INTERPRETER JĘZYKA Z WBUDOWANYM TYPEM WALUTOWYM – DOKUMENTACJA KOŃCOWA

Krótki opis projektu

Celem projekt było stworzenie prostego języka programowania i jego interpretera. Główną rozwijaną cechą tego języka jest wbudowany typ walutowy. Oprócz tego język powinien udostępniać podstawowe możliwości typowego języka programowania.

Opis funkcjonalności

- Instrukcja warunkowe – if else
- Pętla – while
- Operatory arytmetyczne +, -, *, /,
- Operatory relacyjne <, >, <=, >=, !=, ==
- Operatory logiczne ! , and, or
- Typy wbudowane: num – liczba stałoprzecinkowa oraz cur – liczba stałoprzecinkowa walutowa
- Definiowanie funkcji przyjmujących argumenty
- Zmienne lokalne
- Możliwość rekursywnego wywoływania funkcji
- Komentarze #, które są odrzucane po przetworzeniu przez Lexer
- Funkcje wbudowane:
    - print() – funkcja przyjmująca nieokreśloną liczbę argumentów do wypisania na ekranie
    - scanNum() – funkcja przyjmująca standardowe wejście, zwraca wpisaną liczbę stało przecinkową
    - scanCur() – funkcja przyjmująca standardowe wejście, zwraca pojedynczą jednostkę wpisanej waluty
- Stała tekstowa – wspierana tylko w funkcji print()

Biblioteka standardowa

- print() – funkcja pobierająca kolekcję argumentów i wypisująca je na wyjściu
- scanNum() – funkcja pozwalająca wczytać znaki ze standardowego wejścia aż do zatwierdzenia, zwraca wartość typu num zgodną z wprowadzoną wartością
- scanCur() - funkcja pozwalająca wczytać znaki ze standardowego wejścia aż do zatwierdzenia, wprowadzone znaki powinny być identyfikatorem waluty z pliku konfiguracyjnego, zwraca pojedynczą jednostkę tej waluty
- funkcje scanNum() i scanCur() w przypadku 3 krotnego wprowadzenia błędnej wartości zgłaszają błąd podczas wykonania.

Gramatyka

```
program         = functionDef, {functionDef};
funcionDef      = typeFun, identifier, "(", parameters, ")",
                  statemantsBlock;
parameters      = [parameter, {",", parameter}];
parameter       = type, identifier;
statementsBlock= "{", {statement}, "}";
statement       = ifState           |
                  whileState        |
                  returnState       |
                  funCallOrAssign   |
                  initState;

ifState         = "if", "(", condition, ")", statementsBlock,
                  ["else", statementsBlock];
whileState      = "while", "(", condition, ")",
                  statementsBlock;
returnState     = "return", [expression], ";";
funCallOrAssign= identifier, (funCallState |
                              assignState );
funCallState    = "(", arguments, ")", ";" ;
assignState     = "=", expression, ";" ;
initState       = type, identifier, "=", expression, ";";
arguments       = [ argument, {",", argument}];
argument        = expression   |
                  stringLiteral;

condition       = logicFactor, {orOpr, logicFactor};
logicFactor     = relation, {andOpr, relation};
relation        = [negOpr], expression, [relOpr, expression];
expression      = factor, {addOpr, factor};
factor          = term, {multiOpr, term};
term            = [minOpr], currencyConv;
```

```
currencyConv   = (literal          |
                  identOrFuncall    |
                  parenthesis) [currentPart];
identOrFuncall = identifier, ["(", arguments, ")"];
parenthesis    = "(", condition, ")";
```

Warstwa leksyki

```
identifier     = letter, { letter | digit};
literal        = decimalNumber;
currencyPart   = identifier   |
                  "@", identifier;
decimalNumber  = number, [".", {digit}];
number         = "0" | (notZeroDigit, {digit});
digit          = "0" | notZeroDigit;
notZeroDigit   = "1" .. "9";
stringLiteral  = """, {allChars}, """;
letter         = ? isLetter ?
allChars       = ? all visible characters ?;
typeFun        = type | "void";
type           = "cur" | "num";
negOpr         = "!"
minOpr         = "-"
addOpr         = "+" | "-";
multiOpr       = "*" | "/";
relOpr         = "==" | "!=" | "<=" | ">=" | "<" | ">"
andOpr         = "and";
orOpr          = "or";
```

Przykłady kodu – podstawy

Zmienna walutowa - cur

```
cur salary = 2500 PLN;
cur payments = 300 USD;
```

Zmienna stało przecinkowa – num

```
num percent = 0.02;
num vat = 0.37;
```

Określenie waluty

```
cur a = 2500 PLN;
cur b = 300 USD;
# zmienna c jest w Dolarach Kanadyjskich
cur c = a CAD;
# zmienna d jest w Dolarach
cur d = c @ b;
```

Porównywanie walut

```
cur a = 300 PLN;
cur b = 300 USD;
print((a >= b));
# wyświetli 0, czyli fałsz
```

Porównywanie wartości liczbowych

```
num a = 300;
num b = 300;
print((a >= b));
# wyświetli 1, czyli true
```

Rzutowanie na walutę i walutę zmiennej

```
cur a = 300 PLN;
cur b = 300 USD;
cur c = 200 @ b;
cur d = (a + b) CAD;
# zmiennej c zostanie przypisane 200 USD, a d wartość wyrażenia w CAD
```

Operacje warunkowe

```
cur a = 300 PLN;
cur b = 300 USD;
if(a < b) {
	num c = 0.02;
	num d = 0.37;
	if(c != d) {
		print("rozne");
	}
	else {
		print("rowne:", c, d);
	}
}
```

Pętla

```
cur a = 300 PLN;
num b = 4.5;
while(b > 0) {
	a = a + a * 0.2;
	b = b – 0.5;
	print(b, "\n");
}
```

Definicja funkcji

```
cur currencyFun(cur a, num b){
	if(b < 2) {
		return (a * b) @ a;
	}
	if(b > 3) {
		return (a * b) CAD;
	}
	else{
			return (a * b) USD;}
}
num numericFun(cur a, cur b, num c){
	if(a > b){
		return 0;
	}
	if(a < b){
		return 1;
	}
	else{
		return c;
	}
}
void main(){
	cur c = currencyFun (20 PLN, 2);
    #print(c);
	num d = numericFun (33.14 USD, 30 PLN, 5);
	print("Wartosc c: ", c, " wartosc d: ", d);
	# wypisze Wartosc c: 40 wartosc d: 0
}
```

Rekursja

```
num pow(num l, num p){
	if (p == 0){
		return 1;
	}
	return l * pow(l, p - 1);
}
void main(){
	# Wypisze Potega 2 do 3: 8
	print("Potega 2 do 3: ", pow(2, 3));}
```

Gramatyka piliku konfiguracyjnego

```
configfile     =    headerCurrency, {currencyLine};
headerCurrency =    "@", currency, {currency}, ";";
currencyLine   =    currency, {decimalNumber}, ";"
currency       =    identifier;
decimalNumber  =    number, [".", {digit}];
```

Przykładowy plik konfiguracyjny

```
@   USD   CAD  PLN;
USD 1       1.2595  4.2793;
CAD 0.7939  1       3.3967;
PLN 0.2337  0.2944  1     ;
```

Plik konfiguracyjny zbudowany jest w następujący sposób:

- Pierwszy wiersz rozpoczyna się znakiem małpy @ po którym wypisane są wszystkie identyfikatory walut używanych w programie.
- Każdy kolejny rząd rozpoczyna się identyfikatorem waluty, z której przeliczamy. Kolejne liczby oddzielone białymi znakami określają jaki jest przelicznik do waluty docelowej wyróżnionej na szczycie kolumny.
- Każdy wiersz jest zakończony znakiem średnika ;
- Przykładowo z USD do CAD przelicznik wynosi 1.2595

Założenia projektowe

Założenia wstępne

- Wymuszanie jawnej inicjalizacji zmiennych.
- Argumenty przekazywane przez kopię - wartość.
- Funkcje mogą zwracać tylko jeden typ lub nie zwracać nic.
- Funkcje z typem num lub cur musza zwracać wartość.
- Silne i statyczne typowanie.
- Zmienne są mutowalne.
- Program musi składać się z przynajmniej jednej funkcji typu void bez argumentów będącą główną funkcją.
- Nie występują tutaj zmienne logiczne, ale w przypadku operacji logicznych oraz warunków, wartość 0 odpowiada logicznemu fałszu, a wartości różne od 0 są równoważne z prawdą. Prawda jest równoważna z wartością 1.

Wymagania funkcjonalne

- Wczytywanie i przetworzenie pliku źródłowego na sekwencję tokenów – lekser.
- Przetwarzanie sekwencji tokenów na odpowiednie byty semantyczne, na których podstawie zostaną stworzone obiekty odpowiadające instrukcjom.

- Na każdym etapie interpretacji w przypadku wystąpienia błędu lub nieprawidłowości w źródle, pliku konfiguracyjnym, należy obsłużyć błąd i wyświetlić zrozumiały komunikat użytkownikowi.
- W przypadku braku pliku konfiguracyjnego należy wyświetlić informację o jego braku we wskazanej lokalizacji oraz podać przykładową jego zawartość.
- Priorytety operatorów powinny być zachowane i zgodne z dokumentacją.

Wymagania niefunkcjonalne

- Interpreter powinien obsłużyć wszystkie błędy związane z interpretacją.
- Program powinien uruchamiać się na urządzeniach z systemem Linux.
- Do prawidłowego działania programu wymagany jest kompilator gcc.
- W przypadku wystąpienia nieprawidłowości, komunikat przedstawiony użytkownikowi powinien być zrozumiały, wskazywać miejsce wystąpienia w przypadku analizy lekserem i parserem, błędy powinny wskazywać na przyczynę i miejsce, błędy podczas wykonania zawierają tylko informacje dotyczącą jaki błąd wystąpił.
- Analizator leksykalny nie powinien przerywać analizy źródła w przypadku wystąpienia nieprawidłowego/niezidentyfikowanego tokenu, w momencie w którym możliwe jest bezpieczne kontunuowanie analizy.

Wyróżnione słowa kluczowe i znaki – tokeny:

void	**cur**	**num**	**if**	**else**
while	**return**	**(**	**)**	**{**
}	**+**	**-**	*****	**/**
>=	**<=**	**>**	**<**	**!=**
==	**!**	**and**	**or**	**=**
@	**,**	**;**	**#**	****

* Oprócz nich występują tokeny dotyczące literałów, liczb stało przecinkowych, identyfikatorów, komentarzy oraz stałych tekstowych. Wszystkie z nich zostały opisane w gramatyce.

Dostępne operatory

- Arytmetyczne
  - Dodawanie +
  - Odejmowanie -
  - Mnożenie *
  - Dzielenie /
  - Unarny minus -
- Relacyjne
  - Większe lub równe >=
  - Większe >
  - Mniejsze lub równe <=
  - Mniejsze <
  - Różne !=
  - Równe ==
- Logiczne
  - Negacja !
  - Koniunkcja and
  - Alternatywa or
- Przypisania =
- Odniesienia się do waluty zmiennej @

Priorytety operatorów/operacji

Największy priorytet w tym języku mają nawiasy "()" następnie jest operacja rzutowania na walutę oraz odniesienia się do waluty zmiennej. Kolejny jest unarny minus zmieniający wartość za nim na przeciwną. Następne są operatory arytmetyczne mnożenia i dzielenia, po nich dodawanie i odejmowanie. Logiczna negacja. Następnie są wszystkie operatory relacyjne, a po nich koniunkcja logiczna, następna jest logiczna alternatywa i na końcu operator przypisania.

Rzutowanie

Polega na określeniu waluty wyrażenia/literału po lewej stronie rzutowania. Występują dwa sposoby rzutowania. Pierwszy domyślny polega na podaniu identyfikatora waluty zaraz po. Jest to wyrażenie walutowe. Drugim sposobem jest odniesienie się do rodzaju waluty danej zmiennej cur, z użyciem znaku "@". W celu określenia waluty końcowej całego wyrażenia należy umieścić je w nawiasach okrągłych.

```
cur nazwa_zmiennej = wartość/wyrażenie identyfikator_waluty
cur nazwa_zmiennej2 = wartość/wyrażenie @  nazwa_zmiennej
```

Samo przeliczenie polega na przemnożeniu wartości wejściowej przez odpowiedni przelicznik z pliku konfiguracyjnego odpowiadający konwersji z jednej waluty na drugą.

W przypadku wystąpienia wartości stałoprzecinkowej przeliczana jest ona jeden do jednego na wybraną walutę.

Przykład rzutowania:

```
cur d = (a + b) CAD;
cur c = 200 @ d;
# obie wartości będą w Dolarach kanadyjskich
```

Operacje arytmetyczne - dopuszczalność

Ze względu na dziedzinę języka pewne operacje nie mają uzasadnienia, dlatego nie wszystkie operacje będą dopuszczane.

Operacje niedopuszczalne:

- mnożenie dwóch wartości walutowych [cur * cur]
- dodawanie i odejmowanie wartości walutowej z wartością stało przecinkową w obie strony [cur – num] [cur + num] i [num-cur] [num+cur]
- dzielenie wartości stało przecinkowej przez walutę [num/cur]
- operacje relacji na rożnych typach [num opr cur] i [cur opr num]

* Gdzie opr odpowiada operatorą relacji ==, != , <=, < , >=, >

Operacje dopuszczalne:

- dzielenie dwóch walut, wtedy wynik jest wartością stało przecinkową [cur/cur]
- mnożenie waluty i wartości stało przecinkowej [cur*num]
- dzielenie waluty przez wartość stało przecinkową [cur/num]
- mnożenie i dzielenie dwóch wartości stało przecinkowych [num*num][num/num]
- operacje relacji na tych samych typach [num opr num] i [cur opr cur]

Sposób ewaluacji wyrażeń walutowych

W trakcie przetwarzania wyrażeń zgodnie z priorytetami i operacjami dopuszczalnymi, występują przeliczenia w celu przeprowadzenia dalszych obliczeń. Przeliczenia następują zgodnie z kolejnością przeprowadzania operacji, a wartości po prawej stronie operatora są przeliczane na te po lewej stronie. Można przyjąć, że składnik walutowy najbardziej po lewej stronie określa rodzaj waluty końcowego wyniku. By to zmienić należy skorzystać z nawiasowania i rzutowania na inną walutę.

Dla przykładowego wyrażenia:

```
cur a = 200 PLN + 30 USD * 3 – 25 CAD / 30
```

Najpierw wykonają się mnożenia i dzielenia. Następnie wynik 30 USD * 3 jest przeliczany na PLN i sumowany. Kolejnym krokiem jest odjęcie od sumy wyniku 25 CAD / 30. Na końcu otrzymujemy wynik w PLN.

Znak ucieczki w napisach

W programie dopuszczane jest użycie literałów znakowych ograniczonych podwójnymi apostrofami: ". Wewnątrz nich można stosować znak ucieczki – ukośnik wsteczny: \ w celu zapisania znaków tam nie dopuszczanych, takich jak chociażby znak ".

Dopuszczalne znaki ucieczki:

- \” – znak ”
- \\ - znak \
- \n – znak nowej linii
- \r – powrót karetki
- \t – znak tabulacji
- \v – wertykalny znak tabulacji

Informacje techniczne

Sposób uruchomienia/użytkowania:

Program w miejscu swojej lokalizacji oczekuje pliku konfiguracyjnego "currency_config.cfg".

Interpreter oczekuje minimum jednego argumentu w trakcie wywołania z konsoli. W przypadku nie podania nieprawidłowej ilości argumentów interpreter wyświetla odpowiedni komunikat opisujący argumenty.

```
./interpreter plik_wejściowy [nazwa_funkcji][plik_konfiguracyjny]
```

Poszczególne argumenty:

- plik_wejściowy – plik z programem w języku walutowym
- nazwa_funkcji – nazwa funkcji wykonywanej na początku programu (parametr opcjonalny – domyślna wartość: main)
- plik_konfiguracyjny – ścieżka do pliku z przelicznikami walutowymi(parametr opcjonalny – domyślna ścieżka to: currency_config.cfg w miejscu znajdowania się pliku wykonywalnego)

Domyślnym strumieniem wejścia do programu jest strumień plikowy. W testowaniu stosowany jest również strumień znakowy.

Środowisko uruchomieniowe

Docelowym środowiskiem uruchomieniowym są urządzenia z systemem Linux. Interpreter wykonany jest w języku C++ i kompilowany jest z użyciem kompilatora gcc. W procesie budowy do przyśpieszenia i zautomatyzowania wykorzystany został CMake.

Obsługa błędów

Obsług błędów w interpreterze ma przede wszystkim zadbać o prawidłowy przebieg procesu interpretacji źródła, powiadomić programistę, gdy zostanie wykryty błąd. Komunikaty są podzielone na trzy typy Error, Warning oraz RuntimeError, które odpowiadają kolejno błędowi krytycznemu podczas analizy, ostrzeżeniu, które nie zakłóciło procesu analizy oraz krytycznemu błędowi podczas wykonania interpretacji przeanalizowanego programu. Błędy w których możliwe jest wskazanie dokładnej lokalizacji, czyli podczas analizy leksykalnej oraz parsowania, wskazują błąd w pliku źródłowym. Błędy wykryte podczas analizy semantycznej wskazują przyczynę i zestaw wartości mających na celu nakierowanie na lokalizacje błędu.

Błędy podczas wykonania wskazują na przyczynę przerwania.

W przypadku źródła tekstowego pozycja to: numer wiersza, numer kolumny, pozycja w strumieniu oraz długość tokenu, gdzie występuje błąd.

W zależności od miejsca wystąpienia błędu interpreter może lub nie kontynuować interpretację.

Plik konfiguracyjny i plik źródłowy.

- W przypadku braku pliku konfiguracyjnego "currency_config.cfg" lub gdy plik jest pusty. Program zakomunikuje brak konfiguracji walutowej, ale nie przerwie na tym etapie, ponieważ możliwe jest napisanie kodu nie korzystającego z zmiennych walutowych.
- W przypadku braku podanego pliku wejściowego interpreter zakomunikuje jego brak i zakończy działanie.
- W przypadku podania nazwy funkcji niewystępującej w skrypcie interpreter wypisze komunikat o błędzie i zakończy działanie.

Analizator leksykalny

- W przypadku wystąpienia niezidentyfikowanego tokena analizator leksykalny przerywa działanie i wypisuje błąd.
- W przypadku wystąpienia słów kluczowych błędnie napisanych – z użyciem dużych liter, wyświetlany jest komunikat ostrzegawczy
- Nazwy identyfikatorów będące tożsame ze słowami kluczowymi są uznawane za błędne i zgłaszane użytkownikowi.
- Słowa kluczowe z dużej litery są uznawane za nieprawidłowe.
- Zgłasza inne błędy związane z nieprawidłową budową tokenów – powiązane z parametryzowanymi wielkościami określającymi długość maksymalną ciągów znakowych, identyfikatorów zmiennych i walutowych, maksymalnej wielkości liczb stało przecinkowych.
- W przypadku wystąpienia, ETX lub EOF lekser na prośbę o kolejny token powinien zwracać nadal token końca źródła.

Analizator składniowy

- Wszelkie błędy związane z nieprawidłową składnią analizowanego pliku źródłowego.

Analizator semantyczny

- Wykrywa próbę wykonania operacji niedozwolonej, opisanej we wcześniejszej części dokumentacji i powiadamia o niej użytkownika.
- Użycie niezdefiniowanej funkcji, powinno zgłosić wyjątek i zakończyć działanie.
- Przy próbie zdefiniowania już definiowanej funkcji program powinien zgłosić wyjątek i zakończyć działanie. (dotyczy to również funkcji z biblioteki standardowej).
- Inne błędy związane z analizą semantyczną

Biblioteka standardowa

- W przypadku funkcji scanCur() po pierwszym nieprawidłowym podaniu identyfikatora waluty wyświetli on dostępne waluty i umożliwi wprowadzenie jeszcze maksymalnie dwa razy. Za trzecim błędnie wprowadzonym identyfikatorem, program zgłosi wyjątek i zakończy swoje działanie.
- W przypadku wprowadzenia nieprawidłowej liczby w scanNum() program ponownie umożliwi wprowadzenie dwa razy, przy trzeciej porażce zakończy działanie.
- W przypadku przekazania funkcji print(), jako argument funkcji print() działanie funkcji oznaczone jest jako nieokreślone.

W trakcie wykonania

- W przypadku wystąpienia operacji dzielenia przez 0, program kończy swoje działanie.
- W przypadku dojścia do końca funkcji typu cur lub num w momencie w którym nie napotka instrukcji return.
- W przypadku wystąpienia przepełnienia w trakcie operacji, program ma wypisać komunikat.
- W przypadku braku odpowiedniej funkcji głownej.

Opis realizacji

Interpreter jest aplikacją konsolową, uruchamianą z konsoli wraz z argumentami. Wszelkie informacje będą wypisywane na standardowe wyjście – wyświetlane w konsoli.

Program złożony jest z modułów odpowiedzialnych za różne etapy interpretacji. Moduły te komunikują się ze sobą w określony sposób.

Wyróżnione moduły:

- Analizator leksykalny (lekser) – przetwarza znaki z podanego wejścia i zwraca leniwie tokeny posiadające typ, wartość oraz pozycje w źródle. Wartość tokenu jest odpowiednio ciągiem znaków lub liczbą stało przecinkową. Na typ etapie obsługiwane są znaki ucieczki. Maksymalne długości i wartości poczególnych tokenó○ są parametryzowane i określone w pliku. Komentarze są na tym etapie przetwarzane dalej.
- Analizator składniowy (parser) – wykonane zostały dwa parsery jeden przetwarzający plik źródłowy do interpretacji w języku walutowym, drugi do interpretacji pliku konfiguracyjnego. Oba są rekursywnie zstępujące sterowane składnią bez powrotów. Parser pliku konfiguracyjnego tworzy mapę mapującą identyfikator waluty na mapę przeliczników, która mapuje identyfikator drugiej waluty i przelicznik pozwalający na przeliczenie pierwszej waluty (tej w pierwszej wymienionej mapie) na tą drugą w drugiej mapie, po przez przemnożenie przez wskazywaną wartość. Parser pliku źródłowego po przeanalizowaniu tokenów zwraca obiekt programu, który zawiera

mapę nazw funkcji oraz ciała definicji funkcji. Po tym etapie dodawane są funkcjie wbudowane do mapy funkcji.

- Rejestrator drzewka programu(logger) – wykonany został w postaci wizytatora odwiedzającego poszczególne węzły obiektu Program – wszystkich jego funkcji. Odwiedzając dany węzeł wypisuje jego nazwę – nazwę obiektu i kontynuuje schodzenie w głąb.
- Analizator semantyczny – Analizuje wszystkie funkcje w obiekcie Program. Odpowiednio schodzi i sprawdza poprawność typów parametrów, argumentów, funkcji, ilość argumentów, istnienie funkcji, zmiennych. Wykrywa błędne operacje – przypisanie do zmiennej złego typu wyrażenia, używanie niezdefiniowanych zmiennych, funkcji, ponowne inicjowanie zmiennych w danym zakresie, błędne i niedozwolone operacje relacji, arytmetyczne, konwersji. Po przeanalizowaniu przez niego Programu jedyne występujące błędy to błędy podczas samego wykonania.
- Interpreter – Wykonuje przenalizowany program zaczynając od funkcji głownej, w przypadku wystąpienia przepełnienia, dzielenia przez zero lub dojścia do końca funkcji nie void-owej nie napotykając instrukcji return, zgłasza błąd podczas wykonania.

Widoczność zmiennych

Podczas wywołania funkcji ewaluowane są wszystkie argumenty tworzony jest nowy kontekst wykonania. W pierwszym zakresie(Scope) rejestrowane są wyliczone wartości argumentów. W momencie przejść do zagłębionych bloków operacji np. w if/while tworzony jest nowy zakres, w którym to rejestrowane są nowe zmienne. Zmienne są przykrywane to znaczy, że w przypadku zainicjowania zmiennej występującym w poprzednim zakresie zostanie ona zasłonięta do czasu wyjścia z tego zakresu. W przypadku zmiany wartości zmiennej w zagłębionym zakresie wartość ta zostaną przypisana do zmiennej z zakresu zewnętrznego, jeżeli nie jest przysłonięta przez inicjalizacje w bloku wewnętrznym.

Parametry funkcji są pierwszym zakresem i mogą być modyfikowane w zagłębionych zakresach.

Schemat działania programu.

1. Analizator leksykalny wczytuje plik konfiguracyjny przetwarza go leniwie.
2. Analizator składniowy pobiera kolejne tokeny pliku konfiguracyjnego i zapisuje ją w strukturze odpowiadającej za przechowywanie identyfikatorów walut oraz odpowiednich przeliczników.
3. Analizator leksykalny wczytuje plik źródłowy i przetwarza go.
4. Analizator składniowy pobiera kolejne tokeny pliku źródłowego i tworzy obiekty potrzebne do interpretacji. W trakcie wczytywania identyfikatorów walutowych prosi lekser konfiguracyjny o sprawdzenie poprawności identyfikatora i do tworzonego obiektu przypisuje odpowiedni identyfikator oraz mapę przeliczników.

5. Analizator semantyczny sprawdza obiekty stworzone i sens stworzonych obiektów.
6. Wykonanie programu napisanego w języku walutowym.

Testowanie

Testowanie opera się przede wszystkim na dużym pokryciu testami jednostkowymi. Jako biblioteka testująca została użyta GoogleTest dla C++. Wejściem testów są strumienie tekstowe stringstream. Test parsera sprawdzają tylko możliwość przeanalizowania danego ciągu bez błędów. Testy wyświetlają zrozumiały raport dla użytkownika.

Analizator leksykalny

Testowanie głownie testami jednostkowymi - prawidłowe budowanie pojedynczych tokenów. Sprawdzane są również przypadki dla których lekser powinien zgłosić błąd.

Analizator składniowy

Testowany testami jednostkowymi, sprawdzane czy ciąg tokenów zwraca prawidłowe mogą zostać interpretowane bez błędu.

Interpreter

Całość projektu jest testowana dla przykładowych skryptów prostych i zawiłych, przy czym sprawdzane jest czy prawidłowo wykonuje operacje i wyświetla prawidłowe wyniki. Dodatkowo sprawdzane są źle napisane skrypty w celu sprawdzenia obsługi błędów.

Informacje dodatkowe

Zmienne stało przecinkowe

Wykorzystana została implementacja liczby stało przecinkowej – decimal. Dostępna na licencji BSD – wykonanej przez: Piotr Likus. Implementacja zaczerpnięta z Internetu.

Uruchamianie testów

W celu wykonania testów należy przejść do folderu build i wywołać komendę ctest.

Finalny rezultat prac

Wszystkie wyżej przedstawione informacje zostały spełnione przez finalną wersje programu.