Egzamin z Programowania obiektowego, 22 VI 2OO9

Tematem zadania jest interpreter obiektowego języka programowania **Żaba**. Oczekujemy od Państwa projektu reprezentacji poprawnych składniowo programów oraz implementacji tej części interpretera, która będzie sprawdzała, czy poprawny składniowo program spełnia pozostałe warunki poprawności.

Składnia języka

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

Opis języka

Program w języku **Żaba** 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.

Żaba, 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. System typów języka **Żaba** gwarantuje, że podczas wykonania poprawnego pod względem typów programu, nie wystąpi błąd.

Typy w **Żabie** 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**.

W **Żabie** jest też typ, który będziemy nazywali „nic”. Jest on odpowiednikiem typu „**null**” w **Javie** i ma jedną wartość, którą również nazywamy „nic”. Typ ten nie ma swojego identyfikatora, nie możemy więc użyć go w deklaracji.

W opisie języka **Żaba** 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. Dodatkowo, „nic” jest podtypem każdego typu.

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 typ jej wyniku, nazwę, parametry i treść. Każdy parametr ma typ i nazwę. Nazwy parametrów metody muszą być różne.

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 liczba parametrów obu metod musi być równa, typ wyniku metody z podklasy powinien być podtypem typu wyniku metody z nadklasy (kowariancja typu wyniku), a typy parametrów metody z nadklasy podtypami odpowiadających im parametrów metody z podklasy (kontrawariancja typów parametrów).

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. Jeśli treść metody jest pusta, jej wynikiem jest „nic”.

W wyrażeniach języka **Żaba**, 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 lewej strony, następnie prawej i przypisanie wartości prawej strony na lewą. 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 „nic” i wartość „nic”, a skutkiem ubocznym jego obliczenia jest wypisanie treści napisu na wyjście.
- wartość parametru („`nazwa`”):
  Odczytanie wartości parametru metody. Typ jest określony przez jego deklarację. Wyrażenie jest poprawne, jeśli metoda ma parametr o tej nazwie.
- 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(argument1,...,argumentN)`”):
  Wysłanie komunikatu do obiektu, będącego wartością wyrażenia przed kropką, z argumentami, którymi są wartości wyrażeń podanych 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), ma tyle parametrów, ile argumentów przesyłamy, a typy wyrażeń określających argumenty są podtypami typów odpowiadających im parametrów. Typ całego wyrażenia jest taki, jak wyniku tej metody. Obliczenie wyrażenia powoduje obliczenie wartości wyrażenia wskazującego odbiorcę i argumenty, od lewej do prawej. Następnie, spośród metod odbiorcy, wybieramy metodę o nazwie takiej, jak nazwa komunikatu i wykonujemy ją z zadanymi argumentami. Wartością całego wyrażenia jest wynik metody. Gdy odbiorcą jest „nic”, wartością całego wyrażenia staje się „nic”.

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

Przykład programu poprawnego

Poniżej mamy przykład poprawnego programu w **Żabie**. 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 x)     { x.plus1(); }
  _ połowa0(Liczba liczba) { liczba.kolejna(); }
  _ połowa1(Liczba liczba) { ($.pomoc=liczba.razy2().plus1())
                               .razy2().plus($.pomoc).plus1().kolejna(); }
}

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

Jeden : Liczba {
}

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

Przykłady programów niepoprawnych

Poniżej są przykłady programów poprawnych składniowo, które jednak nie spełniają dodatkowych warunków poprawności wymienionych w opisie języka:

brak deklaracji metody main:	`A{ }`
niewłaściwa deklaracja metody main:	`A{ A main(A x){} }`
podwójna deklaracja metody main:	`A{ _ main(){"A";} } B{ _ main(){"B";} }`
brak deklaracji klasy:	`A:B{ C x; _ main(){} D f(E y){@F;} }`
cykl w hierarchii klas:	`A:C{ } B:A{ _main(){} } C:B{ }`
konflikt nazw klas:	`A{ } A{ _ main(){} }`
konflikt nazw składowych:	`A{ A x; } B:A{ _ x; }`
konflikt nazw składowych:	`A{ _ main(){} A x; _ x(){} }`
konflikt nazw parametrów:	`A{ _ main(){} _ f(A x,A x){} }`
brak deklaracji:	`A{ A x;_ main(){x;} }`
brak deklaracji:	`A{ _ main(){$.x="";} }`
nieprawidłowa lewa strona przypisania:	`A{ _ f(){} _ main(){$.f="";@A=@A;} }`
niewłaściwy typ wyniku metody:	`A{ } B{ _ main(){} A f(){@B;} }`
niewłaściwy typ wyniku metody:	`B:A{ _ main(){} } A{ B f(){$;} }`
niewłaściwe typy w przypisaniu:	`A{ _ main(){} } B{ A x; _ f(B y){$.x=y;} }`
źle przedefiniowana metoda:	`A{ _ main(){} _ f(){} } B:A{ _ f(A x){} }`
źle przedefiniowana metoda:	`A{ _ main(){} B f(){} } B:A{ A f(){} }`
źle przedefiniowana metoda:	`A{ _ main(){} _ f(A x){} } B:A{ _ f(B x){} }`
brak deklaracji atrybutu:	`A{ _ main(){} } B:A{ A x;_ f(A y){y=@B;y.x="";} }`
brak deklaracji metody:	`A{ _ main(){$.f(@B);} _ f(A x){} } B{ }`

Polecenie

Pracujemy nad interpreterem wykonującym programy w języku **Żaba**. Oto główna klasa pakietu:

```
package żaba;

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");
        }
    }
}
```

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 **Żaba**. 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 **Żabie**, 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.

Zaprojektuj reprezentację programu, którą ma zbudować analizator składniowy.

Zrealizuj w **Javie** analizator semantyczny - metodę „`sprawdź()`” i niestandardowe metody, z których ona korzysta. Pozostałych części interpretera nie trzeba implementować.