Notes, packet 2  Ch. 4

CFG : context-free grammar

Context-free language

Finite terminal alphabet of tokens \( \Sigma \)
Lower-case words, like \( \text{then} \)
Assign

Finite non-terminal alphabet \( \mathcal{N} \)
Like "variables" \( \sim \) \( \text{initial-\text{op}} \)

Start symbol \( S \in \mathcal{N} \)

Productions \( A \rightarrow X_1 \ldots X_m \)
\( \begin{align*}
&\text{non-terminal} \quad \mathcal{N} \cup \Sigma \\
\end{align*} \)

Allow \( I \) (alternation)

\( A \rightarrow ab \quad \text{\parallel} \quad A \rightarrow ab \text{ } \text{c} \text{ } \text{d} \)
\( A \rightarrow cd \)

Derivation: \( S \Rightarrow D \quad S \Rightarrow \ldots \Rightarrow \)

(only terminals)

\( S \Rightarrow \) \[ \quad \]

Conventions: Leftmost derivation: always expand the leftmost non-terminal in the current sentential form. \( \Rightarrow \text{lm} \)
Rightmost derivation: expands rightmost non-terminal = canonical derivation. Bottom-up parsing: parse discovered in reverse order.

Parse tree: describes a derivation

```
S
├── D
│   └── St
```

Non-terminals

Terminals [i a + b b = ...]

Both leftmost, rightmost derivations yield same parse tree.
Phrase (all terminals descending from a non-terminal)

Complexity of parsing in tokens: \( \mathcal{O}(n^3) \)

Useful CFG's (actual ones for programming langs) can be parsed in \( \mathcal{O}(n) \).

Unreduced grammar: useless non-terminals never generated by any sentence.

```
S \rightarrow A
    \rightarrow B
A \rightarrow a
B \rightarrow B b
C \rightarrow a
```

Want reduced grammars. Unreduced grammars do not naturally occur.
Ambiguous grammar: a single sentence has more than one parse tree. Does naturally occur.

Unavoidable for it.

\[ E \rightarrow E - E \]
\[ E \rightarrow \text{id} \]

Sentence: \( a - b - c \)

BNF (Backus-Naur Form for writing CFG)

metacharacters

\[ [ ] \] to surround optional parts.

\[ \_\_\_ \] to surround parts with implicit Kleene *.

Easily converted to ordinary BNF.

\[ A \rightarrow B[\_c\_] \quad || \quad A \rightarrow B \]
\[ A \rightarrow BC \]
Parser vs. Recognizer vs. Generator
- builds a tree
- returns Boolean
- creates a valid sentence

Bottom-up vs. Top-down parsers

LR(1)
- rightmost

LL(1)
- input in order
- leftmost parse

LL(1): recursive descent

Analyze the BNF:
- Empty: Rule Derives Empty (p)
  Symbol Derives Empty (N)
- First (N)
- Follow (N)
- Predict (p)
Empty check

1) make a list $L$ of $N_s$ that directly derive $\lambda$.

2) For each $N \in L$, for all rules where $N$ is in the RHS, remove $N$ from RHS, if RHS is now empty, put LHS into $L$.

(use a count of RHS elements to make this step easy)

3) $L$ is the set of $N_s$ that derive $\lambda$. The rules that cause us to put all into $L$ are the rules that can derive $\lambda$.

Programming hint: avoid duplicates in $L$.

First($\alpha$) = $\{ b \in \Sigma \mid \alpha \Rightarrow^* b\beta \}$

String of non-terminals and terminals

Hint: if BNF is written top-down: work from the end to the beginning.

Method: 1) Consider first element $\alpha$ of $\alpha$.

   if $\alpha = \lambda$, answer is $\emptyset$

   if $\alpha \in T$ (terminal), answer is $\{ \alpha \}$

2) if $\alpha \in N$,

   for all productions with $\alpha$ as the LHS, recursively compute First(LHS),

   answer = union of all these computations.
\[ E \rightarrow \text{p} \解释{E} \]
\[ \text{p} \rightarrow f \]
\[ T \rightarrow + E \]

\text{First}(\text{p}) = \{ f, \}
\text{First}(\text{f}) = \{ f, \}
\text{First}(\text{+}) = \{ f, \}
\text{First}(\text{E}) = \{ f, \}
\text{First}(\text{f} + E) = \{ f, \}
\text{First}(\text{+} E) = \{ f, \}
\text{First}(\text{E} +) = \{ f, \}
\text{First}(\text{E} + \text{f}) = \{ f, \}
\text{First}(\text{f} + E) = \{ f, \}
\text{First}(\text{+} E) = \{ f, \}
\text{First}(\text{E} +) = \{ f, \}
\text{First}(\text{E} + \text{f}) = \{ f, \}

\text{Follow}(\text{N}) = \text{set of terminals that can come directly after } \text{N} \text{ in a sentential form.}

\[ \{ b \in \sum \mid S \Rightarrow^* \alpha \text{ N b} \beta \} \]

\text{Method: Follow(N)}

\text{for each place } \text{N} \text{ appears in a RHS of production } P

\text{answer } U = \text{First}(\text{tail})

\text{if } \text{tail} \Rightarrow \lambda

\text{answer } U = \text{Follow(LHS}(P))
Top-down parsing (LL)

1) Not as powerful as bottom-up.
   CELs that cannot be parsed top-down
2) Simple, fast, good diagnostics
3) If CFL is LL, it is unambiguous.
4) Recursive descent, table driven.
5) Recursive descent parser:

   One procedure for each non-terminal \( N \)
   That procedure has a separate case
   for each production with \( N \)
   on LHS.
   Which case to use is determined by
   lookahead and predict set.
   Each case is composed of calls to
   match() for terminals in RHS
   procedure for a non-terminal in RHS

---

**Predict set for a production \( P \)**

At least \( \text{First}(\text{RHS}(P)) \)

If \( P \) can derive empty, then also \( \text{Follow}(\text{LHS}(P)) \)

---

A C example:

```c
proc P (Stream <token> ts) {
    token t = ts.peek();
    switch(t) {
        case .; i; d; p; $3; $5; .; match($3);
        default: error
    }
}
```
Proc Ps ( ... )

token t = ts, peek();
switch (t) {
  case $f, $? : D(is); Ds(ts); break;
  case $id, p, $3 : break;
  default: error();
}

Proc E(...) {
  token t = ts, peek();
  switch (t) {
    case $+, $? : match(+); V(ts); E(ts); break;
    case $-, $? : match(-); V(ts); E(ts); break;
    case $id, p, $3 : break;
    default: error();
  }
  E();
}
proc A()
switch (peek())
    case $3$: match (a); procB(); procC(); match (d); break;
    case $4$: break;
    default: error(); break;

Table-driven:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>Q</td>
<td>9</td>
<td>8</td>
<td>9</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Stack: S A B $

Input: $ c A $
Algorithm:

stack. initialized to start non-terminal

while stack not empty:
    t = pop(s);
    if t is terminal:
        match(t)
    else: // t is non-terminal
        p = Table[t, peek(c)]
        if p = 0:
            error()
        else:
            push RHS(p) in reverse order

Not all CFGs are LL(1)
increasing lookahead may help, may not
mod 1: factor out common prefixes

P(56) 1 S \rightarrow \text{if } E \text{ then } Ss \text{ end}
2 \quad \text{if } E \text{ then } Ss \text{ else } Ss \text{ end}
3 Ss \rightarrow Ss ; S
4 S \rightarrow S
5 E \rightarrow \text{id} + E
6 E \rightarrow \text{id}
7 X \rightarrow \text{end}
8 Y \rightarrow \text{else } Ss \text{ end}