Set Equations for **Follow** Sets

**Follow** sets solve a collection of set constraints.

Constraints derived from **right-hand sides** of grammar productions

For \( X \in N \), consider all productions \( Y \rightarrow \alpha X \beta \) where \( X \) occurs on the right.

- \( \text{FIRST}(\beta) \subseteq \text{FOLLOW}(X) \)
- If \( \text{NULLABLE}(\beta) \) or \( \beta = \epsilon \): \( \text{FOLLOW}(Y) \subseteq \text{FOLLOW}(X) \)

If \( X \) occurs several times, each occurrence contributes separate equations.
Set Equations for \textbf{Follow} Sets

\textbf{Follow} sets solve a collection of set constraints.

Constraints derived from right-hand sides of grammar productions

For $X \in N$, consider all productions $Y \rightarrow \alpha X \beta$ where $X$ occurs on the right.

- $\text{FIRST}(\beta) \subseteq \text{Follow}(X)$
- If $\text{Nullable}(\beta)$ or $\beta = \varepsilon$: $\text{Follow}(Y) \subseteq \text{Follow}(X)$

If $X$ occurs several times, each occurrence contributes separate equations.

\[
\begin{align*}
S' & \rightarrow S$ \quad \text{FIRST}(\$) = \{\$\} \subseteq \text{Follow}(S) \\
S & \rightarrow AB \quad \text{FIRST}(B) = \{b\} \subseteq \text{Follow}(A) \\
& \quad \text{Follow}(S) \subseteq \text{Follow}(A) \quad (B \text{ nullable}) \\
A & \rightarrow aAb \quad \text{FIRST}(b) = \{b\} \subseteq \text{Follow}(A) \\
B & \rightarrow bB \quad \text{Follow}(B) \subseteq \text{Follow}(B) \\
& \quad A \rightarrow \varepsilon \text{ and } B \rightarrow \varepsilon \text{ do not contribute.}
\end{align*}
\]

Solve iteratively, starting by $\emptyset$ for all nonterminals.
Set Equations for **Follow** Sets

Follow sets solve a collection of set constraints.

Constraints derived from right-hand sides of grammar productions

For $X \in N$, consider all productions $Y \rightarrow \alpha X \beta$ where $X$ occurs on the right.

- $\text{FIRST} (\beta) \subseteq \text{Follow}(X)$
- If $\text{Nullable} (\beta)$ or $\beta = \varepsilon$: $\text{Follow}(Y) \subseteq \text{Follow}(X)$

If $X$ occurs several times, each occurrence contributes separate equations.

\[
S' \rightarrow SS \quad \ldots \quad \text{FIRST}(\$) = \{\$\} \subseteq \text{Follow}(S) \\
S \rightarrow AB \quad \ldots \quad \text{FIRST}(B) = \{b\} \subseteq \text{Follow}(A) \\
\quad \quad \text{Follow}(S) \subseteq \text{Follow}(A) \quad \text{(B nullable)} \\
\quad \quad \text{Follow}(S) \subseteq \text{Follow}(B) \\
A \rightarrow aAb \quad \ldots \quad \text{FIRST}(b) = \{b\} \subseteq \text{Follow}(A) \\
B \rightarrow bB \quad \ldots \quad \text{Follow}(B) \subseteq \text{Follow}(B) \\
\quad \quad A \rightarrow \varepsilon \text{ and } B \rightarrow \varepsilon \text{ do not contribute.}
\]

Example:

\[
\begin{align*}
\text{Solve iteratively, starting by } \emptyset \text{ for all nonterminals.} \\
\text{Follow}(S) &= \text{Follow}(B) = \{\$\} \\
\quad \text{Follow}(A) &= \{\$, \ b\}
\end{align*}
\]
Putting it Together: Look-ahead Sets and LL(1)

After computing NULLABLE and FIRST for all right-hand sides and FOLLOW for all nonterminals, a parser can be constructed.

**Definition (Look-ahead Sets of a Grammar)**

For every production \( X \rightarrow \alpha \) of a context-free grammar \( G \), we define the Look-ahead set of the production as:

\[
la(X \rightarrow \alpha) = \begin{cases} 
  \text{FIRST}(\alpha) \cup \text{FOLLOW}(X) & \text{, if NULLABLE}(\alpha) \\
  \text{FIRST}(\alpha) & \text{, otherwise}
\end{cases}
\]
Putting it Together: Look-ahead Sets and LL(1)

After computing Nullable and First for all right-hand sides and Follow for all nonterminals, a parser can be constructed.

Definition (Look-ahead Sets of a Grammar)

For every production \( X \rightarrow \alpha \) of a context-free grammar \( G \), we define the Look-ahead set of the production as:

\[
la(X \rightarrow \alpha) = \begin{cases} 
\text{First}(\alpha) \cup \text{Follow}(X), & \text{if Nullable}(\alpha) \\
\text{First}(\alpha), & \text{otherwise}
\end{cases}
\]

LL(1) Grammars

If for each nonterminal \( X \in N \) in grammar \( G \), all productions of \( X \) have disjoint look-ahead sets, the grammar \( G \) is LL(1) (left-to-right, left-most, look-ahead 1).

For an LL(1) grammar, a parser can be constructed which constructs a left-most derivation for valid input with one token look-ahead (predicting the next production from look-ahead).
Recursive Descent with Look-Ahead

The grammar in our example is LL(1):

\[ G' : S \rightarrow AB \]
\[ A \rightarrow aAb \]
\[ A \rightarrow \varepsilon \]
\[ B \rightarrow bB \]
\[ B \rightarrow \varepsilon \]

\[ \text{la}(S \rightarrow AB) = \text{FIRST}(AB) \cup \text{FOLLOW}(S) = \{a, b, \$\} \]
\[ \text{la}(A \rightarrow aAb) = \text{FIRST}(aAB) = \{a\} \]
\[ \text{la}(A \rightarrow \varepsilon) = \text{FIRST}(\varepsilon) \cup \text{FOLLOW}(A) = \{b, \$\} \]
\[ \text{la}(B \rightarrow bB) = \text{FIRST}(bB) = \{b\} \]
\[ \text{la}(B \rightarrow \varepsilon) = \text{FIRST}(\varepsilon) \cup \text{FOLLOW}(B) = \{\$\} \]

```plaintext
fun parseS ()
  = if next = #"a" orelse next = #"b" orelse next = EOF
     then parseA (); parseB (); match EOF else error

and parseA () (* choose by look-ahead *)
  = if next = #"a" then match #"a"; parseA (); match #"b"
    else if next = #"b" orelse next = EOF then ()
    else error

and parseB () = if next = #"b" then match #"b"; parseB ()
    else if next = EOF then ()
    else error
```
Table-Driven LL(1) Parsing

• Stack, contains unprocessed part of production, initially $S$.
• Parser Table: action to take, depends on stack and next input
• Actions (pop consumes input, derivation only reads it)
  Pop: remove terminal from stack (on matching input).
  Derive: pop nonterminal from stack, push right-hand side (in table).
• Accept input when stack empty at end of input.

<table>
<thead>
<tr>
<th>Stack:</th>
<th>Look-ahead/Input:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
</tr>
<tr>
<td>$</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>AB, 1</td>
</tr>
<tr>
<td>A</td>
<td>aAb, 2</td>
</tr>
<tr>
<td>B</td>
<td>error</td>
</tr>
<tr>
<td>a</td>
<td>pop</td>
</tr>
<tr>
<td>b</td>
<td>error</td>
</tr>
<tr>
<td>$</td>
<td>error</td>
</tr>
</tbody>
</table>
Table-Driven LL(1) Parsing

- Stack, contains unprocessed part of production, initially $S$.
- Parser Table: action to take, depends on stack and next input
- Actions (pop consumes input, derivation only reads it)
  - Pop: remove terminal from stack (on matching input).
  - Derive: pop nonterminal from stack, push right-hand side (in table).
- Accept input when stack empty at end of input.

Example run (input $aabbb$):

<table>
<thead>
<tr>
<th>Stack:</th>
<th>Look-ahead/Input:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$a$</td>
</tr>
<tr>
<td>$A$</td>
<td>$aAb$, 2</td>
</tr>
<tr>
<td>$B$</td>
<td>error</td>
</tr>
<tr>
<td>$a$</td>
<td>$pop$</td>
</tr>
<tr>
<td>$b$</td>
<td>error</td>
</tr>
<tr>
<td>$$</td>
<td>error</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input</th>
<th>Stack</th>
<th>Action</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>aabbb$</td>
<td>$S$</td>
<td>derive</td>
<td>$\varepsilon$</td>
</tr>
<tr>
<td>aabbb$</td>
<td>$AB$</td>
<td>derive</td>
<td>1</td>
</tr>
<tr>
<td>aabbb$</td>
<td>$aAbB$</td>
<td>pop</td>
<td>12</td>
</tr>
<tr>
<td>aabbb$</td>
<td>$AbB$</td>
<td>derive</td>
<td>12</td>
</tr>
<tr>
<td>aabbb$</td>
<td>$aAbB$</td>
<td>pop</td>
<td>122</td>
</tr>
<tr>
<td>aabbb$</td>
<td>$AbbB$</td>
<td>derive</td>
<td>122</td>
</tr>
<tr>
<td>$bbB$</td>
<td>$bbB$</td>
<td>pop</td>
<td>1223</td>
</tr>
<tr>
<td>$bbB$</td>
<td>$bB$</td>
<td>pop</td>
<td>1223</td>
</tr>
<tr>
<td>$B$</td>
<td>$B$</td>
<td>derive</td>
<td>1223</td>
</tr>
<tr>
<td>$BB$</td>
<td>$BB$</td>
<td>pop</td>
<td>12234</td>
</tr>
<tr>
<td>$B$</td>
<td>$B$</td>
<td>derive</td>
<td>12234</td>
</tr>
<tr>
<td>$B$</td>
<td>$B$</td>
<td>accept</td>
<td>122345</td>
</tr>
</tbody>
</table>
Eliminating Left-Recursion and Left-Factorisation

Problems that often occur when constructing LL(1) parsers:

- Identical prefixes: Productions $X \rightarrow \alpha \beta \mid \alpha \gamma$.
  Requires look-ahead longer than the common prefix $\alpha$.

  Solution: Left-Factorisation, introducing new productions $X \rightarrow \alpha Y$ and $Y \rightarrow \beta \mid \gamma$. 
Eliminating Left-Recursion and Left-Factorisation

Problems that often occur when constructing LL(1) parsers:

- **Identical prefixes:** Productions $X \rightarrow \alpha \beta | \alpha \gamma$. Requires look-ahead longer than the common prefix $\alpha$.
  
  **Solution:** Left-Factorisation, introducing new productions $X \rightarrow \alpha Y$ and $Y \rightarrow \beta | \gamma$.

- **Left-Recursion:** a nonterminal reproducing itself on the left.
  
  **Direct:** production $X \rightarrow X \alpha | \beta$, or indirect: $X \Rightarrow^* X \alpha$. Cannot be analysed with finite look-ahead!
  
  **Solution:** new (nullable) nonterminals and swapped recursion. $X \rightarrow X \alpha | \beta$, thus $\text{FIRST}(X) \subset \text{FIRST}(X \alpha) \cup \text{FIRST}(\beta)$

  Also works in case of multiple left-recursive productions.

  For indirect recursion: first transform into direct recursion.
Contents

1 Context-Free Grammars and Languages

2 Top-Down Parsing, LL(1)
   Recursive Parsing Functions (Recursive-descent)
   First- and Follow-Sets
   Look-Ahead Sets and LL(1) Parsing

3 Bottom-Up Parsing, SLR
   Parser Generator Yacc
   Shift-Reduce Parsing

4 Precedence and Associativity
Bottom-Up Parsing


$$G'' : S' \rightarrow S\,\$ \quad (0)$$

$$S \rightarrow AB \quad (1)$$

$$A \rightarrow aAb \quad (2)$$

$$A \rightarrow \varepsilon \quad (3)$$

$$B \rightarrow bB \quad (4)$$

$$B \rightarrow \varepsilon \quad (5)$$

Right-most derivation: 1 4 5 2 2 3
Bottom-Up Parsing: Idea for a Machine

\[
\begin{align*}
G'' : S' & \rightarrow S$ \quad (0) \\
S & \rightarrow AB \quad (1) \\
A & \rightarrow aAb \quad (2) \\
A & \rightarrow \varepsilon \quad (3) \\
B & \rightarrow bB \quad (4) \\
B & \rightarrow \varepsilon \quad (5)
\end{align*}
\]
Bottom-Up Parsing: Idea for a Machine

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>ε</td>
<td>aabbb$</td>
<td>shift</td>
</tr>
<tr>
<td>a</td>
<td>abbb$</td>
<td>shift</td>
</tr>
<tr>
<td>aa_</td>
<td>bbb$</td>
<td>reduce 3</td>
</tr>
<tr>
<td>aaA</td>
<td>bbb$</td>
<td>shift</td>
</tr>
<tr>
<td>aaAb</td>
<td>bb$</td>
<td>reduce 2</td>
</tr>
<tr>
<td>aA</td>
<td>bb$</td>
<td>shift</td>
</tr>
<tr>
<td>aAb</td>
<td>b$</td>
<td>reduce 2</td>
</tr>
<tr>
<td>A</td>
<td>b$</td>
<td>shift</td>
</tr>
<tr>
<td>Ab_</td>
<td>$</td>
<td>reduce 5</td>
</tr>
<tr>
<td>AbB</td>
<td>$</td>
<td>reduce 4</td>
</tr>
<tr>
<td>AB</td>
<td>$</td>
<td>reduce 1</td>
</tr>
<tr>
<td>S</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>

\[ G'' : S' \rightarrow S$ (0) \]
\[ S \rightarrow AB (1) \]
\[ A \rightarrow aAb (2) \]
\[ A \rightarrow \epsilon (3) \]
\[ B \rightarrow bB (4) \]
\[ B \rightarrow \epsilon (5) \]
Bottom-Up Parsing: Idea for a Machine

Questions:
- When to accept (solved: separate start production)
- When to shift, when to reduce? Especially $R \rightarrow \varepsilon$. 

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$</td>
<td>aabbb$</td>
<td>shift</td>
</tr>
<tr>
<td>a</td>
<td>abbb$</td>
<td>shift</td>
</tr>
<tr>
<td>aa_</td>
<td>bbb$</td>
<td>reduce 3</td>
</tr>
<tr>
<td>aaA</td>
<td>bbb$</td>
<td>shift</td>
</tr>
<tr>
<td>aaA_b</td>
<td>bb$</td>
<td>reduce 2</td>
</tr>
<tr>
<td>aA</td>
<td>bb$</td>
<td>shift</td>
</tr>
<tr>
<td>aA_b</td>
<td>b$</td>
<td>reduce 2</td>
</tr>
<tr>
<td>A</td>
<td>b$</td>
<td>shift</td>
</tr>
<tr>
<td>A_b_</td>
<td>$</td>
<td>reduce 5</td>
</tr>
<tr>
<td>A_bB</td>
<td>$</td>
<td>reduce 4</td>
</tr>
<tr>
<td>AB</td>
<td>$</td>
<td>reduce 1</td>
</tr>
<tr>
<td>S</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>

\[ G'' : S' \rightarrow S\$ \ (0) \]
\[ S \rightarrow AB \ (1) \]
\[ A \rightarrow aAb \ (2) \]
\[ A \rightarrow \varepsilon \ (3) \]
\[ B \rightarrow bB \ (4) \]
\[ B \rightarrow \varepsilon \ (5) \]
mosmlyac: Yet Another Compiler Compiler in MosML

- Generates bottom-up parser from a grammar specification
- Grammar specification also includes token datatype declaration and other declarations.

Demo mosmlyac

Tradition: Lex and Yacc (GNU: flex and bison)

- Parser generators usually use LALR(1) Parsing\(^2\).
- We use SLR parsing instead:
  Simple \underline{Left-to-right} Right-most analysis with look-ahead 1.

\(^2\)More information about LALR(1) and LR(1) parsing can be found in the Red-Dragon book.
Constructing an SLR-Parser: Items

Each production in the grammar leads to a number of items:

### Shift Items and Reduce Items of a Production

Let $X \rightarrow \alpha$ be a production in a grammar. The production implies:

- **Shift items**: $[X \rightarrow \alpha_1 \cdot \alpha_2]$ for every decomposition $\alpha = \alpha_1 \alpha_2$ (including $\alpha_1 = \varepsilon$ and $\alpha_2 = \varepsilon$);
- **One reduce item**: $[X \rightarrow \alpha \cdot]$ per production.

Items give information about the next action:

- Either to **shift** an item to the stack and read input
- or to **reduce** the top of stack (a production’s right-hand side).
Constructing an SLR-Parser: Items

Each production in the grammar leads to a number of items:

### Shift Items and Reduce Items of a Production

Let $X \rightarrow \alpha$ be a production in a grammar. The production implies:

- **Shift items**: $[X \rightarrow \alpha_1 \bullet \alpha_2]$ for every decomposition $\alpha = \alpha_1 \alpha_2$ (including $\alpha_1 = \varepsilon$ and $\alpha_2 = \varepsilon$);

- **One reduce item**: $[X \rightarrow \alpha \bullet]$ per production.

Items give information about the next action:

- Either to **shift** an item to the stack and read input
- or to **reduce** the top of stack (a production’s right-hand side).
- Stack of the parser will contain **items**, not grammar symbols.
- Therefore, no need to read into the stack for reductions.
Constructing an SLR Parser: Production DFAs

Each production $X \rightarrow \alpha$ suggests a DFA with items as states, and doing the following transitions:

- From $[X \rightarrow \alpha \bullet a\beta]$ to $[X \rightarrow \alpha a \bullet \beta]$ for input tokens $a$. These will be \textit{Shift} action that read input later.
- From $[X \rightarrow \alpha \bullet Y\beta]$ to $[X \rightarrow \alpha Y \bullet \beta]$ for nonterminals $Y$. These will be \textit{Go} actions later, without consuming input.

All items are states, start state is the first item $[X \rightarrow \bullet \alpha]$. 

\[
\begin{align*}
A &\rightarrow aAb \\
A &\rightarrow \varepsilon
\end{align*}
\]
Constructing an SLR Parser: Production DFAs

Each production \( X \rightarrow \alpha \) suggests a DFA with items as states, and doing the following transitions:

- From \([X \rightarrow \alpha \bullet a\beta]\) to \([X \rightarrow \alpha a \bullet \beta]\) for input tokens \(a\). These will be Shift action that read input later.
- From \([X \rightarrow \alpha \bullet Y\beta]\) to \([X \rightarrow \alpha Y \bullet \beta]\) for nonterminals \(Y\). These will be Go actions later, without consuming input.

All items are states, start state is the first item \([X \rightarrow \bullet \alpha]\).

While traversing the DFA: items pushed on the stack.
When reaching a reduce item: use stack to back-track (later).
SLR Parser Construction: Example

Productions

\[ S \rightarrow AB \]
\[ B \rightarrow \varepsilon \]
\[ B \rightarrow bB \]
\[ A \rightarrow \varepsilon \]
\[ A \rightarrow aAb \]

NFA

\[ A \xrightarrow{A} B \xrightarrow{B} C \]
\[ \text{Extra} \]
\[ \text{Extra} \]
\[ \text{Extra} \]
Extra $\varepsilon$-transitions connect the DFAs for all productions:

- From $[X \rightarrow \alpha \bullet Y \beta]$ to $[Y \rightarrow \bullet \gamma]$ for all productions $Y \rightarrow \gamma$. 
SLR Parser Construction: Example

Productions

\[ S \rightarrow AB \]
\[ B \rightarrow \varepsilon \]
\[ B \rightarrow bB \]
\[ A \rightarrow \varepsilon \]
\[ A \rightarrow aAb \]

NFA

Extra \( \varepsilon \)-transitions connect the DFAs for all productions:

- From \([X \rightarrow \alpha \bullet Y \beta]\) to \([Y \rightarrow \bullet \gamma]\) for all productions \(Y \rightarrow \gamma\)

When in front of a nonterminal \(Y\) in a production DFA:
try to run the DFA for one of the right-hand sides of \(Y\) productions.
SLR Parser Construction: Example(2)

Productions

- $S \rightarrow AB$
- $B \rightarrow \varepsilon$
- $B \rightarrow bB$
- $A \rightarrow \varepsilon$
- $A \rightarrow aAb$

Next step: Subset construction of a combined DFA.
SLR Parser Construction: Example(2)

Productions

\[ S \rightarrow AB \]
\[ B \rightarrow \varepsilon \]
\[ B \rightarrow bB \]
\[ A \rightarrow \varepsilon \]
\[ A \rightarrow aAb \]

NFA

Next step: Subset construction of a combined DFA.

Blackboard...
SLR Parsing: Internal DFA and Stack

- Transitions: Shift actions (terminals) and Go actions (nonterminals).

- SLR Parse Table: actions indexed by symbols and DFA states
  
  **Shift n** Terminal transition: push state $n$ on stack, consume input
  
  **Go n** Nonterminal transition: push state $n$ on stack, (no input read)
SLR Parsing: Internal DFA and Stack

- Transitions: Shift actions (terminals) and Go actions (nonterminals).
- Final DFA states: contain reduce items. Reduce actions need to be added to the transition table.
- Reduce action: remove items from stack corresponding to right-hand side, then do a Go action with the left-hand side.

- SLR Parse Table: actions indexed by symbols and DFA states
  - Shift $n$: Terminal transition: push state $n$ on stack, consume input
  - Go $n$: Nonterminal transition: push state $n$ on stack, (no input read)
SLR Parsing: Internal DFA and Stack

- Transitions: Shift actions (terminals) and Go actions (nonterminals).
- Final DFA states: contain reduce items. Reduce actions need to be added to the transition table.
- Reduce action: remove items from stack corresponding to right-hand side, then do a Go action with the left-hand side.

- SLR Parse Table: actions indexed by symbols and DFA states
  Shift \( n \)  Terminal transition: push state \( n \) on stack, consume input
  Go \( n \)  Nonterminal transition: push state \( n \) on stack, (no input read)
  Reduce \( p \)  Reduce with production \( p \)
SLR Parsing: Internal DFA and Stack

- Transitions: Shift actions (terminals) and Go actions (nonterminals).
- Final DFA states: contain reduce items. Reduce actions need to be added to the transition table.
- Reduce action: remove items from stack corresponding to right-hand side, then do a Go action with the left-hand side.

- SLR Parse Table: actions indexed by symbols and DFA states
  - Shift $n$ Terminal transition: push state $n$ on stack, consume input
  - Go $n$ Nonterminal transition: push state $n$ on stack, (no input read)
  - Reduce $p$ Reduce with production $p$
  - Accept Parsing has succeeded (reduce with production 0).
SLR Parser Construction: Conflicts

- After constructing a DFA: shift and go actions.
- Next: add reduce actions for states containing reduce items

**SLR Parser Conflicts**

Subset construction of the DFA might join conflicting items in one DFA state. We call these conflicts

- **Shift-Reduce conflict**, if a DFA state contains both shift and reduce items.
  Typically, productions to $\varepsilon$ generate these conflicts.
- **Reduce-Reduce conflict**, if a DFA state contains reduce items for two different productions.
SLR Parser Construction: Conflicts

- After constructing a DFA: **shift and go** actions.
- Next: add **reduce** actions for states containing reduce items.

### SLR Parser Conflicts

Subset construction of the DFA might join conflicting items in one DFA state. We call these conflicts:

- **Shift-Reduce conflict**, if a DFA state contains both shift and reduce items.
  Typically, productions to $\varepsilon$ generate these conflicts.
- **Reduce-Reduce conflict**, if a DFA state contains reduce items for two different productions.

In SLR parsing: **FOLLOW** sets of nonterminals are compared to the look-ahead to resolve conflicts.
SLR Parser Construction: The Parser Table

A

\[ G'': \ S' \rightarrow S\$ \ (0) \]
\[ S \rightarrow AB \ (1) \]
\[ A \rightarrow aAb \ (2) \]
\[ A \rightarrow \varepsilon \ (3) \]
\[ B \rightarrow bB \ (4) \]
\[ B \rightarrow \varepsilon \ (5) \]

Parser Table:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Follow Sets of Nonterminals:

\[ \text{Follow}(S) = \{ \$ \} \]
\[ \text{Follow}(A) = \{ b, \$ \} \]
\[ \text{Follow}(B) = \{ \$ \} \]
SLR Parser Construction: The Parser Table

\[ G'' : S' \rightarrow S$ \ (0) \]
\[ S \rightarrow AB \ (1) \]
\[ A \rightarrow aAb \ (2) \]
\[ A \rightarrow \varepsilon \ (3) \]
\[ B \rightarrow bB \ (4) \]
\[ B \rightarrow \varepsilon \ (5) \]

- Parser Table:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>$</th>
<th>S</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td></td>
<td></td>
<td>Go 1</td>
<td></td>
<td>Go 3</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>Go 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>Go 7</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td>Go 8</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SLR Parser Construction: The Parser Table

**Parser Table:**

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>$</th>
<th>S</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td></td>
<td></td>
<td>Go 1</td>
<td>Go 3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Go 4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Go 7</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td>Go 8</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Follow* Sets of Nonterminals:

- **Follow**($S$) = \{$\}$
- **Follow**($A$) = \{b, $\}$
- **Follow**($B$) = \{$\}$
SLR Parser Construction: The Parser Table

- **Parser Table:**

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>$</th>
<th>S</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td></td>
<td>red.3</td>
<td>red.3</td>
<td>Go 1</td>
<td>Go 3</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>acc.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>red.3</td>
<td>red.3</td>
<td>Go 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>6</td>
<td>red.5</td>
<td></td>
<td></td>
<td>Go 7</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>red.2</td>
<td>red.2</td>
<td></td>
<td>Go 8</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>6</td>
<td>red.5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>red.1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>red.4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Follow Sets of Nonterminals:**

\[ G'' : S' \rightarrow S\$ (0) \]
\[ S \rightarrow AB (1) \]
\[ A \rightarrow aAb (2) \]
\[ A \rightarrow \varepsilon (3) \]
\[ B \rightarrow bB (4) \]
\[ B \rightarrow \varepsilon (5) \]

\[ \text{FOLLOW}(S) = \{ \$ \} \]
\[ \text{FOLLOW}(A) = \{ b, \$ \} \]
\[ \text{FOLLOW}(B) = \{ \$ \} \]
Table-Driven SLR Parsing

- **Stack** contains DFA states, initially start state 0.
- **SLR Parse Table**: actions and transitions
  
  **Shift**: do a transition consuming input, push new state on stack
  
  **Reduce**: pop length of right-hand-side from stack, then go to a new state with left-hand side non-terminal, push new state on stack
  
- Accept input when accept state reached at end of input.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>S</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>red.3</td>
<td>red.3</td>
<td>Go 1</td>
<td>Go 3</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>acc.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>red.3</td>
<td>red.3</td>
<td>Go 4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>red.5</td>
<td></td>
<td>Go 7</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>red.2</td>
<td>red.2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>red.5</td>
<td></td>
<td>Go 8</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>red.1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>red.4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table-Driven SLR Parsing

- Stack contains DFA states, initially start state 0.
- SLR Parse Table: actions and transitions
  
  **Shift:** do a transition consuming input, push new state on stack
  
  **Reduce:** pop length of right-hand-side from stack, then go to a new state with left-hand side non-terminal, push new state on stack

- Accept input when accept state reached at end of input.

```
<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>$</th>
<th>S</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>red.3</td>
<td>red.3</td>
<td>Go 1</td>
<td>Go 3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>acc.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>red.3</td>
<td>red.3</td>
<td>Go 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>red.5</td>
<td></td>
<td>Go 7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>red.2</td>
<td>red.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>red.5</td>
<td></td>
<td>Go 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>red.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>red.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Example run \((aabbb)\):

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>aabbb$</td>
<td>shift</td>
</tr>
<tr>
<td>02</td>
<td>abbb$</td>
<td>shift</td>
</tr>
<tr>
<td>022_</td>
<td>bbb$</td>
<td>reduce 3</td>
</tr>
<tr>
<td>0224</td>
<td>bbb$</td>
<td>shift</td>
</tr>
<tr>
<td>02245</td>
<td>bb$</td>
<td>reduce 2</td>
</tr>
<tr>
<td>024</td>
<td>bb$</td>
<td>shift</td>
</tr>
<tr>
<td>0245</td>
<td>b$</td>
<td>reduce 2</td>
</tr>
<tr>
<td>03</td>
<td>b$</td>
<td>shift</td>
</tr>
<tr>
<td>036_</td>
<td>$</td>
<td>reduce 5</td>
</tr>
<tr>
<td>0368</td>
<td>$</td>
<td>reduce 4</td>
</tr>
<tr>
<td>037</td>
<td>$</td>
<td>reduce 1</td>
</tr>
<tr>
<td>01</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>
Table-Driven SLR Parsing

- Stack contains DFA states, initially start state 0.
- SLR Parse Table: actions and transitions
  - **Shift**: do a transition consuming input, push new state on stack
  - **Reduce**: pop length of right-hand-side from stack, then go to a new state with left-hand side non-terminal, push new state on stack
- Accept input when accept state reached at end of input.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>$</th>
<th>S</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>red.3</td>
<td>red.3</td>
<td>Go 1</td>
<td>Go 3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>acc.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>red.3</td>
<td>red.3</td>
<td>Go 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>red.5</td>
<td></td>
<td>Go 7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>red.2</td>
<td>red.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>red.5</td>
<td></td>
<td>Go 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>red.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>red.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example run \((aabb$b)\):

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>aabb$b</td>
<td>shift</td>
</tr>
<tr>
<td>02</td>
<td>abbb$</td>
<td>shift</td>
</tr>
<tr>
<td>022</td>
<td>bbb$</td>
<td>reduce 3</td>
</tr>
<tr>
<td>0224</td>
<td>bbb$</td>
<td>shift</td>
</tr>
<tr>
<td>02245</td>
<td>bb$</td>
<td>reduce 2</td>
</tr>
<tr>
<td>024</td>
<td>bb$</td>
<td>reduce 2</td>
</tr>
<tr>
<td>0245</td>
<td>b$</td>
<td>reduce 2</td>
</tr>
<tr>
<td>03</td>
<td>b$</td>
<td>shift</td>
</tr>
<tr>
<td>036</td>
<td>$</td>
<td>reduce 5</td>
</tr>
<tr>
<td>0368</td>
<td>$</td>
<td>reduce 4</td>
</tr>
<tr>
<td>037</td>
<td>$</td>
<td>reduce 1</td>
</tr>
<tr>
<td>01</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>
Contents

1 Context-Free Grammars and Languages

2 Top-Down Parsing, LL(1)
   Recursive Parsing Functions (Recursive-descent)
   First- and Follow-Sets
   Look-Ahead Sets and LL(1) Parsing

3 Bottom-Up Parsing, SLR
   Parser Generator Yacc
   Shift-Reduce Parsing

4 Precedence and Associativity
Ambiguity, Precedence and Associativity

Arithmetic Expressions:

\[ E \rightarrow E + E \mid E - E \]
\[ E \rightarrow E \times E \mid E / E \]
\[ E \rightarrow a \mid (E) \]

- In many cases, grammars are rewritten to remove ambiguity.
- Sometimes, ambiguity is resolved by changes in the parser.

- In both cases: Precedence and associativity guide decisions.
Ambiguity, Precedence and Associativity

Arithmetic Expressions:

\[ E \rightarrow E + E \mid E - E \]
\[ E \rightarrow E \times E \mid E / E \]
\[ E \rightarrow a \mid (E) \]

- In many cases, grammars are rewritten to remove ambiguity.
- Sometimes, ambiguity is resolved by changes in the parser.

- In both cases: **Precedence and associativity** guide decisions.

Problems with this grammar:

1. Ambiguous derivation of \( a - a \times a \).
   Want precedence of \( \times \) over \( + \), \( a + (a \cdot a) \).

2. Ambiguous derivation of \( a - a - a \).
   Want a left-associative interpretation, \( (a - a) - a \).
Operator Precedence in the Grammar

- Introduce precedence levels to get operator priorities
- New Grammar: own nonterminal for each level
- Here: 2 levels, mathematical interpretation:
  \[ a - a \cdot a = a - (a \cdot a) \]
  Precedence of \(*\) and \(/\) over \(+\) and \(-\).
  More precedence levels could be added (exponentiation).

\[
E \rightarrow E + E \mid E - E \\
E \rightarrow E \cdot E \mid E / E \\
E \rightarrow a \mid (E)
\]
Operator Precedence in the Grammar

- Introduce precedence levels to get operator priorities
- New Grammar: own nonterminal for each level
- Here: 2 levels, mathematical interpretation:
  
  \[ a - a \cdot a = a - (a \cdot a) \]

  Precedence of * and / over + and -.

  More precedence levels could be added (exponentiation).

\[
\begin{align*}
E & \rightarrow E + E \mid E - E \\
E & \rightarrow E \cdot E \mid E / E \\
E & \rightarrow a \mid (E)
\end{align*}
\]

\[
\begin{align*}
E & \rightarrow E + E \mid E - E \mid T \\
T & \rightarrow T \cdot T \mid T / T \\
T & \rightarrow a \mid (E)
\end{align*}
\]
Operator Precedence in the Grammar

- Introduce precedence levels to get operator priorities
- New Grammar: own nonterminal for each level
- Here: 2 levels, mathematical interpretation: 
  \[ a - a \cdot a = a - (a \cdot a) \]
  Precedence of \(*\) and \(/\) over + and -.
  More precedence levels could be added (exponentiation).

\[
\begin{align*}
E & \rightarrow E + E \mid E - E \\
E & \rightarrow E \ast E \mid E / E \\
E & \rightarrow a \mid (E)
\end{align*}
\]
About Operator Associativity

Definition (Operator Associativity)

A binary operator $\oplus$ is called

- **left-associative**, if the expression $a \oplus b \oplus c$ should be evaluated from left to right, as $(a \oplus b) \oplus c$.
- **right-associative**, if the expression $a \oplus b \oplus c$ should be evaluated from right to left, as $a \oplus (b \oplus c)$.
- **non-associative**, if expressions $a \oplus b \oplus c$ are disallowed, (and **associative**, if both directions lead to the same result).
About Operator Associativity

Definition (Operator Associativity)

A binary operator $\oplus$ is called

- **left-associative**, if the expression $a \oplus b \oplus c$ should be evaluated from left to right, as $(a \oplus b) \oplus c$.

- **right-associative**, if the expression $a \oplus b \oplus c$ should be evaluated from right to left, as $a \oplus (b \oplus c)$.

- **non-associative**, if expressions $a \oplus b \oplus c$ are disallowed, (and **associative**, if both directions lead to the same result).

Examples:

- Arithmetic operators like $-$ and $/$: left-associative.
# About Operator Associativity

## Definition (Operator Associativity)

A binary operator $\oplus$ is called

- **left-associative**, if the expression $a \oplus b \oplus c$ should be evaluated from left to right, as $(a \oplus b) \oplus c$.
- **right-associative**, if the expression $a \oplus b \oplus c$ should be evaluated from right to left, as $a \oplus (b \oplus c)$.
- **non-associative**, if expressions $a \oplus b \oplus c$ are disallowed, (and **associative**, if both directions lead to the same result).

## Examples:

- Arithmetic operators like - and /: left-associative.
- List constructors in functional languages: right-associative.
- Function arrows in types: right-associative.
About Operator Associativity

Definition (Operator Associativity)

A binary operator $\oplus$ is called
- **left-associative**, if the expression $a \oplus b \oplus c$ should be evaluated from left to right, as $(a \oplus b) \oplus c$.
- **right-associative**, if the expression $a \oplus b \oplus c$ should be evaluated from right to left, as $a \oplus (b \oplus c)$.
- **non-associative**, if expressions $a \oplus b \oplus c$ are disallowed, (and **associative**, if both directions lead to the same result).

Examples:
- Arithmetic operators like $-$ and $/$: left-associative.
- List constructors in functional languages: right-associative.
- Function arrows in types: right-associative.
- ‘less-than’ ($<$) in C:

```c
if (3 < 2 < 1) { printf(stdout, "Awesome!\n"); }
```
About Operator Associativity

Definition (Operator Associativity)

A binary operator $\oplus$ is called

- **left-associative**, if the expression $a \oplus b \oplus c$ should be evaluated from left to right, as $(a \oplus b) \oplus c$.
- **right-associative**, if the expression $a \oplus b \oplus c$ should be evaluated from right to left, as $a \oplus (b \oplus c)$.
- **non-associative**, if expressions $a \oplus b \oplus c$ are disallowed, (and **associative**, if both directions lead to the same result).

Examples:

- Arithmetic operators like `-` and `/`: left-associative.
- List constructors in functional languages: right-associative.
- Function arrows in types: right-associative.
- ‘less-than’ (`<`) in C: **left-associative**

```c
if (3 < 2 < 1) { fprintf(stdout, "Awesome!\n"); }
```
Establishing the Intended Associativity

- limit recursion to the intended side
- When operators are indeed associative, use same associativity as comparable operators.
- Cannot mix left- and right-associative operators at same precedence level.

\[
E \rightarrow E + E \mid E - E \mid T \\
T \rightarrow T \ast T \mid T / T \\
T \rightarrow a \mid (E)
\]
Establishing the Intended Associativity

- limit recursion to the intended side
- When operators are indeed associative, use same associativity as comparable operators.
- Cannot mix left- and right-associative operators at same precedence level.

\[
\begin{align*}
E & \rightarrow E + E \mid E - E \mid T \\
T & \rightarrow T \ast T \mid T/T \\
T & \rightarrow a \mid (E)
\end{align*}
\]
Precedence and Associativity in SLR Parse Tables

Precedence and ambiguity usually materialise as shift-reduce conflicts in SLR parsers.

\[ E \rightarrow E \ast E \mid E + E \mid \ldots \mid a \mid (E) \]

\[ [E \rightarrow E + E \bullet], \]
\[ [E \rightarrow E \bullet + E], \]
\[ [E \rightarrow E \bullet \ast E] \]

\[ \cdots \]

Shift-Reduce conflict!

Instead of rewriting the grammar, resolve conflicts by targeted changes to parser table.
Precedence and Associativity in SLR Parse Tables

Precedence and ambiguity usually materialise as shift-reduce conflicts in SLR parsers.

\[
E \rightarrow E \ast E \mid E + E \mid ... \\
\mid a \mid (E) \\
\Rightarrow [E \rightarrow E + E \bullet], \\
[E \rightarrow E \bullet + E], \\
[E \rightarrow E \bullet \ast E] \\
\ldots
\]

Shift-Reduce conflict!

Instead of rewriting the grammar, resolve conflicts by targeted changes to parser table.

• if operator symbol with higher precedence follows: Shift
• if operator should be right-associative: Shift
• if symbol of lower precedence or left-associative: Reduce
Example: Resolving Precedence and Ambiguity

Regular expressions:

\[ R \rightarrow R' \mid 'R \]
\[ R \rightarrow RR \]
\[ R \rightarrow R'^* \]
\[ R \rightarrow \text{char} \mid (R) \]

1. **Precedence:** star, sequence, alternative.
   
   \[ a \mid b a^* \text{ is } a \mid (b(a^*)) \].

2. **Left-associative derivations:**
   
   \[ \alpha \mid \beta \mid \gamma \text{ is } (\alpha \mid \beta) \mid \gamma \].

New grammar:

Your
grammars
here
Example: Resolving Precedence and Ambiguity

Regular expressions:

\[ R \rightarrow R' \mid 'R \]
\[ R \rightarrow RR \]
\[ R \rightarrow R'^* \]
\[ R \rightarrow \text{char} \mid (R) \]

1. Precedence: star, sequence, alternative.
   \[ a \mid b a^* \text{ is } a \mid (b(a^*)) \].

2. Left-associative derivations:
   \[ \alpha \mid \beta \mid \gamma \text{ is } (\alpha \mid \beta) \mid \gamma \].

New grammar:

\[ R \rightarrow R' \mid 'R2 \mid R2 \]
\[ R2 \rightarrow R2R3 \mid R3 \]
\[ R3 \rightarrow R4'^* \mid R4 \]
\[ R4 \rightarrow \text{char} \mid (R) \]
Example: Resolving Precedence and Ambiguity

Regular expressions:

\[
R \rightarrow R' | 'R
\]
\[
R \rightarrow RR
\]
\[
R \rightarrow R^*
\]
\[
R \rightarrow \text{char} | (R)
\]

New grammar:

\[
R \rightarrow R' | 'R2 | R2
\]
\[
R2 \rightarrow R2R3 | R3
\]
\[
R3 \rightarrow R4^* | R4
\]
\[
R4 \rightarrow \text{char} | (R)
\]

1. **Precedence:** star, sequence, alternative.
   
   \(a \mid b a^*\) is \((a|b(a^*))\).

2. **Left-associative derivations:**
   
   \(\alpha \mid \beta \mid \gamma\) is \((\alpha|\beta)|\gamma\).

Precedence/Associativity declarations:

```
%token BAR STAR LPAREN RPAREN ...
...
%left BAR /* lowest precedence */
%nonassoc CHAR LPAREN
%left seq /* pseudo-token for sequence */
%nonassoc STAR /* highest precedence */
...
R : R BAR R { ... }
| R R %prec seq { ... }
| R STAR { ... }
| CHAR { ... }
| LPAREN R RPAREN { ... }
...
```

Full example:

Mosmlyac Demo (regular expressions)
Example: Resolving Precedence and Ambiguity

Regular expressions:

1. Precedence: star, sequence, alternative.
   
   \( a \mid b \ a^* \) is \( a|(b(a^*)) \).

2. Left-associative derivations:
   
   \( \alpha \mid \beta \mid \gamma \) is \( (\alpha|\beta)|\gamma \).

New grammar:

Precedence/Associativity declarations:

<table>
<thead>
<tr>
<th>mosmlyac file</th>
</tr>
</thead>
<tbody>
<tr>
<td>%token BAR STAR LPAREN RPAREN ...</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>%left BAR /* lowest precedence */</td>
</tr>
<tr>
<td>%nonassoc CHAR LPAREN</td>
</tr>
<tr>
<td>%left seq /* pseudo-token for sequence */</td>
</tr>
<tr>
<td>%nonassoc STAR /* highest precedence */</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>R : R BAR R { ... }</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Full example: Mosmlyac Demo (regular expressions)
One word about the Syntax Trees

- **Concrete Syntax** contains many extra tokens for practical reasons:
  - Parentheses, braces, ... for grouping,
  - Semicolons, commas, ... to separate statements or arguments.
  - `begin`, `end` ... (also a kind of parentheses).

- Following stage works on **abstract syntax** tree without those

```
begin  id := num + id ;  if  id < num  then  id := id + num  end
```

```
S     E     E       B     E     E     E     E
     S     S
     S
```
One word about the Syntax Trees

- **Concrete Syntax** contains many extra tokens for practical reasons:
  - Parentheses, braces, ... for grouping,
  - Semicolons, commas, ... to separate statements or arguments.
  - `begin`, `end` ... (also a kind of parentheses).

- Following stage works on **abstract syntax** tree without those

\[
\text{begin} \; \text{id} := \text{num} + \text{id} \; ; \; \text{if} \; \text{id} < \text{num} \; \text{then} \; \text{id} := \text{id} + \text{num} \; \text{end}
\]
More about Context-Free Languages

- Context-Free languages are commonly processed using a stack machine (Push-Down Automaton, PDA)
- Can count one thing at a time, or remember input.
  \[
  \{ a^n b^n \mid n \in \mathbb{N} \} \text{ context-free.}
  \]
  \[
  \{ a^n b^n c^n \mid n \in \mathbb{N} \} \text{ not context-free!}
  \]
- Palindromes over \( \Sigma \): context-free language.
  However: non-deterministic (need to guess the middle).
  Non-deterministic stack machines are more powerful than deterministic ones (unlike NFAs and DFAs)!
- Context-free languages are closed under union:
  \( L_1, L_2 \text{ context-free } \sim L_1 \cup L_2 \text{ context-free.} \)
- ...but not closed under intersection (famous counter examples above) and complement (by de Morgan’s laws).
Summary

Context-free grammars and languages

- Writing and rewriting grammars can be tricky! :-)

Top-down parsing (recursive-descent)

- FIRST- and FOLLOW-sets;
- Look-ahead sets for decisions in recursive-descent parser.

Bottom-up parsing (shift-reduce parsing, SLR parsing)

- Items, grammar-implied NFA and subset construction;
- Reduce actions in transition table, stack of visited states.

Precedence and associativity

- Solved in the grammar or by manipulation of the SLR parser.