Język do operacji na walutach

Bartłomiej Jurek

Ogólny opis projektu

Projekt zakłada stworzenie języka umożliwiającego podstawowe przetwarzanie zmiennych zawierających wartości liczbowe z jednostkami reprezentującymi waluty. Język będzie umożliwiał deklaratywne zdefiniowanie relacji pomiędzy walutami (przelicznika), która będzie wykorzystywana przy operacji np. dodawania wartości o dwóch różnych walutach.

Ogólny opis realizacji

Projekt zakłada istnienie następujących elementów:

- ✓ Wejściowego pliku z rozszerzeniem .txt zawierającego kod napisany w stworzonym języku
- ✓ Pliku wyjściowego z rozszerzeniem .cpp zawierającego zamieniony kod własny na kod języka C++.
- ✓ Bibliotek Wallet.h oraz CurrenciesLibrary.h napisanych w języku C++, kopiowanych przez kompilator do folderu, w którym zostaje tworzony plik wynikowy .cpp.

Szczegółowy opis języka

Typy danych

- int – liczba całkowita ze znakiem z zakresu <-2 147 483 648, 2 147 483 647>
- float – liczba zmiennopozycyjna z pojedynczą precyzją z zakresu <-2 147 483 648.2 147 483 648, 2 147 483 648. 2 147 483 648>
- każda utworzona waluta będzie oddzielnym typem. Nazwa waluty to 3 duże litery. Przykłady typów: USD, PLN, EUR, CHF.

Gramatyka języka

1. CurrencyName = GreatLetter GreatLetter GreatLetter
2. GreatLetter = (A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|W|X|Y|Z)
3. SmallLetter = (a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z)
4. Letter = SmallLetter | GreatLetter
5. SpecialSigns = (. | , | ; | : | ! | - | % | @ | ? | ‘ | ” | & | „|”)
6. IntValue = [Sign] (1|2|3|4|5|6|7|8|9|0) {Digit}
7. Sign = ‘-‘
8. FloatValue = IntValue “.” Digit {Digit}
9. Digit = (1|2|3|4|5|6|7|8|9|0)
10. VariableName = SmallLetter {Letter} (o ograniczonej długości, sekcja Realizacja)
11. VariableType = ‘int’ | ‘float’ | CurrencyName

12. LogicExpr = ‘(‘SimpleLogExpr’)’ | ‘(‘SimpleLogExpr OrExpr SimpleLogExpr {OrExpr SimpleLogExpr}’)’

13. SimpleLogExpr = TermLog | ‘(‘ TermLog AndExpr TermLog {AndExpr TermLog} ‘)’
14. TermLog = Expr | NotExpr ‘(‘ Expr ‘)’
15. NotExpr = ‘!’
16. OrExpr = ‘||’
17. AndExpr = ‘&&’

18. Expr = SimpExpr | SimpExpr {RelOp SimpExpr}
19. SimpExpr = Term {AddOp Term}
20. Term = Factor {MultOp Factor}
21. Factor = VariableName | IntValue | FloatValue | FunctionCall | ‘(‘Expr’)’
22. RelOp = ‘==’ | ‘>’ | ‘<’ | ‘>=’ | ‘<=’ | ‘<>’
23. AddOp = ‘+’ | ‘-‘
24. MultOp = ‘*’ | ‘/’

25. Comparable = IntValue | FloatValue | VariableName
26. FunctionCall = VariableName ‘(‘[Comparable] {‘,’ Comparable} ‘)’
27. Assignment = ([VariableType] VariableName) ”=” SimpExpr
28. Statement = Assignment | Attribute | FunctionCall |SetCourse | ReadValue | WriteValue
29. InstructionBlock = Statement ‘;’| IfStatement | ForExpr | WhileExpr

30. IfStatement = IfExpr {ElifExpr} [ElseExpr]
31. IfExpr = ‘if’ LogicExpr ‘{‘ {InstructionBlock | Comment} ‘}’
32. ElseExpr = ‘else’ ‘{‘ {InstructionBlock | Comment} ‘}’
33. ElifExpr = ‘elif’ LogiExpr ‘{‘ {InstructionBlock | Comment} ‘}’
34. Comment = ‘//’ {Digit | Letter |SpecialSigns | RelOp | AddOp | MultOp } ‘\n’

35. ForExpr = ‘for’ Assignment ‘to’ VariableName | IntValue ‘do’ ‘{‘{InstructionBlock | Comment} ‘}’
36. WhileExpr = ‘while’ LogicExpr ‘{‘ {IntructionBlock | Comment} ’}’

37. FunctionDef = ‘function’ VariableName ‘(‘ AttributeList ‘)’ ‘:’ VariableType | ‘void’ ‘{‘ {InstructionBlock | Comment} ‘}’
38. AttributeList = [Attribute] | Attribute {‘,’ Attribute}
39. Attribute = VariableType VariableName

40. SetCourse = ‘set’ CurrencyName ‘:’ CurrencyName ‘=’ FloatValue

41. ReadValue = ‘read’ VariableName
42. WriteValue = ‘write’ VariableName | ‘write’ ‘”’ String ‘”’
43. String = { Letter | FloatValue}

44. Program = {InstructionBlock | Comment | FunctionDef}

Reguły translacji

WEJŚCIE	WYJŚCIE	OPIS
`int zmienna ;`	`int zmienna;`	Tworzona jest zmienna typu int.
`float zmienna = 0.23 ;`	`float zmienna = 0.23f;`	Tworzona jest zmienna typu float oraz dopisywana do niej wartość po znaku =
`USD zmienna ;`	`Wallet zmienna(„USD”);`	Tworzony jest obiekt typu Money, przechowujący informację o walucie oraz ilość środków. Zmienna po utworzeniu jest zerowana.
`USD zmienna = 25 ;`	`Wallet zmienna(„USD”,25);`	Tak jak wyżej, jednak do zmiennej dopisana jest również wartość 25
`set EUR : PLN = 2.1 ;`	`CurrenciesLibrary::getInstance().setCourse(“EUR”, “PLN”, 2.1);`	Zostaje utworzona nowa waluta o nazwie ABC i kursie względem PLN 2.1 : 1, o ile waluta ABC jeszcze nie istnieje. Jeśli istnieje zostaje ustalony jedynie kurs waluty.
`write „WYNIK” ;`	`std::cout<<”WYNIK”<<std::endl;`	Na standardowe wyjście zostanie wyprowadzone wyrażenie w cudzysłowiach.
`write zmienna`	`std::cout<<zmienna;`	Na standardowym wyjściu zostanie wypisana wartość przechowywana w zmiennej.
`read zmienna`	`std::cin>>zmienna;`	Ze standardowego wejścia zostanie pobrana wartość i zapisana do zmiennej.
`while (warunek) {instrukcje}`	`while (warunek) {instrukcje;}`	Pętla kończąca się przy spełnieniu warunku w nawiasach.
`for int i = 0 to 10 do {instrukcje}`	`for(int i = 0; i <= 10; ++i) {instrukcje}`	Pętla for wykonująca się określoną ilość razy.
`if(warunek) {instrukcje}`	`if(warunek) {instrukcje}`	Instrukcja warunkowa.
`function f (int tmp) : int` *`{`* *`  f = 12 ;`* *`}`*	`int f(int tmp)` `{` `  int fRes;` `  fRes = 12;` `  return fRes;` `}`	Zostaje utworzona funkcja przyjmująca jeden argument oraz zwracająca zmienną typu int. W pseudojęzyku aby zwrócić wartość wystarczy przypisać wartość do nazwy funkcji.

`//to jest komentarz`		Komentarz zapisany w kodzie. Wykrywane przez Skaner. Komentarze nie zostają przepisane do kodu wyjściowego, są tracone.
`if(warunek) {instrukcje ;} elif(warunek) {instrukcje ;} else {instrukcje ;}`	`if(warunek) {instrukcje;} else if(warunek) {instrukcje;} else {instrukcje;}`	Złożona instrukcja warunkowa składająca się z bloku if, else if oraz else.

Realizacja – opis

Nazwa zmiennej została ograniczona do 20znaków, analogicznie nazwa funkcji. Wypisywany string może mieć długość maksymalnie 200znaków.

Zmienne nie mogą mieć nazw jak słowa kluczowe. Wszelkie nazwy zmiennych jak set, int, float oraz nazw walutowych są zabronione. Nazwa zmiennej jak i funkcji musi zaczynać się z małej litery.

Komentarze są odrzucane. Jedyne komentarze mogące wystąpić w kodzie wynikowym to te wynikające z wykrytych błędów przy wykrytej kompilacji.

Błędy możliwe są na poziomie leksykalnym, składniowym i semantycznym. Każdy błąd kończy się przerwaniem analizy. W takim przypadku tworzony plik wynikowy zostanie usunięty w całości, mimo iż mógłby zawierać już jakieś dane.

Zmienne zadeklarowane w blokach if, while, for jak i w utworzonych funkcjach zachowają status zmiennych lokalnych.

Waluty będą przedstawione jako obiekty klasy Wallet. Klasa ta zawierała wartość oraz nazwę waluty, która stanowi jej unikalne ID. Oprócz tego klasa zawiera przeciążone operatory porównywania przypisania oraz prostych operacji arytmetycznych.

Wynikowy kod C++ będzie całkowicie poprawny, gdyż poprawność kodu zostanie sprawdzona przez „kompilator” podczas konwersji z języka walutowego na C++.

Do folderu zawierającego kod wynikowy, pod koniec procesu kompilacji zostają skopiowane biblioteki, tak, aby uprościć do maksimum uruchomienie przekompilowanego kodu. Biblioteki te są dołączonego do kodu projektu.

Główne moduły projektu

- Compiler – klasa zawiera wyłącznie jedną metodę publiczną compile, przyjmującą jako parametr wejściowy ścieżkę do pliku źródłowego zapisaną w zmiennej typu String. Jej zadaniem jest przygotowanie wszystkich pozostałych modułów, przygotowanie ścieżki do pliku wyjściowego oraz skopiowanie dołączonych bibliotek to tegoż folderu. Kompilację rozpoczyna polecenie generator.generate(parser.program()); , która zleca parserowi dokładne sprawdzenie pliku wejściowego oraz stworzenie drzewa rozbioru, a wynik jego pracy przekazuje do generatora tworzącego kod z rozszerzeniem .cpp.

- Generator – klasa posiadająca jedną metodę publiczną generate(), która jako parametr przyjmuje drzewo rozbioru. W zależności od bloku instrukcji pobranego z listy reprezentującej program, wywołuje konkretne metody prywatne zajmujące się generowaniem poszczególnych instrukcji. Za jakość generowanego kodu w dużej mierze odpowiada zmienna incission, która przechowuje informację o wcięciach jakie należy zrobić w danym miejscu. Dzięki temu wygenerowany kod jest przejrzysty. Generator pełni jeszcze jedną bardzo ważną funkcję, a mianowicie podmienia zmienne o nazwie równej z nazwą funkcji (zmienne te służą do zwracania wyniku przez funkcję) dopisując do niej końcówkę „Res". Umożliwia to rekurencyjne wywoływanie się funkcji, ponieważ nazwa zmiennej różni się od nazwy funkcji, której wynik prezentuje.
- Parser – Zawiera w sobie obiekt MyScanner. Pobiera od niego kolejne tokeny, sprawdzając poprawność synktatyczną. Jest to bardzo prosty w zrozumieniu działania parser schodzący. Posługując się klasą pomocniczą VariablesAndFunctions sprawdza, czy użytkownik nie próbuje dwukrotnie zadeklarować funkcji o tej samej nazwie, analogicznie sprawdzając zmienne. Sprawdza również lokalność zmiennych, oraz nazwę i listę argumentów wywołanej funkcji porównując z funkcjami zadeklarowanymi we wcześniejszych, już sprawdzonych fragmentach kodu wejściowego. Błędy wykryte przez parser w kodzie wynikowym oznaczone są przez symbol [P] na początku błędu. Zawsze zapamiętanym elementem jest pierwszy symbol z nowej instrukcji.
- MyScanner – klasa korzysta z obiektu typu Reader i reprezentuje Lekser. Pobiera od niego kolejne znaki i łączy je w symbole. Rozpoznaje ich typ tworząc obiekt typu Symbol. Tak przygotowany token przekazuje do Parsera. Błędy semantyczne wykrywane przez MyScanner oznaczone są symbolem [S] w kodzie wynikowym.
- Reader – Odczytuje kolejny znak z pliku wejściowego i przekazuje go do MyScanner. Korzysta również z obiektu będącego reprezentantem wzorca projektowego singleton, ustalając w nim odpowiedni nr wiersza i linii po odczytaniu każdego kolejnego znaku. Umożliwia to wypisywanie konkretnych błędów wraz z numerem wiersza i linii zarówno Parser jak i MyScanner.
- Writer – operuje na pliku wyjściowym. Jego dwie główne metody publiczne to write(String, Integer), która wypisuje otrzymanego w argumencie stringa uprzedzając go odpowiednią ilością wcięć zdefiniowaną przez argument typu Integer, oraz error(String), której zadaniem jest wypisanie do pliku wynikowego treści błędu otrzymanego jako argument oraz bezpieczne zamknięcie pliku wynikowego przed zakończeniem procesu kompilacji, tak, aby zachowały się w nim wszystkie wprowadzone zmiany. Writer został napisany zgodnie ze wzorcem projektowym singleton, tak aby umożliwić pisanie do pliku wyjściowego zarówno przez Parser jak i Lekser i Generator.

Klasy pomocnicze

- CursorPosition – klasa zawierająca obecną pozycję kursora w pliku wejściowym. Potrzebne do wypisywania konkretnych i szczegółowych komunikatów o błędach.
- KeyWords – klasa zawiera wszystkie obostrzenia programu, słowa kluczowe oraz operatory. Pomaga klasie MyScanner w decyzji z jakim atomem ma styczność. W sytuacji gdy MyScanner napotka atom będący słowem zaczynającym się od małej litery, nie może sam stwierdzić czy dane słowo jest identyfikatorem (IDENT) czy też którymś ze słów kluczowych. W rozstrzygnięciu tego problemu pomaga właśnie klasa KeyWords, a konkretniej metoda publiczna

getByWord(String), która zwraca typ konkretnego słowa (IDENT lub rozpoznana nazwa słowa kluczowego).

- Symbol – główna klasa służąca komunikacji modułu Parser oraz MyScanner (Lekser), jak i podstawowy obiekt tworzący drzewo rozbioru budowane przez Parser. Zawiera rodzaj słowa kluczowego oraz jego wartość przechowywaną jako atrybut prywatny typu String.
- VariablesAndFunctions – rozbudowana klasa pomagająca obiektowi Parser w sprawdzaniu lokalności zmiennych, sprawdzaniu ich widoczności, decydowaniu czy funkcja o zadanej nazwie i liście parametrów istnieje, oraz zajętości nazw zmiennych i funkcji. Zawiera w sobie 3 kolekcje:
    - variables – jest to lista kolekcji HashMap. Każda HashMapa opisuje zmienne należące do jednej klasy. Zawiera w sobie obiekty typu HashMap, z których każdy opisuje konkretny poziom zagnieżdżenia zmiennej. Każdy poziom zagnieżdżenia reprezentowany jest przez obiekt klasy HashMap, której kluczem jest nazwa zmiennej, a wartością jej typ. W ten sposób możemy sprawdzać listy argumentów oraz przypisania.
    - functions - kolekcja zadeklarowanych w kodzie źródłowym funkcji, reprezentowana przez obiekt klasy HashMap, gdzie kluczem jest nazwa funkcji natomiast wartością lista jej argumentów. Każdy argument reprezentowany przez wartość typu String, która przechowuje jej typ. Przykładowo zadeklarowanie funkcji gdzie parametrem jest obiekt walutowy PLN nie umożliwia jej wywołania z parametrem typu EUR. (Możliwe zmiany)
    - currenciesTypes – lista wszystkich zadeklarowanych jak i wbudowanych walut. Lista zawiera wyłącznie nazwę waluty zapisaną w zmiennej typu String. Lista ta jest niezbędna, aby sprawdzić czy użyta w kodzie waluta istnieje.

Drzewo rozbioru

Drzewo rozbioru reprezentowane jest przez obiekt klasy Program. Wszystkie używane klasy znajdują się w folderze ProgramStructures. Użyte klasy:

- Program – główna klasa drzewa rozbioru. Zawiera listę funkcji (obiektów klasy Function). Jedną z funkcji w liście jest main, niedeklarowany wprost przez użytkownika w kodzie źródłowym. Pozostałe funkcje to funkcje stworzone przez użytkownika za pomocą słowa kluczowego function.
- Function – klasa ta zawiera nazwę funkcji zapisaną w zmiennej typu String, zwracany typ przechowywany jako obiekt klasy Symbol, listę parametrów (parametr reprezentowany jest jako para Symbolu który jest jej typem, oraz Stringa, który jest nazwą zmiennej), oraz listę obiektów dziedziczących po klasie Instruction, które reprezentują instrukcje w ciele funkcji idąc od góry do dołu kodu źródłowego.
- Instruction – definiuje typ enum, który określa rodzaj instrukcji. Instrukcją może być:
    - Expression
    - ForLoop
    - WhileLoop
    - IfExpression
    - SetCourse

    Wszystkie z wymienionych klas dziedziczą po klasie Instruction, co umożliwia ich łatwe przechowywanie w jednej liście instrukcji niezależnie od ich typu.

- Expression – Najprostsza instrukcja składająca się z listy symboli. Wyrażeniem takim jest przypisywanie wartości (assingment) oraz tworzenie nowej zmiennej jak i wywoływanie funkcji.
- ForLoop – Instrukcja zawiera listę Symboli (reprezentującą inicjalizację wartości głównej pętli), Symbol reprezentujący wartość końcową, do której zmierza pętla oraz listę instrukcji reprezentującą ciało pętli.
- WhileLoop – Instrukcja zawiera obiekt typu LogicExpression oraz lilstę instrukcji. Obiekt typu LogicExpression reprezentuje warunek końcowy pętli while, natomiast lista instrukcji ciało pętli.
- IfExpression – Najbardziej rozbudowana instrukcja, zawiera warunek (obiekt typu LogicExpression), listę instrukcji reprezentującą ciało instrukcji warunkowej, listę instrukcji reprezentującą ciało bloku else (jeżeli ta lista jest pusta, nie zostaje wygenerowany blok else), oraz Listę zawierającą Pary <LogicExpression, lista instrukcji>. Lista ta reprezentuje wszystkie bloki elif, których może być nieskończenie wiele. Każda para z tej listy zawiera warunek oraz listę instrukcji, które powinny zostać wykonane po spełnieniu warunku.
- SetCourse – zawiera 3 symbole, pierwszym jest waluta, którą tworzymy lub której kurs zmieniamy, drugim waluta, do której porównujemy kurs aktualnie zmienianej waluty, natomiast trzecim kurs, który należy ustalić.
- LogicExpression – Zawiera listę symboli ułożoną w kolejności, w której powinny zostać przetworzone przez generator kodu C++.

Sposób uruchomienia

Kompilator uruchamiamy jak standardowy program z rozszerzeniem .jar podając w konsoli jego nazwę, a jako parametr ścieżkę do pliku z naszym kodem napisanym w języku walutowym. Jako wynik otrzymamy w tym samym folderze pliki .cpp z przekonwertowanym kodem oraz biblioteki niezbędne do uruchomienia i kompilacji otrzymanego kodu.

Krótko o sposobie działania

Obiektami walutowymi w kodzie są obiekty klasy Wallet. Klasa ta zawiera wartość oraz nazwę waluty, która jest jej unikalnym ID. Oprócz tego klasa zawiera przeciążone operatory porównania, przypisania oraz prostych operacji arytmetycznych. Wszystkie kursy walutowe przechowywane są w HashMapie w klasie CurrenciesLibrary, gdzie kluczem jest ID waluty natomiast wartością jest stosunek do waluty podstawowej, za którą zostało przyjęte PLN. Oznacza to że waluta PLN ma przypisaną wartość 1 natomiast EUR ok. 0.232.

Testowanie

Do projektu zostały dołączone 3 foldery:

- negLekser – przykłady testowe opisujące wszystkie negatywne wyniki działania leksera
- negParser – przykłady testowe opisujące wszystkie negatywne wyniki działania parsera
- pos – kilka pozytywnych przykładów

Przykład uznajemy za pozytywny, jeżeli:

- zostaje utworzony plik wynikowy z rozszerzeniem .cpp
- kod wynikowy udaje się skompilować za pomocą dowolnego kompilatora bez błędów

Każdy test zawiera:

- krótki opis testu
- przewidywany wynik
- kod napisany w języku walutowym, który ma podlegać konwersji

Przykładowy program (deklaracje, obliczenia, strumień wejścia, strumień wyjścia)

Wejście

```
//Ten program w prosty sposób przelicza wartości podane przez użytkownika
//Główne waluty takie jak PLN, EUR, USD mają na start przypisane kursy
set ABC : PLN = 2 ;
set EUR : PLN = 4.2 ;
set USD : PLN = 3.7 ;
PLN kontoPln = 157.1 ;
EUR kontoEur ;
EUR wPortfelu ;
//Nastepuje wczytanie wartości ze standardowego wejscia
read kontoEur ;
read wPortfelu ;
//Dodajemy wszystkie wartości
ABC suma = kontoPln + kontoEur + wPortfelu ;
write suma ;
```

Wyjście

```
#include <iostream>
#include "Wallet.h"
int main( ) {
	CurrenciesLibrary::getInstance().setCourse("ABC", "PLN", 2);
	CurrenciesLibrary::getInstance().setCourse("EUR", "PLN", 4.2);
	CurrenciesLibrary::getInstance().setCourse("USD", "PLN", 3.7);
	Wallet kontoPln("PLN", 157.1);
	Wallet kontoEur("EUR", 0);
	Wallet wPortfelu("EUR", 0);
	std::cin >> kontoEur ;
	std::cin >> wPortfelu ;
	Wallet suma("ABC", kontoPln + kontoEur + wPortfelu);
	std::cout << suma << std::endl;
	return 0;
}
```

Przykładowy program (instrukcje warunkowe, pętla while)

Wejście

```
PLN waluta = 10 ;
while ( ( waluta < 20 ) || ( waluta > 15 ) ) {
	if ( waluta – 2 < 20 ) {
	waluta = waluta + 4 ;
	}
	elif ( waluta + 5 > 15 ) {
	waluta = waluta / 2 ;
	}
}
```

Wyjście

```
#include <iostream>
#include "Wallet.h"

int main( ) {
	Wallet waluta("PLN", 10);
	while( ( waluta < 20 ) || ( waluta > 15 ) ) {
		if ( waluta - 2 < 20 ) {
			waluta = waluta + 4;
		}
		else if( waluta + 5 > 15 ) {
			waluta = waluta / 2;
		}
	}
	return 0;
}
```

Przykładowy program (pętla for)

Wejście

```
//kupowanie biletow dla 10 osob
PLN portfel = 150 ;
EUR bilet = 1.5 ;
for int i = 1 to 10 do {
portfel = portfel – bilet ;
}
write portfel ;
```

Wyjście

```
#include <iostream>
#include "Wallet.h"

int main( ) {
	Wallet portfel("PLN", 150);
	Wallet bilet("EUR", 1.5);
	for ( int i= 1 ; i <= 10; ++i ) {
		portfel = portfel - bilet;
	}
	std::cout << portfel << std::endl;
	return 0;
}
```

Przykładowy program (tworzenie oraz wywoływanie funkcji)

Wejście

```
//wyliczanie silni
function silnia ( int tmp ) : int {
	if ( tmp == 0 ) {
		silnia = 1 ;
	}
	else {
		int n = tmp - 1 ;
		silnia = silnia ( n ) * tmp ;
	}
}
silnia ( 5 ) ;
```

Wyjście

```
#include <iostream>
#include "Wallet.h"

int silnia (int tmp) {
	int silniaRes;
	if ( tmp == 0 ) {
		silniaRes = 1;
	}
	else {
		int n = tmp - 1;
		silniaRes = silnia ( n ) * tmp;
	}
	return silniaRes;
}
```

```
int main( ) {
	silnia ( 5 );
	return 0;
}
```

Przykład z błędem w kodzie

Wejście

```
PLN gotowka = 120
EUR pozyczka = 24
for 5 to 23 do {
gotowka = gotowka - 1
pozyczka = pozyczka + 1
```

Wyjście

```
//[P] Error in line: 3 at char: 6. int variable initialization required after for word.
```