Interpreter języka obsługującego wyjątki
Metody Translacji

Dawid Maksymowski

24 stycznia 2022

1 Opis projektu

Niniejszy dokument stanowi dokumentację oraz opis funkcjonalności interpretera języka programowania **C--**, stanowiącego projekt zaliczeniowy na przedmiocie Techniki Kompilacji / Metody Translacji. Ideą projektu jest stworzenie interpretera prostego języka podobnego do C, zawierającego wbudowany mechanizm wyjątków.

2 Funkcjonalność języka

Program składa się z funkcji, wśród których musi znajdować się funkcja `["int" | "void"] main()` stanowiąca początek wykonywania programu. Język posiada tylko jeden wbudowany typ danych liczbowych `int`. Język nie posiada osobnego typu logicznego `bool`; zamiast tego operatory relacji i porównania zwracają 1 w przypadku, gdy porównanie jest spełnione oraz 0 w przypadku przeciwnym. Również ciało instrukcji warunkowych jest wykonywane wtedy, kiedy warunek ewaluuje się do dowolnej liczby niezerowej. Funkcję nie zwracającą żadnej wartości można zadeklarować ze zwracanym typem `void`. Semantyka typowania: statyczne, słabe, mutowalne. Widoczność zmiennej kończy się wraz z końcem bloku, w którym została zadeklarowana. Zmienna jest widoczna w bloku wewnętrznym względem bloku, w którym została zadeklarowana. Parametry do funkcji są przekazywane przez referencję, natomiast przypisanie następuje poprzez kopię – dwie różne zmienne o typie `int` nigdy nie wskazują na ten sam obiekt, nawet jeżeli mają taką samą wartość.

String Ponadto w instrukcji `print` możliwe jest użycie typu znakowego `string`. Wyróżnikiem stringa jest cudzysłów. Dopuszczone są sekwencje specjalne `"\"", "\n", "\t", "\\"` oznaczające kolejno cudzysłów, znak nowej linii, tabulator i backslash. Niedozwolone jest używanie jakichkolwiek innych sekwencji specjalnych lub stricte znaku końca linii, tj. np string w postaci `"\x"` lub `"\<newline-char>"` wyrzuci błąd (skaner zwróci token `Error`). Możliwa jest konkatenacja stringów przy pomocy operatora „`+`”, również z liczbami. Jednym ze składników takiej konkatenacji musi być wówczas string.

2.1 Operatory

Tabela 1 przedstawia wszystkie dopuszczalne operatory języka wraz z ich pierwszeństwem i łącznością. Ze względu na używanie wyłącznie typu `int`, rzutowanie między typami nie jest kwestią do rozważania. Przypisanie otrzymuje wartość przypisania, tj. jego prawą stronę.

2.2 Instrukcje

Język składa się z następujących instrukcji:

- instrukcja blokowa – ciąg instrukcji zamknięty w klamrach {}
- deklaracja zmiennej
- wywołanie funkcji
- instrukcja wejściowa `read`
- instrukcja wyjściowa `print`
- instrukcja powrotu `return`
- instrukcja przypisania

	Grupa	Operator	Symbol	Łączność
0	nawiasy		(...)	
1	unarne	minus unarny	-	prawostronna
		negacja logiczna	!	
2	multiplykatywne	mnożenie	a*b	lewostronna
		dzielenie	a/b	
3	addytywne	dodawanie	a+b	
		odejmowanie	a-b	
4	relacje		<=, >=, <, >	
5	nie_równość		==, !=	
6	logiczne	AND	&&	
7		OR	||	

Tabela 1: Pierwszeństwo operatorów języka C--.

- instrukcja warunkowa `if`
- pętla `while`
- rzucenie wyjątku `throw`
- `try-catch-finally`

Wyrażenie samo w sobie nie stanowi instrukcji.

3 Gramatyka

Poniżej znajduje się serce języka – czyli gramatyka interpretowanego przez niego języka wraz z tokenami. Struktura języka jest, w porównaniu do możliwości stosowanych w praktyce współczesnych języków programowania, bardzo nieskomplikowana.

```
IDENTIFIER      [a-zA-Z][a-zA-Z0-9]*
INT_CONST       0|([1-9][0-9]*)
STRING          ".*"

"void", "int",  "return", "if", "else", "while", "break"
"try", "catch", "finally", "throw", "when", "Exception",
"(", ")", "{", "}", "-", "+", "*", "/", "||", "&&",
"<", ">", "=", "!", "<=", ">=", "==", "!=", ";", ",", "."

=======================================
program             : { function } EOF
                    ;

function            : ( "void" | type ) IDENTIFIER param_list block
                    ;

type                : "int"
                    ;

param_list          : "(" [ type IDENTIFIER { "," type IDENTIFIER } ] ")"
                    ;

block               : "{" { statement } "}"
                    ;

statement           : simple_statement ";"
                    | block_statement
                    ;
simple_statement    : declaration
                    | return
```

```
                    | "break"
                    | throw
                    | assignment
                    | function_call
                    ;
block_statement     : if
                    | while
                    | try_catch_finally
                    ;

declaration         : type declOptAssign { "," decl_opt_assign }
                    ;
decl_opt_assign     : IDENTIFIER [ "=" expression ]
                    ;

return              : "return" [ expression ]
                    ;

assignment          : IDENTIFIER "=" expression
                    ;

function_call       : IDENTIFIER "(" [ expression { "," expression } ] ")"
                    ;

if                  : "if" "(" expression ")" statement [ "else" statement ]
                    ;

while               : "while" "(" expression ")" statement
                    ;

// ================= EXCEPTIONS =================
throw               : "throw" expression
                    ;

try_catch_finally   : "try" statement
                        catch { catch }
                        [ "finally" statement ]
                    ;
catch               : "catch" [ "Exception" IDENTIFIER ] [ "when" expression ] statement
                    ;

// ================= OPERATORS =================
expression          : logical_or
                    ;

logical_or          : logical_and { "||" logical_and }
                    ;

logical_and         : in_equality { "&&" in_equality }
                    ;

in_equality         : relation [ ( "==" | "!=" ) relation ]
                    ;

relation            : additive [ ( "<=" | ">=" | "<" | ">" ) additive ]
                    ;

additive            : multiplicative { ( "+" | "-" ) multiplicative }
                    ;

multiplicative      : unar { ( "*" | "/" ) unar }
```

```
                    ;

unar                : [ "-" | "!" ]
                      ( ( "(" expression ")" ) | atomic )
                    ;

atomic              : const
                    | IDENTIFIER
                    | function_call
                    | string
                    ;

string              : STRING { "+" ( STRING | const ) }
                    ;
```

4 Struktura rozwiązania

Rozwiązanie zostało zrealizowane w środowisku Visual Studio 2022. Struktura rozwiązania składa się z następujących modułów:

- *Scanner*
- *Parser*
- *Interpreter*
- Moduł obsługi błędów

4.1 Scanner

Scanner jest klasą implementującą interfejs `IScanner`, który z kolei dziedziczy po interfejsie `IEnumerator<Token>` – innymi słowy, implementuje wzorzec iteratora dla `Token`a. Przyjmuje w konstruktorze strumień zawierający źródło programu i leniwie przez niego przechodzi, zmieniając `Current` na następny token za każdym wywołaniem metody `MoveNext()`. W przypadku napotkania nieznanego tokenu, `Current` jest ustawiany na specjalny token `Error`.

`Scanner` został wyposażony w pomocnicze klasy `PositionTrackingTextReader`, `LimitedStringBuilder`, `CommentsFilterScanner` porządkujące rozwiązanie.

4.1.1 MoveNext

Metoda `MoveNext` zwraca wartość `bool` informującą o tym, czy udało się przejść dalej w strumieniu. Nie mówi to jednak nic o poprawności tokenu - w przypadku rozpoznania nieznanego tokenu (`error`), `MoveNext` zwróci wartość `true`. Wartość `false` zostanie zwrócona tylko w przypadku zakończenia czytania strumienia. Oznacza to, że bezpośrednio po napotkaniu znaku `EOF`, zostanie zwrócone `true`, a każde kolejne wywołanie metody `MoveNext` zwróci false.

4.1.2 Reset, Dispose

Metody `Reset` oraz `Dispose` są wynikiem implementacji interfejsu `IEnumerator<Token>` i nie należy ich używać. W szczególności metoda `Reset` rzuca wyjątek, a metoda `Dispose` nic nie robi.

4.1.3 PositionTrackingTextReader

Dekorator wokół klasy abstrakcyjnej `TextReader` śledzący pozycję strumienia na podstawie przeczytanych do tej pory znaków. Udostępnia ponadto dodatkowe metody, ułatwiające pomijanie części wejścia:

- `SkipWhitespaces`
- `SkipLettersAndDigits`
- `SkipDigits`
- `SkipCurrentLine`
- `SkipToQuoteOrNewline`

4.1.4 LimitedStringBuilder

Dekorator wokół klasy `StringBuilder`, który ma ograniczoną pojemność, przekazaną mu jako argument konstruktora. Posiada metodę `Append`, która na podstawie próbowanego do dodania znaku, zwraca informację, czy dodanie go się udało, czy nie.

4.1.5 CommentsFilterScanner

Dekorator wokół `IScanner`, którego zadaniem jest filtrowanie `Token`ów zwracanych przez dekorowaną instancję i przepuszczanie wszystkich poza `Token.Comment`.

4.2 Parser

Parser implementuje interfejs `IParser`. W argumencie konstruktora przyjmuje instancję klasy implementującej interfejs `IScanner` oraz instancję modułu obsługującego błędy. Wystawia metodą `TryParse`, która zwraca 2 wartości:

- `bool` - wartość zwracana podstawowa, informująca o tym, czy parsowanie udało się bez żadnych błędów. W razie napotkania jakichkolwiek, są one przesyłane do obiektu `errorHandler`, po czym parsowanie jest, w miarę możliwości oraz przy bogatych założeniach upraszczających, kontynuowane. Pole pomocnicze `error` informuje o tym, czy został napotkany jakikolwiek błąd w trakcie wykonywania parsowania.
- `Program` - korzeń abstrakcyjnego drzewa składni powstałego w wyniku parsowania. Zwracany jako parametr `out` metody. W przypadku napotkania jakichkolwiek błędów, metoda może poprzez ten parametr zwrócić drzewo AST, którego stan może być niepoprawny (np. część poddrzewa może być nieprawidłowo `null`em).

4.2.1 Nodes

Do tworzenia drzewa składni programu stworzone hierarchię klas reprezentującą gramatykę języka. Każdy węzeł drzewa musi implementować interfejs `INode`. Dla uporządkowania, węzły drzewa zostały podzielone na foldery:

- *Statements*, który z kolei dzieli się na *Simple Statements* i *Block Statements*. Wszystkie implementują interfejs `IStatement`, który oznacza, że dana instrukcja może być częścią instrukcji blokowej.
- *Expressions* - grupuje wszystkie wyrażenia, tj. takie węzły programu, którym można wyewaluować jakąś wartość. Implementują (pusty) interfejs `IExpression`. Wśród nich wyróżnić można operatory binarne, które, z uwagi na swoją bardzo podobną konstrukcję, wszystkie dziedziczą po klasie abstrakcyjnej `BinaryOperator`.

4.3 Interpreter

Klasa implementująca interfejs `INodeVisitor`, która przechodzi po drzewie składni i je interpretuje. Jako parametry konstrukcji wstrzykiwane są 3 obiekty:

- `IErrorHandler` - jego rola jest taka sama, jak dla poprzednich modułówm
- `TextReader stdIn` - standardowe wejście programu, potrzebne dla instrukcji `read`
- `TextWriter stdOut` = standardowe wyjście program, potrzebne dla instrukcji `print`

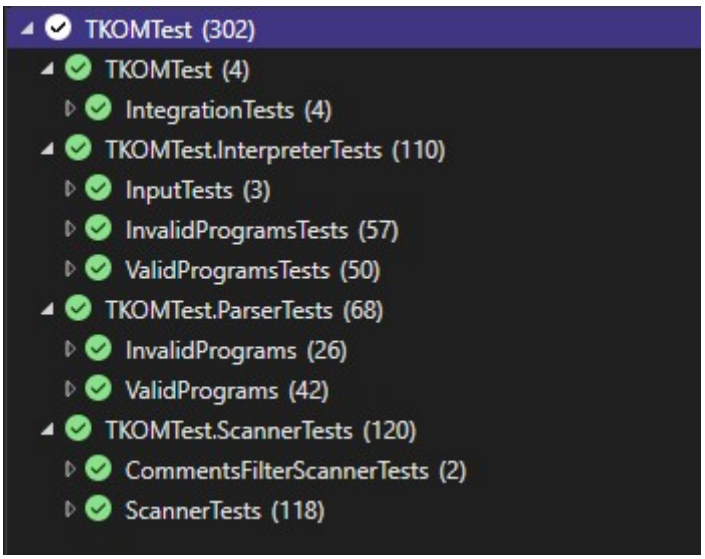
Główną częścią stanu Interpretera jest właściwość `CallStack`, który przchowuje stos kontekstów wywołań funkcji - obiektów klasy `FunctionCallContext`.

4.3.1 FunctionCallContext

Klasa reprezentująca kontekst wywołania funkcji. Posiada referencję do funkcji, której wywołanie reprezentuje oraz listę `Scope`ów, która w trakcie interpretacji zmienia się dynamicznie. Po zakończeniu funkcji dana instancja FCC jest zdejmowana ze stosu znajdującego sie w Interpreterze. Do ramach aktualnego kontekstu można stworzyć nowy lub usunąć najmłoszy `Scope`, dodać lub usunąć zmienną w ramach ostatniego `Scope`'u, lub znaleźć (a przynajmniej spróbować znaleźć) zmienną o danej nazwie.

4.3.2 Scope

`Scoper` reprezentuje obszar programu, który w użytecznym uproszczeniu można zamknąć między dwoma nawiasami klamrowymi. Wszystkie zmienne zadeklarowane w tym obszarze przestają być dostępne po wyjściu z niego.



Rysunek 1: Podsumowanie testós

4.3.3 Function

Klasa abstrakcyjna reprezentująca funkcję - zbiera w sobie zwracany typ, nazwę oraz listę parametrów. Realizuje wzorzec wizytatora z `Interpreter`em. Najważniejszą implementacją tej klasy jest `UserFunction`, która reprezentuje funkcję napisaną przez programistę. Ponadto zostały zaimplementowane klasy `PrintFunction` i `ReadFunction`, które reprezentują odpowiednio funkcje wbudowane `print` i `read`. Dodając nową funkcją wbudowaną do języka należy stworzyć dla niej nową klasę dziedziczącą po `Function` i tym samym zaimplementować metodę w `Interpreter`ze akceptującą wizytację. Dzięki takiej hierarchii `Interpreter` może przechowywać wszystkie funkcje pod jedną kolekcją.

4.3.4 FunctionsCollection

Klasa pomocnicza ułatwiająca zarządzanie zbiorem funkcji znajdujących się w stanie `Interpreter`a. Oprócz uzyskania funkcji z kolekcji metodą `TryGet`, możliwe jest dodanie nowej funkcji do kolekcji na dwa sposoby. Metoda `TryAdd` próbuje dodać funkcję do kolekcji i zwraca informację, czy dodanie się udało. W razie niepowodzenia, stan jest niezmieniony. Metoda `Add` próbuje dodać funkcję bezwarunkowo. W razie niemożliwości dodania funkcji do kolekcji, rzucony jest wyjątek `InvalidOperationException`. Bezwarunkowe dodanie jest przydatne w przypadku, gdy interpreter dodaje do kolekcji funkcje wbudowane. Jeżeli dodaje 2 różne funkcje wbudowane, które są wywoływane w taki sam sposób - robi coś źle i należy go o tym uświadomić wyjątkiem.

4.4 Testy

Na wszystkich modułach zostały przeprowadzone testy jednostkowe przy użyciu biblioteki xUnit. W każdym przypadku zostały one podzielone na 2 pliki, z których w jednym pliku znajdują się testy sprawdzające pozytywną ścieżkę, a w drugim sprawdzane są programy niepoprawne. Klasy testujące zazwyczaj dziedziczą po wspólnej klasie bazowej zawierającą krótką wspólną część logiki.

Napisano łącznie 302 testów, z czego 120 dla skanera, 68 dla Parsera oraz 110 dla interpretera. Ponadto są 4 testy integracyjne.