

CMSC 330: Organization of Programming Languages

Parsing

This Lecture

- ▶ Parsing
 - Recursive descent
 - FIRST sets
- ▶ Rewriting grammars
 - Left factoring
 - Eliminating left recursion
- ▶ Abstract syntax trees (ASTs)

CMSC 330

3

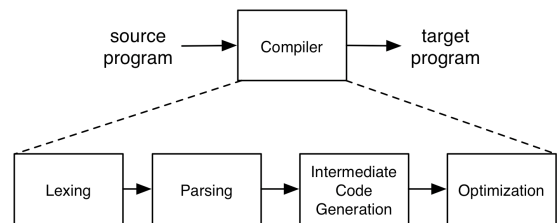
Last Lecture

- ▶ Context free grammars
 - Derivations
 - Parse trees
 - Ambiguity
 - Associativity & precedence
 - Designing grammars

CMSC 330

2

Steps of Compilation



CMSC 330

4

Parsing

- ▶ 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

CMSC 330

5

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

CMSC 330

6

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

CMSC 330

7

Parsing Example

$E \rightarrow id = n \mid \{ L \}$

$L \rightarrow E ; L \mid \epsilon$

- 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

CMSC 330

8

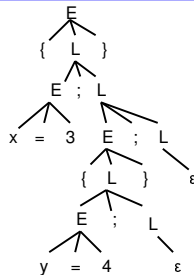
Parsing Example (cont.)

$E \rightarrow id = n \mid \{ L \}$

$L \rightarrow E ; L \mid \epsilon$

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

↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑
lookahead



CMSC 330

9

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

CMSC 330

10

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

CMSC 330

11

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

CMSC 330

12

Calculating First(y)

- ▶ For a terminal a
 - $\text{First}(a) = \{ a \}$
- ▶ For a nonterminal N
 - If $N \rightarrow \epsilon$, then add ϵ to $\text{First}(N)$
 - If $N \rightarrow \alpha_1 \alpha_2 \dots \alpha_n$, then (note the α_i are all the symbols on the right side of one single production):
 - > Add $\text{First}(\alpha_1 \alpha_2 \dots \alpha_n)$ to $\text{First}(N)$, where $\text{First}(\alpha_1 \alpha_2 \dots \alpha_n)$ is defined as
 - $\text{First}(\alpha_1)$ if $\epsilon \notin \text{First}(\alpha_1)$
 - Otherwise $(\text{First}(\alpha_1) - \epsilon) \cup \text{First}(\alpha_2 \dots \alpha_n)$
 - > If $\epsilon \in \text{First}(\alpha_i)$ for all i , $1 \leq i \leq n$, then add ϵ to $\text{First}(N)$

CMSC 330

13

First() Examples

$E \rightarrow \text{id} = n \mid \{ L \}$	$E \rightarrow \text{id} = n \mid \{ L \} \mid \epsilon$
$L \rightarrow E ; L \mid \epsilon$	$L \rightarrow E ; L$
$\text{First}(\text{id}) = \{ \text{id} \}$	$\text{First}(\text{id}) = \{ \text{id} \}$
$\text{First}("=") = \{ "=" \}$	$\text{First}("=") = \{ "=" \}$
$\text{First}(n) = \{ n \}$	$\text{First}(n) = \{ n \}$
$\text{First}("\{") = \{ "\{ " \}$	$\text{First}("\{") = \{ "\{ " \}$
$\text{First}("\}") = \{ "\} " \}$	$\text{First}("\}") = \{ "\} " \}$
$\text{First}(";") = \{ "; " \}$	$\text{First}(";") = \{ "; " \}$
$\text{First}(E) = \{ \text{id}, "\{ " \}$	$\text{First}(E) = \{ \text{id}, "\{ ", \epsilon \}$
$\text{First}(L) = \{ \text{id}, "\{ ", \epsilon \}$	$\text{First}(L) = \{ \text{id}, "\{ ", "; " \}$

CMSC 330

14

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

CMSC 330

15

Parser Implementation (cont.)

- ▶ The body of `parse_N` for a nonterminal N does the following
 - Let $N \rightarrow \beta_1 \mid \dots \mid \beta_k$ be the productions of N
 - > Here β_i is the entire right side of a production: a sequence of terminals and nonterminals
 - Pick the production $N \rightarrow \beta_i$ such that the lookahead is in $\text{First}(\beta_i)$
 - > It must be that $\text{First}(\beta_i) \cap \text{First}(\beta_j) = \emptyset$ for $i \neq j$
 - > If there is no such production, but $N \rightarrow \epsilon$ then return
 - > Otherwise fail with a parse error
 - Suppose $\beta_i = \alpha_1 \alpha_2 \dots \alpha_n$. Then call `parse_α1()`; ... ; `parse_αn()` to match the expected right-hand side, and return

CMSC 330

16

Parser Implementation (cont.)

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

CMSC 330

17

Recursive Descent Parser

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

```

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();
}

```

CMSC 330

18

Recursive Descent Parser

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

Parser

```

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();
}
    
```

CMSC 330

19

Example

- $E \rightarrow id = n \mid \{ L \}$ $\text{First}(E) = \{ id, "{" \}$
- $L \rightarrow E ; L \mid \epsilon$

```

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

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

CMSC 330

20

Things to Notice

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

Examples

- Grammar $S \rightarrow xyz$
- Grammar $S \rightarrow A \mid B$
- String "xyz"
- String "x"

```

parse_S()
match("x")
match("y")
match("z")
    
```

S
 $/ \setminus$
 $x \ y \ z$

```

parse_S()
parse_A()
match("x")
    
```

S
 $|$
 A
 $|$
 x

CMSC 330

21

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
 - Possible infinite recursion
 - Production First sets overlap
 - Production First sets contain ϵ
- Does not mean grammar is not usable
 - Just means this parsing method not powerful enough
 - May be able to change grammar

CMSC 330

22

Left Factoring

- Consider parsing the grammar $E \rightarrow ab \mid ac$
 - $\text{First}(ab) = a$
 - $\text{First}(ac) = a$
 - Parser cannot choose between RHS based on lookahead!
- Parser fails whenever $A \rightarrow \alpha_1 \mid \alpha_2$ and
 - $\text{First}(\alpha_1) \cap \text{First}(\alpha_2) \neq \epsilon \text{ or } \emptyset$
- Solution
 - Rewrite grammar using **left factoring**

CMSC 330

23

Left Factoring Algorithm

- Given grammar
 - $A \rightarrow x\alpha_1 \mid x\alpha_2 \mid \dots \mid x\alpha_n \mid \beta$
- Rewrite grammar as
 - $A \rightarrow xL \mid \beta$
 - $L \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$
- Repeat as necessary
- Examples
 - $S \rightarrow ab \mid ac \Rightarrow S \rightarrow aL \quad L \rightarrow b \mid c$
 - $S \rightarrow abcA \mid abB \mid a \Rightarrow S \rightarrow aL \quad L \rightarrow bcA \mid bB \mid \epsilon$
 - $L \rightarrow bcA \mid bB \mid \epsilon \Rightarrow L \rightarrow bL' \mid \epsilon \quad L' \rightarrow cA \mid B$

CMSC 330

24

Left Recursion

- Consider grammar $S \rightarrow Sa \mid \epsilon$
- First(Sa) = a, so we're ok as far as which production
- Try writing parser

```
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**

CMSC 330

25

Right Recursion

- Consider grammar $S \rightarrow aS \mid \epsilon$
- Again, First(aS) = a
- Try writing parser

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

- Will `parse_S()` infinite loop?
 - > Invoking `match()` will advance lookahead, eventually stop
- Top down parsers handles grammar w/ **right recursion**

CMSC 330

26

Algorithm To Eliminate Left Recursion

- Given grammar
 - $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_n \mid \beta$
 - > Why must β exist?
- Rewrite grammar as
 - $A \rightarrow \beta L$
 - $L \rightarrow \alpha_1 L \mid \alpha_2 L \mid \dots \mid \alpha_n L \mid \epsilon$
- Replaces left recursion with right recursion
- Repeat as necessary

CMSC 330

27

Eliminating Left Recursion (cont.)

- Examples
 - $S \rightarrow Sa \mid \epsilon \quad \Rightarrow \quad S \rightarrow L \quad L \rightarrow aL \mid \epsilon$
 - $S \rightarrow Sa \mid Sb \mid c \quad \Rightarrow \quad S \rightarrow cL \quad L \rightarrow aL \mid bL \mid \epsilon$
- May need more powerful algorithms to eliminate **mutual recursion** leading to left recursion
 - $S \rightarrow Aa \mid b$
 - $A \rightarrow Sb$

CMSC 330

28

Expr Grammar for Top-Down Parsing

```
E → T E'
E' → ε | + E
T → P T'
T' → ε | * T
P → n | ( E )
```

- Notice we can always decide what production to choose with only one symbol of lookahead

CMSC 330

29

Tradeoffs with Other Approaches

- Recursive descent parsers are 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
 - They're unable to handle certain kinds of grammars
- Recursive descent is good for a simple parser
 - Though tools can be fast if you're familiar with them
- Can implement top-down predictive parsing as a table-driven parser
 - By maintaining an explicit stack to track progress

CMSC 330

30

Tradeoffs with Other Approaches

- More powerful techniques need tool support
 - Can take time to learn tools
- Main alternative is bottom-up, shift-reduce parser
 - 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

CMSC 330

31

What's Wrong With Parse Trees?

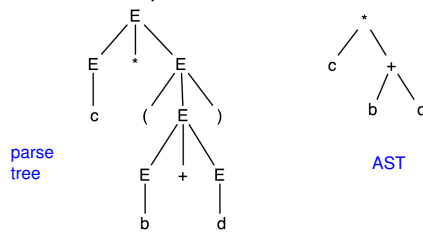
- Parse trees contain too much information
 - Example
 - Parenteses
 - Extra nonterminals for precedence
 - This extra stuff is needed for parsing
- But when we want to reason about languages
 - Extra information gets in the way (too much detail)

CMSC 330

32

Abstract Syntax Trees (ASTs)

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

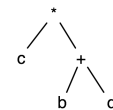


CMSC 330

33

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)
 - $E \rightarrow a \mid b \mid c \mid E+E \mid E-E \mid E^*E \mid (E)$



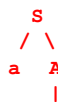
CMSC 330

34

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
 - $S \rightarrow aA$

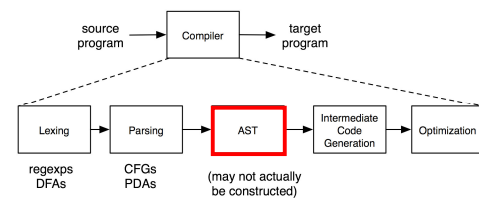
```
Node parse_S() {
  Node n1, n2;
  if (lookahead == "a") {
    n1 = match("a");
    n2 = parse_A();
    return new Node(n1,n2);
  }
}
```



CMSC 330

35

The Compilation Process



CMSC 330

36

Summary

- ▶ Learned a little about parsing
 - Recursive descent parser
 - Predictive parsing using FIRST sets
- ▶ Rewriting grammars for predicative parsing
 - Left factoring
 - Eliminating left recursion
- ▶ Abstract syntax trees (ASTs)