Context-Free Parsing

Outline

- Parsing Overview
- Recursive Descent
- LL(1) Parsing
  - LL(1) Overview
  - First and Follow Sets
  - LL(1) Parsers

Parsing Overview

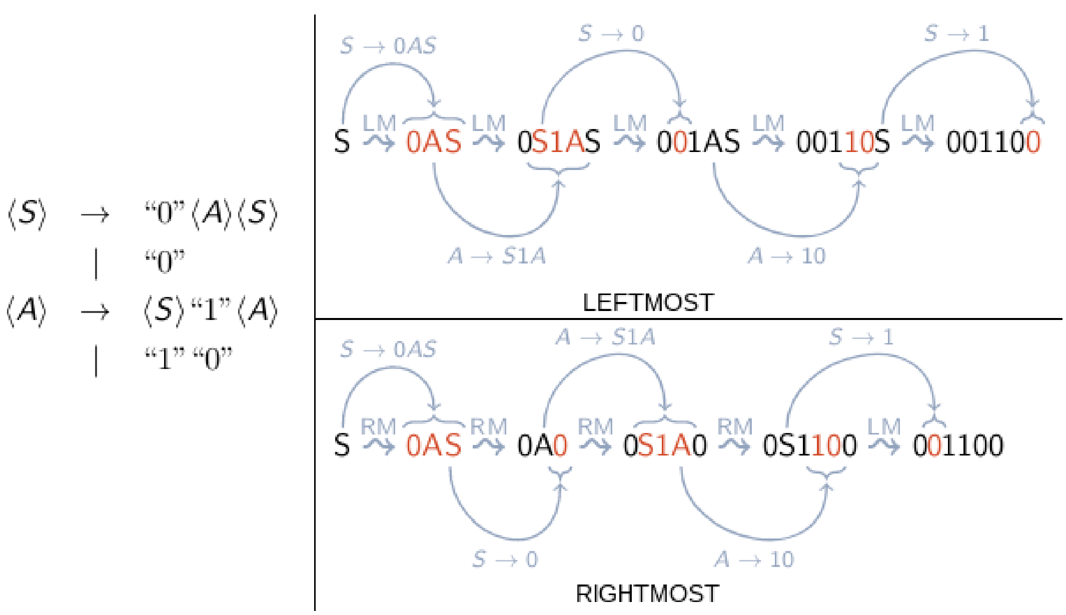
Parsing Problem

- Given: Context-free grammar G
- Find: A program to recognize $L(\text{G})$
- Approach: Construct a derivation from the start symbol to the input string. Many variations, and corresponding trade-offs, on how to do this!

Leftmost and Rightmost Derivations

Leftmost Derivation: at each step in the derivation, apply a production for the **leftmost** nonterminal

Rightmost Derivation: at each step in the derivation, apply a production for the **rightmost** nonterminal.



Common Parsing Algorithms

	Algorithm	Capability	Runtime
Top-Down	Recursive Descent	Most Prog. Constructs	varies
	LL(1) (Lewis & Stearns ’68)	Most Prog. Constructs	O(n)
	LL(*) (Parr 2012)	All LL(k), some non-CFLs	O(n^2) worst, often better
Bottom-Up	LR (Knuth ’65)	Deterministic CFL’s (DCFL)	O(n)
	LALR (DeRemer ’69)	Most (useful) DCFLs	O(n), less memory than LR
	GLR (Lang ’74, Tomita ’84)	All DCFL, some NCFL	O(n^3) worst, often better
Dyn. Prog.	CYK (Younger ’67)	Any CFL	O(n^3)
	Earley ’70	Any CFL	O(n^3)

Recursive Descent

Recursive Descent Parsing:

- Set of mutually recursive procedures
- One procedure for each nonterminal
- Each procedure:
  - pick a production for the nonterminal
  - recursively calls procedures for the RHS

Procedure A

```
Choose a production for A → X1 X2 ... Xk;
for i = 1 to k do
    if Xi is a nonterminal then
        call procedure Xi();
    else if Xi is the current input symbol then
        read the next input symbol;
    else
        ERROR;
```

Recursive Descent Summary

Pros:

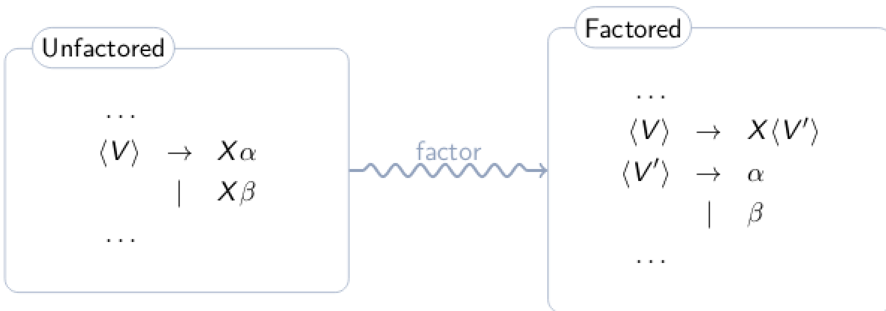
- Usually slightly faster than automatically generated parsers
- Can work-around non-context free language parts (eg. typedef’ed specifier vs identifier/variable name in C)

Cons:

- Must manually write parser based on grammar

Factoring Grammars

Unfactored grammar: a nonterminal with multiple production rules that share the same start symbol.



LL(1) Parsing

LL(1) Overview:

- Algorithmically construct recursive descent-like parsers
- LL(1) Meaning
  - L = Left to right scan of string
  - L = Leftmost derivation
  - (1) = One terminal symbol of lookahead: pick the next derivation step (production) by looking at only one nonterminal
- Not all grammars are LL(1)

First Set

First Set: the first set FIRST(α) is the set of terminals that may begin derivations of α

- Given: CFG G = (V, T, P , S)
- Find: Function FIRST: $(V \cup T)^* \mapsto \mathcal{P}(T)$, where
  - FIRST(α) is the set of terminals that may begin derivation of α
  - FIRST(α) = $\{c \in T \mid a \rightsquigarrow c\gamma\}$

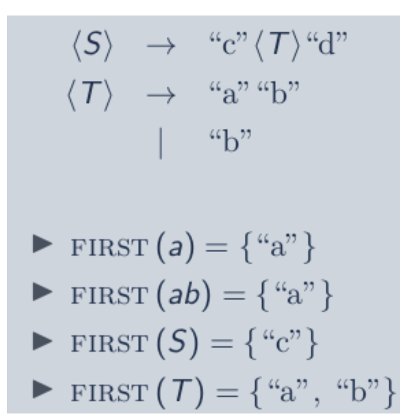


Figure 1: Example of first set

First Set Algorithm

Algorithm 1: FIRST-Symbol

```
Input: G = (V, T, P, S)
/* Terminals */
FIRST(ε) ← {ε};
foreach σ ∈ T do
    FIRST(σ) ← {σ};
/* Nonterminals */
repeat
    foreach (A → α_0 ... α_n) ∈ P do
        i ← 0;
        repeat
            FIRST(A) ← FIRST(A) ∪ (FIRST(α_i) \ ε);
            i ← i + 1;
        until (i < n) ∨ (ε ∉ α_i);
        if i = n then FIRST(A) ← FIRST(A) ∪ (ε) ;
until fixpoint;
```

Follow Set

Follow Set: the follow set FOLLOW(α) is the set of terminals that come after (follow) α is some derivation.

- Given: CFG G = (V, T, P, S)
- Find: Function FOLLOW: V $\mapsto$ P(T), where
  - FOLLOW(A) is the set of terminals that may appear to the right of (following) A during some derivation for G
  - FOLLOW(A) = {c $\in$ T | S $\rightsquigarrow$ αAc$\beta\gamma$ }

$$
\begin{aligned}
\langle S \rangle &\rightarrow \text{"c"} \langle T \rangle \text{"d"} \\
\langle T \rangle &\rightarrow \text{"a"} \text{"b"} \\
&\;|\;\; \text{"b"}
\end{aligned}
$$

- $\textsc{follow}(S) = \{\$\}$
- $\textsc{follow}(T) = \{d\}$

Figure 2: Example of Follow set

Follow Set Algorithm

Algorithm 2: Follow Set

Input: $G = (V, T, P, S)$

$\text{FOLLOW}(S) \leftarrow \{\$\}$;

forall $A \in (V \setminus \{S\})$ **do** $\text{FOLLOW}(A) \leftarrow \emptyset$;

forall $(A \rightarrow \alpha B \beta) \in P$ **do**

  $\text{FOLLOW}(B) \leftarrow \text{FOLLOW}(B) \cup (\text{FIRST}(\beta) \setminus \{\varepsilon\})$;

repeat

  forall $(A \rightarrow \alpha B) \in P$ **do**

    $\text{FOLLOW}(B) \leftarrow \text{FOLLOW}(A) \cup \text{FOLLOW}(B)$;

  forall $(A \rightarrow \alpha B \beta) \in P$ **do**

    if $\varepsilon \in \text{FIRST}(\beta)$ **then**

      $\text{FOLLOW}(B) \leftarrow \text{FOLLOW}(A) \cup \text{FOLLOW}(B)$;

until *fixpoint*;

LL(1) Requirements

Whenever A $\rightarrow \alpha \mid \beta$:

1. α and β cannot derive strings starting with same terminal:

   $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$

2. At most one of α and β derives ϵ

   $\neg((\alpha \rightsquigarrow \epsilon) \wedge (\beta \rightsquigarrow \epsilon))$

3. WLOG, if $\beta \rightsquigarrow \epsilon$, then α does not derive any string beginning with FOLLOW(A):

   $(\beta \rightsquigarrow \epsilon \implies (\text{FIRST}(\alpha) \cap \text{FOLLOW}(A) = \emptyset)$

Trick Cases

FILL THIS SECTION WHEN YOU'RE REVIEWING FOR THE FINAL

Parsing Table

Given: CFG G = (V, T, P, S)

- Current nonterminal $A \in V$
- Next input symbol $\sigma \in T$

Find: Which production $p \in P$ to use for A:

- $M: V \times T \mapsto \{P \cup \{\emptyset\}\}$

Grammar

$$\langle S \rangle \rightarrow \text{“c”} \langle T \rangle \text{“d”}$$
$$\langle T \rangle \rightarrow \text{“a”} \langle U \rangle$$
$$\langle U \rangle \rightarrow \text{“b”} \mid \varepsilon$$

Sym.	First	Follow
$\langle S \rangle$	“c”	\$
$\langle T \rangle$	“a”	“d”
$\langle U \rangle$	“b”, ε	“d”

Table

Nonterm.	Input Symbol			
	“a”	“b”	“c”	“d”
$\langle S \rangle$	$\emptyset$	$\emptyset$	$S \rightarrow cTd$	$\emptyset$
$\langle T \rangle$	$T \rightarrow aU$	$\emptyset$	$\emptyset$	$\emptyset$
$\langle U \rangle$	$\emptyset$	$U \rightarrow b$	$\emptyset$	$U \rightarrow \varepsilon$

Figure 3: Example of constructing parsing table

Algorithm 3: LL(1) Parsing Table Construction

Input: $G = (V, T, P, S)$
Output: $M : V \times T \mapsto \{P \cup \{\emptyset\}\}$

foreach $(A \rightarrow \alpha) \in P$ **do**
  /* Select productions based on first sets. */
  /* A terminal in first of α, means use this production. */
  foreach $b \in \text{FIRST}(\alpha)$ **do**
    $M(A, b) = (A \rightarrow \alpha)$;
  /* Consider follow sets when the RHS can be empty. */
  /* A terminal in follow of α, means α derives ε. */
  if $\varepsilon \in \text{FIRST}(\alpha)$ **then**
    foreach $b \in \text{FOLLOW}(A)$ **do**
      $M(A, b) = (A \rightarrow \alpha)$;

Predictive Parsing Overview

Predictive Parsing

- Given: Grammar, Parsing Table, Input string
- Find: A parse of the input string
- Approach: Store partial derivations on a stack and expand based on the parsing table:

    1. Push the start symbol
    2. While the stack is not empty:

        2.1 Pop the top of the stack

        2.2 If a terminal, match with the next input symbol and pop

        2.3 If a nonterminal, select expansion from the parsing tale according to the next input symbol and push the RHS

    3. Accept if we are at the end of the input string. Otherwise, reject.

Algorithm 4: Predictive Parser

Input: $G = (V, T, P, S)$ // Grammar
Input: $M : V \times T \mapsto \{P \cup \{\emptyset\}\}$ // Parsing Table
Input: $\omega \in T^*$ // String
$i \leftarrow 0$; // string index
$\Phi \leftarrow (S)$; // Start symbol on stack
while Φ **do**
    $\sigma \leftarrow \text{pop}(\Phi)$;
    if $\sigma = \omega_i$ **then** $i \leftarrow i + 1$; // current symbol is next terminal in ω
    else if $\sigma \in T$ **then return** reject ; // unexpected next terminal in ω
    else if $\emptyset = M(\sigma, \omega_i)$ **then return** reject ; // No table entry for current symbol on ω_i
    else // Push a new production onto the stack
        $M(\sigma, \omega_i) = \sigma \rightarrow Y_0 \ldots Y_k$;
        push $Y_k, \ldots, Y_0$ onto Φ, with Y_0 on top;
if $\$ = \omega_i$ **then return** accept; // end of string
else return reject; // More terminals in string

Summary & Considerations

Selecting a parsing algorithm/parser generator:

- Language support
- Capabilities: what constructs are easily handled
- Performance of generated parser

Writing the grammar:

- Parser capabilities: modify grammar to suit
- Efficiency: tailor grammar to parsing algorithm
- Ambiguity: eliminate in grammar (LL(1)) or specify precedence (LALR)
- Cannot fully automate grammar construction