Wybrane narzędzia do tworzenia analizatorów leksykalnych i składniowych w C/C++

by Kapitol Team

Flex a generatory skanerów C++

- 2 sposoby wymuszenia stworzenia skanera w C++
  - flex -+ flexfile.l
  - użycie %option c++
- wygenerowany plik: lex.yy.cc
- lex.yy.cc załącza plik FlexLexer.h definiujący dwie klasy
  - FlexLexer – interfejs głównego skanera
  - yyFlexLexer – dziedziczy po FlexLexer, pomocnicze funkcje
- kompilacja
  - g++ lex.yy.cc -o skaner -lfl

Yet-another Object Oriented Lex (YooLex)

- Podział pliku wejściowego na 3 sekcje (jak w Flexie)

- sekcja 1: - instrukcje na początku pliku wyjściowego
  - nazwy wyrażeń regularnych
  - warunki startu
- sekcja 2: - prolog umieszczony na początku funkcji yyLexCase( )
  - definicje wyrażeń regularnych
  - epilog umieszczony na końcu w/w funkcji (w celach ewentualnego sprzątania)
- sekcja 3: - instrukcje dodane na końcu pliku wyjściowego (pomocnicze klasy, funkcja main)

Yet-another Object Oriented Lex (YooLex)

- instrukcje dodane na początku pliku wyjściowego

```
%{
// code here will be echoed at the beginning of the generated
// C++ source file.
#include <iostream>
using namespace std;
%}
```

- nazwy wyrażeń regularnych

```
IDENT       [A-Za-z_][A-Za-z0-9_]*
NUMBER      [1-9][0-9]*|0
```

- warunki startu
  - wspólny %s CONDITION1 CONDITION2
  - wyłączny %x CONDITION1 CONDITION2

Yet-another Object Oriented Lex (YooLex)

- prolog umieszczony na początku funkcji yyLexCase( )

```
%{
// code here will be inserted at the beginning of class::yyLexCase () function.
// It is useful to has some code that does something whenever yycase () is
// called.
static int yyCaseCallCounter = 0;
%}
```

- definicje wyrażeń regularnych

```
ab(c/d)*e      {std::cout<<"Hello world!"<<std::endl;}
<<EOF>>        {std::cout<<"End of file."<<std::endl;
               yyterminate();
               }
```

- epilog umieszczony na końcu funkcji yyLexCase( )

```
%{
// code here will be inserted at the end of class::yyLexCase () function.
myObject.cleanUp( );
%}
```

Yet-another Object Oriented Lex (YooLex)

- instrukcje dodane na końcu pliku wyjściowego

```
int main (void) {
        std::ios::sync_with_stdio (false);
        DefaultLex scanner;
        while (scanner.yyLex() >= 0)
                ;
        return 0;
}
```

- Kształt całego pliku wejściowego lexera:

```
{section 1}
%%
{section 2}
%%
{section 3}
```

Yet-another Object Oriented Lex (YooLex)

- Konfiguracja pliku wejściowego:

```
%option main line char          /*w sekcji 1*/
```

- main – generuj domyślną
- line – zliczanie linii w pliku wejściowym skanera
- char – zliczanie znaków w pliku wejściowym skanera

- Pomocnicze zmienne, funkcje, makra
  - YYTextType _yyText; /*dopasowany ciąg znaków*/
  - yyleng /*makro zwracające długość dopasowanego ciągu*/
  - size_t yyGetCharNum () const; /*dla %option char, zwraca dotychczasową liczbę dopasowanych znaków w pliku wejściowym*/
  - size_t yyGetLineNum () const; /*dla %option line, zwraca dotychczasową liczbę linii w pliku wejściowym*/
  - bool yyIsBOL () const; /*zwraca true, jeśli jesteśmy na początku linii*/
  - yyterminate () /*kończy pracę skanera*/

Yet-another Object Oriented Lex (YooLex)

- Przykład: Skaner analizujący tekst i zwracający jego statystyki

```
/*Statystyki dla tekstu*/
      %{
            #include <iostream>
            using namespace std;
      %}
            string [A-Za-z_][A-Za-z_0-9]*
            emptysign [ \t\n]
            text {string}{emptysign}+
      %option main line char
%%
            static int counter=0;
      {emptysign}*{text}    {++counter;std::cout<<_yyText<<"|"<<std::endl;}
      <<EOF>>     {std::cout<<"Naliczyłem się: "<<counter<<" słów, ";
                  std::cout<<_yyLineNum<<" linii, ";
                  std::cout<<_yyCharNum<<" liter."<<std::endl;
                  yyterminate();}
%%
```

Yet-another Object Oriented Lex (YooLex)

- Generacja parsera z pliku plik.l:
  ./yoolex plik.l

- generuje pliki DefaultLex.cc i DefaultLex.hh – pliki skanera

- Kompilacja parsera parser:
  g++ DefaultLex.cc -Iścieżka_do_katalogu_yoolex -o parser

Parsery (Analizatory składniowe)

- parser kolejno pobiera tokeny od analizatora leksykalnego (lexera) i stwierdza czy dana sekwencja mogła zostać wygenerowana przez zadaną z góry gramatykę

- parsery tworzy się do gramatyk bezkontekstowych (np. gramatyk języków programowania - łatwe w przetwarzaniu w przeciwieństwie do gramatyk opisujących język naturaly)

- typy parserów:
  a)top-down
  b)bottom-up

Parsery (Analizatory składniowe)

- top-down – analiza zaczyna się od symbolu startowego; następnie podejmowana jest próba wyprowadzania z niego coraz bardziej szczegółowych konstrukcji, która w razie sukcesu kończy się wyprowadzaniem zadanego słowa wejściowego;przykładem parserów top-down są parsery korzystające z algorytmu LL (wywód lewostronny)

- bottom-up – analiza zaczyna się od tokenów podawanych na wejście (terminali);podejmowana jest próba z redukowania ciągu tokenów do symbolu startowego;
przykładem są tu parsery korzystające z algorytmu LR (wywód prawostronny)

Lemon

- stworzony na potrzeby projektu SQLite
- napisany w C
- generuje parsery w C i C++
- generuje parsery LALR(1)
  - parsery LALR – przetwarzają wiecej gramatyk bezkontekstowych niż SLR, ale mnie niż LR(1);LALR stanowi kompromis między wielkością tablicy parsingu, a ilością gramatyk, które przetwarza
- cechy generatora:
  - reentrant – ten sam fragment kodu może być dzielony między wiele niezależnych procesów
  - thread-safe - bezpieczny, gdy mamy do czynienia z aplikacją wykorzystującą wielowątkowość

Lemon

- cechy generatora (c.d):
  - implementuje koncepcję destruktora terminali i nieterminali
- szybszy od yacc czy bisona
- przekształca bezkontekstowe gramatyki (Context Free Grammar)
- nie ma wbudowanego lexera; jest kompatybilny z flexem
- gramatykę umieszcza się w pliku z roszerzeniem .y (tak jak w bisonie)
- ogólna postać reguł:

  expr ::= expr TOK_NR1 expr1.

  expr – nieterminal

  TOK_NR1 – terminal

Lemon

- wykonywanie akcji związanych z dopasowaniem reguły

```
expr ::= expr TOK_NR1  { akcja; }
```

- wykonywanie operacji na wartościach (zwracanie)

```
%type expr {int}
```

- deklarujemy, że nieterminal 'expr' zwraca wartość typu int (możemy sobie tworzyć własne typy)

```
expr(A) ::= expr(B) OPERATOR expr(C)  { A = B + 2 * C; }
```

- domyślnie konfilikty shift/reduce są rozwiązywane na korzyść shift, reduce/reduce na korzyść tej reguły, która pierwsza pojawiła się w definicji gramatyki; jednak lemon pozwala na zmienianie tego zachowania przy użyciu tzw. “precedence rules” , które odnoszą się do terminali, np:

```
%left PLUS MINUS.

%left DIVIDE TIMES.
```

Lemon

- konflikty (c.d)

Przy czym reguły znajdujące się 'wyżej' mają niższy priorytet, stąd:

expr PLUS expr DIVIDE expr

zostanie zinterpretowane jako:

expr PLUS (expr DIVIDE expr)

- Oprócz ww. zależności są jeszcze reguły decydujące o kolejności wiązania:

%left OP1.
%right OP2.

expr OP1 expr OP1 expr

zostanie zinterpretowane jako (expr OP1 expr) OP1 expr

z kolei:

expr OP2 expr OP2 expr

jako:expr OP2 (expr OP2 expr)

Elkhound

- stworzony na potrzeby projektu Elsa (parser C/C++)
- napisany w C++
- generowane parsery w C++ (i Ocaml)
- generuje parsery GLR
  - GLR (Generalized LR) – parsery GLR działają tak samo jak LR , z tym , że tablica parsingu może zawierać konfilikty typu shitf/reduce lub reduce/reduce; gdy wykryty zostaje konflikt stos zostaje podzielony na dwie części i każda gałąź jest utrzymywana oddzielnie
  - ma złożoność $O(n^3)$
  - stworzony by radzić sobie z gramatykami niedeterministycznymi i niejednoznacznymi
- wygenerowane parsery mogą używać dodatkowo algorytmu LR

Elkhound

- nie ma wbudowanego lexera; można użyc np. flexa jako lexera
- gramatykę umieszcza się w pliku .gr
- ogólna postać reguł:

```
nonterm expr {
        -> expr OPERATOR expr
}
```

- akcje przy dopasowaniu:

```
nonterm expr {

        -> expr OPERATOR expr    { akcja; }
}
```

- zwracanie wartości:

```
nonterm(int) expr {

    -> a1:expr PLUS a2:expr1    { return a1 + a2; }
    -> n:LITERAL { return n; }

}
```

Elkhound

- aby zwrócić token jako wartość należy uprzednio zadeklarować jak ma być dany token interpretowany:

```
token(int) LITERAL;
```

- priorytety ustala się w następujący sposób

```
precedence
{
   left  20 MUL DIV;        // wyższy priorytet, łączność lewostronna
   right  10 PLUS MINUS;    // niższy priorytet, łączność prawostronna
}
```

Bibliografia

- http://www.compilers.net/Dir/ScannerGens.htm
- http://www.compilers.net/Dir/ParserGens.htm
- http://dmoz.org/Computers/Programming/Compilers/Lexer_and_Parser_Generators/
- http://www.gnu.org/software/flex/manual/html_node/flex_19.html
- http://yoolex.sourceforge.net/

- http://www.hwaci.com/sw/lemon/
- http://www.sqlite.org/
- http://linuxgazette.net/106/chirico.html

- http://en.wikipedia.org/wiki/GLR_parser
- http://www.cs.berkeley.edu/~smcpeak/elkhound/

- http://en.wikipedia.org/wiki/Lexical_analysis
- http://en.wikipedia.org/wiki/Parser
- http://en.wikipedia.org/wiki/Compiler-compiler

- http://www.google.pl