Przedmiot: Teoria Kompilacji II
Prowadzący: dr inż. Marcin Kuta
Zespół: Michał Mrowczyk, Damian Kudas

Temat: Interpreter języka Logo

Spis treści

1. Wstęp....3
    1. 1. O języku Logo....3
    1.2. Cel projektu....4
    1.3 Stosowane języki, narzędzia....4
2. Gramatyka LOGO....4
3. Tutorial do języka i przykłady....7
    3.1 Wstęp....7
    3.2 Hello Logo!....7
    3.3 Zmienne....7
    3.4 Operatory....8
    3.5 Instrukcje warunkowe: ifelse....9
    3.6 Pętle....9
    3.7 Wołanie funkcji....10
    3.8 Definiowanie funkcji....10
    3.9 Tworzenie domknięć:....11
    3.10 Przykładowy program:....12
    3.11 Więcej przykładów:....13
4. Opis systemu typizacji i scopingu....13
5. Obsługa i użytkowanie programu....14
    5.1 Opis instalacji i dependencji programu....14
    5.2 Opis użytkowania....14
6. Architektura programu....15
7. Napotkane problemy i rozwiązania....15

1. Wstęp

1. 1. O języku Logo

Język Logo jest wieloparadygmatowym językiem powstałym w 1967 roku jako dialekt języka LISP (Więcej: http://en.wikipedia.org/wiki/Lisp_%28programming_language%29). Pierwotnie był to język funkcyjny. Obecnie istnieje wiele implementacji języka Logo (np. PyLogo, UCBLogo i inne). Dziś głównie kojarzony jest on z grafiką żółwia (za pomocą wydawanych komend możemy poruszać żółwiem (kursorem) i rysować po ekranie różnego rodzaju obiekty geometryczne). Tak rozumiany język Logo stał się językiem edukacyjnym. Dzieci uczą się programowania przez zabawę, wydając komendy rysujące różnego rodzaju wzory na ekranie. Okazuje się, że zabawa taka jest świetną formą rozrywki i organizowane są różne konkursy na najbardziej osobliwy kształt stworzony przez program w języku Logo o długości poniżej 15 słów (Więcej: http://www.mathcats.com/gallery/15wordcontest.html). Język Logo w wersji ogólnej nie posiada gramatyki bezkontekstowej. Spowodowane jest to m.in. brakiem nawiasowania argumentów funkcji, a co za tym idzie np. niemożliwością sparsowania wyrażenia: „foo 1 2 bar 1 3” bez informacji o arności (liczbie przyjmowanych argumentów) przez funkcje 'foo' oraz 'bar'. Język Logo zainspirował m.in. programistów popularnego języka Python do stworzenia biblioteki „turtle” umożliwiającej rysowanie po ekranie grafiki żółwia (Więcej: http://docs.python.org/2/library/turtle.html) .

1.2. Cel projektu

Celem projektu jest stworzenie własnej wersji interpretera języka Logo. Ponadto naszym celem jest stworzenie interpretera zdolnego do interpretacji programów z konkursu programistycznego o którym była mowa w rozdziale 1.1: '15wordcontest' oraz umożliwiającego definiowanie funkcji i innych elementarnych konstrukcji każdego języka programowania. Celem projektu nie jest natomiast wierne naśladowanie istniejących już implementacji języka Logo (choćby ze względu na brak ściśle sprecyzowanego standardu języka Logo).

1.3 Stosowane języki, narzędzia.

Na potrzeby parsingu za pomocą biblioteki PLY (Więcej: http://www.dabeaz.com/ply/) oryginalna gramatyka języka Logo zostanie nieco zmodyfikowana (m.in. wprowadzamy nawiasowanie argumentów funkcji), tak aby uzyskać gramatykę bezkontekstową. Z racji używania biblioteki PLY językiem implementacji będzie Python. Wybieramy go, gdyż mamy największe doświadczenie w parsowaniu właśnie w tym języku. Składnia Pythona umożliwia łatwą i wygodną manipulację na tekście oraz strukturach danych, co jest nieocenioną zaletą w przypadku tworzenia interpretera. Dodatkową zaletą Pythona, która znacznie ułatwia nam osiągnięcie celu jest obecność modułu 'turtle' służącego do realizacji grafiki żółwia w Pythonie. Dzięki temu nie musimy martwić się sami o rysowanie po ekranie, tylko opakowujemy funkcjonalność używanego modułu

2. Gramatyka LOGO

Poniżej przedstawiamy bezkontekstową wersję gramatyki języka Logo.

Umożliwia ona definiowanie wyrażeń arytmetycznych, porównania, funkcji, instrukcji przypisania, warunkowych i pętli. Dodajemy możliwość definiowania closures (domknięć, w stylu Pythona)

Domknięcia są nieco eksperymentalne, ale przynajmniej w prostych przypadkach powinny działać...

Konwencje dotyczące specyfikacji gramatyki:

Tokeny umieszczone w pojedynczym cudzysłowie : ' '

Słowa kluczowe pogrubione np. **make**

program:

instr_list

```
        func_list
        func_list instr_list
func_list:
        func
        func_list func
instr_list:
        instr
        instr_list instr
func:
        to word  '(' params ')' extended_instr_list end
extended_instr_list:
        instr
        func
        extended_instr_list instr
        extended_instr_list func
params:
        ref
        params ref
ref:
        ':' word
instr:
        make quoted expr
        repeat expr '[' instr_list ']'
        for  '[' word expr expr expr ']' '[' instr_list ']'
        for  '[' word expr expr ']' '[' instr_list ']'
        ifelse condition '[' instr_list ']'  '[' instr_list ']'
        word '(' expr_list ')'
expr_list:
        expr
        expr_list ',' expr
expr:
        const
```

name

ref

word '(' expr_list ')'

expr '+' expr

expr '–' expr

expr '*' expr

expr '/' expr

'(' expr ')'

condition:

expr '=' expr

expr '!=' expr

expr '<=' expr

expr '>=' expr

expr '<' expr

expr '>' expr

quoted:

'"' word

name:

word

const:

float

integer

word:

sequence of leters and digits without special characters starting with non number

float:

floating point number

integer:

integer number without floating point

3. Tutorial do języka i przykłady

3.1 Wstęp

Rozdział 2 przedstawia gramatykę języka. Warto ją stosować jako punkt odniesienia w czasie czytania tego samouczka. Rozdział 5 tej dokumentacji zawiera opis instalacji i użytkowania programu. W przypadku gdy zechcesz testować przykłady opisane w tutorialu, zalecamy najpierw przeczytać ten rozdział. Gramatyka języka różni się nieco od standardowych implementacji Logo (m.in. ze względu na nawiasowanie argumentów funkcji). Stąd nawet jeśli znasz już jakiś inny dialekt Logo, zalecamy abyś przeczytał ten rozdział.

3.2 Hello Logo!

```
fd(100) // idziemy o 100 pixeli do przodu
```

Tak może wyglądać najprostszy program w języku Logo.

fd to funkcja, która przyjmuje wartość określającą ile pixeli zgodnie z kierunkiem głowicy żółwia mamy się przemieścić. Funkcja ta może być wywołana równoważnie jako:

```
forward(100)
```

Inne elementarne funkcje operujące na żółwiu to:

bk(x) – idziemy fo tyłu o x pixeli

lt(x) – obracamy się w lewo (przeciwnie z ruchem wskazówek zegara) w miejscu o x stopni

rt(x) – obracamy się w prawo (zgodnie z ruchem wskazówek zegara) w miejscu o x stopni

pu() - podnosimy pisak (powoduje że choć żółw się przemieszcza, to linia nie jest rysowana)

pd() - opuszczamy pisak (linia jest rysowana gdy żółw się przemieszcza – domyślnie)

Tak naprawdę interpreter umożliwia nam wykonanie znacznie większej ilości operacji i podane powyżej stanowią tylko niewielki ułamek możliwości. Zainteresowanych odsyłamy do:

http://docs.python.org/2/library/turtle.html

Wszystkie funkcje zdefiniowane w module turtle są także dostępne w naszym interpreterze.

3.3 Zmienne

Elementarną operacją w każdym języku programowania jest operacja przypisania, czyli nadania wartości zmiennej. Zmienna jest nazwą symboliczną określającą miejsce w pamięci, pod którym składowana jest zawartość zmiennej. Język Logo jest językiem dynamicznie typowanym, co oznacza że możemy stworzyć zmienną bez wcześniejszego deklarowania jej typu. Zmienna otrzymuje automatycznie typ taki jaki ma wartość do niej przypisywana.

W języku Logo tworzymy zmienną za pomocą instrukcji **make:**

```
make ”x 3.14159 // przypisujemy zmiennej x wartość typu float
```

make ”y 42 + 24 // przypisujemy zmiennej y wartość typu integer

Uwaga: W powyższym przypisaniu zmienna y otrzyma wartość 66. Dzieję się tak dlatego że interpreter oblicza wartość wyrażenia podanego jako drugi argument instrukcji **make**

Aby wyłuskać wartość zmiennej musimy posłużyć się operatorem **:**

make ”x 100 fd(:x) // ustawiamy wartość zmiennej x na 100 i idziemy do przodu

Przyjrzyjmy się dlaczego Logo nie pozwala nam napisać czegoś bardziej intuicyjnego:

make ”x 100 fd(x)

Taka próba skończy się błędem składniowym. Tak naprawdę w wielu dialektach języka logo napis:

'x' oznacza funkcję bezargumentową 'x'. Operator **:** został wprowadzony, aby odróżnić zmienne od funkcji. W naszym dialekcie wprawdzie funkcje bezargumentowe mają inną składnię, ale zachowujemy operator **:** ze względów historycznych.

3.4 Operatory

Aby tworzyć wyrażenia niezbędne są operatory. W naszym dialekcie Logo wyróżniamy dwa rodzaje operatorów: arytmetyczne i porównania.

Operatory działają zarówno na zmiennych typu integer jak i float i ich zachowanie jest takie jak odpowiednich operatorów języka Python tzn. np. suma integer'a i float'a daję float'a

Poniższa tabela przedstawia operatory:

operator	znaczenie	uwagi
+	Dodaje dwie liczby	
-	Odejmuje dwie liczby	
*	Mnoży dwie liczby	
/	Dzieli dwie liczby	3 / 2 = 1 (dzielenie całkowitoliczbowe dla intów)
=	Sprawdza czy równe	Brak operatora == kojarzonego z wielu języków
!=	Sprawdza czy różne	
<	Mniejsze	
<=	Mniejsze bądź równe	
>	Większe	
>=	Większe bądź równe	

3.5 Instrukcje warunkowe: ifelse

ifelse jest jedyną instrukcją warunkową jaką wprowadziliśmy w nasz. Ten minimalizm wystarcza jednak na symulowanie dowolnych instrukcji warunkowych z innych języków.

Przykładowe użycie:

```
ifelse 1=1 [fd(100) rt(100)] [bk(100) lt(100)]
```

W powyższym przykładzie warunek 1=1 jest oczywiście prawdziwy, więc zostaną wykonane instrukcje zawarte w pierwszej liście: fd(100) rt(100)

W przypadku gdy zmienimy np. warunek z 1=1 na 1!=1, to zostaną wykonane instrukcje na drugiej liście instrukcji: bk(100) lt(100).

Oczywiście **ifelse** może być zagnieżdżane, aby otrzymać dowolne rozgałęzienie instrukcji warunkowych.

3.6 Pętle

Pętle przychodzą w postaci dwóch instrukcji **for** oraz instrukcji **repeat**

Popatrzmy na przykład:

```
for [i 1 10 2] [fd(100)]
```

Powyższy zapis oznacza, że zmienna sterująca pętli nazywać się będzie i. Jej wartość początkowa w momencie startu pętli będzie ustawiona na 1, maksymalna wartość i dla której pętla będzie wciąż wykonywana będzie wynosić 10 i krok będzie wynosił 2 (tzn. po każdym przebiegu pętli zmienna i będzie zwiększana o 2). Instrukcje wykonywane w pętli znajdują się na drugiej liście: w tym przypadku jest to tylko fd(100). Zmienna sterująca i jest dostępna wewnątrz pętli, lecz nie powinniśmy jej zmieniać w trakcie wykonywania pętli manualnie.

Spójrzmy jeszcze na przykład:

```
for [i 1 10 1] [bk(50)]
```

Tak naprawdę jeśli krok w pętli wynosi 1 to zapis ten może być skrócony do:

```
for [i 1 10] [bk(50)]
```

i otrzymamy ten sam efekt.

Spójrzmy na przykład dla pętli **repeat:**

```
repeat 100 [bk(100) pu() fd(100)]
```

Jej konstrukcja jest taka, że 100 razy wykonywane są instrukcje na liście. **repeat** jest prostszą wersją **for**

Ciekawą własnością pętli **repeat** jest to, że w każdej iteracji mamy dostęp do funkcji wbudowanej w tą pętlę: repcount(), która zwraca numer aktualnej iteracji np.

```
repeat 100 [fd(repcount())]
```

powoduje, że najpierw idziemy o 0 pixeli do przodu, później o 1, później o 2 i tak dalej aż do 100

3.7 Wołanie funkcji

Jeśli uważnie przyjrzymy się gramatyce w rodziale 2 to zobaczymy w definicji instrukcji coś na kształt:

word (expr_list)

Konstrukcja ta znaczy, że wołamy funkcję o nazwie word z listą argumentów expr_list.

W ten sposób możemy wołać funkcje z modułu **turtle** (jak to już było wcześniej pokazywane).

Możemy też wołać funkcje z modułu **random** oraz modułu **math** (np. funkcje trygonometryczne). Zawsze można sprawdzić jakie funkcje są dostępne w dokumentacji tych modułów dla języka Python:

Dokumentacja modułu random

Dokumentacja modułu math

Przykład:

fd(ceil(randint(25, 75) * cos(31))

W powyższym przykładzie interpreter losuje liczbę całkowitą z przedziału [25, 75] a następnie mnoży ją przez cos(31) (jednostki w radianach) oraz wywołuje funkcje ceil na wyniku i przesuwa żółwia o tyle pixeli do przodu jaki był wynik obliczeń.

3.8 Definiowanie funkcji

Nasz interpreter umożliwia definiowanie funkcji przez użytkownika. Funkcje mogą zwracać wartość lub wykonywać serię instrukcji bez zwracania wartości (co jest reprezentowane przez zwracanie stałej **None)**

W poniższym przykładzie zdefiniujemy funkcję polygon która rysuje wielokąt foremny o ilości boków podanych jako argument funkcji:

```
to poly(:x)
    repeat :x [fd(30) lt(360.0 / :x)]
end
```

Aby zwrócić wartość zdefinujemy funkcję a_mean zwracającą średnią arytmetyczną dwóch liczb przekazanych jako argumenty

```
to a_mean(:x :y)
    make ”a_mean (:x + :y) / 2.0
end
```

Widzimy, że definicja każdej funkcji musi zacząć się od słowa kluczowego: **to** i zakończyć słowem kluczowym: **end** Pomiędzy tymi słowami występuje nazwa funkcji z listą parametrów, gdzie lista

parametrów ma postać: (:p1 :p2 :p3 :pn) lub może być pusta: () Zauważmy, że parametry **nie** są oddzielane przecinkami jak przy wywołaniu są argumenty!

Zwracanie wartości z funkcji (opcjonalne) odbywa się za pomocą konstrukcji:

```
make ”nazwa_funkcji wartosc
```

gdzie nazwa_funkcji to nazwa zdefiniowanej funkcji, a wartosc – wyrażenie które ewaluuje się do wartości zwracanej z funkcji.

W przypadku trybu interaktywnego możemy redefiniować funkcje tzn. np. jeśli zdefinowaliśmy funkcję: **f** to ponowna definicja funkcji **f** spowoduje zastąpienie poprzedniej funkcji.

Uwaga: Nazwa funkcji nie powinna być równa: __main__ - w przeciwnym wypadku narażamy się na możliwe niepożądane efekty uboczne.

3.9 Tworzenie domknięć:

Spójrzmy na przykład:

```
to gen_func(:x)
    make "y 100
    to func(:z)
        fd(:x)
        rt(:y)
        fd(:z)
    end
    make "gen_func func
end
make "x gen_func(100)
repeat 20 [x(100)]
```

W powyższym kodzie tworzymy funkcję przyjmującą jeden argument: gen_func

Funkcja ta zawiera w sobie zdefiniowaną funkcję: func - także przyjmującą jeden argument.

Funkcja: gen_func zwraca ponadto funkcję: func

Idea jest prosta: funkcja gen_func w zależności od podanego argumentu zwraca różne instancje funkcji: func. Funkcja func zawiera w sobie 'stan' funkcji: gen_func w momencie gdy sterowanie doszło do definicji funkcji: func

Funkcja gen_func może być traktowana jako swoista fabryka funkcji: func, ustawiająca wartości części zmiennych funkcji: func w momencie tworzenia jej instancji. Przykład przedstawia w jaki

sposób można tworzyć proste domknięcie i wykorzystywać je do rysowania po ekranie.

Zobacz również przykład: closures.txt w katalogu examples/

3.10 Przykładowy program:

```
to left_side()
        rt(20) fd(20) lt(20) fd(60)
end

to top_side()
        rt(90) fd(25) rt(90)
end



to right_side()
        fd(60) lt(20) fd(20) rt(20)
end

to return_start()
        rt(90) fd(40) rt(90)
end

to trunk()
        left_side() top_side() right_side() return_start()
end

to center_top()
        pu() fd(80) rt(90) fd(20) lt(90) pd()
end

to circle()
        repeat 360 [fd(1) rt(1)]
```

```
end

to tree()
        pu() bk(100) pd() trunk() center_top() left(90) circle()
end

tree()
```

3.11 Więcej przykładów:

Więcej przykładów znajduje się w katalogu examples projektu. Są tam w większości przerobione na nasz dialekt Logo niektóre przykłady ze strony:

http://www.mathcats.com/gallery/15wordcontest.html

4. Opis systemu typizacji i scopingu.

Logo jest językiem dynamicznie typowanym, dlatego też typ zmiennej jest związany z wartością przypisaną podczas jej tworzenia. Wykonanie niedozwolonej operacji (pod względem typu), powoduje błąd czasu wykonania. W naszym przypadku interpreter działa w języku Python, więc rzucany wyjątek, będzie wyjątkiem rzucanym przez ten język.

W przypadku scopingu gramatyka języka pozwala na definiowanie funkcji, które posiadają własny zestaw parametrów (przysłaniający zmienne definiowane globalnie w ciele funkcji). Ponadto zmienne definiowane lokalnie w ciele funkcji / procedury mogą przysłonić zarówno zmienne definiowane globalnie jak i parametry funkcji. Redefinicja zmiennej zdefiniowanej w danym bloku kodu powoduje utratę wartości poprzednio zdefiniowanej zmiennej i zastąpienie jej przez wartość wynikającą z redefinicji (ponownego przypisania). Jest to konwencja stosowana w wielu językach dynamicznie typowanych (np. języku Python).

Ponadto wprowadzamy osobne scope'y dla instrukcji warunkowych: **IfElse**, pętli: **For** oraz **Repeat**

Przykładowo: po wyjściu z ciała pętli zmienne zdefiniowane w niej **nie** są widoczne na zewnątrz pętli, jak również nie są widoczne wszelkie zmiany, które były dokonane na zmiennych zewnętrznych (z zewnętrznego scope'u) wewnątrz pętli (zmienne w Logo są immutable tzn. zmiana zmiennej skutkuje de facto stworzeniem nowej zmiennej o tej samej nazwie).

Zobacz również przykład: scopes.txt w katalogu examples/

5. Obsługa i użytkowanie programu

5.1 Opis instalacji i dependencji programu

Program wymaga do działania języka z rodziny Python

Na systemach debianowych (np. Ubuntu) możemy zainstalować język Python poleceniem z terminala:

sudo apt-get install python

W przypadku sytemów Windows i innych możemy udać się na stronę:

http://www.python.org/getit/

i zainstalować interesującą nas wersję.

Program do działania wymaga także biblioteki Pythona: PLY

W przypadku Ubuntu wystarczające jest:

sudo apt-get install python-ply

Zawsze można też udać się na stronę PLY:

http://www.dabeaz.com/ply/

w celu przeczytania dokumentacji i instalacji.

5.2 Opis użytkowania

W głównym katalogu programu znajduje się plik logo.py.

Pierwszą linią jest: #!/usr/bin/python

Oznacza to, że na systemach debianowych korzystamy z domyślnego interpretera Pythona, który znajduje się pod powyższą lokalizacją. Jeśli chcemy używać np. Python3, to może okazać się konieczna zmiana tej linii na: #!/usr/bin/python3

W przypadku problemów musimy upewnić się czy zainstalowaliśmy wszystkie wymagane moduły Pythona (np. python-tk dla wersji Python3).

Program możemy uruchamiać z linii poleceń (terminala) następująco:

python logo.py <nazwa_pliku> - z podaniem nazwy interpretowanego pliku

python logo.py – w trybie interaktywnym

lub jeśli chcemy zrobić to bardziej skrótowo, korzystając z domyślnego interpretera

./logo.py <nazwa_pliku>

./logo.py

Przykładowe użycie:

python logo.py example.txt (interpretuje plik example.txt)

W trybie interaktywnym (Użycie: python logo.py) pojawia się znak zachęty: <<

interpreter oczekuje na jedno liniowe definicje funkcji lub / oraz komendy.

Komenda **help** wypisuje pomoc, zaś komenda **exit** powoduje wyjście z trybu interaktywnego.

W rozdziale 3 opisujemy pokrótce możliwości języka w krótkim tutorialu.

6. Architektura programu

Program składa się z dwóch plików w języku Python:

1. logo.py – zawiera funkcje używane przez generator PLY (poprzedzone przedrostkiem: p_), definicje leksemów dla skanera, klasę interpretera, która służy do tworzenia metod eval, mających za zadanie ewaluację kodu w zdefiniowanym przez nas dialekcie Logo. Ponadto plik ten zawiera obsługę tworzenia dwóch trybów: interpretowania pliku oraz trybu interaktywnego. W razie błędów podejmowane są różne akcje w zależności od trybu interpretera.
2. AST.py – zawiera model abstrakcyjnego drzewa syntaktycznego dla naszego dialektu Logo. Model ten składa się z szeregu klas reprezentujących węzły w drzewie, gdzie root'em jest zawsze obiekt klasy: Prog

7. Napotkane problemy i rozwiązania

Problem 1:

Z powodu braku mechanizmu zachłannego dopasowywania, napisanie identyfikatora, rozpoczynającego się od słowa kluczowego po słowie kluczowym np. 'to tol' powodowało błąd składniowy.

Rozwiązanie 1:

Problem został rozwiązany przez dodanie słownika 'reserved', w którym były słowa kluczowe i dopiero po dopasowaniu określenie czy jest to słowo kluczowe, czy identyfikator (wprowadzenie zachłannego dopasowywania).

Problem 2:

Brak bezkontekstowości gramatyki języka Logo.

Rozwiązanie 2:

Skonstruowanie gramatyki jak w rozdziale 2, zrezygnowanie z elegancji składni większości dialektów Logo i wprowadzenie nawiasowania argumentów funkcji oraz znaku: , (przecinek) do oddzielania argumentów przy wywoływaniu funkcji.

Problem 3:

Potrzeba rysowania grafiki żółwia, korzystania z funkcji matematycznych (np. trygonometrycznych), korzystania z generatora liczb pseudolosowych

Rozwiązanie 3:

Zaimportowanie modułów Pythona odpowiednio: turtle, math, random.

Skorzystanie z mechanizmu funkcji Pythona: eval, do szybkiej ewaluacji wbudowanych funkcji w interpreterze języka Logo.

Problem 4:

Potrzeba dynamicznego definiowania komend i funkcji w czasie działania pętli interpretera

Rozwiązanie 4:

Wprowadzenie trybu interaktywnego, rozgraniczenie obsługi błędów składniowych w trybie zwykłym i interaktywnym (dla trybu interaktywnego błąd w komendzie nie powoduje końca działania interpretera, tylko umożliwia użytkownikowi wpisanie komendy od nowa).

Problem 5:

Niezgodność jednostek w funkcjach trygonometrycznych modułu math Pythona (radiany) z jednostkami, które są użyte w domyślnych funkcjach trygonometrycznych głównych dialektów Logo (stopnie).

Rozwiązanie 5:

Pozostawienie radianów jako jednostek (dla zachowania kompatybilności z funkcjami modułu math). Użytkownik powinien sam jawnie przeliczać na stopnie jeśli chce podawać je jako argumenty wywołania

Problem 6:

Potrzeba definiowania funkcji, które zwracają wartość oraz komend, które nie zwracają wartości.

Rozwiązanie 6:

Wprowadzenie mechanizmu analogicznego do języka Pascal, w który wartość zwracana z funkcji może być zdefiniowana przez przypisanie do odpowiedniej zmiennej wewnątrz funkcji.

Przykład:

to func(:x)

make ”func 10 // zwracamy wartość 10

end

Problem 7:

Przy zwracaniu wartości z funkcji konieczność sprawdzania, czy przypisanie do zmiennej będącej jednocześnie nazwą funkcji nastąpiło w ciele tej funkcji i nigdzie indziej !

Rozwiązanie 7:

Wprowadzenie zmiennych globalnych interpretera:

bool returning – wskazującej na to, czy w ciele bieżącej funkcji nastąpiło przypisanie.

True jeśli tak, False jeśli nie

string funkyname – nazwa obecnie interpretowanej funkcji

Domyślnie ustawiona na __main__