Język programowania umożliwiający operacje na walutach

Paweł Budniak (300193)

1. Opis Funkcjonalny

Tematem projektu jest implementacja interpretera do własnego języka ułatwiającego operacje na walutach. Język udostępnia wbudowany typ `currency`, reprezentujący walutę. Zdefiniowana jest specjalna składnia umożliwiająca wskazanie typu waluty (np. USD, PLN). Aby umożliwić wykonywanie operacji arytmetycznych na walutach różnych typów wymagane jest uprzednie rzutowanie. Takie rozwiązanie rozwiązuje problem niejasności jaki typ powinna mieć waluta wynikowa (np. przy działaniu 3.2pln + 4.2usd). Rzutowanie uwzględnia przelicznik między walutami, który można deklaratywnie zdefiniować w programie. W związku z tym, że w przypadku obliczeń finansowych wymagana jest jak największa dokładność, do wewnętrznej reprezentacji liczb zmiennoprzecinkowych jest zastosowany typ tekstowy. Język wspiera dwa takie typy - waluty i zwykłe floaty, które mogą być użyte jako skalar przez który mnożona jest waluta. Operacje arytmetyczne na typach zmiennoprzecinkowych będą operować na reprezentacji tekstowej.

1.1 Typy

A. Typy liczbowe
   - `currency` - waluta, reprezentowana tekstowo
   - `number` - Skalar. Zwykły typ zmiennoprzecinkowy, również reprezentowany tekstowo

B. Inne typy
   - `bool` - zmienna boolowska
   - `string` - zmienna tekstowa

1.2 Język umożliwia/udostępnia:

- Wykonywanie podstawowych operacji arytmetycznych na skalarach: dodawanie, odejmowanie, mnożenie, dzielenie. Używane są znane priorytety operatorów
- Pisanie komentarzy
- Używanie w kodzie stałych każdego z typów
- Podstawowo zdefiniowane typy walut: PLN, USD, EUR, GBP, CHP
- Definiowanie przelicznika z jednej waluty na drugą
- Rzutowanie z jednej waluty na drugą z uwzględnieniem przelicznika
- Odejmowanie, dodawanie walut tego samego typu
- Mnożenie waluty przez skalar
- Dzielenie walut tego samego typu, wynikiem jest `number`
- Porównywanie typów liczbowych (tutaj nie jest wymagane jawne rzutowanie walut różnego typu)

- Wyrażenia logiczne OR i AND
- Warunkowe wykonywanie instrukcji przy pomocy `if`
- Stosowanie pętli `while`
- Definiowanie bloków kodu za pomocą klamer
- Lokalność zmiennych w blokach kodu
- Definiowanie funkcji, możliwość wywołań rekurencyjnych
- Wbudowaną funkcję print(), przyjmującą jako argument `string`
- Przeciążenie operatora "+" dla typu `string`
- Niejawne rzutowanie z typów liczbowych na `string`

1.3 Założenia

- Instrukcje kończą się średnikiem
- Komentarz zaczyna się znakiem '#' i kończy znakiem nowej linii
- Przekazywanie parametrów do funkcji odbywa się przez kopię
- "Główny" kod programu musi znajdować się w ciele funkcji o sygnaturze void main()
- Deklaracja zmiennej wymaga inicjalizacji w tej samej linijce

1.4 Przykładowe konstrukcje językowe

```
# przykładowy komentarz

# definicja i inicjalizacja zmiennej x, która ma typ currency i
# reprezentuje 2.33 PLN
currency x = 2.33pln;

1.23 # stała number
1 # stała number
1.23pln # stała currency
True # stała bool
"Test" # stała string

# definicja przelicznika waluty
exchange from gbp to pln 4.32;

# instrukcja warunkowa
currency pounds= 0.54gbp;
if (2.33pln < franks){
pounds= 0.6gbp;
}

# pętla
number i = 0;
while (i < 10){
i = i + 1;
```

```
}

# definicja funkcji
number random_int(){
return 3;
 }

# funkcja nie zwracająca nic
void printDouble (int x){
print(2*x);
}

# zmienna typu bool
bool x = 3 > 2 || 3 == 4;

# dodawanie walut i rzutowanie
currency franks = 0.53chp;
currency zlotys = 2.33pln;
currency my_franks = franks + [chp]zlotys;

# blok kodu ze zmienną lokalną x
if (True){
      number x = 3;
}
```

2. Formalny opis gramatyki

Między poniższe konstrukcje można wstawiać białe znaki
Komentarze są usuwane przez lekser i ignorowane w gramatyce

```
program = { function_def  };
function_def = fun_type_specifier, identifier, "(", arg_def_list, ")", block;
fun_type_specifier = type_specifier | "void";
block = "{" {statement } "}";


statement = assignOrFunCall | exchange_declaration | if_statement | loop |
return_statement;
return_statement = “return”, rvalue;

assignOrFunCall = (identifier, ( rest_of_funcall| rest_of_assignment)) |
type_and_id, rest_of_assignment

rest_of_assignment = "=", rvalue, ";";
rest_of_funcall = "(" arg_call_list ")", ";";
```

```
type_specifier = "bool" | "currency" | "number" | "string";
arg_def_list = [ type_and_id {",", type_and_id} ];
arg_call_list = [ rvalue {",", rvalue} ];
exchange_declaration =  "exchange from", currency_type, "to", currency_type,
rvalue, ";";
if_statement = "if", "(", bool_expression, ")", block;
loop = "while", "(", bool_expression, ")", block;
lvalue = type_and_id | identifier;
rvalue = arithmetic_expression | bool_expression;

arithmetic_expression = [unary_op], term , { additive_operator , term};
term = factor , { multiplicative_operator , factor};
factor = [unary_op], simple_value |  ( "(" , arithmetic_expression, ")" );

bool_expression = bool_term, {"||", bool_term};
bool_term = bool_factor, {"&&", bool_factor};
bool_factor = [unary_op],
   comparison | simple_value | ( "(", bool_expression, ")" );

type_and_id = type_specifier, identifier;
comparison = simple_value, logic_operator, simple_value;
simple_value =  identifier | literal | function_call;

unary_op = "!" | "-" | cast_op;
cast_op = "[", currency_type, "]";

literal = str_literal | bool_literal | number_literal | currency_literal;
currency_literal = number_literal, currency_type;
```

Między poniższe konstrukcje nie można wstawiać białych znaków

```
number_literal = decimal | integer;
bool_literal = "True" | "False"
str_literal = "'", [^']*  , "'";
identifier = letter, {alnum};
alnum = letter | digit ;
letter = #'[A-Za-z]';
decimal = integer , "." , digit , { digit };
integer = "0" | (non_zero_digit, {digit});
digit = "0" | non_zero_digit;
non_zero_digit = #'[1-9]*';

additive_operator = "+" | "-";
```

```
multiplicative_operator = "*" | "/";
logic_operator = "<" | ">" | "<=" | ">=" | "==" | "!=";
currency_type = "gbp" | "eur" | "pln" | "chp" | "usd";
```

3. Podział na moduły

- Analizator leksykalny - Ze znaków z pliku źródłowego generuje tokeny i przekazuje je do analizatora składniowego
- Analizator składniowy - Sprawdza poprawność gramatyczną otrzymanych tokenów i generuje na ich podstawie konstrukcje językowe i przekazuje je do analizatora semantycznego
- Analizator semantyczny - Analizuje semantyczną poprawność otrzymanych konstrukcji, generuje drzewo instrukcji i przekazuje je do interpretera
- Interpreter - Jego zadaniem jest wykonanie programu na podstawie otrzymanego drzewa instrukcji.
- Moduł obsługi błędów - Zawiera procedury obsługi błędów na różnych poziomach działania programu.
- Moduł testujący - Odpowiada za testowanie innych modułów

4. Opis Implementacji

Program zostanie napisany przy użyciu języka java. Do reprezentacji wewnętrznej typów `number` i `currency` zastosowana zostanie klasa `BigDecimal`.

Wejściem do interpretera jest plik z kodem źródłowym, a wyjściem wynik instrukcji print z kodu.

Przykładowe uruchomienie programu:
`$ java currencyIntp <source file>`