Kombinátoros szintaktikai elemző a Clean nyelvhez

TDK dolgozat

2002.

Diviánszky Péter
Debreceni Egyetem TTK

Témavezetők:

Horváth Zoltán
ELTE ÁSZT
Csörnyei Zoltán
ELTE ÁSZT

Készült az OTKA 37742 sz. pályázat támogatásával.

Kivonat

A Clean egy tisztán funkcionális nyelv, melynek szintaktikai elemzőjét ad-hoc stílusban írták. Az elemzőt újraírtam kombinátorok használatával, megírva ezzel az első ilyen stílusú valós elemzőt. Az így kapott elemző sokkal inkább beleillik a funkcionális gondolatkörbe, és lényegesen áttekinthetőbb is, ami nem csupán esztétikai jeletőségű, hanem segíti a még hátralevő fejlesztéseket, például a Haskell interfész implementálását.

Tartalomjegyzék

1. Bevezetés

A Clean egy, még fejlesztés alatt álló, tisztán funkcionális nyelv [3]. Az idő előrehaladtával a fejlesztőgárda a fordítót átírta C-ről Clean nyelvre, és közkívánatra 2001 végén közreadta a fordító éppen aktuális forráskódját. Úgy tűnik, idő hiányában a kódon semmit sem változtattak, mielőtt kitették a honlapra (www.cs.kun.nl/~clean). Ez az időhiány meglátszik a nyelv szintaktikai elemzőjén is: A szöveg nem tartalmaz megjegyzést, még magán viseli a C nyelv jellegzetességeit, és észrevehetők rajta a különböző gyors kiegészítések nyomai.

Úgy határoztam ezért, újraírom az elemzőt Cleanhez méltóbb, funkcionális stílusban. Munkamódszerem a következő volt: Nem akartam egyik kialakult irányvonalat sem követni, hanem a saját ötleteimre hagyatkoztam. Így akaratlanul is a kombinátoros elemzőkhöz jutottam, bár az általam alkalmazott kombinátorok különböznek a szakirodalomban találhatóaktól [6, 5, 4]. A munkámat még nem tartom teljesnek; a következő lépés egy parser generátor lenne, azonban érdemes megállni és értékelni a kombinátoros megoldást.

Új eredmények: Új típusozással láttam el és kiszélesítettem az funkcionális elemzők körét úgy, hogy ezekre az általánosabb elemzőkre is definiálhatóak az elemi kombinátorok. A másik eredmény, hogy elkészült egy olyan kombinátoros elemző, amely egy, a gyakorlati életben is használt programnyelv teljes nyelvtanát elemzi. Tudtommal ez az első ilyen méretű kombinátoros elemző. A kész szintaktikus elemző forráskódja megtalálható a ??? honlapon.

A dolgozat első részében bemutatom az általam használt kombinátoros elemzőket, a második részben pedig nagy vonalakban bemutatom az ezekkel elkészített Clean forráskód elemzőt.

1.1. Elnevezések

A dolgozatban elemző alatt mindig szintaktikai elemzőt értek. Használni fogom a következő kifejezéseket is: kombinátoros elemző (combinator parser), elemző kombinátor (parser combinator), szerkezeti szabályok (layout rules), határpozíció (offside position).

2. Az általam használt kombinátorok

Az első alfejezet egy rövid Clean bevezetés, ahol a később felhasznált nyelvi elemeket ismertetem nagy vonalakban. A második alfejezetben bemutatok egy rövid gondolatmenetet, ami elvezet a kombinátoros elemzők használatához. Az ezután következő alfejezetekben pedig a szakirodalom követésével, lépésről–lépésre mutatom be az általam használt kombinátorokat.

2.1. Röviden a Clean nyelvről

A dolgozat megértését nagyban elősegíti a Clean, vagy más funkcionális nyelv ismerete. Itt csak néhány nélkülözhetetlen jellemzőjét írom le, részletes dokumentáció található a Clean honlapján (www.cs.kun.nl/~clean), magyar nyelvű bevezetés pedig [9]-ban található.

A Clean egy tisztán funkcionális nyelv. Függvényekből építkezik, amiket definiálhatunk névvel együtt, vagy név nélkül.

A névvel ellátott függvények definiciója két részből áll, a típusdefinícióból (ez el is hagyható) és a függvénytörzs definiálásából:

függvénynév :: *típus*
függvénynév argumentumok = *kifejezés*

Például

```
distance :: Int Int -> Int
distance x y = abs (x - y)
```

A funkcionális nyelvekben az argumentumokat zárójelek és vesszők nélkül soroljuk fel. Abban az esetben, ha a függvény két argumentumot vár, infix jelölést is alkalmazhatunk (mint a példabeli kivonás), ezt azonban korábban jelölni kell a típusdefinícóban azzal, hogy a nevet zárójelbe tesszük. Lehetőség van az asszociativitás jelölésére az `infix`, `infixl`, `infixr` kulcsszavak valamelyikével és utána meg adhatjuk a precedenciát is:

```
(distance‘) infix 3 :: Int Int -> Int    // nem asszociatív
```

A függvénytörzs megadásakor a `where` kulcsszó után megadhatunk lokális definíciókat:

```
square_square_square x = z * z
where
    z = y * y
    y = x * x
```

Név nélküli függvényeknél a függvény neve helyén a rep kulcsszó áll: `\x y= abs(x-y)`. Az így definiált függvénynek nem adhatunk meg típust (ezt a fordító vezeti le helyettünk), és a függvényre nem is hivatkozhatunk máshol, viszont felhasználhatjuk kifejezésként:

```
identity_function =  \ x = x
```

Típus megadásakor a következő építőelemek állnak rendelkezésünkre:

- Az alaptípusok: `Int`, `Real`, `Char`, `File`, stb.
- Típusfüggvények, ezeknek gyakran speciális jelölésük van:
  - [t] jelöli a t típusú listát
  - (t_1 , t_2) azokak a rendezett pároknak a típusa, melyekben az első elem t_1 típusú, a második elem t_2 típusú

– egyéb nagybetűvel kezdődő azonosító, ha utána paraméterek következnek

- Függvénytípus jelölése: $t_1\ t_2\ \ldots t_n$ `->` t, ahol $t_i, i = 1, \ldots, n$ az argumentumok típusa, t pedig a visszaadott érték típusa. A balranyíl jobbra csoportosít. Szemantikailag az előbbi típus ekvivalens a t_1`->`t_2`->`$\ldots$`->`t_n`->`t típussal, ennek megfelelően a függvények hívhatóak kevesebb argumentummal, mint azt a következő ekvivalens átalakítások mutatják:

```
increment x = add 1 x
increment‘ x = (add 1) x
increment‘‘ = add 1
```

- Kisbetűvel kezdődő azonosító típusparamétert jelöl. Helyére tetszőleges típus helyettesíthető (univerzális polimorfizmus). Például az identikus függvény típusa `a -> a`.
- A típus előtti `*` egyszeres hivatkozást jelöl, azaz a `*`-gal megjelölt típusú változókra biztosított, hogy egyidőben legfeljebb egy helyen hivatkozunk rájuk, így lehetővé válik a változó tartalmának felülírása funkcionális környezetben is. Így oldották meg Cleanben többek között a fájlműveleteket.

A *név* $a_1, \ldots, a_n$ `:==` $B(a_1, \ldots, a_n)$ szerkezet a Cleanben makródefiníciót jelöl. Definiálhatunk függvénymakrót és típusmakrót. Ezek közös tulajdonsága, hogy a fordító minden *név* előforduláskor feltétel nélkül helyettesítést véges a paraméterek megfelelő beillesztésével.

A típusmakrókat `::` vezeti be:

```
:: Dubble a :== (a,a)
```

Függvénymakróknál megadhatunk asszociativitást és precedenciát is:

```
(o) :: infixr 9
(o) f g :== \ x = f (g x)    // függvény kompozíció
```

2.2. Hogyan jutottam el a kombinátoros elemzőkig?

Tekintsük egy függvényt, ami egy logikai értéket is visszaad. Előfordulhat, hogy a függvény minden hívása után a kapott logikai érték szerinti feltételvizsgálat van. Nem lehetne-e ezt a feltételvizsgálatot bevinni a függvény törzsébe? A gondolatmenet alapján az úgynevezett folytatás stílushoz jutunk [1]: az új függvény argumentumként megkapja a feltételvizsgálathoz szükséges programágakat, és a feladat elvégeztekor a logikai értéknek megfelelően adja tovább az eredményül kapott többi értéket és vezérlést az egyik vagy a másik programágnak.

Megmutatjuk, hogy az ilyen függvények használata valóban kényelmesebb. Térjünk vissza az elemzőkhöz! Feltételezzük, hogy van egy programállapotunk, ami tartalmazza többek között az elemzésre szánt fájlt, és a pozíciót. Legyen a programállapot típusa `State`. Tegyük még fel, hogy már megvannak a következő függvényeink:

```
string :: String (*State -> *State) (*State -> *State) *State -> *State
skipChar :: (*State -> *State) *State -> *State
```

A `string` függvény ellenőrzi, hogy az aktuális pozíciótól kezdve megtalálható-e az inputban az első argumentumként kapott sztring. Ha igen, a pozíciót a sztring hosszával módosítja és a végrehajtást a második argumentumával folytatja. Ha nem, akkor a harmadiknak adja át a vezérlést. A `skipChar` függvény eggyel növeli a pozíciót, majd az első argumentumának adja a vezérlést.

Ezekkel a függvényekkel így adható meg az a függvény, ami kihagyja a `/*`-gal kezdett megjegyzés hátralevő részét (a megjegyzések egymásba is ágyazhatóak):

```
skipComment :: (*State -> *State) *State -> *State
skipComment continue state =
    string "*/"
        continue
        (string "/*"
            (skipComment (skipComment continue))
            (skipChar (skipComment continue))
        )
        state
```

Ugyanez a kód jóval olvashatóbb két infix makró bevezetésével, lássuk előbb az elsőt:

```
(&>) infixr 4
(&>) f g :== \continue = f (g continue)

skipComment‘ continue state =
    string "*/"
        continue
        (string "/*"
            ((skipComment  &> skipComment) continue)
            ((skipChar  &> skipComment) continue)
        )
        state
```

Kis átalakítással:

```
skipComment‘‘ continue =
    string "*/"
        continue
        ((string "/*"  &> skipComment  &> skipComment)
            continue
            ((skipChar  &> skipComment) continue)
        )
```

A második makró bevezetése után:

```
(|>) infixr 2
(|>) f g :== \continue = f continue (g continue)

skipComment‘‘ continue =
    string "*/"
        continue
        (   (    string "/*"  &> skipComment  &> skipComment
            |>   skipChar  &> skipComment
            )
            continue
        )
```

Végül:

```
skipComment‘‘‘
    =   string "*/"
    |>  string "/*"  &> skipComment  &> skipComment
    |>  skipChar  &> skipComment
```

A kapott kódot hasonlítsuk össze a következő BNF nyelvtanleírással:

```
commentTail
    =   "*/"
    |   "/*" commentTail commentTail
    |   anyChar commentTail
```

Itt meg is érkeztünk a kombinátoros elemzőkhöz.

2.3. Kombinátoros elemzők

A szakirodalom[4] a következőképpen mutatja be a kombinátoros elemzőket:

A (nemdeterminisztikus) funkcionális szintaktikai elemző olyan függvény, amely szimbólumok sorozatából párok listáját állítja elő. Minden pár egy lehetséges elemzésnek felel meg. A párok első eleme a feldolgozott szimbólumok valamilyen reprezentációja, a második elem pedig a szimbólumsorozat nem használt része, amit későbbi elemzőknek szánunk.

```
:: Parser symbol out :== [symbol] -> [(out, [symbol])]
```

Már meglevő elemzőkből újabb elemzők állíthatók elő úgynevezett kombinátorokkal. Lássunk a két alapvető kombinátort:

Az első, amit a p és a q elemzők egymás mellé kapcsolásának fogunk nevezni, az az elemző, amely a p és q által kapott listák konkatenáltját adja vissza. A művelet típusa és hagyományos jelölése a következő:

```
(<|>) infixr 2 :: (Parser symbol out) (Parser symbol out) -> Parser symbol out
```

Nevezzük a p és a q elemzők egymás után kapcsolásának azt az elemzőt, amely a p szerinti elemzés után megmaradt szimbólumsorozatokat q szerint elemzi. Lényeges kérdés, hogy a két elemző által kapott reprezentációpárokat milyen módon egyesítjük. A klasszikus megoldás leovasható a kombinátor típusdefiníciójáról:

```
(<&>) infixr 4 :: (Parser symbol a) (Parser symbol b) -> Parser symbol (a,b)
```

Egyelőre csak a típusdefiníciókra támaszkodunk; a kombinátorok megvalósításával csak a végső forma elérése után fogunk foglalkozni.

Tehát a kívánt elemzőt elemi elemzőkből építhetjük fel, és a kapott szerkezet hasonlítani fog egy BNF nyelvtanleíráshoz. Az egyik különbség az, hogy nem egy természetesen adódó faszerkezetet várunk az elemzés után, hiszen nincs szükségünk például a kommentekre, a zárójelekre (ezek csak a szerkezet kialakításához kellettek), és sok műveletet már elemzés közben is elvégezhetünk (gondoljunk itt a számjegysorozatok számmá alakítására).

A klasszikus megoldás további kombinátorok bevezetése. Az első kettő az „és" kombinátor variációi, amik elhagyják az első illetve a második elemző által adott reprezentációt. A harmadik kombinátor egy adott függvénnyel transzformálja a reprezentációt:

```
(&>) infixr 4 :: (Parser symbol a) (Parser symbol b) -> Parser symbol b
(<&) infixr 4 :: (Parser symbol a) (Parser symbol b) -> Parser symbol a
(<@) infixl 3 :: (Parser symbol a) (a->b) -> Parser symbol b
```

Másik megközelítés az előző `<@` kombinátor használata a monadikus stílusú „és" kombinátorral:

```
(<&>) infixr 4 :: (Parser symbol a) (a -> Parser symbol b) -> Parser symbol b
```

A harmadik Röjemo[7] alapján:

```
(<&>) infixr 4 :: (Parser symbol (a->b)) (Parser symbol a) -> Parser symbol b
```

Én egy negyedik megoldást választottam. Kiszélesítettem az elemzők körét: egy elemző tetszőleges számú értéket kaphat és adhat vissza. Ezt azonban majd csak a determinisztikus esetben lehet úgy megvalósítani, hogy a sokféle elemzőre alkalmazható legyen ugyanaz a néhány kombinátor.

Az elemzők és kombinátorok végső formájának kialakítása után egy rövid példán keresztül hasonlítom össze a négy különböző megoldást.

2.4. Determinisztikus elemzők

A Clean nyelvtanának elemzéséhez elegendőek a hatékonyabb determinisztikus elemzők. Ezek csak egyféle elemzést adnak vissza, vagy egyet se, amit a következő típussal tudunk kifejezni:

```
:: Deterministic_parser symbol out :== [symbol] -> (Maybe out, [symbol])

:: Maybe a = Just a
           | Nothing
```

Ugyanez a típus folytatás stílusban [1] így néz ki:

```
:: Deterministic_parser_with_continuation symbol out end :==
    ([symbol] out -> end)                    // siker esetén ezt hívjuk
    ([symbol] -> end)                        // kudarc esetén ezt hívjuk
    [symbol]
        -> end
```

Itt még nincs jelentősége, de a későbbiekben fontossá válik, hogy a siker esetén hívott függvény két argumentuma közül az `out` a második. Az `end` típusváltozót kihagyhatnánk a típus argumentumaiból, azonban a Clean még nem támogatja megfelelően a lokális univerzális kvantálást. Sajnos ezért mindig fel kell tüntetnünk majd ezt a típusváltozót is.

A továbbiakban azt a szófordulatot használjuk, hogy az elemző *elfogadta* illetve *nem fogadta el* az inputot, attól függően, hogy melyik folyatás függvényt hívta.

2.5. Elemzők egyszeres hivatkozású állapottal

Egyszeres hivatkozású állapot bevezetésére a hatékonyság növelése érdekében van szükség. A szimbólumlistát egy összetettebb adatszerkezettel helyettesítjük, ami tatalmazza az inputfájlt, az aktuális sort, a soron belüli pozíciót, és egyéb információkat, amikre például a hibakezelés során lesz szükség.

Nemdeterminisztikus esetben az egyszeres hivatkozású állapot bevezetése nem triviális [5], azonban determinisztikus esetben simán megy a dolog. Nevezzük `State`-nek az állapot típusát.

```
:: Parser_with_a_uniqe_state out *end
    :== (*State -> *(out -> end)) (*State -> end) *State -> end
```

2.6. Egyszerű és próbálkozó elemzők

Sokszor biztosak lehetünk benne, hogy egy adott elemző mindig elfogadja az inputot (például az `skipChar` egy korábbi példában), így nem szükséges átadni mindkét folytatás függvényt. Nevezzük az ilyen elemzőket *egyszerű* elemzőknek, míg a korábbiakat *próbálkozó* elemzőknek.

Azt hiszem, ezzel a megkülönböztetéssel újat alkottam. Két elemzőt hatékonyabban kapcsolhatunk egymás után, ha a második egyszerű elemző, hiszen biztosak lehetünk benne, hogy a második elemző nem akar majd visszalépni. Az egyszerű elemzők típusa is egyszerűbb:

```
:: Simple_parser out *end :== (*State -> *(out -> end)) *State -> end
```

2.7. Az elemzők típusának végső formája

Az általam használt elemzők típusát a következőképpen tudom összefoglalni:

Egyrészt az elemzőknek egy vagy két függvényargumentuma van attól függően, hogy az egyszerűek vagy a próbálkozók csoportjába tartoznak-e. A függvényargumentumok után következik az állapot, majd esetlegesen további argumentumok.

Másrészt az elemzők különböző számú adatot várhatnak és adhatnak vissza. A bemenő adatok a programállapot után sorakoznak, a kimenő adatokat pedig az első függvényargumentumának adja tovább az elemző.

Ezt a két szempontot tükröznie kell a típusdefiníciónak, amit a következőképpen oldottam meg: Minden elemzőtípus $Parser\_N\_M\ a_1\ \ldots\ a_N\ b_1\ \ldots\ b_M$ `e` alakú, ahol a $Parser$ helyén `Try` vagy `Simple` áll az első szempont alapján, N és M egy–egy, a kimenetek és bementek számát jelző számjegy, $a_1\ \ldots\ a_N$ a bemenetek típusai, $b_1\ \ldots\ b_N$ a kimenetek típusai, az `e` típusváltozót pedig mindig ki kell írnunk.

Nézzünk néhány példát és a típusdefiníciókat:

```
///////////////////////////típuspéldák
skipSpace                  :: Simple_0_0 e
identToken                 :: Try_0_1 String e
identToken‘                :: Simple_0_1 String e  // azonosítónak kell következnie
identToken <&> identToken  :: Try_0_2 String String e
classDefinition            :: Try_1_1 Position String e  // vár egy pozíciót
apply2                     :: (x y->a) -> Simple_2_1 x y a e

///////////////////////////típusdefiníciók
:: *State // a programállapot

:: Simple *a *b   :== (*State->a)             -> *State->b
:: Try *a *b      :== (*State->a) (*State->b) -> *State->b

:: Simple_0_0       *e  :== Simple e            e
:: Simple_0_1 a     *e  :== Simple (a->e)       e
:: Simple_0_2 a b   *e  :== Simple (b->*(a->e))  e
:: Simple_1_1 x a   *e  :== Simple (a->e)       (x->e)
:: Simple_2_1 x y a *e  :== Simple (a->e)       (y->*(x->e))
...

:: Try_0_0       *e  :== Try e            e
:: Try_0_1 a     *e  :== Try (a->e)       e
:: Try_1_1 x a   *e  :== Try (a->e)       (x->e)
...
```

Az elemzőket csoportosíthatjuk egy harmadik szempont szerint is, amit azonban nem tükröz a típusuk. Az elemző megváltoztathatja, vagy változatlanul hagyhatja a programállapotot. Mindig teljesulnie kell annak, hogy ha egy elemző nem fogadja el az inputot, akkor változatlanul adja vissza a programállapotot.

Vannak olyan elemzők, amik elfogadják az inputot, de változatlanul hagyják a programállapotot. Ezt a tulajdonságot a `check` előtaggal jelezem majd az elemzők nevében. Végül lesz sok olyan egyszerű elemző, amely megváltoztatja a programállapotot, de nem váltotztatja benne a pozíciót. Ezek mind az üres szó elemzőiként foghatók fel. „Elemző”-ként hivatkozunk rájuk, mivel nem valódi elemzők, csak a programállapot változtatására szolgálnak.

2.8. A végső kombinátorok

Érdemes megfogalmazni, hogy mit is tekintünk kombinátornak. Ezek olyan függvények, amelyeknek legalább az egyik argumentumuk egy elemző, és egy elemzőt ad vissza. Találkozunk majd olyan függvényekkel, amik elemzőt adnak vissza, de az argumentumuk közt nem szerepel elemző. Ezek általánosított elemzőként foghatóak fel, és szintén elemzőként hivatkozunk rájuk. Erre lesz példa a későbbi `keyword`.

Az eddigi felépítés lehetővé teszi, hogy nem kellenek minden elemzőtípushoz újabb és újabb kombinátorok. Ráadásul ez a néhány kombinátor is meglepően egyszerű. A legtöbb célra elegendő a következő négy kombinátor:

```
(|>)  infixr 2  //:: (Try a b) (Simple a b) -> Simple a b
(|>)  f g       :== \succes= f success (g success)

(<|>) infixl 2  //:: (Try a b) (Try a b) -> Try a b
(<|>) f g       :== \success fail= f success (g success fail)

(&>)  infixr 4  //:: Try_N_M Simple_K_L -> Try_N+K_M+L
                //:: Simple_N_M Simple_K_L -> Simple_N+K_M+L
(&>)  f g       :== \success= f (g success)

(<&>) infixr 4  //:: Try_M_0 Try_K_N -> Try_M+K_N
(<&>) f g       :== \success fail= f (g success fail) fail
```

A megjegyzésekben szereplő típusdefiníciókat a korábbi sablon szerint értelmezhetjük.

Hogy megértsük, miért használható például az `&>` kombinátor a `Try_0_1 Simple_0_1` típusú elemzők egymás után kapcsolására, elkészítjük erre a speciális esetre a megfelelő kombinátort, és triviális átalakítások sorozatával eljutunk az `&>` fenti definíciójához:

```
and_for_Try_0_1_Simple_0_1 :: (Try_0_1 a e) (Simple_0_1 b e) -> Try_0_2 a b e
and_for_Try_0_1_Simple_0_1 f g = parser
where
    parser success fail state = f success‘ fail state
    where
        success‘ state a = g success‘‘ state
        where
            success‘‘ state b = success state b a

and_for_Try_0_1_Simple_0_1‘ f g = parser
where
    parser success fail state = f success‘ fail state
    where
        success‘ state a = g success‘‘ state a
        where
            success‘‘ state b = success state b

and_for_Try_0_1_Simple_0_1‘‘ f g = parser
where
    parser success = f success‘
    where
        success‘ = g success‘‘
        where
            success‘‘ = success

and_for_Try_0_1_Simple_0_1‘‘‘ f g = parser
where
    parser success = f (g success)

and_for_Try_0_1_Simple_0_1‘‘‘‘ f g = \success = f (g success)
```

Visszatérve a négy kombinátorhoz, egyedül a `<&>` nem eléggé általános, nem használható például `Try_0_1` és `Try_N_M` típusú elemzők összekapcsolására. Ez annak köszönhető, hogy ilyen elemzők egymás után írásakor, ha az első elfogadta az inputot, de a második nem, akkor az első elemző által visszaadott reprezentációt el kell dobnunk:

```
(<&+>) infixr 4  //:: Try_M_1 Try_K_L -> Try_M+K_L+1
(<&+>) f g       :== \success fail= f (g success (drop1 fail)) fail

drop1              //:: Simple_M_N -> Simple_M_N-1
drop1 continue state _ :== continue state
```

Így egy egész sorozatot kellene írnunk a `<&>` kombinátorból, de nekünk elég lesz az előbbi kettő.

2.9. Visszatérés korábbi pozícióhoz

Jogosan merül fel a kérdés, hogy hogyan biztosítjuk két próbalkozó elemző egymás után kapcsolásakor, hogy a kapott elemző, ha nem fogadta el az inputot, változatlan programállapotot adjon vissza. Ezt beépíthetnénk az „és” kombinátorba, de ez lassuláshoz vezne többszörös „és” kombináció esetén, vagy abban az esetben, ha az első elemző `check` tulajdonságú.

A következő megoldást választjuk:

Definiálunk egy `toSafe :: (Try a b) -> Try a b` típusú kombinátort. A kombinátor végrehajtása során eltároljuk az adott sor és oszlop sorszámát és a fájlpozíciót, meghívjuk a kapott elemzőt, majd ha az sikertelen volt, visszaugrunk a megfelelő oszlopra. Ha a sor is megváltozott, újraolvassuk a fájlból a korábbi sort.

Ezután minden helyen, ahol összetételből kapunk próbálkozó elemzőt, azaz a `<&>` és a `<&+>` kombinátorok használata során, ha az első elemző nem `check` tulajdonságú, alkalmaznunk kell a `toSafe` kombinátort is:

```
toSafe (keyword "(" <&> identToken <&+> keyword ")")
```

Definiálunk egy hatékonyabb `toSafe_in_line` kombinátort is arra az esetre, ha az argumentum elemző biztosan ugyanabban a sorban marad. Ebben az esetben csak az oszlopszámot kell eltárolnunk.

2.10. Összehasonlítás

A korábbiakban négyféle módot láttunk elemzők egymás után kapcsolására. Most vessünk egy pillantást a gyakorlati alkalmazásukra. A Clean-stílusú feltételes kifejezés elemzőjét fogjuk látni négyszer. Mind a négy elemző determinisztikus, a részletekre nem térünk ki, csak a kapott forma érdekel.

A klasszikus:

```
if_expression =
    keyword "if"     &>
    expression       <&>
    expression       <&>
    expression       <@
    \(condition, (then_experssion, else_expression))
        = If_expression condition then_expression else_expression
```

Röjemo alpján [7]:

```
if_expression =
    keyword "if"     &>
    expression       &.>
    expression       &.>
    expression       <@@@
    If_expression
```

A `<@@@` operátor, ami gyengébb kötésű a többinél (a fenti kódon ez nem látszik), az egész addig felépített elemző által visszaadott értékre alkalmaz egy függvényt úgy, hogy a függvény már az elemzés kezdetén bekerül a memóriába. Ez olyan esetekben, amikor csak utólag tudjuk eldönteni, milyen függvényt akarunk alkalmazni, kifejezetten hátrányos. Ezért ezt a stílust elvetjük.

A monadikus:

```
if_expression =
    keyword "if"     &>
    expression       &.> \condition=
    expression       &.> \then_expression=
    expression       &.> \else_expression=
    return
        (If_expression condition then_expression else_expression)
```

A legújabb:

```
if_expression =
    keyword "if"     &>
    expression       &>
    expression       &>
    expression       &>
    apply3           If_expression
```

Az utóbbi két stílus jöhet leginkább szóba. Tisztább, és a futási ideje is jobb a legújabbnak.

3. A szintaktikus elemző

Ez a fejezet az elmélet szempontából már nem nyújt újat, hanem a következő célokra szolgál: egyrészt bemutatja, milyen további nehézségekkel kellett még szembenéznem az elemző megírása közben, másrészt dokumentáció azoknak, akik az elemző kódját olvassák.

3.1. Tagolás

A korábbi Clean fordítóban is jól elkülöníthetőek a hagyományos elemző egységek: a lexikális elemző (ez a `scanner` modulban található), a szintaktikus elemző (a `parse` modulban) és a szemantikus elemző (a `postparse` modulban).

A szemantikus elemzőt és a fordító további részeit is eredeti formájában, változatlanul hagytam. Ezt azért tehettem meg, mert a lexikális és szintaktikus elemző csak a `wantModule` függvényen keresztül érintkezik a külvilággal, így csak ezt a függvényt kellett kicserélnem.

Korábban a lexikális elemző tokeneket szolgáltatott, és visszalépés esetén a megjegyzett tokeneken ment végig újra a szintaktikus elemző. Az új elemzőben minden egyes tokentípusokhoz egy–egy kombinátoros elemző tartozik, visszalépés esetén ismét az eredeti soron megyünk végig. Ez nagyobb szabadságot ad, hiszen a visszalépés után teljesen új értelmezésre is lehetőség nyílik, másrészt nem lassítja lényegesen a futási időt, mivel a visszalépések ritkák. Valószínűleg még gyorsulás is fellép, mert a kulcsszavak tokeneit nem kell létrehozni, majd eldobni.

Miért ritkák a visszalépések? Ezt a Clean nyelvtanának köszönhetjük. Csak két három helyen kell a korábban definiált `toSafe` kombinátort használni, ami a visszalépésért felelős ráadásul a visszalépés esetén is csak kevés számú karaktert kell újra elemzeni. Még nem találkoztam olyan forráskóddal, ahol sorokat kellett volna visszaugrani, vagy kommenteket kellett volna újraelemezni. Ez csak a következőhöz hasonló extrém kódok esetén történne meg:

```
(<.>            /* új sor és komment egy zárójelezett függvénynévben! */
    ) infix 4
```

Azzal, hogy minden tokentípust egy kombinátoros elemző képvisel, megszűnik a válaszfal a lexikális elemző és a szintaktikus elemző között. Az egész számok elemzője például éppúgy kombinátoros elemző, mint a kifejezések elemzője. Mégis meghagytam a két részre osztást, mivel a Clean nyelvtana erősen sugallja ezt. A két rész két modulban foglal helyet, ezekre lexikális és szintaktikus modulként fogok hivatkozni.

3.2. A szintaktikus modul

A szintaktikus modul tartalmazza szinte az egész nyelvtanleírást. Bemutatása mégis egyszerű, mivel a lexikális modul tartalmazza a kényes részleteket.

A szintaktikus modul a korábbi `parse` nevet örökölte, egyetlen exportált függvénye a `wantModule`. A módosított `wantModule` függvény nem tesz mást, mint meghívja a lexikális modulban található `wantModule‘` függvényt egy további argumentummal. Ez az argumentum a „legfelső” kombinátoros elemző. A `wantModule` közvetlen és közvetett módon a következő függvényeket hívja meg:

- A `wantModule‘` függvényt a lexikális modulban. Ez a függvény hozza létre a programállapotot és indítja be az elezést. Később még beszélünk róla.
- Néhány makrót, amik a munkát segítik.
- A Clean egy–egy nyelvtani szerkezetének kombinátoros elemzőit, mint a `header`, `pattern`, `listExpression`, `typeDefinition`, stb. Szinte a modul egészét ezek definíciója alkotja.
- Konverziós rutinokat. A korábbi elemző olyan feladatokat is ellátott, mint például a típusdefiníciókban a `.x` → `x:x` konverzió, vagy a `!` annotációk kigyűjtése. Ezeket a feladatokat nekem is át kellett vennem.

- A korábbi tokenektípusoknak megfelelő elemzőket. Ezek a lexikális modulban foglalnak helyet. Sikerült a számukat minimálisra csökkenteni azzal, hogy a kulcsszavakat egyetlen elemő elemzi.
- A kombinátorokat. Ezek definiciója is a lexikális modulban található.
- Néhány „elemzőt”, amik semmit nem csinálnak, csak a programállapotot módosítják. Szintén a lexikális modulban vannak. Ilyen például a `typeContext` vagy a `setLayoutTrue` függvény.

3.3. A lexikális modul

A lexikális modul tartalmazza a lexikális részek elemzőit és a kombinátorokat. Ide helyeztem még az elemzés környezetét biztosító `wantModule‘` függvényt is, hogy egyetlen modulban legyen az összes függvény, amire hatással lehet a programállapot típusának megváltoztatása. Először ezt a függvényt mutatom be, majd a programállapotot. Ezután témakörönként következnek a különböző elemzők.

3.3.1. Az elmzés kezdete és vége

A kombinátoros elemzéshez a `wantModule‘` függvény biztosítja a környezetet. A függvény feladata egy adott nevű fájl megnyitása és elemzése egy adott kombinátoros elemzővel. A függvény típusa:

```
wantModule‘ ::
    !Bool                       // Igaz: icl modul;  Hamis: dcl modul
    !Ident                      // a modulnév
    !Position                   // az a pozíció, ahonnan ezt a modult importálták
    !Bool                       // Igaz: generic-eket is elemzünk
    !*HashTable                 // hashtábla az azonosítók hatékony tárolására
    !*File                      // hiba-fájl
    !SearchPaths                //
    (ModTimeFunction *Files)    // segédfüggvény
    !*Files                     // a fájlrendszer
    (Simple_0_1 ParsedModule (ParsedModule,*State))
                                // ezzel az elemzővel elemzünk majd

    -> ( !Bool                  // Igaz: hibátlan elemzés
       , !ParsedModule          // az elemzett fájl reprezentációja
       , !*HashTable            //
       , !*File                 // a hiba-fájl a hibákkal és figyelmeztetésekkel
       , !*Files                // a módosult fájlrendszer
       )
```

A függvény első lépésben a kapott név, elérési utak és a fájlrendszer segítségével megnyitja az elemzendő fájlt. Ezután létrehozza a kezdő programállapotot, amibe többek között ez a fájl és a kezdőpozíció kerül. Most jön a kapott elemző hívása a kezdő programállapottal és egy befejező folytatás függvénnyel. A befejező függvény a kapott programállapotot és a reprezentációt egyszerűen egy párba foglalja (még egyszer megjegyezzük, hogy a programállapot nem tartalmazza az elemzett részek reprezentációit). Ezt a párt kapja vissza a `wantModule‘`. Utolsó lépésként már csak ki kell szedni az inputfájlt a programállapotból és be kell zárni, majd visszatérni a reprezentációval.

3.3.2. A programállapot

A programállapot típusdefiníciója:

```
:: *ST =
    { input             :: !*File          // az inputfájl
    , imput_name        :: !String         // an inputfájl neve
    , line              :: !String         // az aktuális sor
```

```
    , row              :: !Int           // a sor száma
    , column           :: !Int           // az oszlop száma
    , offside_column   :: !Int           // lásd szerkezeti szabályok

    , error_file       :: !*File         // hibakezeléshez szükséges információk
    , okey             :: !Bool
    , skipping         :: !Bool

    , tabsize          :: !Int           // a tabulátor mérete
    , underscoreAllowed :: !Bool         // engedéyezett-e "_"-sal kezdődő azonosító
    , context          :: !Int           // lásd kulcsszavak
    , hashTable        :: !*HashTable    // lásd azonosítók
    }
```

3.3.3. Új sor beolvasása

Újabb sor beolvasását a `newLine` függvény végzi. Ez is egy „elemző”, amely nem csinál semmit, csak a programállapotot változtatja meg, típusa ezért `Simple_0_0`.

Miután az inputfájlból beolvasunk egy új sort, kisebb átalakításokat végzünk rajta (előfeldolgozás), majd eltároljuk a programállapotban. Az átalakítások a következők:

- A tabokat kicseréljük a megfelelő számú szóközre. A tabulátor méretét a programállapot tartalmazza. A csere után biztosak lehetünk, hogy az oszlop sorszáma megegyezik a sorindexszel, ezért azt a későbbiekben nem kell külön kiszámolnunk.
- A sorvégeket egységesen Unix stílusúra cseréljük. Biztosítjuk, hogy a fájl üres sorral végződjön.
- Mindezek előtt vagy után jöhetnek a feltételes fordításhoz szükséges transzformációk (a három pont tetszőleges karaktersorozatot jelöl):

```
"/*2.0..." => ""
"0.2*/..." => ""
"//1.3..." => "/*"
"//3.1..." => "*/"
```

3.3.4. Kommentek, szóközök

A kommentek és szóközök elemzését egy egyszerű elemző végzi, nem vár és nem ad vissza értéket:

```
skipSpace :: Simple_0_0 e
```

Ez már nem elemi elemző. Felhasználja többek közt a `string` nevű általánosított elemzőt, ami egy sztringből előállít egy olyan próbálkozó elemzőt, ami a kapott sztringgel kezdődő inputot fogadja el:

```
string :: !String -> Try_0_0 e
string s =  checkString s  &>  skipChar (size s)

checkString :: !String -> Try_0_0 e
checkString string success fail state=:{line,column} =
    if (compare column 0) success fail state
where
    compare i j =
        j == size string  ||
        line.[i] == string.[j]  &&  compare (inc i) (inc j)
```

A `skipSpace`-t minden elfogadott kulcsszó, azonosító vagy elemi érték után hívjuk automatikusan.

3.3.5. Kulcsszavak

A Kulcsszavak elemzését rá lehet bízni egy, az előző `string`-hez hasonló általánosított elemzőre, legyen ennek neve `keyword`. Elvárjuk, hogy a `keyword "in"` ne fogadja el a `"inside"` inputot. Ez úgy oldható meg, hogy három osztályba soroljuk a karaktereket: alfanumerikus, speciális és egyéb. (A karakterek osztályzását táblázat segítségével gyorsítuk.) Ha egyezés van, és a kulcsszó utolsó karaktere az egyéb osztályba tartozik, vagy az utána következő karakter más osztályba tartozik, akkor elfogadjuk az inputot.

A Clean nyelvtana azt is elvárja, hogy a `keyword "!"` fogadja el a `"!*"` inputot, noha a `"!"` és `"*"` is speciális karakterek. Ezt úgy oldottam meg, hogy más karaktereket tekintek speciálisnak normál környezetben és típuskörnyezetben. Ennek megvalósításához kell a programállapot `context` mezője. Ez a mező két értéket vehet fel, a két értéket egy–egy „elemző” állítja be, a `normalContext` és `typeContext`.

A hatékonyság növelése érdekében a `keyword`-ot függvényosztályként definiáltam, és nem csak sztringekre, hanem karakterekre is példányosítottam. A karakteres példány nem végzi el a fenti ellenőrzéseket, így például `keyword 'a'` minden `a`-val kezdődő inputot elfogad (`keyword '('` a valós felhasználása ennek).

3.3.6. Azonosítók

Kétféle azonosító lehet a Cleanben: kisbetűvel kezdődőek és nagybetűvel vagy speciális karakterrel kezdődőek. Mindkét esetben addig tart az azonosító, míg különböző karakterosztályba tartozó karakterrel nem találkozunk.

Nem fogadjuk el azonosítónak a kulcsszavakat. Ezek felismerését táblázat segítségével gyorsítjuk. Egy–egy táblázatra van szükség a típuskörnyezethez és a normálkörnyezethez.

Az azonosítókat hashtábla rakjuk, így tárhelyet takarítunk meg, és az azonos nevű azonosítók adatait egyszerre tudja majd változtani a fordító. A hashtáblába rakáshoz egyéb információra is szükség van, ezért ezt a `into_hashTable` „elemző” végzi el a `keyword` helyett.

3.3.7. Hibakezelés

A hibakezelés maradt a régi pánikszerű hibakezelés, amihez három mezőre van szükség a programállapotban:

- `error_file` a fájl, ahová a hibaüzenetek és figyelmeztetések kerülnek.
- `okey` egy logikai érték, ami akkor igaz, ha az elemzés során még nem fordult elő hiba.
- `skipping` szintén egy logikai érték. Ha ez igaz, akkor a hibaüzeneteket nem írjuk ki a fájlba. Kezdetben hamis, és minden hiba előforduláskor igazra állítjuk. Ha sosem állítanánk vissza, csak egy hibaüzenetet kapnánk.

  Pillanatnyilag a felsorolások és többszörös értékek elemzését általánosító kombinátorok állítják csak hamisra vissza ezt a változót, de ez elégnek is bizonyult. Felsorolások elemzése közben, mielőtt a változót visszaállítanánk hamisra, ugrunk az inputban a követkerő elválasztó elemig (ez általában a vessző), vagy a felsorolás végét jelző elemig.

Három „elemzőt” exportál a lexikális modul a hibakezelésre: az egyiket hibák, a másikat „expected instead of” típusú hibák, a harmadikat figyelmeztetések kiírására.

3.3.8. Szerkezeti szabályok

A modern funkcionális nyelvek lehetővé teszik az úgynevezett szerkezeti szabályok (layout rules) használatát: kapcsos zárójelek és pontosvesszők használata helyett különböző méretű behúzásokkal szabályozzuk a csoportba foglalást. Ezzel természetesen áttekinthetőbb kódokat kapunk, és egyúttal egy mindenki által követendő egységes behúzási stílust.

Az előző fordítóban a lexikális elemző feladata volt, hogy a megfelő helyre beszúrja ezeknek a láthatatlan kapcsos zárójeleknek és a pontosvesszőknek megfelelő tokeneket a tokensorozatba, így a szintaktikus elemzőnek már nem kellett vizsgálni a behúzások mértékét. A következő szabályok szerint történt a beszúrás:

- Nyitó kapcsos zárójelet bizonyos kulcsszavak után szúrunk be, ezek közül a legfontosabbak: `where`, `in`, `of`, `#`.

  A beszúrással egyidejűleg a programállapotban eltároljuk annak az oszlopnak a sorszámát, ahol következő token kezdődik. Erre a pozícióra *határpozíció*ként hivatkozuk majd.

  Mivel a kapcsos zárójelek egymásba ágyazhatóak (egyre bentebbi behúzásokkal), ezért a programállapot határpozíciók listáját tartalmazza. A lista természetéből adódóan rendezett, a fej tartalmazza a legutóbbi, legmagasabb pozíciót. A kezdő programállapotban ez a lista már tartalmazza a nulla pozíciót, aminek eredményeként a globális definiciók közé pontosvesszőket szúr majd a lexikális elemző.

- Pontosvesszőt szúrunk minden token elé, ami határpozíción kezdődik és nem azonos néhány kulcsszóval, például a `where`-rel.

- Csukó kapcsos zárójeleket szúruk minden token elé, amik a határpozíciótól alacsonyabb pozíción kezdődik. A beszúrt zárójelek száma megegyezik azon határpozíciók számával, amik a határpozíciólistában szerepelnek és magasabbak az aktuális pozíciótól. Ezeket a határpozíciókat elhagyjuk a listából, így az aktuális határpozíció is módosul.

Az új programállapotban csak az aktuális határpozíció szerepel, ugyanis a `where`, `let`, stb. kulcsszavak után a régi határpozíciót tárolhatjuk a veremben a blokk elemzésének végéig.

3.4. Értékelés

3.4.1. Állapot

Az új elemző még nem megbízható hibás bemenet esetén. Szükséges lenne egy gondos egybevetés a korábbi elemző kódjával, és sok–sok tesztelés a megbízhatóbbá tételhez. Az egybevetéskor a Clean nyelvspecifikációra nem támaszkodhatok, mert sok helyen pontatlan, például a szerkezeti szabályok tárgyalásánál.

A következő hibátlan bemenetekre teszteltem az új elemzőt:

- A fordítóval együtt terjesztett `Examples` könyvtár `Small Examples` alkönyvtárának implementációs moduljaira.
- Az előbb említett `Examples` konyvtár `ObjectIO Examples 1.2.2` alkönyvtárában található `hanoi.icl` modulra.
- Az új elemzőre.

A tesztelés csupán abból állt, hogy lefordítottam az említett modulokat, és megnéztem, hogy a linkelés után működnek-e a programok. Szemmel láthatóan működtek.

A mohó listákat, generic-eket és dinamic-okat nem támogatja még az új elemző, de ezek felismerése valószínűleg akadály nélkül beépíthető.

3.4.2. Időigény

Az új elemző méréseim szerint hatszor lassabb az eredetinél, ez a fordítás közben körülbelül kétszeres lassulást eredményez.

A lassulást nem sikerült elvi okokra visszavezetnem, egy része a fordító nem kielégítő optimalizálásával magyarázható. Tekintsük a következő definíciókat:

```
//     1  2  3  4  5  6  7  8  9  10
f1 x = f (f (f (f (f (f (f (f (f (f x)))))))))
```

```
//   1   2   3   4   5   6   7   8   9   10
f2 = f o f o f o f o f o f o f o f o f o f

(o) infixr  9    // függvény kompozíció
(o) f g :== \ x = f (g x)

f [] = [0]
f [_] = []
```

A második függvény ekvivalens az elsővel, mégis, méréseim szerint másfélszer lassabb. (A függvények futási ideje 1.1 GHz-es Pentium III processzorral Windows alatt 1.30 illetve 1.90 másodperc 10000000 futtatás esetén.)

Hivatkozások

[1] Appel, A.: *Compiling with Continuations.* Cambridge University press. 1992.

[2] Fokker, J.: *Functional Parsers.* In: Advanced functonal programming, Baastad Summerschool Tutorial Text. LNCS 925, 1995, 1–23.

[3] Plasmeijer, M., van Eekelen M.: *The concurrent Clean Language Report. Version 2.0.* Nijmegen University, The Netherlands. 2001. http://www.cs.kun.nl/ clean.

[4] Swierstra, D., Duponcheel, L.: *Deterministic, Error-Correcting Combinator Parsers.* In: Advanced Functional Programming. LNCS 1129, 1996, 185-207.

[5] Koopman, P., Plasmeijer, M.: *Layered Combinator Parsers with a Unique State.* In: Proceedings of the 13th International workshop of the Implementation of Functional Languages, IFL 2001, Alvsjö, Sweden, September 24–26, 2001.

[6] Koopman, P., Plasmeijer, M.: *Efficient Combinator Parsers.* In: Proceedings of Implementation of Functional Languages, IFL 1998, London, UK, Springer Verlag, LNCS 1595, pp. 120–136.

[7] Röjemo, N.: *Garbage collection, and memory efficiency, in lazy functional languages.* PhD thesis, Chalmers, rojemo@cs.chalmers.se, 1995.

[8] Walder, P.: *Monads for functional programming.* In: Advanced functonal programming, Baastad Summerschool Tutorial Text. LNCS 925, 1995, 24–52.

[9] Nyékyné Gaizler Judit szerk., Nyékyné G.J.-Horváth Z. és mások: *Programozási nyelvek összehasonlító elemzése.* Kiskapu Kiadó, Budapest, 2002. Fejezet (Horváth Z.): Funkcionális programozási nyelvek elemei, 56 oldal, megjelenés: 2002. december