TKOM - dokumentacja końcowa projektu Currex

Kamil Strójwąs

Czerwiec 2024

1 Opis wstępny

Celem projektu jest stworzenie interpretera języka programowania umożliwiającego operacje walutowe na danych z różnymi zdefiniowanymi walutami. W języku tym zdefiniowane są podstawowe typy danych znane z innych języków programowania wysokiego poziomu. Ponadto, występuje typ walutowy, będący reprezentacją kwoty pieniężnej w danej walucie. Interpreter jest stworzony w języku Java.

2 Specyfikacja języka

W języku istnieją następujące silnie typowane, statyczne typy wbudowane:

- int reprezentujący typ całkowitoliczbowy,
- float będący typem zmiennoprzecinkowym,
- bool czyli typ reprezentujący wartości "prawda"lub "fałsz",
- string będący ciągiem znaków,
- currency będący typem złożonym

Typ cureency składa się z następujących pól:

- name - nazwa waluty, możliwe wartości są jedynie takie które znajdują się w pliku konfiguracyjnym
- value - wartość tej waluty wyrażona w typie stałoprzecinkowym

Dla typu currency domyślnie możliwe są operacja dodawania, odejmowania, mnożenia, dzielenia i przyrównywania. Nie są dozwolone operacje na zmiennych typu walutowego z różnymi walutami, przypadek taki zgłosi błąd. Jeżeli jednak i tak chcemy takiej operacji dokonać, jedna ze zmiennych musi mieć walutę zmienioną na tą którą ma druga zmienna.

Operator przyrównania '==' dla typu currency zwraca wartość typu bool. W przypadku w którym obie zmienne mają taką samą wartość w polu "name"i taką samą wartość w polu "value", przyrównanie zwróci wartość "true". W każdym innym przypadku otrzymamy "false". Operatory przyrównania '<=', '<', '>=' i '>' mogą zostać wykorzystane dla typu walutowego tylko dla zmiennych mających ten sam rodzaj waluty po czym następuje porównanie wartości w polu "value", w innym przypadku zgłoszony zostanie błąd.

Możliwe jest również rzutowanie wartości waluty na inną walutę za pomocą operatora rzutowania '@'. Nie zmienia on wartości nominalnej waluty którą chcemy rzutować. Możliwa jest też konwersja zmiennej typu walutowego z jednej waluty w inną wykorzystując jego przelicznik zdefiniowany w tabeli konwersji znajdującej się w pliku konfiguracyjnym za pomocą operatora '->'. Oba operatory mają lewostronną łączność. Zmienne są mutowalne i dozwolona jest ich modyfikacja po stworzeniu.

Każda linijka zakończona jest znakiem średnika ';'. Bloki kodu odzielone są klamrami '{}'. Komentarze zaczynają się od podwójnych forward slashy '//'. Do wypisania czegoś do konsoli stosowana jest funkcja

wbudowana print(). Wartości zmiennych są kopiowane przy przypisaniu i podaniu jako argument wywołania funkcji (przekazywanie przez wartość).

Język wymaga istnienia funkcji main, bloki kodu nie mogą być również puste i muszą zawierać co najmniej jedną instrukcję, w innym wypadku sygnalizowane jest to jako błąd.

Priorytety operatorów wyglądają następująco, 1 to najwyższy priorytet, 9 to najniższy:

Priorytet	Operator	Symbol
1	Dostęp do pola	'.'
2	Operatory unarne	'-', '!'
3	Operatory rzutowania	'@', '->'
4	Mnożenie i dzielenie	'*', '/'
5	Dodawanie i odejmowanie	'+', '-'
6	Przyrównanie	'==', '!=', '<=', '<', '>', '>='
7	AND	'&&'
8	OR	'||'
9	Przypisanie	'='

3 Gramatyka

3.1 Pliku konfiguracyjnego

```
tabela             = waluty, {rząd_konwersji};
rząd_konwersji     = identyfikator, {przelicznik}, ";";
waluty             = {identyfikator}, ";";
przelicznik        = float;
```

3.2 Języka

```
program            = {definicja_funkcji};
blok               = "{", {instrukcja}, "}";
instrukcja         = (deklaracja | przypisanie | return), ";" | wyrażenie_if | wyrażenie_while;

przypisanie        = wyrażenie_dostępu, [operator_przypisu, wyrażenie];
wyrażenie_while    = "while", "(", wyrażenie, ")", blok;
wyrażenie_if       = "if", "(", wyrażenie, ")", blok, {"else", ["if", "(", wyrażenie, ")"], blok};
return             = "return", [wyrażenie];

wyrażenie          = wyrażenie_or;
wyrażenie_or       = wyrażenie_and, {operator_or, wyrażenie_and};
wyrażenie_and      = porównanie, {operator_and, porównanie};
porównanie         = wyrażenie_dodania, [przyrównanie, wyrażenie_dodania];
wyrażenie_dodania  = wyrażenie_mnożenia, {operator_addytywny, wyrażenie_mnożenia};
wyrażenie_mnożenia = wyrażenie_rzutu, {multiplikacje, wyrażenie_rzutu};
wyrażenie_rzutu    = wyrażenie_unarne, {(operator_rzutu | operator_wymiany), wyrażenie_unarne}
wyrażenie_unarne   = [operator_unarny], (wyrażenie_dostępu | literał);
wyrażenie_dostępu  = podst_wyrażenie, {operator_dostępu, iden_lub_wywołanie};
podst_wyrażenie    = iden_lub_wywołanie | literał | "(", wyrażenie ")";

definicja_funkcji  = [typ], nazwa_funkcji, "(", [lista_parametrów], ")", blok;
```

```
iden_lub_wywołanie = identyfikator, ["(", [lista_argumentów], ")"];
lista_parametrów   = deklaracja, {",", deklaracja};
deklaracja         = typ, identyfikator, [operator_przypisania, wyrażenie];
lista_argumentów   = wyrażenie, {",", wyrażenie};
nazwa_funkcji      = identyfikator;

literał            = boolean | ((liczba | float), [nazwa_waluty]) | string;
typ                = "int" | "float" | "string" | "bool" | "currency";
identyfikator      = (litera | "_"), {"_" | litera | cyfra};

nazwa_waluty       = {litera}
string             = """, {znak}, """;
float              = liczba, ".", {cyfra};
liczba             = "0" | ([operator_unarny], (cyfra_bez_zera, {cyfra}));
boolean            = "true" | "false";

operator_dostepu   = ".";
operator_unarny    = "-" | "!";
multiplikacje      = "*" | "/";
operator_addytywny = "+" | "-";
przyrównanie       = "<" | "<=" | ">=" | ">" | "==" | "!=";
operator_and       = "&&";
operator_or        = "||";
operator_przypisania  = "=";
operator_rzutu     = "@";
operator_wymiany   = "->";

średnik            = ";";
znak               = cyfra | litera | symbol;
symbol             = "," | "." | "!" | "?" | ":" | "-" | "_" | "/";
litera             = "a" | "b" | ... | "z" | "A" | "B" | ... | "Z";
cyfra              = "0" | cyfra_bez_zera;
cyfra_bez_zera     = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
```

4 Sposób uruchomienia

Do uruchomienia wymagany jest plik konfiguracyjny zawierający konwersje walut w postaci tabelarycznej. Przykład takiego pliku wygląda następująco:

```
EUR USD PLN;
EUR 1 1.08 4.68;
USD 0.92 1 4.31;
PLN 0.18 0.23 1;
```

5 Konstrukcje języka

W języku, dostępne jest kilka wbudowanych konstrukcji, takich jak pętle, wyrażenia porównania lub rzutowanie walut.

5.1 Funkcje wbudowane w język

Język posiada kilka funkcju wbudowanych, które mają przede wszystkim ułatwić operacje na typie currency.

- print - wypisuje do konsoli komunikat dla użytkownika
- getCurrency - metoda typu walutowego, zwraca ona nazwę waluty używaną przez zmienną
- getBalance - metoda typu walutowego, zwraca ona nominał używany przez zmienną

5.2 Domyślne reguły języka

W języku, zdefiniowane zostało kilka reguł globalnych dotyczących całego programu:

- maksymalna długość identyfikatora - 100 znaków
- maksymalna długość zmiennej typu string - 100 znaków
- maksymalna wartość zmiennej typu int - 2147483647
- maksymalna liczba cyfr po przecinku dla zmiennej typu currency - 10
- maksymalny rozmiar tabeli konwersji - 5 rzędów x 5 kolumn
- domyślna nazwa funkcji od której rozpoczynane jest uruchamianie programu - main

Wartości te mogą być modyfikowane przez użytkowników.

5.3 Operacje arytmetyczne

Język zezwala na podstawowe operacje arytmetyczne na zmiennych typu całkowitego, zmiennoprzecinkowego i walutowego.

```
int one = 1;
int two = 2;
int three = one + two;
// three = 3

int diff = one - two;
// diff = -1

float half = 0.5;
float one_and_half = 1.5;
float multiplication = half * one_and_half;
// multiplication = 0.75
float division = one_and_half / half;
// one_and_half = 3.0
```

5.4 Dostępne operacje na typie walutowym

Z typem walutowym możemy wykonywać podstawowe operacje arytmetyczne takie jak dodawanie, odejmowanie, mnożenie i dzielenie. Wynikiem takiej operacji zawsze będzie typ walutowy. Typ ten zezwala także na rzutowanie do innej waluty bez zmiany nominału, lub przekonwertowanie jej do innej przy użyciu licznika.

```
10.45 PLN + 5 = 15.45 PLN;
10.45 PLN - 15 = -4.55 PLN;
10.45 PLN * 10.5 = 109.725 PLN;
10.45 PLN / 3.5 = 2.98571428571 PLN;
10.45 PLN @ ABC = 10.45 ABC;
// zakładając przelicznik PLN na ABC równy 1 PLN = 2.00 ABC
10.45 PLN -> ABC = 20.90 ABC;
```

Aby zachować jak najlepszą precyzję, typ walutowy jest zaokrąglany do 10 miejsc po przecinku.

5.5 Operacje na typie string

Typ string pozwala jedynie na konkatenację łańcuchów znaków za pomocą operatora dodawania.

```
string str1 = "Projekt "
string str2 = "wstępny"
string str3 = str1 + str2;
// str3 = "Projekt wstępny"
```

5.6 Porównania i instrukcje warunkowe

W języku dostępne są porównania wartości dwóch zmiennych. Możliwe jest także wykorzystanie dwóch operatorów logicznych AND (&&) lub OR (||) aby połączyć kilka warunków w jeden. Instrukcje warunkowe tworzone są za pomocą słów kluczowych "if"oraz "else", które można razem łączyć do postaci "else if"aby stworzyć bardziej zaawansowane warunki. Operator negacji logicznej (!) służy do zanegowania wartości logicznej na przeciwną, z true na false i z false na true.

```
bool isOk = true;
int value1 = 0;
int value2 = 1;

if (value == 1) {
  isOk = false;
}
else if (value == 0 && value2 >= 5) {
  isOk = true;
}
else if (value != 0 || value2 < 0) {
  isOk = true;
}
else {
  isOk = !isOk;
}
// isOk = false
```

5.7 Wyświetlanie w konsoli

W języku dostępne jest wyświetlanie na wyjście komunikatów w konsoli dla użytkownika za pomocą słowa kluczowego "print".

```
int one = 1;
print(one);
// 1

int ten = 10;
print(one + ten);
```

```
// 11

print(ten >= one);
// true

currency euros = 2.50 EUR;
currency dollars = 1.00 USD;
print(euros + dollars @ EUR);
// 3.50 EUR
```

5.8 Funkcje

W języku możliwe jest definiowanie swoich własnych funkcji. Mogą one zwracać jakąś wartość, ale jest także dozwolone definiowanie funkcji nie zwracających niczego.

```
currency first = 1.00 ABC;

currency addTen(currency value) {
  currency result = value + 10.00 @ value.getCurrency();
  result = result @ XYZ;
  return result;
}

currency final = addTen(first);
// final = 11.00 XYZ

showValue(currency value) {
  currency result = value @ ABC;
  print(result);
}

showValue(final);
// 11.00 ABC
```

5.9 Pętle

W języku dostępna jest jedna pętla tworzona przy pomocy słowa kluczego "while".

```
int counter = 0;

while (counter < 5) {
  print(counter);
  print("\n");
  counter = counter + 1;
}
// 0
// 1
// 2
// 3
// 4
```

6 Błędy sygnalizowane w języku

Poniżej pokazano przykłady błędów na jakie można się napotkać przy pisaniu kodu w języku currex.

6.1 Błędy leksykalne

Przekroczenie dozwolonej długości identyfikatora

```
// IDENTIFIER_MAX_LENGTH = 10;

string too_long_identifier = "short value";

// IdentifierTooLongError: TOO LONG IDENTIFIER ERROR!
```

Przekroczenie dozwolonej długości stringa

```
// STRING_MAX_LENGTH = 10;

string too_long_stringr = "not so short value";

// StringTooLongError: TOO LONG STRING!
```

Przekroczenie zakresu liczby całkowitej

```
// MAXIMUM_INTIGER = 1000;

int first = 500;
int second = 950;
int sum = first + second;

// OverflowError: INTEGER IS TOO BIG!
```

Przekroczenie zakresu precyzji liczby zmiennoprzecinkowej

```
// MAXIMUM_PRECISION = 5;
// po przecinku dozwolone jest maksymalnie 5 cyfr

float smallFraction = 0.0000001;

// FloatingPointError: FLOAT NUMBER IS TOO BIG!
```

Nierozpoznany token

```
str^&ing unknown_token = "this is not a valid token";

// UnknownTokenError: str^&ing is not a known token

bool isTrue = (first_value > 0 & second_value < 10);

// UnknownTokenError: & is not a known error

bool isTrue = (first_value > 0 | second_value < 10);

// An error has occured in line 1 in column 5:
// UnknownTokenError: UNKNOWN TOKEN FOUND!
```

6.2 Błędy składniowe

Brak średnika w odpowiednim miejscu

```
string value = "word"

// MissingSemicolonError: MISSING SEMICOLON AT THE END OF THE LINE!
```

Użycie nieprawidłowego identyfikatora

```
string string = "word";

// InvalidIdentifierError: INVALID IDENTIFIER NAME!
```

Użycie wyrażenia zamiast identyfikatora lub wartości

```
bool thisIsFalse = 90 - 51;

// InvalidVariableTypeError: INVALID VARIABLE OF TYPE INTEGER!

string ExampleFunction() {}

currency value = ExampleFunction();

// InvalidVariableTypeError: INVALID VARIABLE OF TYPE STRING!
```

Brak zdefiniowanej funkcji main w programie

```
man() {
int a = 1;
int b = 3;
a = a + b;
}

// MainFunctionNotDefinedError: MAIN FUNCTION WAS NOT DEFINED!
```

Brak średnika na koniec rzędu tabeli w tabeli konwersji

```
    EUR USD PLN;
EUR 1 1.08 4.68
USD 0.92 1 4.31;
PLN 0.18 0.23 1;

// InvalidCurrencyRateError: INVALID CURRENCY RATE USD
```

Tabela konwersji mająca więcej walut w nagłówku tabeli niż rzędów

```
    EUR USD PLN CAD;
EUR 1 1.08 4.68
USD -0.92 1 4.31;
PLN 0.18 0.23 1;

// InvalidCurrencyTableError: HEADER CURRENCIES COUNT OF 4
DOES NOT MATCH ROW CURRENCIES COUNT OF 3!
```

6.3 Błędy semantyczne

Dzielenie przez zero

```
int zero = 0;
int one = 1;
int division = one / zero;

// ZeroDivisionError: UNHANDLED DIVISION BY ZERO!
```

Przypisanie zmiennej złego typu

```
int value = 0.5;

// InvalidVariableTypeError: INVALID VARIABLE OF TYPE FLOAT!
```

Niepoprawna liczba podanych argumentów funkcji

```
float sum(float a, float b) {}

sum(14.0, 3.3, 9.1);

// InvalidFunctionCallError: INVALID NUMBER OF ARGUMENTS FOR FUNCTION add EXPECTED: 2 BUT RECEIVED:
```

Ponowne zadeklarowanie zmiennej o tej samej nazwie

```
int a = 10;
int a = 50;

// VariableAlreadyExistsError: VARIABLE a HAS BEEN ALREADY DEFINED!
```

Użycie niezadeklarowanej zmiennej

```
int sum = a + 3;

// VariableDoesNotExistError: VARIABLE a DOES NOT EXIST IN ANY CONTEXT!
```

Użycie argumentu funkcji z typem innym niż zdefiniowany parametr

```
printSomething(string value) {
    print(value);
}

int a = 1;
printSomething(a);

// InvalidVariableTypeError: INVALID TYPE PROVIDED FOR PARAMETER value
```

Użycie liczby z minusem w tabeli konwersji

```
    EUR USD PLN;
EUR 1 1.08 4.68
USD -0.92 1 4.31;
PLN 0.18 0.23 1;

// NegationNotAllowedError: NEGATION IS NOT ALLOWED!
```

Tabela konwersji zawiera inny przelicznik niż 1 dla tej samej waluty

```
    EUR USD PLN;
EUR 1 1.08 4.68
USD 0.92 1000 4.31;
PLN 0.18 0.23 1;

// InvalidCurrencyRateError: CURRENCY RATE FOR CURRENCY USD IS
EQUAL TO 1000.0 AND NOT TO 1!
```

7 Implementacja

Do stworzenia projektu języka, stworzone zostały klasy oddelegowane do sprawdzenia poprawności wejściowego ciągu znaków w celu rozpoznania czy charakteryzuje on język. Currex jest językiem statycznym i silnie

typowanym, nie jest dozwolona pełna swoboda w redefiniowaniu zmiennych z jednego typu w inny.
Do implementacji zostały stworzone następujące elementy:

- Lekser
- Parser
- Interpreter

7.1 Token

Klasa tokenu ma za zadanie przechowywać elementy rozpoznane przez lekser. Tokeny zawierają swoją pozycję w wejściowym ciągu znaków oraz rodzaj tokenu, czyli to jaki element języka reprezentuje. Możliwe jest również aby przechowywał wartość, co jest wykorzystywane przy kilku rodzajach tokenów:

- INTEGER_VALUE - wartości całkowitoliczbowe
- FLOAT_VALUE - wartości zmiennoprzecinkowe
- STRING_VALUE - łańcuchy znaków
- IDENTIFIER - identyfikatory

Ostatni token typu `IDENTIFIER` przechowuje albo identyfikator, albo łańcuchy znaków.
Oprócz tego zdefiniowane są również tokeny dla poniższych elementów języka takich jak operatory:

- LEFT_BRACKET - '['
- RIGHT_BRACKET - ']'
- LEFT_CURLY_BRACKET - '{'
- RIGHT_CURLY_BRACKET - '}'
- LEFT_PARENTHESIS - '('
- RIGHT_PARENTHESIS - ')'
- DOT - '.'
- COMMA - ','
- SEMICOLON - ';'
- LESSER - '<'
- GREATER - '>'
- LESSER_OR_EQUAL - '<='
- GRATER_OR_EQUAL - '>='
- INEQUALITY - '!='
- EQUALITY - '=='
- EXCLAMATION = '!'
- AND - '&&'
- OR - '||'
- EQUALS - '='
- AT - '@'
- ARROW - '->'
- PLUS - '+'

- MINUS - '-'
- ASTERISK - '*'
- SLASH - '\'
- COMMENT - \\

Oraz słowa kluczowe:

- TRUE = 'true'
- FALSE = 'false'
- WHILE - 'while'
- IF - 'if'
- ELSE - 'else'
- RETURN - 'return'
- INTEGER - 'int'
- FLOAT - 'float;
- STRING - 'string'
- BOOL - 'bool'
- CURRENCY - 'currency'

Następnie tokeny te przechowywane są w obiektach klasy `Token` które dla pierwszego typu tokenu przechowują także konkretną wartość zaczytaną przez lekser.

7.2 Lekser

Lekser ma za zadanie generować tokeny leniwie na podstawie wejściowego ciągu znaków który otrzymuje. Rozpoznaje on znaki jeden po drugim ze źródła danych i generuje tokeny tylko wtedy, kiedy zostanie o to poproszony. W razie wykrycia komentarza, jest zwracany typ tokenu komentarza który jest ignorowany w dalszej części przetwarzania.
W przypadku kiedy jakiś token nie zostanie rozpoznany, zwracany jest token typu `UNKNOWN` i sygnalizowany jest błąd.

7.3 Parser

Zadaniem parsera jest przekształcić ciąg tokenów podawany przez lekser w drzewo obiektów, gdzie każdy obiekt stanowi inne wyrażenie w języku. Łączy on także tokeny typów `INTEGER\_VALUE` i `IDENTIFIER` w wyrażenia typu `currency`.
Przy wywołaniu metody `parse()`, zwracany zostaje obiekt typu `Program` który jest strukturą reprezentującą program zbudowany z wejściowego strumienia znaków. Wartości zmiennych są przechowywane w specjalnych obiektach o interfejsie `Primitive`, są to:

- IntPrimitive
- FloatPrimitive
- StringPrimitive
- BoolPrimitive
- CurrencyPrimitive

7.4 CurrencyPrimitive

Klasa ta reprezentuje zmienne typu `currency` w drzewie obiektów. Ma ona dwa pola klasy:

- `value` - nominał walutowy reprezentowany przez obiekty `BigDecimal`
- `name` - nazwa waluty będąca ciągiem znaków

Dla zmiennych tego typu dostępne są także dwie wbudowane metody:

- `getBalance()` - zwraca nominał przechowywany przez zmienną
- `getCurrency()` - zwraca typ waluty jaki reprezentuje zmienna

7.5 TableParser

Klasa ta ma za zadanie stworzyć drzewo obiektów reprezentujące plik konfiguracyjny zawierający tabelę konwersji walut. Na koniec każdego rzędu musi znaleźć się średnik. Przy parsowaniu sprawdzana jest od razu poprawność tej struktury, na przykład czy liczba walut w nagłówku zgadza się z liczbą rzędów.
Nagłówek tabeli reprezentuje kolumny tabeli, rzędy zawierają nazwę jednej z tych walut i przeliczniki dla każdej następnej. Nie jest możliwe zdefiniowanie konwersji jako liczby ujemnej, jest to sygnalizowane jako błąd. nie jest także możliwe zdefiniowanie przelicznika dla tej samej waluty jako innej niż 1.

7.6 ConversionTable

Klasa ta reprezentuje tabelę konwersji. W konstruktorze sprawdzane są przeliczniki i czy nie mają wartości niepoprawnych dla tych samych walut.

7.7 Interpreter

Interpreter ma za zadanie przejść przez drzewo obiektów i wykonać odpowiednie polecenia poprzez poprawną interpretację. Sprawdza on zgodność typów w instrukcjach w kolejnych węzłach drzewa obiektów. Realizowane jest to przy pomocy wzorca wizytatora.
Funkcje wbudowane w język są dodawane na samym początku do pola klasy przechowującego definicje funkcji. Przy pomocy klasy pomocniczej `ContextManager` zarządza ona dostępami do zmiennych w poszczególnych blokach kodu tak, aby zmienne z bloku wewnętrznego nie były dostępne dla instrukcji na zewnątrz niego.

7.8 PrinterVisitor

Jest to klasa pomocnicza, która prezentuje sparsowane drzewo obiektów w sposób czytelny dla użytkownika. Realizuje ona wzorzec wizytatora. Zostały stworzone dwie klasy, jedna dla drzewa obiektów instrukcji i druga dla drzewa obiektów tabeli konwersji z pliku wejściowego.

8 Testowanie

Implementacja została sprawdzona przy pomocy testów jednostkowych dla leksera, parsera, tokenu, tabeli konwersji i menadżera kontekstów. Sprawdzone zostało wyrzucanie wyjątków jak i wykrywanie poprawnych wartości. Testy zostały napisane z użyciem biblioteki jUnit 4.
Do przetestowania parsera zostały ponadto użyte testy integracyjne z lekserem.
Interpreter został przetestowany przy użyciu wielu testów ręcznych i testów jednostkowych.