# Bevezetés

[TODO]

# Metaprogramozásról általában

## Domain-specific languages (DSL)

[TODO]

# Metaprogramozás a mai programozási nyelvekben

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

## 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).

### 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;  
};

### 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.

### 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.

### Makrók

Metaprogramozás szempontjából a legérdekesebb nyelvi konstrukciója az előfordítónak maga a makrók használata. A makrók úgy viselkednek, mint a függvények, lehetnek nekik formális paraméterei, majd ezeknek a makróknak 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 makrók 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

## 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.

### 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]](#footnote-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 = {  
 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

Azzal, hogy futási időben tudunk új adattagokat hozzáadni az objektumainkhoz, új lehetőségek nyílnak meg a fejlesztők számára a metaprogramozásban.

Ezzel a technikával dinamikusan állíthatunk elő objektumokat valamilyen gyártó (factory) minta alapján. A következő példa is ezt szemlélteti:

var person = function(name, age, job) {  
 // a result változóban fogjuk tárolni azt az objektumot,  
 // amivel később visszatérünk  
 var result = {  
 name: name,  
 age: age  
 };  
 result.toString = function() {  
 return result.name + " (" + result.age + ")";  
 };  
 // nem minden esetben fogjuk ugyanazt   
 // az objektumot visszakapni, ugyanis ha a felhasználó   
 // valamilyen munkakört is megadott a job paraméternek,  
 // akkor az eredmény objektumunkat kiegészítjük   
 // egy új adattaggal  
 if (job) {  
 result.job = {  
 name: job  
 };  
 // nemcsak új adattagot adunk hozzá,   
 // hanem újradefiniáljuk a toString függvényét is  
 result.toString = function() {  
 return result.name + " (" +   
 result.age + ") - " +   
 result.job.name;

}  
 }  
 return result;  
}  
// használatához egyszerűen meghívjuk a person függvényt  
// és az elkészíti   
var p = person("John Doe", 35);  
// p = { name: "John Doe", age: 35, toString = function(){...} }  
console.log(p.toString()); // output: John Doe (35)  
// a p2-nek már megadunk egy munkakört is  
var p2 = person("Sam", 30, "vadász");  
// p2 = { name: "Sam", age: 30, job: { name: "vadász" },  
// toString = function() {...} }   
console.log(p2.toString()); // output: Sam (30) – vadász

Egy nagy előnye az itt bemutatott metaprogramozási technikák, hogy ugyanúgy működnek más dinamikusan típusos szkript nyelvek esetében is (úgymint Python vagy Ruby).

## 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.

### 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.

### 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]](#footnote-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.

A gyakorlatban a generált szintaxisfák a scala.reflect.api.Trees nevű trait[[3]](#footnote-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)  
 }  
}

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.

### 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]](#footnote-4) és Any[[5]](#footnote-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:

// 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  
}

### Implicit makrók

[TODO]

### Sztringek interpolációja

[TODO]

### Quasiquote-ok használata a kódgenerációhoz

[TODO]

### 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]](#footnote-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:

// 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.).

Ha eredetileg nem case class-ként definiáltunk egy osztályt, akkor sincs nagy gond, hiszen ezt kézzel is el tudjuk végezni. Az ilyen objektumokat extractor object-eknek nevezzük a Scala-ban.

#### Case class makró

A következő példában az előző alfejezetben bemutatott technikát fogjuk automatizálni egy makró segítségével. A megoldásunk egy makró annotáció lesz, amivel a klasszikus osztályokból tudunk extractor object-eket generálni.

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)  
 }  
 }  
 }  
}

### Makró csomagok

[TODO]

## 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.

### 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]

## 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.

### 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.

### 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).

### 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); #>.

## Aspektus-orientált programozás Java-ban

[TODO]

# Metaprogramozást támogató programozási nyelv tervezése

## 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.

## 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 makrók segítségével. Ezek a makrók 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 makrókat 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 makrókat kell végrehajtani, minden más esetben kódgenerálóként fog működni, azaz úgy fog viselkedni, mint egy klasszikus fordítóprogram.

## Metaprogramozást támogató eszközök a nyelvben

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

A makrók 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 makrókat, 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 makrók 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 makrókat. 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 makrók 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.

## 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.

### Szintaxisfa definíciója

Legyen egy összefüggő, irányított, körmentes gráf, ahol a csúcsok halmaza, az élek halmaza. Ezt a gráfot szintaxisfának fogjuk nevezni. Az halmazt szintaxisfák halmazának nevezzük.

Jelöljük -el a T szintaxisfa gyökércsúcsát és -vel az üres szintaxisfát.

### 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 üres szintaxisfa jól definiált.

### Szintaxisfa részfája

Legyenek és szintaxisfák. Azt mondjuk, hogy részfája -nek (jelölés: ), ha , . A valódi részfája -nek (jelölés: ), ha és .

### Szintaxisfa komplementere

Legyenek és szintaxisfák. Ha , akkor a -nek -re vonatkozó komplementerén a szintaxisfát értjük (jelölése: ). Megjegyzés: Egy szintaxisfa komplementere nem feltétlenül jól definiált.

### Két szintaxisfa uniója

Legyenek és szintaxisfák. Két szintaxisfa unióján a   
 szintaxisfát értjük.

### Két szintaxisfa metszete

Legyenek és szintaxisfák. Két szintaxisfa metszetén a   
 szintaxisfát értjük.

Két szintaxisfa diszjunkt, ha , azaz a metszetük üres fa.

### Két szintaxisfa különbsége

Legyenek és szintaxisfák. Két szintaxisfa különbségén a   
 szintaxisfát értjük.

### Unió- és metszetképzés tulajdonságai a szintaxisfákon

Könnyen belátható, hogy az unió- és metszetképzés tulajdonságai ugyanúgy érvényesek a szintaxisfákra, mint a halmazokra.

#### Tétel (szintaxisfa idempotenciája)

Ha szintaxisfa, akkor és .

Bizonyítás: A tételt a 4.4.5 és a 4.4.6 definíciók alapján fogjuk belátni:

Mivel a és halmazok, ezért rájuk érvényes az unió- és metszetképzés idempotenciája, így ezzel az állítást beláttuk.

#### Tétel (szintaxisfa unió- és metszetképzésének kommutativitása)

Ha és szintaxisfák, akkor és .

Bizonyítás: A tételt a 4.4.5 és a 4.4.6 definíciók alapján fogjuk belátni:

Mivel a és halmazok, ezért rájuk érvényes az unió- és metszetképzés kommutativitása, így ezzel az állítást beláttuk.

#### Tétel (szintaxisfa unió- és metszetképzésének asszociativitása)

Ha , és szintaxisfák, akkor és .

Bizonyítás: A tételt a 4.4.5 és a 4.4.6 definíciók alapján fogjuk belátni:

Mivel a és halmazok, ezért rájuk érvényes az unió- és metszetképzés asszociativitása, így ezzel az állítást beláttuk.

#### Tétel (szintaxisfa unió- és metszetképzésének disztributivitása)

Ha , és szintaxisfák, akkor és .

Bizonyítás: A tételt a 4.4.5 és a 4.4.6 definíciók alapján fogjuk belátni:

Mivel a és halmazok, ezért rájuk érvényes az unió- és metszetképzés disztributivitása, így ezzel az állítást beláttuk.

### Szelektor definíciója

Szelektoroknak nevezzük azokat az leképezéseket, ahol

Feltesszük továbbá azt is, hogy a jól definiált szintaxisfa. A 1. ábra – szelekor működése **Error! Reference source not found.**mutatja, hogy hogyan is kell elképzelni a szelektorokat. Az összes szelektorok halmazát -val fogjuk jelölni. Identikus szelektornak fogjuk nevezni az szelektort.



1. ábra – szelekor működése

### Makró definíciója

Makróknak nevezzük azokat a leképezéseket, ahol

Feltesszük még továbbá azt is, hogy és jól definiált szintaxisfák. Az összes makrók halmazát -vel fogjuk jelölni. A 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 szintaxisfa

### Szintaxisfa transzformációjának definíciója

Legyen egy jól definiált szintaxisfa, egy makró és a hozzá tartozó szelektor. Egy relációt a szintaxisfa transzformációjának nevezzük, ahol

Feltesszük továbbá azt is, hogy a jól definiált szintaxisfa.

Megjegyzés: Vegyük észre, hogy a szintaxisfa transzformációja nem feltétlenül determinisztikus.

### Metaprogramozás definíciója

Legyen egy jól definiált szintaxisfa, makrók, makrók és a hozzájuk tartozó szintaxisfa transzformációk, ahol . Metaprogramozásnak hívjuk a szintaxisfán végrehajtott transzformációk sorozatát:

Könnyű belátni, hogy a transzformációk után eredményül kapott szintaxisfa is jól definiált, így a szemantikus ellenőrző már egy szintaktikailag helyes fát fog bemenetként megkapni.

### Tétel (szintaxisfa transzformációi nem cserélhetőek fel)

Ha egy jól definiált szintaxisfa, akkor létezik olyan és transzformáció, illetve a hozzájuk tartozó makrók, és szelektorok, amire az igaz, hogy

Bizonyítás: Tegyük fel, hogy a szintaxisfa jól definiált és létezik a fának egy olyan csúcsa, ami nem levélcsúcs: . 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 leképezés eredménye



3. a) ábra b) ábra – a leképezés eredménye

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

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

( új csúcs)

Ezután már csak a két transzformációs leképezést kell megadnunk:

Legyen szintaxisfa, és tegyük fel róla, hogy . A szelektor definíciója alapján .

( új csúcs) új csúcs)át kapjuk eredményül.elcseréléseivel, nem garantált, hogy ugyanazt a szintaxisfát kapjuk eredményül.

( új csúcs)

Azt kaptuk, hogy , amiből az következik, hogy . A szintaxisfa transzformációjának definíciója alapján:

.

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

## 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 4.4.11), 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 makró által visszaadott fát „vissza kell csatolni” az eredeti szintaxisfához és az egyáltalán nem mindegy, hogy milyen sorrendben tesszük ezt meg.

### Diszjunkt részfák esete

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

Tegyük fel, hogy az eredeti szintaxisfánk és egy szelektor, amire az igaz, hogy és . Továbbá legyen halmaz. A 4. ábra – Diszjunkt részfák kiválasztása a szintaxisfában az ilyen esetekre mutat egy példát.



4. ábra – Diszjunkt részfák kiválasztása a szintaxisfában

Könnyen belátható, hogy az ilyen esetekben van a legegyszerűbb dolgunk, hiszen a kiválasztott szintaxisfáink egymástól függetlenek, így az sem számít, hogy milyen sorrendben hajtjuk végre a kiválasztott részfákon a makrót.

#### Megvalósítás a gyakorlatban

Lássuk, hogy a gyakorlatban ilyen esetekben mit is lehet tenni annak érdekében, hogy a fordítónk hatékony és gyors legyen. Mivel a részfák feldolgozásának sorrendje nem számít, ezért az implementációban lehetőségünk van némi optimalizációra.

#### A feldolgozás párhuzamosítása

Az egyik talán leghatékonyabb módszer, hogy párhuzamosítjuk a makró végrehajtását a részfákon ezzel érve el jelentős sebességnövekedést. Viszont ahhoz, hogy ezt meg lehessen valósítani, több feltételnek kell megfelelni.

Az első feltétel az, hogy nem szabad megengedni, hogy a makró a végrehajtásuk alatt olyan információhoz jusson, amit egymás között megosztanak. Azaz csakis lokális változók értékéhez férhessenek hozzá, globálishoz nem. Ha mégis meg szeretnénk engedni, akkor lehetőséget kell biztosítanunk a nyelvünk használóinak ahhoz, hogy az egyes erőforrásokat valamilyen módszerrel tudják zárolni.

Ha a nyelv tervezésénél megengedtük az előbb taglalt feltételt, akkor újabb problémával szembesülhet a felhasználó egyes esetekben. Ilyen lehet az, hogy egy makró lefutása után a fordító állapota megváltozik, azaz a makró olyan külső állapotot változtatott meg, amitől a működése is függ. Ha ettől az állapottól függ a makró végrehajtása, akkor a párhuzamosítás miatt előfordulhat az a probléma, hogy nem lesz determinisztikus a fordítás és így nem mindig ugyanazt a szintaxisfát kapjuk eredményül, ami akár nem biztonságos kódot is eredményezhet.

Mindenképpen tudnunk kell garantálni a felhasználónak, hogy a fordítás mindig determinisztikus legyen. Ha erre nem vagyunk képesek, akkor már maga a generált kód megbízhatatlan lehet, a tesztelés nagyon nehézzé válhat, sőt fordításonként változhat az alkalmazás működése is.

Nem mindegy az sem, hogy milyen módon van reprezentálva az adott nyelv absztrakt szintaxisfája. Ugyanis ha nyilvántartjuk az adott csúcsok szülőcsúcsát is (vagyis inkább a felhasználó lekérdezheti annak az értékét), akkor könnyedén kiléphetünk a szelektor által kiválasztott részfából. Ilyen esetben semmiképp sem lehet párhuzamosítani, ugyanis ilyenkor nem garantált, hogy a szelektor által kiválasztott részfán végezzük-e az adott műveletet.

Továbbá a nyelv használójának tudnia kell azt is, hogy milyen optimalizációkat végez a fordító annak érdekében, hogy redukálja a fordítási időt. Jó megoldás lehet az, hogy a programozóra bízzuk, hogy mely makrók végrehajtását optimalizálhatja a fordító és melyeket nem. Esetleg egy másik lehetőség, hogy maga a fordító dönti el, hogy a párhuzamosítás biztonságos-e vagy sem. Azonban minden ilyen tervezési döntésnél mérlegelni kell azt is, hogy a fordítás sebessége vajon milyen mértékben fog változni.

#### Részfák diszjunktságának eldöntése

Probléma lehet az is, hogy hogyan állapítsuk meg hatékonyan, hogy a kiválasztott részfák egymásnak diszjunktak-e? Legegyszerűbben úgy tehetjük ezt meg, hogy a szintaxisfa bejárása alatt nyilvántartjuk azt is, hogy adott csúcs egy már megtalált részfa csúcsa-e vagy sem. Ha igen és az abból leágazó részfát is kiválasztja a szelektorunk, akkor már tudjuk, hogy nem lehet párhuzamosítani. Az alábbi pszeudokód írja le a fenti algoritmust:

procedure visit(node: Node of AST, isSubTree: Boolean)  
 // ha éppen egy kiválasztott részfán belül vagyunk  
 // és az adott node-ra is illik a kiválasztás feltétel  
 // akkor tájékoztatjuk a fordítót, hogy nem lehet  
 // optimalizálni a makró használatát  
 // Továbbá a selector(node) true-val tér vissza ha   
 // az adott csúcsból (node) leágazó részfa  
 // megfelel a feltételeinek  
 if isSubTree and selector(node) then  
 canBeOptimalized := false  
 end if  
 // bejárjuk a gyerekelemeket is   
 foreach (child in children(node))  
 visit(child, selector(node))  
 end foreach  
end procedure  
// a szintaxisfa gyökerétől kezdjük a keresést  
visit(root(AST), false)

### Egymást tartalmazó részfák esete

Olyan szelektorokat fogunk vizsgálni, amik egy adott szintaxisfa alapján kiválasztott részfák között van legalább két olyan, ami egyik a másiknak a részfája. A gyakorlatban ez azt jelenti, hogy a szelektor olyan szintaktikai elemeket talált, amik egymást tartalmazzák.

Ilyen eset lehet, pl. amikor elágazásokat szeretnénk megkeresni és van olyan elágazásunk, aminek valamelyik ágában van még egy elágazás. Ekkor a szelektor mindkét szintaktikai elem szintaxisfájával visszatér. Kicsit formálisabban a következőről van szó:

Tegyük fel, hogy az eredeti szintaxisfánk és egy szelektor, amire az igaz, hogy és . Mivel a szelektor egy halmazzal tér vissza, így csak olyan esetekre vagyunk kíváncsiak, amik egymásnak a valódi részfái. Az 5. ábra egy ilyen lehetséges esetet szemléltet.



5. ábra – Egymást tartalmazó részfák   
(a piros színnel jelölt az bővebb részfa, míg a sárga színnel jelölt annak a részfája)

A 4.5.1. alfejezetben bemutatott esethez képest itt már bonyolódik a helyzetünk, ugyanis lehetnek olyan makrók, amik megváltoztatják a részfák struktúráját és ezáltal maga a szelektor által kijelölt részfák inkonzisztensek lesznek. Ebből következik, hogy pontosan meg kell tudnunk határozni, hogy milyen sorrendben akarjuk végrehajtani a részfákon végzett műveletet.

Ilyen inkonzisztencia lehet pl. az is, hogy olyan részfákat akarunk kijelölni a szelektor segítségével, ami további részfákat tartalmaz. Ha olyan részfákat tartalmaz, amikre megint érvényes a fenti feltétel és azokat töröljük először, akkor már az elsőnek kiválasztott részfára nem fog fennállni a szelektor által meghatározott feltétel, azaz a kijelölés érvényét vesztette.

Ezért kellene olyan stratégiákat meghatározni, amikkel biztosítani lehet azt, hogy db részfa feldolgozása után is érvényes még a szelektor által meghatározott halmaz.

#### Részfák feldolgozásának stratégiája (definíció)

Legyen jól definiált szintaxisfa és hozzá egy szintaxisfa transzformáció. Az függvényt részfák feldolgozási stratégiájának fogjuk nevezni, ha az igaz rá, hogy

#### Stratégia (legkisebb részfa feldolgozása először)

A stratégia lényege, hogy mindig azokat a részfákat dolgozzuk fel előbb, ami sokkal kisebbek, azaz kevesebb csúcsot tartalmaznak. Könnyű belátni azt, hogy ebben az esetben azok a részfák is hamarabb sorra fognak kerülni, amik egy másik kiválasztott szintaxisfa részfái.

Formalizálva a stratégiát, az függvényt az alábbi módon definiáljuk:

Ezzel az egyszerű módszerrel leginkább a törlésen alapuló makrók használatát tehetjük biztonságossá. Tegyük fel, hogy van egy olyan makrónk, ami akkor törli az adott részfát, amikor az már nem tartalmaz további részfákat egy adott szelektor alapján, azaz

A gyakorlatban a kiválasztott részfákra a makrót szekvenciálisan, azaz egymás után tudjuk végrehajtani és hatékonysági okok miatt könnyen lehet, hogy nem is tároljuk az eredeti szintaxisfát. Ilyenkor a szintaxisfa transzformációja ekvivalens a következő metaprogramozással:

A 4.4.13. tétel alapján pedig már tudjuk, hogy a transzformációk nem cserélhetőek fel egymással. Azaz ha nem határoznánk meg, hogy a legkisebb csúcsszámú részfákat dolgozzuk fel előbb, akkor könnyedén kaphatnánk két eltérő szintaxisfát eredményül, amit a 6. ábra egy példán keresztül szemléltet.



6. ábra – a részfák feldolgozásának sorrendje nem minden esetben felcserélhető  
a felső sorban először a kisebb csúcsszámú fát jelöljük ki, míg az alsó sorban a nagyobbat

Az alább látható pszeudokód erre a stratégiára mutat be egy lehetséges megoldást. Az algoritmus a postfix fabejáráson alapul, azaz először bejárjuk a gyerekelemeket, és ha szükséges elvégezzük a szintaxisfa transzformációját:

procedure postfix\_visit(node: Node of AST)  
 // bejárjuk a gyerekelemeket is   
 foreach (child in children(node))  
 prefix\_visit(child, selector(node))  
 end foreach  
 // ha az adott csúcsból kiinduló részfa  
 // megfelel a szelektornak,   
 // akkor végrehajtuk rajta a makrót  
 if selector(node) then  
 // lecseréljük a régi részfát, az újra  
 replace\_node(node, macro(node))  
 end if  
end procedure  
// a szintaxisfa gyökerétől kezdjük a keresést  
postfix\_visit(root(AST))

#### Stratégia (a legnagyobb részfa feldolgozása előbb)

Az előző stratégiával az a legnagyobb probléma, hogy a makró el tudja rontani a fa konzisztenciáját, azaz pl. ha a tartalmazott részfát törli, akkor lehet, hogy a még feldolgozásra váró részfák már teljesítik a szelektor által definiált követelményeket. Ez megint csak akkor fordulhat elő, ha nem tároljuk az eredeti szintaxisfát.

Ezekben az esetekben az a stratégia válhat be, hogy mindig a legnagyobb részfákat dolgozzuk fel előbb. Formálisan a következőképpen néz ki az függvény:

A feldolgozásnál választhatunk továbbá abból is, hogy globálisan (azaz az összes kiválasztott részfa közül) vagy lokálisan (csak az egymást tartalmazó részfák közül) szeretnénk kiválasztani a legnagyobbat. A kettő között inkább csak implementációbeli különbség van, hiszen a probléma típusú esetekkel van, ezért elég lehet csak lokálisan a legnagyobbat először kiválasztani.

Az alábbi pszeudokód a lokálisan legnagyobb részfát dolgozza fel előbb. Az algoritmus alapelve a prefix módon való fabejárás, azaz ha már biztosan tudjuk, hogy az adott csúcshoz tartozó részfa megfelel a szelektornak, akkor kiválasztjuk azt és transzformáljuk a makró segítségével.

procedure prefix\_visit(node: Node of AST)  
 // ha az adott csúcsból kiinduló részfa  
 // megfelel a szelektornak,   
 // akkor végrehajtuk rajta a makrót  
 if selector(node) then  
 // lecseréljük a régi részfát, az újra  
 replace\_node(node, macro(node))  
 end if  
 // bejárjuk a gyerekelemeket is   
 foreach (child in children(node))  
 prefix\_visit(child, selector(node))  
 end foreach  
end procedure  
// a szintaxisfa gyökerétől kezdjük a keresést  
prefix\_visit(root(AST))

#### Részfák lusta kiértékelése

A szelektor definíciója (lásd 4.4.9) alapján úgy tűnhet, hogy először meghatározzuk az eredeti szintaxisfának azon részfáit, amik kielégítik a megfelelő feltételeket és csak utána alkalmazzuk rá az adott makrót.

Ezzel a mohó kiértékeléssel az lehet a baj, hogy ha nem tároljuk az szintaxisfát a memóriában, hanem a makrót mindig a már transzformált fára alkalmazzuk, akkor előfordulhat olyan helyzet, hogy az előzőleg kiválasztott szintaxisfa már nem lesz a transzformált szintaxisfa része. Erre nyújt megoldást a részfák lusta kiértékelése, aminek segítségével garantálva van, hogy sosem lesz inkonzisztens a részfák kijelölése.

Egyik legnagyobb előnye, hogy az előzőleg bemutatott algoritmusokat nem kell megváltoztatni, hiszen a lusta kiértékelést elég a selector(node) függvényen belül implementálni.

#### Stratégiák összegzése és értékelése

Amint azt látni lehet, bármilyen stratégiát is használunk a feldolgozáskor más és más problémákba ütközhetünk, ezért is nehéz jól működő stratégiát választani, ami automatikusan a legjobb opciót választja ki a fordítás alatt.

Amit tehetünk az az, hogy a felhasználót egyértelműen értesítjük az előforduló problémákról (valamilyen figyelmeztetést mutatunk neki), ami alapján el tudja dönteni, hogy melyik stratégiát kívánja használni, esetleg más definiálja az adott makróját.

Továbbá érdemes még a szelektorok kiértékelését lusta módon végezni akkor, ha nem szeretnénk eltárolni az eredeti szintaxisfát. Ezzel sok problémától kímélhetjük meg magunkat nem beszélve arról, hogy hatékonyságbeli növekedés is várható, hiszen a makrók használata közben akár csökkenhet a kiválasztott részfák száma is.

## Implicit makrók végrehajtásának sorrendje

A 4.4.13 tétel kimondja, hogy a szintaxisfa transzformációk 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. Ez a probléma már a 4.5.2 alfejezetben is előjött, hiszen a gyakorlatban nem mindig lehet a kiválasztott részfákat párhuzamosan feldolgozni.

A következő oldalakon különböző lehetséges megoldásokat fogunk tárgyalni arra vonatkozóan, hogy milyen stratégiák alapján lehet végrehajtani a makrókat.

### Transzformációk végrehajtása definiálásuk sorrendjében

Az egyik legegyértelműbb és leghatékonyabb megoldás az, ha abban a sorrendben futtatjuk le a transzformációkat, amilyen sorrendben azok definiálva lettek. Ezzel a programozó pontosan tudni fogja, hogy hogyan kerülnek végrehajtásra a fordítás során.

Ennek a stratégiának a használata segítségével a fordítási idő nem növekszik, ugyanis a fordítónak nem kell semmilyen plusz szabályrendszer alapján eldönteni, hogy mely transzformációt kell hamarabb végrehajtani.

Ennek a megoldásnak a használata egy fordítási egységen belül hatékony és követhető, de mi történik akkor, ha külső függőségeket kezdünk el használni a forráskódban?

Ilyenkor a felhasználóra kell bíznunk azt, hogy ő milyen sorrendben szeretné, hogy azok lefussanak. Ezt valamilyen nyelvi elem vagy fordítási argumentum formájában tudjuk biztosítani, amivel explicite meg tudja határozni, hogy a fordító milyen sorrendben hajtsa végre a külső függőségekben elérhető szintaxisfa transzformációkat.

### Transzformációk végrehajtása szelektorok specialitásai sorrendjében

Az előző alfejezetben (lásd 4.6.1) a definiálásuk sorrendjében hajtottuk végre a transzformációkat, de egy sokkal kényelmesebb megoldás lehet a nyelv használója számára az, ha a fordító maga állít fel egy sorrendet az alapján, hogy a szelektorok mennyire speciálisak.[[7]](#footnote-7)

Azt mondjuk, hogy a szelektor szigorúan speciálisabb egy másik szelektortól, ha igaz rá, hogy , azaz a bármilyen szintaxisfa esetében kevesebb részfát választ ki, mint a . Gyengén speciálisabbnak mondjuk, ha .

A sorrend meghatározásához a fordítónak ismernie kell, hogy a szelektorok pontosan milyen részfákat jelöltek ki, még a transzformációk végrehajtása előtt, ami azt vonja maga után, hogy le kell mondanunk a szelektorok lusta kiértékeléséről.

Egy másik hátránya, hogy a fordítónak pluszt terhet jelenthet, hiszen a szelektorok mohó kiértékelésével további ellenőrzéseket kell végrehajtania annak érdekében, hogy a transzformációk biztonságban végrehajtódjanak. Ilyen lehet például az, hogy a transzformációk végrehajtása alatt állandóan figyelni kell, hogy a kiválasztott részfák megfelelnek-e továbbra is az adott szelektornak.

A programozó számára is egy pluszt terhet jelenthet, hiszen nem tudhatja pontosan, hogy mikor melyik transzformáció fut le a fordítás alatt. Nem beszélve arról, hogy újabb transzformáció definiálásával akár az egész sorrend felborulhat.

Előnye viszont az, hogy leveszi a terhet a felhasználó válláról, azaz nem kell döntenie arról, hogy milyen sorrendben is kellene definiálni őket. Egy másik előny az is, hogy nem kell foglalkozni a külső fordítási egységek transzformációival, ugyanis azokra is ugyanezek a szabályok vonatkoznak.

### Transzformációk végrehajtása prioritásuk sorrendjében

Megtehetjük azt is, hogy a felhasználóra bízzuk, hogy szerinte milyen sorrendben kellene lefutni az egyes transzformációknak. Ezt a transzformációk közötti prioritások definiálásával tehetjük meg.

Ez a fajta stratégia nagyon hasonlít a 4.6.1-ban bemutatott stratégiához, azaz ugyanúgy teljesen egyértelmű a kód írója számára, hogy mikor melyik transzformáció fog végrehajtódni a fordítás alatt.

Hátránya is van ennek a módszernek, hiszen valamilyen módon el kell tudnia döntenie a fordítónak, hogy különböző fordítási egységekben lévő transzformációkat milyen sorrendben hajtson végre, ha azoknak ugyanaz a prioritása. Meg lehetne tiltani a nyelv használójának, hogy két azonos prioritás nem lehet két fordítási egység között, de ez nehézkes használatot tenne lehetővé. Ehelyett inkább adjuk megint a felhasználó kezébe a döntést és hagyjuk, hogy felüldefiniálhassa a prioritását a transzformációknak.

Ennek a stratégiának is az az előnye, hogy fordító hatékonyan meg tudja határozni, hogy milyen sorrendben hajtsa végre a szintaxisfa transzformációját, ezzel is csökkentve a fordítási időt.

### Transzformációk közötti függőségek definiálása

Ez a stratégia egy speciálisabb módszernek számíthat, hiszen lehetnek olyan esetek, amikor egyes transzformáció végrehajtása függhet egy- vagy több másiktól. Az eddig bemutatott módszerekkel egyszerűbb függőségeket lehetett definiálni, viszont bonyolultabbak esetében már trükköznie kellhet a nyelv használójának.

Ez a stratégia azonban megengedi a felhasználónak, hogy egyes transzformáció végrehajtását más transzformációk végrehajtásától függjön, azaz, csak akkor futhasson le, ha a függőségei már készen vannak.

Előnye, hogy ezzel a bonyolultabb transzformációkat több kisebb, de egyszerűbb transzformációvá lehet dekomponálni, amivel a kód karbantarthatósága nagymértékben javul, és a kisebb transzformációkat sokkal nagyobb eséllyel lehet újra felhasználni.

Hátránya, hogy további terhet jelent a fordító számára, ugyanis a függőségek alapján a memóriában egy gráfot kell építenie, és az alapján kell meghatároznia, hogy milyen sorrendben hajtsa végre a transzformációkat. Továbbá a függőségi gráfban ellenőriznie kell, hogy a felhasználó nem definiált-e véletlenül kört, azaz nincs-e olyan eset, amikor a transzformációk egymástól függnek, és így soha nem tudnak lefutni.

Könnyen előfordulhat az is, hogy a függőségi gráf nem összefüggő, amit a párhuzamosítás során könnyen ki tudunk használni, úgy hogy a nem összefüggő részeket egyszerre próbáljuk meg feldolgozni, ezzel is növelve a fordítás hatékonyságát.

### Végrehajtási stratégiák összegzése

Az előbb felsorolt stratégiáknak, módszereknek mind-mind megvan a maga előnye és hátránya. Egyenként is hatásosak tudnak lenni a gyakorlatban, de igazi erejük akkor mutatkozik meg igazán, amikor felváltva, vegyítve próbáljuk meg őket használni.

A definiálás sorrendjén alapuló módszer (lásd 4.6.1) mellé könnyedén implementálni lehet a prioritáson alapuló stratégiát (lásd 4.6.3). Ezt a gyakorlatban úgy lehet megvalósítani, hogy a felhasználónak nem kell kötelezően prioritásokat definiálnia, csak a kitüntetett transzformációk esetében. Ilyenkor a fordító úgy fog működni, hogy először a prioritással megjelölteket hajtja végre (azoknak a sorrendje alapján), majd utána a definiálásuk sorrendje szerint az összes többit.

Előnye, hogy a felhasználó sokkal kényelmesebben testreszabhatja a sorrendet, és a fordítás ugyanolyan gyors tud maradni.

A definiálás sorrendjén alapuló módszert (lásd 4.6.1) a függőségeken alapuló stratégiával (lásd 4.6.4) is könnyedén tudjuk vegyíteni, hasonló ötlet szerint. Alapértelmezetten a felhasználónak nem kötelező függőséget meghatároznia, ekkor a definiálás sorrendjében történik a végrehajtás. Minden más esetben, a fordító ugyanúgy tesz, mint a második stratégiánál, azaz először a függőségeket futtatja le és csak azután az adott transzformációt.

Előnye, hogy a memóriában kisebb függőségi gráfot kell építeni és ezzel sokat tud gyorsulni maga a fordítás ideje is.

[TODO]

## Makrók végrehajtása transzformált szintaxisfákra

A metaprogramozás definíciója (lásd 4.4.12) azt mondja, hogy a transzformációkat szekvenciálisan hajtsuk végre, és mint egy adatcsatornán keresztül manipuláljuk a szintaxisfát.

Viszont a gyakorlatban definiálnunk kell, hogy a szelektorok az éppen aktuális szintaxisfából mely részeket választhatják ki a transzformációra, megengedjük-e a szelektor számára, hogy olyan részfákat válasszon ki, amelyeket egy előző transzformáció generált bele a fába.

Az egyik lehetséges megoldás lehet, ha a transzformáció által generált részfákat „lezárjuk”, azaz csakis olvashatóvá tesszük, nem engedjük meg a makrók számára, hogy manipulálhassák azokat. Ha ennél is szigorúbbak akarunk lenni, akkor az olvashatóságot is megtilthatjuk, azaz már a szelektoroknak is megtilthatjuk, hogy egyáltalán kijelöljék őket.

Az előzőleg bemutatott megoldásnak vannak azért hátulütői. Tegyük fel, hogy egy olyan transzformáció hajtódott végre először, ami a teljes forráskódot megváltoztatta szintaktikailag, de szemantikailag helyben hagyta, pl. nyomkövetés, naplózás miatt. Így azonban az ezután következő transzformációk közül egy sem fog lefutni, mivel az egész fát „lezártuk”, nincs lehetőségünk módosítani azt.

Mi lenne, ha a felhasználóra bíznánk azt a döntést, hogy az általa definiált transzformációk milyen részfákkal szeretnének dolgozni? Megtehetjük azt, hogy úgy tervezzük a programozási nyelvet, hogy a nyelvi elemeket metainformációkkal egészíthetjük ki (pl. egy attribútummal, annotációval stb.), amik nem befolyásolják a forráskód szemantikáját, de plusz információt nyújthat a transzformáció számára, hogy azt a felhasználó írta-e vagy dinamikusan lett-e generálva. Esetleg maga a fordító is számon tarthatná a szintaxisfában, és így nem kellene manuálisan megjelölni ezeket a szerkezeteket.

## A transzformációk megszorításai és a szintaxisfa reprezentációja

Eddig arról volt szó, hogy milyen stratégiák alapján tudjuk végrehajtani a makrókat, hogy azoknak az eredménye determinisztikus és minél kiszámíthatóbb legyen. Viszont nem beszéltünk még arról, hogy hogyan érdemes reprezentálni az absztrakt szintaxisfát, olyan nyelvekben, ahol van valamilyen lehetőség arra, hogy transzformáljuk őket.

Első dolgunk eldönteni, hogy mennyi információt osztunk meg a feldolgozott kód szintaxisfájára a felhasználóval. Ha keveset, akkor kevesebb felelősséget teszünk a programozóra, de sokkal kevesebb lesz a szabadsága, míg ha többet, akkor könnyen írhat olyan kódot, amit szinte lehetetlen karbantartani.

Szelektálhatjuk továbbá azt is, hogy az eredeti szintaxisfa mely részeit engedjük meg a felhasználónak a szerkesztésre. Ha teljes hozzáférést biztosítunk, akkor az ismeretlen forrásból érkező kódok biztonsági ellenőrzésére sokkal több időt kell fordítani, viszont a felhasználó teljesen szabad kezet kap a transzformációkhoz.

Ha a programozási nyelvünk megengedi, hogy a saját szintaxisával definiáljunk makrókat, akkor arról is tudnunk kell dönteni, hogy a makrókat lehet-e metaprogramozni, azaz megváltoztathatjuk-e azoknak a szintaxisfáját? Tervezési szempontból nem kifizetődő ezt megengedni, hiszen akkor könnyen lehetne követhetetlen transzformációkat definiálni, amivel a kód karbantarthatósága és megértése szinte lehetetlen feladat.

Ha a makrók transzformációját nem, akkor mit engedjünk meg? Vezérlési szerkezetek, esetleg új típusok, vagy akár egész fordítási egységek vezérlését? Itt is érvényes az, hogy ha minél kevesebbet engedünk meg, annál követhetőbb lesz, hogy a transzformációk hogyan működnek és mit generálnak nekünk.

Kellően kifejező szintaxissal és a felhasználó megfelelő informálásával ezeken a feltételeken is lehet enyhíteni és minél többet megengedni.

Milyen fastruktúrát érdemes választani ahhoz a szintaxisfa reprezentációjához? Szeretnénk, hogy minden változtatást az eredeti szintaxisfán végeznénk el (mutable) vagy nem szeretnénk megváltoztatni annak a belső állapotát és minden egyes műveletnél egy másikat építenénk a memóriában (immutable).

Ha az előzőt választjuk, akkor garantálnunk kell párhuzamosított transzformációk esetén, hogy szálbiztos (thread-safe) a rajta végzett művelet, hogy biztonságosan lehessen manipulálni azt. Utóbbi esetben erre nincs szükség, hiszen sosem lesz lehetőség arra, hogy közös erőforrást használjanak, így a szálbiztosság már garantálva van. Másik előnye a reprezentációnak, hogy sokkal gyorsabban végezhetőek el rajta a transzformációs műveletek, hiszen kevesebb adatot kell mozgatni, mint immutable esetben.

Az utóbbi megoldásnál már tárgyaltuk, hogy alapértelmezetten garantálja nekünk a szálbiztosságot, amit az egyik legnagyobb előnye, viszont sokkal több a memóri használat, mivel minden transzformációnál újra le kell másolni az eredeti szintaxisfa egy-egy részét. Ezen persze a kód optimalizációjával segíteni tudunk.

## Makrók által szimulálható programozási paradigmák

Ebben az alfejezetben sorra vesszük, hogy a szintaxisfa transzformációk segítségével milyen programozási paradigmákat és eseteket válthatunk ki.

### Design by Contract

### Aspektus-orientált programozás

### Saját konstansok definiálása

### Generikus programozás

### Tesztelés makrókkal (mockolás)

# Irodalomjegyzék

1. Eugene Burmako: Philosophy of Scala Macros. St. Loius,

<http://scalamacros.org/paperstalks/2013-09-19-PhilosophyOfScalaMacros.pdf>,

2013. szeptember 19.

1. Macro paradise plugin,

<http://docs.scala-lang.org/overviews/macros/paradise.html>,

2014. április 19.

1. Ha az adott függvényünk a globális környezetben (global scope) van definiálva, akkor a this paraméter alapértelmezetten a window objektumra mutat. [↑](#footnote-ref-1)
2. Érthető módon a paraméter értéke általában nem áll rendelkezésre fordítási időben. [↑](#footnote-ref-2)
3. Scala-ban interfészek helyett trait-ek vannak, amik leginkább az absztrakt osztályokhoz hasonlítanak funkcionalitásukat tekintve. [↑](#footnote-ref-3)
4. A Unit típust a típusos -kalkulusból lehet ismerős. Scala-ban ez a típus jelzi, hogy az adott függvénynek nincs visszatérési értéke. [↑](#footnote-ref-4)
5. Az Any típusból származik minden más típus a Scala-ban, azaz megfelel a .NET keretrendszerben ismert System.Object típussal. [↑](#footnote-ref-5)
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. [↑](#footnote-ref-6)
7. Hasonló megoldást használnak a CSS szelektorok esetében a feldolgozást végző motorok. Minél speciálisabban van definiálva egy szelektor, annál erősebbnek számítanak az általa definiált tulajdonságok (azaz azok kerülnek beállításra), mint más általánosabb szabályoknál esetében. [↑](#footnote-ref-7)