Gradescope

- Sign up: Gradescope.com
- CMSC330 Invite code: posted on piazza
- Full Name and Student id#
  - First letter of your last name
  - Last five digits of your ID
- Email: You can use any email address
- For Example:
  - Anwar Mamat  UID: 123456789
  - Gradescope full name and id:  m56789. Use it for future exams.
  - Email: xyz123@gmail.com (fake email, don’t send email)
Recall: Steps of Compilation

source program \(\xrightarrow{}\) Compiler \(\xrightarrow{}\) target program

Lexing \(\xrightarrow{}\) Parsing \(\xrightarrow{}\) Intermediate Code Generation \(\xrightarrow{}\) Optimization
Implementing Parsers

- Many efficient techniques for parsing
  - I.e., turning strings 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 → id = n | { L }
L → E ; L | ε

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

Show parse tree for
{ x = 3 ; { y = 4 ; } ; }
Bottom-up Parsing

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

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
- Complicated to use; requires tool support
  - *Bison*, *yacc* produce shift-reduce parsers from CFGs
Tradeoffs

- Recursive descent parsers
  - Easy to write
    - 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
  - Fast
    - Can be implemented with a simple table

- Shift-reduce parsers handle more grammars
  - 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 → id = n | { L }
L → E ; L | ε

• Here n is an integer and 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.)

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

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

lookahead
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 $S \rightarrow xyz \mid abc$
    - Select $S \rightarrow xyz$ since 1st terminal in RHS matches $x$
  - Given grammar $S \rightarrow A \mid B \quad A \rightarrow x \mid y \quad B \rightarrow z$
    - 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

- **First(γ)**, for any terminal or nonterminal γ, is the set of initial terminals of all strings that γ may expand to.
- We’ll use this to decide what production to apply.

Examples

- Given grammar **S → xyz | abc**
  - First(xyz) = { x }, First(abc) = { a }
  - First(S) = First(xyz) U First(abc) = { x, a }

- Given grammar **S → A | B**  
  - A → x | y 
  - B → z
  - First(x) = { x }, First(y) = { y }, First(A) = { x, y }
  - First(z) = { z }, First(B) = { z }
  - First(S) = { x, y, z }
Calculating First(γ)

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

- For a nonterminal \( N \)
  - If \( N \to \epsilon \), then add \( \epsilon \) to \( \text{First}(N) \)
  - If \( N \to \alpha_1 \alpha_2 \ldots \alpha_n \), then (note the \( \alpha_i \) are all the symbols on the right side of one single production):
    - 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
      - \( \text{First}(\alpha_1) \) if \( \epsilon \not\in \text{First}(\alpha_1) \)
      - Otherwise \( (\text{First}(\alpha_1) - \epsilon) \cup \text{First}(\alpha_2 \ldots \alpha_n) \)
    - If \( \epsilon \in \text{First}(\alpha_i) \) for all \( i, 1 \leq i \leq k \), then add \( \epsilon \) to \( \text{First}(N) \)
First( ) Examples

E → id = n | { L }
L → E ; L | ε

First(id) = { id }
First("=") = { "=" }
First(n) = { n }
First("{")= { "{" }
First(";")= { ";" }
First(E) = { id, "{" }
First(L) = { id, "{", ε }

E → id = n | { L } | ε
L → E ; L

First(id) = { id }
First("=") = { "=" }
First(n) = { n }
First("{")= { "{" }
First(";")= { ";" }
First(E) = { id, "{", ε }
First(L) = { id, "{", ";" }
First Set Quiz

Given the following grammar:

\[
\begin{align*}
S & \rightarrow aAB \\
A & \rightarrow CBC \\
B & \rightarrow b \\
C & \rightarrow cC \; | \; \text{epsilon}
\end{align*}
\]

What is First(S)?

A. \{a\}  
B. \{b, c\}  
C. \{b\}  
D. \{c\}
First Set Quiz

Given the following grammar:

\[
\begin{align*}
S & \rightarrow \text{aAB} \\
A & \rightarrow \text{CBC} \\
B & \rightarrow \text{b} \\
C & \rightarrow \text{cC} | \text{epsilon}
\end{align*}
\]

What is First(S)?

A. \{a\}  
B. \{b, c\}  
C. \{b\}  
D. \{c\}
Given the following grammar:

\[
\begin{align*}
S & \rightarrow \text{aAB} \\
A & \rightarrow \text{CBC} \\
B & \rightarrow \text{b} \\
C & \rightarrow \text{cC} \mid \text{epsilon}
\end{align*}
\]

What is First(B)?

A. {a}
B. {b, c}
C. {b}
D. {c}
Given the following grammar:

\[
\begin{align*}
S & \rightarrow aAB \\
A & \rightarrow CBC \\
B & \rightarrow b \\
C & \rightarrow cC \mid \text{epsilon}
\end{align*}
\]

What is First(B)?

A. \{a\}  
B. \{b, c\}  
C. \{b\}  
D. \{c\}
First Set Quiz

Given the following grammar:

\[
\begin{align*}
S & \rightarrow aAB \\
A & \rightarrow CBC \\
B & \rightarrow b \\
C & \rightarrow cC \mid \text{epsilon}
\end{align*}
\]

What is First(A)?

A. \{a\}
B. \{b, c\}
C. \{b\}
D. \{c\}
First Set Quiz

Given the following grammar:

\[
\begin{align*}
S & \rightarrow \text{aAB} \\
A & \rightarrow \text{CBC} \\
B & \rightarrow \text{b} \\
C & \rightarrow \text{cC} \mid \text{epsilon}
\end{align*}
\]

What is First(A)?

A. \{a\}  
B. \{b,c\}  
C. \{b\}  
D. \{c\}
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
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
Recursive Descent Parser

- Given grammar $S \rightarrow xyz \mid abc$
  - $\text{First}(xyz) = \{ x \}$, $\text{First}(abc) = \{ a \}$

- Parser

  ```c
  parser_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 $S \rightarrow A \mid B \quad A \rightarrow x \mid y \quad B \rightarrow z$
  - $\text{First}(A) = \{ x, y \}$, $\text{First}(B) = \{ z \}$

Parser

```plaintext
parse_S( ) {
    if ((lookahead == "x") ||
        (lookahead == "y"))
        parse_A( ); // S → A
    else if (lookahead == "z")
        parse_B( ); // S → B
    else error( );
}

parse_A( ) {
    if (lookahead == "x")
        match("x"); // A → x
    else if (lookahead == "y")
        match("y"); // A → y
    else error( );
}

parse_B( ) {
    if (lookahead == "z")
        match("z"); // B → z
    else error( );
}
```
Example

E → id = n | { L }
L → E ; L | ε

First(E) = { id, "{" }

parse_E( ) {
    if (lookahead == "id") {
        match("id");
        match("="); // E → id = n
        match("n");
    }
    else if (lookahead == "{")
        parse_L( ); // E → { L }
    else error( );
}

parse_L( ) {
    if ((lookahead == "id") ||
        (lookahead == "{")) {
        parse_E( );
        match(";"); // L → E ; L
        parse_L( );
    } else ; // L → ε
Things to Notice

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

Examples

- Grammar
  S → xyz
  S → abc
- String “xyz”
  ```
  parse_S()
  match("x") / \ match("y")
  match("z")
  ```

- Grammar
  S → A | B
  A → x | y
  B → 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 $E \rightarrow ab | ac$
  - First(ab) = a
  - First(ac) = a
    
    Parser cannot choose between RHS based on lookahead!

- Parser fails whenever $A \rightarrow \alpha_1 | \alpha_2$ and
  - $\text{First}(\alpha_1) \cap \text{First}(\alpha_2) \neq \epsilon$ or $\emptyset$

- 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
  - \( S \rightarrow ab \mid ac \Rightarrow S \rightarrow aL \quad L \rightarrow b \mid c \)
  - \( S \rightarrow abcA \mid abB \mid a \Rightarrow S \rightarrow aL \quad L \rightarrow bcA \mid bB \mid \varepsilon \)
  - \( L \rightarrow bcA \mid bB \mid \varepsilon \Rightarrow L \rightarrow bL' \mid \varepsilon \quad L' \rightarrow cA \mid B \)
Alternative Approach

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

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

```plaintext
parse_E( ) {
    match("a"); // common prefix
    if (lookahead == "+") { // E → a+b
        match("+"); match("b");
    }
    if (lookahead == "*") { // E → a*b
        match("*"); match("b");
    }
    else { } // E → a
}
```
Left Recursion

Consider grammar $S \rightarrow Sa \mid \varepsilon$

- Try writing parser

```c
parse_S( ) {
    if (lookahead == "a") {
        parse_S( );
        match("a");  // S → Sa
    } else {
    }
}
```

- Body of `parse_S( )` has an infinite loop
  - if (lookahead = "a") then `parse_S( )`
- Infinite loop occurs in grammar with left recursion
Right Recursion

Consider grammar \( S \rightarrow aS \mid \epsilon \)

- Again, \( \text{First}(aS) = a \)
- Try writing parser

```c
parse_S() {
    if (lookahead == "a") {
        match("a");
        parse_S(); // S \rightarrow aS
    }
    else {
    }
}
```

- Will \( parse_S() \) infinite loop?
  - Invoking `match()` will advance lookahead, eventually stop
- Top down parsers handles grammar w/ right recursion
Algorithm To Eliminate Left Recursion

Given grammar
- $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta$
  - $\beta$ must exist or derivation will not yield string

Rewrite grammar as (repeat as needed)
- $A \rightarrow \beta L$
- $L \rightarrow \alpha_1 L \mid \alpha_2 L \mid \ldots \mid \alpha_n L \mid \epsilon$

Replaces left recursion with right recursion

Examples
- $S \rightarrow Sa \mid \epsilon$  $\Rightarrow S \rightarrow L \quad L \rightarrow aL \mid \epsilon$
- $S \rightarrow Sa \mid Sb \mid c$  $\Rightarrow S \rightarrow cL \quad L \rightarrow aL \mid bL \mid \epsilon$
Parsing Quiz

What Does the following code parse?

```c
parse_S () {
    If (lookahead == 'a'){
        match("a");
        match("x");
        match("y");
    }
    Else if (lookahead == 'q'){
        match("q");
    }
    Else error();
}
```

A. S -> aqxy
B. S -> a | q
C. S -> aaxy | qq
D. S -> axy | q
Parsing Quiz

What Does the following code parse?

```c
parse_S () {
    If (lookahead == ‘a’){
        match(“a”);
        match(“x”);
        match(“y”);
    }
    Else if (lookahead == ‘q’){
        match(“q”);
    }
    Else error();
}
```

A. S -> aqxy
B. S -> a | q
C. S -> aaxy | qq
D. S -> axy | q
Parsing Quiz

What Does the following code parse?

```cpp
parse_S () {
    If (lookahead == 'a'){
        match("a");
        parse_S();
    }
    Else if (lookahead == 'q'){
        match("q");
        match("p");
    }
    Else error();
}
```

A. S -> aS | qp
B. S -> a | S | qp
C. S -> aqSp
D. S -> a | q
Parsing Quiz

What Does the following code parse?

```c
parse_S () {
    If (lookahead == ‘a’){
        match(“a”);
        parse_S();
    }
    Else if (lookahead == ‘q’){
        match(“q”);
        match(“p”);
    }
    Else error();
}
```

A. S → aS | qp
B. S → a | S | qp
C. S → aqSp
D. S → a | q

CMSC 330 - Fall 2016
Can recursive descent parse this grammar?

S -> aBa
B -> bC
C -> epsilon | Cc

A. Yes
B. No
Parsing Quiz

Can recursive descent parse this grammar?

S → aBa
B → bC
C → ε | Cc

A. Yes
B. No
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.
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)
- $E \rightarrow a \mid b \mid c \mid E+E \mid E-E \mid E*E \mid (E)$
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 → aA`

```java
Node parse_S() {
    Node n1, n2;
    if (lookahead == "a") {
        n1 = match("a");
        n2 = parse_A();
        return new Node(n1, n2);
    }
}
```
The Compilation Process

- Lexing: regexps, DFAs
- Parsing: CFGs, PDAs
- AST: (may not actually be constructed)
- Intermediate Code Generation
- Optimization

source program → Compiler → target program