Recap: Question 1

If passwords are strings starting with an uppercase letter and ending in a single digit and characters in between may be either letters or numbers, how many passwords of length 4 are there?

Recap: Question 2

When writing a method called add(String s, int pos) to add a data element of type String to the pos entry in a singly linked list, what cases should be handled in the code?

Recap Question 3

- Legal? `int a = 5 + (int b = 4);`
- Spot the bugs:

```
double [] scores = {50.2, 121.0, 35.03, 14.27};
double mine;
for (int in = 1; in = 4; ++in) {
    mine = mine + scores[in]; }
```

- What does this do when called with abc(scores,4):

```
public double abc(double anArray[], int x) {
    if (x == 1) { return anArray[0];}
    else { return anArray[x-1] * abc(anArray, x-1); } }
```

Grammars: Defining Languages

Walls & Mirrors Ch. 6.2

Rosen Ch. 13.1

Language, grammar

- Postfix expressions form a **language**: a set of valid strings ("sentences"), so do infix expressions
- In order to manipulate these sentences we need to know which strings are **valid sentences** (belong to the language)
- To define the valid sentences we need a mechanism to construct them: **grammars**
- A grammar defines a set of **valid symbols** and a set of **production rules** to create sentences out of symbols.

Arithmetic Postfix expressions: symbols

- Symbols: integer numbers and operators

  int : digit sequence
- There are many mechanisms to define a digit sequence, e.g. regular grammars, or regular expressions:

  dig: “0”|”1”|”2”|”3”|”4”|”5”|”6”|”7”|”8”|”9”

  num: dig$^+$
- operator: “+” | “-” | “*” |”/”

| stands for: OR (choice)

+ stands for: 1 or more of these (repetition)

what does * stand for?

don’t confuse the META symbols | $^+$ with the language symbols “+”, “-”, …

Arithmetic Postfix expressions

- An arithmetic postfix expression is

  a number, or

  two arithmetic postfix expressions followed by an operator

Notice that the operators in this example are binary

- The mechanism (context free grammar) to describe this needs more than choice and repetition, it also needs to be able to describe (block) structure

  APFE ::= num | APFE APFE operator

Notice that context free grammars are recursive in nature.

Quick check

Which are valid APFEs:

a b +

1 2 3 * +

1 2 3 + *

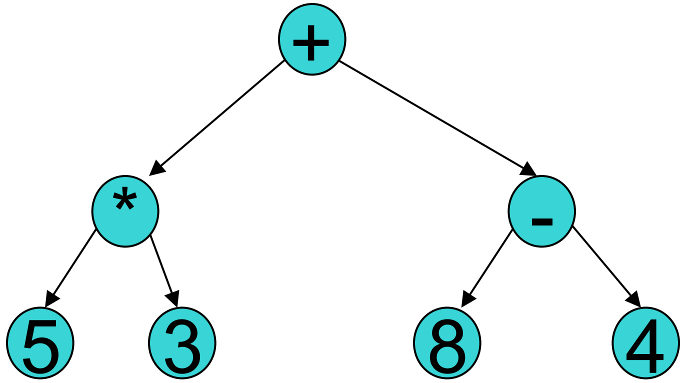
1 2 * + 3

11 22 – 33 + 44 *

If valid, what is their corresponding infix expression?

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.

  what is the alphabet for APFEs?

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

- ***Grammar*** is a set of rules that construct valid strings (sentences).
- ***Parsing Algorithm*** determines whether a string is a member of the language.

Basics of Grammars

Example: a Backus-Naur form (BNF) for identifiers

> *<identifier>* = *<letter>* | *<identifier>* *<letter>* |
> *<identifier>* *<digit>*
>
> *<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>* = …

This is called Context Free, because where-ever <word> occurs it can be replaced by one of its right hand sides.

Identifier grammar

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

<letter> = a | b | … | z | A | B | … | Z
<digit> = 0 | 1 | … | 9

Use all the alternatives of <identifier> to make 5 different shortest possible identifiers

Example

Consider the language that the following grammar defines:

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

Write strings that are in this language, which ones are right / wrong?

A. xy

B. xy, xxyy

C. xy, xyxy, xyxyxy, xyxyxyxy ….

D. xy, xxyy, xxxyyy, xxxxyyyy ….

Can you describe the language in English?

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,S}, 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 symbol S, i.e.,

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

Example as Phrase Structure

BNF: <W> = xy | x <W> y

V={x, y, W}

T={x,y}

S=W

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

Derivation:

Starting with start symbol, applying productions, by replacing a non-terminal by a rhs alternative, to obtain a legal string of terminals:

e.g., *W->xWy, W->xxyy*

Derivation

V={x, y, W}

T={x,y}

S=W

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

Derive:

- **xy**
- **xxxyyy**

Types of Phrase-Structure Grammars

- Type 0: no restrictions on productions
- Type 1 (Context Sensitive): productions such that

  $w1 \rightarrow w2$, where $w1=lAr$, $w2=lwr$, A is a nonterminal, l and r (called "the context") are strings of 0 or more terminals or nonterminals and w is a nonempty string of terminals or nonterminals. A can now only derive w in the right context $l\ \ r$.
- Type 2 (Context Free): productions such that

  $w1 \rightarrow w2$ where $w1$ is a single nonterminal including S, and $w2$ *a sequence of terminals and nonterminals*

  Equivalent to BNF

Type 3: Regular Languages

- A language generated by a type 3 (regular) grammar can have productions only of the form *A->aB* or *A->a* where *A* & *B* are non-terminals and a is a terminal.

- Notice that A->x A is **repetition** (tail recursion) and A-> aB and A -> cD and A -> x is **choice**

- Regular expressions are equivalent to regular grammars

Type 3: Regular Expressions

- Regular expressions are equivalent to regular grammars
- Regular expressions are defined recursively over a set I:
  - $\emptyset$ is the empty set { }
  - λ is the set containing the empty string { “” }
  - x whenever $x \varepsilon I$ *is the set* $\{ x \}$
  - (AB) concatenates any element of set A and any element of set B
  - (A U B) or (A | B) is the 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 | 1)*
- *Regular expression notation (…) (…)* (…)+ is often used in context free grammars as well (nice notation).*
- *Java has implementations of regular expressions.*

Identifiers

A grammar for identifiers:

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

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

<*digit*> = 0 | 1 | … | 9

Notation [a-z] stands for a | b | … | z

- 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

// or you could check is_letter(first) and
// is_letter_or digit_sequence(rest) in a loop
// going left to right through the input
```

Prefix Expressions

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

<*prefix*> = <*identifier*> | <*operator*> <*prefix*> <*prefix*>
<*operator*> = + | - | * | /
<*identifier*> = a | b | … | z

or
<identifier> = [a-z]|[A-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; }
    }
}
// notice that reading and advancing the characters is left out
// you will play with this in recitation
```

Postfix Expressions

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

  <*postfix*> = <*identifier*> | <*postfix*> <*postfix*> <*operator*>

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

  <*identifier*> = [a-z]

Recognizing a b c *+

Do it do it

<postfix>
<postfix> <postfix> <operator>
<identifier> <postfix> <operator>
a <postfix> <operator>
a <postfix> <postfix> <operator> <operator>
a <identifier> <postfix> <operator> <operator>
a b <postfix> <operator> <operator>
a b <identifier> <operator> <operator>
a b c <operator> <operator>
a b c * <operator>
a b c * +

We have already seen a different way of recognizing and evaluating postfix expr-s, using a stack.

what does red mean?
which non terminal is replaced?

Palindromes

Palindromes = {*w* : *w* reads the same left to right as right to left, when spaces and special characters are ignored, and uppercase is translated to lower case}

Examples: RADAR, racecar, [A nut for a jar of tuna], [Madam, I'm Adam], [Sir, I'm Iris]

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(s)?

Grammar for Palindromes

Why not
<ch><pal><ch>?

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

<*ch*> = [a-z] | [A-Z]

Recursive Method for Recognizing Palindrome

```
isPal(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 first and last
                             characters)
  } else {
       return false
  }
```

Example isPal ("RADAR")

Me isPal ("ADA") ec isPal ("D")

TRUE ndrome TRUE TRUE

```java
isPal(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 first and last
                         characters)
    } else {
          return false
    }
```