Context-free grammars

Formal way of specifying rules about the structure/syntax of a program

- terminals - tokens
- non-terminals - represent higher-level structures of a program
- start symbol, productions

Example:

\[
E \rightarrow E \ op \ E \ | \ (E) \ | \ - \ E \ | \ id
\]

\[
op \rightarrow + \ | \ - \ | \ * \ | \ / \ | \ %
\]

NOTE: recall token vs. lexeme difference

Derivation: starting from the start symbol, use productions to generate a string (sequence of tokens)

Parse tree: pictorial representation of a derivation
Left-most derivation: at each step, replace left-most non-terminal

Ambiguous grammar: \( G \) is ambiguous if a string has > 1 left-most (or right-most) derivation

ALT: \( G \) is ambiguous if > 1 parse tree can be constructed for a string

Examples:

1. \( E \rightarrow E + E \quad E \rightarrow E \ast E \)

2. \( stmt \rightarrow \text{if expr then stmt} \)
   \[ \rightarrow \text{if expr then stmt else stmt} \]

[ SOLUTION: \( stmt \rightarrow \text{matched | unmatched} \)
  \( \text{matched} \rightarrow \text{if E then matched else matched} \)
  \( \text{unmatched} \rightarrow \text{if E then stmt} \)
  \[ \rightarrow \text{if E then matched else unmatched} \] ]
**Recursive descent parsing:** corresponds to finding a leftmost derivation for an input string
Equivalent to constructing parse tree in pre-order
Example:

Grammar: $S \rightarrow cAd \quad A \rightarrow ab \mid a$
Input: cad

**Problems:**
1. backtracking involved ($\Rightarrow$ buffering of tokens required)
2. left recursion will lead to infinite looping
3. left factors may cause several backtracking steps
Left recursion: $G$ is left recursive if for some non-terminal $A$, $A \Rightarrow A\alpha$

Simple case I: $A \rightarrow A\alpha | \beta \Rightarrow A \rightarrow \beta A' \quad A' \rightarrow \alpha A' | \epsilon$

Simple case II:

\[
\begin{align*}
A & \rightarrow A\alpha_1 | A\alpha_2 | \ldots | A\alpha_m | \\
& \quad | \beta_1 | \ldots | \beta_n \\
\downarrow & \\
A & \rightarrow \beta_1 A' | \ldots | \beta_n A' \\
A' & \rightarrow \alpha_1 A' | \alpha_2 A' | \ldots | \alpha_m A' | \epsilon
\end{align*}
\]
Top-down parsing - I

General case:

Input: $G$ without any cycles ($A \to A$) or $\varepsilon$-productions
Output: equivalent non-recursive grammar

Algorithm:

Let the non-terminals be $A_1, \ldots A_n$.

for $i = 1$ to $n$ do
  for $j = 1$ to $i - 1$ do
    replace $A_i \to A_j \gamma$ by $A_i \to \delta_1 \gamma | \delta_2 \gamma | \ldots | \delta_k \gamma$
    where $A_j \to \delta_1 | \delta_2 | \ldots | \delta_k$ are the current $A_j$
    productions
  end for
end for

Eliminate immediate left-recursion for $A_i$. 

end for
Top-down parsing - I

Left factoring:
Example: \( stmt \rightarrow \text{if ( expr ) stmt} \)
\[ \downarrow \]
\( \text{if ( expr ) stmt else stmt} \)

Algorithm:
while left factors exist do
  for each non-terminal \( A \) do
    Find longest prefix \( \alpha \) common to \( \geq 2 \) rules
    Replace \( A \rightarrow \alpha \beta_1 | \ldots | \alpha \beta_n | \ldots \)
    by \( A \rightarrow \alpha A' | \ldots \)
    \( A' \rightarrow \beta_1 | \ldots | \beta_n \)
end for
end while
**Predictive parsing:** recursive descent parsing without backtracking

**Principle:** Given current input symbol and non-terminal, we should be able to determine which production is to be used

Example: \( stmt \rightarrow \text{if} ( expr ) \ldots \)

\[\text{while} \ldots\]
\[\text{for} \ldots\]
Implementation: use transition diagrams (1 per non-terminal)

\[ A \rightarrow X_1 X_2 \ldots X_n \]

1. If \( X_i \) is a terminal, match with next input token and advance to next state.

2. If \( X_i \) is a non-terminal, go to the transition diagram for \( X_i \), and continue. On reaching the final state of that transition diagram, advance to next state of current transition diagram.

Example:

\[ E \rightarrow E + T \mid T \]
\[ T \rightarrow T \times F \mid F \]
\[ F \rightarrow (E) \mid \text{id} \]
Non-recursive implementation:

Table: 2-d array s.t. $M[A, a]$ specifies $A$-production to be used if input symbol is $a$

Algorithm:

0. Initially: stack contains $\langle$EOF $S\rangle$, input pointer is at start of input

1. if $X = a = \text{EOF}$, done

2. if $X = a \neq \text{EOF}$, pop stack and advance input pointer

3. if $X$ is non-terminal, lookup $M[X, a] \Rightarrow X \rightarrow UVW$
   pop $X$, push $W, V, U$
**FIRST and FOLLOW**

**FIRST**\(\alpha\): set of terminals that begin strings derived from \(\alpha\)
- if \(\alpha \Rightarrow^* \epsilon\), then \(\epsilon \in \text{FIRST}\(\alpha\)\)

**FOLLOW**\(A\): set of terminals that can appear immediately to the right of \(A\) in some sentential form

\[
\text{FOLLOW}\(A\) = \{a \mid S \Rightarrow^* \alpha Aa\beta\}
\]
- if \(A\) is the rightmost symbol in any sentential form, then \(\text{EOF} \in \text{FOLLOW}\(A\)\)
**FIRST and FOLLOW**

**FIRST:**

1. if $X$ is a terminal, then $\text{FIRST}(X) = \{X\}$
2. if $X \rightarrow \epsilon$ is a production, then add $\epsilon$ to $\text{FIRST}(X)$
3. if $X \rightarrow Y_1Y_2 \ldots Y_k$ is a production:
   - if $a \in \text{FIRST}(Y_i)$ and $\epsilon \in \text{FIRST}(Y_1), \ldots, \text{FIRST}(Y_{i-1})$, 
     add $a$ to $\text{FIRST}(X)$
   - if $\epsilon \in \text{FIRST}(Y_i) \ \forall i$, add $\epsilon$ to $\text{FIRST}(X)$

**FOLLOW:**

1. Add EOF to $\text{FOLLOW}(S)$
2. For each production of the form $A \rightarrow \alpha B \beta$
   (i) add $\text{FIRST}(\beta) \setminus \{\epsilon\}$ to $\text{FOLLOW}(B)$
   (ii) if $\beta = \epsilon$ or $\epsilon \in \text{FIRST}(\beta)$, then add everything in $\text{FOLLOW}(A)$ to $\text{FOLLOW}(B)$
**Table construction**

**Algorithm:**

1. For each production $A \rightarrow \alpha$
   
   (i) for each terminal $a \in FIRST(\alpha)$, add $A \rightarrow \alpha$ to $M[A, a]$
   
   (ii) if $\epsilon \in FIRST(\alpha)$, add $A \rightarrow \alpha$ to $M[A, b]$ for each terminal $b \in FOLLOW(A)$

2. Mark all other entries “error”

**Example:**

Grammar:

\[
\begin{align*}
E & \rightarrow E + T \mid T \\
T & \rightarrow T \ast F \mid F \\
F & \rightarrow (E) \mid id
\end{align*}
\]

Input: id + id * id
If the table has no multiply defined entries, grammar is a

\[ LL(1) \]

\[ L - \text{left-to-right} \quad L - \text{leftmost} \quad 1 - \text{lookahead} \]

If \( G \) is \( LL(1) \), then \( G \) cannot be left-recursive or ambiguous

Example:

\[
S \quad \rightarrow \quad i \ E \ t \ S \ S' \mid a \\
S' \quad \rightarrow \quad e \ S \mid \epsilon \\
E \quad \rightarrow \quad b
\]

\[ M[S', e] = \{ S' \rightarrow \epsilon, S' \rightarrow e \ S \} \]

Some non-\( LL(1) \) grammars may be transformed into equivalent \( LL(1) \) grammars
Error recovery:
1. if $X$ is a terminal, but $X \neq a$, pop $X$
2. if $M[X, a]$ is blank, skip $a$
3. if $M[X, a] = synch$, pop $X$, but do not advance input pointer

Synch sets:
use $FOLLOW(A)$
add the $FIRST$ set of a higher-level non-terminal to the $synch$ set of a lower-level non-terminal
**Bottom-up parsing**

Example: Grammar: 

\[
E \rightarrow E + T \mid T \\
T \rightarrow T * F \mid F \\
F \rightarrow (E) \mid id
\]

Input: id + id * id

Sentential form: any string \( \alpha \) s.t. \( S \Rightarrow^* \alpha \)

Handle: for a sentential form \( \gamma \), handle is a production \( A \rightarrow \beta \) and a position of \( \beta \) in \( \gamma \), s.t. \( \beta \) may be replaced by \( A \) to produce the previous right-sentential form in a rightmost derivation of \( \gamma \)

Properties:

1. string to right of handle must consist of terminals only
2. if \( G \) is unambiguous, every right-sentential form has a unique handle

Advantages:

1. No backtracking  
2. More powerful than \( LL(1) \) / predictive parsing
Bottom-up parsing

Implementation scheme:

0. Use input buffer, stack, and parsing table.
1. Shift ≥ 0 input symbols onto stack until a handle β is on top of stack.
2. Reduce β to A (i.e. pop symbols of β and push A).
3. Stop when stack = ⟨EOF, S⟩, and input pointer is at EOF.

Stack: $s_0X_1s_1 \ldots X_ms_m$, where each $s_i$ represents a “state” (current situation in the parsing process)

Table:
- used to guide steps 2 and 3
- 2-d array indexed by ⟨state, input symbol⟩ pairs
- consists of two parts (action + goto)
Algorithm:
1. Initially, stack = $\langle s_0 \rangle$ (initial state)
2. Let $s$ - state on top of stack
   $a$ - current input symbol
   if action$[s, a] = \text{shift } s'$
     push $a, s'$ on stack, advance input pointer
   if action$[s, a] = \text{reduce } A \rightarrow \beta$
     pop $2 \times |\beta|$ symbols
     let $s'$ be the new top of stack
     push $A$, goto$[s', A]$ on stack
   if action$[s, a] = \text{accept, done}$
   else error
Grammar augmentation:
Create new start symbol $S'$; add $S' \rightarrow S$ to productions

Item (LR(0) item): production of $G$ with a dot at some position in the RHS, representing how much of the RHS has already been seen at a given point in the parse
Example: $A \rightarrow \epsilon \Rightarrow A \rightarrow \cdot$

Closure:
Let $I$ be a set of items
$\text{closure}(I) \leftarrow I$
repeat until no more changes
for each $A \rightarrow \alpha \cdot B\beta$ in $\text{closure}(I)$
for each production $B \rightarrow \gamma$ s.t. $B \rightarrow \cdot \gamma \notin \text{closure}(I)$
add $B \rightarrow \cdot \gamma$ to $\text{closure}(I)$

Example: $\text{closure}(E' \rightarrow \cdot E)$
Goto construction:
\[ \text{goto}(I, X) = \text{closure}(\{ A \rightarrow \alpha X \cdot \beta \mid A \rightarrow \alpha \cdot X \beta \in I \}) \]

Example: Let \( I = \{ E' \rightarrow E \cdot, E \rightarrow E \cdot + T \} \)
\[ \text{goto}(I, +) = \text{closure}(\{ E \rightarrow E + \cdot T \}) \]

Canonical collection construction:
1. \( C \leftarrow \{ \text{closure}(\{ S' \rightarrow \cdot S \}) \} \)
2. repeat until no more changes:
   for each \( I \in C \), for each grammar symbol \( X \)
   if \( \text{goto}(I, X) \) is not empty and not in \( C \)
   add \( \text{goto}(I, X) \) to \( C \)
Table construction:

1. Let $C = \{I_0, \ldots, I_n\}$ be the canonical collection of $LR(0)$ items for $G$.

2. Create a state $s_i$ corresponding to each $I_i$. The set containing $S' \rightarrow \cdot S$ corresponds to the initial state.

3. If $A \rightarrow \alpha \cdot a\beta \in I_i$ and $\text{goto}(I_i, a) = I_j$, then $\text{action}(s_i, a) = \text{shift} s_j$.

4. If $A \rightarrow \alpha \cdot \in I_i (A \neq S')$, then $\text{action}(s_i, a) = \text{reduce} A \rightarrow \alpha$ for all $a \in \text{FOLLOW}(A)$.

5. If $S' \rightarrow S \cdot \in I_i$, then $\text{action}(s_i, \text{EOF}) = \text{accept}$.

6. If $\text{goto}(I_i, a) = I_j$, then $\text{goto}(s_i, a) = s_j$.

7. Mark all blank entries error.
Conflicts

1. Shift-reduce conflict: 

   \[ stmt \rightarrow \text{if ( expr ) stmt} \] 
   \[ \text{if ( expr ) stmt else stmt} \]

2. Reduce-reduce conflict: 

   \[ stmt \rightarrow \text{id ( param_list ) ;} \]
   \[ \text{id ( expr_list )} \]
   \[ \vdots \]
   \[ \text{id} \]
   \[ \text{id} \]

Example:

Grammar: 

\[ S \rightarrow L = R \]
\[ S \rightarrow R \]
\[ L \rightarrow *R \]
\[ L \rightarrow \text{id} \]
\[ R \rightarrow L \]

Canonical collection:

\[ I_0 = \{ S' \rightarrow \cdot S, \ldots \} \]
\[ I_2 = \{ S \rightarrow L \cdot = R, R \rightarrow L \cdot \} \]

Table: 

\[ \text{action(2, =) = shift} \]
\[ \text{action(2, =) = reduce} \]

NOTE: \( SLR(1) \) grammars are unambiguous, but not vice versa.
Motivation: Reduction by $A \to \alpha \cdot$ not necessarily proper even if $a \in FOLLOW(A)$

⇒ explicitly indicate tokens for which reduction is acceptable

LR(1) item: pair of the form $\langle A \to \alpha \cdot \beta, a \rangle$, where $A \to \alpha \beta$ is a production, $a$ is a terminal or EOF

Properties:

1. $\langle A \to \alpha \cdot \beta, a \rangle$ - lookahead has no effect
   $\langle A \to \alpha \cdot, a \rangle$ - reduce only if input symbol is $a$

2. $\{ a \mid \langle A \to \alpha \cdot, a \rangle \in \text{canonical collection} \} \subseteq FOLLOW(A)$
Canonical LR parsers

Closure:
Let $I$ be a set of items
$\text{closure}(I) \leftarrow I$
repeat until no more changes
  for each item $\langle A \rightarrow \alpha \cdot B\beta, \ a \rangle$ in $\text{closure}(I)$
    for each production $B \rightarrow \gamma$ and each terminal $b \in \text{FIRST}(\beta a)$
      if $\langle B \rightarrow \cdot \gamma, \ b \rangle \notin \text{closure}(I)$
        add $\langle B \rightarrow \cdot \gamma, \ b \rangle$ to $\text{closure}(I)$

Goto:
$\text{goto}(I, X) = \text{closure}(\{ \langle A \rightarrow \alpha X \cdot \beta, \ a \rangle \mid \langle A \rightarrow \alpha \cdot X\beta, \ a \rangle \in I \})$

Canonical collection construction:
1. $\mathcal{C} \leftarrow \{ \text{closure}((\{ \langle S' \rightarrow \cdot S, \ \text{EOF} \rangle \})) \}$
2. /* Similar to SLR algorithm */
Table construction:

1. If $\langle A \rightarrow \alpha \cdot a\beta, b \rangle \in I_i$ and $\text{goto}(I_i, a) = I_j$ then $\text{action}(i, a) = \text{shift } j$.

2. If $\langle A \rightarrow \alpha \cdot, b \rangle \in I_i$, then $\text{action}(i, b) = \text{reduce } A \rightarrow \alpha \ (A \neq S')$.
   If $\langle S' \rightarrow S \cdot, \text{EOF} \rangle \in I_i$, then $\text{action}(i, \text{EOF}) = \text{accept}$.
**Motivation:** try to combine efficiency of SLR parser with power of canonical method

**Core:** set of $LR(0)$ items corresponding to a set of $LR(1)$ items

**Method:**

1. Construct canonical collection of $LR(1)$ items,
   
   $$C = \{I_0, \ldots, I_n\}.$$

2. Merge all sets with the same core. Let the new collection be
   
   $$C' = \{J_0, \ldots, J_m\}.$$

3. Construct the action table as before.

4. If $J = I_1 \cup \ldots \cup I_k$, then $\text{goto}(J, X) = \text{union of all sets with the same core as } \text{goto}(I_1, X)$.
SLR vs LR(1) vs LALR

No. of states: \( SLR = LALR \leq LR(1) \) (cf. Pascal)

Power: \( SLR < LALR < LR(1) \)

SLR vs. LALR: LALR items can be regarded as SLR items, with the core augmented by appropriate subsets of \( FOLLOW(A) \) explicitly specified

LALR vs. LR(1):

1. If there were no shift-reduce conflicts in the \( LR(1) \) table, there will be no shift-reduce conflicts in the \( LALR \) table.

2. Step 2 may generate reduce-reduce conflicts.
   
   Example: \( I_1 = \{ \langle A \rightarrow \alpha \cdot, a \rangle, \langle B \rightarrow \beta \cdot, b \rangle \} \)
   \( I_2 = \{ \langle A \rightarrow \alpha \cdot, b \rangle, \langle B \rightarrow \beta \cdot, a \rangle \} \)

3. Correct inputs: LALR parser mimics LR(1) parser
   Incorrect inputs: incorrect reductions may occur on a lookahead \( a \Rightarrow \) parser goes back to a state \( I_i \) in which \( A \) has just been recognized. But \( a \) cannot follow \( A \) in this state \( \Rightarrow \) error
Error recovery

Detection:
Canonical $LR$ - errors are immediately detected (no unnecessary shift/reduce actions)
$SLR/LALR$ - no symbols are shifted onto stack, but reductions may occur before error is detected

Panic mode recovery:
1. Scan down the stack until a state $s$ with a goto on a “significant” non-terminal $A$ (e.g. $expr$, $stmt$, etc.) is found.
2. Discard input until a symbol $a$ which can follow $A$ is found.
3. Push $A$, goto$(s, A)$ and resume parsing.

ExpIn: $s \equiv \alpha \cdot Aa\beta \quad \Rightarrow \quad \alpha \underbrace{\gamma}_{\text{location of error}} a\beta$
Phrase-level error recovery: recovery by local correction on remaining input e.g. insertion, deletion, substitution, etc.

Scheme:
1. Consider each blank entry, and decide what the error is likely to be.
2. Call appropriate recovery method
   - should consume input (to avoid infinite loop)
   - avoid popping a “significant” non-terminal from stack

Examples:

State: $E' \rightarrow \cdot E$
Input: + or *
Action: push id, goto appropriate state
Message: missing operand

State: $E \rightarrow E + \cdot T$
Input: )
Action: skip ’)’ from input
Message: extra ’)’
Usage:  $ yacc myfile.y  (generates y.tab.c)

File format:
  declarations
  %%
  grammar rules (terminals, non-terminals, start symbol)
  %%
  auxiliary procedures (C functions)

Semantic actions:
- $$ - attribute value associated with LHS non-terminal
- $i - attribute value associated with i-th grammar symbol on RHS
- specified action is executed on reduction by corresponding production
- default action: $$ = $1
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double /* double type for Yacc stack */
%
}

%token NUMBER
%left '+' '-'
%left '*' '/'
%right UMINUS

%
lines : lines expr newline { printf("%g\n", $2); }
   | lines newline
comment : /* $ */
   | ;
expr : expr '+' expr { $$ = $1 + $3; }
   | expr '-' expr { $$ = $1 - $3; }
   | expr '*' expr { $$ = $1 * $3; }
   | expr '/' expr { $$ = $1 / $3; }
   | '(' expr ')' { $$ = 2; }
   | '-' expr %prec UMINUS { $$ = - $2; }
   | NUMBER
   ;

%
yylex() {
    int c;
    while ( ( c = getchar() ) == ' ' );
    if ( (c == '.') || (isdigit(c)) ) {
        ungetc(c, stdin);
        scanf("%f", &yyval);
        return NUMBER;
    }
    return c;
}
Lexical analyzer: yylex() must be provided

- should return integer code for a token
- should set yylval to attribute value

Usage with lex:

% lex scanner.l  
% yacc parser.y  
% cc y.tab.c -ly -ll

Add #include "lex.yy.c" to third section of file

Declared tokens can be used as return values in scanner.l
Implicit conflict resolution:
1. Shift-reduce: choose shift
2. Reduce-reduce: choose the production listed earlier

Explicit conflict resolution:
- **Precedence**: tokens are assigned precedence according to the order in which they are declared (lowest first)
- **Associativity**: left, right, or nonassoc
- Precedence/assoc. of a production $= $ precedence/assoc. of rightmost terminal or explicitly specified using \texttt{%prec}
- Given $A \rightarrow \alpha \cdot$, $a$:
  - if precedence of production is higher, reduce
  - if precedence of production is same, and associativity of production is left, reduce
  - else shift