



JPS Ćw 8

Parser

W ramach naszego projektu trzeba będzie wysyłania do serwera zapytań w postaci ciągu znaków. Na przykład „Osoba where imie=”Maciej””. Dotychczas doszliśmy do poziomu, gdzie jesteśmy w stanie wywołać wyliczenie zapytania gdy jest ono przedstawione w postaci struktury – drzewa AST.

Parser posłuży nam do „załatania” tej dziury, czyli dokona transformacji zapytania z ciągu znaków do drzewa AST.

Możliwe są dwa rozwiązania tego problemu. Wybór zależy od technologii wybranej (w Javie rozwiązanie pierwsze jest łatwiejsze niż w .NET czy Python -ie). Drugie jest łatwiejsze do zrozumienia, lecz wymagać będzie prawdopodobnie więcej czasu na zaprogramowanie

IMPLEMENTACJA I LEX + CUP

Rozwiązanie to oparte jest o dwa narzędzie dostępne dla Javy (dla .NET są ale słabe i stwarzają problemy) czyli lekser i parser. Należy przygotować dwa pliki, lexer.lex oraz parser.cup.

Lexer.lex zawierać będzie definicje w postaci wyrażeń regularnych definiujących słowa kluczowe jakie mogą się pojawić w zapytaniu, specjalne znaki (nawiasy, średniki, operatory). Przykład jak to wygląda dla Odry można zobaczyć pod adresem

<https://dev.kis.p.lodz.pl:8443/svn/egovbus/ODRA2/trunk/res/lexer.lex>

Oczywiście zawiera on znacznie więcej operatorów i słów kluczowych niż będą nam potrzebne ale pozwala zrozumieć zasadę powstawania leksera.

Parser.cup zawiera „gramatykę” naszego języka czyli definicje w jaki sposób fragmenty (tokeny) zwrócone przez lekser mogą się łączyć w większą całość czyli nasze zapytanie. To w gramatyce definiowane jest że plus składa się z zapytania lewego, znaku „plus” i prawego zapytania.

```
additive_expr ::= expr:e1 PLUS:o expr:e2
```

Dodatkowo, w ramach przetwarzania wyrażenia należy zwrócić odpowiedni element drzewa AST, na przykład

```
RESULT = new PlusExpression(e1, e2);
```



Przykład pliku CUP można znaleźć pod adresem

<https://dev.kis.p.lodz.pl:8443/svn/egovbus/ODRA2/trunk/res/parser.cup>

Uruchomienie jlex i cup najłatwiej zrobić poprzez plik build.xml i narzędziem ANT (wbudowanym w Eclipse i NetBeans). Należy utworzyć target i umieścić w nim kod:

```
<target name="generate">
  <java jar="jflex.jar" fork="true" failonerror="true">
    <arg path="lexer.lex" />
    <arg value="--nobak" />
    <arg line="-d $parser" />
  </java>
  <java jar="cup.jar" fork="true" failonerror="true">
    <arg line="-symbols Symbols" />
    <arg line="-parser SBQLParser" />
    <arg line="-interface" />
    <arg line="-destdir parser" />
    <arg path="parser.cup" />
  </java>
</target>
```

IMPLEMENTACJA II

PARSOWANIE RĘCZNE, REKURENCYJNE

Sposób prostszy choć pewnie bardziej czasochłonny to zaprogramowanie parsowania ręcznie. Można stworzyć metodę, która rekurencyjnie wywoływana przeparsuje nasze zapytanie.

Prosty przykład radzący sobie z zapytaniami typu $1+2*3-4$ (kod w C#)

```
static Expression parse(string s)
{
    Regex r;
    Match m;

    r = new Regex(@"(?<left>.*)\-(?<right>.*)");
    m = r.Match(s);
    if (m.Success)
    {
        string left = m.Groups["left"].value.Trim();
        string right = m.Groups["right"].value.Trim();

        Minus ex = new Minus();
        ex.leftE = parse(left);
        ex.rightE = parse(right);

        return ex;
    }
    r = new Regex(@"(?<left>.*)\+(?<right>.*)");
    m = r.Match(s);
    if (m.Success)
    {
        string left = m.Groups["left"].value.Trim();
        string right = m.Groups["right"].value.Trim();

        Plus ex = new Plus();
        ex.leftE = parse(left);
        ex.rightE = parse(right);
    }
}
```



```
        return ex;
    }
    r = new Regex(@"(?<left>.*)\*(?<right>.*)");
    m = r.Match(s);
    if (m.Success)
    {
        string left = m.Groups["left"].value.Trim();
        string right = m.Groups["right"].value.Trim();

        Multiplication ex = new Multiplication();
        ex.leftE = parse(left);
        ex.rightE = parse(right);

        return ex;
    }

    Regex integer = new Regex("[0-9]+");
    if(integer.IsMatch(s))
    {
        int value = Int16.Parse(s);
        IntegerTerminal ex = new IntegerTerminal();
        ex.value = value;
        return ex;
    }

    return null;
}
```

Jak widzimy, przy pomocy wyrażeń regularnych definiujemy jak ma wyglądać zapytanie (na przykład coś plus coś) i wywołujemy przetwarzanie rekurencyjnie dla prawego i lewego podzapytania.

Kolejność w jakiej umieścimy bloki odpowiedzialne za operatory decydować będzie o ich priorytetach. To co powinno się znaleźć na dole drzewa (ma wysoki priorytet) należy umieścić na dole metody. Jak widać w naszym przykładzie, minus jest przed mnożeniem przez co najpierw będziemy mnożyć a potem odejmować, co jest zgodne z zasadami arytmetyki.

Rozgraniczenia można zaimplementować w postaci wyrażeń regularnych lub prościej przy pomocy operowania na łańcuchach na zasadzie `s.StartsWith`, `s.EndsWith`, `s.IndexOf` itp.

ZALICZENIE

Zaliczenie parsera polega na jego zaimplementowaniu i sprawdzeniu, że dla każdego zapytania zawierającego przydzielone operatory możliwe jest jego przetworzenie i wykonanie.

Należy się przygotować na dowolne zapytania od prowadzącego.

SPRAWY ORGANIZACYJNE

Pozostały nam dwa tygodnie zajęć. Na pierwszym naszym spotkaniu w Nowym Roku (19-21 stycznia) należy zaprezentować mini-projekt 6 czyli przydzielone operatory a na spotkaniu ostatnim (26-28 stycznia) parser i pełny projekt.