

CMSC 330: Organization of Programming Languages

Context-Free Grammars

Motivation

- Programs are just strings of text
 - But they're strings that have a certain structure
 - A C program is a list of declarations and definitions
 - A function definition contains parameters and a body
 - A function body is a sequence of statements
 - A statement is either an expression, an if, a goto, etc.
 - An expression may be assignment, addition, subtraction, etc
- We want to solve two problems
 - We want to describe programming languages precisely
 - We need to describe more than the regular languages
 - Recall that regular expressions, DFAs, and NFAs are limited in their expressiveness

CMSC 330

2

Program structure

Syntax

- What a program looks like
- BNF (context free grammars) - a useful notation for describing syntax.

Semantics

- Execution behavior
- **Static semantics** - Semantics determined at compile time:
 - var A: integer; **Type and storage for A**
 - int B[10]; **Type and storage for array B**
 - float MyProcC(float x;float y){...}; **Function attributes**
- **Dynamic semantics** - Semantics determined during execution - e.g. in SNOBOL4:
 - X = "ABC" **X now a string with value "ABC"**
 - X = 1 + 2; **X now an integer with value 3**
 - :(X) **X an address; Transfer control to statement at label X**

CMSC 330

3

Context-Free Grammars (CFGs)

- A way of generating sets of strings or languages
- They subsume regular expressions (and DFAs and NFAs)
 - There is a CFG that generates any regular language
 - (But regular expressions are a better notation for languages which are regular.)
- They can be used to describe programming languages
 - They describe the parsing process (mostly)

CMSC 330

4

Formal Definition

- A context-free grammar G is a 4-tuple:
 - Σ – a finite set of *terminal* or *alphabet* symbols
 - Often written in lowercase
 - N – a finite, nonempty set of *nonterminal* symbols
 - Often written in uppercase
 - It must be that $N \cap \Sigma = \emptyset$
 - P – a set of *productions* of the form $N \rightarrow (\Sigma|N)^*$
 - Informally this means that the nonterminal can be replaced by the string of zero or more terminals or nonterminals to the right of the \rightarrow
 - $S \in N$ – the *start symbol*

Example: Arithmetic Expressions (Limited)

- $E \rightarrow a \mid b \mid c \mid E+E \mid E-E \mid E^*E \mid (E)$
 - An expression E is either a letter a , b , or c
 - Or an E followed by $+$ followed by an E
 - etc.
- This describes or generates a set of strings
 - $\{a, b, c, a+b, a+a, a^*c, a-(b^*a), c^*(b+d)\}$
- Example strings not in the language
 - $d, c(a), a+, b^{**}c$, etc.

Notational Shortcuts

- Formally, the grammar we just showed is
 - $\Sigma = \{+, -, *, (,), a, b, c\}$
 - $N = \{E\}$
 - $P = \{E \rightarrow a, E \rightarrow b, E \rightarrow c, E \rightarrow E-E, E \rightarrow E+E, E \rightarrow E^*E, E \rightarrow (E)\}$
 - $S = E$
- If not specified, assume the left-hand side of the first listed production is the start symbol
- Usually productions with the same left-hand sides are combined with $|$
- If a production has an empty right-hand side it means ϵ

Backus-Naur Form

- Context-free grammar production rules are also called Backus-Naur Form or **BNF**
 - A production like $A \rightarrow B c D$ is written in BNF as $\langle A \rangle ::= \langle B \rangle c \langle D \rangle$ (Non-terminals written with angle brackets and $::=$ instead of \rightarrow)
 - Often used to describe language syntax
- John Backus
 - Chair of the Algol committee in the early 1960s
- Peter Naur
 - Secretary of the committee, who used this notation to describe Algol in 1962

Uniqueness of Grammars

- Grammars are not unique. Different grammars can generate the same set of strings.
- The following grammar generates the same set of strings as the previous grammar:

$$E \rightarrow E+T \mid E-T \mid T$$

$$T \rightarrow T^*P \mid P$$

$$P \rightarrow (E) \mid a \mid b \mid c$$

Another Example Grammar

- $S \rightarrow aS \mid T$
- $T \rightarrow bT \mid U$
- $U \rightarrow cU \mid \epsilon$

Sentential Forms and Derivations

- A *sentential form* is a string of terminals and nonterminals produced from that start symbol
 - The start symbol is a sentential form for a grammar
 - If $\alpha A \delta$ is a sentential form for a grammar, where (α and $\delta \in (N \mid \Sigma)^*$), and $A \rightarrow \gamma$ is a production, then $\alpha \gamma \delta$ is a sentential form for the grammar
 - In this case, we say that $\alpha A \delta$ *derives* $\alpha \gamma \delta$ in one step, which is written as $\alpha A \delta \Rightarrow \alpha \gamma \delta$
- \Rightarrow^+ is used to indicate a derivation of one or more steps
- \Rightarrow^* indicates a derivation of zero or more steps

Example

$$S \rightarrow aS \mid T$$

$$T \rightarrow bT \mid U$$

$$U \rightarrow cU \mid \epsilon$$

- A derivation:
 - $S \Rightarrow aS \Rightarrow aT \Rightarrow aU \Rightarrow acU \Rightarrow ac$
 - Abbreviated as $S \Rightarrow^+ ac$
 - So S, aS, aT, aU, acU, ac are all sentential forms for this grammar
 - $S \Rightarrow T \Rightarrow U \Rightarrow \epsilon$
- Is there any derivation
 - $S \Rightarrow^+ ccc ?$ $S \Rightarrow^+ Sa ?$
 - $S \Rightarrow^+ bab ?$ $S \Rightarrow^+ bU ?$

The Language Generated by a CFG

- The *language generated by a grammar G* is

$$L(G) = \{ \omega \mid \omega \in \Sigma^* \text{ and } S \Rightarrow^+ \omega \}$$

- (where S is the start symbol of the grammar and Σ is the alphabet for that grammar)
- I.e., all sentential forms with only terminals
- I.e., all strings over Σ that can be derived from the start symbol via one or more productions

Example (cont'd)

$$S \rightarrow aS \mid T$$

$$T \rightarrow bT \mid U$$

$$U \rightarrow cU \mid \epsilon$$

- Generates what language?
- Do other grammars generate this language?

$$S \rightarrow ABC$$

$$A \rightarrow aA \mid \epsilon$$

$$B \rightarrow bB \mid \epsilon$$

$$C \rightarrow cC \mid \epsilon$$

- So grammars are not unique

Parse Trees

- A *parse tree* shows how a string is produced by a grammar
 - The root node is the start symbol
 - Each interior node is a nonterminal
 - Children of node are symbols on r.h.s of production applied to that nonterminal
 - Leaves are all terminal symbols
- Reading the leaves left-to-right shows the string corresponding to the tree

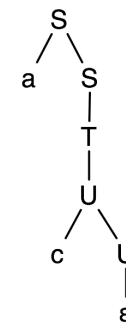
Example

$$S \Rightarrow aS \Rightarrow aT \Rightarrow aU \Rightarrow acU \Rightarrow ac$$

$$S \rightarrow aS \mid T$$

$$T \rightarrow bT \mid U$$

$$U \rightarrow cU \mid \epsilon$$



Parse Trees for Expressions

- A *parse tree* shows the structure of an expression as it corresponds to a grammar

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

a

```

    E
    |
    a
  
```

a*c

```

      E
     /|\
    E * E
    |   |
    a   c
  
```

c*(b+d)

```

      E
     /|\
    E * E
    |   |
    c   (
        E
       /|\
      E + E
      |   |
      b   d
    )
  
```

Another Example

- Is *a* in the language generated by this grammar? $S \rightarrow a \mid SbS$

- How about *aba*?

- Yes, there are two possible derivations

- $S \Rightarrow SbS \Rightarrow abS \Rightarrow aba$

- This is a *leftmost derivation*

- At every step, apply production to leftmost nonterminal

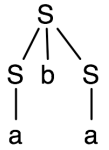
- $S \Rightarrow SbS \Rightarrow Sba \Rightarrow aba$

- This is a *rightmost derivation*

- Both derivations have the same parse tree

- A parse tree has a unique leftmost and a unique rightmost derivation (BUT we will soon see that not every string has a unique parse tree.)

- Parse trees don't show the order productions are applied



Another Example (cont'd)

$$S \rightarrow a \mid SbS$$

- Is *ababa* in this language?

A leftmost derivation

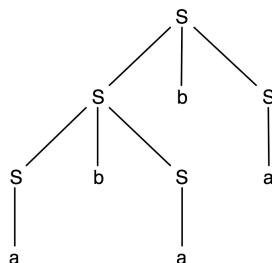
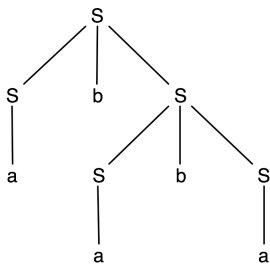
$$S \Rightarrow SbS \Rightarrow abS \Rightarrow$$

$$abSbS \Rightarrow ababS \Rightarrow ababa$$

Another leftmost derivation

$$S \Rightarrow SbS \Rightarrow SbSbS \Rightarrow$$

$$abSbS \Rightarrow ababS \Rightarrow ababa$$



Ambiguity

- A string is *ambiguous* for a grammar if it has more than one parse tree
 - Equivalent to more than one leftmost (or more than one rightmost) derivation
- A grammar is *ambiguous* if it generates an ambiguous string
 - It's can be hard to see this with manual inspection
- Exercise: can you create an unambiguous grammar for $S \rightarrow a \mid SbS$?

Are these Grammars Ambiguous?

(1) $S \rightarrow aS \mid T$
 $T \rightarrow bT \mid U$
 $U \rightarrow cU \mid \epsilon$

(2) $S \rightarrow T \mid T$
 $T \rightarrow Tx \mid Tx \mid x \mid x$

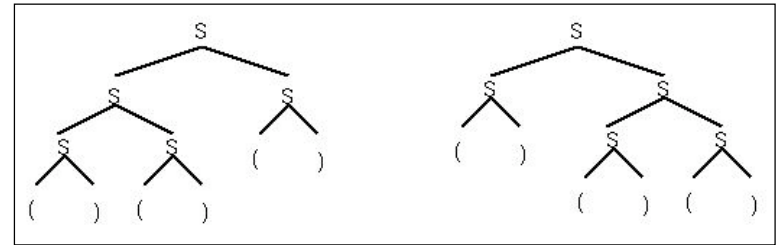
(3) $S \rightarrow SS \mid () \mid (S)$

Ambiguity of Grammar (3)

- 2 different parse trees for the same string: $()()()$
- 2 distinct leftmost derivations :

$S \Rightarrow SS \Rightarrow SSS \Rightarrow ()SS \Rightarrow ()()S \Rightarrow ()()()$

$S \Rightarrow SS \Rightarrow ()S \Rightarrow ()SS \Rightarrow ()()S \Rightarrow ()()()$



- We need unambiguous grammars to manage programming language semantics

More on Leftmost/Rightmost Derivations

- Is the following derivation leftmost or rightmost?

$S \Rightarrow aS \Rightarrow aT \Rightarrow aU \Rightarrow acU \Rightarrow ac$

- There's at most one nonterminal in each sentential form, so there's no choice between left or right nonterminals to expand

- How about the following derivation?

– $S \Rightarrow SbS \Rightarrow SbSbS \Rightarrow SbabS \Rightarrow ababS \Rightarrow ababa$

Tips for Designing Grammars

1. Use recursive productions to generate an arbitrary number of symbols
 - $A \rightarrow xA \mid \epsilon$ Zero or more x 's
 - $A \rightarrow yA \mid y$ One or more y 's
2. Use separate nonterminals to generate disjoint parts of a language, and then combine in a production

$G = S \rightarrow AB$

$A \rightarrow aA \mid \epsilon$

$B \rightarrow bB \mid \epsilon$

$L(G) = a^*b^*$

Tips for Designing Grammars (cont'd)

3. To generate languages with matching, balanced, or related numbers of symbols, write productions which generate strings from the middle

$\{a^n b^n \mid n \geq 0\}$ (not a regular language!)

$S \rightarrow aSb \mid \epsilon$

Example: $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$

$\{a^n b^{2n} \mid n \geq 0\}$

$S \rightarrow aSbb \mid \epsilon$

Tips for Designing Grammars (cont'd)

$\{a^n b^m \mid m \geq 2n, n \geq 0\}$

$S \rightarrow aSbb \mid B \mid \epsilon$

$B \rightarrow bB \mid b$

The following grammar also works:

$S \rightarrow aSbb \mid B$

$B \rightarrow bB \mid \epsilon$

How about the following?

$S \rightarrow aSbb \mid bS \mid \epsilon$

Tips for Designing Grammars (cont'd)

$\{a^n b^m a^{n+m} \mid n \geq 0, m \geq 0\}$

Rewrite as $a^n b^m a^m a^n$, which now has matching superscripts (two pairs)

Would this grammar work?

$S \rightarrow aSa \mid B$

$B \rightarrow bBa \mid ba$

Corrected:

$S \rightarrow aSa \mid B$

$B \rightarrow bBa \mid \epsilon$

The outer $a^n a^n$ are generated first, then the inner $b^m a^m$

Tips for Designing Grammars (cont'd)

4. For a language that's the union of other languages, use separate nonterminals for each part of the union and then combine

$\{a^n(b^m|c^m) \mid m > n \geq 0\}$

Can be rewritten as

$\{a^n b^m \mid m > n \geq 0\} \cup$

$\{a^n c^m \mid m > n \geq 0\}$

Tips for Designing Grammars (cont'd)

$\{ a^n b^m \mid m > n \geq 0 \} \cup \{ a^n c^m \mid m > n \geq 0 \}$

$S \rightarrow T \mid U$

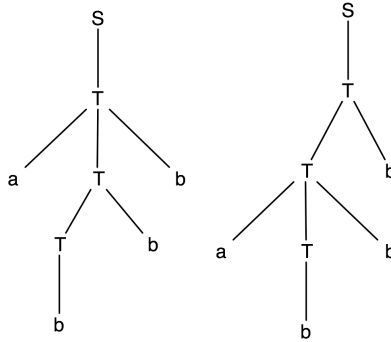
$T \rightarrow aTb \mid Tb \mid b$

$U \rightarrow aUc \mid Uc \mid c$

T generates the first set

U generates the second set

- What about the string **abbb**?
 - Ambiguous!



Tips for Designing Grammars (cont'd)

$\{ a^n b^m \mid m > n \geq 0 \} \cup \{ a^n c^m \mid m > n \geq 0 \}$

$S \rightarrow T \mid U$

$T \rightarrow aTb \mid bT \mid b$

$U \rightarrow aUc \mid cU \mid c$

Will this fix the ambiguity?

- It's not ambiguous, but it can generate invalid strings such as **babb**

Tips for Designing Grammars (cont'd)

$\{ a^n b^m \mid m > n \geq 0 \} \cup \{ a^n c^m \mid m > n \geq 0 \}$

Unambiguous version

$S \rightarrow T \mid V$

$T \rightarrow aTb \mid U$

$U \rightarrow Ub \mid b$

$V \rightarrow aVc \mid W$

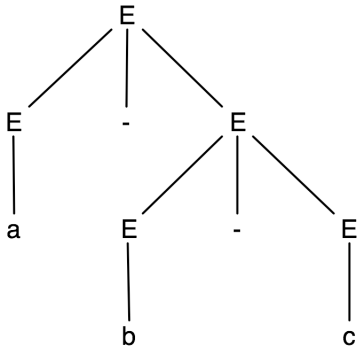
$W \rightarrow Wc \mid c$

CFGs for Languages

- Recall that our goal is to describe programming languages with CFGs
- We had the following example which describes limited arithmetic expressions
 - $E \rightarrow a \mid b \mid c \mid E+E \mid E-E \mid E^*E \mid (E)$
- What's wrong with using this grammar?
 - It's ambiguous!

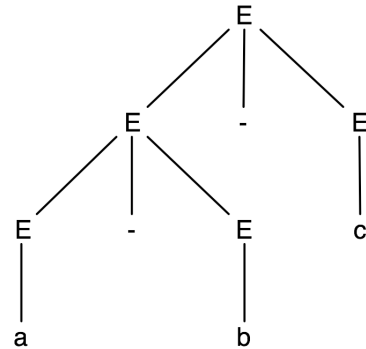
Example: a-b-c

$E \Rightarrow E-E \Rightarrow a-E \Rightarrow a-E-E \Rightarrow$
 $a-b-E \Rightarrow a-b-c$



Corresponds to $a-(b-c)$

$E \Rightarrow E-E \Rightarrow E-E-E \Rightarrow$
 $a-E-E \Rightarrow a-b-E \Rightarrow a-b-c$



Corresponds to $(a-b)-c$

The Issue: Associativity

- Ambiguity is bad here because if the compiler needs to generate code for this expression, it doesn't know what the programmer intended
- So what do we mean when we write $a-b-c$?
 - In mathematics, this only has one possible meaning
 - It's $(a-b)-c$, since subtraction is *left-associative*
 - $a-(b-c)$ would be the meaning if subtraction was *right-associative*

Another Example: If-Then-Else

$\langle \text{stmt} \rangle ::= \langle \text{assignment} \rangle \mid \langle \text{if-stmt} \rangle \mid \dots$

$\langle \text{if-stmt} \rangle ::= \text{if} (\langle \text{expr} \rangle) \langle \text{stmt} \rangle \mid$

$\text{if} (\langle \text{expr} \rangle) \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle$

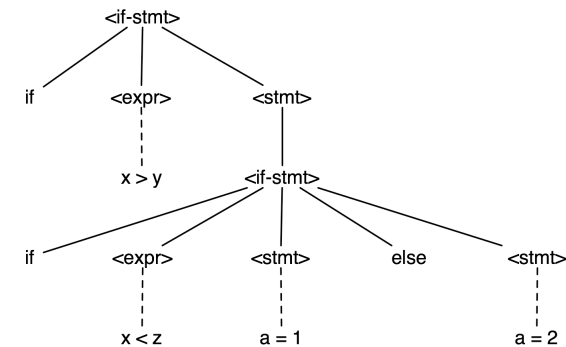
- (Here $\langle \rangle$'s are used to denote nonterminals and $::=$ for productions)

- Consider the following program fragment:

```
if (x > y)
  if (x < z)
    a = 1;
  else a = 2;
```

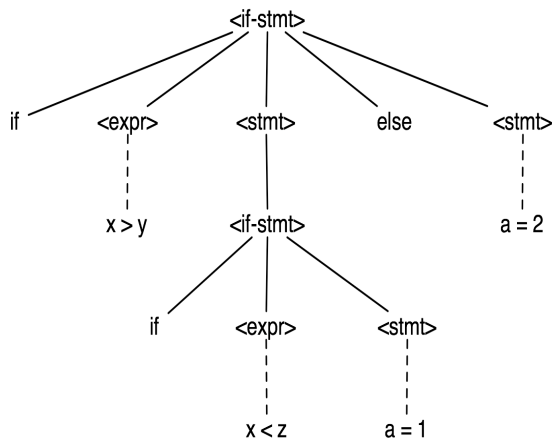
- Note: Ignore newlines

Parse Tree #1



- Else belongs to inner if

Parse Tree #2



- Else belongs to outer if

Fixing the Expression Grammar

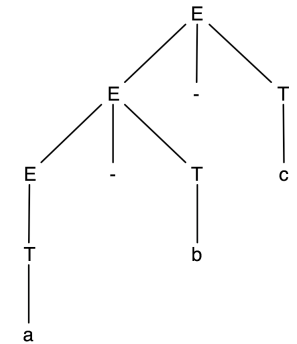
- Idea: Require that the right operand of all of the operators is not a bare expression

$$E \rightarrow E+T \mid E-T \mid E*T \mid T$$

$$T \rightarrow a \mid b \mid c \mid (E)$$

- Now there's only one parse tree for $a-b-c$

- Exercise: Give a derivation for the string $a-(b-c)$



What if We Wanted Right-Associativity?

- Left-recursive productions are used for left-associative operators
- Right-recursive productions are used for right-associative operators

- Left:

$$E \rightarrow E+T \mid E-T \mid E*T \mid T$$

$$T \rightarrow a \mid b \mid c \mid (E)$$

- Right:

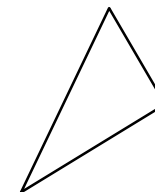
$$E \rightarrow T+E \mid T-E \mid T*E \mid T$$

$$T \rightarrow a \mid b \mid c \mid (E)$$

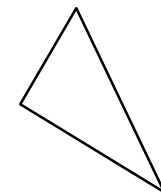
Parse Tree Shape

- The kind of recursion/associativity determines the shape of the parse tree

left recursion



right recursion



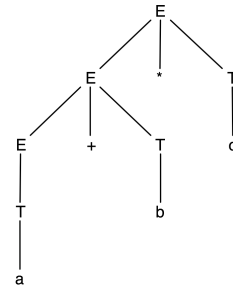
- Exercise: draw a parse tree for $a-b-c$ in the prior grammar in which subtraction is right-associative

A Different Problem

- How about the string $a+b*c$?

$E \rightarrow E+T \mid E-T \mid E*T \mid T$

$T \rightarrow a \mid b \mid c \mid (E)$



- Doesn't have correct precedence for $*$
 - When a nonterminal has productions for several operators, they effectively have the same precedence
- How can we fix this?

Final Expression Grammar

$E \rightarrow E+T \mid E-T \mid T$

$T \rightarrow T*P \mid P$

$P \rightarrow a \mid b \mid c \mid (E)$
(parentheses)

lowest precedence operators

higher precedence

highest precedence

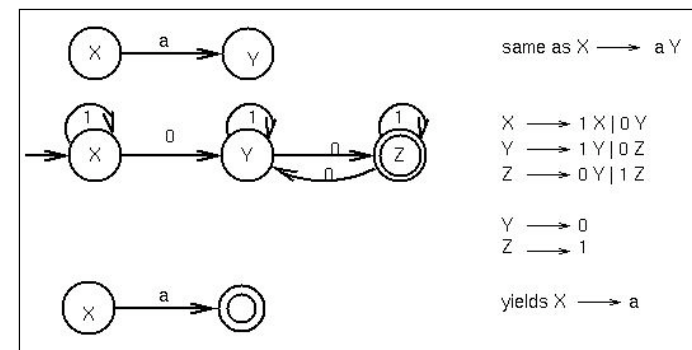
- Exercises:
 - Construct tree and left and right derivations for
 - $a+b*c$ $a*(b+c)$ $a*b+c$ $a-b-c$
 - See what happens if you change the last set of productions to $P \rightarrow a \mid b \mid c \mid E \mid (E)$
 - See what happens if you change the first set of productions to $E \rightarrow E+T \mid E-T \mid T \mid P$

Regular expressions and CFGs

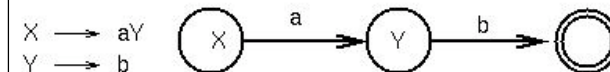
	Description	Machine
regular languages	regular expressions	DFAs, NFAs
context-free languages	context-free grammars	pushdown automata (PDAs)

- Programming languages are neither regular nor context-free
 - Usually almost context-free, with some hacks

Equivalence of DFA and regular grammars



To go from regular grammar to FSA, make the following transformations:



Pushdown Automaton (PDA)

- A **pushdown automaton** (PDA) is an abstract machine similar to the DFA
 - Has a finite set of states
 - Also has a *pushdown stack*
- Moves of the PDA are as follows:
 - An input symbol is read and the top symbol on the stack is read
 - Based on both inputs, the machine
 - Enters a new state, and
 - Writes zero or more symbols onto the pushdown stack
 - String accepted if the stack is empty at end of string

Power of PDAs

- PDAs are more powerful than DFAs
 - $a^n b^n$, which cannot be recognized by a DFA, can easily be recognized by the PDA
 - Stack all **a** symbols and, for each **b**, pop an **a** off the stack.
 - If the end of input is reached at the same time that the stack becomes empty, the string is accepted
- As with NFA, we can also have a NDPDA
 - NDPDA are more powerful than DPDA
 - NDPDA can recognize even length palindromes over $\{0,1\}^*$, but a DPDA cannot. Why? (Hint: Consider palindromes over $\{0,1\}^2\{0,1\}^*$)
- It is true, but less clear, that the languages accepted by NDPDAs are equivalent to the context-free languages

Context-free Grammars in Practice

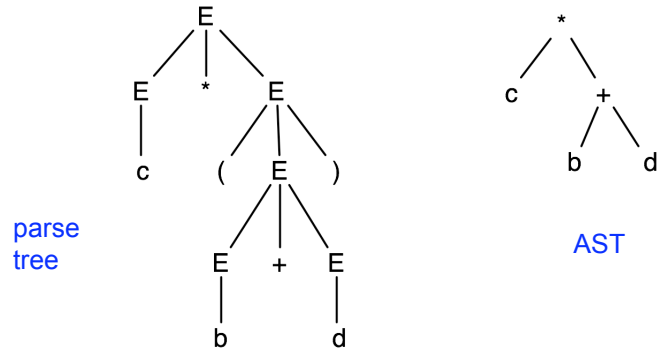
- Regular expressions are used to turn raw text into a string of *tokens*
 - E.g., “if”, “then”, “identifier”, etc.
 - Whitespace and comments are simply skipped
 - These tokens are the input for the next phase of compilation
 - Standard tools used are lex and flex
- CFGs are used to turn tokens into parse trees
 - This process is called *parsing*
 - Standard tools used are yacc and bison
- Those trees are then analyzed by the compiler, which eventually produces object code

Using Parse Trees

- Parse trees contain too much information
 - E.g., they have parentheses and they have extra nonterminals for precedence
 - This extra stuff is needed for parsing
- But when we want to *reason* about languages, it gets in the way (it's 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



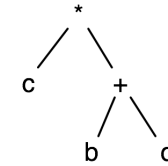
CMSC 330

49

ASTs (cont'd)

- Intuitively, ASTs correspond to the data structure you'd use to represent strings in the language
 - Note that grammars describe trees, and so do OCaml datatypes
 - $E \rightarrow a \mid b \mid c \mid E+E \mid E-E \mid E*E \mid (E)$

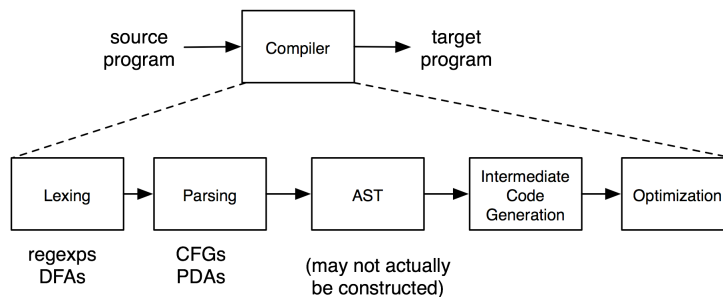
```
type ast =  
  Letter of char  
| Plus of ast * ast  
| Minus of ast * ast  
| Times of ast * ast
```



CMSC 330

50

The Compilation Process



CMSC 330

51

Parsing

- There are many efficient techniques for turning strings into parse trees or ASTs
 - They all have strange names, like LL(k), SLR(k), LR(k)...
 - Take CMSC 430 for more details
- We will look at one very simple technique:
recursive descent parsing
 - This is a "top-down" parsing algorithm because we're going to begin at the start symbol and try to produce the string

CMSC 330

52

Example

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

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

– Here n is an integer and id is an identifier

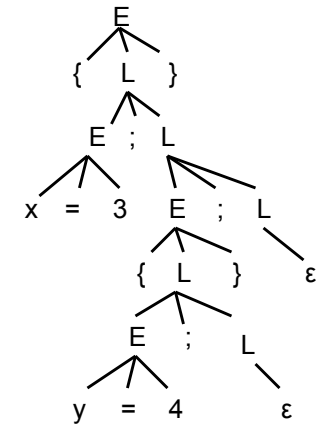
- One input might be
 - $\{ x = 3 ; \{ y = 4 ; \} ; \}$
 - This would get turned into a list of tokens
 $\{ x = 3 ; \{ y = 4 ; \} ; \}$
 - And we want to turn it into a parse tree

Example (cont'd)

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

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

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



Parsing Algorithm

- Goal: determine if we can produce a string to be parsed from the grammar's start symbol
- At each step, we'll keep track of two facts
 - What tree node are we trying to match?
 - What is the *next* token (*lookahead*) of the input string?
- There are three cases:
 - If we're trying to match a terminal and the next token (lookahead) is that token, then succeed, advance the lookahead, and continue
 - If we're trying to match a nonterminal then pick which production to apply based on the lookahead
 - Otherwise, fail with a parsing error

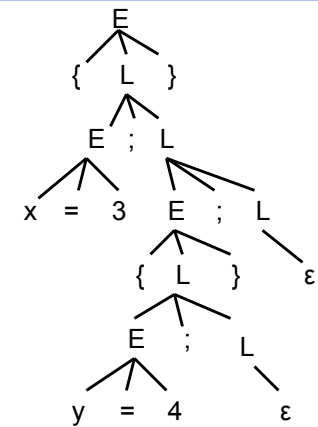
Example (cont'd)

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

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

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

↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑
lookahead



Definition of First(γ)

- **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

Definition of First(γ), cont'd

- For a terminal a , **First(a)** = { a }
- For a nonterminal N :
 - If $N \rightarrow \epsilon$, then add ϵ to **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 **First($\alpha_1 \alpha_2 \dots \alpha_n$)** to **First(N)**, where **First($\alpha_1 \alpha_2 \dots \alpha_n$)** is defined as
 - **First(α_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 , $1 \leq i \leq k$, then add ϵ to **First(N)**

Examples

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

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

First(id) = { id }

First("=") = { "=" }

First(n) = { n }

First("{") = { "{" }

First("}") = { "}" }

First(";") = { ";" }

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

First(L) = { id, "{", ϵ }

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

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

First(id) = { id }

First("=") = { "=" }

First(n) = { n }

First("{") = { "{" }

First("}") = { "}" }

First(";") = { ";" }

First(E) = { id, "{", ϵ }

First(L) = { id, "{", ";", ϵ }

Implementing a Recursive Descent Parser

- For each terminal symbol a , create a function **parse_a**, which:
 - If the lookahead is a it consumes the lookahead by advancing the lookahead to the next token, and returns
 - Otherwise fails with a parse error if the lookahead is not a
- For each nonterminal N , create a function **parse_N**
 - This function is 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 process

Implementing a Recursive Descent Parser, con't.

- 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 `First(β_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, then fail with a parse error
 - Suppose $\beta_i = \alpha_1 \alpha_2 \dots \alpha_n$. Then call `parse_ α_1 ()`; ... ; `parse_ α_n ()` to match the expected right-hand side, and return

CMSC 330

61

Example

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

```
let parse_term t =  
  if !lookahead = t  
  then lookahead := <next token>  
  else raise <Parse error>
```

```
let rec parse_E () =  
  if lookahead = 'id' then begin  
    parse_term 'id';  
    parse_term '=';  
    parse_term 'n'  
  end  
  else if lookahead = '{' then begin  
    parse_term '{';  
    parse_L ();  
    parse_term '}';  
  end  
  else raise <Parse error>;
```

(not quite
valid OCaml)

CMSC 330

62

Example (cont'd)

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

mutually recursive with previous `let rec`

```
and parse_L () =  
  if lookahead = 'id' || lookahead = '{' then begin  
    parse_E ();  
    parse_term ';';  
    parse_L ()  
  end  
  (* else return (not an error) *)
```

CMSC 330

63

Things to Notice

- If you draw the execution trace of the parser as a tree, then you get the parse tree
- This parsing strategy may fail on certain grammars because the `First` sets overlap
 - This doesn't mean the grammar is not usable in a parser, just not in this type of parser
- This is a *predictive* parser because we use the lookahead to determine exactly which production to use

CMSC 330

64

More on Limitations

- Consider parsing the grammar $E \rightarrow E + n \mid n$
 - $\text{First}(E) = n = \text{First}(n)$, so we can't use this technique
 - Exercise: Rewrite this grammar so it becomes amenable to our parsing technique
- How about the grammar $S \rightarrow Sa \mid \epsilon$
 - $\text{First}(Sa) = a$, so we're ok as far as which production
 - But the body of `parse_S()` has an infinite loop
 - if (lookahead = "a") then `parse_S()`
 - This technique cannot handle left-recursion
 - Exercise: rewrite this grammar to be right-recursive

Producing an AST

- To produce an AST, we modify the `parse()` functions to construct the AST along the way

```
type ast =  
  Assn of string * int  
  | Block of ast list
```

```
let parse_term t =  
  let temp = !lookahead in  
  if lookahead = t  
  then lookahead := <next token>  
  else raise <Parse error>;  
  temp
```

Producing an AST (cont'd)

```
type ast =  
  Assn of string * int  
  | Block of ast list
```

```
let rec parse_E () =  
  if lookahead = 'id' then  
    let id = parse_term 'id' in  
    let _ = parse_term '=' in  
    let n = parse_term 'n' in  
    Assn(id, int_of_string n)  
  else if lookahead = '{' then begin  
    let _ = parse_term '{' in  
    let l = parse_L () in  
    let _ = parse_term '}' in  
    Block l  
  end  
  else raise <Parse error>;
```

Producing an AST (cont'd)

```
type ast =  
  Assn of string * int  
  | Block of ast list
```

```
and parse_L () =  
  if lookahead = 'id' then  
    let e = parse_E () in  
    let _ = parse_term ';' in  
    let l = parse_L () in  
    e::l  
  else []
```

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
- More powerful techniques need tool support, such as yacc and bison (which can be slower)
- Recursive descent is good for a quick hack
 - Though using the tools is pretty fast if you're familiar with them

General parsing algorithms

- NDPDA and DPDA do not accept the same languages.
 - (Remember that DFA and NFA **do** accept the same sets.)
- Knuth in 1965 showed that the deterministic PDAs were equivalent to a class of grammars called LR(k) [Left-to-right parsing with k symbol lookahead]
 - Create a PDA that decides whether to stack the next symbol or pop a symbol off the stack by looking k symbols ahead.
 - This is a deterministic process, and for k=1 is efficient.
- LR(k), SLR(k) [Simple LR(k)], and LALR(k) [Lookahead LR(k)] are all techniques used today to build efficient parsers.
 - Recursive descent is a form of LL(k) parsing
 - More in CMSC 430 ...