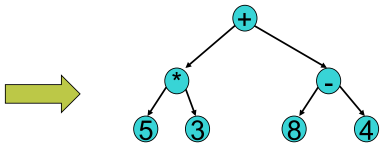
Grammars: Defining Languages

Walls & Mirrors Ch. 6.2
Rosen Ch. 13.1

Parsing

5 * 3 + (8 - 4)



1. Recognize the structure of the expression

terminology: PARSE the expression

2. Build the tree (while parsing)

Definitions

- ***Language*** is a set of strings of symbols from a finite alphabet.

  JavaPrograms = {string w : w is a syntactically correct Java program}

- ***Grammar*** is a set of rules that the strings must follow.
- ***Recognition Algorithm*** determines whether a string is a member of the language.

Basics of Grammars

Example: a Backus-Naur grammar for Java identifiers

<*identifier*> = <*letter*> | <*identifier*> <*letter*> |
<*identifier*> <*digit*> |
$<*identifier*> | _<*identifier*>

<*letter*> = a | b | …| z | A | B | … | Z

<*digit*> = 0 | 1 | … | 9

- *x* | *y* means "x or y"
- *x y* means "x followed by y"
- <*word*> is called a non-terminal, which can be replaced by other symbols depending on the rules.
- Terminals are symbols (e.g., letters, words) from which legal strings are constructed.
- Rules have the form <*word*> = …

Example

- Consider the language that the following grammar defines:

<*W*> = *xy* | *x* <*W*> *y*

Write all strings that are in this language

A. xy

B. xy, xxyy

C. xy, xyxy, xyxyxy, xyxyxyxy ….

D. xy, xxyy, xxxyyy, xxxxyyyy ….

Formally: Phrase-Structure Grammars

A phrase-structure grammar G=(V,T,S,P) consists of a vocabulary V, a subset T of V consisting of terminal elements, a start symbol S from V, and a finite set of productions P.

- Example: Let G=(V,T,S,P) where V={0,1,A}, T={0,1}, S is the start symbol and P={S->AA, A->0, A->1}.

The language generated by G is the set of all strings of terminals that are derivable from the starting state S, i.e.,

$$L(G) = \left\{ w \in T^* \mid S \stackrel{*}{\Rightarrow} w \right\}$$

Example as Phrase Structure

<*W*> = *xy* | *x* <*W*> *y*

V={x, y, W}

T={x,y}

S=W

P={W->xy, W->xWy}

Derivation (applying productions to obtain a legal string): *W->xWy, W->xxyy*

Types of Phrase-Structure Grammars

- Type 0: no restrictions on productions
- Type 1 (Context Sensitive): productions such that *w1 -> w2*, where *w1=lAr, w2=lwr, A* is a nonterminal, *l* and *r* are strings of 0 or more terminals or nonterminals and *w* is a nonempty string of terminals or nonterminals. It can have *S->λ* (empty string) provided *S* is not on any right hand side (RHS).
- Type 2 (Context Free): productions such that *w1->w2* where *w1* is a single nonterminal or *S*

Type 3: Regular Languages

- A language generated by a type 3 grammar which can have productions only of the form *A->aB* or *A->a* where *A* & *B* are non-terminals and a is a terminal.
- Regular expressions are defined recursively over a set *I*:
  - ∅ is the empty set
  - λ is the set containing the empty string
  - *x* whenever *x ε I*
  - (AB) concatenates sets A and B
  - (A U B) takes union of sets A and B
  - A* is 0 or more repetitions of elements in A
  - A+ is 1 or more repetitions of elements in A
- Example: 0(0 U 1)*

Java Identifiers

A grammar for Java identifiers:

<*identifier*> = <*letter*> | <*identifier*> <*letter*> |
<*identifier*> <*digit*> |
$<*identifier*> | _<*identifier*>

<*letter*> = a | b | …| z | A | B | … | Z

<*digit*> = 0 | 1 | … | 9

- How do we determine if a string *w* is a valid Java identifier, i.e. belongs to the language of Java identifiers?

Recognizing Java Identifiers

```
isId(in w:string):boolean
  if (w is of length 1)
     if (w is a letter)
        return true
     else
        return false
  else if (the last character of w is a letter
           or a digit)
           return isId(w minus its last character)
        else
           return false
```


Prefix Expressions

- Grammar for prefix expression (e.g., * - a b c):

<*prefix*> = <*identifier*> | <*operator*> <*prefix*> <*prefix*>

<*operator*> = + | - | * | /

<*identifier*> = a | b | … | z

Recognizing Prefix Expressions Top Down

Grammar:

<prefix> = *<identifier>* | *<operator>* *<prefix>* *<prefix>*

<operator> = + | - | * | /

<identifier> = a | b | … | z

Given "* - a b c"

1. *<prefix>*
2. *<operator>* *<prefix>* *<prefix>*
3. * *<prefix>* *<prefix>*
4. * *<operator>* *<prefix>* *<prefix>* *<prefix>*
5. * - *<prefix>* *<prefix>* *<prefix>*
6. * - *<identifier>* *<prefix>* *<prefix>*
7. * - a *<prefix>* *<prefix>*
8. * - a *<identifier>* *<prefix>*
9. * - a b *<prefix>*
10. * - a b *<identifier>*
11. * - a b c

Recognizing Prefix Expressions

```
boolean prefix() {
   if (identifier()) { // rule <prefix> = <identifier>
      return true;
   }
   else {  //<prefix> = <operator> <prefix> <prefix>
      if (operator()) {
          if (prefix()) {
             if (prefix()) {
                return true;
             }
             else { return false;}
             }
          else  { return false;}
          }
      else  { return false; }
    }
}
```


Palindromes

Palindromes = {*w* : *w* reads the same left to right as right to left}

Examples: RADAR, [A NUT FOR A JAR OF TUNA]

Recursive definition:

w is a palindrome if and only if

the first and last characters of *w* are the same

And

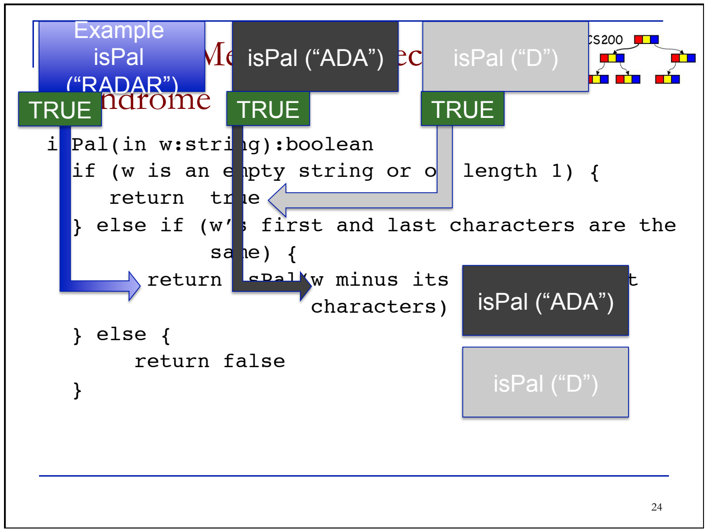
w minus its first and last characters is a palindrome

Base case?

Grammar for Palindromes

<pal> = *empty string* | *<ch>* | a *<pal>* a | … | Z *<pal>* Z

<ch> = a | b | … | z | A | B | … | Z


Example
isPal
(“RADAR”)
isPal (“ADA”)
isPal (“D”)
CS200
TRUE
TRUE
TRUE
Pal(in w:string):boolean
if (w is an empty string or of length 1) {
return true
} else if (w’s first and last characters are the
same) {
return isPal(w minus its
characters)
isPal (“ADA”)
isPal (“D”)
} else {
return false
}
24