Egzamin poprawkowy z Programowania obiektowego
04 IX 2OO9

Tematem zadania jest interpreter obiektowego języka programowania **Ropucha**. Oczekujemy od Państwa projektu reprezentacji poprawnych (składniowo i semantycznie) programów oraz implementacji tej części interpretera, która będzie wykonywała poprawny program.

Składnia języka

Składnię **Ropuchy** opiszemy w rozszerzonej notacji **BNF**. Symbole nieterminalne zostały ujęte w nawiasy kątowe a symbole terminalne w cudzysłowy. Symbolami terminalnymi są też: **`IDENTYFIKATOR`**, czyli ciąg liter i cyfr zaczynający się od litery, oraz **`NAPIS`**, będący ujętym w cudzysłowy ciągiem znaków, w którym mogą wystąpić znane z **C** i **Javy** sekwencje **`\n`**, **`\t`**, **`\"`** i **`\\`**.

Lewą stronę produkcji od prawej oddziela **`::=`**. Fragmenty opcjonalne ujęte są w nawiasy kwadratowe, a w klamrowe te, które mogą się powtórzyć zero lub więcej razy. Kreska pionowa oddziela warianty alternatywne.

```
<program>                ::= { <deklaracja-klasy> }
<deklaracja-klasy>       ::= <nazwa-klasy-pochodnej> [ ":" <nazwa-klasy> ]
                               "{" { <deklaracja-atrybutu>
                                   | <deklaracja-metody> } "}"
<nazwa-klasy-pochodnej>  ::= IDENTYFIKATOR
<nazwa-klasy>            ::= "_" | <nazwa-klasy-pochodnej>
<deklaracja-atrybutu>    ::= <typ> <nazwa-atrybutu> ";"
<typ>                    ::= <nazwa-klasy>
<nazwa-atrybutu>         ::= IDENTYFIKATOR
<deklaracja-metody>      ::= <typ> <nazwa-metody> "(" <typ> ")"
                               "{" <treść-metody> "}"
<nazwa-metody>           ::= IDENTYFIKATOR
<treść-metody>           ::= <wyrażenie> { ";" <wyrażenie> }
<wyrażenie>              ::= <przypisanie> | <wyrażenie-proste>
<przypisanie>            ::= <wyrażenie-proste> "=" <wyrażenie>
<wyrażenie-proste>       ::=   <aktualny-obiekt>  | <nowy-obiekt>
                             | <napis-na-wyjście> | <wartość-parametru>
                             | <wartość-atrybutu> | <komunikat>
                             | <asercja> | "(" <wyrażenie> ")"
<aktualny-obiekt>        ::= "$"
<nowy-obiekt>            ::= "@" <nazwa-klasy>
<napis-na-wyjście>       ::= NAPIS
<wartość-parametru>      ::= "&"
<wartość-atrybutu>       ::= <wybór-składowej>
<wybór-składowej>        ::= <wyrażenie-proste> "." <nazwa-składowej>
<nazwa-składowej>        ::= <nazwa-atrybutu> | <nazwa-metody>
<komunikat>              ::= <wybór-składowej> "(" <wyrażenie> ")"
<asercja>                ::= "!(" <wyrażenie> ")"
```

Opis języka

Program w języku **Ropucha** to ciąg deklaracji klas. Tworzą one hierarchię, której korzeniem jest wbudowana klasa o nazwie „_” (czytaj „coś”), będąca odpowiednikiem klasy **Object** w **Javie**. Deklarując klasę, podajemy jej nazwę (identyfikator) i wskazujemy nadklasę. Pominięcie deklaracji nadklasy oznacza, że jest nią klasa „_”.

W programie nie może być dwóch klas o tej samej nazwie. W hierarchii klas nie może być cyklu.

Ropucha, tak jak **Java**, jest językiem z silnym, statycznym systemem typów. Wyrażenia, oprócz wartości, mają też określony typ. Analizując program przed jego uruchomieniem, na podstawie typu wyrażenia decydujemy, czy jego użycie jest poprawne. Zajmuje się tym analizator semantyczny. System typów języka **Ropucha** gwarantuje, że podczas wykonania poprawnego pod względem typów programu (a więc w projektowanej przez Ciebie części interpretera), nie wystąpi błąd. Jedyny wyjątek, opisany poniżej, stanowią asercje.

Typy w **Ropusze** są związane z klasami i odpowiadają typom referencyjnym języka **Java**. Wartością wyrażenia, którego typ jest związany z klasą **K**, jest obiekt klasy **K**, lub klasy dziedziczącej (bezpośrednio lub pośrednio) z klasy **K**. Ponadto wartością wyrażenia dowolnego typu może być specjalna wartość „nic”, będąca odpowiednikiem wartości „**null**” w **Javie** (wartość ta nie ma swojego indentyfikatora, nie może więc być bezpośrednio użyta w programie).

W opisie języka **Ropucha** posługujemy się relacją bycia podtypem. Powiemy, że typ **T1** jest podtypem **T2**, gdy oba typy są związane z klasami i klasa typu **T1** jest w części hierarchii klas, której korzeniem jest klasa typu **T2**. W szczególności, każdy typ jest swoim podtypem.

Deklaracja klasy zawiera ciąg deklaracji składowych: atrybutów i metod. Oprócz nich klasa ma też składowe, które dziedziczy z nadklasy. W klasie „_” żadnych składowych nie ma.

Deklaracja atrybutu określa jego typ i nazwę. Wartością początkową wszystkich atrybutów jest „nic”.

Deklaracja metody określa jej nazwę, typ jej wyniku, typ jej parametru i treść. Każda metoda ma dokładnie jeden parametr nie mający nazwy.

W klasie nie może być dwóch składowych o tej samej nazwie nawet jeśli jedna z nich jest atrybutem a druga metodą. Wolno jednak w podklasie przedefiniować metodę z nadklasy. W takim wypadku typ wyniku metody z podklasy powinien być podtypem typu wyniku metody z nadklasy (kowariancja typu wyniku), a typ parametru metody z nadklasy podtypem typu parametru metody z podklasy (kontrawariancja typu parametru).

Treścią metody jest ciąg wyrażeń, które są obliczane podczas jej wykonania. Wynikiem metody jest wartość ostatniego z nich. Jego typ musi być podtypem typu wyniku metody.

W wyrażeniach języka **Ropucha**, oprócz nawiasów służących do grupowania, mogą wystąpić:

- przypisanie („**`lewa=prawa`**”):
  Łączy dwa wyrażenia, nazywane lewą stroną i prawą stroną. Ma typ taki, jak lewa strona. Typ prawej strony musi być podtypem typu lewej strony. Lewa strona musi być odwołaniem do atrybutu obiektu lub do parametru metody. Wynik jej obliczenia określa miejsce, w którym ma się znaleźć wartość prawej strony. Obliczenie wartości przypisania powoduje obliczenie jego prawej strony, następnie lewej i przypisanie wartości prawej strony na lewą (chyba że lewa strona jest postaci „wyrażenie.nazwa”, a wartością wyrażenia jest „nic”, wówczas wartość prawej strony nie jest nigdzie przypisywana). Wartością przypisania jest obliczona wartość jego prawej strony.
- aktualny obiekt („**`$`**”):
  Odwołanie do obiektu, dla którego wykonuje się metoda (odpowiednik „**this**” w **Javie**). Typem tego wyrażenia jest klasa, w której została zadeklarowana metoda, w treści której występuje to wyrażenie.
- nowy obiekt („**`@Nazwa`**”):
  Utworzenie nowego obiektu wskazanej klasy (odpowiednik „**new**” w **Javie**). Typem wyrażenia jest klasa, której obiekt tworzymy. Wyrażenie jest poprawne, jeśli istnieje klasa o podanej nazwie.
- napis na wyjście („**`"..."`**”):
  Ma typ „_” i wartość „nic”, a skutkiem ubocznym jego obliczenia jest wypisanie treści napisu na wyjście.
- wartość parametru („**`&`**”):
  Odczytanie wartości parametru metody. Typ jest określony w deklaracji metody.
- wartość atrybutu („**`obiekt.nazwa`**”):
  Odczytanie wartości atrybutu obiektu. Obiekt jest określony przez wyrażenie, którego typem musi być klasa posiadająca atrybut o podanej nazwie (może być dziedziczony z nadklasy). Typ całego wyrażenia odczytujemy z deklaracji tego atrybutu. Jeśli wartością wyrażenia przed kropką jest „nic”, wartością całego wyrażenia jest „nic”.
- komunikat („**`obiekt.nazwa(argument)`**”):
  Wysłanie komunikatu do obiektu, będącego wartością wyrażenia przed kropką, z argumentem, którym jest wartość wyrażenia podanego w nawiasach. Wyrażenie to jest poprawne, jeśli w klasie, która jest typem wyrażenia przed kropką, jest metoda o wskazanej nazwie (może być dziedziczona z nadklasy), a typ wyrażenia określającego argument jest podtypem zadeklarowanego typu parametru. Typ całego wyrażenia jest taki, jak wyniku tej metody. Obliczenie wyrażenia powoduje obliczenie wartości wyrażenia wskazującego odbiorcę oraz wartości argumentu. Następnie, spośród metod odbiorcy, wybieramy metodę o nazwie takiej, jak nazwa komunikatu i wykonujemy ją z zadanym argumentem. Wartością całego wyrażenia jest wynik metody. Gdy odbiorcą jest „nic”, wartością całego wyrażenia staje się „nic”.
- asercja („**`!(wewnętrzne)`**”):
  Typem tego wyrażenia jest typ wyrażenia wewnętrznego. Obliczenie wyrażenia powoduje obliczenie wyrażenia wewnętrznego. Jeśli jego wartość jest inna niż „nic”, staje się ona wartością całego wyrażenia. W przeciwnym przypadku wypisywany jest komunikat o błędzie, a wykonywanie programu natychmiast się kończy.

Dokładnie jedna z klas poprawnego programu powinna mieć metodę o nazwie „**main**”. Ma to być metoda o typie parametru i typie wyniku „_”. Wykonanie programu oznacza utworzenie obiektu tej klasy i wysłanie do niego komunikatu „**main**” z „nic” jako argumentem.

Przykład programu poprawnego

Poniżej mamy przykład poprawnego programu w **Ropusze**. Wypisuje on, w kolejnych wierszach, wyrazy ciągu Collatza, zaczynając od liczby **13** a kończąc na **1**. Każda liczba jest zapisana w postaci ciągu gwiazdek odpowiedniej długości.

```
Liczba {
  Liczba pomoc;
  _ kolejna(_)              { "" }
  Liczba plus1(_)           { @JedenPlus.inicjalizacja($) }
  Liczba razy2(_)           { $.plus($) }
  _ pisz(_)                 { "*\n" }
  Liczba plus(Liczba)       { &.plus1($) }
  _ połowa0(Liczba)         { &.kolejna($) }
  _ połowa1(Liczba)         { ($.pomoc=&.razy2($).plus1($))
                              .razy2($).plus($.pomoc).plus1($).kolejna($) }
}

JedenPlus : Liczba {
  Liczba ogon;
  Liczba inicjalizacja(Liczba)   { $.ogon=&; $ }
  _ kolejna(_)                   { $.pisz($); $.ogon.połowa0(@Jeden) }
  _ pisz(_)                      { "*"; $.ogon.pisz($) }
  Liczba plus(Liczba)            { $.ogon.plus(&.plus1($)) }
  _ połowa0(Liczba)              { $.ogon.połowa1(&) }
  _ połowa1(Liczba)              { $.ogon.połowa0(&.plus1($)) }
}

Jeden : Liczba {
}

Główna {
  _ main(_) { @Jeden.razy2($).plus1($).razy2($).razy2($).plus1($).kolejna($);
              "*\n" }
}
```

Polecenie

Pracujemy nad interpreterem wykonującym programy w języku **Ropucha**. Interpreter będzie się składał z czterech części:

- analizatora leksykalnego, który w tekście wejściowym rozpozna reprezentacje terminali gramatyki języka.
- analizatora składniowego, który sprawdzi, czy ciąg terminali, przekazany mu przez analizator leksykalny, jest słowem języka bezkontekstowego zdefiniowanego gramatyką języka **Ropucha**. Analizator składniowy zbuduje też reprezentację programu. Zadania analizatora leksykalnego i składniowego wykona metoda „**`wczytaj()`**”, która zgłosi wyjątek **`BłądSkładniowy`**, jeśli stwierdzi, że program wczytany z wejścia nie jest poprawny składniowo.
- analizatora semantycznego, który sprawdzi, czy program, o którym już wiemy, że jest poprawny składniowo, spełnia też pozostałe warunki poprawności programów w **Ropusze**, w szczególności te, które dotyczą typów. Analizę semantyczną przeprowadzi metoda „**`sprawdź()`**”, która zgłosi wyjątek **`BłądSemantyczny`**, gdy wykryje błąd.
- modułu wykonującego program, uruchamianego metodą „**`wykonaj()`**”. Analizator semantyczny musi zagwarantować, że podczas wykonania programu, który pomyślnie przeszedł badanie poprawności, nie wystąpi błąd. Nie może się więc zdarzyć np. próba sięgnięcia do składowej obiektu, której on nie posiada, próba przypisania na coś, na co przypisać się nie da itp.

Oto główna klasa pakietu:

```
package ropucha;

import java.io.Reader;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class Interpreter {
    public static void main(String []args) {
        try {
            Reader dane = null;
            try {
                if (args.length > 0) {
                    dane = new BufferedReader(new FileReader(args[0]));
                }
                else {
                    dane = new BufferedReader(new InputStreamReader(System.in));
                }
                new AnalizatorSkładniowy(dane).wczytaj().sprawdź().wykonaj();
            }
            finally {
                if (dane != null) {
                    dane.close();
                }
            }
        }
        catch(IOException wyjątek) {
            System.err.println("Błąd podczas wczytywania programu");
        }
        catch(BłądSkładniowy wyjątek) {
            System.err.println("Błąd składniowy");
        }
        catch(BłądSemantyczny wyjątek) {
            System.err.println("Błąd semantyczny");
        }
        catch(Exception wyjątek) {
            System.err.println("Błąd w interpreterze");
        }
    }
}
```

Zaprojektuj moduł wykonujący program oraz reprezentację programu, którą ma zbudować analizator semantyczny. Możesz założyć, że wcześniejsza reprezentacja programu (np. zwracana przez analizator składniowy) jest inna, a analizator semantyczny w pewnym momencie przekształca program do zaprojektowanej przez Ciebie reprezentacji. W projekcie powinny być zaznaczone wszystkie metody i atrybuty projektowanych klas: umożliwiające zbudowanie reprezentacji programu oraz jego wykonanie, w szczególności wszystkie potrzebne konstruktory, getery i setery.

Zaimplementuj w **Javie** moduł wykonujący program - metodę „**`wykonaj()`**” i niestandardowe metody, z których ona korzysta. Pozostałych części interpretera nie trzeba implementować.