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
- Zdefiniowanie obsługiwanych typów walut i ich przeliczników w pliku data/exchangeRates.json
- Definiowanie przelicznika z jednej waluty na drugą w kodzie programu
- 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 również wymagane są waluty tych samych typów, ponieważ przeliczniki walut mogą być niesymetryczne, więc użytkownik powinien jawnie zdecydować którego przelicznika chce użyć przy pomocy rzutowania)
- 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
- Lokalność zmiennych w funkcji
- Definiowanie funkcji, możliwość wywołań rekurencyjnych
- Wbudowaną funkcję print(), przyjmującej dowolną liczbę argumentów dowolnego typu - wszystkie rzutuje na String (tutaj typ javowy) i wypisuje
- Przeciążenie operatora "+" dla typu `string`
- "Cieniowanie" zmiennych zewnętrznych przez zmienne lokalne

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
- Wszystkie słowa kluczowe pisane są w całości małymi literami

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 < [pln] 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 (number 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 = bool_expression;

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

bool_expression = bool_term, {"||", bool_term};
bool_term = bool_factor, {"&&", bool_factor};
bool_factor = [unary_op],
   comparison | arithmetic_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 = #'[a-z]';
```

3. Podział na moduły

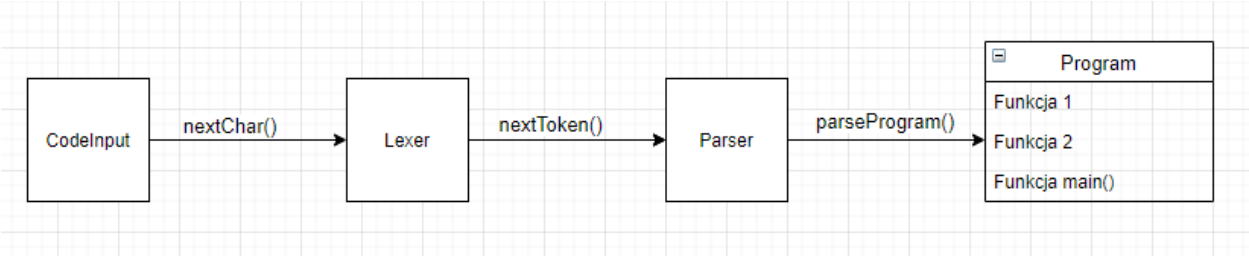
- reader - odpowiada za odczyt z pliku/strumienia danych. Przekazuje informacje o pozycji znaków w pliku, co zostaje wykorzystane przy obsłudze błędów. Podczas obsługi błędów pozwala na wyświetlenie okolicy w której wystąpił błąd.
- lexer - analizator leksykalny - Ze znaków z pliku źródłowego generuje tokeny i przekazuje je do analizatora składniowego
- parser - Analizator składniowy - Sprawdza poprawność gramatyczną otrzymanych tokenów i generuje na ich podstawie struktury językowe
- structures - struktury językowe generowane przez parser. Konstrukcje typu *Statement* implementują metodę `execute` , a *RValue* metodę `getValue`, odpowiadające za właściwą część wykonywalną interpretera.
- types - definicje podstawowych typów języka i operacji na nich.
- execution - zawiera klasy pomocnicze wykorzystywane przez moduł `structures` do części wykonującej program, m.in klasę *Scope* zarządzającą lokalnością zmiennych w blokach i funkcjach.
- error - zawiera klasę odpowiadającą za obsługę błędów i definicję bazowej klasy dla wyjątków wykorzystywanych w programie - *InterpreterException*
- test/java - moduł testujący, odpowiada za testowanie innych modułów

4. Opis Implementacji

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

Wejściem do interpretera jest plik z kodem źródłowym i plik z definicjami walut, a wyjściem wynik instrukcji print z kodu. Definicje walut reprezentowane są w formacie json w postaci słownika słowników przeliczników walut. Kody walut wczytane z pliku .json są traktowane jako słowa kluczowe języka podczas analizy składniowej. Przykładowy plik:

Poniższy diagram pokazuje proces analizy kodu przed wykonaniem. Program udostępnia metodę execute(), która powoduje wykonanie funkcji main() i wypisanie na standardowe wyjście rezultatów funkcji print() w kodzie programu.



Ciekawsze zagadnienia implementacyjne:

- Każda struktura mogąca pojawić się po prawej stronie operatora przypisania dziedziczy po abstrakcyjnej klasie *RValue* - musi zaimplementować metodę *getValue*() zwracającą *CType*
- *CType* to abstrakcyjna klasa bazowa dla wszystkich podstawowych typów języka nazywanych wg konwencji C<Nazwa typu z wielkiej litery>, np: *CString*.
- Każdy z typów posiada swój *truthValue()* czyli wartość boolowską używaną w kontekście użycia typu jako warunek instrukcji *if* lub *while* lub w wypadku zastosowania na zmiennej operatora “!”. Dla *CNumber* i *CCurrency* wartość ta jest fałszywa wtedy i tylko wtedy kiedy mają wartość 0 (niezależnie od precyzji), a *CString* wtedy kiedy jest to pusty napis.
- Każda z klas w module `structures` nadpisuje funkcję *toString()*. Ułatwia to testowanie i debugowanie.
- Obiekty struktur złożonych typu *BoolTerm* są tworzone tylko wtedy, kiedy mają co najmniej dwa operandy (wbrew formalnemu opisowi gramatyki). Dzięki temu unikamy niepotrzebnego zagnieżdżania struktur, co również ułatwia testowanie i debugowanie.
- Żeby testowanie parsera przy pomocy metod *toString()* było rzetelne zostały napisane osobne testy dla metod *toString()* struktur dziedziczących po *RValue* (tylko dla nich testy parsera wykorzystują reprezentację tekstową).
- Podczas wykonywania operacji arytmetycznych na obiektach *CType* występuje problem tzw. *Double dispatch*. Wynika to stąd, że mechanizm polimorfizmu w javie zapewnia jedynie tyle, że wywołania metod na referencjach do typu bazowego są wirtualne, ale przypisania parametrów metod są już nie są - są wykonywane na etapie kompilacji. Oznacza to, że próbując dodać 2 obiekty typu *CNumber* mając referencje na nie typu *CType*, na etapie kompilacji nie wiemy że należy do tego użyć metody z klasy *CNumber* przyjmującej argument typu *CNumber*. Problem można rozwiązać dokonując sprawdzania operatorem instanceof i rzutowaniem w czasie wykonania programu. Innym rozwiązaniem, zastosowanym w projekcie jest wykorzystanie pewnej wersji wzorca wizytatora. Tzn. trzeba dodać co najmniej jeden dodatkowy poziom wywołań metod, mający za zadanie dynamiczne uzyskanie informacji o typie drugiego obiektu.

W przykładzie poniżej odpowiada za to funkcja *acceptAdd* - wywołujemy w niej funkcję visitAdd z odwrotną kolejnością parametrów, dzięki temu parametr *other* skorzysta z wirtualnego wywołania, a parametr this, będący wywoływany z klasy *CNumber* jest już świadomy swojego typu. Dowiedzieliśmy się więc dynamicznie informacji o typach obydwu parametrów. W implementacji stosowane są dodatkowo funkcje *visit**, które mają za zadanie odwrócić kolejność parametrów do tej oryginalnej. Jest to istotne przy nieprzemiennych operacjach jak odejmowanie, ale też np. dodawanie do *CString*.

```
@Override
public CType acceptAdd(CType other) { return other.visitAdd(this); }

@Override
protected CType visitAdd(CNumber other) { return other.add(this); }
```

```
public CNumber add(CNumber other) { return new CNumber(number.add(other.getValue())); }
```

- Pilnowanie zadeklarowanych typów zmiennych odbywa się tylko w fazie wykonywalnej interpretera. To znaczy błąd zostanie zgłoszony dopiero kiedy podczas egzekucji np. do zmiennej typu *CNumber* spróbujemy przypisać wartość typu *CString*, czy spróbujemy zwrócić z funkcji wartość niezgodną z jej zadeklarowanym typem.
- W wielu strukturach przechowywane są obiekty Token konstruowane na poziomie Lexera. Zawierają one informacje m.in. o pozycji w pliku źródłowym. Takie informacje przekazywane są w atrybutach wyjątków i moduł obsługi błędów wykorzystuje je żeby pokazać użytkownikowi konkretne miejsce w którym wystąpił błąd
- Funkcje mogą być definiowane tylko na “zerowym” poziomie drzewa programu - tzn. tym samym co funkcja main() i ich zasięg (ang. *scope*) jest globalny.
- Nie ma możliwości przeciążania funkcji - może być tylko jedna z danym identyfikatorem.

5. Opis użytkowy

Do automatyzacji budowy aplikacji wykorzystywane jest narzędzie *maven*.
Przykładowe uruchomienie programu:

Wejście do głównego folderu (zawiera m.in folder src/):
```
$ cd currencyInterpreter
```

Pobranie zależności:
```
$ mvn install
```

Uruchomienie klasy Main:
```
$ mvn exec:java
```

Uruchomienie testów jednostkowych:
```
$ mvn test
```