Język obiektowy

Podstawowe elementy

1. **Int** - liczba całkowita
   Object - na wzór klasy z języków programowania jak Python czy C++
   None - wykorzystywany np. w momencie, gdy chcemy zadeklarować atrybut, a nie wiemy co będzie się w nim zawierało

2. **Stałe znakowe**: 'txt'

3. **Operatory arytmetyczne**: $+ \quad - \quad * \quad /$

4. **Operatory logiczne**: $< \quad > \quad <= \quad >= \quad == \quad != \quad || \quad \&\& \quad !$

5. **Operator przypisania:** $=$

6. **Komentarz**: #

7. **Wypisanie danych**: out<<

8. **Wczytanie danych**: in>>

9. **Wyrażenia**

```
a = 3;
b = 3 * a;
c = silnia(3);
d = Obiekt(3);
out<< 'tekst';
silnia(5);
```

Każde wyrażenie musi być zakończone średnikiem.
Język jest dynamicznie typowany.

Złożone elementy

1. Intrukcja warunkowa **when**

```
when <warunek> {
    ...
} else when <warunek> {
    ...
} else {
    ...
}

when i > 0 {
    out << 'Wieksza od zera';
} else when i < 0 {
    out << 'Mniejsza od zera';
} else {
    out << 'Równa zero';
}
```

   Kod umieszczony wewnątrz { } jest wykonany w przypadku spełnienia warunku lub dla intrukcji else w przypadku, gdy żaden z wcześniejszych warunków nie został spełniony.

2. Pętle **loop**

```
loop(<start>; <koniec>; <krok>) {
    ...
}

loop( i = 0; i < 5; i = i + 1){...}
```

   Pętla działa dopóki jest spełniony warunek końca.

3. Definiowanie metod/funkcji

```
nazwaMetody([<argumenty>])[extends <NazwaObiektu>] {...}
dodaj(listaZakupów OUT, produkt IN)extends Klient{...}
```

Argumenty nie są obligatoryjne, występują 3 typy argumentów:

```
arg1 IN, arg2 OUT, arg3 IN OUT
```

Słowo kluczowe **IN** oznacza, iż argument jest tylko do odczytu, **OUT** tylko do zapisu. W przypadku, gdy chcemy przekazać zmienną do funkcji i ją w niej edytować stosujemy **IN OUT**. Domyślnie wszystkie argumenty są traktowane jakby występowało po nich słowo kluczowe **IN**.

Gdy nie chcemy definiować metod wewnątrz obiektu stosujemy zwrot **extends nazwaObiektuDlaKtóregoDefiniujemyMetodę**

Możliwe jest również przeciążanie metod tzn. definiowanie wielu metod o tej samej nazwie lecz o innej ilości argumentów.
W przypadku tworzenia metod należy zwrócić szczególną uwagę na nazewnictwo argumentów, nazwy te nie mogą pokrywać się z atrybutami obiektu. W przypadku nie przestrzegania powyższej reguły interpreter zwróci stosowny błąd.

Język obiektowy

4. Obiekt **object**

```
NazwaObiektu [extends NazwaObiektuBazowego]{
    attr = int | stała_znakowa | None;

    -- Konstruktor obiektu z wywołanym domyślnym
    -- konstruktorem dla obiektu bazowego
    NazwaObiektu() {}


    # Konstruktor obiektu z jawnie wywołanym
    # konstruktorem dla obiektu bazowego
    NazwaObiektu([<argumenty>]):ObiektBazowy([<argumenty>]) {}

    # Przeciążenie operatora
    operator == (lhs, rhs, result OUT) {
        ...
    }

}
```

Obiekt jest najbardziej złożonym elementem języka.

- Przez konstruktor domyślny będę rozumiał kontruktor bezargumentowy.
Możliwe jest dziedziczenie jednobazowe poprzez użycie słowa kluczowego **extends** i podania nazwy obiektu bazowego. W przypadku, gdy wywołujemy konstruktor dla obiektu i nie podamy konstruktora obiektu bazowego zostanie wywołany konstruktor domyślny dla owego obiektu.

- Wywołanie konstruktora tworzy instancję obiektu. Zostaje ona przypisana do pierwszego argumentu posiadającego flagę **OUT**. W przypadku stworzenia konstruktora, w którym żaden z argumentów nie posiada flagi **OUT** nie jest możliwe przypisanie utworzonego obiektu do zmiennej. Mają jednak konstrukturory tego typu inne zastosowania. Wykorzystywane są w momencie, gdy występuje dziedziczenie. W momencie tworzenia obiektu posiadającego klasę bazową nie jest najpierw tworzony obiekt klasy bazowej zatem konstruktor nie musi posiadać argumentu ze zmienną **OUT**. W momencie, gdy nie chcemy, aby istniała możliwość tworzenia obiektów klasy bazowej możemy stworzyć jedynie konstruktor oparty na argumentach z flagą **IN**. Można traktować taką klasę na wzór klasy abstrakcyjnej w C++.
- Atrybuty są publiczne i podczas deklaracji muszą być zainicjowane.

Język obiektowy

Tworzenie programu

1. Interpreter będzie poszukiwał funkcji main, którą wywoła

```
main() {}
```

2. Maksymalna długość identyfikatorów to 30 znaków.

Przykład

```
Vehicle {
    id = None;
    company_name = None;
    Vehicle() {}
    Vehicle(id, cn) {
        this.id = id;
        company_name = cn;
    }
}
Car extends Vehicle{
    max_speed = 100;

    Car(){}
    Car(ms, id, cn, car out):Vehicle(id, cn) {
        max_speed = ms;
    }
}
_str_(txt OUT) extends Car {
    txt = id;
}
main() {
    id = 1;
    company = 'Toyota';
    speed = 100;
    car = None;
    Car(speed, id, company, car);
}
```

Język obiektowy

```
(*Notacja EBNF*)

program = { obiekt | funkcja };

obiekt = identyfikator, [ słowo_kluczowe_extends, identyfikator ], lewy_nawias_klamrowy,
{ atrybut | metoda | przeciążanie_operatora }, prawy_nawias_klamrowy;

metoda = identyfikator, lewy_nawias, [ lista_argumentów ], prawy_nawias, [ dwukropek,
    identyfikator, lewy_nawias, [ lista_argumentów ], prawy_nawias ], blok;

blok = lewy_nawias_klamrowy, { instrukcja }, prawy_nawias_klamrowy;

atrybut = identyfikator, [ lewy_nawias_klamrowy, [słowo_kluczowe_get, średnik],
    [słowo_kluczowe_set, średnik], prawy_nawias_klamrowy ], operator_przypisania,
    (  stała_znakowa | słowo_kluczowe_none | operator_indeksu |
    wyrażenie_arytmetyczne ), średnik;

przeciążanie_operatora = słowo_kluczowe_operator, operator, lewy_nawias,
    argument, [słowo_kluczowe_in], przecinek, argument, [słowo_kluczowe_in], przecinek,
    słowo_kluczowe_out, prawy_nawias, blok;

funkcja = identyfikator, lewy_nawias, [ lista_argumentów ], prawy_nawias
    [słowo_kluczowe_extends, identyfikator ], blok;

wywołanie_funkcji_metody = wywołanie_metody | wywołanie_funkcji;
wywołanie_metody = identyfikator, kropka, identyfikator, lewy_nawias,
    [ lista_argumentów ], prawy_nawias, średnik;
wywołanie_funkcji = identyfikator, lewy_nawias, [ lista_argumentów ],
    prawy_nawias, średnik;

instrukcja_warunkowa = słowo_kluczowe_when, warunek, blok,
    { słowo_kluczowe_else, słowo_kluczowe_when, blok },
    [ słowo_kluczowe_else, warunek, blok ];

pętla = słowo_kluczowe_loop, lewy_nawias, ( przypisanie, średnik, warunek,
    średnik, krok | identyfikator, dwukropek, identyfikator ),
    prawy_nawias, blok;

instrukcja = przypisanie | wywołanie_funkcji_metody | instrukcja_warunkowa | pętla |
    obsługa_wejścia_wyjścia;

przypisanie = [ słowo_kluczowe_this, kropka ], identyfikator, operator_przypisania,
```

```
    wyrażenie_arytmetyczne, średnik;

obsługa_wejścia_wyjścia = obsługa_wejścia | obsługa_wyjścia;
obsługa_wejścia = słowo_kluczowe_in, operator_wczytywania, identyfikator, średnik;
obługa_wyjścia = słowo_kluczowe_out, operator_wypisywania, (wyrażenie_arytmetyczne |
    stała_znakowa), średnik;

krok = identyfikator, operator_przypisania, wyrażenie_arytmetyczne;

(*
Warunek pierwszeństwo:
1. operator_negacji
2. operator_relacyjny
3. operator_logiczny
*)

warunek = składowa_warunku, { operator_logiczny, składowa_warunku };
składowa_warunku = czynnik, { operator_relacyjny, czynnik };
czynnik = [ operator_negacji ], lewy_nawias, warunek, prawy_nawias |
    wyrażenie_arytmetyczne;

wyrażenie_arytmetyczne = składowa, { (operator_dodawania | operator_odejmowania), składo
składowa = element, { operator_mnożenia, element };
element = int | identyfikator | wywołanie_funkcji_metody | odwołanie_do_atrybutu |
    lewy_nawias, wyrażenie_arytmetyczne, prawy_nawias |
    operator_odejmowania, element;

(*
Wyrażenie_arytmetyczne pierwszeństwo:
1. operator_mnożenia
2. operator_dodawania
*)

odwołanie_do_atrybutu = (słowo_kluczowe_this | identyfikator), kropka, identyfikator;

identyfikator = [:alpha:], [:word:]*;

lista_argumentów = { argument_z_przecinkiem }, argument;
argument = identyfikator, [ słowo_kluczowe_in ], [ słowo_kluczowe_out ];
argument_z_przecinkiem = argument, przecinek;

operator = operator_matematyczny | operator_logiczny | operator_przypisania |
```

```
    operator_negacji | operator_relacyjny;

operator_przypisania = "=";
operator_negacji = "!";
operator_relacyjny = "<=" | ">=" | "==" | "!=" | "<" | ">";
operator_logiczny = "||" | "&&";
operator_matematyczny = operator_dodawania | operator_mnożenia | operator_odejmowania;
operator_odejmowania = "-";
operator_dodawania = "+";
operator_mnożenia = "*" | "/";
operator_indeksu = "[]";
operator_wypisywania = "<<";
operator_wczytywania = ">>";

słowo_kluczowe_else = "else";
słowo_kluczowe_when = "when";
słowo_kluczowe_loop = "loop";
słowo_kluczowe_operator = "operator";
słowo_kluczowe_none = "none";
słowo_kluczowe_extends = "extends";
słowo_kluczowe_in = "in";
słowo_kluczowe_out = "out";
słowo_kluczowe_this = "this";
słowo_kluczowe_get = "get";
słowo_kluczowe_set = "set";

przecinek = ",";
dwukropek = ":";
średnik = ";";
kropka = "."
lewy_nawias = "(";
prawy_nawias = ")";
lewy_nawias_klamrowy= "{";
prawy_nawias_klamrowy = "}";

int  = "0" | ( cyfra_bez_zera, { cyfra });
stała_znakowa = ( "\"", ciag_znakow, "\"") |  ("\'", ciag_znakow, "\'" );


cyfra_bez_zera = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
cyfra = [:digit:];
```

```
ciag_znakow = [:alnum:]* | [:punct:]* | [:blank:]*;

[:alnum:] = [a-zA-Z0-9];
[:alpha:] = [a-zA-Z];
[:digit:] = [0-9];
[:word:] = [a-zA-Z0-9];
[:punct:] = [ ! " # $ % & ’ ( ) * + , - . / : ; < = > ? @ [ \ ] ^ _ ‘ { | } ~ ];
[:blank:] = space or tab;
```

Sposób uruchomienia

Interpreter otrzyma plik z kodem źródłowym lub stałą znakową reprezentującą kod, który następnie zinterpretuje i wykona. Źródło odpowiada za pobieranie kolejnych bajtów danych, następnie lekser prosi źródło o kolejne znaki i na ich podstawie tworzy kolejne tokeny. Tworzy je jednak w momencie, gdy parser poprosi o to, aby lekser dał mu kolejny token. Parser na podstawie otrzymanych tokenów utworzy drzewo przeszukiwań wykorzystywane przez interpreter i ewentualne obiekty pomocnicze. W przypadku przekroczenia dozwolonej długości identyfikatora, wprowadzeniu danych niezgodnych z gramatyką lub innych niedozwolonych operacjach użytkownik zostanie poinformowany o błędzie wykonania. W momencie, gdy dane są zgodne z gramatyką interpreter rozpoczny interpretację. Kontrola błędów występuje w trakcie interpretacji kodu zatem użytkownik może zostać poinformowany o błędzie w trakcie działania programu. Zostanie on w tym momencie przerwany, a na ekranie pojawi się komunikat, jaki błąd wystąpił.

Testowanie

Przeprowadzone zostały testy jednostkowe sprawdzające działanie leksera.
Przeprowadzone zostały testy działania parsera wraz z lekserem.
Przeprowadzone zostały też testy działania całej struktury interpretera.

Język obiektowy

Struktura

Na podstawie gramatyki zostały utworzone klasy przechowujące dane. Każda z klas posiada metodę execute, która jako argument przyjmuje kontekst. Kontekst odpowiada za przechowywanie aktualnego stanu interpretowanego programu. W metodach execute obsłużone są funkcjonalności poszczególnych klas. Np. dla klasy MethodCall metoda execute stworzy nowy kontekst i wywoła metodę. Metody execute wyłapują też błędy w trakcie interpretacji kodu. Ropoczęcie interpretacji polega na wykonaniu metody execute na bezargumentowej funkcji main. Występuje też klasa odpowiedzialna za generowanie informacji o błędach. Pobiera ona ze źródła fragment kodu, zaczynając od tokenu poprzedzającego ten, na którym działanie interpretera zostało przerwane.

Lokalność zmiennych

Funkcja, metoda, instrukcja warunkowa **when**, pętla **loop**.
Powyższe elementy posiadają własny kontekst w trakcie działania programu. Zmienne są jednak widoczne w zagnieżdżonych kontekstach. Na przykład:

```
main(){
i = 4;
    when(i > 2){
        out<<i; # zwróci 4
        i = 2;
        a = 4;
    }
out<<i; # zwróci 2
out<<a; # błąd - zmienna 'a' nie istnieje w bieżącym kontekście
}
```

W przypadku wywołania funkcji/metody nie mają one wglądu do wcześniej utworzonych kontekstów. Przypadek ten wsytępuje jedynie dla instrukcji i pętli.

Wczytywanie danych za pomocą in»

Jeżeli wartość wprowadzona przez użytkownika może zostać przekonwertowana na integer to do zmiennej podanej po wyrażeniu **in»** przypisana zostanie wartość liczbowa, w pozostałych przypadkach przypisana zostanie stała znakowa.