Wydział Informatyki, Elektroniki i Telekomunikacji

Interpreter Prostego Języka Obiektowego

Teoria Kompilacji II

Prowadzący:
dr inż. Marcin Kuta

Autorzy:
Łukasz Cieśla
Tomasz Kwiecień

Wykorzystane narzędzia:

Projekt realizowaliśmy w języku Python, w środowisku PyCharm, korzystając z narzędzi takich jak:

- ply - narzędzie służące do tworzenia kompilatorów oraz translatorów z opisu gramatyki, bezpośrednia implementacja idei programów Lex oraz Yacc w języku Python
- git - system kontroli wersji, repozytorium w serwisie github

Uruchomienie:

Interpretowany kod znajduję się w pliku *example.txt*. Po zapisaniu kodu programu w podanym pliku należy uruchomić skrypt *main.py*.

Specyfikacja języka:

Zaproponowany przez nas język obiektowy zapewnia mechanizmy definiowania klas, dziedziczenia (niedopuszczalne jest dziedziczenie wielobazowe), trzech poziomów dostępności elementów klasy. Dostępne są trzy type proste int, float oraz string. Język narzuca rygorystyczną konwencję pisania programów: na początku musza się znaleźć definicje klas, następnie klas dziedziczących po nich, inicjalizowanie zmiennych, definicje funkcji oraz blok wywołań instrukcji.

Definicja klasy zawiera zbiór pól oraz definicję metod klasy. Ważne jest, że bloki te oddzielać musi średnik (“ ; ”). Jeżeli nie chcemy, aby klasa miała pola tylko metody, jej definicję jej ciała należy rozpocząć średnikiem, podobnie jęzli chcemy, aby klasa nie miała metod, tylko same pola, wówczas definicję kończymy średnikiem.

Elementy interpretera:

Zaimplementowany przez nas interpreter składa się z definicji węzłów AST (Abstact Syntax Tree, plik AST.py), skanera (plik scanner.py), parsera (Cparser.py), modułu sprawdzającego typy (TypeChecker.py) oraz modułu interpretującego (Interpreter.py).

1. AST

Zawiara definicję węzłów drzewa rozbioru kodu wejściowego. Wszystkie dziedziczą po klasie *Node*, która ma zdefiniowane dwiemetody:

- *accept*, używaną przez TypeCheckera
- *acceptInt*, używaną przez Interpreter

2. Skaner

Bezpośrednio wykorzystywany przez parser. Zawiera listę literałów, słów kluczowych oraz definicje wyrażen regularnych wykorzystywanych przy podziale kodu wejściowego na tokeny.

3. Parser

Wykorzystując skaner oraz definicję AST dokuneje analizy syntaktycznej tworząc drzewo rozbioru kodu wejściowego na drzewo zgodne z zdefiniowaną gramatyką (podaną poniżej)

4. Type Checker

W celu intepretacji skanera semantycznego użyliśmy wzorca projektowego wizytor. Dla każdego węzła drzewa syntaktycznego wywoływana jest funkcja accept, która odpowiada za testowanie poprawności semantycznej.

Informacje o przetworzonej części programu interpretowanego przechowywane są w tablicy symboli SymbolTable, w której wyróżniliśmy dwa specjalne symbole: FunSymbol i ClassSymbol. ClassSymbol przechowuje w sobie informacje o:

- klasie bazowej
- modyfikatorze dostępu pól
- modyfikatorze dostępu klasy

Udostępnia metodę pozwalającą na określenie czy w danym zakresie dana składowa klasy jest widoczna.

W związku z tym, że w aktualnej implementacji stworzenie ClassSymbol'u wymaga wcześniejszego zaakceptowania zawartości klasy, niemożliwe jest zadeklarowanie pola o typie deklarowanej klasy.

Przy inicjalizowaniu obiektu należącego do zdefiniowanej klasy, do tablicy symboli dodawane są pola i metody danej klasy, oraz wszystkich przodków (klasy bazowej, klasy bazowej klasy bazowej itd.). Nazw dodanego symbolu ustalana jest przez metodę makeClassContentName. Aby uniknąć kolizji nazw w tablicy symboli, przy dodawaniu zawartości klas przodków dodawane są one, tak jak by były zawartością klasy potomnej. Analogicznie, przy próbie dostępu do zawartości klasy, jeśli w danym scopie przy pomocy metody ClassSymbolu udało się potwierdzić, że zawartość jest dostępna, przy pomocy makeClassContentName tworzona jest nazwa, której szukamy w tablicy symboli.

5. Interpreter

Podobnie jak w przypadku TypeCheckera użyliśmy wzorca *visitor* wywołując tym razem metodę *acceptInt*. Odpowiedzialny jest za wykonanie sprawdzonego pod względem definicji programu.

Informację o przetworzonej części kodu programu przechowywane są na stosie *StackMemory*, którego składowe zawierają słowniki odwzorowujące nazwę elemtu na jej wartość. Do

reprezentacji funkcji użyto zdefiniowanek klasy *Function* (definicja powyższych elementów znajduję się w pliku *Memory.py*).

Przy definiowaniu klas dodajemy na stos odkładane są elementy postaci {<nazwa-klasy>.<nazwa_pola_lub_metody>, *<wartość>*}, jeżeli mamy do czynienia z klasą dziedziczącą wówczas tworzymy adekwatne elementy kopiując wartości dodanych podczas definiowania klasy bazowej

Przy deklaracji na stos odkładane są elementy postaci {nazwa, wartość}, jeżeli deklaracja dotyczy klasy wówczas dodajemy na stos elementy {*<nazwa_deklarwanej_klasy>.<nazwa>*, *<wartość>*}, które kopiowane są z odłożonych podczas definicji klasy elementów

Użyta gramatyka:

program	->	classdefs declarations fundefs instructions
declarations	->	declarations declaration
declarations	->	<empty>
declaration	->	TYPE inits ;
declaration	->	ID classinits ;
declaration	->	error ;
inits	->	inits , init
inits	->	init
init	->	ID = expression
classinits	->	classinits , classinit
classinits	->	classinit
classinit	->	ID
instructions	->	instructions instruction
instructions	->	instruction
instruction	->	expression
instruction	->	print_instr
instruction	->	labeled_instr
instruction	->	assignment
instruction	->	choice_instr
instruction	->	while_instr
instruction	->	repeat_instr

instruction -> return_instr

instruction -> break_instr

instruction -> continue_instr

instruction -> compound_instr

print_instr -> PRINT expression ;

print_instr -> PRINT error ;

labeled_instr -> ID : instruction

assignment -> access = expression ;

access -> ID

access -> ID.ID

choice_instr -> IF (condition) instruction

choice_instr -> IF (condition) instruction ELSE instruction

choice_instr -> IF (error) instruction

choice_instr -> IF (error) instruction ELSE instruction

while_instr -> WHILE (condition) instruction

while_instr -> WHILE (error) instruction

repeat_instr -> REPEAT instructions UNTIL condition ;

return_instr -> RETURN expression ;

continue_instr -> CONTINUE ;

break_instr -> BREAK ;

compound_instr -> { declarations instructions }

condition -> expression

const -> INTEGER

const -> FLOAT

const -> STRING

expression -> const

expression -> access

expression -> expression + expression

expression -> expression - expression

expression -> expression * expression

expression -> expression / expression

expression -> expression % expression

expression -> expression | expression

expression -> expression & expression

expression -> expression ^ expression

expression -> expression AND expression

expression -> expression OR expression

expression -> expression SHL expression

expression -> expression SHR expression

expression -> expression EQ expression

expression -> expression NEQ expression

expression -> expression > expression

expression -> expression < expression

expression -> expression LE expression

expression -> expression GE expression

expression -> (expression)

expression -> (error)

expression -> access (expr_list_or_empty)

expression -> access (error)

expr_list_or_empty -> expr_list

expr_list_or_empty -> <empty>

expr_list -> expr_list , expression

expr_list -> expression

fundefs -> fundef fundefs

fundefs -> <empty>

fundef -> TYPE ID (args_list_or_empty) compound_instr

fundef -> ID ID (args_list_or_empty) compound_instr

args_list_or_empty -> args_list

args_list_or_empty -> <empty>

args_list -> args_list , arg

args_list -> arg

arg -> TYPE ID

arg -> ID ID

classdefs -> classdef classdefs

classdefs -> <empty>

classdef -> accessmodificator CLASS ID classcontent

classdef -> accessmodificator CLASS ID EXTENDS ID classcontent

classcontent -> { fieldefs methoddefs }

fielddefs -> fielddefs fielddef

fielddefs -> <empty>

fielddef -> accessmodificator declaration

methoddefs -> methoddefs methoddef

methoddefs -> <empty>

methoddef -> accessmodificator fundef

accessmodificator -> PRIVATE

accessmodificator -> PROTECTED

accessmodificator -> PUBLIC

Bibliografia:

1. Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman - *Compilers: Principles, Techniques, and Tools*

2. http://kompilatory.agh.edu.pl/pages/main/wyklady.php?dir=tk-wyklady&format=pdf

3. http://pl.wikipedia.org/wiki/Odwiedzaj%C4%85cy