Marcin Hałasa

# Interpreter języka z typem walutowym.

Celem proejktu jest stworzenie prostego języka, wysposażonego w podstawowe instrukcje, możlowość definiowania własnych funkcji oraz wbudowany typ walutowy. Użytkownik będzie miał możliwość zdefiniowania na początku pliku własnych typów walutowych wraz z ich przelicznikami na typ domyślny. Język będzie umożliwać wykonywanie prostych operacji na walutach (takich jak dodawanie, odejmowanie, mnożenie razy stała i dzielenie przez stałą).

**Funkcje jakie język powinien realizować i ich przykładowe użycie**

- zdefiniowanie własnych walut i ich przeliczników

  *default currency: dol*
  *pln: 3.40*
  *eur: 0.82*

  Użytkownik ustala walutę domyślą według której będą przeliczane inne waluty względem siebie. Dalej podaje inne waluty i ich kurs względem waluty domyślnej. W przykładzie mamy zależność 1dol = 3.40pln oraz 1dol = 0.82eur. Konwersja pln do eur nastąpi przez walutę dol.

- konwersja jednej waluty do drugiej:

  *a = 5 dol;*
  *b = a pln;*

- dodawanie i odejmowanie walut:

  *a = 5 dol;*
  *b = 3 pln;*
  *c = (a + b) dol;*
  *d = (a – b) dol;*

  Możliwe będzie dodanie do siebie albo odjęcie dwóch lub więcej walut, jeżeli typy dodawanych/odejmowanych walut nie są zgodne następuje ich konwersja do waluty domyślnej i wynik jest w takiej właśnie walucie. Możliwe jest również zrzutowanie całego wyrażenia na dowolną walutę.

- mnożenie waluty przez stałą

  *a = 5 dol;*
  *b = a * 2;*

- instrukacja warunkowa i porównywanie walut

  ```
  a = 3 dol;
  b = 5 pln;
  if ( a > b)
  {
      ...
  }
  else
  {
     ...
  }
  ```

  Możliwe będzie użycia opretatorów porównania takich jak >, >=, <, <=, ==. Jeżeli typy walut będą zgodne nastąpi zwykłe porównanie wartości, jeżeli będą się różnić to nastąpi porówanie

wartości przekonwertowanych do typu domyślnego. W warunku wyrażenia oprócz operatorów porównania może znaleźć się również zmienna lub literał jeżeli są one równe 0 to mamy fałsz a w przeciwnym przypadku prawdę.

- Pętla

```
i = 0;
a = 5.3 dol;
while (i < 10){
a = 10 * a ;
i = i + 1;
}
```

Poza typami walutowymi możliwe będzie zdefiniowanie typu zmiennoprzecinkowego (jeżeli przy deklaracji zmiennej nie piszemy żednej waluty to jest to typ zmiennoprzecinkowy), który przyda się między innymi do iteracji pętli.

- tworzenie własnych funkcji i ich wywoływanie

```
function fun(a,b){ … }          //własna funkcja
c = fun(a,b);                   //wywołanie funkcji
```

Funkcje będą mogły przyjmować parametry i zwracać wartości o dowolnym typie.

**Przykładowy kod**

default currency: dol
pln: 3.40
eur: 0.82

```
function fun(a, b){

        i = 0;
        if (a > b){
                while(i < 10){
                        if ( a > 1000){
                                break;
                        }
                        a = a + b;
                        i = i + 1;
                }
                return a;
        }else{
                i = 14;
                c = (i*b + 5pln – a)pln;
                return c dol;
        }
}

function main(){
        a = 30dol;
        b = 20pln;
        c = fun(a,b);
        print( c); }
```

**Gramatyka**

```
program  = [ configBlock ] { functionDef } ;
configBlock = "defaultCurrency" ":" id {id ":" number} ;
functionDef  = "function" id "(" [ parameters ] ")" statementBlock;
parameters =  id { "," id } ;
statementBlock = "{" { ifStatement | whileStatement | assignStatement | funCall
                       | returnStatement  | printStatement | statementBlock  } "}" ;
ifStatement = "if" "(" condition ")" statementBlock [ "else" statementBlock ] ;
whileStatement = "while" "(" condition ")" statementBlock ;
assignStatement = id  "=" expression  ";"  ;
returnStatement = "return" expression ";" ;
printStatement = "print"  "( " id  ")"  " ;" ;

expression = multiplicativeExpr { additiveOp multiplicativeExpr } ;
multiplicativeExpr = convertExpr  {multiplicativeOp convertExpr };
convertExpr = primaryExpr [ id { " " id} ];
primaryExpr = literal | id | funCall | "(" expression ")";

condition = andCond { "||"  andCond } ;
andCond = relationalCond { "&&" relationalCond } ;
relationalCond = primaryCond { relationOp primaryCond } ;
primaryCond = [ "!" ] (id | literal | "(" condition ")" ) ;

relationOp = "<" | ">" | "<=" | ">=" | "==" | "!=" ;
additiveOp = "+" | "-" .;
multiplicativeOp = "*" | "/";

literal = number [ id ];
funCall = id "(" [ parameters ] ")"  ";" ;
id = letter { digit | letter } ;
number = [ "-" ]   ( "0"  "." | nonzerodigit { digit } [ "." ] ) { digit };
 letter = "a".."z" | "A".."Z" ;
digit = "0" | nonzerodigit;
nonzerodigit = "1".."9";
```

**Lista tokenów**

"defaultCurrency" ":" "function" "(" ")" "," "{" "}" ";" "if" "else" "while" "return" "!" "="

"||" "&&" "==" "!=" "<" ">" "<=" ">=" "+" "-" "*" "/"

**Obsługa programu**

Program będzie prostą aplikacją konsolową, uruchamianą poprzez wywołanie wraz z prametrami uruchomieniowymi. Pierwszy z nich będzie reprezentował ścieżkę do pliku ze skryptem do interpretacji a drugi ścieżkę do pliku ze zdefiniowanymi walutami i ich przlicznikami.

**Budowa programu**

Program będzie złożony z modułów odpowiedzialnych za kolejne etapy analizy plików wejściowych.

Cały proces analizy i wykonywania skryptów będzie odbywał się w następujących etapach:
1. Analiza leksykalna (moduł lexera)
2. Analiza składniowa (moduł parsera)
3. Analiza semantyczna (moduł analizatora semantycznego)
4. Wykonanie zbioru instrukcji (moduł wykonawczy)

**Struktury do przechowywania danych:**

- Node - główna klasa dla obiektów w drzewie, dziedziczą po niej wszystkie obiekty drzewa
- Program – klasa przechowująca całą zawartość pliku programu, zawiera zbiór funkcji programu oraz blok konfiguracyjny
- ConfigBlock – klasa przechowująca nazwę domyślnej waluty oraz mapująca nazwy innych walut na ich przeliczniki do waluty domyślnej
- Function – klasa przechowująca nazwę funkcji, jej parametry oraz blok
- StatementBlock – klasa reprezentująca blok kodu zawiera listę instrukcji oraz obiekt zasięgu (Scope)
- AssignStatement – zawiera zmienną oraz przpisaną jej wartość w postaci wyrażenia
- AdditiveExpression – składa się z listy operandów i operacji, lista operacji może zawierać znak + lub – a operandy są typu MultiplicativeExpression
- MultiplicativeExpression - składa się z listy operandów i operacji, lista operacji może zawierać znak * lub / a operandy są typu ConvertExpression
- ConvertExpression - składa się z jednego operandu typu PrimaryExpression oraz listy walut na które wyrażenie ma być konwertowane
- PrimaryExpression – klasa zawierająca jedno z podstawowych wyrażeń
- Condition – klasa składa się z operatora na najwyższym poziomie może to być operator &&, później operator || a najniżej może być jeden z operatorów relacji, dla pojedynczego opearandu typ operatora to undefined. Klasa składa się również z listy opearandów dla których te opeartory są stosowane i z pola mówiącego czy dany opearand jest zanegowany
- IfStatement – klasa reprezentująca instrukcję warunkową, zawiera warunek i bloki true i else
- WhileStatement - klasa reprezentująca pętlę, zawiera warunek oraz blok kodu
- Variable – klasa reprezentująca zmienną w progrmie, zawiera pole na jej nazwę oraz wartość
- Literal – klasa reprezentująca stałą (walutową lub zwykłą) w programie, zawiera wartość double oraz nazwę waluty (dla zwykłych stałych wartość null)
- Scope – klasa przechowująca listę zmiennych znajdujących się w danym zasięgu oraz referencję na swojego rodzica o ile go posiada