Generator skanerów Flex

Grzegorz Jacek Nalepa

17.7.2000, Kraków, *Revision* : 1.5

Streszczenie

Artykuł prezentuje generator skanerów leksykalnych Flex, będący implementacją standardowego pakietu Lex. Zawarte jest krótkie wprowadzenie dotyczące języków formalnych i ich analizy. Następnie przedstawiona jest praca z generatorem Flex. Kolejne etapy tworzenia skanera przy pomocy Flexa są zaprezentowane na przykładach.

Spis treści

1 Wprowadzenie **2**
- 1.1 Translatory komputerowych języków programowania . . . 2
- 1.2 Formalizacja języków . . . 2
- 1.3 Generatory skanerów i parserów . . . 3

2 Praca z programem Flex **3**
- 2.1 Struktura pliku wejściowego . . . 4
- 2.2 Wzorce . . . 4
- 2.3 Akcje . . . 6

3 Przykład skanera **7**

4 Generowanie skanera **9**

5 Podsumowanie **10**

[1]Tekst ukazał się w: *Magazynie Linux & Unix*, nr 9/2000, wydawanym przez TAO Systems.

[2]Kontakt z autorem: mail:gjn@agh.edu.pl

[3]Tytuł angielski: *Introduction to Flex scanner generator*

[4]Tekst jest rozpowszechniany na zasadach licencji *GNU Free Documentation License*, której pełny tekst można znaleźć pod adresem: http://www.gnu.org/copyleft/fdl.html

1. Wprowadzenie

1.1. Translatory komputerowych języków programowania

Jedną z metod wykorzystania komputerów jest pisanie programów w językach programowania. Narzędzia komputerowe służące do przetworzenia tekstu napisanego w języku programowania (programu) można ogólnie nazwać translatorami. Translatory tłumaczą kod programu do postaci wykonywanej przez komputer, lub nadającej się do przetworzenia przez inny program. Przykładem tych pierwszych są translatory popularnych języków programowania, takich jak ANSI C, które tłumaczą kod w języku programowania do kodu maszynowego, wykonywalnego przez procesor komputera. Przykładem tych drugich mogą być na przykład translatory języków opisów dokumentów takich jak SGML, XML czy LaTeX.

Z innego punktu widzenia translatory można podzielić na kompilatory i interpretery. Te pierwsze dokonują jednorazowego tłumaczenia kodu, który jest następnie wykonywany, niezależnie od kompilatora, przykładem jest kompilator języka C. Przykładem interpretera jest interpreter Sh, który analizuje kod na bieżąco i wykonuje polecenia. Kod interpretowany wymaga każdorazowo do uruchomienia interpretera, w tym przypadku powłoki Sh. Programy interpretowane są przeważnie wolniejsze od kompilowanych. Często jednak prościej jest je pisać i odnajdywać w nich błędy.

Oprócz języków programowania, często korzysta się z języków umożliwiających formatowanie tekstu. Najlepszymi przykładami są LaTeX, SGML, czy XML. Kod w tych językach zawiera tekst i dodatkowe informacje opisujące między innymi jego strukturę (paragrafy, sekcje, rozdziały), wygląd (czcionki, kolory) powiązania pomiędzy fragmentami dokumentu (referencje, indeks, url).

Za bardzo prosty przypadek języka można uznać język zawierający opis opcji przekazywanych do programu w linii poleceń w trakcie jego uruchamiania. Tak więc parser i skaner można również wykorzystać do analizy takiego języka.

1.2. Formalizacja języków

Wspólną cechą tych wszystkich języków jest to, iż są sformalizowane. Formalizacja polega na dokładnym określeniu alfabetu języka, znaków których można używać w tekstach pisanych w tym języku (programach), określenia dopuszczanego łączenia tych znaków – gramatyki języka.

Proces sprawdzania poprawności znaków użytych w tekście nazywa się analizą leksykalną, a część translatora która go realizuje analizatorem leksykalnym lub skanerem. Sprawdzanie poprawności gramatycznej, analizy syntaktycznej, wykonuje parser. Po sprawdzeniu poprawności tekstu, wykryciu i ewentualnej korekcji błędów, może nastąpić translacja do innego języka lub generacja kodu wynikowego.

Podstawowym zadaniem analizatora leksykalnego jest sprawdzenie czy tekst napisany w języku komputerowym nie zawiera niedozwolonych znaków (i ich ewentualne usunięcie) i odnalezienie i rozpoznanie w tekście programu jednostek leksykalnych, często nazywanych tokenami. Tokenami mogą być na przykład słowa kluczowe języka, operatory, łańcuchy tekstowe, czy liczby. Skaner może przekazywać rozpoznawane tokeny do parsera.

Analizator syntaktyczny czyli parser analizuje tekst napisany w języku i dokonuje jego rozbioru gramatycznego. Rozbiór jest przeprowadzany na podstawie formalnego opisu języka. Najczęściej stosowaną notacją, służącą do zapisywania gramatyk jest notacja BNF. Równocześnie parser sprawdza poprawność gramatyczną. Aby to było możliwe, parser przechowuje definicję gramatyki języka.

1.3. Generatory skanerów i parserów

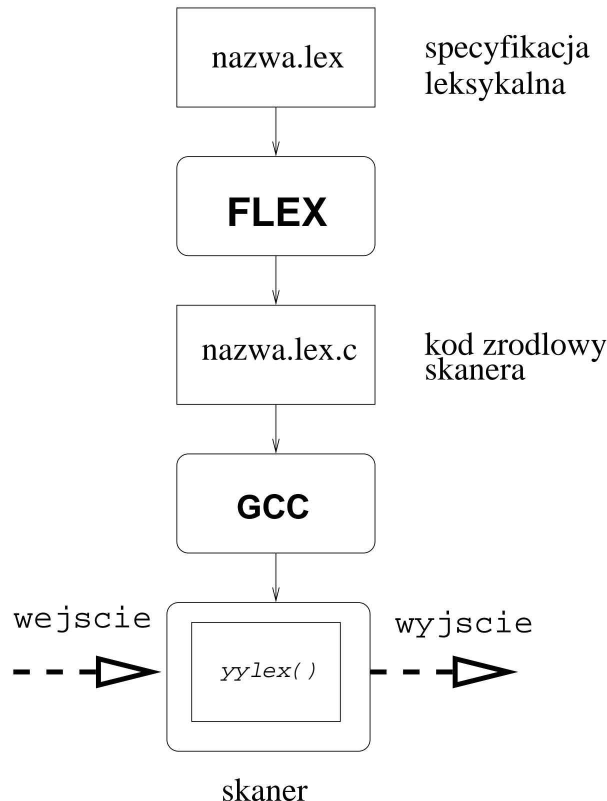
Ponieważ skanery i parsery są bardzo szeroko wykorzystywane, powstały narzędzia ułatwiające ich konstrukcję. Do takich narzędzi należą: generator skanerów Flex (kompatybilny z Lex) i generator parserów Bison (kompatybilny z Yacc).

Wykorzystanie generatorów polega najczęściej na napisaniu plików konfiguracyjnych zawierających definicje tokenów w przypadku skanera i definicję gramatyki w przypadku parsera. Na tej podstawie generatory stworzą kod źródłowy w języku ANSI C skanera i parsera. Kod ten można następnie dołączyć do własnego programu.

2. Praca z programem Flex

Flex, czyli *Fast Scanner Generator*, jest uniwersalnym narzędziem do konstruowania analizatorów leksykalnych. Flex jest kompatybilny z programem Lex. Tworzony przy pomocy Flexa skaner jest opisywany przy pomocy wyrażeń regularnych.

Po uruchomieniu Flex przetwarza podany plik, który zawiera opis skanera i generuje kod źródłowy skanera w języku ANSI C. Proces tworzenia skanera jest pokazany na Rysunku 1.



Rysunek 1: Schemat tworzenia skanera

Skaner jest opisywany przy pomocy reguł składających się z wyrażenia regularnego i odpowiadającego mu fragmentu kodu w języku ANSI C.

```
wyrażenie_regularne   kod_skanera
```

Na przykład reguła:

```
[0-9]+   printf("Wpisano liczbę całkowitą");
```

Powoduje uruchomienie podanej instrukcji `printf` w przypadku napotkania ciągu cyfr o długości co najmniej 1 cyfry.

2.1. Struktura pliku wejściowego

Plik wejściowy zawierający opis skanera ma następującą postać:

```
definicje

%%

reguły

%%

dodatkowy kod użytkownia
```

Część `definicje` zawiera deklaracje etykiet, które są potem wykorzystywane w `regułach`. Deklaracja składa się z nazwy etykiety i odpowiadającego jej wyrażenia regularnego. Przykładem takiej etykiety mogą być etykiety:

```
DIGIT    [0-9]
ID       [a-z][a-z0-9]*
```

Pierwsza z nich określa cyfrę, druga identyfikator, składający się z przynajmniej jednej litery i dowolnego ciągu liter lub cyfr. Zadeklarowanie tego typu etykiet ułatwia pisanie reguł i poprawia czytelność kodu skanera.

Dodatkowo w części `deklaracje` można umieszczać fragmenty kodu, które mają być przekopiowane do kodu źródłowego skanera. Takie fragmenty kodu należy umieścić pomiędzy znakami `%` i `%`.

Część `reguły` zawiera opis skanera w postaci przedstawionej w przykładzie. Wszystkie reguły mają ogólną postać:

```
wzorzec     akcja
```

`Wzorzec` jest specjalnie rozszerzonym wyrażeniem regularnym i może zawierać odwołania do poprzednio zadeklarowanych etykiet. `Akcja` jest najczęściej odpowiadającym `wzorcowi` fragmentem kodu w języku C, który może wykorzystywać specjalne zmienne i funkcje Flexa. Zarówno `wzorce` jak i `akcje` zostaną bliżej przedstawione.

Ostatnia część pliku Flexa to dodatkowy kod w języku C, który jest kopiowany do kodu samego skanera bez modyfikacji. W tej części pliku programista umieszcza własne funkcje wykorzystywane przez niego w skanerze.

2.2. Wzorce

Wzorce wykorzystywane przez Flex do opisu skanera wykorzystują rozszerzone wyrażenia regularne. Przykłady najważniejszych wzorców są pokazane poniżej:

- `x` dowolny podany znak ”x”
- `.` każdy znak, z wyjątkiem znaku nowej linii

- `[xyz]` „klasa znaków", w tym przypadku znak "x", lub "y", lub ź"
- `[abf-mo]` również klasa znaków, lecz z podanym zakresem, w tym przypadku znak ą", lub "b", lub dowolny znak z zakresu "f"do "m", lub znak ó"
- `[^a-z]` zanegowana klasa znaków, czyli w tym przypadku dowolny znak poza małą literą
- `W*` zero lub więcej powtórzeń wyrażenia regularnego W
- `W+` jedno lub więcej powtórzeń wyrażenia regularnego W
- `W?` zero lub jedno powtórzenie wyrażenia regularnego W
- `W3,8` od trzech do ośmiu powtórzeń wyrażenia regularnego W
- {`ETYKIETA`} odwołanie się do zadeklarowanej etykiety
- `"[foo]b̈ar"` napis `[foo]"bar`
-

  `123` znak o wartości 123 podanej ósemkowo
-

  `xa0` znak o wartości a0 podanej szesnastkowo
- `(W)` wyrażenie W, nawiasy są użyte do wymuszenia kolejności dopasowywania
- `WZ` kontakenacja wyrażeń W i Z
- `W|Z` wyrażenie W lub wyrażenie Z
- `W/Z` wyrażenie W, lecz tylko takie, po którym występuje wyrażenie Z
- `^W` wyrażenie W lecz tylko na początku linii
- `W$` wyrażenie W lecz tylko na końcu linii

Dodatkowo dostępne są predefiniowane klasy znaków, z których najważniejsze to:

```
[:alnum:]
[:alpha:]
[:digit:]
[:lower:]
[:upper:]
```

oznaczają one odpowiednio wszystkie znaki alfanumeryczne, alfabetyczne, cyfry, małe litery, wielkie litery. Trzeba pamiętać o wpisywaniu otaczających ich nawiasów klamrowych. Ponieważ klasa znaków jako taka musi być otoczona takimi nawiasami, odwołanie do wyrażenia, które odpowiada wszystkim małym literom wygląda następująco `[[:lower:]]`.

Przy konstruowaniu wyrażeń złożonych trzeba pamiętać o kolejności operatorów. Na przykład wyrażenie:

```
foo|bar*
```

jest równoważne:

```
(foo)|(ba(r*))
```

pasujące do jednego wystąpienia napisu `foo` lub jednego wystąpienia napisu składającego się z `ba` i zero lub większej liczby `r`. Innymi słowy to wyrażenie pasuje do napisów takich jak: `foo`, `ba`, `bar`, `barr`, `barrrrrr` itd.

2.3. Akcje

Każdemu wzorcowi odpowiada pewna akcja, czyli działanie, które ma podjąć skaner po dopasowaniu napisu wejściowego do danego wzorca. Działanie jest fragmentem kodu w języku ANSI C. Jeżeli nie jest podana akcja, to jest przyjmowana akcja pusta, powodująca usunięcie fragmentu napisu pasującego do wzorca ze strumienia wejściowego. Na przykład reguła:

```
"ala ma kota"
```

składa się wyłącznie z wyrażenia regularnego, w tym przypadku napisu, ąla ma kota”. Ponieważ akcja jest pusta, to ten napis zostanie usunięty ze strumienia wejściowego.

Przykład reguły skanera, która zamienia wszystkie ciągi napotkanych spacji lub tabulatorów na pojedynczą spację jest następujący:

```
[ \t]+        putchar( ' ' );
```

Pole działania może zawierać nawiasy klamrowe, które pozwalają na zamieszczanie bloków instrukcji w języku C. Jako działanie można również podać znak pionowej kreski „—”, oznaczający, że dla podanego wzorca ma być przyjęta taka sama reguła jak dla następnego podanego wzorca, na przykład:

```
if|case|while|for {
                  printf( "Słowo kluczowe C");
                  }
```

W akcjach można się odwoływać do globalnego wskaźnika `char *yytext` wskazującego na aktualnie analizowany napis, oraz do zmiennej globalnej `int yylen` zawierającej długość napisu `yytext`. Kod programisty zawarty w akcji możne dowolnie modyfikować napis `yytext`, ale nie może go wydłużać. Można również używać dodatkowych makr i funkcji:

- `ECHO` użycie tego makra powoduje wypisanie `yytext` na standardowe wyjście
- `BEGIN` pozwala na definiowanie dodatkowych pętli analizujących napisy, co jest opisane w podanym dalej przykładzie
- `REJECT` powoduje przejście skanera do następnej pasującej reguły. Na przykład, mając skaner, który oprócz dopasowywania słów powinien je zliczać, można użyć tego makra następująco:

  ```
  %{        int licznik = 0; %}
  %%
  ala       pamietaj(); REJECT;
  [^ \t\n]+ ++licznik;
  ```

  Pojawienie się napisu „ala” spowoduje wywołanie funkcji `pamietaj()`, a następnie przejście do kolejnej reguły, w tym przypadku reguły zliczającej słowa. Nie podanie `REJECT` spowodowałoby, że słowo „ala” nie zostałoby policzone, ponieważ normalnie skaner wykonuje dla każdego tokenu dokładnie jedną regułę.

- `yymore()` użycie tej funkcji powoduje dopisanie aktualnej zawartości `yytext` do następnej wartości. Na przykład reguły:

  ```
  foo-   ECHO; yymore();
  bar    ECHO;
  ```

Spowoduję, że po pojawieniu się na wejściu skanera napisu `foo-bar` zostanie wypisane `foo-foo-bar`. Dzieje się tak dlatego, że pierwsze `ECHO` powoduje wypisanie `yytext`, które na początku będzie zawierało `foo-`, a funkcja `yymore()` sprawia, że następny token, w tym przypadku `bar` będzie dopisany do aktualnego. Dlatego drugie `ECHO` wypisuje `foo-bar`.

- `yyless(n)` zwraca ostatnie `n` znaków napisu z `yytext` z powrotem do bufora wejściowego, co oznacza, że pojawią się w następnej regule.
- `unput(z)` dopisuje znak `z` na początek analizowanego tokenu
- `input()` czyta następny znak ze strumienia wejściowego
- `yyterminate()` kończy pracę skanera

Poza podanym przez programistę zbiorem reguł jest przyjmowana zawsze reguła domyślna, sprawiająca, że jeżeli analizowany znak nie pasuje do żadnego z podanych wzorców, jest on po prostu przekazywany na standardowe wyjście.

3. Przykład skanera

Poniżej przedstawiony jest średnio złożony przykład skanera języka będącego pewnym podzbiorem ANSI C.

```
%{
#include <stdio.h>

#define STRING_BUFFER 256
#define DEFINE_ARRAY 128

int line_number = 0;
char string_buffer[STRING_BUFFER],
  *p_string_buffer;
%}

DIGIT   [0-9]
ID      [[:alpha:]][[:alnum:]]*
INTEGER {DIGIT}+
REAL    {DIGIT}+"."{DIGIT}*

%x string comment

%%

if|case|switch|while|return {
  printf("Słowo kluczowe: %s\n", yytext); }
"+"|"-"|"="|"*"|"/"|"==" {
  printf("Operator: %s\n", yytext); }
{ID} {
  printf("Identyfikator: %s\n", yytext);}
{INTEGER} {
  printf("Liczba całkowita: %s\n", yytext);}
```

```
{REAL} {
  printf("Liczba rzeczywista: %s\n", yytext);}

\"     {
  p_string_buffer = string_buffer;
  BEGIN(string); }
<string>[^\\\n\"]+ {
  char *yptr = yytext;
  while (*yptr)
    *p_string_buffer++ = *yptr++; }
<string>\n {
  line_number++; }
<string>\" {
  BEGIN(INITIAL);
  *p_string_buffer = '\0';
  printf("Napis: \"%s\"\n", string_buffer); }

"/*"    {
  BEGIN(comment); }
<comment>[^*\n]+
<comment>"*"+[^*/\n]*
<comment>\n {
  line_number++; }
<comment>"*"+"/"  {
  BEGIN(INITIAL);
  printf("Komentarz /* (...) */\n");}

[ \t]+
\n     {
  printf("Linia nr: %d\n", line_number++); }

%%
/* Dodatkowy kod */
```

Na początku umieszczone są fragmenty kody niezbędne do poprawnego działania funkcji skanera i do jego kompilacji, czyli deklaracje zmiennych i instrukcje preprocesora.

Dalej deklarowane są etykiety używanych wyrażeń regularnych. Linia `%x string comment` zawiera deklaracje dodatkowych stanów (pętli) skanera, które umożliwiają przetwarzanie komentarzy i napisów.

Skaner zawiera kilka prostych reguł rozpoznających wybrane słowa kluczowe, operatory, a także liczby i identyfikatory. Bardziej interesujące są dwa zestawy reguł rozpoznające łańcuchy znaków, czyli ciągi znaków zawarte pomiędzy znakami `"` i komentarze.

Pojawienie się znaku apostrofu powoduje przejście do wcześniej zadeklarowanej pętli `string` która umożliwia wczytanie całego łańcucha do stworzonego bufora. Jeżeli łańcuch zawiera znak nowej linii to jest on zliczany przy pomocy dodatkowej reguły, ponieważ znajdująca się pod koniec kodu reguła zliczająca nie jest uruchamiana w pętli `string`. Po napotkaniu zamykającego apostrofu jest wypisywany wczytany łańcuch znaków, a skaner wychodzi z pętli `string` i przechodzi do głównej pętli `BEGIN(INITIAL)`.

Rozpoznawanie komentarzy jest zrealizowane analogicznie, ale sama treść komentarzy nie jest zapamiętywana. Są natomiast zliczane znaki nowej linii.

Na końcu kodu są reguły pochłaniające białe spacje i zliczające linie.

4. Generowanie skanera

Aby z przedstawionego powyżej przykładu powstał program wykonywalny realizujący skaner, należy uruchomić program Flex.

```
flex skaner.lex
```

Jeżeli plik `skaner.lex` nie zawiera błędów, jest tworzony plik o domyślnej nazwie `lex.yy.c` zawierający kod źródłowy skanera w ANSI C. Podanie opcji `-o` umożliwia podanie innej nazwy pliku. Opcja `-v` umożliwia wyświetlenie dodatkowych informacji na temat stworzonego skanera. Interesująca może być opcja `-T` pozwalająca na śledzenie pracy Flexa.

W związku z tym, aby wygenerować wykonywalny skaner, należy podać sekwencję podobną do poniższej:

```
flex -v -oakner.c skaner.lex
gcc -Wall -lfl -o skaner skaner.c
```

Biblioteka `fl` zawiera funkcje Flexa konieczne do pracy skanera. W praktyce można jej nie używać po odpowiednim uzupełnieniu kodu samego skanera. W wersji Flex 2.5.4 zawiera ona tylko funkcję `yywrap`. Jeżeli na końcu kodu skanera dopisze się linię

```
int yywrap() {return 1;}
```

to linkowanie z biblioteką `fl` staje się zbędne.

Przykład pracy wy generowanego skanera jest pokazany poniżej:

```
if a=2.2 printf("aaa!");
Słowo kluczowe: if
Identyfikator: a
Operator: =
Liczba rzeczywista: 2.2
Identyfikator: printf
(Napis: "aaa!"
);Linia nr: 0
/* komentarz */
Komentarz /* (...) */
Linia nr: 1
```

Podczas testowania skanera przydatne jest skompilowanie go z włączonym *debuggingiem*, czyli z opcją `-d` Flexa. Wtedy można dokładnie obserwować jego pracę:

```
--(end of buffer or a NUL)
d="cos"
--accepting rule at line 25 ("d")
Identyfikator: d
--accepting rule at line 23 ("=")
Operator: =
--accepting rule at line 31 (""")
--accepting rule at line 34 ("cos")
--accepting rule at line 40 (""")
```

```
Napis: "cos"
--accepting rule at line 54 ("
")
Linia nr: 0
```

Wszystkie linie zaczynające się od `--` są generowane automatycznie, a numery reguł odpowiadają numerom w kodzie opisującym skaner.

Komunikacja z parserem odbywa się przez zmienne zewnętrzne `char *yytext`, `int yylval`. Wygenerowany przez Flex skaner ma postać funkcji `int yylex()` która wczytuje po jednym tokenie i zwraca jego numer, oraz ewentualnie jego wartość dodatkową - np. w przypadku zmiennych. Strumień wejściowy jest definiowany przez wskaźnik `FILE *yyin`.

5. Podsumowanie

Flex jest narzędziem mającym wiele zastosowań. Pozwala na tworzenie złożonych skanerów umożliwiających analizę leksykalną dowolnych plików. Przedstawiony przykład prezentuje zaledwie ułamek możliwości programu. Szerokim polem zastosowań Flexa jest wspomniane we wstępie tworzenie skanerów dla parserów generowanych przez GNU Bisona.

Analiza języków formalnych jest jednym z ciekawszych pól zastosowań informatyki. Warto uświadomić sobie fakt, że dotyczy ona nie tylko języków programowania, takich jak np. ANSI C, lecz również języków formalizujących zapis dokumentów, takich jak zyskujący popularność XML. Również w tej dziedzinie Flex i Bison mogą znaleźć liczne zastosowania.