1. Bevezetés

[TODO]

2. Metaprogramozásról általában

[TODO]

3. Metaprogramozás a mai programozási nyelvekben

A következő fejezetekben körül fogom járni, hogy jelen pillanatban milyen eszközök állnak a programozók rendelkezésére metaprogramozás szempontjából.

3.1. A C/C++ előfordítója

Nem hagyományos értelemben a C előfordítóját is nevezhetjük a metaprogramozás egyik eszközének, azzal a különbséggel, hogy közvetlenül a forráskódon végez transzformációkat.

Működésének az alapelve nagyon egyszerűnek tekinthető, hiszen egyszerűen szövegbeszúrásokat és szöveghelyettesítéseket végez a forráskódon. Egyszerűségében rejlik ereje is, ugyanis rendkívüli szabadságod ad a programozó kezébe maga az előfordító, de sajnos ez a gyakorlatban több problémát is eredményezhet.

Magát az előfordítót egy különálló nyelvnek is tekinthetjük, ami a C-től független, mivel még a C nyelv feldolgozása előtt feldolgozásra kerül, minden egyes fordításkor. Több feladat is van az előfordítónak, úgymint a fizikailag több sorban lévő, de logikailag egy sornak számító kódok összefűzése, a preprocesszor direktíváinak tokenekre bontása, megjegyzések törlése a kódból, és a felhasználó által definiált utasítások végrehajtása (szimbólum behelyettesítés, makrók, esetleg feltételes fordítás).

3.1.1. Az include direktíva

Az **`include`** direktíva az előfordító leggyakrabban használt utasítása. A fordító megkeresi a programozó által megadott fájlt és annak a tartalmát egyszerűen bemásolja a fordítás alatt lévő fájl tartalmába:

```
// megkeresi az iostream fájlt és annak a tartalmát bemásolja
#include <iostream>
int main() {
      std::cout << "Hello World!" << std::endl;
      return 0;
}
```

Sajnos egyszerűségében rejlik legnagyobb hátránya is, hiszen a külső függőségek kezelését nem elég ilyen alacsony szinten kezelni. Tipikus, hogy egyes külső fájlokat, több helyen is használni szeretnénk. Ahhoz, hogy ezt megtehessük, minden egyes fájlnál hivatkozni kell rájuk. De az előfordító nem tartja számon azt, hogy mely fájlok kerültek már felhasználásra, ezért könnyen előfordulhat az, hogy egy fájlt kétszer vagy annál többször másolja be. Ez a kódnak a duplikációjához vezet, ami pedig később fordítási hibához. Ezt elkerülendően feltételes fordítással oldják meg az ilyen problémákat.

Az alapötlet az, hogy szimbólumok segítségével tartjuk nyilván, hogy az adott adott header fájlt betöltötte-e már a fordító vagy sem. Ha igen, akkor a feltételes fordítást használva, egy üres fájlt adunk vissza neki, ellenkező esetben az eredeti tartalmat:

```
#include <iostream>
// megnézi a fordító, hogy definiálták-e már
// az adott szimbólumat
#ifndef PERSON_H
// ha nem, akkor megtesszük
#define PERSON_H
// és visszaadjuk a valódi tartalmat
struct Person {
      std::string Name;
      int Age;
};
#endif
```

Ez a sok ellenőrzés, illetve a tartalom beillesztése, több ezer fájl esetében komoly fordítási időt emészthet fel, ami nagy céges projektek esetében jelenleg is komoly probléma. Egyes *C++* fordítóknál (ilyen pl. a *GCC*, vagy a *Microsoft Visual C++* fordítója) egy új direktíva bevezetésével a **`pragma once`**-al próbálkoztak. Mivel nem szabványos, ezért szemantikailag különbözhetnek és a *C* nyelvvel se kompatibilis:

```
#pragma once
struct Person {
      std::string Name;
      int Age;
};
```

3.1.2. Konstansok definiálása

Lehetőségünk van egyszerű konstansokat is definiálni a preprocesszor segítségével. Ezek a konstansok nem azonosak a *C* nyelvben lévő konstansokkal, hiszen típus függetlenek és a gyakorlatban itt is csak szövegbehelyettesítés történik, amikor használjuk őket. Így azt is mondhatjuk, hogy lustán, a futás közben értékelődnek ki. A következőképpen tudjuk őket definiálni:

```
#define [a konstans neve] [a konstans értéke]
```

A forráskódban bárhol elhelyezhetünk ilyen konstansokat, de konvenció szerint általában a fordítási egység legelején szoktuk őket definiálni. A következő példában bemutatásra kerül néhány konstans és megnézzük, hogy a preprocesszor milyen kódot állít elő belőlük:

```
#include <stdio.h>
// definiálunk egy egyszerű lebegőpontos értéket PI néven
#define PI 3.14159265359
int main() {
   printf("%f\n", PI);  // output: 3.14159265359
// a preprocesszor lefutása után,
// ezt a kódot dolgozza fel a C fordítója:
// printf("%f\n", 3.14159265359);
   return 0;
}
```

A következő példában már csak kódrészleteket fogok közölni, és megmutatom, hogy miért lehet veszélyes a konstansok használata, ha azok rosszul vannak definiálva:

```
// definiáljuk a következőképpen a 360 fokot:
#define PI 3.14159265359
#define DOUBLE_PI PI + PI
printf("%f\n", DOUBLE_PI);      // output: 6.283185
printf("%f\n", DOUBLE_PI * 2);  // output: 9.424778
```

Amikor beszoroztuk a **DOUBLE_PI** konstanst kettővel, akkor valószínűleg nem az elvárt értéket kaptuk eredményül. Nézzük meg, hogy a fordító valójában milyen kódot generált nekünk:

```
// output: 6.283185
printf("%f\n", 3.14159265359 + 3.14159265359);
// output: 9.424778
printf("%f\n", 3.14159265359 + 3.14159265359 * 2);
```

Látható, hogy valójában jól működött a fordító, csupán mi deklaráltuk rosszul a **DOUBLE_PI** konstanst, mivel nem figyeltünk a lebegőpontos számokon végzett műveletek sorrendjére. Ha zárójelbe tennénk a kifejezést, akkor megoldódna a problémánk és máris a helyes végeredményt kapnánk:

```
#define PI (3.14159265359)
#define DOUBLE_PI (PI + PI)
// mostmár a helyes és elvárt eredményt is kapjuk
printf("%f\n", DOUBLE_PI * 2);  // output: 12.566371
```

Könnyen be lehet látni, hogy már egyszerű konstansok definiálásánál is komoly, nehezen kikövetkeztethető problémákba futunk, ha nem vigyázunk eléggé.

Konstansokat lehetőségünk van érték nélkül is definiálni, ezeket *szimbólumoknak* fogjuk nevezni.

3.1.3. Feltételes fordítás

A feltételes fordítás segítségével a programozó meghatározhatja, hogy a forráskód mely részeit hagyja meg, illetve melyekre nincs szükség a fordításkor. Tipikusan nyomkövetés szempontjából, esetleg platformfüggő kódok írásakor lehet hasznos. Feltételként azt lehet ellenőrizni, hogy egy adott szimbólum definiálva lett-e vagy sem. Az előző példánál a kód duplikációjának elkerülése érdekében már használtuk ezt a nyelvi szerkezetet.

3.1.4. Makrók

Metaprogramozás szempontjából a legérdekesebb nyelvi konstrukciója az előfordítónak maga a *makró*k használata. A *makró*k úgy viselkednek, mint a függvények, lehetnek nekik formális paraméterei, majd ezeknek a *makró*knak a meghívása esetében a forráskódba bemásolódik annak a törzse az aktuális paraméterekkel együtt. Valójában nem történik semmilyen klasszikus értelemben vett függvényhívás, hiszen ugyanúgy, mint az **include** direktíva esetében is, egyszerű szövegbehelyettesítés történik.

A *makró*k használatával a programozónak rengeteg lehetősége nyílik arra, hogy olyan dolgokat is automatizálni tudjon, amit a nyelv szintaxisával, sokkal bővebben kellene kifejteni. Szintaxisa nagyon hasonlít a 3.1.2-ben bemutatott konstansokhoz, és rájuk is érvényes az, hogy ugyanazok a buktatók jöhetnek elő a használatuknál:

```
#define [makró neve]([paraméterek]) [makró törzse]
```

Tegyük fel, hogy szeretnénk makróként definiálni a maximum kiválasztást, mégpedig úgy, hogy a neki átadott elem közül visszatér

```
// így definiálhatjuk a MAX makrót
// a szintaxis hasonló a konstansoknál megismertnél
// azzal a különbséggel, hogy zárójelben felsoroljuk a
// makró formális paramétereit is
#define MAX(a,b) ((a) > (b) ? (a) : (b))
```

Mivel a makrók típus függetlenek, így a formális paraméterei is azok, ezért definiálásnál sem kell meghatározni, hogy milyen értékeket kaphatnak. A legtöbb esetben célszerű (ahol lehet) bezárójelezni a makró törzsében a formális paramétereket, hogy ne forduljanak elő ugyanazok a hibák, mint a konstansok esetében.

```
printf("%f\n", MAX(9,10));  // output: 10
// printf("%f\n", 9 > 10 ? 9 : 10);
```

A makró törzsébe bármit írhatunk, és a **** karakter segítségével akár több sorba is tördelhetjük azt:

```
// segítségével procedúrákat tudunk definiálni
#define DEFINE_PROC(name, params, body) \
  void name(params) { \
    body \
  }
// használata nagyon egyszerű
DEFINE_PROC(sayHello, , printf("Hello World!\n"); )
```

A **#[token neve]** operátorral átalakíthatjuk a token értékét karaktersorozattá. Ezzel lehet egyszerűsíteni a nyomkövetést egyes helyzetekben:

```
// definiáljuk a DEBUG_NUMBER makrót
#define DEBUG_NUMBER(num) printf("debug: " #num " = %d\n", num)
DEBUG_NUMBER(1 + 2 + 3); // output: 1 + 2 + 3 = 6
```

3.2. Metaprogramozás JavaScript nyelven

A *JavaScript* egy prototípus-alapú szkript nyelv, dinamikus típusozással, aminek legfőbb tulajdonsága, hogy a függvényeket kifejezésként is fel lehet használni.

Napjainkra egy nagyon elterjedt nyelv lett belőle, hiszen könnyen tanulható, szintaxisa hasonlít a *C* alapú programozási nyelvekre és a webfejlesztés elsődleges nyelve, amit minden böngésző támogat. A *JavaScript* annyira népszerű lett, hogy egyes rendszereken (mint pl.: *Ubuntu* vagy *Windows*) már tervezik (vagy már be is vezették), hogy önállóan futtatható grafikus alkalmazásokat is lehessen a segítségével készíteni.

Elterjedtségének az oka, hogy rendkívül testreszabható a nyelv, azaz olyan nyelvi elemeket is lehet vele szimulálni (mint pl.: osztályok, tulajdonságok, objektumok közötti öröklődés stb.), amit eredetileg nem támogat.

Ebben a fejezetben viszont a metaprogramozáshoz szükséges nyelvi eszközeit fogjuk részletesebben megvizsgálni.

3.2.1. JavaScript nyelvi alapjai, érdekességei

Lássuk, hogy pontosan milyen nyelvi lehetőségek állnak rendelkezésünkre ahhoz, hogy dinamikusan, futásidőben újabb és újabb struktúrákat hozzunk létre.

Függvények a nyelvben

Az egyik legfontosabb lehetőségünk az lesz, hogy a függvények a nyelvben ugyanúgy objektumokként vannak definiálva, ezért bármilyen változónak értékül adhatjuk, illetve akár függvények aktuális paramétereként is használhatjuk őket.

```
// a max nevű változónak egy függvényt adunk értékül
var max = function(a, b) {
     return a > b ? a : b;
}
```

```
// mivel a max egy függvényt tartalmazó változó, ezért
// függvényként fog viselkedni, vagyis ugyanúgy meghívhatjuk
console.log(max(1, 3)) // output: 3
```

A függvények akár visszatérési értékek is lehetnek egy függvényen belül, ami rengeteg lehetőséget ad a fejlesztő kezébe. A következő példa megmutatja, hogy hogyan is lehet használni a függvényeket visszatérési értékként:

```
// definiálunk egy add függvényt,
// amivel két számot lehet
var add = function(a) {
     // visszatérési értékként egy függvényt fogjunk visszaadni
     return function(b)
          // itt végezzük el végül az összeadást
          return a + b;
     }
}
// definiálunk egy olyan addToOne függvényt is,
// amivel az (1 + b) műveleteket lehet elvégezni
var addToOne = add(1);
// mindkét esetben az eredmény ugyanannyi lesz
console.log(add(1)(10));    // output: 11
console.log(addToOne(10))   // output: 11
```

A függvényeket nemcsak a **()** operátorral lehet meghívni, hanem használhatjuk az **apply()** és **call()** metódusait is. Ezekkel a metódusokkal megmondhatjuk, hogy a függvény **this** paramétere éppen melyik objektumra mutasson[1].

```
// definiáljuk a person objektumnak a toString metódust
person.toString = function() {
     // a this ebben a pillanatban a person-ra mutat
     return this.name + " (" + this.age + ")";
};
// definiálunk egy bobby nevű változót, aminek
// megegyeznek az adattagjai a person objektummal
var bobby = {
```

[1] Ha az adott függvényünk a globális környezetben (*global scope*) van definiálva, akkor a **this** paraméter alapértelmezetten a **window** objektumra mutat.

```
      name: "Bobby",
      age: 60
}
// ha alapértelmezetten használjuk,
// akkor a this a person-ra fog mutatni
console.log(person.toString())        // output: John Doe (34)

// ha az apply első paraméterének megadjuk a bobby objektumot
// akkor a this a bobby-ra fog mutatni
console.log(person.toString.apply(bobby))  // output: Bobby (60)
```

Az **apply()** és a **call()** között az a különbség, hogy az előbbinek az aktuális paramétereket egy tömbben, míg az utóbbinak egyenként kell átadni, de mindkettő függvény első paraméterének azt kell megadni, hogy a **this** paraméter hova mutasson.

Objektumok a nyelvben

A *JavaScript* nyelvben szinte minden objektumnak számít. Egyedül a vezérlési szerkezetek és a primitív típusok (**number**, **string**, **boolean**, **undefined**) nem számítanak annak. Az objektumok (**Object**) több olyan tulajdonsággal is rendelkeznek, amire a későbbiekben szükségünk lehet.

Az egyik ilyen tulajdonság az az, hogy hogyan reprezentálja a benne található adattagokat és metódusokat, ugyanis minden objektum valójában egy asszociatív tömb és az adattagok nevei a kulcsok a hozzájuk tartozó értékekhez. Ezt szemlélteti a következő példa is:

```
// definiálunk egy person nevű objektumot
var person = {
      name: "John Doe",  // a name adattag
      age: 34            // az age adattag
}
// a . operátorral érjük el a name adattagot
person.name = "John";
// de az előző utasítás ekvivalens ezzel,
// ugyanis az objektumok asszociatív tömbökként
// vannak értelmezve:
```

```
person["name"] = "John";
console.log(person["age"])  // output: 34
```

Egy másik jellegzetes tulajdonsága az objektumoknak (ami a *Ruby* szkriptnyelvnél is szintén jelen van) az, hogy bármikor újabb adattagokkal egészíthetjük őket. Továbbá ha olyan adattagra szeretnénk hivatkozni, amit nem tartalmaz maga az objektum, akkor fordítási (vagy inkább futási) hiba dobása nélkül egy **`undefined`** értékkel tér vissza. Ezt szemlélteti a következő példa (ami az előző folytatása):

```
// mivel még nem deklaráltuk a citizenship-et
// ezért undefined értéket kapunk eredményül
console.log(person.citizenship)  // output: undefined
// egyszerűen kiegészíthetjük plusz adattaggal
person.citizenship = "hun";
// ezután már undefined helyett a "hun" értéket kapjuk
console.log(person.citizenship)  // output: hun
```

[TODO]

3.3. Metaprogramozás Scala nyelven

A *Scala* egy objektum-funkcionális programozási (*object-functional programming*) és szkript nyelv, amit *Martin Odersky* kezdett el fejleszteni a 2000-es évek elején és először 2003-ban jelent meg a nagyközönség előtt.

A *Scala* az utóbbi években egy rendkívül sikeres nyelv lett, amit leginkább a szintaktikájának, bővíthetőségének és több programozási paradigma támogatásának köszönhet. A programozási nyelv egy nagyon sajátos módon vegyíti a funkcionális nyelvekre jellemző tulajdonságokat (körrizés, mintaillesztés, algebrai adattípusok, lusta kiértékelés, végrekurzió stb.) az objektum-orientált programozás paradigmájával. Ehhez hozzájárul az eddig megszokott imperatív nyelvektől (*Java, C#, C++* stb.) erősebb és szigorúbb típusrendszer, illetve olyan szintaktikai szabályok, amivel sokkal kifejezőbb és olvashatóbb kódot tudunk írni.

Másik nagy előnye a nyelvnek, hogy a *JVM* (*Java Virtual Machine*) számára alkalmas bájtkódra fordítja a forráskódot, így bármilyen *JVM* alapú platformon képes futni az alkalmazásunk. Ami viszont az informatika ipar nagyvállalatait is meggyőzte, az az volt,

hogy a *Scala* nyelven írt programok kompatibilisek a *Java* nyelven írt alkalmazásokkal, ezért a már meglévő *Java* technológiákat változtatás nélkül lehet használni a *Scala*-ban is. Volt egy alternatív implementáció a nyelvnek a *.NET platformra* is, de erőforrás és támogatottság hiányában ezt a projektet végül leállították.

A nyelv a 2.10-es verziójától elkezdte kísérleti jelleggel támogatni a makrókat, ami a metaprogramozás egyik legerősebb eszközei és olyan problémákat lehet megoldani velük, amit eddig a nyelv erős típusrendszere ellenére se lehetett teljesen elérni. Ezek a nyelvi eszközök még erőteljesen fejlesztés és tervezés alatt állnak, ezért a közeljövőben még változhat a használati módjuk.

3.3.1. Scala makrókról általánosságban

A Scala makróinak ötlete először *Eugene Burmako* fejéből pattant ki még 2011-ben, aki akkoriban egy elsőéves PhD hallgató volt *Martin Odersky* kutatói projektében. Az eredeti ötlet az volt, hogy a nyelvbe egy teljesen önálló metaprogramozási eszközt, a makrókat, implementáljanak.

Martin sok ideig szkeptikus volt a projekt sikerével kapcsolatban, illetve meg kellett győzni őt arról is, hogy makrók használata rendkívül egyszerű és szépen illeszkedik a nyelv szintaktikájához. Végül az eredmények őt is meggyőzték, ami mi sem bizonyít jobban, hogy a nyelv 2.10-es verziójától már ki is lehet próbálni. A makrók bevezetéséről még többet lehet olvasni a [1] előadás diákon.

A *C/C++* nyelvektől eltérően a *Scala*-ban a makrók nem a forráskód szövegén, hanem a szintaktikus elemző által generált absztrakt szintaxisfán (AST) végez műveleteket. Ezzel a módszerrel fordítási időben generálhatunk szintaxisfát, esetleg transzformálhatjuk azokat. De nemcsak kódgenerációra alkalmasak, hiszen különböző típusműveleteket, fordítási időben végrehajtható ellenőrzéseket is végezhetünk vele.

3.3.2. Függvény makrók

A *függvény makrók* (*def macros*) jelentek meg legelőször a Scala nyelvben szigorúan kísérleti lehetőségként és a mai napig támogatja őket a nyelv. Minden függvény makrót a következőképpen kell elképzelni:

```
// bármilyen módon paraméterezhetjük őket
def macro(param: Type) : ReturnType = macro implReference
```

A függvény makrók nagyon hasonlítanak az egyszerű függvényekre, kivéve a függvénytörzsük, hiszen ott hívjuk meg a **macro** kulcsszóval a makrónk implementációját. A makró használata teljesen természetes, ugyanúgy kell meghívni, mint egy egyszerű függvényt:

```
// Az alábbi módon definiáljuk a debug makrót
def debug(param: Any) : Unit = macro debugImpl
...
val x = 10
debug(x + 1 > 0) // ugyanúgy hívjuk meg, mint a függvényeket
```

A függvény makrók a következőképpen működnek: a fordító a típusellenőrzés alkalmával felismeri, hogy mikor hívtunk meg makrókat. Amikor ez megtörténik, egyből meghívja a makróhoz tartozó implementációt, átadva neki az aktuális paraméterek absztrakt szintaxisfáját [2]. Miután az implementáció lefutott és kiértékelődött egy szintaxisfával tér vissza. Visszatérve az előző példánkhoz, a **debug** makró kiértékelésekor a következő dolog történik:

```
val x = 10
debug(x + 1 > 0) // meghívjuk a debug makrót
// megkeresi a fordító a típusellenőrzés alkalmával a definíciót
// és azután meghívja a debugImpl implementációt a következő
// paraméterezéssel
debugImpl(context)(<[ x + 1 > 0 ]>)
```

A **<[…]>** nem nyelvi elem, csak az **x + 1 > 0** kifejezés absztrakt szintaxisfáját reprezentálja. Ha jobban szemügyre vesszük az implementáció hívását, akkor láthatjuk, hogy a fordító nemcsak a szintaxisfát adta át paraméterül, hanem egy **context** változót is, ami a környezet kontextusát reprezentálja. Ezen a változón keresztül tudunk olyan információkhoz jutni, amit a fordító a fordítás alatt gyűjtött össze.

[2] Érthető módon a paraméter értéke általában nem áll rendelkezésre fordítási időben.

A gyakorlatban a generált szintaxisfák a **scala.reflect.api.Trees** nevű trait[3]-ből származnak. Az előző példa kifejezésének a szintaxisfája a következőképpen néz ki:

```
Expr(Apply(Select(Apply(Select(Select(This(TypeName("Test")),
TermName("x")), TermName("$plus")), List(Literal(Constant(1)))),
TermName("$greater")), List(Literal(Constant(0)))))
```

A **TypeName("Test")** elem csak azért jelent meg, mert a **Test** osztályon belül használtuk az **x** változót. Láttuk, hogy hogyan lehet használni a makrókat, most pedig nézzük meg, hogy hogyan van megvalósítva a **debug** makró implementációja:

```
// betöltjük a szükséges névtereket
import language.experimental.macros
import scala.reflect.macros.whitebox.Context
// a debug makró implementáció, a paraméterek hasonlítanak a
// debug függvényhez, attól eltekintve, hogy nem a formális
// paraméterek típusának megfelelő értéket kell átadni nekik,
// hanem a szintaxisfájukat
def debugImpl(c: Context)(param: c.Expr[Any]) : c.Expr[Unit] = {
    // a c.universe névtéren belül találhatóak az AST
    // transzformálását segítő függvények
    import c.universe._
    // a show függvény egy adott AST-t visszaalakít emberileg
    // olvasható karaktersorozattá
    val paramRep : String = show(param.tree)
    // a paramRep-ből újra egy AST-t kell készítenünk, ezt úgy
    // érhetjük el, hogy becsomaguljuk
    // egy Literal(Constant(...)) csúcsba
    val paramRepTree = Literal(Constant(paramRep))
    // a literálból egy kifejezést kell csinálnunk
    val paramExpr = c.Expr[String](paramRepTree)
    // végül a reify makró segítségével legeneráljuk azt a
    // szintaxisfát, amit a debug függvény
    // hívásával cserélünk ki
    reify {
      println(paramExpr.splice + " = " + param.splice)
```

[3] *Scala*-ban interfészek helyett *trait*-ek vannak, amik leginkább az absztrakt osztályokhoz hasonlítanak funkcionalitásukat tekintve.

```
    }
}
```

A **debug** makró úgy működik, hogy fordítási időben kiértékeli a neki átadott aktuális paraméter szintaxisfáját és a hívást egy **println** függvénnyel helyettesíti, így futási időben már a konzolra íratja ki, hogy az adott kifejezésnek, mi az aktuális értéke. A példából kiindulva a következő jelenik meg a képernyőn:

```
val x = 10
debug(x + 1 > 0)
// => println("Test.this.x.+(1).>(0)" + " = " + "true")
// output: Test.this.x.+(1).>(0) = true
```

Érdekesség, hogy a **reify** függvény is egy makró, ami az adott kifejezésből legenerálja a nekünk szükséges absztrakt szintaxisfát, így könnyítve meg a fejlesztőknek a kódgenerációt. A **splice** függvényt csak a **reify** makrón belül lehet meghívni, és arra való, hogy az adott szintaxisfa futási időben kiértékelésre kerüljön. Ha nem így teszünk, akkor fordítási hibát kapunk.

3.3.3. Generikus függvény makrók

A nyelv típusrendszere lehetővé teszi számunkra, hogy generikus típusokat is definiálhassunk a *Scala*-ban. Tegyük fel, hogy felmerül az igény arra, hogy a **debug** makrónkat kifejezések közepén is szeretnénk használni. Viszont, így már a visszatérési értéke nem lehet **Unit**[4] és **Any**[5] sem, hiszen a kifejezés pontos típusával kell visszatérnie. Ezt csak generikus típussal tudjuk implementálni, amire lehetőségünk is van, hiszen *generikus függvény makrókat* (*generic def macro*) is definiálhatunk a nyelvben.

Mind a makró definíciója, mind pedig a makró implementációja lehet generikus, amit ki is fogunk használni a későbbiekben. Ha a makró implementációja tartalmaz típusparamétert, akkor explicite át kell adnunk neki, amikor a makró definíciójának függvénytörzsében hívjuk meg azt. Nézzük meg, hogy hogyan alakítottuk át a **debug** makrónk definícióját:

---

[4] A **Unit** típust a típusos λ-kalkulusból lehet ismerős. *Scala*-ban ez a típus jelzi, hogy az adott függvénynek nincs visszatérési értéke.

[5] Az **Any** típusból származik minden más típus a *Scala*-ban, azaz megfelel a .NET keretrendszerben ismert **System.Object** típussal.

```
// a T típusparaméter fogja tárolni, hogy milyen típusú
// kifejezéssel hívtuk meg a makrót
def debug[T](param: T): T =
   // átadjuk az implementációnak a típusparamétert
   macro debugImpl[T]
```

Az implementáció szignatúrája is változik egy kicsit, hiszen egy új típusparaméterrel egészítjük ki azt, illetve a visszatérési érték típusát is megváltoztatjuk:

```
// ugyanúgy kiegészítjük egy T típusparaméterrel
def debugImpl[T : c.WeakTypeTag](c: Context)(param: c.Expr[T])
   : c.Expr[T] = { /* ... */ }
```

Ebben az esetben a **`c.WeakTypeTag`** azt jelzi a fordítónak, hogy a **`T`** típusparaméter az alkalmazás oldalán lesz példányosítva akkor, amikor a fordító kibontja az adott makrót. A függvény törzse majdnem teljesen megegyezik, az utolsó sort leszámítva, ugyanis a generált kódnak vissza is kell térnie a kifejezéssel. Így a **`reify`** a következőképpen változik:

```
reify {
  println(paramExpr.splice + " = " + param.splice)
  param.splice  // visszatérünk a kifejezés értékével
}
```

3.3.4. Implicit makrók

[TODO]

3.3.5. Sztringek interpolációja

[TODO]

3.3.6. Quasiquote-ok használata a kódgenerációhoz

[TODO]

3.3.7. Makró annotációk

Egy másik metaprogramozási eszköz, amit a nyelv kínál az a *makró annotációk* (*macro annotations*) használata. Az annotációk (vagy .NET keretrendszerben attribútumok) használata nem újkeletű nyelvi eszköz az imperatív nyelveknél, hiszen már elég régóta léteznek a *Java*-ban és *C#*-ban is, de csak futásidőben lehet feldolgozni őket a reflexió segítségével.

A *Scala* ugyanúgy támogatja az annotációkat, mint a *Java*, viszont a makrók bevezetésével új lehetőségek nyílnak meg a fejlesztők számára. A makró annotációk a Scala 2.10 verziójától már elérhetőek a *macro paradise*[6] kiterjesztésen keresztül.

A makró annotációkat nemcsak osztályokat (**`class`**), objektumokat (**`object`**), hanem bármilyen önálló definíciót meg lehet velük jelölni. Segítségükkel az általuk megjelölt típusokat ki lehet egészíteni új függvényekkel, adattagokkal, de akár teljesen új típusokat is létrehozhatunk vele.

A lehetőségek tárháza szinte végtelen, egyszerűen csak arra kell figyelni, hogy felelősségteljesen használjuk ezt a nyelvi eszközt, hiszen gyorsan a kódolvashatóság és a biztonság rovására mehet.

A makró annotációk használatát egy példán keresztül fogom bemutatni. Egy olyan makrót fogunk létrehozni, ami a megjelölt átlagos osztályokat fogja felvértezni a *case class*-okhoz hasonló tulajdonságokkal, azaz támogatni fogja a mintaillesztéseket és a **`new`** operátor nélküli példányosítást.

Még mielőtt azonban belefognánk, ismerjük meg kicsit közelebbről, hogy hogyan is működnek a *case class*-ok a *Scala* nyelvben.

Case class-ok a Scala-ban

Scala-ban a *case class*-ok teljesen normális osztályok, azzal a különbséggel, hogy a konstruktor paramétereik ki vannak exportálva és támogatják a mintaillesztést. Szintaxisuk nagyon egyszerű:

[6] A *macro paradise* egy plugin a *Scala* fordítóhoz (lásd [2]). Arra tervezték, hogy megbízhatóan működjön a *scalac* fordítóval és a legújabb, bevezetés előtt álló makró fejlesztéseket ki lehessen próbálni, még mielőtt kijönne az új fordító következő verziója.

```
// készítünk egy Person osztályt, aminek két adattagja lesz:
// a name a személy nevét reprezentálja
// az age a személy életkorát reprezentálja
case class Person(name: String, age: Int)
```

Használatuk teljesen megegyezik az osztályokéval, azt leszámítva, hogy a példányosításnál nem kell a **new** operátort használni. Ezt mutatja az alábbi példa is:

```
// példányosítunk egy Person objektumot
val p = Person("John Doe", 30)
val name = p.name  // lekérdezzük a nevét
val age = p.age    // és az életkorát
println(s"$name ($age)")  // output: John Doe (30)
```

Ha egy átlagos osztály lenne a **Person**, akkor a fenti módon kellene kinyernünk a nekünk szükséges adatokat. De egy *case class* esetében ezt elegánsabban is megtehetjük a mintaillesztés segítségével:

```
// a p-ből kinyerjük a name és age változókba
// a p.name és p.age értékeket
val Person(name, age) = p
```

A *case class*-ok viszont a nevük ellenére nem térnek el az átlagos osztályoktól. A háttérben valójában az történik, hogy a fordító egy átlagos osztályt generál és hozzá egy **object**-et azonos néven, amiben definiálja az **apply** és **unapply** függvényeket.

Az **apply** függvény akkor hívódik meg, amikor az **object**-et függvényként akarjuk használni, míg az **unapply**-t a fordító hívja meg akkor, amikor a mintaillesztés történik. A **Person** osztály esetében valami hasonló kód generálódik a háttérben:

```
// egy átlagos osztály generálódik
class Person(val name: String, val age: Int)


// egy object is generálódik az osztályhoz
object Person {
    // az apply függvényen keresztül is tudjuk
    // példányosítani az osztályt
    def apply(name: String, age: Int) = new Person(name, age)
    // míg az unapply a mintaillesztésnél hívódik meg
```

```
    // és egy Option típussal fog visszatérni
    def unapply(p: Person) : Option[(String, Int)]
          = Some((p.name, p.age))
}
```

Persze a gyakorlatban ennél egy kicsivel több is történik, hiszen más segédfüggvényeket is elkészít a fordító (**`toString`** stb.).

Case class makró

[TODO]

```
object CaseClassMacro {
  // a makró implementációja
  def implementation(c: Context)(annottees: c.Expr[Any]*):
c.Expr[Any] = {
    // betöltjük a kontextus univerzumát
    import c.universe._
    // lekérjük az annotált objektumokat egyenként
    annottees.map(_.tree).toList match {
      // mintaillesztéssel kiválogatjuk azokat az eseteket,
      // amikor egy osztályt jelöltünk meg az annotációnkkal
      case q"class $name(..$params) extends ..$parents { ..$body
}" :: Nil => {
        // lekérjük az osztály nevét
        val termName : TermName = name.toTermName
        // kinyerjük az elsődleges konstruktor formális
        // paramétereinek a nevét
        val parameterNames = params.map(param => param.name)
        // lekérdezzük az elsődleges konstruktor formális
        // paramétereinek a típusát
        val parameterTypes = params.map(
           (param : ValDef) => param.tpt
        )
        // beállítjuk a adattagok elérését
        val selections = parameterNames.map((param: TermName) =>
Select(Ident(newTermName("obj")), param))
        // végül elkészítjük az absztrakt szintaxisfát (AST),
```

```
        // amivel majd visszatérünk
        // az AST-t quasiquote segítségével készítjük el
        val tree = q"""
          // az eredeti osztályt is legeneráljuk azzal a
          // különbséggel, hogy az elsődleges konstruktor
          // láthatóságát protected-re állítjuk
          class $name protected (..$params)
                      extends ..$parents { ..$body }
          // majd elkészítjük hozzá az extractor object-umot
          object $termName {
            // így mostmár mint case class-t
            // lehet példányosítani
            def apply(..$params) = new $name(..$parameterNames)
            // a mintaillesztéshez szükséges unapply
            // függvényt is legeneráljuk
            def unapply(obj: $name) :
Option[(..$parameterTypes)] = Some((..$selections))
          }
        """
        // visszatérünk a legenerált fával
        c.Expr[Any](tree)
      }
      case _ => {
        // ha a felhasználó rossz helyen használta
        // az annotációt, akkor fordítási hibát dobunk
        c.error(c.enclosingPosition, "Unsupported expression!")
        // egy üres fával térünk vissza
        c.Expr[Any](EmptyTree)
      }
    }
  }
}
```

3.3.8. Makró csomagok

[TODO]

3.4. Metaprogramozás Boo nyelven

A *Boo* egy objektum-orientált, statikusan típusos, általános célú programozási nyelv *Microsoft .NET* és *Mono* keretrendszerekre. A *Python* nyelv szintaxisa ihlette magát a nyelvet, amelyet összekötöttek a .NET keretrendszer adta lehetőségekkel, és olyan nyelvi eszközökkel, mint a *generátorok*, *multimetódusok* (*multimethods*), *típuskikövetkeztetés* és *makrók* támogatása. A nyelv tervezői különös figyelmet fordítottak arra, hogy mind a nyelv, mind pedig maga a fordító is könnyen kiterjeszthető és bővíthető legyen.

3.4.1. Boo szintaktikus makrók

A *szintaktikus makrók* (*syntactic macros*) egy rendkívül érdekes nyelvi eszközzel bővítik a *Boo* nyelv fegyvertárát. Segítségükkel a programozó képes fordítási időben kódot (pontosabban szintaxisfát) generálni, így téve a forráskódot sokkal kifejezőbbé és kompaktabbá.

A makrók használata nagyon egyszerű, a nyelv standard könyvtára több ilyen beépített lehetőséget is biztosít a fejlesztők számára. Egyszerűen csak úgy kell használni, mintha függvényt hívnánk meg, attól eltekintve, hogy nem kell kiírni a zárójeleket utána. Ezután a fordító lefuttatja a makró törzsét és az eredményt visszaírja a meghívás helyére, ami egy szintaxisfa.

A következőkben megnézünk pár beépített szintaktikus makrót, amivel képet kaphatunk arról, hogy miért is olyan erőteljes nyelvi eszközök a programozók kezében.

Az assert makró

Amikor valamilyen függvényt, vagy procedúrát tervezünk, implementálunk, az első dolgunk az, hogy a bemeneti paraméterek értékét ellenőrizzük, amivel garantáljuk, hogy a felhasználó a megfelelő eredményt fogja kapni a meghívás esetén.

Imperatív nyelvek esetében tipikusan valamilyen elágazás segítségével szoktuk ellenőrizni az előfeltételeket, majd azokra valamilyen módon reagálunk (gyakran kivételek dobásával). Szemantikailag helyes ez a megoldás, viszont megtöri a kód olvashatóságát és elrejti előlünk a függvény valódi feladatát.

Erre a problémára a *Boo* készítői az *assert* makró bevezetésével próbáltak meg válaszolni. Egy vagy két paraméterrel hívhatjuk meg, amik mindkettő esetben egy elágazást fognak nekünk generálni:

```
// használata a következőképpen néz ki:
assert <kifejezés>
// ha ehhez az utasításhoz ér a fordító,
// akkor az alábbi szintaxisfával fog visszatérni:
unless (<kifejezés>):
      raise Boo.AssertionFailedException('(<kifejezés)')
```

Egy másik túlterhelt változatának kettő aktuális paramétert adhatunk át neki, ahogy azt a példa is mutatja:

```
// használata a következőképpen néz ki:
assert <kifejezés>, <üzenet>
// ha ehhez az utasításhoz ér a fordító,
// akkor az alábbi szintaxisfával fog visszatérni:
unless (<kifejezés>):
      raise Boo.AssertionFailedException(<üzenet>)
```

Könnyen látható, hogy rendkívül egyszerű használni, mégis nagy segítséget tud nyújtani a programozók számára.

A lock makró

Párhuzamos szálakon végzett műveletek esetében szükség lehet a szálak között valamilyen információ megosztására.

Magának a *C#* nyelvnek van egy *lock* nevezetű vezérlési szerkezete, ami garantálja azt, hogy a törzsébe egyszerre csak egy szál léphet be. De ez valójában csak egy szintaktikai cukorka, hiszen a fordító a **`System.Threading.Monitor`** statikus osztály **`Enter()`** és **`Exit()`** metódusai közé illeszti a *lock* szerkezet törzsét.

Felvetődik a kérdés, hogy miért kellett egy új szintaktikai elemet bevezetni ahhoz, hogy használhassuk ezt a funkcióját a .NET keretrendszernek? A válasz természetesen az, hogy szemantikailag lehet, hogy megegyezik mindkét megoldás, de újra csak kódolvashatóság szempontjából mégis sokkal kifejezőbb a *lock* használata.

Ezt a nyelvi szerkezetet azonban ugyanúgy meg lehet fogalmazni a *Boo* nyelvben szintaktikai makróként, mint az előbb az *assert*-et. Ezt a fejlesztők *lock* makrónak nevezték el. Az alábbi példa szemlélteti, hogy mi történik valójában a háttérben, a makró milyen szintaxisfát állít elő:

```
// használata a következőképpen néz ki:
lock <kifejezés>: <blokk>
// az alábbi szintaxisfa generálódik a makró kiértékelésénél
__monitor1__ = <kifejezés>
// belépünk a lezárt kódrészletbe
System.Threading.Monitor.Enter(__monitor1__)
try:
    // ide kerül a lezárásra váró blokk
    <blokk>
ensure:
    // bármilyen hiba is történjen, az ensure rész
    // biztosít minket arról, hogy lépjünk ki a monitorból
    System.Threading.Monitor.Exit(__monitor1__)
```

A using makró

A .NET keretrendszer virtuális gépe leveszi a terhet a felhasználó válláról azáltal, hogy a memória kezelését a *szemétgyűjtő* (*garbage collector*) végzi el. Azonban lehetnek olyan esetek, amikor szeretnénk pontosan irányítani azt, hogy egy erőforrás igényesebb objektum mikor szabadítja fel az általa lefoglalt memóriát. Az ilyen objektumokat *felszabadítható* (*disposable*) objektumoknak nevezzük és egy **`IDisposable`** interfészt valósítanak meg.

A *using* vezérlési szerkezet használata garantálja a felhasználónak, hogy az ilyen objektumok erőforrásai a törzsének a lefutása után biztosan felszabadulnak, azaz meghívódik rajta az **`IDisposable.Dispose()`** utasítása, történjen bármi. Ez megint csak egy szintaktikai cukorka a felhasználók részére, mégis javítja a kódbiztonságot (hiszen a programozó így biztosan nem felejti el meghívni a **`Dispose()`** metódust) és a kód olvashatóságát.

Ezt a nyelvi lehetőséget ugyancsak ki lehet váltani a *Boo* nyelvben egy szintaktikai makróval, aminek a *using* nevet adták a nyelv készítői. Az alábbi példa a makró használatát szemlélteti, illetve azt, hogy milyen szintaxisfa generálódik belőle:

```
// az egyik lehetőség, hogy csak az objektumot
// adjuk át a makrónak, illetve a blokkot
using <objektum>: <blokk>
// ebben az esetben az alábbi szintaxisfát kapjuk vissza
// a makró kiértékelésénél
try:
    <blokk>     // a using blokkja
ensure:
    // a blokk lefutása utána garantáltan felszabadítjuk
    // az objektum által lefoglalt erőforrásokat
    if (__disposable__ = (<objektum> as System.IDisposable)):
        // meghívjuk a Dispose() metódust
        __disposable__.Dispose()
        // és beállítjuk null értékre
        __disposable__ = null
```

Egy másik megvalósításnál nemcsak az objektumot, hanem az objektum inicializálását is átadhatjuk a makrónak:

```
// nemcsak az objektumot, hanem magát az inicializálást is
// megadjuk a makrónak
using <objektum> = <kifejezés>: <blokk>
// az előző megvalósításhoz nagyon hasonló szintaxisfát
// generál a using makró
try:
    // a blokk előtt még lefut az inicializálás
    <object> = <expr>
    <block>
ensure:
    if (__disposable__ = (<object> as System.IDisposable)):
        __disposable__.Dispose()
        __disposable__ = null
```

A szintaktikai makrók működése a Boo nyelvben

Még mielőtt rátérnénk a saját makrók definiálására, meg kell értenünk, hogy hogyan is működnek a háttérben ezek a nyelvi elemek. A szintaktikai makrók a Boo nyelvben teljes hozzáférést biztosítanak a fordítóhoz és a forráskód teljes absztrakt szintaxisfájához.

Mivel a *Boo* nyelv egy objektum-orientált nyelv, ezért a makrók is *CLI* (*Common Language Infrastructure*) osztályokként vannak reprezentálva a gyakorlatban, amik a **`Boo.Lang.Compiler.IAstMacro`** interfészt valósítják meg. Ez a megoldás azt jelenti, hogy a szintaktikai makrókat bármilyen *CLI* nyelven meg lehet írni, nem kell ragaszkodnunk a *Boo* nyelvhez.

Miután a *Boo* fordító feldolgozta szintaktikailag a kódot (lefutatta a szintaktikus elemzőt) utána egyből meghívja a felhasználó által használt makrókat. A makrók törzse kiértékelődik és visszatérési értékként egy szintaxisfát kapunk, ami a makró helyett lesz a forráskódban.

Az előző folyamatot úgy oldja meg, hogy amikor a fordító egy ismeretlen szintaktikai szerkezetet talál a fordítás közben, akkor megpróbálja megkeresni a neki megfelelő **`IAstMacro`** interfészt megvalósító osztályt. Egy egyszerű névkonvenció alapján teszi ezt meg: minden ilyen osztálynak a makró nevével kell kezdődnie és a **`Macro`** szóval kell végződnie. Továbbá az is elvárás, hogy *Pascal Case* elnevezési konvenciót kell használni, azaz minden szónak nagybetűvel kell kezdődnie.

Ha megtalálta, akkor példányosítja azt és utána megkéri az objektumot, hogy fejtse ki az adott makrót, azaz meghívja rajta az **`Expand()`** metódust. Az **`Expand()`** metódus felelős azért, hogy a makró helyét valamilyen szintaxisfával helyettesítse. Ha olyan eredményt adtunk értékül, ami megsérti a nyelv szintaktikai szabályait, akkor fordítási hibát fogunk kapni.

Két további osztály, a **`DepthFirstVisitor`** és a **`DepthFirstTransformer`**, nyújt segítséget a programozónak ahhoz, hogy be tudja járni a fordító által generált absztrakt szintaxisfát. Ezekből az osztályokból akár örököltethetünk is és saját bejáró algoritmusokat implementálhatunk a makróinkhoz.

Az egyke (singleton) tervezési minta implementálása makróval

Tegyük fel, hogy szeretnénk egy olyan makrót készíteni, ami egy megadott osztályból készít egy hozzá tartozó *egyke* (*singleton*) osztályt. [TODO]

3.5. Text Template Transformation Toolkit (T4)

A Microsoft egyik alapvető szöveggeneráló eszköze a *Text Template Transformation Toolkit* (későbbiekben csak *T4*), amit több technológiájánál, úgymint *Windows Communication Foundation (WCF)*, *Entity Framework (EF)*, használ előszeretettel.

A *T4* a C/C++ előfordítójának általánosításaként is értelmezhető, ugyanis nemcsak *C#* nyelven, hanem *Visual Basic* nyelven is lehet programozni, nem beszélve arról, hogy sokkal több lehetőséget biztosít a fejlesztő számára.

A fejlesztők a *PHP* nyelvhez nagyon hasonló megoldással álltak elő a *T4* esetében is. Itt is vannak kitüntetett blokkok, amik között a fordító értelmezi a forráskódot és végrehajtja, míg a blokkon kívüli szöveget egy az egyben legenerálja.

A *T4* szöveg sablonokat három különálló részre lehet osztani: *direktívák*, *szöveg* és *vezérlő blokkok*.

3.5.1. T4 direktívák

A T4 direktívái általános információkat szolgáltatnak a sablont generáló motornak, hogy hogyan transzformálja a kódot és milyen kimeneti fájlt állítson elő. A direktíváknak a szintaxisa az alább látható módon van definiálva.

```
<#@ DirektívaNeve [AttribútumNeve = "AttribútumÉrtéke"] ... #>
```

Több direktívát is megkülönböztet a *T4* attól függően, hogy mit is szeretnénk beállítani. Az alábbiakban az opcionális attribútumokat kapcsos zárójelek közé fogom írni.

T4 sablon direktíva

A *T4 sablon direktívával* (*T4 Template Directive*) azt állíthatjuk be, hogy hogyan kellene feldolgozni az adott sablont. A szintaxisa a következőképpen néz ki:

```
<#@ template [language="[sablon nyelve]"] [culture="[kultúra]"]
[inherits="[ősosztály neve]"] [visibility="[láthatóság]"] #>
```

Az egyik legfontosabb attribútumon, a *language* attribútumon keresztül adhatjuk meg, hogy mely programozási nyelvet szeretnénk használni a sablon generálására (*C#* és *Visual Basic* közül választhatunk). Alapértelmezetten a *C#* van beállítva.

Az *inherits* attribútummal öröklődést is definiálhatunk, ugyanis ezen keresztül adhatjuk meg, hogy az adott sablonhoz generált osztályunk, mely osztályból öröklődjön.

A *visibility* attribútummal pedig a sablonhoz generált osztályunknak milyen láthatóságot szeretnénk beállítani. Két opció közül választhatunk: publikus (*public*) és internál (*internal*).

T4 paraméter direktíva

Ha külső környezetből használjuk a sablonok generálását (ilyen lehet, amikor futásidőben akarjuk legenerálni egy másik alkalmazásunkban), akkor felmerülhet az igény arra vonatkozóan, hogy különböző paraméterekkel lássuk el a sablonjainkat, amivel a szöveg generálását szabályozhatjuk. Ezt az úgynevezett *paraméter direktívákkal* (*T4 Parameter Directive*) tudjuk elérni a gyakorlatban.

```
<#@ parameter type="[típus neve]" name="[paraméter neve]" #>
```

A fenti sorban a paraméter direktíva szintaxisa látható. Két attribútumot kell átadni a számára. A *type* attribútummal a paraméter típusát határozzuk meg, aminek kötelezően egy .NET keretrendszerbeli típusnak kell lennie, míg a *name* attribútummal a paraméter nevét mondhatjuk meg.

Ha megadtunk egy ilyen direktívát, akkor utána már egyszerűen használhatjuk a sablonunkban, azzal a névvel, amit meghatároztunk neki.

T4 kimeneti direktíva

A *kimeneti direktívával* (*T4 Output Directive*) határozhatjuk meg, hogy a sablont generáló osztály, milyen kiterjesztésű fájlba generálja a végeredményt. Két attribútumot tudunk átadni neki, az *extension*-el a kimeneti fájl kiterjesztését, míg az *encoding*-al a karakterkódolását határozhatjuk meg. Az alábbi sorban látható a direktíva szintaxisa.

```
<#@ output extension=".[generált fájl kiterjesztése]"
            [encoding="karakterkódolás"] #>
```

T4 szerelvény direktíva

A .NET keretrendszer az újrafelhasználható osztályokat, típusokat úgynevezett *szerelvényekben* (angolul *assembly*) tárolja. Sablonok készítésénél is szükség lehet olyan funkciókra, amik nem feltétlenül találhatóak meg az alapértelmezetten elérhető névterekben. Ekkor jöhet jól az úgynevezett *szerelvény direktíva* (*T4 Assembly Directive*), amivel újabb szerelvényeket lehet betölteni a sablon számára.

Egy kötelező attribútumot a *name* attribútum értékét kell átadnunk, amivel meghatározhatjuk, hogy pontosan melyik szerelvényt szeretnénk betölteni. Az attribútum értéke kétféle lehet, vagy a pontos nevét adjuk meg (úgynevezett *assembly strong name*) vagy a pontos elérési útvonalat. Az alábbi sorban a direktíva szintaxisa található.

```
<#@ assembly name="[szerelvény elérési útvonala vagy neve]" #>
```

T4 import direktíva

Az *import direktíve* (*T4 Import Directive*) funkciója teljesen megegyezik a C# nyelv using nyelvi szerkezetéhez, amivel az adott névterekben lévő típusok nevét oldhatjuk fel. Az alábbi sorban a direktíva szintaxisa látható.

```
<#@ import namespace="[névtér neve]" #>
```

A *namespace* attribútum segítségével adhatjuk meg, hogy mely névtérben található típusok nevét szeretnénk feloldani a sablonunkon belül.

T4 include direktíva

Lehetőségünk nyílik arra is, hogy újrafelhasználható sablonokat készítsünk és ezeket egy másik sablonban újra és újra felhasználhassuk. Az *include direktíva* (*T4 Include Directive*) segítségével meglévő sablonfájlokat importálhatunk az adott fájlunkba. Ezzel a referált fájlt tartalma be fog másolódni a sablonunkba. A következő sorban a direktíva szintaxisa látható.

```
<#@ include file="[fájl neve]"
            [once="[csak egyszer töltődjön be a fájl]"] #>
```

A *file* attribútummal a referált fájl nevét és elérési útvonalát tudjuk megadni, míg a *once* opcionális attribútummal azt, hogy csak egyszer vagy többször töltődjön be a fájl tartalma.

3.5.2. Szöveg blokkok

Talán a legegyszerűbb szintaktikai eleme a *T4*-nek a *szöveg blokkok* rész, ugyanis, az ide beírt szöveg változtatás nélkül kerül bele a sablon által generált kimeneti fájlba. Ezeket a részeket nem kell semmilyen módon megjelölni. Az alábbi példa is ezt mutatja.

```
<#@ output extension=".txt" #>
Helló Text Template Transformation Toolkit (T4)
```

A fenti kódrészletből a T4 egy .txt kiterjesztésű fájlt fog generálni, aminek a tartalma a következő: **`Helló Text Template Transformation Toolkit (T4).`**

3.5.3. Vezérlő blokkok

A vezérlő blokkok segítségével adhatunk dinamizmust a sablonok generálásához, azaz segítségükkel mondhatjuk meg, hogy a sablon egyes részeit hogyan, mikor és hányszor generálja le nekünk. Ezeken a blokkokon belül definiálhatunk új típusokat, változókat és értékelhetünk ki különböző kifejezéseket.

Alapértelmezett vezérlő blokkok

Az *alapértelmezett vezérlő blokkok* (*standard control blocks*) programkódok szakasza, amely a kimeneti fájl egy részét generálják valamilyen algoritmus alapján. Bármilyen vezérlési szerkezetet írhatunk a blokkon belül, kezdve a szekvenciával, az elágazásokon keresztül, egészen a ciklusokig. A vezérlő blokkokat **`<# ... #>`** között definiáljuk.

A vezérlő blokkok közé zárt szöveg blokkok az adott vezérlési szerkezet szemantikája alapján működik. Ez azt jelenti, hogy egy elágazás igaz ágában található szöveg blokk akkor fog megjelenni a kimeneti fájlban, amikor az elágazás feltétele igaz lesz a sablon kiértékelése során. Az alábbi kódrészlet a **`Hello`** szót fogja kigenerálni a kimenetre:

```
<# var isTrue = true; #>
<# if (isTrue){ #> Helló <# }
else { #>  Világ! <# } #>
```

Egy ciklus törzsében definiált szöveg blokk, annyiszor fog megjelenni a kimeneten, ahányszor a ciklus törzse kiértékelésre került. Az alábbi sorban lévő kódrészlet ötször fogja kiírni a kimenetre az *alma* szót:

```
<# for (int i = 0; i < 5; i++) { #> alma <# } #>
```

Fontos megjegyezni, hogy a blokkon belül csak vezérlési szerkezeteket lehet megadni, típusokat (osztályokat, enumerációkat stb.) máshol kell definiálnunk.

Kifejezés-orientált vezérlő blokkok

Vannak olyan helyzetek, ahol vezérlési szerkezet helyett elég lenne csak egy kifejezést kiértékelni. Ilyen esetekben használhatjuk a *kifejezés-orientált vezérlő blokkokat* (*expression control block*) a sablonokon belül.

Szintaxisa hasonlít az alapértelmezett vezérlő blokkokéhoz, azzal a különbséggel, hogy a blokkon belül kifejezést kell írni vezérlési szerkezet helyett: **`<#= ... #>`**. A következő példában a számokat fogjuk kigenerálni egytől tízig:

```
<# for (int i = 0; i < 10; i++) { #>
  <#= i + 1 /* itt történik a kifejezés kiértékelése */ #>
<# } #>
```

Kifejezésként bármit írhatunk, ugyanis a T4 kiértékeli az adott kifejezést, utána pedig meghívja rajta a **`ToString()`** metódust és annak a visszatérési értéke fog a kimeneten megjelenni.

Osztály-orientált vezérlő blokkok

A T4 úgy működik, hogy a háttérben létrehoz minden sablonhoz egy osztályt, ami a **`TextTransformation`** osztályból származik közvetlenül. Ezt a saját osztályt mi is kibővíthetjük további metódusokkal, tulajdonságokkal vagy akár újabb típusokkal is. Ehhez az *osztály-orientált vezérlő blokkokat* (*class feature control block*) kell használnunk.

Ennek a vezérlő blokknak a szintaxisa is hasonlít az eddig bemutatott blokkok szintaxisához, viszont a blokkon belül nem kifejezést, vagy vezérlési szerkezetet kell megadnunk, hanem valamilyen metódust, tulajdonságot vagy típust: **<#+ ... #>**. Ezeket a blokkokat gyakran használjuk kisegítő metódusok deklarálására.

Az alábbi kódrészlet az osztály-orientált vezérlő blokkok használatát hivatott reprezentálni. A **Person** osztállyal reprezentáljuk a személyeket, ami a két információt tárol róluk: a nevüket (**Name** tulajdonság) és az életkorukat (**Age** tulajdonság). A **persons** változóba eltároljuk két személy adatait és az alapértelmezett vezérlő blokk segítségével kigeneráljuk az adatait a fájlba a következő módon: **név (életkor)**:

```
<# var persons = new[] { new Person("Gipsz Jakab", 35),
new Person("Mekk Elek", 24) }; #>
<# foreach (var person in persons) { #>
<#=person.Name#> (<#=person.Age#>)
<# } #>
<#+
    class Person {
        public string  Name { get; set; }
        public int Age { get; set; }
        public Person(string name, int age) {
            Name = name;
            Age = age;
        }
    }
#>
```

További előnye ezeknek a blokkoknak, hogy szövegrészletek generálására is felhasználhatóak. Az előző példát egészítjük ki azzal, hogy megadunk egy **PrintPerson(Person person)** metódust, amit a következőképpen definiálunk:

```
<#+ void PrintPersons(Person[] persons) {
    foreach (var person in persons) { #>
       <#= person.Name #> (<#= person.Age #>)
<#+ }
} #>
```

Ezután már egyszerűen helyettesíthetjük a **`foreach`** ciklusunkat az alábbi sorral: **`<# PrintPersons(persons); #>`**.

3.6. Aspektus-orientált programozás Java-ban

[TODO]

4. Metaprogramozást támogató programozási nyelv tervezése

4.1. A fordítóprogramokról általában

Egy átlagos fordítóprogramok működése négy különálló fázisra bomlik. A forráskód elemzése és fordítása legelőször a lexikális elemző futásával kezdődik. Feladata, hogy a neki átadott szöveget egy reguláris nyelvtan alapján tokenek sorozatára bontsa.

Ha a lexikális elemző befejezte a működését, akkor a fordító átlép a következő fázisba a szintaktikus elemzésbe. A szintaktikus elemző feladat, hogy a neki átadott token sorozatokból és egy környezet független nyelvtan segítségével szintaxis fát építsen. Ebben a fázisban még nincs lehetőség kiszűrni az olyan hibákat, mint pl. a típushibák, esetleg olyan változóra való hivatkozás, amit előzőleg nem definiáltunk stb.

Ha elkészült a szintaxisfa, akkor jöhet a szemantikai ellenőrzése a forráskódnak. Itt a fa alapján megpróbálja felderíteni a fordító az olyan hibákat, amely futásidőben problémákat okozna. Ilyenek lehetnek a nyelv típusrendszere által meg nem engedett műveletek, típusellenőrzés közben elkövetett hibák, esetleg névütközések stb. Egy erős típusrendszerű nyelv esetében, mint pl.: a *Scala* vagy a legtöbb tisztán funkcionális nyelv, aminek statikus típusrendszere van (ilyen a *Haskell* és a *Clean*) sokkal több hibát fel lehet deríteni a szemantikus ellenőrzés során.

Ha a szemantikus ellenőrzés hiba nélkül lefutott, akkor ideje a már meglévő szintaxis fából (ami most már ki van egészítve szemantikus információkkal is) futtatható kódot generálni. Mindig a futtatókörnyezettől függ, hogy milyen kódot kell generálni ebben a fázisban. Olyan natív nyelvek esetében, mint a *C*, *C++*, *Delphi* a fordító assembly kódot generál, majd az fordul le a számítógép által is értelmezhető gépikóddá.

Felügyelt nyelvek esetében, mint pl. a *C#, F#, Java, Scala*, kicsit más a helyzet, ugyanis bináris állományok helyett, bájtkódot generál a fordító. A *C#* programozási nyelvnél egy assembly nyelvekhez nagyon hasonló, *CIL* (*Common Intermediate Language*) kódot generál a fordító, ezt viszont csak a *.NET Framework* virtuális gépe képes megérteni. Mind a *Java*, mind pedig a *.NET Framework* virtuális gépe úgy működik, hogy ezt a bájtkódot, futásidőben értékeli ki és fordítja le a számítógép

processzorának is érthető utasításokra. Ezzel a megoldással egy absztrakt réteget húzunk a tényleges processzor és a kód közé, így a programunk platformfüggetlen lesz.

Szkriptnyelvek esetében (*JavaScript, Ruby, Python*) kicsit máshogy működik a fordítóprogram, mivel a kódgenerálás helyett az utasítások azonnal végrehajtódnak. Ennek hátránya, hogy a fordítási időben észrevehető hibák is csak futási időben derülhetnek ki.

Nem feltétlenül kell azonban alacsonyszintű kódot generálnia a fordítónak. Jó példák tudnak lenni erre a *CoffeeScript, Dart* vagy *TypeScript,* melyek mindegyike *JavaScript* kódot generál, így téve lehetővé a böngészők számára, hogy ezeken a nyelveken írt programokat értelmezni tudják. *Haskell* esetében is van lehetőség arra, hogy *C* nyelvre fordítsa a kódot, így a programozók képesek a *Haskell* nyelven írt függvényeket felhasználni.

4.2. Szintaktikus elemek generálása fordítási időben

Ahhoz, hogy fordítási időben forráskód manipulációkat tudjunk végezni, a 4.1-es szakaszban bemutatott fordítóprogram megvalósítása nem ideális számunkra. A gyakorlatban ezek a fordítóprogramok úgy vannak implementálva, hogy a szintaktikus elemzés közben már részben szemantikus ellenőrzések is végrehajtásra kerülnek. Így azonban, ha a meglévő szintaxisfán valamilyen változtatást hajtunk végre, akkor lehetséges, hogy inkonzisztenssé válik a kódunk, mivel szemantikailag megsértjük a nyelv valamelyik szabályát. Így a transzformáció után újabb szemantikai ellenőrzést kell végrehajtani.

A legjobb megoldás, ha teljesen különválasztjuk a szintaktikus ellenőrzést, a szemantikaitól, és a harmadik fázisba csak akkor fogunk belépni, ha már elkészült a végleges szintaxisfánk.

A szintaxisfa transzformálását a szintaktikai elemzés közben fogjuk elvégezni *makró*k segítségével. Ezek a *makró*k a nyelv részei, nagyon hasonlóak a függvényekhez, attól eltekintve, hogy aktuális paraméterül az absztrakt szintaxisfát kapják és fordítási időben képesek végrehajtódni.

A fordítóprogramunknak képesnek kell lennie fordítási időben a programozási nyelv segítségével definiált *makró*kat értelmezni és végrehajtani. Ehhez az kell, hogy két állapotban kell tudnia futni: *értelmezőként* (*interpreter*) és *kódgenerálóként*.

Az *értelmező állapot* azt jelenti, hogy úgy fog működni, mint egy szkript nyelv, azaz a nyelvi utasításokból nem kódot fog generálni, hanem már fordítási időben végre fogja hajtani azokat. *Értelmező állapotba* csak akkor léphet, ha a *makró*kat kell végrehajtani, minden más esetben kódgenerálóként fog működni, azaz úgy fog viselkedni, mint egy klasszikus fordítóprogram.

4.3. Metaprogramozást támogató eszközök a nyelvben

Idáig többször is esett szó, hogy metaprogramozáshoz *makró*kat fogunk használni és ezeknek a segítségével tudunk majd változtatásokat végrehajtani a kódban fordítási időben.

A *makró*k nagyon hasonlóak a függvényekhez, attól eltekintve, hogy ezek fordítási időben hajtódnak végre, aktuális paraméterként egy szintaxisfát lehet átadni és eredményül is valamilyen szintaxisfát fogunk visszakapni. A nyelvünkben kétféleképpen lehet majd használni a *makró*kat, attól függően, hogy mikre szeretnénk majd használni azokat.

Lehetőségünk lesz arra, hogy explicite meghívjuk őket kódból és mi adjuk át nekik a szintaxisfát, aminek eredményéül az általa generált fa fog beillesztődni a kódba.

Ami sokkal nagyobb szabadságot ad, az az *implicit makró*k használata. Ahelyett, hogy nekünk kellene függvényként hívogatni őket, sokkal célszerűbb lenne, ha *szelektorok* segítségével a fordítóprogram automatikusan adná át a megfelelő részfáit a szintaxisfának és hajtaná végre a *makró*kat. Ezzel a megoldással nem szemeteljük a már meglévő kódbázisunkat, mégis számunkra fontos változtatásokat hajthatunk végre.

A *szelektorok* speciális nyelvi eszközök, amikkel különböző mintákat, sablonokat definiálhatunk arra, hogy mely részfákat szeretnénk kiválasztani és átadni a *makró*k számára. Nagyon hasonlóak a *CSS (Cascading Style Sheets)* nyelv által bevezetett szelektorokhoz, csak itt a *DOM (Document Object Model)* helyett a szintaxisfán fogunk keresni. Szintaxisa nagyon hasonló lesz a *CSS nyelv* szelektoraihoz.

4.4. Metaprogramozás matematikai modellje

A következőkben egy egyszerű matematikai modellt fogunk definiálni a nyelvben bevezetett metaprogramozás használatához és ezzel fogjuk szemléltetni, hogy milyen problémák merülhetnek fel az implementáció közben.

4.4.1. Szintaxisfa definíciója

Legyen $T = (V, E) \in AST$ egy összefüggő, irányított, körmentes gráf, ahol V a csúcsok halmaza, $E \subset V \times V$ az élek halmaza. Ezt a T gráfot *szintaxisfának* fogjuk nevezni. Az AST halmazt *szintaxisfák halmazának* nevezzük.

Jelöljük $\rho(T) \in V$-el a T szintaxisfa *gyökércsúcsát* és $\omega \in AST$-vel az *üres szintaxisfát*.

4.4.2. Jól definiált szintaxisfa

Jól definiált szintaxisfának nevezzük azokat a fákat, amelyek megfelelnek az adott programozási nyelv által definiált szintaktikai szabályoknak. Feltesszük továbbá azt is, hogy a ω üres szintaxisfa jól definiált.

4.4.3. Szintaxisfa részfája

Legyenek $T = (V, E)$ és $T' = (V', E')$ szintaxisfák. Azt mondjuk, hogy T' *részfája* T-nek (jelölés: $T' \subseteq T$), ha $V' \subseteq V$, $E' \subseteq E$.

4.4.4. Szintaxisfa komplementere

Legyenek $T = (V, E)$ és $T' = (V', E')$ szintaxisfák. Ha $T' \subseteq T$, akkor a T'-nek T-re vonatkozó *komplementerén* a $(V, E \backslash E')$ szintaxisfát értjük (jelölése: T_C). *Megjegyzés*: Egy T szintaxisfa komplementere nem feltétlenül jól definiált.

4.4.5. Két szintaxisfa uniója

Legyenek $T = (V, E)$ és $T' = (V', E')$ szintaxisfák. *Két szintaxisfa unióján* a $T \cup T' = (V \cup V', E \cup E')$ szintaxisfát értjük.

4.4.6. Két szintaxisfa metszete

Legyenek $T = (V, E)$ és $T' = (V', E')$ szintaxisfák. *Két szintaxisfa metszetén* a $T \cap T' = (V \cap V', E \cap E')$ szintaxisfát értjük.

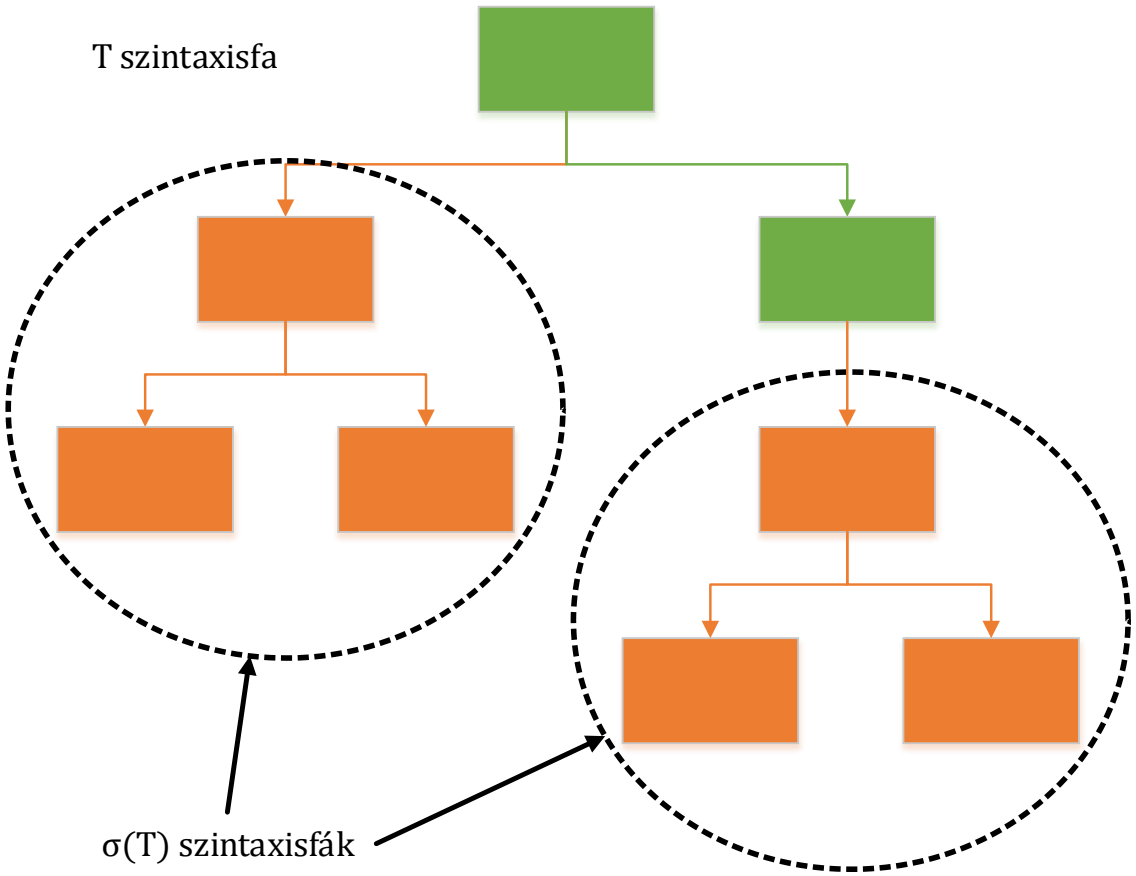
Két szintaxisfa *diszjunkt*, ha $T \cap T' = (\emptyset, \emptyset) = \omega$, azaz a metszetük üres fa.

4.4.7. Szelektor definíciója

Szelektoroknak nevezzük azokat az $\sigma: AST \times 2^{AST}$ leképezéseket, ahol

$$\forall T' \in \sigma(T) \subseteq 2^T : T' \subseteq T.$$

Feltesszük továbbá azt is, hogy a T' jól definiált szintaxisfa. A 1. ábra – $\boldsymbol{\sigma(T)}$ szelekor működése **Error! Reference source not found.**mutatja, hogy hogyan is kell elképzelni a szelektorokat. Az *összes szelektorok halmazát* Σ-val fogjuk jelölni. *Identikus szelektornak* fogjuk nevezni az $\sigma_{id}(T) \coloneqq \{T\}$ szelektort.



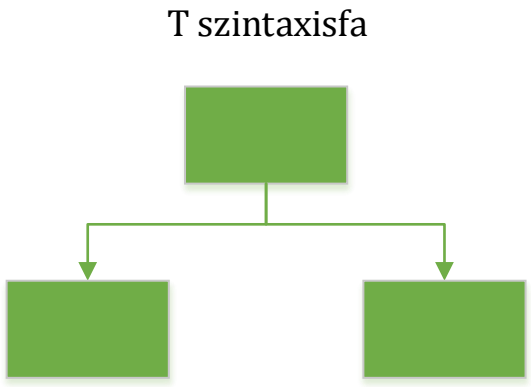
1. ábra – $\boldsymbol{\sigma(T)}$ szelekor működése

4.4.8. Makró definíciója

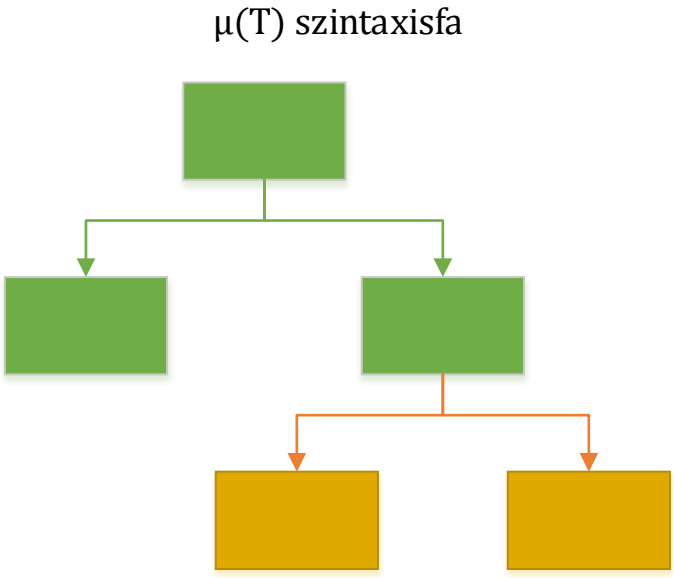
Makróknak nevezzük azokat a μ: AST $\rightarrow$ *AST* leképezéseket, ahol

$$\mu(T) \coloneqq T' \qquad (T, T' \in AST).$$

Feltesszük még továbbá azt is, hogy T és T' jól definiált szintaxisfák. Az *összes makrók halmazát* M-vel fogjuk jelölni. A μ_{id}(T) $\coloneqq$ T makrót *identikus makrónak* fogjuk nevezni. A 2. a) ábra – Az eredeti T szintaxisfa a) és b) ábra mutatja, hogy hogyan is kell elképzelni a makrók működését.



2. a) ábra – Az eredeti T szintaxisfa



2. a) ábra – Az eredeti T szintaxisfa b) ábra – A $\mu(T) = T'$ szintaxisfa

4.4.9. Szintaxisfa transzformációjának definíciója

Legyen $T \in AST$ egy jól definiált szintaxisfa, $\mu \in \mathrm{M}$ egy makró és a hozzá tartozó $\sigma \in \Sigma$ szelektor. Egy $\tau_{\mu,\sigma}: AST \to AST$ leképezést a *szintaxisfa transzformációjának* nevezzük, ahol

$$T' \coloneqq \{\mu(t) \in AST \mid t \in \sigma(T)\}$$

$$\tau_{\mu,\sigma}(T) \coloneqq \left(\bigcap_{t \in \sigma(T)} t_C\right) \cup \left(\bigcup_{t' \in T'} t'\right).$$

Feltesszük továbbá azt is, hogy a $\tau_{\mu,\sigma}(T)$ szintaxisfa is jól definiált.

4.4.10. Metaprogramozás definíciója

Legyen $T \in AST$ egy jól definiált szintaxisfa, $\mu_1, \dots, \mu_n \in M$ makrók, $\sigma_1, \dots \sigma_n \in \Sigma$ makrók és a hozzájuk tartozó $\tau_1, \dots, \tau_n$ szintaxisfa transzformációk, ahol $\tau_i \coloneqq \tau_{\mu_i,\sigma_i}$. *Metaprogramozásnak* hívjuk a T szintaxisfán végrehajtott $\tau_1, \dots, \tau_n$ transzformációk sorozatát:

$$T' = \tau_1\big(\tau_2(\dots \tau_n(T) \dots)\big) = (\tau_1 \circ \dots \circ \tau_n)(T).$$

Könnyű belátni, hogy a transzformációk után eredményül kapott T' szintaxisfa is jól definiált, így a szemantikus ellenőrző már egy szintaktikailag helyes fát fog bemenetként megkapni.

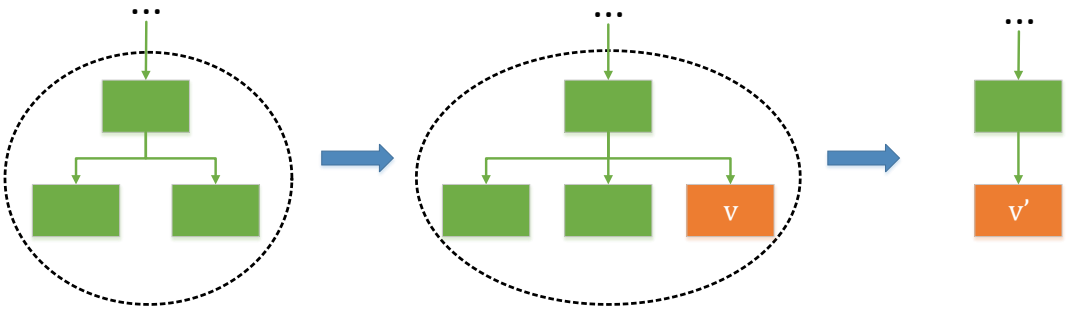
4.4.11. Tétel (szintaxisfa transzformációi nem cserélhetőek fel)

Ha $T \in AST$ egy jól definiált szintaxisfa, akkor létezik olyan $\tau_{\mu,\sigma}$ és ${\tau'}_{\mu',\sigma'}$ transzformáció, illetve a hozzájuk tartozó $\mu, \mu' \in \mathrm{M}$ makrók, és $\sigma, \sigma' \in \Sigma$ szelektorok, amire az igaz, hogy

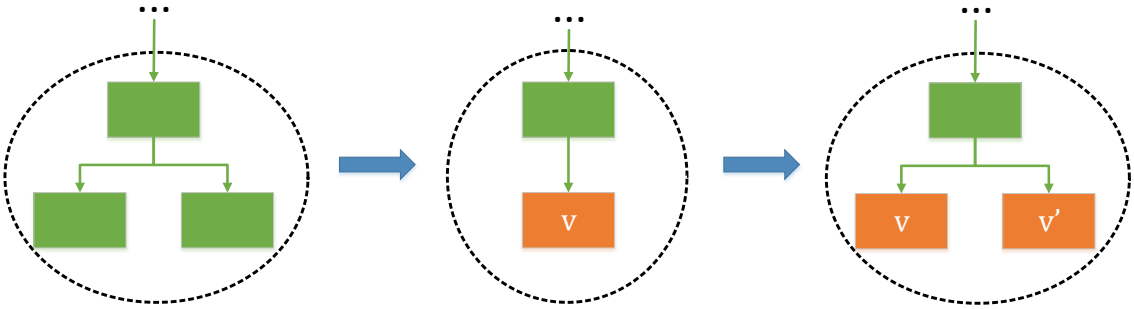
$$\tau_{\mu,\sigma}({\tau'}_{\mu',\sigma'}(T)) \neq {\tau'}_{\mu',\sigma'}(\tau_{\mu,\sigma}(T)).$$

Bizonyítás: Tegyük fel, hogy a $T = (V, E) \in AST$ szintaxisfa jól definiált és létezik a fának egy olyan v csúcsa, ami nem levélcsúcs: $\exists v \in V : E(v) \neq \emptyset$. Ez valójában nem megkötés, hiszen minden programozási nyelv szintaxisfájában van olyan csúcs, aminek vannak gyerekei.

A bizonyítás alapötlete az lesz, hogy ha bármilyen programozási nyelv által jól definiált szintaxisfát veszünk is, annak biztosan lesz legalább egy olyan szintaktikai szabálya, ami úgy van reprezentálva a fában, hogy több gyermekcsúcs kapcsolódik hozzá. A bizonyítás könnyebb megértésében a 3. a) ábra a) és b) ábra nyújt segítséget.



3. a) ábra – a $\tau_{\mu,\sigma}$ leképezés eredménye



3. a) ábra b) ábra – a $\tau'_{\mu',\sigma'}$ leképezés eredménye

Ezen az ötleten elindulva, a σ szelektor segítségével olyan részfákat fogunk keresni, amik a gyökércsúcsának egy vagy több gyermeke van:

$$\sigma(T) \coloneqq \sigma'(T) \coloneqq \{\, T' \subseteq T \mid E\big(\rho(T')\big) \neq \emptyset \,\}.$$

Ha megtaláltuk ezeket a részfákat, akkor kétféleképpen fogunk cselekedni hozzáadunk egy plusz gyermekcsúcsot a részfa gyökércsúcsához, vagy töröljük annak az összes gyermekét:

$$\mu(T) \coloneqq (V \cup \{v\}, E \cup \{\rho(T), v\}) \qquad (v \text{ új csúcs})$$
$$\mu'(T) \coloneqq (\{\rho(T), v\}, \{(\rho(T), v)\}).$$

Ezután már csak a két transzformációs leképezést kell megadnunk:

$$\tau_1(T) \coloneqq \tau_{\mu,\sigma}(T)$$
$$\tau_2(T) \coloneqq \tau_{\mu',\sigma'}(T) = \tau_{\mu',\sigma}(T).$$

Legyen $H = (V_H, E_H)$ szintaxisfa, és tegyük fel róla, hogy $H \in \sigma(T)$. A σ szelektor definíciója alapján $H \in \sigma(H)$.

$H_1 \coloneqq \tau_1(H) = \mu(H) = (V_H \cup \{v\}, E_H \cup \{\rho(H), v\})$ (v új csúcs)

$$\boxed{H' \coloneqq \tau_2\big(\tau_1(H)\big) = \tau_2(H_1) = (\{\rho(H_1), v\}, \{(\rho(H_1), v)\}) = (\{\rho(H), v\}, \{(\rho(H), v)\})}$$

$H_2 \coloneqq \tau_2(H) = \mu'(H) = (\{\rho(H), v\}, \{(\rho(H), v)\})$

$V_{H''} \coloneqq \{\rho(H_1), v, v'\} = \{\rho(H), v, v'\}$ (v' új csúcs)

$E_{H''} \coloneqq \{(\rho(H_1), v), (\rho(H_2), v')\} = \{(\rho(H), v), (\rho(H), v')\}$

$$\boxed{H'' \coloneqq \tau_1\big(\tau_2(H)\big) = \tau_1(H_2) = (V_{H''}, E_{H''}) = (\{\rho(H), v, v'\}, \{(\rho(H), v), (\rho(H), v')\})}.$$

Azt kaptuk, hogy $H' \neq H''$, amiből az következik, hogy $\tau_2\big(\tau_1(H)\big) \neq \tau_1\big(\tau_2(H)\big)$. A szintaxisfa transzformációjának definíciója alapján:

$$H \in \sigma(T) \wedge \tau_2\big(\tau_1(H)\big) \neq \tau_1\big(\tau_2(H)\big) \Longrightarrow \tau_2\big(\tau_1(T)\big) \neq \tau_1\big(\tau_2(T)\big).$$

Ezzel a bizonyítást beláttuk, azaz a szintaxisfa transzformációinak felcseréléseivel, nem garantált, hogy ugyanazt a szintaxisfát kapjuk eredményül. ■

4.5. Szelekciós stratégiák

Az első probléma, amit az implementáció során felmerülhet az az, hogy milyen szelekciós stratégiákat használjunk a szintaxisfa transzformációja során.

Ha megnézzük a *szintaxisfa transzformációjának definícióját* (lásd 4.4.9), akkor láthatjuk, hogy nem tér ki arra, hogy milyen sorrendben végezzük el a szelektor által visszaadott szintaxisfákon a transzformációt. Ezzel önmagában nem is lenne gond, de a μ makró által visszaadott fát „vissza kell csatolni” az eredeti szintaxisfához és az egyáltalán nem mindegy, hogy milyen sorrendben tesszük ezt meg.

4.5.1. Diszjunkt részfák esete

Olyan szelektorokat fogunk vizsgálni, amik egy adott T szintaxisfa alapján páronként diszjunkt részfákkal tér vissza. Kicsit formálisabban a következőről van szó:

Tegyük fel, hogy T az eredeti szintaxisfánk és $\sigma: AST \times 2^{AST}$ egy szelektor, amire az igaz, hogy $\sigma(T) \neq \emptyset$ és $\forall T_i, T_j \in \sigma(T): T_i \cap T_j \neq \omega$. Továbbá legyen $H \coloneqq \sigma(T) = \{T_1, \dots, T_n\}$ halmaz.

[TODO]

4.6. Implicit makrók végrehajtásának sorrendje

A 4.4.11 tétel kimondja, hogy a makrók végrehajtásának sorrendjének felcserélésével teljesen más szintaxisfát kaphatunk eredményül, amely akár a programunk szemantikáját is megváltoztathatja. Ezzel a metaprogramozási eszközzel egy nem-determinisztikus fordítást kaptunk, ami megnehezítheti a programozók munkáját.

Fontos, hogy olyan stratégiákat és szabályokat definiáljunk, amik egyértelművé teszik a transzformációk végrehajtásának sorrendjét. A következő oldalakon különböző lehetséges megoldásokat fogunk tárgyalni a problémára. [TODO]

4.6.1. Makrók végrehajtása definiálásuk sorrendjében

Az egyik legegyszerűbb és leghatékonyabb megoldás az, ha abban a sorrendben futtatjuk le a makrókat, amilyen sorrendben azok definiálva lettek. Így a programozó pontosan tudni fogja a makrók lefutásának sorrendjét. [TODO]

4.7. Makrók által szimulálható programozási paradigmák

4.7.1. Design by Contract

4.7.2. Aspektus-orientált programozás

4.7.3. Saját konstansok definiálása

4.7.4. Generikus programozás

4.7.5. Tesztelés makrókkal (mockolás)

5. Irodalomjegyzék

[1] Eugene Burmako: Philosophy of Scala Macros. St. Loius, http://scalamacros.org/paperstalks/2013-09-19-PhilosophyOfScalaMacros.pdf, 2013. szeptember 19.

[2] Macro paradise plugin, http://docs.scala-lang.org/overviews/macros/paradise.html, 2014. április 19.