Kraków, 9 czerwca 2007

Krótki tutorial z Flex'a

1 Wprowadzenie

Jedną z metod wykorzystania komputerów jest pisanie programów w różnych językach językach programowania, które następnie trzeba przetworzyć na postać wykonywalną przez komputer. W tym celu używa się translatorów, które albo tłumaczą program na pewną formę pośrednią, albo odrazu tworzą formę kodu maszynowego przeznaczoną do bezpośredniego wykonania na komputerze.

Wspólną cechą wszystkich języków, które są poddawane automatycznej translacji jest ich sformalizowanie.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. Przykładem sformalizowanego opisu języka można znaleźć w Internecie, np. w Wikipedi pod adresem
`http://pl.wikipedia.org/wiki/BNF` opisana jest budowa liczby naturalnej za pomocą notacji BNF. Podobnie został opisany język Pascal w książce Wirth'a "Algorytmy+Struktury Danych = Programy".

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.

2 Automatyczne narzędzia typu 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.

Poniżej przedstawiono proces tworzenia analizatora leksykalnego na komputerze `artemis.wszib.edu.pl`:

1. najpierw logujemy się komputerze `artemis.wszib.edu.pl` używając swojego normalnego *logina* i *hasla*.
2. później w dowolnym edytorze tworzymy plik z opisem skanera np *nazwa.lex*,
3. następnie za pomocą komendy $flex\ -o$ "$nazwa.c$" "$nazwa.lex$" tworzymy program w języku c.
4. w końcu za pomocą komendy $gcc\ -o$ "$nazwa$" "$nazwa.c$" $-lfl$ tworzymy program wynikowy,
5. program wynikowy *nazwa* można w tym momencie uruchomić jak każdy inny program.

3 Struktura pliku z opisem skanera

Ogólnie struktura pliku z opisem skanera wygląda następująco:

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

Część definiująca 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]*
```

gdzie pierwsza etykieta określa cyfrę, a druga identyfikator zaczynający się od litery.

Część definiująca reguły ma ogólną postać:

```
wzorzec     akcja
```

Wzorzec jest rozbudowanym wyrażeniem regularnym mogącym również zawierać odwołania do poprzednio zdefiniowanych etykiet.

Przykłady wyrażeń regularnych:

- x – pojedyncza litera "x",
- . – pojedyncza litera, z wyjątkiem znaku nowej linii,
- $[abc]$ – klasa znaków, w tym przypadku "a" lub "b" lub "c",
- $[abc - fg]$ – klasa znaków, w tym przypadku "a", "b", "c", "d", "e", "f" lub "g",
- $[\wedge a - z]$ – negacja klasy znaków, w tym przypadku bez małych literek,
- $W*$ – 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,
- 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,
- $\wedge W$ wyrażenie W lecz tylko na początku linii,
- $W\$$ wyrażenie W lecz tylko na końcu linii.

Oprócz powyższej możliwości definiowania wzorców dopasowania we flexie istnieje kilka klas predefiniowanych, z których najważniejsze to:

1. [: $alnum$:] – wszystkie znaki alfanumeryczne,
2. [: $alpha$:] – wszystkie litery,
3. [: $digit$:] – cyfry,

4. [: *lower* :] – małe litery,

5. [: *upper* :] – duże litery.

Uwaga:
Klasa znaków przy odwołaniu musi być otoczona nawiasami kwadratowymi, zatem odwołanie do powyższych klas ma postać [[: *lower* :]].

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

- $foo|bar*$– jest równoważne wyrażeniu $(foo)|(ba(r*))$, do którego pasuje słowo foo lub słowo ba z dowolną ilością literek r, czyli np. $ba, bar, barr, \ldots$.

Wyrażenia regularne flex'a można przetestować pod adresem:
`http://www.stratulat.com/technical/regexpflex/`

4 Akcje

Jak wcześniej zostało wspomniane, każdemu wzorcowi odpowiada pewna akcja, zgodnie ze schematem:

```
wzorzec     akcja
```

W przypadku kiedy akcja nie jest określona czyli jest pusta spowoduje to usunięcie wzorca ze strumienia wejściowego, przykładowo reguła

```
pies
```

spowoduje usunięcie ze strumienia wejściowego napisów "pies".

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

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

Przykład zamiany słów kluczowych na napis "Słowa kluczowe":

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

W akcjach można wykorzystać globalny wskaźnik $char * yytext$ wskazującego na aktualnie analizowany napis, oraz do zmieniej globalnej $int\ yylen$ zawierającej długość napisu $yytext$. Kod zawarty w akcjach może w dowolny

sposób modyfikować napis *yytext*, ale nie może go wydłużać.

Oprócz zwykłego kodu dostępnego w C, można również używać specjalnych makr:

- *ECHO* – powoduje wypisanie *yytext* na standardowe wyjście,
- *BEGIN* – pozwala na zdefiniowanie dodatkowej pętli analizującej napisy,
- *REJECT* – powoduje przejście skanera do następnej pasującej reguły.

  W przykładzie:

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

  po podaniu napisu "ala" wywołana zostanie funkcja *pamietaj*(), a następnie przejście do następnej reguły w tym przypadku reguły zliczającej słowa. W przypadku gdyby opuszczono słowo *REJECT*, słowo "ala" nie zostałoby policzone, ponieważ normalnie jest wykonywana tylko reguła powiązana z pierwszym dopasowanym wzorcem.

- *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 $foo-foo-bar$. 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 tez znaki się pojawią w następnej regule.
- *unput*(z) – dopisuje znak z na początek analizowanego tokenu.

- *input*() – czyta następny znak ze strumienia wejściowego.
- *yyterminate*() – kończy pracę skanera

Oprócz podanego przez programistę zestawu reguł, mamy jeszcze jedną standardową regułę, sprawiającą, że jeśli aktualnie analizowany znak nie pasuje do żadnej reguły to jest on on przekazywany na standardowe wyjście.

5 Większy Przykład

Poniżej przedstawiono dłuższy przykład skanera analizującego język będący pewnym podzbiorem języka *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 */
```

Przykład pracy takiego skanera dla strumienia wejściowego:

```
if a=2.2 printf("aaa!");
/* głupi przykład */
```

Daje wynik:

```
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 opcją $-d$, wtedy można obserwować pracę tego skanera:

```
--(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
```

6 Dodatkowe proste przykłady

Dodatkowe przykłady różnych skanerów zostały umieszczone w pliku *Samples.zip*. Dla każdego z umieszczonego w tym pliku przykładów proszę:

1. Postarać się obejrzeć kod skanera i bez testowania odpowiedzieć co dany skaner robi,
2. Wygenerować z pliku zawierającego kod skanera kod wykonywalny,
3. Sprawdzić, czy rzeczywiście kod wynikowy robi to co przeidzieliście państwo w punkcie 1.

7 Bibliografia

Gdzie można znaleźć więcej opisów, narzędzi do testowania:

1. `http://man.przez.net/flex.1.html`,
2. `http://www.stratulat.com/technical/regexpflex/`,