Bisonc++ tutorial

Dévai Gergely

A Bisonc++ egy szintaktikuselemző-generátor: egy környezetfüggetlen nyelvtanból egy C++ programot generál, ami egy tokensorozat szintaktikai helyességét képes ellenőrizni.

A Bisonc++ forrásfájlok két részből állnak, ezeket `%%`-ot tartalmazó sorok választják el egymástól. Az első részbe konfigurációs opciók és a tokentípusok definíciói írhatók. A második részbe a nyelvtan szabályai kerülnek:

```
opciók

tokendefiníciók

%%

nyelvtan
```

A nyelvtanleírás konvenciói:

- A kezdőszimbólum neve `start`.
- A terminálisok (tokenek) nagybetűsek, a nemterminálisok kisbetűsek.
- A szabály bal- és jobboldalát `:` választja el egymástól.
- Az alternatívák között `|` szerepel.
- A szabályalternatívák sorozatát `;` zárja le.
- C++ stílusú megjegyzések írhatók a szabályokhoz.
- Az ϵ-t üres szabályjobboldal valósítja meg, a gyakorlatban egy `// ures` megjegyzést szokás írni helyette.
- A jobboldalak után `{` és `}` között C++ kód írható, ami mindannyiszor végrehajtódik, amikor az adott szabályt az elemző használja.

Ezek alapján a
$S \to aC \,|\, C$
$C \to \epsilon \,|\, bC$
nyelvtannak a következő felel meg:

```
start:
  A c
|
  c
;

c:
  // ures
|
  B c
;
```

1. példa

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

1.1. Az `1/lista.y` fájl tartalma

- Opciók: `%baseclass-preinclude <iostream>`
  A generálandó osztályhierearchia ősosztályába beilleszti az `iostream` fejállományt. Ez azokban a példákban lesz majd fontos, ahol a szabályokhoz csatolt akciókban írni akarunk a standard outputra.

- Tokenek: `%token ELEM NYITO CSUKO VESSZO`
  A tokentípusokat a `%token` direktíva segítségével definiáljuk. A nyelvtan terminálisai az itt felsorolt négy elem, melyekből a Bisonc++ az általa generálandó `Parser` osztályba egy felsorolási típust fog létrehozni.

- A nyelvtan egy zárójelbe tett, vesszővel elválasztott elemekből álló lista szintaxisát adja meg.

1.2. Az `1/lista.l` fájl tartalma

Ez egy Flex forrásfájl, melyről részletes leírás itt található:
`http://deva.web.elte.hu/fordprog/flex-help.pdf`
A szintaktikus elemzés számára fontos részletek a következők:

- `#include "Parserbase.h"`
  Ezt a fejállományt a Bisonc++ fogja generálni. Beillsztésével láthatóvá tesszük a `lista.y` fájlban megadott tokeneket.

- `return Parser::ELEM;`
  Az egyes reguláris kifejezések sikeres illesztésekor a lexikális elemző vissza fog térni a megfelelő tokennel.

1.3. Az `1/lista.cc` fájl tartalma

Ez a C++ forrás tartalmazza a `main` függvényt, amelyben ellenőrizzük a panancssori argumentum meglétét és megpróbáljuk megnyitni a megadott fájlt. Ha ez sikeres, akkor ezzel az inputtal létrehozunk egy szintaktikus elemző objektumot (`pars`), melynek `parse()` metódusával indítjuk el az elemzést.

1.4. Fordítás I.

- `cd 1`
- `flex lista.l`
- `bisonc++ lista.y`
- Ekkor keletkeznek a `Parserbase.h`, `Parser.ih`, `Parser.h`, `parse.cc` állományok. A `Parser.ih` és `Parser.h` állományokat szerkeszthetjük, mert a Bisonc++ a következő futtatásakor nem fogja felülírni ezeket a fájlokat.

1.5. Az 1/Parser.h fájl

Ez a fejállomány definiálja a `Parser` osztályt. Ahhoz, hogy a szintaktikus elemző együtt tudjon működni a lexikális elemzővel, include-oljuk a `FlexLexer.h` fejállományt, felvesszük a lexikális elemzőt a `Parser` osztály adattagjai közé (`lexer`), és hozzáadunk az osztályhoz egy konstruktort, ami a kapott bemeneti adatfolyammal inicializálja a lexert. (Ezt a konstruktort hívtuk meg a main függvényben.)

1.6. Az 1/Parser.ih fájl

Ebben az implementációs fejállományban az `error` tagfüggvény átírásával szabhatjuk testre a hibaüzeneteket. Ez a fejállomány definiálja továbbá a `Parser` osztály `lex()` függvényét: Valahányszor a szintaktikus elemzőnek szüksége van a szöveg következő tokenjére, ezt a függvényt hívja meg. Ebben a példában ennek a függvénynek összesen annyi a teendője, hogy meghívja a `Parser` osztály adattagjai közé felvett lexikális elemző objektum `yylex()` metódusát, és a kapott eredményt adja vissza. Ez az eredmény az, amit a Flex forrásfájlban látható `return` utasítások adnak.

1.7. Fordítás II.

- `g++ -o lista lista.cc lex.yy.cc parse.cc`
- A fordítás eredménye a `lista` futtatható állomány, a szintaktikus elemző.
- `./lista jo.txt`
- `./lista hibas.txt`

- A fordítás a mellékelt `Makefile` segítségével is elvégezhető a `make` parancs kiadásával.

2. példa

A `2` könyvtárban található példa C stílusú függvénydeklarációk sorozatát képes elemezni. Nincs benne újdonság az előző példához képest, ezért érdemes abból kiindulva gyakorlatként megcsinálni: A `jo.txt` és `hibas.txt` fájlok alapján kitalálható, hogy milyen nyelvet szeretnénk elemezni.

A `2-hibakezeles` könyvtár tartalma azt mutatja meg, hogyan lehet jobb hibaüzenetet adni szintaktikus hiba esetén, illetve ilyen esetben is tovább folytatni az elemzést.

- A Flex forrásfájlban az `yylineno` opció segítségével gondoskodunk róla, hogy a lexikális elemző számlálja a sorokat.
- A `Parser.ih` fájlban a `lex` függvényben a `lineno` metódussal kérjük el a lexikális elemzőtől az aktuális sorszámot, és ezt a `Parser` osztály `d_loc__` adattagjának egyik mezőjébe mentjük el.
- A `Parser.ih` fájlban definiált `error` függvényt úgy módosítjuk, hogy felhasználja ezt a helyinformációt.
- Az `fl.y` fájl nyelvtani szabályait kibővítjük úgy, hogy használja a speciális `error` nemterminális szimbólumot. Ha az elemző szintaktikus hibát észlel, akkor megpróbálja illeszteni az `error`-t tartalmazó hibaalternatívákat. Vigyázni kell arra, hogy mindig egy jól meghatározott terminális zárja le ezeket a hibaalternatívákat, különben könnyen konfliktusokat okoznak a nyelvtanban.

Javaslat: A beadandót először a hibaalternatívák nélkül érdemes elkészíteni, és ha elfogadásra került, akkor lehet próbálkozni az `error` szimbólumok bevezetésével.

3. példa

Ez a példa a `begin` és `end` kulcsszavakkal körbezárt blokkokból és `skip` kulcsszavakból álló nyelvet elemzi. Az üres fájl helyes, valamint `skip`-ek és blokkok tetszőleges sorozata is helyes. Egy blokkban szintén tetszőleges `skip` illetve blokk sorozat lehet, az üres blokkok is megngedettek. Éredemes ezt a példát a `jo.txt`-ben adott példa alapján gyakorlásként megoldani.

4. példa

Ez a példa a nulladrendű logikai formulákat képes elemezni. A formulák logikai literálokból, változókból és logikai összekeötő jelekből (negáció, konjunkció,

diszjunkció, implikáció, ekvivalancia) állnak. A nyelvtan nagyon egyszerű, a formulák megadása a következő sémát követi:

```
formula:
IGAZ
...
formula VAGY formula
|
formula ES formula
;
```

Ez a nyelvtan azonban nem egyértelmű! Például a `true & true & true`, vagy a `true & true | true` formulákhoz több különböző szintaxisfa is rajzolható. Ezek közül a helyes fát a logikai összekötő jelek bal- illetve jobbasszociativitása és a rájuk vonatkozó precedenciaszabályok határozzák meg. Ezeket az információkat a Bisonc++ forrás elején adjuk meg a következő formában:

```
%right EKV
%right IMPL
%left VAGY
%left ES
%right NEM
```

A `%right` és `%left` direktívák a jobb- illetve balasszociativitást szabályozzák, míg a sorrend a precedenciát adja meg növekvő sorrendben. Ezt a technikát a beadandóban szereplő aritmetikai és logikai operátorok esetén is érdemes alkalmazni.