Top-down parsing

- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal

```
  E
 /\   \
T   +
 /\   \
int * int + int
```

Top-down parsing II

- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal

- The leaves at any point form a string $\beta \Delta \gamma$
  - $\beta$ contains only terminals
  - The input string is $\beta b \delta$
  - The prefix $\beta$ matches
  - The next token is $b$

```
  E
 /\   \
T   +
 /\   \
int * int + int
```

Top-down parsing III

- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal

- The leaves at any point form a string $\beta \Delta \gamma$ ($A=T, \gamma=\epsilon$)
  - $\beta$ contains only terminals
  - $\gamma$ contains any symbols
  - The input string is $\beta b \delta$ ($b=int$)
  - So $A\gamma$ must derive $b \delta$
Top-down parsing IV

- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal

• So choose production for T that can eventually derive something that starts with int

LL(k) parsing

Current sentential form: int * T + E

Look-ahead (1): int
Look-ahead (2): int +
Look-ahead (3): int + int

LL(1) parser: determines next production in leftmost derivation, looking ahead by one terminal
Key question: How do we choose the next production systematically?

Overview

- We will focus on LL(1) parsers.
- Generalization: LL(k) parsers

LL(1) parsers require three sets called
  - nullable
  - FIRST
  - FOLLOW

Given these sets, you can write down a recursive-descent parser

Simplified
  - nullable and FOLLOW are only required if the grammar has ε productions

Game plan
  - start with grammars without ε productions (we saw this informally)
  - then add ε productions
  - end with an iterative, stack-based implementation of top-down parsing

Example 1

- Restriction on grammar:
  - for each non-terminal
    - productions begin with terminals
    - no two productions begin with same terminal
  - so no ε productions

- Algorithm for parsing:
  - one procedure for each non-terminal
  - In each procedure, peek at the next token to determine which rule to apply

Example:

S → id := E | if E then S else S | while E do S

procedure S
  case peekAtToken() of
    id : match(id); match(:=); E; break;
    if : match(if); E; match(then); S; match(else); S; break;
    while : match(while); E; match(do); S; break;
    otherwise error
**LL(1) Parsing Table**

<table>
<thead>
<tr>
<th>T</th>
<th>b</th>
<th>E</th>
<th>if</th>
<th>then</th>
<th>else</th>
<th>do</th>
<th>while</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>=</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>=</td>
<td>if</td>
<td>E</td>
<td>then</td>
<td>S</td>
<td>else</td>
<td>S</td>
</tr>
<tr>
<td>while</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Consider the T[S, if] entry
- Means “When current non-terminal is S and next input token is “if”, use production $S \rightarrow \text{id } E \text{ if } E \text{ then } S \text{ else } S \text{ while } E \text{ do } S$
- Given this table, we can construct the recursive code trivially.
- How do we generate parsing tables automatically?

**FIRST sets**

- **FIRST**: non-terminal $\rightarrow$ subset of terminals
  - $b \in \text{FIRST}(N)$ if $N \rightarrow \ast b \delta$
- **Construction**:
  - for each non-terminal $A \rightarrow \gamma$, add constraint $\gamma \in \text{FIRST}(A)$
  - find smallest sets that satisfy all constraints
- For our example grammar,
  - set of terminals = {id, :=, if, then, else, while, do}
  - Constraints:
    - id $\in \text{FIRST}(S)$
    - if $\in \text{FIRST}(S)$
    - while $\in \text{FIRST}(S)$
  - There are many sets that satisfy these constraints
    - (eg) {id, if, while}, {id, if, while, :=}, {id, if, while, do, :=}, ...
  - We want the smallest set that satisfies all constraints
    - $\text{FIRST}(S) = \{\text{id, if, while}\}$
  - Extension: it is convenient to extend FIRST to any string $\gamma$
    - $b \in \text{FIRST}(\gamma) \rightarrow \gamma \rightarrow \ast b \delta$

**Constructing Parsing Tables**

- Construct a parsing table $T$ for CFG $G$
- For each production $A \rightarrow \alpha$ in $G$ do:
  - For each terminal $b \in \text{First}(\alpha)$ do
    - $T[A, b] = A \rightarrow \alpha$
- **Conflict**: two or more productions in one table entry
  - Grammar is not LL(1)
  - We may or may not be able to rewrite grammar to be LL(1)

**Example 2**

- Some productions may begin with non-terminal
- **Example**:
  - $S \rightarrow XY \mid YX$
  - $X \rightarrow a b$
  - $Y \rightarrow b a$

  It is clear that we can parse $S$ as follows:

  ```plaintext```
  procedure S
  case peekAtToken() of
  a: X ; Y
  b: Y ; X
  otherwise error
  ```plaintext```
**FIRST sets**

- **Construction**: for each non-terminal A
  - for each rule \( A \rightarrow t \gamma \), \( t \in \text{FIRST}(A) \)
  - for each rule \( A \rightarrow B \gamma \), \( \text{FIRST}(B) \subseteq \text{FIRST}(A) \)

- **For our example, rules give**
  - \( \text{FIRST}(X) \subseteq \text{FIRST}(S) \)
  - \( \text{FIRST}(Y) \subseteq \text{FIRST}(S) \)
  - \( a \in \text{FIRST}(X) \)
  - \( b \in \text{FIRST}(Y) \)

- **If we solve these constraints, we get**
  - \( \text{FIRST}(X) = \{a\} \)
  - \( \text{FIRST}(Y) = \{b\} \)
  - \( \text{FIRST}(S) = \{a,b\} \)

**Constructing Parsing Tables**

- **Same as before**
- **For each production**

| \( A \rightarrow \alpha \) in \( G \) do:
- For each terminal \( t \in \text{First}(\alpha) \)
  - \( T[A, t] = A \rightarrow \alpha \)
\( \varepsilon \) productions

- Non-terminal \( N \) is nullable if \( N \rightarrow \varepsilon \)
- Example:
  \[ S \rightarrow AB\$
  A \rightarrow a \mid \varepsilon 
  B \rightarrow b \]
- When should you use the \( A \rightarrow \varepsilon \) production?
- One solution:
  - Ignore \( \varepsilon \) productions and compute FIRST
  - Table \([A,a]=A \rightarrow a\]
  - all other entries for \( A: A \rightarrow \varepsilon \)
- This is bad practice:
  - errors should be caught as soon as possible
  - what if next input token was \( \$ \)?
- Solution:
  - if we use \( A \rightarrow \varepsilon \) production to derive a legal string, next token in input must be \( b \)
  - if next token is \( b \), use \( A \rightarrow \varepsilon \) production; otherwise report error
- How do we describe this formally?

FOLLOW sets

- FOLLOW: Non-terminal \( \rightarrow \) subset of terminals
- \( b \in \text{FOLLOW}(A) \) if \( S \rightarrow^* \ldots Ab\ldots \)
- To compute FOLLOW(\( A \)), we must look at RHS of productions that contain \( A \)
- Example:
  \[ S \rightarrow AB\$
  A \rightarrow a \mid \varepsilon 
  B \rightarrow b \]
  \( \text{FOLLOW}(B) = \{\$\} \)
  \( \text{FOLLOW}(A) = \text{FIRST}(B) \)
- But \( \varepsilon \) rules change FIRST computation as well!
  - FIRST(S) needs to take into account the fact that \( A \) is nullable
- How do we get all this straight?

Game plan

1. Compute set of nullable non-terminals
2. Use nullable set to compute FIRST
3. Use FIRST to compute FOLLOW
4. Use FIRST and FOLLOW sets to populate LL(1) parsing table

Computing Nullable

- Set up constraints for nullable set of non-terminals as follows:
  - Nullable \( \subseteq \) Non-terminals
  - \( A \rightarrow \varepsilon \)
    - \( A \in \text{Nullable} \)
  - \( A \rightarrow \ldots \)...
    - no constraint
  - \( A \rightarrow BC \ldots M \)
    - if \( B,C, \ldots, M \in \text{Nullable} \), then \( A \in \text{Nullable} \)
- Find least set that satisfies all constraints
Example

\[
\begin{align*}
Z & \rightarrow d & \text{(d) \subseteq FIRST(Z)} \\
Y & \rightarrow \epsilon & \text{no constraint} \\
X & \rightarrow Y & \text{FIRST(Y) \subseteq FIRST(X)} \\
Z & \rightarrow XYZ & \text{FIRST(X) \subseteq FIRST(Z)} \\
Y & \rightarrow c & \text{(c) \subseteq FIRST(Y)} \\
X & \rightarrow a & \text{(a) \subseteq FIRST(X)}
\end{align*}
\]

Solution: 
FIRST(X) = \{a,c\} \\
FIRST(Y) = \{c\} \\
FIRST(Z) = \{a,c,d\}

Computing First Sets

Definition \( \text{First}(X) = \{ b \mid X \rightarrow \gamma \ b \alpha \} \)

1. \( \text{First}(b) = \{ b \} \) for \( b \) any terminal symbol

2. For all productions \( X \rightarrow A_1 \ldots A_n \)
   - \( \text{First}(A_i) \subseteq \text{First}(X) \)
   - \( \text{First}(A_j) \subseteq \text{First}(X) \) if \( A_i \in \text{Nullable} \)
   - \( \ldots \)
   - \( \text{First}(A_n) \subseteq \text{First}(X) \) if \( A_1, \ldots, A_{n-1} \in \text{Nullable} \)

Note: \( X \rightarrow \epsilon \) does not generate any constraint

3. Solve

Example

\[
\begin{align*}
Z & \rightarrow d & \text{no constraint} \\
Y & \rightarrow \epsilon & \text{Y \in Nullable} \\
X & \rightarrow Y & \text{if Y \in Nullable, X \in Nullable} \\
Z & \rightarrow XYZ & \text{if X,Y,Z \in Nullable, Z \in Nullable} \\
Y & \rightarrow c & \text{no constraint} \\
X & \rightarrow a & \text{no constraint}
\end{align*}
\]

So constraints are
- \( Y \in \text{Nullable} \)
- If \( Y \in \text{Nullable} \) then \( X \in \text{Nullable} \)
- If \( X,Y,Z \in \text{Nullable} \) then \( Z \in \text{Nullable} \)

Solution: nullable = \{X,Y\}

Computing Follow Sets

Definition \( \text{Follow}(X) = \{ b \mid S \rightarrow \beta X b \omega \} \)

1. For all productions \( Y \rightarrow X A_1 \ldots A_n \)
   - \( \text{Follow}(A_i) \subseteq \text{Follow}(X) \)
   - \( \text{Follow}(A_j) \subseteq \text{Follow}(X) \) if \( A_i \in \text{Nullable} \)
   - \( \ldots \)
   - \( \text{Follow}(A_n) \subseteq \text{Follow}(X) \) if \( A_1, \ldots, A_{n-1} \in \text{Nullable} \)

Note: \( X \rightarrow \epsilon \) does not generate any constraint

2. Solve.
Example

\[
\begin{align*}
Z & \to d \quad \text{no constraint} \\
Y & \to c \quad \text{no constraint} \\
X & \to Y \quad \text{FOLLOW}(X) \subseteq \text{FOLLOW}(Y) \\
Z & \to X Y Z \quad \text{FIRST}(Z) \subseteq \text{FOLLOW}(X) \quad \text{FIRST}(Z) \subseteq \text{FOLLOW}(Y) \\
Y & \to c \quad \text{no constraint} \\
X & \to a \quad \text{no constraint}
\end{align*}
\]

Solution:

- FOLLOW(X) = \{a, c, d\}
- FOLLOW(Y) = \{a, c, d\}
- FOLLOW(Z) = {} \\

Computing nullable, FIRST, FOLLOW

\[
\begin{align*}
\text{for each symbol } Y & \quad \text{FIRST}(X) = \{a, c, d\}, \text{nullable}[X] = \text{false} \\
\text{for each terminal symbol } t & \quad \text{FIRST}(t) = \{t\} \\
\text{repeat} & \quad \text{for each production } X \to Y_1 Y_2 \ldots Y_k, \text{ if all } Y_j \text{ are nullable then nullable}[X] = \text{true} \\
& \quad \text{if } Y_1 \text{ and } Y_k \text{ are nullable then} \\
& \quad \text{FIRST}(X) = \text{FIRST}(X) \cup \text{FIRST}(Y_1) \cup \text{FIRST}(Y_k) \\
& \quad \text{if } Y_1 \text{ and } Y_k \text{ are nullable then} \\
& \quad \text{FOLLOW}(Y_1) = \text{FOLLOW}(Y_1) \cup \text{FOLLOW}(X) \\
& \quad \text{FOLLOW}(Y) = \text{FOLLOW}(Y) \cup \text{FOLLOW}(X) \\
& \quad \text{until } \text{FIRST}, \text{FOLLOW}, \text{nullable do not change}
\end{align*}
\]

Constructing Parsing Table

- For each production \( A \to \alpha \) in \( G \):
  - For each terminal \( b \in \text{First}(\alpha) \) do
    - \( T[A, b] = \alpha \)
  - If \( \alpha \) is nullable, for each \( b \in \text{Follow}(A) \) do
    - \( T[A, b] = \alpha \)

LL(1) Parsing Table Example

<table>
<thead>
<tr>
<th>( E ) \to T X</th>
<th>( X \to + E \mid \varepsilon )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T ) ( ( E ) \mid \text{int} \ Y )</td>
<td>( (E) )</td>
</tr>
<tr>
<td>( X ) ( + E \mid \varepsilon )</td>
<td>( \varepsilon )</td>
</tr>
<tr>
<td>( Y ) ( \ast T \mid \varepsilon )</td>
<td>( \varepsilon )</td>
</tr>
</tbody>
</table>

- Follow(\( E \)) = \{\( \), \$\}
- Follow(\( X \)) = \{\( \), \$\}
- Follow(\( Y \)) = \{\( + \), \$\}
- Follow(\( T \)) = \{\( + \), \$\}

First(\( T \)) = \{\( \text{int} \), \( \)\}

First(\( E \)) = \{\( \text{int} \), \( \)\}

First(\( X \)) = \{\( \)\}

First(\( Y \)) = \{\( + \)\}

First(\( T \)) = \{\( + \)\}

X and Y are nullable.
Notes on LL(1) Parsing Tables

• If any entry is multiply defined then G is not LL(1). This happens
  – If G is ambiguous
  – If G is left recursive
  – If G is not left-factored
  – *And in other cases as well*

• Most programming language grammars are not LL(1)
• We can produce the recursive parser systematically from the parsing table.

Iterative LL(1) parser

• It is also possible to design an iterative parser that uses an explicit stack and
  – pushes and pops stuff from the stack
  – examines token from input
to decide how to parse the program.
• Useful to study this to make a connection with bottom-up parsing, which are always presented using an iterative parser.

Pushdown automata

• Here's one way of thinking about context-free grammars and parsing
  – write down a "transition diagram" for each production rule that it has transitions
  – the pushdown automaton begins execution with the transition diagram for the start symbol by pushing that state on the stack
  – as long as the states it encounters have transitions labeled with terminals, it behaves just like a real FSA
  – however, when it encounters a transition labeled with a non-terminal (say, A), it pushes the "transition diagram" for A by pushing the start state of that transition diagram on the stack
  – when the transition diagram for A reaches an accepting state, it is popped from the stack and the program continues execution by taking an A transition
  – the string is accepted if the pushdown automaton reaches the end of the input, and the stack only contains the final state of the transition diagram of the start symbol

Transition diagram

• Convenient to label states using productions with dots to show how far parsing has gotten
  – (eg) P→S.S: we have seen S and we are expecting to see a $
Building an iterative LL(1) parser

• Draw dashed arrows as shown to denote the pushdown of state
  – these would have been procedure calls in the recursive code
• Now you can just number the states and perform combinations of
  – eat one token from input
  – push a new state on the pushdown stack
  – topmost transition diagram accepts a substring of input

Summary

• Given an LL(1) grammar, you can
  – generate parsing table for grammar
    • compute NULLABLE, FIRST, FOLLOW
  – write a recursive-descent parser from that table, using template
• LL(1) parser-generator
  – given LL(1) grammar
    • computes NULLABLE, FIRST, FOLLOW sets
    • uses those sets and transition diagram of grammar to produce an iterative parser that maintains an explicit stack
  – examples: ANTLR, JAVACC
Iterative parser

• We can read off the recursive parser from the parsing table.
• We can also use an iterative parser that is driven by the parsing table.
• Advantage:
  – smaller space requirements
  – usually faster