# Syntactic Analysis: Top-down parsing

# Main ideas

- Parsing: check for grammatical correctness and determine a sentence's phrase structure
- Formal approaches to describing syntax
  - Recognizer
  - Generators
- Study derivation process to find a way to synchronize the derivation steps with a scan through the token string
- Predictive recursive-descent parsing: an important variation of top-down parsing (simple, effective)
- Requirements for a predictive recursive-descent parser:
  - Unambiguous grammar
  - LL(1): remove left-recursion, left-factoring, first/follow sets

# Top-down parsing

- LL parsing: parse the input scanning tokens from Left to right, doing a Leftmost derivation.

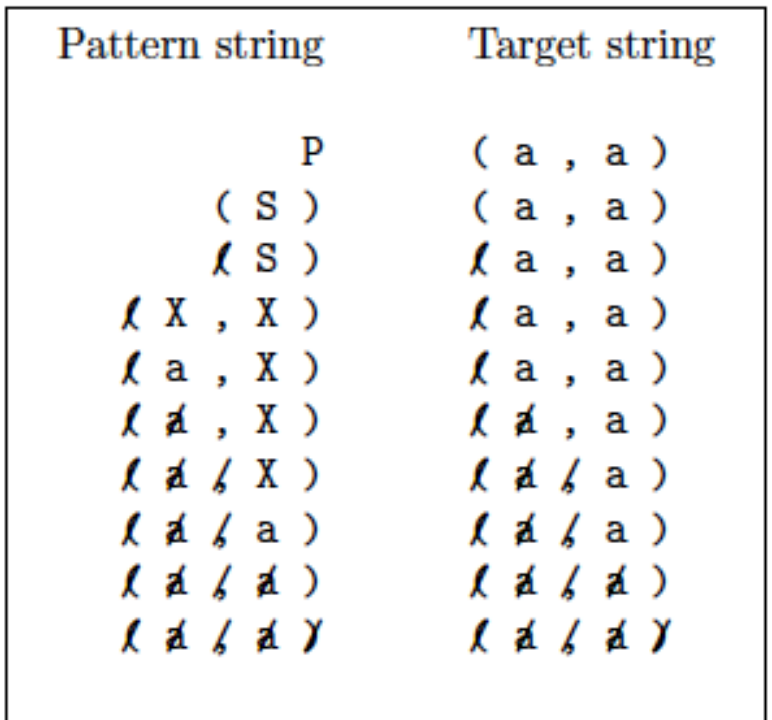
Technique: try to match pattern (from grammar rules) with target string

- Begin current pattern with the start symbol
- Pattern starts with a **non-terminal**?
  - Replace it with the right-hand side of its grammar rule
  - Can require backtracking if we expand the wrong right-hand side!!
- Pattern starts with a **terminal**? Check if it matches the next token on the target string.
  - Yes? **Consume** the token and remove the terminal from the pattern.
  - No? There is an error.
- If both the pattern and target strings are empty, then the parse succeeds.

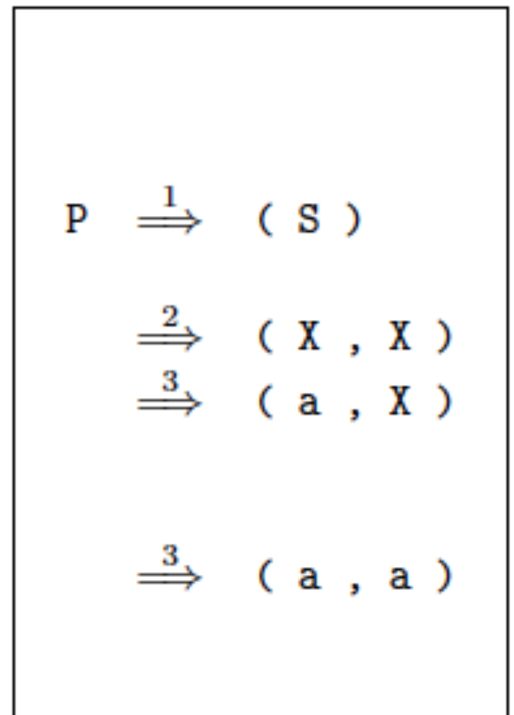
# Top-down parse of string: (a,a)

| | | | | |
|---|---|---|---|---|
| 1 | P | → | ‘(’ S ‘)’ |
| 2 | S | → | X ‘,’ X |
| 3 | X | → | ‘a’ |

Grammar rules




(a) Top-down parse




(b) Derivation

# LL(0) and LL(1)

**LL(0)**

```
1 P → '(' S ')'
2 S → X ',' X
3 X → 'a'
```

- Each non-terminal has a single production.
- No lookahead needed to know which production to apply.

**LL(1)**

```
1 P → '(' S ')'
2 S → X ',' X
3 X → 'a'
4   → 'b'
```

- Some non-terminals have multiple productions.
- A lookahead is needed to know which production to apply

LL(k) – looks ahead **k** tokens.

A top-down parser is also referred to as a predictive parser because there's the possibility of having to predict which of multiple rules to apply by doing a lookahead.

# Recursive-descent parsing

- A top-down strategy
- Each **non-terminal** N in the grammar is implemented as a method `parseN()`
  - Method is responsible for parsing a single N-phrase (a right-hand side for a non-terminal N)
  - Decides what to do next based on its understanding of the grammar and the value of the current token
- Requires **backtracking** if we follow a false trail

# Example: micro-English

| | | |
|---|---|---|
| Sentence | ::= | Subject Verb Object . |
| Subject | ::= | me \| a Noun \| the Noun |
| Object | ::= | me \| a Noun \| the Noun |
| Noun | ::= | cat \| mat \| rat |
| Verb | ::= | like \| is \| see \| sees |

```
private void parseSentence() {     //  Sentence ::=
  parseSubject();                  //     Subject
  parseVerb();                     //     Verb
  parseObject();                   //     Object
  accept( '.' );                   //     .
}
```

# Predictive Recursive Descent Parsing

- Want to avoid backtracking; requires *knowing* which production rule to apply next
- Given the current symbol ***a***, the non-terminal ***A*** to be expanded, and alternatives of production $A \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_n$, which is the unique alternative that derives a string beginning with ***a***?
- Key to this is having an LL(1) grammar: Left to right scan of input symbols doing a Leftmost derivation using 1 symbol of lookahead

First and follow sets:

- First set: what terminals can begin strings derivable from some terminal A
- Follow set: what terminals can immediately follow some terminal A

# What makes a grammar LL(1)?

For a grammar to be LL(1), we have the following requirements for every pair of productions $A \rightarrow \alpha |\ \beta$

- $First(\alpha) - \{\epsilon\}$ and $First(\beta) - \{\epsilon\}$ must be disjoint
- If $\alpha$ is nullable (goes to $\epsilon$), then $First(\beta)$ and $Follow(A)$ must be disjoint.

Remember:

- *First* set: what terminals can begin strings derivable from some terminal A
- *Follow* set: what terminals can immediately follow some terminal A

# Generating the First Sets

- $First(\epsilon) = \{\epsilon\}$
- $First(t) = \{t\}$ where $t$ is a terminal symbol
- $First(X\ Y) = First(X) \cup First(Y)$ if X generates $\epsilon$
- $First(X\ Y) = First(X)$ if X does not generate $\epsilon$
- $First(X \mid Y) = First(X) \cup First(Y)$
- $First(X^*) = First(X)$

# First Set Example

Generate the First set for the following grammar

$A \rightarrow BDi \mid D$

$B \rightarrow Ca \mid \epsilon$

$C \rightarrow b$

$D \rightarrow c$

Do as an exercise.

- $First(\epsilon) = \{\epsilon\}$
- $First(t) = \{t\}$ where $t$ is a terminal symbol
- $First(X\ Y) = First(X) \cup First(Y)$ if X generates $\epsilon$
- $First(X\ Y) = First(X)$ if X does not generate $\epsilon$
- $First(X \mid Y) = First(X) \cup First(Y)$
- $First(X^*) = First(X)$

# Generating the Follow sets

- Place $\$$ in $Follow(S)$ where $S$ is the start symbol and $\$$ is the input right end marker

- If there is a production $A \rightarrow \alpha B \beta$, then everything in $First(\beta)$ except for $\epsilon$ is placed in $Follow(B)$

- If there is a production $A \rightarrow \alpha B$ or a production $A \rightarrow \alpha B \beta$ where $First(\beta)$ contains $\epsilon$, then everything in $Follow(A)$ is in $Follow(B)$

# Follow Set Example

Generate the Follow set for the following grammar

$A \rightarrow BDi \mid D$

$B \rightarrow Ca \mid \epsilon$

$C \rightarrow b$

$D \rightarrow c$

Do as an exercise.

- Place \$ in $Follow(S)$ where $S$ is the start symbol and \$ is the input right end marker
- If there is a production $A \rightarrow \alpha B \beta$, then everything in $First(\beta)$ except for $\epsilon$ is placed in $Follow(B)$
- If there is a production $A \rightarrow \alpha B$ or a production $A \rightarrow \alpha B \beta$ where $First(\beta)$ contains $\epsilon$, then everything in $Follow(A)$ is in $Follow(B)$

# Left Recursive Grammars

- A grammar is *left-recursive* if it has a non-terminal $A$ such that there is a derivation $A \to A\alpha$ for some string $\alpha$

- Bad for a recursive decent parser - why?
- Consider the parse method for $A \to ABc$

```
private void parseA() {    //  A::=
  parseA();                //     A
  parseB();                //     B
  accept( 'c' );           //     c
}
```

- Need to eliminate the left recursion

$$A \to A\alpha \mid \beta \quad \Rightarrow \quad \begin{array}{l} A \to \beta A' \\ A' \to \alpha A' \mid \varepsilon \end{array}$$

# Eliminate Left-Recursion

- What's wrong with the following?

Command ::= single-Command | Command; single-Command

- Look at the First sets produced:

$First$(Command) = { Identifier, if, while, let, begin}

$First$(single-Command) = {Identifier, if, while, let, begin }

- Eliminate the left recursion to produce:

Command ::= single-Command (; single-Command)*

# Left-Recursion Example

Eliminate the left-recursion in the following grammar

(1) $E \rightarrow E + T \mid T$

(2) $T \rightarrow T * F \mid F$

(3) $F \rightarrow (E) \mid$ Identifier

$$A \rightarrow A\alpha \mid \beta \quad \Rightarrow \quad \begin{array}{l} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' \mid \varepsilon \end{array}$$

(1) $E \rightarrow TP$

(2) $P \rightarrow + TP \mid e$

(3) $T \rightarrow FQ$

(4) $Q \rightarrow * FQ \mid e$

(5) $F \rightarrow (E) \mid$ Identifier

Now generate the First and Follow sets.

Is the grammar LL(1)?

# Do Left-factoring

- Consider the following:

```
stmt ::= if expr then stmt else stmt
       | if expr then stmt
```

- When the parser receives the **if** token, it does not know which alternative to select
- Rewrite the grammar to eliminate the confusion
- How is this different from left-recursion?