Implementing Parsers

- Many efficient techniques for parsing
  - i.e., turning strings (token lists) into parse trees
  - Examples
    - LL(k), SLR(k), LR(k), LALR(k)...
    - Take CMSC 430 for more details
- One simple technique: recursive descent parsing
  - This is a top-down parsing algorithm
  - Other algorithms are bottom-up

Top-Down Parsing

\[
E \rightarrow \text{id} = n | \{ L \} \\
L \rightarrow E ; L | \epsilon
\]

(Assume: id is variable name, n is integer)

Show parse tree for
\{ x = 3 ; \{ y = 4 ; \} ; \}

Recall: Steps of Compilation

- Lexing
  - Converts strings to tokens
- Parsing
  - Converts tokens to trees
- Intermediate Code Generation
- Optimization
- Compiler
- Source program
- Target program
**Bottom-up Parsing**

\[
E \rightarrow id = n \mid \{ L \\
L \rightarrow E ; L \mid \epsilon
\]

Show parse tree for
\[
\{ x = 3 ; \{ y = 4 ; \} ; \}
\]

Note that final trees constructed are same as for top-down; only order in which nodes are added to tree is different

---

**Example: Shift-Reduce Parsing**

- Replaces RHS of production with LHS (nonterminal)
- Example grammar
  - \( S \rightarrow aA, A \rightarrow Bc, B \rightarrow b \)
- Example parse
  - \( abc \Rightarrow aBc \Rightarrow aA \Rightarrow S \)
  - Derivation happens in reverse
- Something to look forward to in CMSC 430

---

**Tradeoffs**

- Recursive descent parsers: easy to write & fast
  - The formal definition is a little clunky
    - but if you follow the code then it’s almost what you might have done if you weren’t told about grammars formally
  - Implemented with a simple table (and/or recursion)
- Shift-reduce parsers handle more grammars
  - Complicated to write; requires tools
    - yacc, bison, etc. convert a CFG to a shift-reduce parser
  - Error messages may be confusing
- Most languages use hacked parsers (!)
  - Strange combination of the two

---

**Recursive Descent Parsing**

- **Goal**
  - Determine if we can produce the string to be parsed from the grammar’s start symbol
- **Approach**
  - Recursively replace nonterminal with RHS of production
- **At each step, we'll keep track of two facts**
  - What tree node are we trying to match?
  - What is the **lookahead** (next token of the input string)?
    - Helps guide selection of production used to replace nonterminal
Recursive Descent Parsing (cont.)

- At each step, 3 possible cases
  - If we’re trying to match a terminal
    - If the lookahead is that token, then succeed, advance the lookahead, and continue
  - If we’re trying to match a nonterminal
    - Pick which production to apply based on the lookahead
  - Otherwise fail with a parsing error

Parsing Example

\[ E \rightarrow \text{id} = n \mid \{ \text{L} \} \]
\[ \text{L} \rightarrow E \ ; \ L \mid \varepsilon \]

- Here \( n \) is an integer and \( \text{id} \) is an identifier

- One input might be
  \[ \{ x = 3 ; \{ y = 4 ; \} ; \} \]
  - This would get turned into a list of tokens
  \[ \{ x = 3 \ ; \{ y = 4 \ ; \} \ ; \} \]
  - And we want to turn it into a parse tree

Parsing Example (cont.)

\[ E \rightarrow \text{id} = n \mid \{ \text{L} \} \]
\[ \text{L} \rightarrow E \ ; \ L \mid \varepsilon \]

\[ \{ x = 3 ; \{ y = 4 ; \} ; \} \]

Recursive Descent Parsing (cont.)

- Key step
  - Choosing which production should be selected

- Two approaches
  - Backtracking
    - Choose some production
    - If fails, try different production
    - Parse fails if all choices fail
  - Predictive parsing
    - Analyze grammar to find FIRST sets for productions
    - Compare with lookahead to decide which production to select
    - Parse fails if lookahead does not match FIRST
First Sets

Motivating example
• The lookahead is \( x \)
• Given grammar \( S \rightarrow xyz \mid abc \)
  \( \rightarrow \) Select \( S \rightarrow xyz \) since 1st terminal in RHS matches \( x \)
• Given grammar \( S \rightarrow A \mid B \quad A \rightarrow x \mid y \quad B \rightarrow z \)
  \( \rightarrow \) Select \( S \rightarrow A \), since \( A \) can derive string beginning with \( x \)

In general
• Choose a production that can derive a sentential form beginning with the lookahead
• Need to know what terminal may be first in any sentential form derived from a nonterminal / production

First Sets

Definition
• \( \text{First}(y) \), for any terminal or nonterminal \( y \), is the set of initial terminals of all strings that \( y \) may expand to
• We’ll use this to decide what production to apply

Examples
• Given grammar \( S \rightarrow xyz \mid abc \)
  \( \rightarrow \) \( \text{First}(xyz) = \{ x \} \), \( \text{First}(abc) = \{ a \} \)
  \( \rightarrow \) \( \text{First}(S) = \text{First}(xyz) \cup \text{First}(abc) = \{ x, a \} \)
• Given grammar \( S \rightarrow A \mid B \quad A \rightarrow x \mid y \quad B \rightarrow z \)
  \( \rightarrow \) \( \text{First}(x) = \{ x \} \), \( \text{First}(y) = \{ y \} \), \( \text{First}(A) = \{ x, y \} \)
  \( \rightarrow \) \( \text{First}(z) = \{ z \} \), \( \text{First}(B) = \{ z \} \)
  \( \rightarrow \) \( \text{First}(S) = \{ x, y, z \} \)

Calculating First(\( y \))

For a terminal \( a \)
• \( \text{First}(a) = \{ a \} \)

For a nonterminal \( N \)
• If \( N \rightarrow \varepsilon \), then add \( \varepsilon \) to First(\( N \))
• If \( N \rightarrow \alpha_1 \alpha_2 \ldots \alpha_n \), then (note the \( \alpha_i \) are all the symbols on the right side of one single production):
  \( \rightarrow \) Add \( \text{First}(\alpha_1 \alpha_2 \ldots \alpha_n) \) to \( \text{First}(N) \), where \( \text{First}(\alpha_1 \alpha_2 \ldots \alpha_n) \) is defined as
  • \( \text{First}(\alpha_i) \) if \( \varepsilon \not\in \text{First}(\alpha_i) \)
  • Otherwise \( (\text{First}(\alpha_1) \sim \varepsilon) \cup \text{First}(\alpha_2 \ldots \alpha_n) \)
  \( \rightarrow \) If \( \varepsilon \in \text{First}(\alpha_i) \) for all \( i \), \( 1 \leq i \leq k \), then add \( \varepsilon \) to First(\( N \))

First(\( \cdot \)) Examples

\[
\begin{align*}
E & \rightarrow \text{id} = n \mid \{ L \} \\
L & \rightarrow E ; L \mid \varepsilon \\
E & \rightarrow \text{id} = n \mid \{ L \} \mid \varepsilon \\
L & \rightarrow E ; L
\end{align*}
\]

\[
\begin{align*}
\text{First}(\text{id}) & = \{ \text{id} \} \\
\text{First}(\text{"="}) & = \{ \text{"="} \} \\
\text{First}(n) & = \{ n \} \\
\text{First}(\text{id}) & = \{ \text{id} \} \\
\text{First}(\text{"="}) & = \{ \text{"="} \} \\
\text{First}(n) & = \{ n \} \\
\text{First}(\text{"="}) & = \{ \text{"="} \} \\
\text{First}(\text{"\{\})} & = \{ \text{"\{\}} \} \\
\text{First}(\text{"\{"\}}) & = \{ \text{"\{"\}} \} \\
\text{First}(\text{\{"\})} & = \{ \text{\{"\}} \} \\
\text{First}(\text{\{"\})} & = \{ \text{\{"\}} \} \\
\text{First}(\text{\{\})} & = \{ \text{\{\}} \} \\
\text{First}(\text{\{"\})} & = \{ \text{\{"\}} \}
\end{align*}
\]
Recursive Descent Parser Implementation

For terminals, create function `match(a)`
- If lookahead is `a` it consumes the lookahead by advancing the lookahead to the next token, and returns
- Otherwise fails with a parse error if lookahead is not `a`
- In algorithm descriptions, consider `parse_a`, `parse_term(a)` to be aliases for `match(a)`

For each nonterminal `N`, create a function `parse_N`
- Called when we’re trying to parse a part of the input which corresponds to (or can be derived from) `N`
- `parse_S` for the start symbol `S` begins the parse

Parser Implementation (cont.)

The body of `parse_N` for a nonterminal `N` does the following
- Let `N → β₁ | ... | βₖ` be the productions of `N`
  - Here `βᵢ` is the entire right side of a production - a sequence of terminals and nonterminals
  - Pick the production `N → βᵢ` such that the lookahead is in `First(βᵢ)`
    - It must be that `First(βᵢ) ∩ First(βⱼ) = ∅` for `i ≠ j`
    - If there is no such production, but `N → ε` then return
    - Otherwise fail with a parse error
  - Suppose `βᵢ = α₁ α₂ ... αₙ`. Then call `parse_α₁(); ... ; parse_αₙ()` to match the expected right-hand side, and return

Parser Implementation (cont.)

Parse is built on procedure calls
- Procedures may be (mutually) recursive

Note:
- These procedures are imperative: assume there is a global list of tokens we are consuming
- In Ocaml, the list of tokens would be passed as an argument, and returned in the result
  - E.g., `match(a)` becomes `match_tok(a)`
    - Argument `l` is list of tokens
    - Return value is token list minus `a`, if `a` was at the front, or throws an exception if not

Recursive Descent Parser

Given grammar `S → xyz | abc`
- `First(xyz) = { x }, First(abc) = { a }`
- Parser
  ```ocaml```
  ```
  parse_S() {
    if (lookahead == "x") {
      match("x"); match("y"); match("z");  // S → xyz
    } else if (lookahead == "a") {
      match("a"); match("b"); match("c");  // S → abc
    } else error();
  ```
  ```
  ```
Recursive Descent Parser

Given grammar $S \rightarrow A \mid B \quad A \rightarrow x \mid y \quad B \rightarrow z$
• First(A) = { x, y }, First(B) = { z }

Parser

```java
parse_S( ) {
    if ((lookahead == "x") ||
        (lookahead == "y"))
        parse_A( );  // S \rightarrow A
    else if (lookahead == "z")
        parse_B( );  // S \rightarrow B
    else error( );
}
parse_A( ) {
    if (lookahead == "x")
        match("x");  // A \rightarrow x
    else if (lookahead == "y")
        match("y");  // A \rightarrow y
    else error( );
}
parse_B( ) {
    if (lookahead == "z")
        match("z");  // B \rightarrow z
    else error( );
}
```

Example

$E \rightarrow id = n \mid \{ L \}$
$S \rightarrow E ; L \mid \epsilon$

```java
First(E) = { id, "{" }
L \rightarrow E ; L | \epsilon
```

Things to Notice

- If you draw the execution trace of the parser
  • You get the parse tree

Examples

- Grammar
  $S \rightarrow xyz$
  $S \rightarrow abc$
- String “xyz”
  parse_S( )
  match("x")
  match("y")
  match("z")
- String “x”
  parse_S( )
  parse_A( )
  match("x")

Things to Notice (cont.)

- This is a **predictive** parser
  • Because the lookahead determines exactly which production to use

This parsing strategy may fail on some grammars
- Production First sets overlap
- Production First sets contain $\epsilon$
- Possible infinite recursion

- Does not mean grammar is not usable
  • Just means this parsing method not powerful enough
  • May be able to change grammar
Conflicting FIRST Sets

- Consider parsing the grammar $E \rightarrow ab \mid ac$
  - $\text{First}(ab) = a$  
  - $\text{First}(ac) = a$
  
Parser fails whenever $A \rightarrow \alpha_1 \mid \alpha_2$ and

  - $\text{First}(\alpha_1) \cap \text{First}(\alpha_2) \neq \epsilon$ or $\varnothing$

Solution
  - Rewrite grammar using left factoring

Left Factoring Algorithm

- Given grammar
  - $A \rightarrow x\alpha_1 \mid x\alpha_2 \mid \ldots \mid x\alpha_n \mid \beta$
- Rewrite grammar as
  - $A \rightarrow xL \mid \beta$
  - $L \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$
- Repeat as necessary

Examples
- $S \rightarrow ab \mid ac \Rightarrow S \rightarrow aL \quad L \rightarrow b \mid c$
- $S \rightarrow abcA \mid abB \mid a \Rightarrow S \rightarrow aL \quad L \rightarrow bcA \mid bB \mid \epsilon$
- $L \rightarrow bcA \mid bB \mid \epsilon \quad \Rightarrow L \rightarrow bL' \mid \epsilon \quad L' \rightarrow cA \mid B$

Alternative Approach

- Change structure of parser
  - First match common prefix of productions
  - Then use lookahead to choose between productions

Example
- Consider parsing the grammar $E \rightarrow a+b \mid a*b \mid a$

\[
\text{parse}_E( ) \{
\begin{align*}
\text{match}("a"); & \quad \text{// common prefix} \\
\text{if (lookahead == "+") \{ } & \quad \text{// } E \rightarrow a+b \\
& \text{match("+"); match("b"); } & \\
\text{if (lookahead == "+") \{ } & \quad \text{// } E \rightarrow a*b \\
& \text{match("*"); match("b"); } & \\
\text{else \{ } & \quad \text{// } E \rightarrow a \\
\end{align*}
\}
\]

Left Recursion

- Consider grammar $S \rightarrow Sa \mid \epsilon$
- Try writing parser

\[
\text{parse}_S( ) \{
\begin{align*}
\text{if (lookahead == "a") \{ } & \quad \text{// } S \rightarrow Sa \\
& \text{parse}_S( ); \\
\text{else \{ } & \quad \text{// } S \rightarrow \epsilon \\
\end{align*}
\}
\]

- Body of $\text{parse}_S( )$ has an infinite loop
  - if (lookahead = "a") then $\text{parse}_S( )$
- Infinite loop occurs in grammar with left recursion
Right Recursion

- Consider grammar $S \rightarrow aS | \epsilon$
  - Again, $\text{First}(aS) = a$
  - Try writing parser
    ```
    parse_S( ) {
      if (lookahead == "a") {
        match("a");
        parse_S( );  // S -> aS
      } else {
        //
      }
    }
    ```
  - Will $\text{parse}_S( )$ infinite loop?
    ➢ Invoking $\text{match}( )$ will advance lookahead, eventually stop
  - Top down parsers handles grammar w/ right recursion

Algorithm To Eliminate Left Recursion

- Given grammar
  - $A \rightarrow A\alpha_1 | A\alpha_2 | \ldots | A\alpha_n | \beta$
    ➢ $\beta$ must exist or derivation will not yield string
  - Rewrite grammar as (repeat as needed)
    - $A \rightarrow \beta L$
    - $L \rightarrow \alpha_1 L | \alpha_2 L | \ldots | \alpha_n L | \epsilon$
  - Replaces left recursion with right recursion
  - Examples
    - $S \rightarrow Sa | \epsilon \Rightarrow S \rightarrow L \quad L \rightarrow aL | \epsilon$
    - $S \rightarrow Sa | Sb | c \Rightarrow S \rightarrow cL \quad L \rightarrow aL | bL | \epsilon$

What's Wrong With Parse Trees?

- Parse trees contain too much information
  - Example
    ➢ Parentheses
    ➢ Extra nonterminals for precedence
  - This extra stuff is needed for parsing
- But when we want to reason about languages
  - Extra information gets in the way (too much detail)

Abstract Syntax Trees (ASTs)

- An abstract syntax tree is a more compact, abstract representation of a parse tree, with only the essential parts

```
 parse tree
       E
      /|
     E E
    /|
   E b
  /|
 c E

 AST
     *     
    /|
   c + 
   /|
  b d
```
Abstract Syntax Trees (cont.)

- Intuitively, ASTs correspond to the data structure you’d use to represent strings in the language
  - Note that grammars describe trees
  - So do OCaml datatypes (which we’ll see later)
  - $E \rightarrow a | b | c | E+E | E-E | E*E | (E)$

Producing an AST

- To produce an AST, we can modify the `parse()` functions to construct the AST along the way
  - `match(a)` returns an AST node (leaf) for $a$
  - `Parse_A` returns an AST node for $A$
    - AST nodes for RHS of production become children of LHS node

Example

- $S \rightarrow aA$

```
Node parse_S() {
    Node n1, n2;
    if (lookahead == 'a') {
        n1 = match('a');
        n2 = parse_A();
        return new Node(n1,n2);
    }
}
```

The Compilation Process

- Source program $\rightarrow$ Compiler $\rightarrow$ Target program
- Lexing, Parsing, AST, Intermediate Code Generation, Optimization
- (may not actually be constructed)