Politechnika Warszawska, WEiTI

Techniki kompilacji

Dokumentacja Końcowa

Interpreter własnego języka z definiowaniem typu wyliczeniowego oraz obsługą zdarzeń wyjątkowych

Autor: Halyna Polekha (nr alb. 294866)

1 Opis projektu

Projekt ma na celu wykonanie intepretatora prostego języka z definiowaniem typu wyliczeniowego oraz z wbudowanym mechanizmem obsługi zdarzeń wyjątkowych, umożliwiającym także filtrowanie wyjątków.

2 Opis zakładanej funkcjonalności

Przyjęte zostały następujące zasady:

- Język własny jest imperatywnym, funkcyjnym, językiem programowania, zgodnym z paradygmatem programowania proceduralnego,
- Istnienie możliwości tworzenia, definiowania oraz używania zmiennych,
- Brak zmiennych globalnych,
- Jedyne elementy składni języka które mogą być globalne to definicje funkcji oraz definicje typów wyliczeniowych,
- Istnienijące typy wbudowane:
    - Typ liczbowy całkowity w zakresie od -2,147,483,447 do 2,147,483,447,
    - Typ znakowy o długości do 2048,
    - Typ wyliczeniowy, zawierający listę wartości o maksymalnej długości 500,
- Istnieje możliwości definiowania własnych funkcji oraz ich późniejszego wywołania,
- Punktem wejścia programu jest zawsze funkcja *int main()*,
- Istnieje możliwość przeprowadzenia operacji arytmetycznych oraz warunkowych,
- Możliwe jest wykonanie operacji konkatenacji oraz porównanie danych typu znakowego,
- Mechanizm obsługi wyjątków pozwala na zgłoszenie sytuacji wyjątkowej w dowolnym miejscu kodu,
- Także dla dowolnej części kodu możliwe jest zdefiniowanie bloku obsługi, który przechwytuje określone rodzaje wyjątków z możlowością filracji,
- Filtrowanie wyjątków jest możliwe po wartości oraz po zakresie wartości
- Maksymalna długość identyfikatora wynosi 25 znaków,
- Odpowiednikiem zmiennej binarnej są znaczenia 1 (prawda) i 0 (falsz),

- Wykorzystanie znaków specjalnych (takich jak apostrof) w łańcuchu znaków wymaga poprzedzenie ich ukośnikiem.

3 Przykład obrazujący dopuszczalne konstrukcje językowe

Poniżej przedstawiono przykład użycia języka w realizacji operacji sprawdzenia czy podana przez użytkownika liczba jest liczbą prostą.

```
enum Exception
{
   ArgumentOutOfRange = 123;
   InvalidOperation = 124;
}

enum Boolean
{
    False = -1;
    True = 1;
}

Boolean isPrime(int n)
{
    if(int n <= 0)
    {
      throw Exception.ArgumentOutOfRange;
    }

    if (n == 1)
    {
        return Boolean.False;
    }
    int i = 2;
    while (i < n)
    {
        if (n % i == 0)
        {
            return Boolean.False;
        }
        i = i + 1;
    }

    return Boolean.True;
}

void main()
```

```
{
    print("Please enter a natural number to check whether it is a prime
number or not.");
    int number;
    Boolean result;
    try{
        number = read();
        Result = isPrime(number);
        print(result.Name);
    }
    catch ([Exception.ArgumentOutOfRange ...
Exception.InvalidOperation])
    {
        if(errno ==Exception.ArgumentOutOfRange)
        {
            print("Argument is out of range.");
        }
        if(errno == Exception.InvalidOperation)
        {
            print("Invalid operation.");
        }
    }
    return 0;
}
```

4 Formalna specyfikacja i składnia

4.1 Gramatyka języka

```
program = {enumDefStatement | funDefStatement};

enumDefStatement = "enum", correctName, "{", enumDefArg, {enumDefArg}
,"}";
correctName = letter, {symbol};
letter = "a" | "b" | "c" | ... | "y"  | "z" | "A" ... "Z";
symbol = letter | digit | "_";
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
enumDefArg = correctName, [ "=", number];
number = digit, {digit};

funDefStatement = (dataType | "void"), correctName, "(", [funDefArg
{",", funDefArg}], ")", block;
```

```
dataType = "int" | "string" | correctName;
funDefArg = dataType, correctName;
block = "{", {statement, ";"} ,"}";
statement = varInitStatement | varAssignStatement | ifElseStatement |
loopStatement | funCallStatement | returnStatement | throwStatement |
tryCatchStatement;

varInitStatement = dataType, correctName, ["=", condition];
varAssignStatement = correctName, "=", condition;
ifElseStatement = "if", "(", condition, ")" , block, ["else", block];
loopStatement = "while", "(", condition, ")", block;
funCallStatement = correctName, "(", [condition, {",", condition}],")";
returnStatement = "return", condition;
throwStatement = "throw", condition;

tryCatchStatement = tryBlock, catchBlock, {catchBlock};
tryBlock =  "try", block;
catchBlock = "catch", "(",[catchFilter] ,")", block;
catchFilter = condition | filterByRange;
filterByRange = "[", condition ,"...", condition ,"]";

condition = andCondition,{ "||", andCondition};
andCondition = relationCondition, {"&&", relationCondition};
relationCondition = primaryConditionKey , [relationOperator,
primaryConditionKey];
relationOperator = "<" | "<=" | ">" | ">=" | "==" | "!=";
primaryConditionKey = ["!"], arithmeticStatement;

arithmeticStatement = addArg, {("+"| "-"), addArg};
addArg = arithmArg, {multOperator, arithmArg};
multOperator = "*" | "/" | "%";
arithmArg = text | ( ["-" | "+"], (number | enumArgNameExtended |
correctName | ("(", condition, ")") | funCallStatement));
text = "'", anyCharacter, "'";
anyCharacter = ? all characters ?;
enumArgNameExtended = correctName, ".", correctName;
```

5 Wymagania

5.1 Wymagania funkcjonalne

- Odczytywanie, parsowanie i analiza plików źródłowych
- Kontrola poprawności wprowadzonych danych oraz odpowiednie zgłaszanie błędów wykrytych podczas kolejnych etapów analizy.
- Poprawne wykonywanie wszystkich poprawnie zapisanych instrukcji w plikach źródłowych

- Przestrzeganie logicznego porządku instrukcji sterujących
- Przeprowadzenie operacji arytmetycznych zgodnie z ich priorytetem
- Wyświetlanie wyników wykonania na standardowym wyjściu terminala

5.2 Wymagania niefunkcjonalne

- Proste i przejrzyste komunikaty o błędach analizy kodu wejściowego

6 Obsługa błędów

W przypadku zgłoszenia błędu przez program interpretera na jednym z etapów zostanie wyświetlony odpowiedni komutikat informujący użytkowika o rodzaju oraz miejscu wystąpienia błędu w pliku interpretowanym, np

```
(13:5) SemicolonExpectedException: Semicolon is expected
```

gdzie w nawiasach jest podana informacja o wierszu i kolumnie w którym wystąpił błąd, nastempnie rodzaj błędu oraz odpowiedni komunikat.

Wyjątki zgłaszane przez program interpretera należą do jednej z następujących grup:

- *Errors* - wyjątki zgłaszane przez moduł analizatora leksykalnego;
- *Exceptions* - wyjątki zgłaszane przez moduł analizatora składniowego;
- *LogicExceptions* - wyjątki zgłaszane przez modół interpretera.

7 Sposób uruchomienia

Interpreter własnego języka został zaimplementowany w języku C# jako aplikacja konsolowa, która jest wywoływana wraz z argumentem określającym ścieżkę do pliku z programem, który będzie interpretowany.

7.1 Dane wejściowe

Instrukcje zapisane w interpretowaym języku są wczytywane z pliku wejściowego.

7.2 Dane wyjściowe

Wynik programu oraz ewentualne błędy są wyświetlane na standardowym wyjściu.

8 Sposób realizacji

Program interpretera jest złożony z modułów odpowiedzialnych za kolejne etapy analizy plików wejściowych, a także z dodatkowych modułów pomocniczych, wspomagających cały proces.

9.1 Moduły

8.1.1 Analizator leksykaly

Modół analizatora leksykalnego jest odpowiedzialny za czytanie programu źródłowego i przekształcenie wejściowego ciągu znaków w sekwęcję atomów leksykalnych.

8.1.1.1 Lista definiowanych tokenów

Kod tokenu	**Token**
T_LBRACES	“{“
T_RBRACES	“}”
T_LPARENT	“(“
T_RPARENT	“)”
T_LSBRACKET	“[“
T_RSBRACKET	“]”
T_IDENTIFIER	identyfikator
T_MAIN	“main”
T_INTCONST	liczba
T_TEXT	Ciąg znaków
T_ASSIGN	“=”
T_ENUM	“enum”
T_INT	“int”
T_STRING	“string”
T_IF	“if”
T_WHILE	“while”
T_ELSE	“else”
T_EQUAL	“==”
T_NEQUAL	“!=”
T_MORETHAN	“>”
T_LESSTHAN	“<”
T_MOREEQUAL	“>=”

T_LESSEQUAL	“<=”
T_THROW	“throw”
T_DOT	“.”
T_ELLIPSIS	“...”
T_SEMICOLON	“;”
T_RETURN	“return”
T_MODULO	“%”
T_VOID	“void”
T_TRY	“try”
T_CATCH	“catch”
T_MINUS	“-”
T_PLUS	“+”
T_DIV	“/”
T_MUL	“*”
T_AND	“&&”
T_OR	“||”
T_NOT	“!”

8.1.1.2 Interface

Modół analizatora leksykalnego jest zrealizowany jako klasa publiczna *Scanner*, do której jest przekazywany objekt pomocniczej klasy *Source*, zawierający kod źródłowy. Za wczytania kolejnego tokena z pliku źródłowego odpowiada metoda *Token GetNextToken()*.

8.1.2 Analizator składniowy

Modół analizatora składniowego jest odpowiedzialny za grupowanie atomów leksykalnych w struktury składniowe oraz jednocześnie sprawdzenie czy tworzone konstrukcję są zgodne z gramatyką języka.

8.1.2.1 Wykaz struktur składniowych

Po analizie składniowej zostaje stworzona odpowiednia struktura danych (drzewo), składająca się z następujących struktór składniowych:

- MainProgram;
- FunDefStatement;
- FunDefArgs;

- EnumDefStatement;
- EnumDefArgs;
- Block;
- Statement;
- VarDefInitStatement;
- VarAssigbStatement;
- ThrowStatement;
- ReturnStatement;
- LoopStatement;
- IfElseStatement;
- FunCalltatement;
- TryCatchStatement;
- CatchFilter;
- Catch;
- RelatioCondition;
- PrimaryConditionKay;
- PrimaryConditionKayArithm;
- Expression;
- Condition;
- AndCondition;
- ArithmeticStatement
- MultOperator;
- ArithmeticArg;
- ArgVariable;
- ArgText;
- ArgNumber
- ArgFunCallStatement;
- ArgEnumNameExtended;
- ArgEnumeratorName;
- ArgEnumeratorValue;
- ArgCondition;
- AddOperator;
- AddArg.

8.1.2.2 Interface

Modół analizatora składniowego jest zrealizowany jako klasa publiczna *Parser,* przyjmująca objekt analizatora leksykalnego. Za przetworzenie strumienia tokenów w odpowiednią strukturę danych odpowiada metoda *Token ParseProgram()*.

8.1.3 Interpreter

Wykonuje program dostarczony przez analizator składniowy w postaci sekwencji instrukcji.

8.1.3.1 Środowisko

Wykorzystany został pomocniczy podmodół Interpretera *Environment* zarządzający kontekstem wywołań za pomocą mechanizmu tworzenia kontekstów wywołań i kontekstów blokowych.

W danym kontekście wywołania nie jest dozwolone tworzenie dwóch zmiennych o takiej samej nazwie. Obiekty języka przekazywane są przez kopie.

Interface środowiska

Środowisko *Environment* posiada następujące właściwości oraz medody do zarządzania nimi:

- do zapisu wyników: object *lastResult*;
- do przechowywania typu zapisanego w *lastResult* wyniku: DataType *lastResultType*;
- flaga powrotu: bool *returnFlag*;
- kolekcja zdefiniowanych funkcji: Dictionary<string, FunDefStatement> *globalFunctions*;
- kolekcja zdefiniowanych obiektów typów wyliczeniowych: Dictionary<string, EnumDefStatement> *globalEnums*;
- stos kontekstów wywołania: Stack<CallContext> *callContexts*;
- flaga rzucenia wyjątku: bool *throwFlag*;
- do zapisu rzuconej wartości: ErrnoObject *errno*;

Dodatkowe metody środowiska *Environment* sterujące kontekstem wywołań:

- tworzenie i usunięcie kontekstu wywołania: *bool CreateCallContext()* i *bool DeleteCallContext()*;
- tworzenie i usunięcie kontekstu bloku: *public void CreateBlockContext()* i *bool DeleteBlockContext()*;
- tworzenie nowych zmiennych: *bool CreateVariable(Variable variable)*;
- wyszukiwnie zmiennej: *Variable GetVariable(string name)*;
- wyszukiwanie funkcji: *FunDefStatement GetGlobalFunction(string name)*

8.1.3.1 Interface

Modół interpreter został zrealiziwany jako klasa publiczna *Interpreter*, do której zostaje przekazany objekt drzewiasty *MainProgram*, stworzony przez analizator składniowy. Architektura modułu została oparta o wzorzec *INodeVisitor*.

Metoda *void Execute()* odpowiada za wykonanie programu.

9 Opis sposobu testowania

Do testowania działania programu interpretera języka własnego zostały sporządzone testy, sprawdzające poprawność działania każdego modułu programu.

- Dla testowania modułu analizatora leksykalnego są stworzone testy jednostkowe.
- Testy jednostkowe są także stosowane do sprawdzania poprawności działania analizatora składniowego.
- Metoda testowania manualnego została zastosowana w przypadku modułu interpretera.