Wybrane narzędzia do tworzenia analizatorów leksykalnych i składniowych w Javie

Andrzej Augustynowicz

Wstęp

Przez ostatnie lata Java zdobyła dużą popularność w internecie. Bardzo duże podobieństwo do C++ dało podstawy do prostego przeportowania wielu lekserów i parserów do Javy.
Stało się tak w przypadku YACC (Yet Another Compiler Compiler), którego odpowiednikiem w Javie stał się CUP. Natomiast ANTLR jest odpowiednikiem PCCTS.
Jednak ze względu na to, że narzędzia te były pisane w zamyśle pod inny język programowania, nie potrafią teraz czerpać korzyści z wielu dodatnich aspektów stosowania Javy.
Prezentacja przedstawia zarówno leksery, parsery portowane z C/C++, jak również te pisane bezpośrednio pod Jave.

Spis niektórych narzędzi

- Chaperon - generator parserów i lekserów
- JavaCC - generator parserów i lekserów
- RunCC – generator parserów i lekserów
- SableCC - generator parserów i lekserów
- Beaver – generator parserów LALR
- CUP - generator parserów LALR
- JLex – generator lekserów
- JFlex – generator lekserów

JLex

- Napisany w Javie dla Javy – dostępny kod źródłowy
- Licencja open source
- Plik z opisem gramatyki podobny jak w Lexie
- Współpracuje z generatorem parserów CUP
- Generuje jako wynik klasę Yylex – klasa ta jest w pełni funkcjonalnym skanerem
- Niezbyt bogata dokumentacja

http://www.cs.princeton.edu/~appel/modern/java/JLex/

ostatnia aktualizacja projektu: 7 Luty 2003

JLex - przykład

```
class Sample {
    public static void main(String argv[]) throws java.io.IOException {
        Yylex yy = new Yylex(System.in);
        while (yy.yylex() != null);
    }
}

class Yytoken {}

%%

DIGIT=                  [0-9]
WHITESPACE=             [ \t\n]
%line

%%

{DIGIT}+                {System.out.println(yyline+1 + ": INT");}
"="                     {System.out.println(yyline+1 + ": ASSIGN");}
{WHITESPACE}*           { }
.                       {System.out.println(yyline+1 + ": bad char");}
```

JFlex

- Stanowi jakby rozwinięcie JLexa (pełna kompatybilność)
- Generacja szybszych skanerów
- Generacja różnego rodzaju skanerów (szybsze/mniejsze itp)
- Współpraca z generatorami parserów LALR:
  - CUP
  - BYacc
  - ANTLR
- Licencja GPL
- Rozbudowana dokumentacja

http://jflex.sourceforge.net
ostatnia aktualizacja projektu: 7 listopad 2004

JFlex - przykład

```
%%

%public
%class Subst
%standalone

%unicode

%{
  String name;
%}

%%

"name " [a-zA-Z]+   { name = yytext().substring(5); }
[Hh] "ello"         { System.out.print(yytext()+" "+name+"!"); }
```

Cup

- Popularny generator parserów dla Javy
- Tworzy parsery LALR
- Licencja open source
- Współpracuje z wieloma generatorami skanerów
- Rozbudowana dokumentacja

http://www2.cs.tum.edu/projects/cup/
ostatnia aktualizacja projektu: 16 maja 2005

Cup - przykład

```
import java_cup.runtime.*;

/* definicja funkcji służących do obsługi skanera */
init with {: scanner.init();              :};
scan with {: return scanner.next_token(); :};


/* definicja symboli terminalnych */
terminal           SEMI, PLUS, MINUS;
terminal Integer    Number

/* definicja nieterminali */
non terminal            expr_list, expr_part;

/* definicja pierwszeństwa */
precedence left PLUS, MINUS;
```

Cup - przykład

```
/* definicja gramatyki */

expr_list ::= expr_list expr_part | expr_part;
expr_part ::= expr SEMI;

expr      ::= expr PLUS expr
              | expr MINUS expr
              | NUMBER;
```

Po przetworzeniu ww. definicji przez parser cup uzyskujemy pliki sym.java oraz parser.java. Następnie definiujemy klase scanner i metody init oraz next_token.

```
import java_cup.runtime.*;

public class scanner {
  protected static int next_char;
  protected static void advance()
    throws java.io.IOException
    { next_char = System.in.read(); }

  public static void init()
    throws java.io.IOException
    { advance(); }
```

Cup - przykład

```
 /* rozpoznaje i zwraca token */
public static Symbol next_token()
  throws java.io.IOException
  {
    for (;;)
      switch (next_char)
        {
          case '0': case '1': case '2': case '3': case '4':
          case '5': case '6': case '7': case '8': case '9':
            /* parse a decimal integer */
            int i_val = 0;
            do {
              i_val = i_val * 10 + (next_char - '0');
              advance();
            } while (next_char >= '0' && next_char <= '9');
          return new Symbol(sym.NUMBER, new Integer(i_val));

          case ';': advance(); return new Symbol(sym.SEMI);
          case '+': advance(); return new Symbol(sym.PLUS);
          case '-': advance(); return new Symbol(sym.MINUS);

          case -1: return new Symbol(sym.EOF);

          default:
            advance();
          break;
        }}.
```

Cup - przykład

Ostatnim krokiem jest integracja z lekserem.

Dopisujemy linijki:

```
%implements java_cup.runtime.Scanner
%function next_token
%type java_cup.runtime.Symbol
```

do pliku *.lex (w przypadku Jlex),

dzięki czemu uzyskujemy działający skaner leksykalno-składniowy.

Beaver

- Tworzy parsery LALR
- Licencja open source
- Współpracuje z wieloma generatorami skanerów
- Uboga dokumentacja
- Dobra integracja z JFlex, JLex
- Akceptuje gramatykę w EBNF

http://beaver.sourceforge.net/

ostatnia aktualizacja projektu: 1 grudnia 2005

Beaver - przykład

```
%%
// definicja pierwszeństwa
%left RPAREN;
%left MULT, DIV;
%left PLUS, MINUS;

// definicja nieterminali
%typeof NUMBER = "Number";
%typeof expr = "Expr";

%%

// definicja gramatyki
expr = expr.a MULT  expr.b   {: return new Expr(a.value * b.value); :}
     | expr.a DIV   expr.b   {: return new Expr(a.value / b.value); :}
     | expr.a PLUS  expr.b   {: return new Expr(a.value + b.value); :}
     | expr.a MINUS expr.b   {: return new Expr(a.value - b.value); :}
     | NUMBER.n              {: return new Expr(n.doubleValue());   :}
     | LPAREN expr.e RPAREN  {: return e; :}
     ;
```

Beaver - przykład

- integracja z JFlex, większość funkcji jest generowana automatycznie przez parser

```
import beaver.Symbol;
import beaver.Scanner;

import example.ExampleParser.Terminals;

%%

%class LanguageScanner
%extends Scanner
%function nextToken
%type Symbol
%yylexthrow Scanner.Exception
%eofval{
        return new Symbol(Terminals.EOF, "end-of-file");
%eofval}

// ...

%%
```

SableCC

- Lekser obsługujący w pełni unicode
- Obsługa syntaktyki EBNF
- Parser LALR(1)
- Automatyczne rozwiązywanie konfliktów LALR(1)
- Intuicyjna składnia
- Pełna dokumentacja
- Dostarcza abstrakcyjne drzewo składniowe

http://sablecc.org/

ostatnia aktualizacja projektu: 24 grudnia 2005

SableCC - przykład

Poniższy lekser ilustruje zastosowanie pomijania komentarzy /* */ w kodzie

```
Helpers
  all = [0 .. 0xffff];
  letter = [['a' .. 'z'] + ['A' .. 'Z']];

States
  normal,
  comment;

Tokens
{normal} blank = (' ' | 10 | 13 | 9)*;
{normal} identifier = (letter | '_')*;

{normal->comment, comment}
          comment = '/*';

{comment} comment_end = '*/';
{comment} comment_body = [all -['*' + '/']]*;
{comment} star = '*';
{comment} slash = '/';
```

SableCC - przykład

Poniższy kod pokazuje użycie parsera i leksera dla prostej gramatyki arytmetycznej. (produkcje typu 0,((0+0)+0))

```
Token /* definicje tokenów */
  l_par = '('; r_par = ')';
  plus = '+'; number = ['0'..'9'];

Productions /* grammar */
  exp = number |
        add;
  add = l_par exp plus exp r_par;
```

JavaCC

- Oparty na licencji BSD
- Generacja parserów LL(1)
- Cały czas rozwijany
- Rozbudowana dokumentacja i tutoriale
- Generacja kodu w Javie
- Opis gramatyki - EBNF

https://javacc.dev.java.net/

ostatnia aktualizacja projektu: 3 stycznia 2006

JavaCC - przykład

```
PARSER_BEGIN(Simple2)
public class Simple2 {
  public static void main(String args[]) throws ParseException {
    Simple2 parser = new Simple2(System.in);
    parser.Input();
   }
}

PARSER_END(Simple2)

SKIP :
{              // pomijamy poniższe znaki, a przy wszystkich innych rzucamy wyjątkiem
  " " | "\t" | "\n" | "\r"
}

void Input() :
{}
{ <EOF> }
```

Plusy i minusy używania JLex+CUP

plusy:

- Leksery pisane w JLex są zazwyczaj dużo szybsze niż pisane ręcznie
- JLex posiada wsparcie dla stanów leksera (jak gnu flex)
- CUP generuje parser LALR(1) i radzi sobie z niektórymi dwuznacznościami gramatyki - usuwając konflikty
- parsery LALR(1) są zazwyczaj szybsze od równoważnych parserów LL(l) zastosowanych w ANTLR
- JLex i CUP są dostępne w postaci kodu źródłowego

minusy:

- JLex wspiera znaki 8 bitowe, a java jest przystosowana do 16 bitowego kodowania Unicode
- JLex posiada błędy pojawiające się przy użyciu złożonych makr
- Makra JLex są traktowane podobnie jak C, co oznacza że mogą się pojawiac błędy (trudne do wykrycia) podobne do tych w C w obecności pewnych makr np. makro M zdefiniowane jako a|b oraz wyrażenie aMb, będzie interpretowane jako aa|bb, a nie a(a|b)b jak zamierzono
- CUP nie wspiera budowy abstrakcyjnych drzew składniowych (AST przy niewielkich cenach pamięci operacyjnej jest obecnie bardzo poszukiwaną cechą generatorów parserów)

Plusy i minusy używania JavaCC

plusy:

- wspiera kodowanie Unicode 16 bitowe
- posiada dobrą integrację skanera z parserem
- wspiera EBNF
- wspiera budowę abstrakcyjnego drzewa składniowego (AST)
- jest dobrze wspierane (dedykowane grupy dyskusyjne)

minusy:

- kod szybko się rozrasta powodując duże trudności w wyszukiwaniu błędów
- integralność i poprawność AST jest pozostawiona programiście, w przypadku błędów, powstanie zdegenerowane drzewo, błędy te są bardzo trudne do odnalezienia

Podsumowanie

Spośród wymienionych generatorów parserów i lekserów, każde ma pewne wady, ale ma również zalety, które sprawiają, że narzędzie takie staje się bardzo użyteczne w zastosowaniach przy tworzeniu kompilatorów. Porównując pary CUP+JLex, SableCC, JavaCC, zwycięzcą zostaje SableCC. Został on napisany specjalnie pod Jave. Bardzo dobrze wspiera budowę AST, które jest bardzo łatwe do rozbudowywania. Poprzez używanie technik obiektowych, bardzo dobrze oddziela kod generowany maszynowo od tego pisanego ręcznie, dzięki czemu dużo łatwiej jest odnajdywać błędy w kodzie. SableCC wykorzystuje wzorce projektowe i jest wysoce modularny. Na niekorzyść używania oddzielnych programów do analizy składniowej i leksykalnej świadczy fakt, który wskazuje na generowanie wielu błędów podczas poprawiania kodu pisanego pod jeden generator, a które pojawiają się dla drugiego generatora.