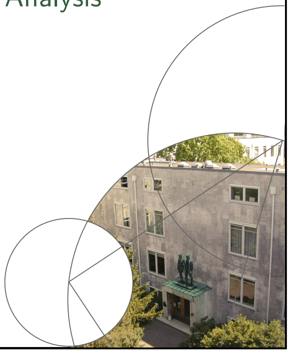
Faculty of Science

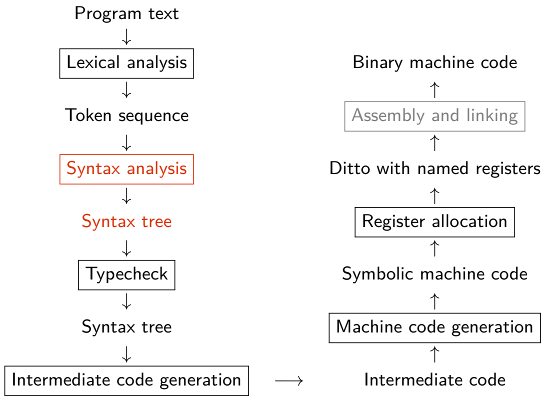
Compilers (Oversættere): Syntax Analysis

Jost Berthold
berthold@diku.dk
Department of Computer Science



November 2012

Syntax Analysis (Parsing)



Program text
↓
Lexical analysis
↓
Token sequence
↓
Syntax analysis
↓
Syntax tree
↓
Typecheck
↓
Syntax tree
↓
Intermediate code generation ⟶ Intermediate code
↑
Machine code generation
↑
Symbolic machine code
↑
Register allocation
↑
Ditto with named registers
↑
Assembly and linking
↑
Binary machine code

Syntax Analysis

syntax analysis	This analysis
covers	covers
lecture	lecture
This.	syntax.
Syntax error!	(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

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 Σ (the alphabet of the resulting language),
- a set of nonterminals N,
- a start symbol $S \in N$
- a set P of productions $X \to \alpha$ 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.

$$\begin{aligned} G : S &\to aSB \\ S &\to \varepsilon \\ S &\to B \\ B &\to Bb \mid b \end{aligned}$$

(alternative notation)

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

Example, Derivation of Words

$$\begin{aligned} G : S &\to aSB & (1) \\ S &\to \varepsilon & (2) \\ S &\to B & (3) \\ B &\to Bb & (4) \\ B &\to b & (5) \end{aligned}$$

$$S = \underbrace{\{\mathrm{a} \cdot x \cdot y \mid x \in S, y \in B\}}_{(1)} \cup \underbrace{\{\varepsilon\}}_{(2)} \cup \underbrace{B}_{(3)}$$

$$B = \underbrace{\{x \cdot \mathrm{b} \mid x \in B\}}_{(4)} \cup \underbrace{\{\mathrm{b}\}}_{(5)}$$

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

$$S \overset{1}{\Rightarrow} \underline{\mathrm{a}SB} \overset{1}{\Rightarrow} \mathrm{a}\underline{\mathrm{a}SB}B \overset{5}{\Rightarrow} \mathrm{aa}S\underline{\mathrm{b}}B \overset{1}{\Rightarrow} \mathrm{aa}\underline{\mathrm{a}SB}\mathrm{b}B$$
$$\overset{2}{\Rightarrow} \mathrm{aaa}\_B\mathrm{b}B \overset{4}{\Rightarrow} \mathrm{aaa}\underline{B\mathrm{b}}\mathrm{b}B \overset{5}{\Rightarrow} \mathrm{aaa}B\mathrm{bb}\underline{\mathrm{b}} \overset{5}{\Rightarrow} \mathrm{aaa}\underline{\mathrm{b}}\mathrm{bbb}$$

Derivation Relation

Definition (Derivation $\Rightarrow$)

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 $(X \to \beta) \in P$ of the grammar, $\underline{\alpha_1 X \alpha_2 \Rightarrow \alpha_1 \beta \alpha_2}$ 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 ($\overset{k}{\Rightarrow}$).
- Can indicate left-most (or right-most) derivation ($\overset{k}{\Rightarrow}_l, \overset{k}{\Rightarrow}_r$).

$$\begin{array}{llll} G: S & \to & aSB & (1) \\ S & \to & \varepsilon & (2) \\ S & \to & B & (3) \\ B & \to & Bb & (4) \\ B & \to & b & (5) \end{array}$$

$$S \overset{1}{\Rightarrow} \underline{\mathrm{a}SB} \overset{1}{\Rightarrow} \mathrm{a}\underline{\mathrm{a}SB}B \overset{2}{\Rightarrow} \mathrm{aa}\underline{\ }BB \overset{4}{\Rightarrow} \mathrm{aa}\underline{B\mathrm{b}}B \overset{5}{\Rightarrow} \mathrm{aa}\underline{\mathrm{b}}\mathrm{b}B \overset{5}{\Rightarrow} \mathrm{aabb}\underline{\mathrm{b}}$$

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).

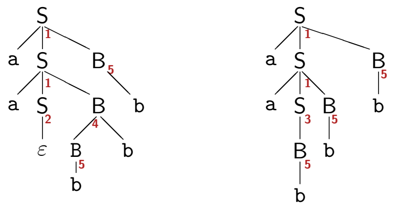
More generally, this is known as the transitive closure of a relation.
In our previous examples, we saw $S \Rightarrow^*$ aaabbbb and $S \Rightarrow^*$ aabbb.
That means, both words are in the language of G.

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

$$\begin{array}{llll} G: S & \to & aSB & (1) \\ S & \to & \varepsilon & (2) \\ S & \to & B & (3) \\ B & \to & Bb & (4) \\ B & \to & b & (5) \end{array}$$



- Syntax trees describe the derivation independent of the direction.
- Left-most derivation: depth-first left-to-right tree traversal.
- $S \overset{1}{\Rightarrow} \underline{\mathrm{a}SB} \overset{1}{\Rightarrow} \mathrm{a}\underline{\mathrm{a}SB}B \overset{2}{\Rightarrow} \mathrm{aa}\underline{\ }BB \overset{4}{\Rightarrow} \mathrm{aa}\underline{B\mathrm{b}}B \overset{5}{\Rightarrow} \mathrm{aa}\underline{\mathrm{b}}\mathrm{b}B \overset{5}{\Rightarrow} \mathrm{aabb}\underline{\mathrm{b}}$

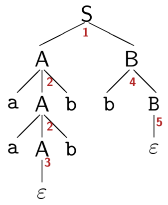
Nevertheless: $S \Rightarrow^*$ aabbb can be derived in two ways.

- $S \overset{1}{\Rightarrow} \underline{\mathrm{a}SB} \overset{1}{\Rightarrow} \mathrm{a}\underline{\mathrm{a}SB}B \overset{3}{\Rightarrow} \mathrm{aa}\underline{B}BB \overset{5}{\Rightarrow} \mathrm{aa}\underline{\mathrm{b}}BB \overset{5}{\Rightarrow} \mathrm{aab}\underline{\mathrm{b}}B \overset{5}{\Rightarrow} \mathrm{aabb}\underline{\mathrm{b}}$

The grammar G is said to be ambiguous.

Avoiding Ambiguity (Changed Grammar)

$$\begin{array}{lll} G: S & \to & aSB \\ S & \to & \varepsilon \\ S & \to & B \\ B & \to & Bb \\ B & \to & b \end{array} \qquad \begin{array}{llll} G': S & \to & AB & (1) \\ A & \to & aAb & (2) \\ A & \to & \varepsilon & (3) \\ B & \to & bB & (4) \\ B & \to & \varepsilon & (5) \end{array}$$



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)
- Left-most derivation: (1 2 2 3 4 5)

$$S \overset{1}{\Rightarrow}_l \underline{AB} \overset{2}{\Rightarrow}_l \underline{\mathrm{a}A\mathrm{b}}B \overset{2}{\Rightarrow}_l \mathrm{a}\underline{\mathrm{a}A\mathrm{b}}\mathrm{b}B \overset{3}{\Rightarrow}_l \mathrm{aa}\underline{\ }\mathrm{bb}B \overset{4}{\Rightarrow}_l \mathrm{aabb}\underline{\mathrm{b}B} \overset{5}{\Rightarrow}_l \mathrm{aabbb}\underline{\ }$$

Parsing

Token sequence
↓
Syntax analysis
↓
Syntax tree

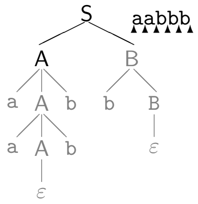
- Producing a syntax tree from a token sequence.
- Representation of the tree: left-most or right-most derivation

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



How can we decide which production to use?

- 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)

Token sequence
↓
Syntax analysis
↓
Syntax tree

- 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 $A \to \alpha$ 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

- $\textsc{First}(\alpha) = \{c \in \Sigma \mid \exists_{\beta \in (\Sigma \cup N)^*} :\ \alpha \Rightarrow^* c\beta\}$
  (all terminals at the start of what can be derived from α)
- $\textsc{Nullable}(\alpha) = \begin{cases} \mathit{true} & \text{, if } \alpha \Rightarrow^* \varepsilon \\ \mathit{false} & \text{, otherwise} \end{cases}$

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

$$\begin{array}{lcl}
\textsc{Nullable}(\varepsilon) & = & \mathit{true} \\
\textsc{Nullable}(\mathrm{a}) & = & \mathit{false} \text{ for } \mathrm{a} \in \Sigma \\
\textsc{Nullable}(\alpha\beta) & = & \textsc{Nullable}(\alpha) \wedge \textsc{Nullable}(\beta) \text{ for } \alpha, \beta \in (\Sigma \cup N)^* \\
\textsc{Nullable}(A) & = & \textsc{Nullable}(\alpha_1) \vee \ldots \vee \textsc{Nullable}(\alpha_n), \\
 & & \text{using all productions for } A,\ A \to \alpha_i\ (i \in \{1..n\})
\end{array}$$

- Equations for nonterminals of the grammar:

$$\begin{array}{lclclcl}
G' : S & \to & AB & \quad \textsc{Nullable}(S) & = & \textsc{Nullable}(AB) = \mathit{true} \\
A & \to & aAb \mid \varepsilon & \quad \textsc{Nullable}(A) & = & \textsc{Nullable}(aAb) \vee \textsc{Nullable}(\varepsilon) = \mathit{true} \\
B & \to & bB \mid \varepsilon & \quad \textsc{Nullable}(B) & = & \textsc{Nullable}(bB) \vee \textsc{Nullable}(\varepsilon) = \mathit{true}
\end{array}$$

- Equations for the right-hand side

$$\begin{array}{lcl}
\textsc{Nullable}(AB) & = & \textsc{Nullable}(A) \wedge \textsc{Nullable}(B) \\
\textsc{Nullable}(aAb) & = & \textsc{Nullable}(a) \wedge \textsc{Nullable}(A) \wedge \textsc{Nullable}(b) = \mathit{false} \\
\textsc{Nullable}(bB) & = & \textsc{Nullable}(b) \wedge \textsc{Nullable}(B) = \mathit{false} \\
\textsc{Nullable}(\varepsilon) & = & \mathit{true}
\end{array}$$

Compute smallest solution of system, starting by *false* for all.

Computing FIRST by Set Equations

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

- Equations for nonterminals of the grammar:

$$\begin{array}{lclclcl}
G' : S & \to & AB & \quad \textsc{First}(S) & = & \textsc{First}(AB) = \textsc{First}(A) \cup \textsc{First}(B) = \{a, b\} \\
A & \to & aAb \mid \varepsilon & \quad \textsc{First}(A) & = & \textsc{First}(aAb) \cup \textsc{First}(\varepsilon) = \{a\} \\
B & \to & bB \mid \varepsilon & \quad \textsc{First}(B) & = & \textsc{First}(bB) \cup \textsc{First}(\varepsilon) = \{b\}
\end{array}$$

- Equations for the right-hand side

$$\begin{array}{lcl}
\textsc{First}(aAB) & = & \textsc{First}(a) = \{a\} \\
\textsc{First}(bB) & = & \textsc{First}(b) = \{b\} \\
\textsc{First}(\varepsilon) & = & \emptyset
\end{array}$$

Compute smallest solution of system, starting by $\emptyset$ for all sets.

FOLLOW Sets for Nonterminals

FIRST Sets are often not enough.
In production $X \to \alpha$, if $\textsc{Nullable}(\alpha)$, we need to know what can follow X (FIRST set of α 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

- $\textsc{Follow}(X) = \{c \in \Sigma \mid \exists_{\alpha,\beta \in (\Sigma \cup N)^*} :\ S \Rightarrow^* \alpha \underline{Xc}\beta\}$
  (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 $S' \to S\$$ to the grammar.

Thereby, we always have $\$ \in \textsc{Follow}(S)$.

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 $Y \to \alpha X \beta$ where X occurs on the right.

- $\underline{\textsc{First}(\beta) \subseteq \textsc{Follow}(X)}$
- If $\textsc{Nullable}(\beta)$ or $\beta = \varepsilon$: $\underline{\textsc{Follow}(Y) \subseteq \textsc{Follow}(X)}$

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

Example:

$$\begin{array}{lcrl}
S' \to S\$ & \ldots & \textsc{First}(\$) = \{\$\} & \subseteq \textsc{Follow}(S) \\
S \to AB & \ldots & \textsc{First}(B) = \{b\} & \subseteq \textsc{Follow}(A) \\
 & & \textsc{Follow}(S) & \subseteq \textsc{Follow}(A) \ \text{(B nullable)} \\
 & & \textsc{Follow}(S) & \subseteq \textsc{Follow}(B) \\
A \to aAb & \ldots & \textsc{First}(b) = \{b\} & \subseteq \textsc{Follow}(A) \\
B \to bB & \ldots & \textsc{Follow}(B) & \subseteq \textsc{Follow}(B)
\end{array}$$

$A \to \varepsilon$ and $B \to \varepsilon$ do not contribute.

Solve iteratively, starting by $\emptyset$ for all nonterminals.

$$\curvearrowright \begin{array}{ll} \textsc{Follow}(S) = & \textsc{Follow}(B) = \{\$\} \\ \textsc{Follow}(A) = & \{\$,\ \mathrm{b}\} \end{array}$$

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 $X \to \alpha$ of a context-free grammar G, we define the Look-ahead set of the production as:

$$la(X \to \alpha) = \begin{cases} \textsc{First}(\alpha) \cup \textsc{Follow}(X) & \text{, if } \textsc{Nullable}(\alpha) \\ \textsc{First}(\alpha) & \text{, otherwise} \end{cases}$$

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):

$$\begin{array}{llll} G' : S & \to & AB & la(S \to AB) = \textsc{First}(AB) \cup \textsc{Follow}(S) = \{a, b, \$\} \\ A & \to & aAb & la(A \to aAb) = \textsc{First}(aAB) = \{a\} \\ A & \to & \varepsilon & la(A \to \varepsilon) = \textsc{First}(\varepsilon) \cup \textsc{Follow}(A) = \{b, \$\} \\ B & \to & bB & la(B \to bB) = \textsc{First}(bB) = \{b\} \\ B & \to & \varepsilon & la(B \to \varepsilon) = \textsc{First}(\varepsilon) \cup \textsc{Follow}(B) = \{\$\} \end{array}$$

```
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.

	Look-ahead/Input:		
Stack:	a	b	$
S	AB, 1	AB, 1	AB, 1
A	aAb, 2	ε, 3	ε, 3
B	*error*	*bB*, 4	ε, 5
a	*pop*	*error*	*error*
b	*error*	*pop*	*error*
$	*error*	*error*	*accept*

Example run (input `aabbb`):

Input	Stack	Action	Output
aabbb$	$S\$$	derive	ε
aabbb$	$AB\$$	derive	1
aabbb$	$aAbB\$$	pop	12
abbb$	$AbB\$$	derive	12
abbb$	$aAbbB\$$	pop	122
bbb$	$AbbB\$$	derive	122
bbb$	$bbB\$$	pop	1223
bb$	$bB\$$	pop	1223
b$	$B\$$	derive	1223
b$	$bB\$$	pop	12234
$	$B\$$	derive	12234
$	$	accept	122345

Eliminating Left-Recursion and Left-Factorisation

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

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

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

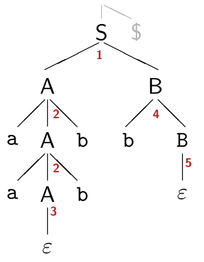
- Left-Recursion: a nonterminal reproducing itself on the left.
  Direct: production $X \to X\alpha \mid \beta$, or indirect: $X \Rightarrow^* X\alpha$.
  Cannot be analysed with finite look-ahead!
  $X \to X\alpha \mid \beta$, thus $\textsc{first}(X) \subset \textsc{first}(X\alpha) \cup \textsc{First}(\beta)$
  Solution: new (nullable) nonterminals and swapped recursion.
  $X \to \beta X'$ and $X' \to \alpha X' \mid \varepsilon$
  Also works in case of multiple left-recursive productions.
  For indirect recursion: first transform into direct recursion.

Contents

Bottom-Up Parsing

LL(1) Parser works top-down. Needs to guess used productions.
Bottom-Up approach: build syntax tree from leaves.

$$\begin{array}{llll} G'' : S' & \to & S\$ & (0) \\ S & \to & AB & (1) \\ A & \to & aAb & (2) \\ A & \to & \varepsilon & (3) \\ B & \to & bB & (4) \\ B & \to & \varepsilon & (5) \end{array}$$

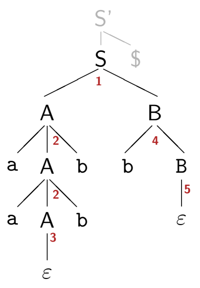


$$S' \overset{0}{\Rightarrow}_r S\$ \overset{1}{\Rightarrow}_r \underline{AB} \overset{4}{\Rightarrow}_r A\underline{bB} \overset{5}{\Rightarrow}_r Ab\_ \overset{2}{\Rightarrow}_r \underline{aAb}b \overset{2}{\Rightarrow}_r a\underline{aAb}bb \overset{3}{\Rightarrow}_r aa\_bbb$$

Right-most derivation: 1 4 5 2 2 3

Bottom-Up Parsing: Idea for a Machine

Stack	Input	Action
ε	aabbb$	**shift**
a	abbb$	**shift**
aa_	bbb$	**reduce 3**
aa$A$	bbb$	**shift**
a$\underline{\mathrm{a}A\mathrm{b}}$	bb$	**reduce 2**
aA	bb$	**shift**
$\underline{\mathrm{a}A\mathrm{b}}$	b$	**reduce 2**
A	b$	**shift**
Ab_	$	**reduce 5**
$A\underline{\mathrm{b}B}$	$	**reduce 4**
$\underline{AB}$	$	**reduce 1**
$\underline{S}$	$	**accept**



$$\begin{aligned} G'' : S' &\rightarrow S\$ \ (0)\\ S &\rightarrow AB \ (1)\\ A &\rightarrow aAb (2)\\ A &\rightarrow \varepsilon \quad (3)\\ B &\rightarrow bB \ (4)\\ B &\rightarrow \varepsilon \quad (5)\end{aligned}$$

Questions:

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

`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 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: $[X \rightarrow \alpha_1 \bullet \alpha_2]$ 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:

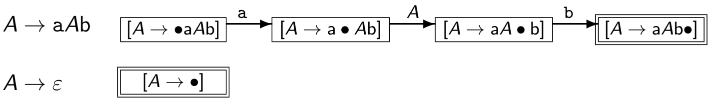
- Either to shift an item to the stack and read input
- or to reduce the top of stack (a production's right-hand side).
- Stack of the parser will contain items, not grammar symbols.
- Therefore, no need to read into the stack for reductions.

Constructing an SLR Parser: Production DFAs

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

- From $[X \rightarrow \alpha \bullet \mathrm{a}\beta]$ to $[X \rightarrow \alpha\mathrm{a} \bullet \beta]$ 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 NFA

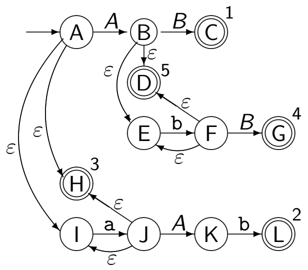
$S \rightarrow AB$

$B \rightarrow \varepsilon$

$B \rightarrow \mathrm{b}B$

$A \rightarrow \varepsilon$

$A \rightarrow \mathrm{a}A\mathrm{b}$



Extra ε-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 NFA

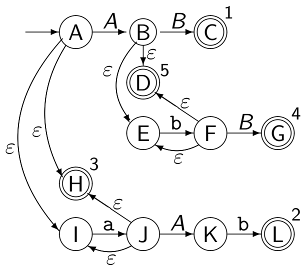
$S \rightarrow AB$

$B \rightarrow \varepsilon$

$B \rightarrow \mathrm{b}B$

$A \rightarrow \varepsilon$

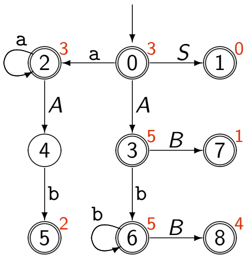
$A \rightarrow \mathrm{a}A\mathrm{b}$



Next step: Subset construction of a combined DFA.

Blackboard. . .

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

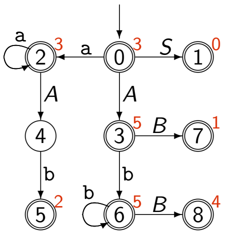
SLR Parser Conflicts

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 ε 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



$$
\begin{array}{rcll}
G'' : S' & \rightarrow & S\$ & (0)\\
S & \rightarrow & AB & (1)\\
A & \rightarrow & aAb & (2)\\
A & \rightarrow & \varepsilon & (3)\\
B & \rightarrow & bB & (4)\\
B & \rightarrow & \varepsilon & (5)
\end{array}
$$

- Parser Table:

	a	b	$	S	A	B
0	2	red.3	red.3	Go 1	Go 3	
1			acc.			
2	2	red.3	red.3		Go 4	
3		6	red.5			Go 7
4		5				
5		red.2	red.2			
6		6	red.5			Go 8
7			red.1			
8			red.4			

- FOLLOW Sets of Nonterminals:

$$
\begin{array}{rcl}
\textsc{Follow}(S) & = & \{\$\}\\
\textsc{Follow}(A) & = & \{b, \$\}\\
\textsc{Follow}(B) & = & \{\$\}
\end{array}
$$

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.

	a	b	$	S	A	B
0	2	red.3	red.3	Go 1	Go 3	
1			acc.			
2	2	red.3	red.3		Go 4	
3		6	red.5			Go 7
4		5				
5		red.2	red.2			
6		6	red.5			Go 8
7			red.1			
8			red.4			

Example run (`aabbb`):

Stack	Input	Action
0	aabbb$	**shift**
02	abbb$	**shift**
022_	bbb$	**reduce 3**
0224	bbb$	**shift**
02245	bb$	**reduce 2**
024	bb$	**shift**
0245	b$	**reduce 2**
03	b$	**shift**
036_	$	**reduce 5**
0368	$	**reduce 4**
037	$	**reduce 1**
01	$	**accept**

Contents

Ambiguity, Precedence and Associativity

Arithmetic Expressions:

$$
\begin{array}{rcl}
E & \rightarrow & E + E \mid E - E\\
E & \rightarrow & E * E \mid E / E\\
E & \rightarrow & \mathtt{a} \mid (E)
\end{array}
$$

- 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:

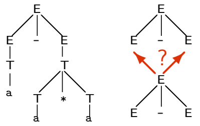
1. Ambiguous derivation of `a - a * a`.
   Want precedence of `*` over `+`, $a + (a \cdot a)$.
2. Ambiguous derivation of `a - a - a`.
   Want a left-associative interpretation, $(a - a) - a$.

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).

$$\begin{array}{lcl} E & \rightarrow & E + E \mid E - E \\ E & \rightarrow & E * E \mid E / E \\ E & \rightarrow & \mathrm{a} \mid (E) \end{array}$$

$$\begin{array}{lcl} E & \rightarrow & E + E \mid E - E \mid T \\ T & \rightarrow & T * T \mid T / T \\ T & \rightarrow & \mathrm{a} \mid (E) \end{array}$$



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

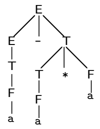
```
if (3 < 2 < 1) { fprintf(stdout, "Awesome!\n"); }
```

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.

$$\begin{array}{lcl} E & \rightarrow & E + E \mid E - E \mid T \\ T & \rightarrow & T * T \mid T / T \\ T & \rightarrow & \mathrm{a} \mid (E) \end{array}$$

$$\begin{array}{lcl} E & \rightarrow & E + T \mid E - T \mid T \\ T & \rightarrow & T * F \mid T / F \mid F \\ F & \rightarrow & \mathrm{a} \mid (E) \end{array}$$



Precedence and Associativity in SLR Parse Tables

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

$$\begin{array}{lcl} E & \rightarrow & E * E \mid E + E \mid \ldots \\ & & \mid \mathrm{a} \mid (E) \end{array} \quad \Longrightarrow \quad \begin{array}{l} [E \rightarrow E + E \bullet], \\ [E \rightarrow E \bullet + E], \\ [E \rightarrow E \bullet * E] \\ \ldots \end{array}$$

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:

$$\begin{array}{lcl} R & \rightarrow & R\text{`}|\text{'}R \\ R & \rightarrow & RR \\ R & \rightarrow & R\text{`}*\text{'} \\ R & \rightarrow & \mathrm{char} \mid (R) \end{array}$$

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

② Left-associative derivations:
α `|` β `|` γ is $(\alpha|\beta)|\gamma$.

New grammar:

$$\begin{array}{lcl} R & \rightarrow & R\text{`}|\text{'}R2 \mid R2 \\ R2 & \rightarrow & R2R3 \mid R3 \\ R3 & \rightarrow & R4\text{`}*\text{'} \mid R4 \\ R4 & \rightarrow & \mathrm{char} \mid (R) \end{array}$$

Precedence/Associativity declarations:

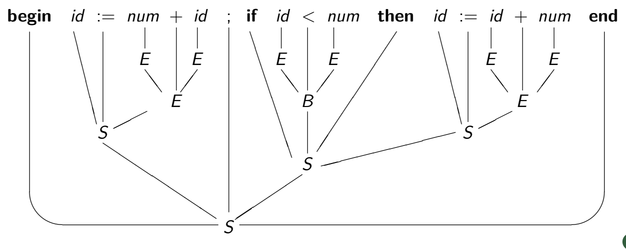
mosmlyac file

```
%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)

One word about the Syntax Trees

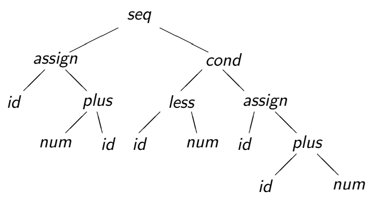
- Concrete Syntax contains many extra tokens for practical reasons:
  - Parentheses, braces, . . . for grouping,
  - Semicolons, commas, . . . to separate statements or arguments.
  - `begin`, `end` . . . (also a kind of parentheses).
- Following stage works on abstract syntax tree without those



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**



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.
  $\{a^n b^n \mid n \in \mathbb{N}\}$ context-free.
  $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ not context-free!
- Palindromes over Σ: 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 context-free $\curvearrowright$ $L_1 \cup L_2$ 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.