INTERPRETER JĘZYKA Z WBUDOWANYM TYPEM WALUTOWYM – DOKUMENTACJA WSTĘPNA

Krótki opis projektu

Celem projekt jest 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 argumenty do wypisania, scanNum() i scanCur() – funkcje przyjmujące znaki ze standardowego wejścia i zwracające odpowiednio liczbę stało przecinkową i liczbę stałoprzecinkową, walutową (dla funkcji scanCur wyświetla się dodatkowa prośba o podanie 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, może przyjmować wartości liczbowe typu num
- scanCur() - funkcja pozwalająca wczytać znaki ze standardowego wejścia aż do zatwierdzenia, może przyjmować wartości liczbowe typu cur. Po wprowadzeniu liczby prosi o podanie identyfikatora waluty.

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;
}
```

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);
	num d = numericFun (33.14 USD, 30 EUR, 5);
	print("Wartość c: ", c, " wartość d: ", d);
	# wypisze Wartość c: 9.52 USD wartość d: 5
}
```

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          1.2583     4.2506;
CAD   0.2340     1          0.2960;
PLN   0.7948     3.3788     1     ;
```

Plik konfiguracyjny zbudowany jest w następujący sposób:

- Pierwszy wiersz rozpoczyna się znakiem ukośnika wstecznego \ 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 dla USD do CAD przelicznik wynosi 1.2583

Założenia projektowe

Założenia wstępne

- Wymuszanie jawnej inicjalizacji zmiennych.
- Argumenty przekazywane przez kopię - wartość.
- Funkcje mogą zwracać tylko jeden typ lub nie zwracać nic.
- Silne i statyczne typowanie.
- Zmienne są mutowalne.
- Program musi składać się z przynajmniej jednej funkcji.

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 im instrukcją.
- Na każdym etapie interpretacji w przypadku wystąpienia błędu lub nieprawidłowości w źródle lub 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 wszystkich błędów z informacją czego oczekiwał program i co otrzymał, linii w źródle
- Analizator leksykalny nie powinien przerywać analizy źródła w przypadku wystąpienia nieprawidłowego/niezidentyfikowanego tokenu.

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. 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 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]

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 ".

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]
```

Poszczególne argumenty:

- plik_wejściowy – plik z programem w języku walutowym
- nazwa_funkcji – nazwa funkcji wykonywanej na początku programu(opcjonalne – domyślnie main)

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 wykorzystywany zostanie CMake.

Obsługa błędów

Obsług błędów w interpreterze ma przede wszystkim zadbać o prawidłowy przebieg procesu interpretacji źródła, powiadomić programistę, gdy zostanie wykryty błąd. Komunikaty powinny dokładnie wskazywać co oczekiwał program i co otrzymał z podaniem lokalizacji w źródle.

W przypadku źródła tekstowego pozycja to: numer wiersza i numer kolumny 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 program powinien kontynuować analizę w celu znalezienia większej ilości błędów.
- W przypadku wystąpienia słów kluczowych błędnie napisanych – z użyciem dużych liter, wyświetlany jest komunikat.
- 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.
- Statycznie wykrywa operacje dzielenia przez zero i zgłasza wyjątek.
- 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)

Biblioteka standardowa

- W przypadku funkcji scanCur() po pierwszym nieprawidłowym podaniu identyfikatora waluty wyświetli on dostępne waluty i poprosi o wprowadzenie jeszcze maksymalnie dwa razy. Za trzecim błędnie wprowadzonym identyfikatorem, program zgłosi wyjątek i zakończy swoje działanie.
- W przypadku wprowadzenia ciągu znaków zamiast liczby lub nieprawidłowego ciągu cyfr, program wypisze komunikat i zakończy swoje działanie.

W trakcie wykonania

- W przypadku wystąpienia operacji dzielenia przez 0, program kończy swoje działanie i wypisuje komunikat o operacji, w której to wystąpiło.
- W przypadku wystąpienia przepełnienia w trakcie operacji, program ma wypisać komunikat o operacji która go wywołała i zakończyć działanie.

Opis realizacji

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

Program będzie złożony z modułów odpowiedzialnych za różne etapy interpretacji. Moduły te muszą się komunikować ze sobą w określony sposób przez odpowiednie interfejsy.

Analizator leksykalny będzie oczekiwał pobrania tokenu, dopiero wtedy będzie go tworzył i zwracał – leniwa tokenizacja. Zwracane przez niego tokeny w przypadku literałów będą liczbami, a nie ciągami znaków (w przypadku liczb stało przecinkowych). Już na tym etapie obsługiwany jest escaping – znaki są podmieniane. Korzysta z parametryzowanych wartości określających maksymalne długości stałych znakowych, identyfikatorów, jak i wielkość liczb stałoprzecinkowych i ich maksymalną precyzję domyślnie 6 miejsc po przecinku. Analizator leksykalny będzie przetwarzał komentarze, ale nie będą one trafiać do analizatora składniowego.

Analizator składniowy będzie rekursywnie zstępujący sterowany składnią bez powrotów.

Strukturą do przechowywania zdefiniowanych funkcji i identyfikatorów wraz z przelicznikami będzie – unordered_map(słownik).

Schemat działania programu.

1. Analizator leksykalny wczytuje plik konfiguracyjny przetwarza go.
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.
5. Analizator semantyczny sprawdza obiekty stworzone i sens stworzonych obiektów.
6. Wykonanie programu napisanego w języku walutowym.

Testowanie

Testowanie będzie przede wszystkim opierało się na dużym pokryciu testami jednostkowymi. Jako biblioteka testująca zostanie użyta catch2 lub doctest dla C++. Wejściem testów będą strumienie tekstowe stringstream. Testy powinny wypisywać zrozumiały raport dla użytkownika.

Analizator leksykalny

Testowanie głownie testami jednostkowymi - prawidłowe budowanie pojedynczych tokenów. Sprawdzane będą również przypadki dla których lekser powinien zgłosić błąd. Dodatkowo testowane będą przykładowe skrypty, dzięki którym będzie możliwość sprawdzania czy prawidłowo zwróci wszystkie tokeny.

Analizator składniowy

Testowany testami jednostkowymi, sprawdzane czy ciąg tokenów zwraca prawidłowe obiekty do interpretacji. Dodatkowo testowany testami akceptacyjnymi sprawdzającymi prawidłowy przepływ informacji z analizatorem leksykalnym.

Interpreter

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