Formal Languages, Grammars, and Parsing

CHAPTER 2 OF PROGRAMMING LANGUAGES PRAGMATICS

Outline

- **Languages and Grammars**
- Regular Languages (Scanning)
- Context Free Languages (Parsing)
  - Derivation/Parse Trees
  - Recursive Descent Parsing
  - Ambiguity
  - LL and LR Grammars

Languages

- Syntax vs Semantics
  - Syntax - form, internal structure
  - Semantics - meaning
- Language Generation vs Language Recognition
  - Regular expressions and grammars
  - Scanner and parsers

Formal Languages

- Basis for the design and implementation of programming languages
- Alphabet: finite set Σ of symbols
- String: finite sequence of symbols
  - ϵ : the empty string, a sequence of length 0
  - Σ^* : the set of all strings over Σ, including ε
  - Σ^+ : the set of all non-empty strings over Σ
- Language: a set of strings $L \subseteq \Sigma^*$
  - For example in Java, Σ is Unicode, a program is a string, and L is defined by a grammar in the language specification

Formal Grammars

- A grammar $G = (N, \Sigma, P, S)$ is a 4-tuple with
  - Finite set of non-terminal symbols N
  - Finite set of terminal symbols/an alphabet Σ
  - Finite set of productions P
  - The starting non-terminal symbol $S \in N$
- The grammar describes a language $L \subseteq \Sigma^*$
  - Sometimes say that a grammar *generates* a language
- Productions are “rules” of the form $x \rightarrow y$
  - x is a non-empty sequence of terminals and non-terminals
  - y is a sequence of terminals and non-terminals

Example: A grammar for non-negative integers

- $N = \{ I, D\}$
- $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- $S = I$
- $P = \{ \; I \rightarrow D, I \rightarrow DI,$
  $D \rightarrow 0, D \rightarrow 1, D \rightarrow 2, D \rightarrow 3, D \rightarrow 4,$
  $D \rightarrow 5, D \rightarrow 6, D \rightarrow 7, D \rightarrow 8, D \rightarrow 9 \; \}$
- Writing out the grammar seems pretty tedious, right?

Grammar notation in practice

- This is how the same grammar would typically be presented:
- $I \rightarrow D \mid DI$
  $D \rightarrow 0 \mid 1 \mid 2 \mid \ldots \mid 9$
- Here I has two production alternatives, D has 10
- We can infer the rest of the grammar from this
  - Our terminals/alphabet are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  - Our non-terminals are I and D
  - Our starting non-terminal is I

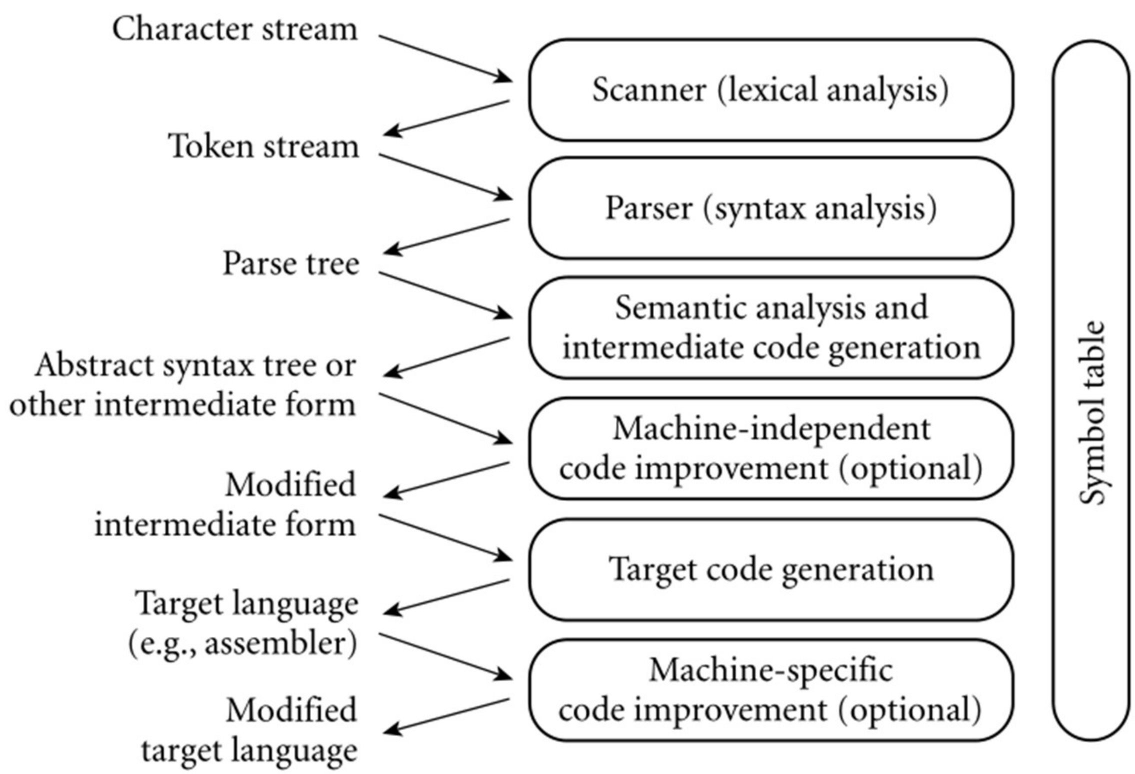
Grammar for non-negative integers

- We generate the strings in the language from the starting non-terminal by applying the production rules
  - Match the left-hand side of a rule to a substring and replacing with the right-hand side
- Example:
  - Grammar:
    $I \rightarrow D \mid DI$
    $D \rightarrow 0 \mid 1 \mid 2 \mid \ldots \mid 9$
  - $I \Rightarrow DI$
    $DI \Rightarrow DDI$
    $DDI \Rightarrow D6I$
    $D6I \Rightarrow D6D$
    $D6D \Rightarrow 36D$
    $36D \Rightarrow 361$

Languages and Grammars

- String derivation (let w_i be a string)
  - $w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_n$ is a **derivation sequence**
  - We can indicate that there is a derivation sequence from w_1 to w_n by writing $w_1 \Rightarrow^* w_n$
  - If we want to be clear that $n > 1$ we can write $w_1 \Rightarrow^+ w_n$
- We want to talk about the **language generated by grammar G**
  - $L(G) = \{ w \in \Sigma^* \mid S \Rightarrow^+ w \}$
- Fundamental theoretical characterization of languages: Chomsky Hierarchy
  - Regular languages ⊂ Context-free languages ⊂ Context-sensitive languages ⊂ Unrestricted languages
- Our interest as computer scientists:
  - Regular languages for lexical analysis
  - Context free languages for syntax analysis

Recall the overview of compilation



Regular languages in compilers & interpreters

stream of **characters**

w,h,i,l,e,(,a,1,5,>,b,b,),d,o,...

Scanner (uses a regular grammar to perform lexical analysis)

stream of **tokens**

keyword[while], **leftparen**, **id**[a15], **op**[>], **id**[bb], **rightparen**, **keyword**[do], ...

Parser (uses a context-free grammar to perform syntax analysis)

parse tree

each token is a leaf in the parse tree

... more compiler/interpreter components

Outline

- Languages and Grammars
- **Regular Languages (Scanning)**
- Context Free Languages (Parsing)
  - Derivation/Parse Trees
  - Recursive Descent Parsing
  - Ambiguity
  - LL and LR Grammars

Regular languages

- Regular languages – languages that can be recognized by a read-only Turing machine
- Operations on languages that preserve regularity
  - Union: $L \cup M$ = all strings in L or M
  - Concatenation: LM = all strings ab where $a \in L$ and $b \in M$
  - Closure and positive closure: Let $L^0 = \{\, \epsilon \,\}$ and $L^i = L^{i-1}L$
    - Closure: $L^* = L^0 \cup L^1 \cup L^2 \cup \cdots$
    - Positive Closure: $L^+ = L^1 \cup L^2 \cup \cdots$
- Regular expressions: notation to express languages constructed with the help of these operations
    - Example: (0|1|2|3|4|5|6|7|8|9)+

Regular languages

- Given some alphabet, a regular expression is
  - The empty string ϵ
  - Any symbol from the alphabet
  - If r and s are regular expressions, then so are r|s, rs, r*, r+, r?, and (r)
    - r|s – accepts strings accepted by r or accepted by s (set union)
    - rs – accepts strings that are the concatenation of a string accepted by r and a string accepted by s (set concatenation)
    - r* - accepts ϵ or more strings accepted by r, all concatenated together (set closure)
    - r+ - accepts 1 or more strings accepted by r, all concatenated together (positive closure)
    - r? – accepts ϵ or 1 string accepted by r
  - */+/? have higher precedence than concatenation, and concatenation has higher precedence than |
  - All operations are left associative

Regular expressions

- Each regular expression r defines a language L(r)
  - $L(\epsilon) = \{ \epsilon \}$
  - $L(a) = \{ a \}$ for alphabet symbol a
  - $L(r|s) = L(r) \cup L(s)$ for regular expressions r, s
  - $L(rs) = L(r)L(s)$
  - $L(r^*) = \left(L(r)\right)^*$
  - $L(r^+) = \left(L(r)\right)^+$
  - $L(r?) = \{ \epsilon \} \cup L(r)$
  - $L\big((r)\big) = L(r)$
- Example: What is the language defined by this regular expression?
  0(x|X)(0|1|2|3|4|5|6|7|8|9|a|b|c|d|e|f|A|B|C|D|E|F)+

Regular expression exercises

- L(gray|grey) = ?
- L(gr(a|e)y) = ?
- L(colou?r) = ?
- L(go*gle) = ?

- Try these on your own (answers on the next slide)

Regular expression exercises

- L(gray|grey) = {gray, grey}
- L(gr(a|e)y) = {gray, grey}
- L(colou?r) = {color, colour}
- L(go*gle) = {ggle, gogle, google, gooogle, goooogle, ...}

Regular expression exercises

- Which of the following languages over the alphabet {0, 1} is described by the regular expression
  (0+1)*0(0+1)*0(0+1)*
  - Set of all strings containing 00
  - Set of all strings containing at most two 0's
  - Set of all strings containing at least two 0's
  - Set of all strings that begin and end with either 0 or 1
  - None of these
- Try this out, answer on next slide.
  - I suggest trying to come up with strings that will eliminate answers, i.e. if you find a string that contains 00 but is not accepted by the regular expression that would eliminate the first option

Regular expression exercises

- Which of the following languages over the alphabet {0, 1} is described be the regular expression
  $(0+1)^*0(0+1)^*0(0+1)^*$
  - Set of all strings containing 00
    - 001 is in this set, not accepted by the RE
  - Set of all strings containing at most two 0's
    - 001 is in this set, not accepted by the RE
  - Set of all strings containing at least two 0's
    - 001 is in this set, not accepted by the RE
  - Set of all strings that begin and end with either 0 or 1
    - 0 is in this set, not accepted by the RE
  - **None of these**

Regular expression exercises

- The regular expression 0*(10*)* is equivalent to
  - (1*0)*1*
  - 0+(0+10)*
  - (0+1)*10(0+1)*
  - None of these

- Answer on the next slide

Regular expression exercises

- The regular expression 0*(10*)* is equivalent to
  - **(1*0)*1***
  - 0+(0+10)*
  - (0+1)*10(0+1)*
  - None of these

Regular languages

- Let G be a grammar
- We say G is a regular grammar if
  - Either all productions are of the form $A \to wB$ and $A \to w$, where
    - A, B are non-terminals in G
    - w is a sequence of terminals or the empty string
  - Or all productions are of the form $A \to Bw$ and $A \to w$, where
    - A, B are non-terminals in G
    - w is a sequence of terminals or the empty string
- In addition, in the first case we say G is **right-regular**, and in the second case we say G is **left-regular**
- If a language is generated by a regular grammar, then we say the language is regular

Regular grammars

- Example: $L = \{\ a^n b \mid n > 0\ \}$ is a regular language
  - Grammar to prove this:

$$S \rightarrow Ab$$
$$\mathrm{A} \rightarrow Aa \mid a$$

  this is a left-regular grammar

- What about our grammar from before for non-negative integers?

$$\mathrm{I} \rightarrow D \mid DI$$
$$D \rightarrow 0 \mid 1 \mid 2 \mid \ldots \mid 9$$

Regular grammars

- $\mathrm{I} \rightarrow D \mid DI$
  $D \rightarrow 0 \mid 1 \mid 2 \mid \ldots \mid 9$
- Can we modify this grammar so it is regular? And while we are at it, can we eliminate the leading 0's the grammar currently allows?
  - I.e. 509 is allowed, but 059 is not

Regular Languages

- Equivalaent formalisms for regular languages
  - Regular expressions
  - Regular grammars
  - Nondeterministic finite automata
  - Deterministic finite automota
  - Additional details: Sections 2.2, 2.4 (or Wikipedia)
- Our interest?
  - Regular languages are the foundation for the lexical analysis done by a scanner
  - Your first project will be to implement a scanner

Regular languages in compilers & interpreters

stream of **characters**

w,h,i,l,e,(,a,1,5,>,b,b,),d,o,...

Scanner (uses a regular grammar to perform lexical analysis)

stream of **tokens**

keyword[while], **leftparen**, **id**[a15], **op**[>], **id**[bb], **rightparen**, **keyword**[do], ...

Parser (uses a context-free grammar to perform syntax analysis)

parse tree

each token is a leaf in the parse tree

... more compiler/interpreter components

Outline of a simple scanner

- The parser asks the scanner for the next token
- The scanner reads the next character x from the character stream
- If x is a special character like ; , + - * ( ) the scanner returns the corresponding token
  - If x is =, peek at the next character y
    - If y is not =, return the token for = (ASSIGN)
    - If y is =, read the characer and return the token for == (EQUAL)

Outline of a simple scanner

- Some tokens have additional information attached
- When parsing we will want the type and the specific value (i.e. identifier token and identifier string, or constant token and constant value)
- If x is a letter, keep reading characters and stop before the first non-letter or non-digit character (assuming our language requires identifiers to start with a letter and then have only letters or digits)
  - If the string is a keyword, return the token for that keyword
  - If the string is a not a keyword, return the id token with the string
- If x is a digit, keep reading characters and stop before the first non-digit character
  - Return the token const with the numeric value

Outline

- Languages and Grammars
- Regular Languages (Scanning)
- **Context Free Languages (Parsing)**
  - Derivation/Parse Trees
  - Recursive Descent Parsing
  - Ambiguity
  - LL and LR Grammars

Context-free languages

- They subsume regular languages
  - Every regular language is CF
  - Not every CF language is regular
    - Is $L = \{ a^n b^n \mid n > 0 \}$ regular?
- CF languages are generated by a context-free grammar (CFG)
  - Each production must be of the form $A \to w$, where
    - A is a nonterminal
    - w is a sequence of terminals **and non-terminals**

CFG notation

- For CFGs, we traditionally use Backus-Naur form (BNF)
  - John Backus and Peter Naur, for Algol-58 and Algol 60
- ::= is read as "defined as"
- < > encloses a nonterminal name
- Example, a grammar fragment:
  $E \rightarrow T \mid E + T$
  $T \rightarrow i \mid c \mid (E)$
- Becomes:
  <expr> ::= <term> | <expr> + <term>
  <term> ::= id | const | (<expr>)

Outline

- Languages and Grammars
- Regular Languages (Scanning)
- Context Free Languages (Parsing)
  - **Derivation/Parse Trees**
  - Recursive Descent Parsing
  - Ambiguity
  - LL and LR Grammars

Derivation sequence for a string

- Describes a particular way to derive a string based on a CFG
  - A derivation is a sequence of replacement operations that show how to derive a string from the starting symbol
  - A sentential form is each string of symbols in the derivation
  - The yield of the derivation is the string of terminals at the end of the derivation

Example of a derivation sequence

<expr> ::= <term> | <expr> + <term>

<term> ::= id | (<expr>)

Derivation sequence for (x+y)+z

<expr> ⇒ <expr>+<term> ⇒ <expr>+z ⇒ <term>+z ⇒ (<expr>)+z
⇒ (<expr>+<term>)+z ⇒ (<expr>+y)+z ⇒ (<term>+y)+z
⇒ (x+y)+z

Derivation tree for a string

- Also called a parse tree or a concrete syntax tree
  - Leaf nodes: terminals
  - Inner nodes: non-terminals
  - Root: starting non-terminal of the grammar
- Describes a particular way to derive a string based on a CFG
  - Leaf nodes read from left to right are the string
  - To get this string: depth first traversal of the tree, always visiting the leftmost unexplored branch

Example of a derivation tree

<expr> ::= <term> | <expr> + <term>

<term> ::= id | (<expr>)

Parse tree for (x+y)+z

Equivalent derivation sequences

Two derivation sequences are equivalent if they have the same parse tree

<u>One derivation:</u>

<expr> ⇒ <expr>+<term> ⇒ <expr>+z ⇒ <term>+z ⇒ (<expr>)+z ⇒ (<expr>+<term>)+z ⇒ (<expr>+y)+z ⇒ (<term>+y)+z
⇒ (x+y)+z

<u>Another derivation:</u>

<expr> ⇒ <expr>+<term> ⇒ <term>+<term> ⇒ (<expr>)+<term> ⇒
(<expr>+<term>)+<term> ⇒ (<term>+<term>)+<term> ⇒
(x+<term>)+<term> ⇒ (x+y)+<term> ⇒ (x+y)+z

And there are many more

Outline

- Languages and Grammars
- Regular Languages (Scanning)
- Context Free Languages (Parsing)
  - Derivation/Parse Trees
  - **Recursive Descent Parsing**
  - Ambiguity
  - LL and LR Grammars

The Core grammar

<prog> ::= **program** <decl-seq> **begin** <stmt-seq> **end**

<decl-seq> ::= <decl> | <decl><decl-seq>

<stmt-seq> ::= <stmt> | <stmt><stmt-seq>

<decl> ::= **int** <id-list> **;**

<id-list> ::= **id** | **id ,** <id-list>

<stmt> ::= <assign> | <if> | <loop> | <in> | <out> | <decl>

<assign> ::= **id =** <expr> **;**

<in> ::= **input** <id> **;**

<out> ::= **output** <expr> **;**

<if> ::= **if** <cond> **then** <stmt-seq> **endif ;**

| **if** <cond> **then** <stmt-seq> **else** <stmt-seq> **endif ;**

<loop> ::= **while** <cond> **begin** <stmt-seq> **endwhile ;**

<cond> ::= <cmpr> | **! (** <cond> **)**

| <cmpr> **or** <cond>

<cmpr> ::= <expr> **==** <expr> | <expr> **<** <expr>

| <expr> **<=** <expr>

<expr> ::= <term> | <term> **+** <expr> | <term> **–** <expr>

<term> ::= <factor> | <factor> ***** <term>

<factor> ::= **const** | **id** | **(** <expr> **)**

Parser vs scanner projects

- id and const are terminal symbols for the grammar of the Core language
  - In the scanner they were non-terminals
- Consider 9-5+5 in the Core grammar
  - What is the parse tree? Is this a problem?
  - If we wanted to fix this, how could we?

Recursive descent

- Several uses
  - Parsing technique
    - Call the scanner to obtain tokens, build the parse tree from the root down
  - Traversal of a given parse tree
    - Parse tree represents a program, traverse it to
      - Print the program (check we have the correct parse tree)
      - Code generation
      - Interprete the program
- Basic idea: Use a separate procedure for each non-terminal of the grammar
  - The body of the procedure “applies” some production for that non-terminal
  - Start by calling the procedure for the starting non-terminal

Example: Simple expressions

<expr> ::= <term> | <term> + <expr>

<term> ::= id | const | (<expr>)

```
procedure Expr() {
        Term();
        if (currentToken() == PLUS) {
                nextToken(); //consume the plus
                Expr();
        }
}
```

Ignore error checking for now...

Example: Simple expressions

<expr> ::= <term> | <term> + <expr>

<term> ::= id | const | (<expr>)

```
procedure Term() {
        if (currentToken() == ID) nextToken();
        else if (currentToken() == CONST) nextToken();
        else if (currentToken == LPAREN) {
                nextToken(); // consume left parenthesis
                Expr();
                nextToken(); // consume right parenthesis
        }
}
```

Error checking

- What checks of currentToken() do we need to make in Term()?
  - For example, how to catch malform strings like “+a” and “(a+b”
- Unexpected leftover tokens: add “metatokens” to facilitate communication between scanner and parser
  - Parser should check for EOS token

Which alternative to use?

- The key issue for recursive descent parsing, how to pick which production alternative from the grammar to follow?
  - Predictive parsing: predict correctly (without backtracking) what we need to do, by looking ahead a few tokens
  - Ideally, look at just one token (the current one)
- For each production alternative, find the FIRST set: what is the set of all terminals that can be at the very beginning of strings derived from that alternative?
- If the FIRST sets are all disjoint, we can decide exactly which alternative we should use

FIRST set

<decl-seq> ::= <decl> | <decl><decl-seq>
<decl> ::= int <id-list> ;

- FIRST set for the <decl-seq> alternatives: { INT } and { INT }
  - Not disjoint!
- Make them disjoint by introducing a helper non-terminal <decl-rest>
  <decl-seq> ::= <decl><decl-rest>
  <decl-rest> ::= <decl-seq> | ϵ
  <decl> ::= int <id-list> ;
- FIRST sets for alternatives of <decl-rest>
  - { INT } for <decl-seq>
  - { BEGIN } for ϵ, because <prog> ::= program <decl-seq> begin ...

Parser code

```
procedure DeclSeq() {
        ...
        Decl();
        DeclRest();
        ...}
procedure DeclRest() {
        ...
        if (currentToken() == BEGIN) return;
        if (currentToken() == INT) { ... DeclSeq(); ... }
}
```

Parser code simplified

You may have realized that was unnecessary, can do this instead:

```
procedure DeclSeq() {
        ...
        Decl();
        ...
        if (currentToken() == BEGIN) return;
        if (currentToken() == INT) {
                ...
                DeclSeq();
                ...
        }
}
```

Practice with FIRST sets

For each of these, find the FIRST set for all production alternatives

- <id-list> ::= id | id, <id-list>
- <stmt> ::= <assign> | <if> | <loop> | <in> | <out> | <decl>
- <stmt-seq> ::= <stmt> | <stmt><stmt-seq>
- <cond> ::= <cmpr | !(<cond>) | <cmpr> or <cond>
  <cmpr> ::= <expr> == <expr> | <expr> < <expr> | <expr> <= <expr>
- <expr> ::= <term> | <term> + <expr> | <term> - <expr>
  <term> ::= <factor> | <factor> * <term>
  <factor> ::= const | id | (<expr>)

The Core grammar

<prog> ::= **program** <decl-seq> **begin** <stmt-seq> **end**

<decl-seq> ::= <decl> | <decl><decl-seq>

<stmt-seq> ::= <stmt> | <stmt><stmt-seq>

<decl> ::= **int** <id-list> **;**

<id-list> ::= **id** | **id ,** <id-list>

<stmt> ::= <assign> | <if> | <loop> | <in> | <out> | <decl>

<assign> ::= **id =** <expr> **;**

<in> ::= **input** <id> **;**

<out> ::= **output** <expr> **;**

<if> ::= **if** <cond> **then** <stmt-seq> **endif ;**

| **if** <cond> **then** <stmt-seq> **else** <stmt-seq> **endif ;**

<loop> ::= **while** <cond> **begin** <stmt-seq> **endwhile ;**

<cond> ::= <cmpr> | **! (** <cond> **)**

| <cmpr> **or** <cond>

<cmpr> ::= <expr> **==** <expr> | <expr> **<** <expr>

| <expr> **<=** <expr>

<expr> ::= <term> | <term> **+** <expr> | <term> **–** <expr>

<term> ::= <factor> | <factor> ***** <term>

<factor> ::= **const** | **id** | **(** <expr> **)**

Practice with FIRST sets answers

For each of these, find the FIRST set for all production alternatives

- <id-list> ::= id | id, <id-list>
  - {id} and {id}
- <stmt> ::= <assign> | <if> | <loop> | <in> | <out> | <decl>
  - {id}, {IF}, {WHILE}, {INPUT}, {OUTPUT}, {INT}
- <stmt-seq> ::= <stmt> | <stmt><stmt-seq>
  - {id, if, while, input, output, int} and {id, if, while, input, output, int}
- <cond> ::= <cmpr> | !(<cond>) | <cmpr> or <cond>
<cmpr> ::= <expr> == <expr> | <expr> < <expr> | <expr> <= <expr>
  - For <cond>: { const, id, LPAREN }, { NEGATION }, { const, id, LPAREN }
  - For <cmpr>: { const, id, LPAREN }, { const, id, LPAREN }, { const, id, LPAREN }
- <expr> ::= <term> | <term> + <expr> | <term> - <expr>
<term> ::= <factor> | <factor> * <term>
<factor> ::= const | id | (<expr>)
  - For <expr>: { const, id, LPAREN }, { const, id, LPAREN }, { const, id, LPAREN }
  - For <term>: { const, id, LPAREN }, { const, id, LPAREN }
  - For <factor>: {const}, {id}, {LPAREN}

More general parsing

- We have
  <expr> ::= <term> | <term> + <expr> | <term> - <expr>
- Why not instead use
  <expr> ::= <term> | <expr> + <term> | <expr> - <term>
- Left recursive grammars are not suitable for predictive, recursive descent parsing
- What is used in real parsers?
  - Traditionally, bottom-up parsing with LR(k) grammars?
  - Several modern compilers with recursive descent parsing

Outline

- Languages and Grammars
- Regular Languages (Scanning)
- Context Free Languages (Parsing)
  - Derivation/Parse Trees
  - Recursive Descent Parsing
  - **Ambiguity**
  - LL and LR Grammars

Ambiguous grammars

- A grammar is **ambiguous** ⇔ There is some string with multiple (different) parse trees
- An ambiguous grammar gives more freedom to the compiler writer
  - For example, ambiguity can allow optimizations for better performance
  - This also creates difficulty, how to choose between alternatives
- For real work programming languages, we typically have non-ambiguous grammar
  - We need a deterministic specification for the parser so the user knows what their code will do
- We can remove ambiguity by adding more non-terminals

Eliminating ambiguity

- Consider this grammar:
  <expr> ::= <expr> + <expr> | <expr> * <expr> | id | (<expr>)
- How many parse trees are there for the string “a + b * c”?

Eliminating ambiguity

- Consider this grammar:
  <expr> ::= <expr> + <expr> | <expr> * <expr> | id | (<expr>)
- How many parse trees are there for the string "a + b * c"?
  - One is equivalent to (a + b) * c
  - The other is equivalent to a + (b * c)
- Operator precedence:
  - When several operators are without parentheses, which is an operand of what?
  - Is a+b and operand of *, or is b*c and operand of +?
- Operator associativity:
  - For multiple operators with the same precedence, are they evaluated left-to-right or right-to-left?
  - Is a-b-c equivalent to (a-b)-c or a-(b-c)?

Eliminating ambiguity

- In most languages, * has higher precedence than +, and both are left-associative
- Exercise: Change
  <expr> ::= <expr> + <expr> | <expr> * <expr> | id | (<expr>)
  So that
  - It is unambiguous
  - It has the correct precedence
  - It has the correct associativity

Eliminating ambiguity

- In most languages, * has higher precedence than +, and both are left-associative
- Exercise: Change
  <expr> ::= <expr> + <expr> | <expr> * <expr> | id | (<expr>)
  So that
  - It is unambiguous
  - It has the correct precedence
  - It has the correct associativity
- Solution: add new non-terminals
  <expr> ::= <expr> + <term> | <term>
  <term> ::= <term> * <factor> | <factor>
  <factor> ::= id | (<expr>)

Classic ambiguity problem: The “dangling-else”

- Is this a reasonable grammar for including if/then and if/then/else statements in our language?
- <stmt> ::= if <expr> then <stmt>
| if <expr> then <stmt> else <stmt>
- if a then if b then c=1 else c=2

Classic ambiguity problem: The “dangling-else”

- Is this a reasonable grammar for including if/then and if/then/else statements in our language?
- <stmt> ::= if <expr> then <stmt>
  | if <expr> then <stmt> else <stmt>
- if a then if b then c=1 else c=2
  - Two possible parse trees
- Traditional solution: match the else with the last unmatched then
  - This is an example of a disambiguation rule, something external to the grammar

Classic ambiguity problem: The “dangling-else”

- Is this a reasonable grammar for including if/then and if/then/else statements in our language?
- <stmt> ::= if <expr> then <stmt>
| if <expr> then <stmt> else <stmt>
- if a then if b then c=1 else c=2
  - Two possible parse trees, very different meanings
- Traditional solution: match the else with the last unmatched then
  - This is an example of a disambiguation rule, something external to the grammar

Classic ambiguity problem: The “dangling-else”

- Is this version non-ambiguous?
- <stmt> ::= <matched>
  | <unmatched>
  <matched> ::= <non-if-stmt>
  | if <expr> then <matched> else <matched>
  <unmatched> ::= if <expr> then <stmt>
  | if <expr> then <matched> else <unmatched>
- Is this version easy to understand?
- if a then if b then c=1 else c=2

Outline

- Languages and Grammars
- Regular Languages (Scanning)
- Context Free Languages (Parsing)
  - Derivation/Parse Trees
  - Recursive Descent Parsing
  - Ambiguity
  - **LL and LR Grammars**

Regular languages in compilers & interpreters

stream of **characters**

w,h,i,l,e,(,a,1,5,>,b,b,),d,o,...

Scanner (uses a regular grammar to perform lexical analysis)

stream of **tokens**

keyword[while], **leftparen**, **id**[a15], **op**[>], **id**[bb], **rightparen**, **keyword**[do], ...

Parser (uses a context-free grammar to perform syntax analysis)

parse tree

each token is a leaf in the parse tree

… more compiler/interpreter components

Parsing

- A CFG is a CF language generator; a **parser** is a language recognizer
- For CFLs, it is proven that we can create a parser that runs in time $O(n^3)$, where n is the length of the program
- For CFLs that admit an unambiguous grammar, $O(n^2)$
  - An example of a language with no unambiguous grammar is
$$\{ a^n b^m c^m d^n : n, m > 0 \} \cup \{a^n b^n c^m d^m : n, m > 0 \}$$
- For CFLs that admit an LL or LR grammar, $O(n)$
  - LL/LR grammars are unambiguous grammars

LL and LR grammars

Class	Direction of scanning	Derivation discovered	Parse tree construction	Algorithm
LL	Left-to-right	Leftmost	Top-down	Predictive/ Recursive descent
LR	Left-to-right	Rightmost	Bottom-up	Shift-reduce

LL/Top-down example

- <id_list> ::= <id><id_list_tail>
  <id_list_tail> ::= , <id><id_list_tail> | ;
  <id> ::= A | B | C
- Parse the string "A, B, C;"

LR/Bottom-up example

- <id_list> ::= <id_list_prefix> ;
  <id_list_prefix> ::= <id_list_prefix>, <id> | <id>
  <id> ::= A | B | C
- Parse the string “A, B, C;”

LL/LR grammars

- LL/LR grammars are further classified based on how much *look ahead* is needed for parsing
  - LL(k)/LR(k) indicates a look ahead of at most k tokens is needed to parse
- Working with LL(1) would be particularly nice, but it is difficult to write LL(1) grammars for our languages
  - Common difficulties:
    - Left recursion: <sub> ::= <sub> - <id>
    - Common prefixes: <stmt> ::= <matched> | <unmatched>
- We can get close to LL(1)
  - Handle left-associativity as a special case instead of through left-recursion
  - Use disambiguation rules, or change the language to avoid common prefixes