Informatik II (D-ITET)

hithnawi@inf.ethz.ch

Distributed Systems Group, ETH Zürich

What does XKCD mean?

It means calling the Ackermann function with Graham's number as the arguments just to horrify mathematicians.

\[ A(g_{64}, g_{64}) = \text{AUGHHA} \]
Outlook

- Exercise 3: solution discussion
- Exercise 4: overview (Stack, Ackermann, Bytecode)
Solution Ex3.Q1

- Difference between
  - String
  - StringBuffer

- Task: "NO Garbage-Collector"
  - Objects remain in memory!

- Debugger displays only the defined references
/**
 * Decrypts input text based on the CaesarChiffre (i.e., removing 3 from the
 * ASCII code of each character). The decryption employs
 * StringBuffers
 * (instead of Strings).
 *
 * @param s ciphertext to be decrypted
 */

public static String decrypt(String s) {
    StringBuffer ret = new StringBuffer();
    for (int i = 0; i != s.length(); ++i) {
        ret.append((char) (s.charAt(i) - 3));
    }
    return ret.toString();
}
Solutions Eq3.Q1

- Run as Java Application

Starting encryption (using Strings)
Done - Duration: 4174 ms.

Starting decryption (using StringBuffer)
Done - Duration: 70 ms.

Decryption successful :-)

- String is immutable (remember String literals created in memory) → modifying a chain of characters is more efficient with StringBuffer
Solution Ex3.Q2 – Syntax Analysis

Expr:

Let us do them together!
Solution Ex3.Q3 – Syntax Diagram
Solution Ex3.Q3 – Syntax Checker

Idea: int parseXY(kd, pos)
- Parse XY at Position \textit{pos} in String \textit{kd}
- Return value: Position in String after processing XY
- If the string \textit{kd} is not correct, a \texttt{ParseException} is thrown during execution
Solution Ex3.Q3 – Class LKD

```java
public class KD {
    // string parsing
    public static void parse(String kd) throws ParseException;

    // parse helpers (entity parsing)
    private static int parseEmptyOrSubTree(String kd, int position)
        throws ParseException;
    private static int parseSubTree(String kd, int position)
        throws ParseException;
    private static int parseChildren(String kd, int position)
        throws ParseException;
    private static int parseNode(String kd, int position)
        throws ParseException;

    // atomic helpers (single character parsing)
    private static boolean isCharAt(String kd, int position, char expected);
    private static int parseChar(String kd, int position, char expected)
        throws ParseException;
}
```
KD.parse(String)

Parse a "KlammerDarstellung" (KD) tree

- An empty tree is coded as ‘-’
- A node is coded with a capital character: A,B,C, ….
- Successors following a father are separated by ‘,’ in a bracket: V(C1,C2)
- Empty subtree leave at the end of the list of successors
- Omit empty list

```java
public static void parse(String kd) throws ParseException {
    int position = parseEmptyOrSubTree(kd, 0);
    if (position != kd.length()) {
        throw new ParseException("expected end of string", position);
    }
}
```
boolean isCharAt( String kd, int position, char expected )
{
    return ( position < kd.length() ) &&
            ( kd.charAt( position ) == expected );
}

int parseChar( String kd, int position, char expected )
    throws ParseException
{
    if( !isCharAt( kd, position, expected ) )
        throw new ParseException( "expected character " + expected, position );

    return position + 1;
}
KD.parseEmptyOrSubTree(String)

```
int parseEmptyOrSubTree(String kd, int position)
    throws ParseException
{
    if (isCharAt(kd, position, '-') )
        return parseChar(kd, position, '-');

    return parseSubTree(kd, position);
}
```
int parseSubTree(String kd, int position) throws ParseException {
    position = parseNode(kd, position);

    if (isCharAt(kd, position, '(')) {
        position = parseChar(kd, position, '(');
        position = parseChildren(kd, position);
        position = parseChar(kd, position, ')');
    }

    return position;
}
int parseChildren(String kd, int position) throws ParseException
{
    for (position = parseEmptyOrSubTree( kd, position );
        isCharAt( lkd, position, ',' );
        position = parseEmptyOrSubTree( kd, position ) )
    {
        position = parseChar( kd, position, ',' );
    }

    return position;
}
int parseNode( String kd, int position ) throws ParseException {
    if( position >= kd.length() )
        throw new ParseException( "expected a node", position );

    char ch = kd.charAt( position );
    if( !Character.isUpperCase( ch ) )
        throw new ParseException( "invalid character " + ch, position );

    return position + 1;
}
Outlook

- Exercise 3 solution discussion
- Exercise 4 overview (Stack, Ackermann, Bytecode)
Excercise 4

1. Stack
   - Possible implementation using arrays
   - Interface is known but what happens in the background (depends on the programmer). We make a better implementation with lists later!

2. Ackermann – exploding recursion... :-)
   - How much is (4, 2)?

3. Java Bytecode
   - Finally we will have a look at low-level stuff 😊
Hints Ex4.Q1 - Stack

- Data structure
- Only the last element is accessed
  - last-in-first-out queue (LIFO queue)
Exercise 4 - Q1 (a-c)

- **Constructor**
  - Initializes internal Array
  - Capacity is an argument to the constructor

- **toString() with StringBuffer**
  - Expected Output: "[e0, e1, e2, …]"
  - Concatenation
    - **String**: `str += "bar";`
    - **StringBuffer**: `buf.append("bar");`

- **grow()**
  - Capacity doubled, copy old values
Exercise 4 - Q1 (d)

- `push()`, `pop()`, `peek()`, `empty()`
  - Standard stack functions
  - Arguments are of type `int`
  - If necessary, call `grow()`

- `size()`
  - Number of elements currently on the stack

- `capacity()`
  - Total number of elements which fit on the current stack until the next grow
Exercise 4 – Ackermann Function Q2

- Recursive Definition

\[
A(0, m) = m + 1 \\
A(n + 1, 0) = A(n, 1) \\
A(n + 1, m + 1) = A(n, A(n + 1, m))
\]

- Grows extremely fast
  - \( A(3, 3) = 61 \)
  - \( A(4, 2) \) has already 19729 decimal places!!

Why is this function useful? \( \rightarrow \) Theoretical computer science: example of a function that is computable but not primitive recursive – is a famous example for an incredibly fast rising function, and we will use it to investigate how Java internally handles recursion.
Exercise 4 – Q2

- You should calculate $A(n,m)$...
  - First, calculate $A(2,1)$ by hand on paper
    - Write down all steps then (b) gets easier ...
      $A(2,1) = A(1+1,0+1) = A(1,A(2,0))$...
  - Then, write the Pseudocode
    - «Descriptive», but in the form of programming language
    - Think about Stacks... :-)
    - The function has the property that one can not say in advance how deep the recursion is $\rightarrow$ use while instead of for-loop!
  - Implement using the stack implementation in Question 1 with an iterative algorithm

http://www.wolframalpha.com/input/?i=Ackermann+function+%282%2C2%29
Exercise 4 - Iterative Approach Q2c

- Ackermann’s formula always requires (exactly) two values:
  - The currently required values should be at the top of the stack…
  - What does it mean when there is one item left in the stack?

```java
Stack stack = new Stack();
stack.push(4);
stack.push(7);

while (stack.size() != 1)
{
    ...    
}
```

Exercise 4 – Implementation Q2c

```java
stack.push(m)
stack.push(n)

if n == 0 → result = m+1
else if m == 0 → push(n-1), push(1)
else push(n-1), push(n), push(m-1)
```

```java
m = stack.pop();
n = stack.pop();
```
Exercise 4 – Implementation Q2c

- Stack
  - Use the stack implementation in Q1
  - The interface should NOT be modified

- “Snapshots”
  - With toString() method of the stack

- What if I can not do Q1?
  - Use java.util.Stack<Integer>
    - you just need push(), pop(), size und toString()
  - If necessary: send me an Email
Exercise 4 – Java Bytecode Q3

```java
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

The Java program is compiled to bytecode, which is then executed by the JVM on different operating systems (Win32, UNIX, MacOS).
Java-Bytecode

- Bytecode ist die Maschinensprache der Java-VM
- Bytecode ist ziemlich kompakt: Die meisten Instruktionen („Operationen“) sind nur 1 Byte (= 8 Bit) lang
  - Kennzeichnung durch einen 8-Bit-Operationscode
  - Haben zusätzlich auch eine „symbolische“ Bezeichnung, z.B.:
    - `add` mit Operationscode 01100000
      (dezimal 96, hexadezimal 0x60)
    - `iconst_3` mit Operationscode 00000100
    - `pop` mit Operationscode 01010111
**Exercise 4 – Java Bytecode Q3**

Method int f(int, int, int)
- 0 iload_0
- 1 iload_1
- 2 iadd
- 3 iload_2
- 4 idiv
- 5 ireturn

Method int g(int, int)
- 0 iload_0
- 1 iload_1
- 2 iconst_3
- 3 invokestatic #f
- 6 ireturn
Exercise 4 – Java Bytecode Q3

Method int f(int, int, int)

0 iload_0  \( a_0 \)

1 iload_1  \( a_1 \)

2 iadd  \( a_0 + a_1 \)

3 iload_2  \( a_2 \)

4 idiv \( (a_0 + a_1) / a_2 \)

5 ireturn

http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html
Demo: Bytecode and Strings/Stringbuffer

D:\Projects\DisassemblerDemo>
javac JavaTip.java  //compiler
java JavaTip        //run
javap -c -private JavaTip  //disassembler

- **Common mistake:** "javap is not recognized as an internal or external command, operable program or batch file"
- **Reason:** java binaries are not defined in System variable PATH
- **Solution:** Right Click on Computer → Properties → Advanced System Settings → Environment Variables → PATH → add (where you installed the Java JDK) and restart Windows
  
  C:\Program Files\Java\jdk1.7.0_31\bin
Have Fun!