Świat parserów

Krzysztof Leszczyński
Polska Grupa Użytkowników Linuxa
chris@camk.edu.pl

Streszczenie

Pisząc oprogramowanie, stykamy się bardzo często z potrzebą analizy języków programowania. Niniejsza praca ma pomóc czytelnikowi zorientować się w gęstwinie różnych mniej lub bardziej rozbudowanych generatorów parserów.

1. Wprowadzenie

Języków programowania jest dzisiaj dużo, żeby nie powiedzieć za dużo. Ankieta, przeprowadzona wśród koleżanek i kolegów zajmujących się programowaniem, wykazała, że programiści twierdzą, że posługują się przeciętnie pięcioma językami. Okazuje się, że jest to przekonanie błędne. Indagowani przeze mnie, przyznają, że co prawda znają (lepiej lub gorzej) języki „podstawowe": C, C++, jeden z P* (Perl, Python, PHP), SQL, to jeszcze posługują się wieloma innymi językami, o których istnieniu nie wiedzą; podobnie jak nie wiemy, że prawie cały czas mówimy prozą. Gdybyśmy policzyli liczbę rodzajów plików, które edytujemy, to różnych języków doliczymy się prawdopodobnie co najmniej dwudziestu. Postarajmy się wobec tego zdefiniować, co w ogóle uważamy za język programowania. Bardzo nieformalna definicja brzmi: językiem nazywamy sposób zapisu pewnego algorytmu, bądź chociażby pewnych danych, w taki sposób, że jest on możliwy do ręcznej edycji przy pomocy edytora tekstowego. Za języki danych (choć nie za języki programowania) uznalibyśmy także pliki konfiguracyjne niektórych programów, na przykład `/etc/named.conf`. Z drugiej strony, formatu kompresji ZIP nie uznamy za język programowania, gdyż nie wyobrażam sobie potrzeby uczenia się algorytmów i zapisu algorytmu *Deflate* w innych celach niż sama analiza algorytmu.

Za język nie uznamy też formatu `/etc/passwd` lub `/etc/shadow`, gdyż są to typowi przedstawiciele plików CSV (*comma-separated values*) i, oczywiście, analiza takich plików jest zbyt prosta, żeby stosować do nich techniki analizy.

2. Gramatyki

Bardzo często pliki, takie jak wymieniony już `/etc/named.conf`, musimy analizować z poziomu pisanych przez nas programów, wówczas napisanie analizatora może kosztować nas wiele czasu, a otrzymana procedura nie musi być wolna od błędów.

Zdecydowana większość używanych przez nas języków mieści się w klasie języków generowanych gramatykami bezkontekstowymi. Czym innym jest jednak zdefiniowanie gramatyki, a czym innym napisanie do niej parsera. A parsery musimy pisać dosyć często. W przypadku języków opisu danych strukturalnych, pewną nadzieję dawało rozpowszechnienie się języka XML. Nadzieja okazała się złudną, gdyż XML naprawdę niewiele pomógł. Zamiast analizować mniej lub bardziej swobodny tekst, musimy analizować tekst XML-owy pokrojony na leksemy XML-owe. Analizy jest niewiele mniej niż w czasach przed-XML-owych.

Byłoby najlepiej, gdybyśmy mogli opisać formalną gramatykę i opisać reguły przetwarzania określonych struktur. Do tego właśnie służą generatory parserów.

Przed rozpoczęciem poszukiwania odpowiedniego narzędzia należy odpowiedzieć na kilka pytań:

1. W jakim języku (językach) ma być wygenerowany docelowy parser.
2. Jak bardzo nam zależy na bezpieczeństwie, w szczególności czy ufamy osobie piszącej skrypt, który ma być interpretowany.
3. Czy zależy nam na prędkości analizy.

Odpowiedź na pytanie 2 jest bardzo istotna, szczególnie jeśli językiem docelowym ma być jakiś język interpretowany. Na przykład, bardzo dużo programów pisanych w Perlu ma pliki konfiguracyjne pisane również w Perlu, na ogół jako ciąg przypisań. Jest to bardzo wygodne pod warunkiem, że wierzymy w czystość intencji użytkownika – nikt nie zabroni w pliku konfiguracyjnych uruchomić dowolnie szatańskiego kodu! Dlatego sporo programów, które działają w trybie użytkownika posługuje się tą metodą – klasycznym przykładem jest tu program `mirror`. Z kolei programy uruchamiane w trybie *daemon*, unikają takich metod. Stare wersje programu `SpamAssassin` posługiwały się tą metodą. Nowe mają własny, nieperlowy opis plików, do tego stopnia, że nawet nie pozwalają stosować wyrażeń regularnych, gdyż Perl pozwala w regułach stosować dowolny kod – również ten o bardzo kosmatych skutkach.

Nawet pomijając kwestie bezpieczeństwa, „pliki konfiguracyjne” pisane w tym samym języku, w którym jest główny program, są czasami zbyt silne dla zwykłych użytkowników. Przy programach bardziej rozpowszechnionych lepiej czasami ograniczyć możliwości, gdyż potem będzie bardzo trudno wprowadzać zmiany.

Na pytanie 3, w przypadku analizatorów krótkich plików konfiguracyjnych odpowiedzią z pewnością będzie „nie”. Nie jest dla nas istotne, czy plik się będzie się przetwarzał jedną tysięczną czy jedną dziesięciotysięczną sekundy; jeśli używamy analizatora leksykalnego jako części analizatora pakietów w routerze, to jego prędkość może mieć dla nas duże znaczenie.

3. Gramatyki bezkontekstowe

Poniższe definicje będą bardzo uproszczone, ale ilość miejsca na ten referat jest dosyć ograniczona, poza tym chcemy się skupić na istniejących rozwiązaniach. Czytelników pragnących zapoznać się z dokładnymi informacjami na temat gramatyk, pozwolę sobie odesłać do literatury [1].

W skrócie: gramatykę zapisujemy w postaci tzw. reguł produkcji. Każda gramatyka zawiera symbol startowy (na przykład *program*) oraz zbiór reguł, które pozwalają złożyć *program* z innych elementów, aż do elementów terminalnych, czyli leksemów, takich jak *liczba* albo „+”. Na przykład, jeśli *program* może być pusty lub zawierać ciąg instrukcji zakończonych średnikiem, to możemy napisać takie reguły:

```
program ::= . /* program może być pusty */
  /* definicja rekursywna */
program ::= program instrukcja .
```

oczywiście trzeba napisać co to jest instrukcja, jeśli instrukcją jest wyrażenie zakończone średnikiem, to definicja może być taka

```
instrukcja ::= wyrażenie ';' .
```

Gramatykę zawierającą po lewych stronach produkcji jedynie po jednym elemencie nieterminalnym, nazywamy gramatyką bezkontekstową. Gramatyka, w której byśmy inaczej interpretowali produkcje w zależności od kontekstu, nazywamy niebezkontekstową. Przykładem takiej produkcji jest wyrażenie, po którego obu stronach stoją wyrażenia identyczne

```
wyr1 wyr2 wyr1 ::= ...
```

Na szczęście gramatyki niebezkontekstowe (*non-context-free grammars*), nie występują często w praktycznych zastosowaniach i nie będę ich w ogóle rozpatrywał. Generatory parserów dla takich gramatyk są przeważnie drogie w użyciu i, poza bardzo specjalnymi przypadkami, nieprzydatne. Jedynym, potwierdzającym regułę, wyjątkiem jest gramatyka wyrażeń regularnych `pcre`, która nie jest, w ogólnym wypadku, bezkontekstowa, a do której istnieją szybkie parsery.

Parsery działają na leksemach. Leksemem nazywamy najmniejszą niepodzielną część języka. W popularnych językach programowania, leksemami są na przykład identyfikatory, liczby całkowite, liczby rzeczywiste, ale również tak skomplikowane twory jak stałe napisowe. Oczywiście, co jest a co nie jest leksemem zależy od konkretnego języka. W językach C lub Pascal nie istnieje identyfikator `123abc` – żadna zmienna, ani funkcja nie może mieć takiej nazwy. Wynika to z zasady, że identyfikatory w tych językach nie mogą mieć nazw zaczynających się od cyfry. Z drugiej strony, w Lispie identyfikatorem może być niemal dowolny ciąg znaków.

4. Leksery

Parsery żywią się leksemami, których musimy dostarczyć. Leksemy na ogół składamy z jeszcze mniejszych elementów, nazywanych *znakami.* Zbiór znaków nazywamy *alfabetem.* Alfabetem może na przykład ASCII lub ISO-8859-2, czyli coś co nam się najbardziej kojarzy z alfabetem. Może też być UTF-8, który się charakteryzuje tym, że znaki nie mają równej długości w bitach. Dla dekompresora `gunzip` alfabetem będą pojedyncze bity.

Naszym zadaniem jest zbudować leksem z ciągu znaków, według określonych reguł – na ogół według pewnego zbioru wyrażeń regułowych. W przypadku prostych reguł możemy sami napisać kod, który podzieli ciąg znaków na odpowiednie leksemy, ewentualnie możemy się posiłkować biblioteką `pcre` (*Perl-compatible regular expression library*), dostępną obecne każdej dużej dystrybucji Linuxa.

Tam gdzie mamy bardziej skomplikowane reguły, lepiej jest użyć prawdziwych lekserów. Do celów testowych zdefiniujmy sobie mały `C`-podobny język. Język będzie się składał z:

- liczb całkowitych, zdefiniowanych jako ciągi cyfr, pomijamy dopuszczalność zakresów,
- identyfikatorów, takich jak w C lub Pascalu,
- znaków operacji: „+”, „-”, „*”, „/”,
- ciągi „`/*`*ciąg znaków*`*/`” są traktowane jak komentarze i nie generują leksemów.
- słowa kluczowe: `for`, `if`, `else`. Słowa kluczowe, nie mogą być identyfikatorami.

Wyposażeni w powyższe reguły możemy poznać pierwszego i zarazem najstarszego z przedstawicieli lekserów.

4.1. Lex

Program Lex [2] jest związany z najwcześniejszymi systemami UNIX, był dostępny już na PDP-11. Produkuje lekser w języku C. Obecnie głównie używamy jego wersji GNU: `flex` [3].

Plik dla programu `(f)lex` dzieli się na trzy części oddzielone znakami `%%`:

definicje
`%%`
reguły
`%%`
dodatkowy kod

część trzecia jest opcjonalna, a pierwsza musi występować, ale może być pusta. Reguły są w postaci

wzorzec akcja,

gdzie *wzorzec* jest rozszerzonym wyrażeniem regularnym (*regular expression*), zaś akcja jest fragmentem w C, który jest wykonywany jeśli *wzorzec* jest spełniony.

Sprawdźmy jak będzie wyglądać lekser dla naszego przykładowego małego języka, pomińmy na razie skanowanie komentarzy.

```
%{
#include "leksdef.h"
%}
%%
"for"                      return LEKSEM_FOR;
"if"                       return LEKSEM_IF;
"else"                     return LEKSEM_ELSE;
[0-9]+                     return LEKSEM_LICZBA;
[A-Za-z_][A-Za-z_0-9]* return LEKSEM_ID;
%%
```

W pierwszej części widzimy jedną instrukcję `#include`, drugiej części w zasadzie nie trzeba tłumaczyć, jest oczywista. Dla każdego wyrażenia regularnego zwracamy rodzaj leksemu. Możemy ten plik skompilować poleceniem

```
flex przyklad1.lex
```

Otrzymamy wówczas plik wynikowy o charakterystycznej nazwie `lex.yy.c`. Plik ten możemy skompilować, o ile w pliku `leksdef.h` dostarczymy odpowiednich definicji leksemów, na przykład w postaci

```
enum {LEKSEM_FOR, LEKSEM_IF,
  LEKSEM_ELSE, LEKSEM_LICZBA, LEKSEM_ID};
```

Plik wynikowy zawiera funkcję *yy_lex*(), która wywoływana w pętli, zwraca kolejne leksemy, zgodnie z akcjami.

Same leksemy jednak nam nie wystarczą, co z tego, że wiemy, że lekser natrafił na liczbę, skoro nie wiemy co to za liczba! Wartość leksemu (w postaci tekstu), jest zachowana w zmiennej globalnej, zdefiniowanej jako

extern char **yytext;*

do której możemy zaglądać pomiędzy wywołaniami funkcji *yy_lex*().

A co mamy zrobić, jeśli chcemy mieć w jednym programie kilka lekserów? W przypadku oryginalnego Lexa mamy kłopot i raczej bez przetwarzania pliku wynikowego się nie obejdzie, a i wtedy mamy kłopot, gdyż Lex zbyt chętnie używa zmiennych globalnych. `flex` pozwala wygenerować klasy C++, dla każdego leksera oddzielnie. Użycie metod takich klas nie powoduje efektów ubocznych. Na koniec jeszcze usprawiedliwienie, w przykładzie nie pokazałem, jak rozwinąć skaner, tak aby obejmował skanowanie komentarzy. Taki skaner by zajął więcej niż 60 wierszy, czyli dwie strony, a co najważniejsze, jest on dokładnie opisany jako przykład w manualu *flex(1)*, więc przepisywanie go tutaj jest niepotrzebne. Niemniej, namawiam do zapoznania się z tym przykładem, gdyż wprowadza on kilka pojęć, takich jak warunki startowe leksera, których w tym przeglądowym tekście nie omawiam.

4.2. re2c

Moim ulubionym lekserem jest `re2c` [4]. Również generuje kod w C/C++, niemniej kod jest czystszy – da się przeczytać oczami programisty, w odróżnieniu od kodu `lex/flex`, który jawi się jako kilkaset liczb całkowitych wraz z tajemniczymi indeksami przeskakującymi po nich.

Pisząc kod w `re2c`, musimy zbudować szkielet programu, taki szkielet dla naszego języka został przedstawiony na przykładzie 1. Struktura `struct Leksem`, opisana w wierszu 2 została przeze mnie wymyślona *ad hoc* dla potrzeb przykładu. Jedynie wiersze 5–9 są narzucone przez program, gdyż musimy jakoś zdefiniować symbole `YYCTYPE`, `YYCURSOR`, `YYLIMIT`, `YYMARKER` oraz `YYFILL`, które są używane przez generator. Symbole te mogą być zdefiniowane lokalnie, dlatego nie zaśmiecają przestrzeni nazw. W odróżnieniu od Lexa, `re2c` nie próbuje czytać standardowego wejścia, tylko wymaga zdefiniowania funkcji `YYFILL`, dostarczającej znaki alfabetu. W naszym przykładzie ta funkcja jest pusta, gdyż funkcja *Leksuj*() zakłada, dla prostoty, że całe wejście jest dostarczone jednorazowo w napisie *p*. W moim odczuciu jest to zachowanie bardziej poprawne, gdyż funkcję dostarczającą znaków ze strumienia buduje się banalnie – zwykła funkcja *fread*() załatwia sprawę; jeśli zaś nie korzystamy bezpośrednio z funkcji plikowych, to `re2c` nie wymaga, w odróżnieniu od `lexa` całego wielkiego `<stdio.h>`.

```
#include "leksdef.h"
struct Leksem { int leksem; char *pozycja;};
int Leksuj(char *p,struct Leksem *l){
  char *q;
#define YYCTYPE char
#define YYCURSOR p
#define YYLIMIT p
#define YYMARKER q
#define YYFILL(n)
#define R(leks) {l->leksem=leks; \
                 l->pozycja=p; return 0;}
 start:
  /*!re2c
    "if"   {R(LEKSEM_IF);}
    "else" {R(LEKSEM_ELSE);}
    "for"  {R(LEKSEM_FOR);}
    [0-9]+ {R(LEKSEM_LICZBA);}
    [A-Za-z_][A-Za-z_0-9]* {R(LEKSEM_ID);}

    "/*"  {goto scan_comment;}
  */
  goto start;

 scan_comment:
  /*!re2c
    "*/" {goto start;}
    [\000-\377] {goto scan_comment;}
   */
}
```

Przykład 1: Skaner `re2c` dla przykładowego minijęzyka

Jeśli chcemy pisać nasze parsery w C++, to zamiast tak prostego nagłówka, możemy stworzyć klasę i wypełnić odpowiednie miejsca kodu metod pseudokomentarzami `/*!re2c...*/` – kod produkowany przez `re2c` jest *odporny na C++*.

Zwolennicy programowania bez skoków, zapewne skrzywią się widząc mnogość instrukcji `goto`. W pliku wynikowym, znajdziemy ich jeszcze więcej, w dużym lekserze nawet kilkaset. Niestety, leksery mają taką naturę, że chętnie z `goto` korzystają. Są oczywiście takie, które tego nie robią, ale za to albo generują kilkadziesiąt razy zagnieżdżone pętle i warunki, albo pracują na wielkich tablicach i zamiast `goto` wykonują *de facto* pseudokod. Na szczęście owe skoki są ograniczone do jednej funkcji, więc nie psują nam reszty programu.

4.3. Inne leksery

Lekserów jest naprawdę dużo i nie sposób ich choćby pobieżnie wymienić. W Perlu możemy używać modułu `Parse::Lex` oraz `Parse:YYLex`, ten ostatni jest kompatybilny z Lexem. Reguły możemy ustalać w czasie uruchamiania programu, a nie w czasie jego kompilacji. Ceną za to jest znacznie mniejsza prędkość od lekserów w C.

5. Parsery

Gramatyki bezkontekstowe dzielą się na wiele podrodzajów. Na szczęście większość języków jest opisywalna gramatykami typu LR(1) lub LL(1). Dla gramatyk LR(1), a konkretniej LALR(1) mamy parsery Yacc, Bison oraz Lemon, dla gramatyk LL* mamy do dyspozycji ANTLR [7].

5.1. Yacc i Bison

Yacc (*Yet Another Compiler-Compiler*) jest bardzo sędziwym programem napisanym dla systemów UNIX. Jest dostępny z różnymi komercyjnymi systemami, na przykład z SunOS lub Solarisem. Jego licencja nie pozwala na swobodne rozpowszechnianie, za to możemy zupełnie swobodnie dysponować kodem wygenerowanym przez Yacca. Istnieją dwie wolne wersje tego programu: `byacc`, czyli Berkeley Yacc, na licencji *public domain* oraz Bison na licencji GPL. W przypadku używania kodu wygenerowanego przez Bisona, kod wynikowy albo sam musi być na licencji GPL albo musi spełnić dosyć restrykcyjne wymagania. To może się zemścić, nawet jeśli nie mamy zamiaru pisać oprogramowania komercyjnego z kodem zamkniętym. Oto prawdziwa historia: W połowie lat 90-tych, do produkcji jednej ze stref gry typu AberMUD Northern Lights użyto Bisona. Wtedy Bison miał bardziej restrykcyjną licencję, mógł być użyty wyłącznie do produkcji wolnego oprogramowania. Oczywiście, gracze nie mogli widzieć kodu źródłowego strefy, bo znaliby rozwiązanie. Sam kod AberMUD-a jest dostępny i otwarty, jedynie strefy są zamknięte z oczywistych powodów. Niemniej, jeden z graczy zażądał jego udostępnienia powołując się na licencję Bisona. Zrobiła się awantura, która oparła się o samego RMS-a. Jej efektem było paniczne przepisywanie kodu na `byacc`-a i miesiącami trwające dysputy czy kod *przez pomyłkę zarażony* GPL, może być *uzdrowiony* poprzez kompilację innym narzędziem.[1]

Plik w Yaccu lub Bisonie wygląda bardzo podobnie jak plik w Lex. Podobnie są trzy strefy rozdzielone znakami `%%`. W przykładzie 2 znajduje się przykład czterodziałaniowego kalkulatora, wraz z kolejnością działań. Gramatyka nie wymagałaby tłumaczenia, gdyby nie dziwna reguła

```
skladnik: skladnik '-' skladnik { $$=$1-$2;}
```

widzimy tu rekurencje, która nie wiadomo w którą stronę ma działać. Czy wyrażenie $1-2-3$ ma

[1] Autor strefy stwierdził, że dyskusja jest bezprzedmiotowa, bo poprzedni kod (bisonowy) zgubił, nie ma kopii zapasowej i nie jest go w stanie odzyskać.

```
%{
#define YYSTYPE double
#include <math.h>
%}
%token NUM
%left '-' '+'
%left '*' '/'
%%
input: /* puste */
       | input wiersz
       ;
wiersz: '\n'
   | wyr '\n'  {printf ("wynik\t%.10g\n", $1);}
        ;

wyr: '(' wyr ')' {$$ = $1;}
wyr: skladnik {$$ = $1;}
skladnik: czynnik {$$=$1;}
skladnik: skladnik '+' skladnik { $$=$1+$2;}
skladnik: skladnik '-' skladnik { $$=$1-$2;}
czynnik: NUM { $$ = $1; }
czynnik: czynnik '*' czynnik {$$ = $1*$2 }
czynnik: czynnik '/' czynnik {$$ = $1*$2 }
%%
```

Przykład 2: Parser w języku Bison (Yacc) dla kalkulatora czterodziałaniowego

być zinterpretowane jako $(1-2)-3=-4$ czy jako $1-(2-3)=2$? Na szczęście reguła `%left '+' '-'` mówi, że `'-'` *wiąże w lewo*, czyli tak jakbyśmy się spodziewali. Podobnie wiąże `'+'`, przy czym akurat to nie ma w tym przypadku większego znaczenia. Gdybyśmy zechcieli wprowadzić operator potęgowania `'^'`, który na ogół wiąże w prawo, czyli `2^3^4` oznacza 2^{3^4}, a nie $(2^3)^4$, to musielibyśmy dopisać deklarację `%right '^'`.

Ten prosty przykład pokazuje, że co prawda gramatyki w Bisonie wyglądają zrozumiale, to jednak trzeba uważać na pułapki, na szczęście większość z nich Bison wykryje na etapie kompilacji.

Yacc lubi pracować z programem `lex/flex`. Nie jest łatwo go przekonać do pracy z innymi lekserami, np. `re2c` lub innym, zrobionym ręcznie. Powodem jest fakt, że `yacc` zakłada, że istnieje funkcja *yylex*(), która zwraca leksemy i która umieszcza wartość leksemu w zmiennych *yyval* i *yytext*.

W przypadku Bisona jest lepiej, jeśli umieścimy deklarację `%pure_parser`, to nasz parser będzie *czysty* i nie będzie zależał od żadnych zmiennych globalnych, za to funkcja *yylex*() będzie dostawać zmienne lokalne *yyval* jako argument przez referencję.

5.2. Lemon

No właśnie! Bison sam z siebie chce wołać lekser tyle razy ile potrzebuje. W momencie kiedy wywołamy parser tracimy kontrolę, aż do momentu kiedy Bison przeanalizuje cały kod, bądź zgłosi błąd syntaktyczny. To może nie być problem w przypadku przetwarzania wsadowego, ale stwarza trudności w przypadku programów interakcyjnych kiedy parser ma działać przez cały czas działania programu, a program musi mieć sterowanie – choćby po to, żeby je oddać funkcji *gtk_main*().

Jeśli nie chcemy stosować wielowątkowości – rzecz z parserami generowanymi przez `yacca` sama w sobie niebezpieczna – musimy użyć innego generatora parserów.

Lemon[5] jest generatorem parserów gramatyk LALR(1), o podejściu zupełnie różnym od rodziny Yacc:

- używa innej gramatyki, która jest bardziej odporna na błędy;
- parsery są wielobieżne, wielowejściowe (*reentrant*) oraz bezpieczne w środowisku wielowątkowym (*thread safe*);
- leksemy użyte w wyrażeniach są starannie niszczone, przez co kod nie cieknie, co się niestety nagminnie zdarza parserom.

Lemon jest również gruntownie przetestowany przez projekt SQLite [6]. Generator jest dostępny na zasadach *public domain*, co jest wygodne, gdyż kod wygenerowany Lemonem nie podlega żadnym restrykcjom.

Na ilustracji 3 znajduje się przykładowy kalkulator czterodziałaniowy. W porównaniu z parserami Bisona widać, że nie stosujemy tajemniczej notacji `$$=$1+$2`, tylko mamy zmienne z normalnymi nazwami. W dodatku, wszelkie symbole nieterminalne niosące informację (takie jak `czynnik`) muszą mieć przypisane nazwy, zaś wszelkie symbole terminalne nie niosące informacji – takie jak leksem `LEWYNAWIAS` nie mogą mieć nazw.

Te cechy powodują, że programy w Lemonie są dużo czytelniejsze od swoich bisonowych odpowiedników, są też bardziej odporne na błędy i wygodniejsze w użyciu. Dlatego, moim zdaniem, nie ma sensu używać w nowych projektach parserów bison/yacc. Po prostu, większe programy są w `yaccu` mało czytelne, a notacja *$liczba* jest w przypadku bardziej skomplikowanych wyrażeń po prostu nieczytelna i uderza swoją *byleyaccością*.

5.3. ANTLR – przedstawiciel parserów LL

ANother Tool For Language Recognition [7] jest ostatnim z omawianych generatorów parserów. Napisany został przez prof. Terence'a Parra. W odróżnieniu od poprzednich generatorów, używa gramatyk typu LL, a nie LR. Nie da się ukryć wrażenia,

```
%token_type {double}
%token_destructor {void(0);}
%include {#include "toy.h"}
input ::= lista_instr .
lista_instr  ::= .
lista_instr ::= instrukcja .
lista_instr ::= lista_instr SEMICOLON instrukcja .
instrukcja ::= .
instrukcja ::= wyr(E). {print("%g\n",E);}
%type wyr {double}
%type skladnik {double}
wyr(T) ::= skladnik(S). { T=S; }
wyr(T) ::= wyr(TA) PLUS skladnik(S). { T=TA+S; }
wyr(T) ::= wyr(TA) MINUS skladnik(S). { T=TA-S; }
%type czynnik {double}
skladnik(S) ::= czynnik(C). {S=C;}
skladnik(S) ::= skladnik(SA) RAZY czynnik(C). {S=SA*C;}
skladnik(S) ::= skladnik(SA) DZIELI czynnik(C). {S=SA/C;}
czynnik(C)  ::= LICZBA(L). {C=L;}
czynnik(C)  ::= LEWYNAWIAS wyr(T) PRAWYNAWIAS. {C=T}
```

Przykład 3: Parser w języku Lemon dla kalkulatora czterodziałaniowego

że autor stara się walczyć z LR-ami, a zwłaszcza z powszechnym ignorowaniem gramatyk LL – skoro nazwał swój produkt nazwą, którą się czyta podobnie do anty-LR.

Jest to bardzo potężne narzędzie. Potrafi produkować parsery w językach: Java, C++, C# oraz Python. Inne języki też są dostępne, o ile ktoś zaprojektuje odpowiedni generator kodu, jest kilka takich dostępnych w sieci.

Nie ma sensu deliberować na wyższością ANTLR nad poprzednimi parserami, gdyż niektóre języki wygodniej się zapisuje w LL, a niektóre LR. Generalnie gramatyka zapisane w LL jest bardziej iteracyjna niż czysto rekursywna. Na przykład nasz czterodziałaniowy przykład byśmy zapisali jako

```
wyr: skladnik (('+'|'-') skladnik)*
skladnik:  czynnik (('*'|'/') czynnik)*
atom: liczba | ('(' wyr ')')
liczba: ('0'..'9')+
```

Pierwszy wiersz oznacza, że wyrażenie składa się ze składnika oraz zero lub więcej składników rozdzielonych znakami „+" lub „-". W przypadku definicji liczby, widzimy że w ANTLR możemy pisać również leksery – według identycznego schematu.

Tu trzeba jednak przestrzec, że ANTLR jest narzędziem dużym i trudnym do nauczenia, z pewnością nie jest dobrym wyborem na jednodniowy projekt, niemniej przy większych zadaniach warto je rozważyć, zwłaszcza, że jest to prawdopodobnie najbardziej starannie rozwijany, testowany i pielęgnowany generator, który widziałem.

6. Podsumowanie

Dostępnych generatorów parserów jest dużo, nie sposób ich choćby pobieżnie omówić. W ogóle nie zająłem się generatorami z zamkniętym kodem, między innymi dlatego, że nie mogę się oprzeć wrażeniu, że ich wartość *na rynku* jest raczej marginalna. Mam nadzieję, że przekonałem czytelników, że pisząc swój program lepiej jest skorzystać z ułatwienia jakim jest generator parserów lub lekserów, niż pisać własny kod „palcami".

Literatura

[1] Ravi Sethi, Jeffrey D. Ullman, Alfred V. Aho *Kompilatory. Reguły, metody i narzędzia*, Wydawnictwa Naukowo-Techniczne, Styczeń 2002

[2] John R. Levine, Tony Mason, Doug Brown, *Lex & Yacc*, O'Reilly & Associates

[3] Strona projektu Flex, `http://www.gnu.org/software/flex/`

[4] re2c project page, `http://re2c.sourceforge.net/`

[5] The Lemon Parser, Generator `http://www.hwaci.com/sw/lemon/`

[6] The SQLite Project, `http://www.sqlite.org/`

[7] ANother Tool For Language Recognition, `http://www.antlr.org/`