# Metaprogramozásról általában

[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;  
};

### 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.

[TODO]

## Metaprogramozás JavaScript nyelven

[TODO]

## Metaprogramozás Scala nyelven

[TODO]

### Függvénymakrók

### Implicit makrók

### Makró annotációk

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

## 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 3.1-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 , 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.

### Szelektor definíciója

Szelektoroknak nevezzük azokat az leképezéseket, ahol

Feltesszük továbbá azt is, hogy a jól definiált szintaxisfa. A 1. ábra 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) és b) ábra mutatja, hogy hogyan is kell elképzelni a makrók működését.



2. a) ábra – Az eredeti T szintaxisfa



2. b) ábra – A szintaxisfa

### Szintaxisfa transzformációjának definíciója

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

Feltesszük továbbá azt is, hogy a szintaxisfa is jól definiált.

### 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) és b) ábra nyújt segítséget.



3. a) ábra – a leképezés eredménye



3. 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 3.4.9), 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 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.

[TODO]

## Implicit makrók végrehajtásának sorrendje

A 3.4.11 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]

### 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]

## Makrók által szimulálható programozási paradigmák

### Design by Contract

### Aspektus-orientált programozás

### Saját konstansok definiálása

### Generikus programozás