Tweede Hertentamen OOP

vrijdag 31 augustus 2007, 09:00-12:00

LEES DIT EERST !

- Dit tentamen behoort bij het 5-EC vak Object-georiënteerd Programmeren.
- Vul de kop van het eerste in te leveren blad **volledig in**.
- Nummer de bladen en zet bovenaan het eerste blad het totaal aantal ingeleverde bladen; voorzie elk blad van je naam.
- Werk zorgvuldig en schrijf netjes met blauwe of zwarte pen; **geen** potlood!
- Het werk inleveren: leg je bladen op volgorde, vouw ze dubbel en doe ze in de envelop. Plak de envelop **niet** dicht. Voorzie de envelop van je naam en je studentnummer.
- Het gecorrigeerde werk is na ongeveer drie weken af te halen bij het onderwijsbureau voor Informatica. Je wordt verzocht dit ook daadwerkelijk te doen!
- Indien niet aan de gestelde practicumeis is voldaan zal geen tentamenbriefje worden uitgereikt.
- Dit is het laatste hertentamen.

VEEL SUCCES !

begin tentamen >>>

Opgave 1 [25%] (OO-theorie)

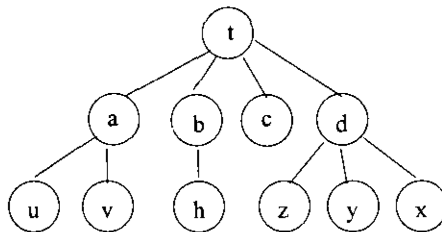
Belangrijke begrippen in object-georiërenteerd programmeren zijn: *klasse*, *object*, *abstracte klasse*, *interface*, *dynamic binding* en *inheritance* (= overerving).

(a) Beschrijf kort en duidelijk het verschil tussen de begrippen *klasse* en *object*.
(b) Beschrijf kort en duidelijk het begrip *inheritance*.
(c) Leg uit welke relatie aangeeft dat er inheritance toegepast kan worden.
(d) Leg uit wat *dynamic binding* is.
(e) Beschrijf wat de verschillen en overeenkomsten zijn tussen een *abstracte klasse* en een *interface*.

Geef waar nodig voorbeelden om je antwoorden te verduidelijken.

Opgave 2 [25%] (Datastructuren)

In deze opgave wordt gevraagd een *generieke* boom datastructuur te maken, waarbij het maximaal aantal kinderen van elke knoop van te voren *niet* vastgelegd is. Beschouw bijvoorbeeld de boom in figuur 1 hieronder. Deze boom bevat elementen a,b,c,d,h,t,u, v,x,y en z. Er is 1 knoop met 4 kinderen, 1 knoop met 3 kinderen, 1 knoop met 2 kinderen, 1 knoop met 1 kind, en 7 bladeren d.w.z. knopen zonder kinderen.



Figuur 1: Voorbeeld van een generieke boom

We representeren een knoop in de generieke boom met de volgende klasse:

```
import java.util.LinkedList;

class TreeNode<T> {
    T item;                                  // het opgeslagen item
    LinkedList<TreeNode<T>> children; // de kinderen van deze knoop
    // het meest linkerkind staat vooraan in deze lijst

    public TreeNode() {                      // default constructor
      item = null;
      children = new LinkedList<TreeNode<T>>();
    }
}
```

Een knoop bevat dus een element van type T en een lijst van knopen, de kinderen van deze knoop.

(a) Voorzie de klasse TreeNode van een constructor die een element van het type T meekrijgt als parameter, het element dat in de knoop moet worden geplaatst.

>>> lees verder >>>

De generieke boom zelf representeren we met de klasse Tree. We geven alvast een aanzet:

```
import java.util.LinkedList;

class Tree<T> {
    TreeNode<T> root; // wortel van de boom

    public Tree() {   // default constructor
        root = null;
    }
}
```

In de volgende onderdelen van deze opgave wordt gevraagd deze gegeven code aan te vullen. Het staat je hierbij vrij om, waar je dat nodig acht, hulpmethoden te introduceren.

(b) Voeg aan de klasse Tree een methode

```
public int size()
```

toe, die het totaal aantal in de boom opgeslagen elementen oplevert.

(c) Een andere maat voor de omvang van de boom is het maximale aantal kinderen dat een knoop in de boom heeft. Voor een binaire boom is dit twee. Gevraagd wordt om aan de klasse Tree een methode

```
public int maxChildren()
```

toe te voegen, die het grootste aantal kinderen dat een knoop in de boom heeft oplevert.

Merk op dat de boom uit figuur 1 een knoop met 4 kinderen heeft, en dus is het maximale aantal kinderen dat een knoop in deze boom heeft, vier is.

(d) Voeg aan de klasse Tree een methode

```
public boolean contains(T x)
```

toe, die teruggeeft of een element x bevat is in de boom.

(e) Gevraagd wordt voor de klasse Tree een methode

```
public String toString()
```

te schrijven die de string-representatie van (alle elementen in) de boom oplevert. De boom dient hierbij niveau voor niveau, van links naar rechts per niveau, gerepresenteerd te worden. De elementen van een niveau lager komen nadat alle elementen van het niveau erboven gerepresenteerd zijn. Bijvoorbeeld: de string-representatie van de boom uit figuur 1 is `"t a b c d u v h z y x"`

>>> **volgende opgave** >>>

Opgave 3 [30%] (Recursive-descent parsing)

In deze opgave wordt een herkennende parser gevraagd voor de volgende grammatica voor eenvoudige programma's:

```
<program>    ::= <statements> .
<statements> ::= { <statement> ';' } .
<statement>  ::= <increment> | <decrement> | <assignment> | <repetition> .
<increment>  ::= 'INC' <var> [ <value> ] .
<decrement>  ::= 'DEC' <var> [ <value> ] .
<assignment> ::= 'LET' <var> '=' <value> .
<repetition> ::= 'WHILE' <var> 'DO' <statements> 'END' .
<value>      ::= <var> | <num> .
<var>        ::= <char> { <char> } .
<num>        ::= <digit> { <digit> } .
<char>       ::= 'a' | 'b' | ... | 'z' .
<digit>      ::= '0' | '1' | ... | '9' .
```

De karakters in een variabelenaam (hulpsymbool `<var>`) en de cijfers in een getal (hulpsymbool `<num>`) staan aaneengesloten; tussen alle andere elementen mag een willekeurige hoeveelheid witruimte staan. Onder witruimte verstaan we spaties, tab- en regelovergangs-symbolen. Een programma dient gevolgd te worden door een eindebestand (ook wel end-of-file) symbool.

(a) Beschrijf wat een tokenizer doet.

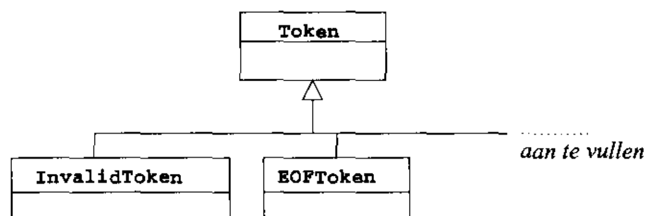
(b) Gesteld dat je een Tokenizer voor bovenstaande grammatica zou moeten gaan maken, geef aan welke tokens deze kunnen opleveren. Je mag uit gaan van een abstracte klasse Token met de volgende publieke constructoren en methoden:

```java
/** creeert een Token met stringrepresentatie s */
public Token(String s)

/** creeert een Token met characterrepresentatie ch */
public Token(char ch)

/** retourneert de stringrepresentatie van het Token */
public String toString()
```

Om je op weg te helpen hebben we hieronder een deel van de Token-hiërarchie gegeven. Vul deze verder in. (Je mag ook volstaan met een opsomming van alle tokens).



>>> lees verder >>>

Ga er vanuit dat Tokenizer een klasse is die de opgesomde tokens oplevert. Hieronder geven we nog kort even de beschikbare constructoren en methoden van de klasse Tokenizer zoals deze op college en het practicum aan de orde zijn geweest:

```
public Tokenizer(InputStream is) // constructor

public Tokenizer(Reader r)       // constructor

/** Levert het huidige token als resultaat */
public Token getCurrent()

/** Schuift de tokenizer door naar het volgende token */
public void moveNext()
```

Gevraagd wordt een recursive-descent parser te schrijven voor de gegeven grammatica. We geven alvast een klasse Parser

```
abstract class Parser {
  protected static void reportError(String message) {
    System.out.println("FOUT: " + message);
    System.exit(0);
  }
}
```

(c) Beschrijf voor welke hulpsymbolen er subklassen van `Parser` gemaakt moeten worden.

(d) Geef voor elk van deze klassen de bijbehorende implementatie van de volgende methode

```
public static boolean tryParse(Tokenizer tok)
```

Merk op dat er niet gevraagd wordt een parse-tree op te bouwen, of excepties op te gooien. Er wordt slechts een *herkennende* parser gevraagd. Bij het constateren van een fout is het voldoende om de reportError methode aan te roepen.

>>> **volgende opgave** >>>

Opgave 4 [20%] (I/O en excepties)

De volgende expressie voldoet aan de grammatica van de vorige opgave:

```
LET x = 1; LET y = 4; WHILE y DO LET z = y;
WHILE z DO INC x z; DEC z; END; DEC y; END;
```

Een (goed geschreven) parser heeft er helemaal geen moeite mee om de structuur van deze expressie te ontrafelen. Een menselijke lezer heeft echter meer profijt van een betere layout. Het bovenstaande programma is veel beter te begrijpen als we wat meer indentatie en regelovergangen introduceren. Bijvoorbeeld:

```
LET x = 1;
LET y = 4;
WHILE y DO
  LET z = y;
  WHILE z DO
    INC x z;
    DEC z;
  END;
  DEC y;
END;
```

Door een nieuw blok van statements twee spaties in te laten springen (indentatie) t.o.v. de omgeving en elk simpel statement (increment, decrement en assignment) op een nieuwe regel te plaatsen, onstaat een opmaak die de structuur van het programma beter weergeeft.

Schrijf een programma dat met één parameter wordt aangeroepen. Deze parameter is de naam van een bestand waarvan je programma ten eerste moet nagaan of de inhoud van dit bestand voldoet aan de grammatica van **opgave 3**. Is dat niet zo, dan dient een foutmelding afgedrukt te worden. Voldoet de inhoud wel aan de grammatica, dan moet een geformatteerde versie zoals hier boven beschreven van deze inhoud naar het scherm afgedrukt worden.

Zorg voor een nette afhandeling van situaties die niet aan de gestelde eisen voldoen. Maak daar waar nodig gebruik van excepties.

Aanwijzingen:

- Als de invoer aan de grammatica voldoet, is het redelijk eenvoudig om de uitvoer in het gevraagde formaat op te leveren. Het is *echt niet nodig* om de expressie eerst in een boomstructuur op te slaan. Kijk goed naar wanneer er een regelovergang moet komen, en wanneer het inspringniveau groter danwel kleiner moet worden.
- Voor het inlezen kent Java een FileReader en BufferedReader. In **opgave 3** staan de constructoren van de Tokenizer.
- Pas allereerst de parser uit **opgave 3** aan om de geformatteerde uitvoer op te leveren. De tryParse methoden zullen wellicht geen Boole'se resultaat moeten opleveren. De reportError methode uit Parser moet wellicht herschreven worden om een exceptie op te gooien. Schrijf na deze aanpassingen vervolgens het gevraagde hoofdprogramma.

einde tentamen <<<