AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ INFORMATYKI, ELEKTRONIKI I TELEKOMUNIKACJI
KATEDRA INFORMATYKI

Języki formalne i kompilatory

Temat projektu:
Translator podzbioru języka Java do C

Autor: Piotr Skurski

1. Gramatyka

```
compilation
   :   packageDeclaration? importDeclaration* typeDeclaration*
       EOF
   ;

packageDeclaration
   :   'package' qualifiedName ';'
   ;

importDeclaration
   :   'import' 'static'? qualifiedName ('.' '*')? ';'
   ;

typeDeclaration
   :   classModifier* ( classDeclaration )
   |   ';'
   ;

classModifier
   :   'public'
   |   'protected'
   |   'abstract'
   ;

classDeclaration
   : 'class' className
     '{' classBodyDeclaration '}'
   ;

classBodyDeclaration
   :   member*
   ;

member
   : methodDeclaration
   | fieldDeclaration
   ;

methodDeclaration
   :   modifier? type methodName methodParameters ('[' ']')* methodBody
   ;
```

```
fieldDeclaration
    :   modifier? constant? variableDeclarator ';'
    ;

modifier
    : 'public'
    | 'private'
    | 'protected'
    ;

constant
    : 'static final'
    | 'final static'
    ;

methodParameters
    :   '(' methodParametersDeclaration? ')'
    ;

methodParametersDeclaration
    :   type variableName (',' methodParametersDeclaration)?
    ;

methodBody
    : '{' instruction+ '}' ;

variableDeclarator
    :   type variableName ('=' variableInitializer)?
    ;

instruction
    :   variableDeclarator ';'
    |   statement
    ;

variableInitializer
    :   expression
    |   '\'' expression '\''
    ;

type
    :   primitiveType ('[' ']')*
    |   'void'
    ;
```

```
statement
   :  'return' expression? ';'                                          #returnStatement
   |  'if' parExpression '{' statement '}' ('else {' statement '}')?    #condStatement
   ;

expression
   :  literal                                                           #literalExpression
   |  expression op=('+'|'-'|'*'|'/') expression                        #calcExpression
   |  expression op=('=='|'!=') expression                              #equalityExpression
   ;

parExpression
   :  '(' expression ')'
   ;

literal
   :  INT
   |  variableName
   |  'null'
   ;

primitiveType
   :  'char'
   |  'short'
   |  'int'
   |  'long'
   |  'float'
   |  'double'
   ;

className : ID ;
methodName : ID ;
qualifiedName : ID ;
variableName : ID ('[' ']')* ;

INT : [0-9]+ ;
ID : [a-zA-Z0-9\.]+ ;
WS: [ \t\n\r]+ -> skip ;
```

2. Zdefiniowane stałe słownikowe

a) Operatory
   - dodawanie '+'
   - odejmowanie '-'
   - mnożenie '*'
   - dzielenie '/'
   - przypisania '='
   - średnik ';'
   - przecinek rozdzielający parametry metody ','
   - logiczny, równości '=='
   - logiczny, nierówności '!='

b) Nawiasy
   - otwierający '('
   - zamykający ')'
   - otwierający dla klas, metod, bloków '{'
   - zamykający dla klas, metod, bloków '}'
   - otwierający dla tablic '['
   - zamykający dla tablic ']'

c) Instrukcje
   - 'if'
   - 'else'
   - 'return'

d) Modyfikatory
   - 'public'
   - 'private'
   - 'protected'
   - 'static final'
   - 'final static'
   - 'Abstract'

3. Przykładowy kod w języku Java

```
public class Demo {

    private float number = 44.66;

    private long bigNumber = 5432535932452;

    public final static int MAX = 999;

    public final static char FLAG = 'A';

    public double squareValue(double value) {
        return value * value;
    }

    public int addTo(int value) {
        int add = 1000;
        return value + add;
    }

    private int multiply(int x, int y) {
        int temp = x;
        int z = 1000;

        if (x == y) {
            return z;
        }

        return temp;
    }

    void doNothing() {
        float sign = 45.99;
    }

    float calculate(int value) {
        if (value != 100) {
            return 50.55;
        } else {
            return 0.0;
        }
    }
}
```

4. Opis typizacji tłumaczonego języka

W stworzonym translatorze dostępne są:

- znaki zdefiniowane w stałych słownikowych
- typy prymitywne z języka Java a więc liczby całkowite i zmiennoprzecinkowe: (char, short, int, long, float, double) oraz void
- zmienne - mogą zawierać duże i małe litery, cyfry oraz kropki (muszą zaczynać się od litery)

5. Uzasadnienie wyboru generatora parserów

Wybrałem generator parserów ANTLR (wersja 4) dlatego że jest zaimplementowany w języku Java, posiada interfejs w języku Java oraz jest dobrze udokumentowany w internecie. Jak również jest stosunkowo prosty w użyciu, wymaga jedynie napisania gramatyki a na jej podstawie narzędzie już samo generuje potrzebne klasy (parser, lexer, visitor, listener).

6. Opis napotkanych problemów

a) Wybór wzorca - sposobu przejścia przez drzewo - Listener albo Visitor. Obydwa udostępniają podobną funkcjonalność ale Visitor jest bardziej czytelny i generyczny. Listener wymaga tworzenia dodatkowych pól w klasie które by przetrzymywały dane zbierane podczas przechodzenia przez drzewo a później pobierania tych danych przez gettery. Visitor bezpośrednio zwraca interesujące nas dane, wymuszając również konieczność stworzenia własnego modelu dla poszczególnych danych.

b) Rozpoznanie konkretnej reguły z gramatyki - np. w deklaracji "expression" mamy kilka reguł, skąd będziemy wiedzieć która reguła została zastosowana, jednym z rozwiązań jest sprawdzanie jaka operacja wystąpiła (np. + / -) co wiąże się ze stosowaniem instrukcji warunkowych, prostszym i bardziej czytelnym rozwiązaniem dla mnie było zastosowanie etykiet. Do każdej reguły w "expression" dołączona jest etykieta dzięki temu parser tworzy osobną metodę dla każdej reguły a więc każdej regule odpowiada dokładnie jedna metoda w klasie ExpressionVisitor. To czyni kod bardziej zrozumiałym i czytelnym, łatwiejszym w debugowaniu i wprowadzaniu zmian.

c) Jak sobie poradzić w przypadku instrukcji zagnieżdżonych wielokrotnie - np. w deklaracji "statement" mamy reguły w których występuje "expression" oraz kolejne "statement". Z pomocą przyszedł wzorzec Visitor, przy przetwarzaniu konkretnej metody możemy dla instrukcji zagnieżdżonej wywołać rekursywnie tą samą klasę (StatementVisitor), która sama rozpozna jaka reguła jest zastosowana w instrukcji zagnieżdżonej. Natomiast w modelu, w klasie Statement zastosowałem wzorzec Kompozyt a więc obiekt statement może być obiektem prostym jak i zawierać w sobie kolekcję obiektów statement.

d) Gdzie dokonać translacji? Visitor pozwala nam przemierzyć całe drzewo ale w którym miejscu dokonać translacji. Ja stworzyłem modele dla interesujących mnie danych i te modele przechowują dane oraz implementują interfejs Printer z metodą print() w której następuje translacja z Javy na C. Klasa która wywołuje translację to klasa Output.

Link do repozytorium:
https://github.com/skurski/antlr-java2c-translator

Bibliografia:
"The Definitive ANTLR 4 Reference" Terence Parr
http://jakubdziworski.github.io/java/2016/04/01/antlr_visitor_vs_listener.html
https://stackoverflow.com/questions/29971097/how-to-create-ast-with-antlr4