Flex tutorial

Dévai Gergely

A Flex (Fast Lexical Analyser) egy lexikáliselemző-generátor: reguláris kifejezések sorozatából egy C/C++ programot generál, ami szövegfájlokat képes lexikai elemek sorozatára tördelni.

A Flex forrásfájlok három részből állnak, ezeket `%%`-ot tartalmazó sorok választják el egymástól. Az első részbe konfigurációs opciók és (`%{` és `%}` között) C++ kód írható, valamint a gyakran előforduló reguláris kifejezésekhez makrók definiálhatók. A második részbe a reguláris kifejezések, a harmadikba ismét tetszőleges C++ kód kerül:

```
opciók

%{
   C++ kód
%}

makrók

%%

reguláris kifejezések

%%

C++ kód
```

Összefoglaló a Flex által használt reguláris kifejezésekről:

`x`	az `x` karakter
`.`	tetszőleges karakter, kivéve újsor
`[xyz]`	karakterhalmaz: vagy egy `x`, vagy egy `y`, vagy egy `z`
`[abj-oZ]`	karakterhalmaz intervallummal
`[^A-Z]`	komplementer halmaz: bármilyen karakter kivéve a nagybetűket
`[^A-Z\n]`	bármilyen karakter kivéve a nagybetűket és újsort
r`*`	nulla vagy több *r*, ahol *r* tetszőleges reguláris kifejezés
r`+`	egy vagy több *r*
r`?`	nulla vagy egy *r* (opcionális *r*)
r`{2,5}`	kettőtől ötig valahány *r*
r`{2,}`	kettő vagy több *r*
r`{4}`	pontosan négy *r*
`{`*name*`}`	a deklarációs részben magadott *name* makró kifejtése
`"[xyz]\"foo"`	az `[xyz]"foo` sztringliterál
`\`*x*	a sepciális karakterek, pl. `\n` `\t` stb.
`\x2a`	a *hexadecimális 2a* kódú karakter
`(`*r*`)`	*r*; a zárójelek a műveleti sorrend jelölésére használhatók
rs	*r* után egy *s* (konkatenáció)
r`|`*s*	*r* vagy *s* (unió)
`^`*r*	*r* a sor elején
r`$`	egy *r* a sor végén

1. példa

Ez a tutorial a `http://deva.web.elte.hu/fordprog/flex-peldak.zip` címen elérhető példasort használja, ezt érdemes letölteni a következők elolvasása előtt.

1.1. Az `1/flex1.l` fájl tartalma

- Opciók:
  - `noyywrap`: A forrásfájl végén az elemzésnek is vége lesz.
  - `c++`: A generált program C++ nyelvű lesz.
- C++ kód: A szükséges fejállományok beillesztése.
- Reguláris kifejezések: Ebben a példában egy sincs.
- C++ kód: Az egyszerűség kedvéért ide írjuk a `main` függvényt.
  - Megnyitjuk az `input.txt` fájlt.
  - Létrehozunk egy lexikális elemző objektumot (`fl`) a flex által generálandó osztályból (`yyFlexLexer`). A konstruktorparaméterekkel adjuk meg, hogy honnan olvasson és hova írjon az elemző.

– Az `yylex()` tagfüggvény meghívásával indítjuk el az elemzést.

Abban az esetben, ha a Flex által generált lexikális elemző a forrásfájlban lexikális hibát talál, azaz a megadott reguláris kifejezések közül egyikre sem illeszthető az input, akkor az aktuális karaktert a megadott kimenetre továbbítja. Mivel ebben a példában egyetlen reguláris kifejezést sem adtunk meg, a teljes input a kimenetre kerül. Ez a lexikális elemző tehát a Unix `cat` parancsának bonyolult megvalósítása. :)

1.2. Fordítás és futtatás

- `cd 1`
- `flex flex1.l`
- Ekkor keletkezik a `lex.yy.cc` fájl, benne kb. 1500 sornyi C++ kóddal. Megatalálhatóak benne a `flex1.l` fájlban általunk írt kódrészletek, pl. a `main` függvény kódja is.
- `g++ -o flex1 lex.yy.cc`
- A fordítás eredménye a `flex1` futtatható állomány, a lexikális elemző.
- `./flex1`
- A futtatás során az `input.txt` tartalma jelenik meg a standard kimeneten.

2. példa

Az előző programot úgy módosítjuk, hogy az input szövegben a `username` szó összes előfordulását cserélje le egy megadott felhasználói névre (pl. `deva`). Ehhez a `username` szóra illeszkedő reguláris kifejezést kell írni, ami a `username`.

Flexben a reguláris kifejezésekhez C++-ban írt akciókat lehet társítani. Amikor a lexikális elemtő a megtalál egy lexikai elemet, a megfelelő reguláris kifejezéshez csatolt kódrészlet lefut. A `username` reguláris kifejezéshez tehát a megfelelő felhasználóinevet kiíró kódrészletet kell csatolni, ezért kerül a `flex2.l` fájlba a következő sor:

```
username cout << "deva";
```

Az input minden olyan része, amely erre a reguláris kifejezésre nem illeszkedik, változtatás nélkül kerül a kimenetre.

A projekt fordítása és futtatása az 1.2. pontban írtakkal analóg módon végezhető.

3. példa

Olyan eszközt szeretnénk, amely C/C++ programkódokat HTML oldalon megjeleníthető formára alakít. Ehhez a HTML számára jelentéssel bíró karaktereket (pl. <, >) és az indentálás miatt fontos fehér szóközöket (szóköz, tab, sorvége) a nekik megfelelő HTML kódra kell cserélni. Ennek megvalósítása látható a `flex3.l` fájlban.

4. példa

Az input szöveg minden sora helyett annak sorszámát és hosszát szeretnénk kiírni (a sorvége karaktert is beleszámolva). Például az

```
egy
ketto
harom
```

input esetén a következő eredményt várjuk:

```
1 4
2 6
3 6
```

A megoldás készítésekor feltételezzük, hogy minden sor (az utolsó is) sorvége karakterrel terminálva van.

4.1. Első megoldás

A Flex forrás első részében található C++ kódban változókat is definiálhatunk, amelyek használhatók a reguláris kifejezésekhez társított akciókban is. Létrehozunk tehát két változót az aktuális sor és oszlop pozíciók nyilvántartása céljából és kezdőértéket is adunk nekik.

Ha sorvége karaktert találunk, akkor ki kell írni a két változó aktuális értékét, majd inkrementálni kell a sorok számát, és alaphelyzetbe kell állítani az oszlopok számát. Ha tetszőleges más karakter jön, akkor az `oszlop` változót kell inkrementálni. Ez a megoldás a `flex4.l` fájlban található.

4.2. Második megoldás

A Flex által generált C++ osztály biztosítja az `YYLeng()` tagfüggvényt, ami az aktuális lexikális elem hosszát adja meg. Ezt felhasználva is megoldhatjuk a feladatot, de ebben az esetben teljes sorokra illeszkedő reguláris kifejezést kell írni:

```
.*\n    {
          cout << sor << ' ' << YYLeng() << endl;
          ++sor;
        }
```

Ezt a megoldást használja a `flex4v2.l` fájl.

5. példa

Az utolsó példában az input szöveget szavakra szeretnénk tördelni, és minden szóhoz megadni az első karakterének sor/oszlop pozícióját. A *szó* ebben a feladatban a bemenet olyan (nem bővíthető) részét jelenti, amely nem tartalmaz fehér szóközöket.
Példa bemenet:

```
Ez    egy
tobb    sorbol allo
    szoveg.
```

Példa kimenet:

```
1 1 Ez
1 7 egy
2 1 tobb
2 9 sorbol
2 16 allo
3 5 szoveg.
```

5.1. Első megoldás

A `flex5.l` fájlban implementált megoldás is egy-egy változót használ a sor és oszlop információk tárolására, és három reguláris kifejezést definiál:

- Újsort, tabot és szóközt nem tartalmazó, nem üres sorozat: `[\n\t ]+`
  Ebben az esetben a felismert szó pozícióját és — az `YYText()` függvény segítségével — magát a szót is a kimenetre kell írni, majd az oszlop változót a szó hosszával (`YYLeng()`) növelni kell.
- Újsor: `\n`
  A sorok száma eggyel nő, az `oszlop` változó alapértékre áll vissza.
- A fenti reguláris kifejezésekre nem illeszkedő, tetszőleges karakter (esetünkben tab és szóköz): `.`
  Az `oszlop` változót kell inkrementálni.

5.2. Második megoldás

A `flex5v2.l` megoldás kihasználja, hogy a lexikális elemző számon tudja tartani a sorok számát az `yylineno` opció használata esetén. Az előző megoldás szerkezete nem változik, csupán a sor változót váltjuk ki a `lineno()` függvény hívásával.

6. Tippek a beadandó elkészítéséhez

- A beadandó lexikális elemző elkészítéséhez érdemes tanulmányozni a következő példát:
  - `http://deva.web.elte.hu/fordprog/lexikalis-pelda.zip`
  - Ebben a példában már nem a Flex forrás végén található a `main` függvény, hanem külön forrásfájlban. A C++ fordítónak tehát két fájlt (`lex.yy.cc` és a `main` függvényt tartalmazó fájl) kell megadni.
- A reguláris kifejezések sorrendje fontos! Ha az input több reguláris kifejezésre is illeszkedik (pl. a `while` egy kulcsszó, de akár változónév is lehetne), akkor a sorrendben korábbi kategóriát választja az elemző. Ebből következik, hogy a kiemelt lexikális elemeket (pl. kulcsszavak) a lista elején kell megadni.
- Az egyes kulcsszavakhoz érdemes külön-külön reguláris kifejezést készíteni, azaz

```
while | if | then     {
                         std::cout << „kulcsszo: ”
                         << YYText() << std::endl;
                       }
```

  helyett jobb megoldás a következő:

```
while     std::cout << „kulcsszo: ” << YYText() << std::endl;
if        std::cout << „kulcsszo: ” << YYText() << std::endl;
then      std::cout << „kulcsszo: ” << YYText() << std::endl;
```

  Ennek fontossága az első beadandónál még nem látszik, de a szintaktikus elemző számára majd fontos lesz, hogy különbséget tudjon tenni ez egyes kulcsszavak között.
- Ha az automatikus ellenőrző szoftver nem fogadja el a beadott megoldást, érdemes ellenőrizni, hogy hibás bemenet esetén valóban `1` visszatérési értékkel fejeződik-e be az elemző. (Gyakorlatban: `exit(1)` utasítás a hibajelzés után.)