EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

PROGRAMOZÁSELMÉLET ÉS SZOFTVERTECHNOLÓGIAI TANSZÉK

Metaprogramozást támogató eszközök tervezése és megvalósítása

Giachetta Roberto
Egyetemi tanársegéd,
Programtervező matematikus

Szabó Tamás
Programtervező informatikus MSc

Budapest, 2014

Tartalomjegyzék

1. Bevezetés

Az utóbbi években az informatika fejlődésével párhuzamosan, a szoftverfejlesztés eszközei is nagy változásokon mentek keresztül. Most már nem elég, ha az alkalmazásaink kellően gyorsak, sokkal fontosabb, hogy biztonságosak, jól felépítettek, karbantarthatóak legyenek, azért, hogy az egyre gyorsabban változó igényeket könnyebb legyen követni. Ennek megfelelően a programozási nyelvek is változás előtt állnak, a cél, hogy minél kifejezőbb, olvashatóbb kódokat lehessen írni.

Az előbb említett indokok miatt, nem is meglepő, hogy előtérbe kerültek a programozási nyelvekben a metaprogramozást támogató nyelvi lehetőségek támogatása, amelyeknek egyik célja a futási idejű tevékenységek fordítási idejű elvégzése, ezen belül is a programhelyesség magasabb szintű biztosítása. A metaprogramozással egy új világ nyílik meg a programozók előtt, ami teljes szabadságot ad a fejlesztéshez, de ezzel együtt rengeteg felelősség is jár vele.

A metaprogramozás nem újkeletű gondolat, ha jobban belegondolunk, már évtizedek óta velünk van, hiszen a magasabb szintű programozási nyelvek fordítóprogramjai is metaprogramoknak minősülnek. Az igazi áttörést a sablonok, reflexió vagy a makrók használata hozta el ebben a témában. A hardverek fejlődése is segített, hiszen segítségükkel sokkal bonyolultabbak lehetnek a fordítóprogramok úgy, hogy a fordítási ideje már nem nő nagyságrendekkel.

Ennek ellenére a metaprogramozás még mindig egy sokat kutatott téma, mert nagyon nehéz olyan megoldásokkal előrukkolni, ami illeszkedne az eddig megszokott fejlesztési módszertanokhoz. Még mindig kérdéses, hogy hogyan lehetne minél könnyebben és biztonságosabban alkalmazni ezeket az eszközöket, hiszen ez lenne a kulcsa a sikerének.

Nem kétséges, hogy a metaprogramozást támogató programozási nyelvek lehetőségei száma szinte végtelen, segítségükkel rengeteg programozási paradigmát meg lehet valósítani, anélkül, hogy a nyelv adottságai alapértelmezetten támogatná azokat, saját DSL nyelveket lehet implementálni, de akár a tesztelésben is támogatást nyújthat.

Diplomamunkám témája részben az, hogy bemutatásra kerüljenek a jelenleg a szoftverfejlesztésben elérhető metaprogramozási eszközök, illetve a matematika módszereinek segítségével elemezni, hogy hogyan is működnek a metaprogramok, illetve azonosítani milyen problémák, akadályok merülhetnek fel a fordítóprogram implementációjánál.

A 2. fejezetben ismertetésre kerülnek olyan, napjainkba használt, programozási nyelvek és technológiák, amik ezeket az eszközöket használják a még hatékonyabb kódgeneráláshoz és szoftverfejlesztéshez.

A 3. fejezet a metaprogramozást egy kicsit formálisabb oldaláról mutatja be az olvasónak. Definiálásra kerülnek olyan új fogalmak, amikkel a későbbiekben a fordítás során felmerülő akadályokat lehet elemezni a matematika eszközeivel. A formális eszközök mellett szó esik arról is, hogy a gyakorlatban egyes funkciókat hogyan lehetne implementálni egy adott fordítóprogramba. A fejezet legvégén pedig egy rövid ismertető található arról, hogy milyen paradigmákat és felhasználói eseteket válthatunk ki a metaprogramozás segítségével.

A 4. fejezetben egy megvalósítást mutatok arra, hogy hogyan is lehetne kényelmesen használni az addig tárgyalt lehetőségeket egy imperatív programozási nyelv esetében.

Dolgozatom célja, hogy az olvasó betekintést nyerjen ebbe az izgalmas világba, megismerje az alapelveket és meggyőzze őt arról, hogy a metaprogramozás megfelelő szabályok betartása mellett mennyire meg tudja könnyíteni a programozó életét a szoftverfejlesztés bármelyik területén.

2. Metaprogramozás napjainkban

Ebben a fejezetben arról lesz szó, hogy mit is értünk napjainkban metaprogramozás címszó alatt, miért érdemes foglalkozni vele, mi lehet a jövő. A témát különböző programozási nyelveken keresztül fogom bemutatni, hogy jelenleg milyen lehetőségek állnak rendelkezésünkre.

2.1. A metaprogramozásról általában

A szoftverfejlesztésben *metaprogramozás* (lásd [20] és [21]) alatt olyan programok írását értjük, amik képesek más programokat készíteni, vagy meglévőket (akár saját magukat) is módosítani fordítási, vagy futási időben. Már a megfogalmazásból is látszik, hogy elég tág fogalom az, hogy mit értünk alatta, ezért is nehéz erről a témáról egy átfogó elemzést készíteni.

Több kategória szerint is elemezni lehet a fogalomkört, de talán a legérdekesebb az, amikor azt vizsgáljuk, hogy mikor történik maga (azaz fordítási vagy futásidőben) az adatmanipuláció. Jellemzően a fordítóprogramok, szkript nyelvek és szöveggeneráló eszközök támogatják a futásidejű metaprogramozást, míg más nyelveknél a fordítási időt részesítik előnyben.

Magát a reflexiót is vehetjük nem szigorú értelemben egy futásidejű eszköznek, hiszen a *JVM*[1] és a *.NET* keretrendszer alatt is támogatva van új programkódok dinamikus létrehozása.

Valójában már a mai fordítókba is vannak olyan funkciók, amik a metaprogramozásra hasonlítanak. Gondoljunk csak azokra a nyelvi szerkezetekre, amik a könnyebb olvashatóságot segíti a nyelvben, de szemantikailag semmivel se nyújtanak többet, azaz más

[1] *JVM: Java Virtual Machine*

nyelvi elemekkel ki lehetne őket váltani. Az ilyeneket szoktuk szintaktikai cukorkáknak[2] nevezni.

Ilyen szintaktikai cukorka például a *C#* nyelvben a **`using`** vezérlési szerkezet[3], ami a fordítás alatt egy **`try ... finally`** blokkra konvertálódik:

```
using (IDisposable disposable = GetDisposable()) {
  // a using szerkezet törzse
}
```

A fent lévő pár soros kódból a fordító egy ehhez hasonló szintaxisfát fog nekünk generálni:

```
try {
  IDisposable disposable = GetDisposable();
  // a using szerkezet törzse ide kerül
}
finally {
  // a using így garantálja azt, hogy minden esetben meghívódjon
  // a IDisposable.Dispose metódus
  if (disposable != null)
    ((IDisposable)disposable).Dispose();
}
```

Egy másik izgalmasabb téma, de ez inkább már programozási nyelvek szempontjából nézve, hogy milyen nyelvi eszközök állnak a felhasználó rendelkezésére metaprogramozás szempontjából.

A tendencia azt mutatja, hogy a programozási nyelvek tervezői keresik azokat a lehetőségeket, hogy hogyan lehetne megkönnyíteni, elegánssá, de egyben biztonságossá is

[2] az angol megfelelőjét használjuk inkább: *syntactic sugar*

[3] A **`using`** vezérlési szerkezet pontos működéséről ezen az oldalon informálódhat az olvasó: *http://msdn.microsoft.com/en-us/library/yh598w02.aspx*

tenni azt, hogy fordítási időben tudjon a felhasználó programkódot generálni. Ez egy nehéz feladat, hiszen meg kell oldani a fordítási időben lévő nyomkövetést, tesztelést, szintaktikailag egyszerűvé kell tenni a kódgenerációt, nem beszélve a sebességről és a kód optimalizációjáról. A problémák megoldásával egy rendkívül erős eszközt kapna kezéhez a programozó, amivel a szoftverfejlesztés sokkal intuitívabb, gyorsabb és hatékonyabb lenne.

2.2. A C/C++ előfordítója

Nem hagyományos értelemben a *C* előfordítóját[4] is nevezhetjük a metaprogramozás egyik eszközének, azzal a különbséggel, hogy közvetlenül a forráskódon végez transzformációkat.

Működésének az alapelve nagyon egyszerűnek tekinthető, hiszen 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 eredményez.

Magát az előfordítót egy különálló nyelvnek is tekinthetjük, ami az eredeti nyelvtől független, mivel még a *C* nyelven írt forráskód értelmezése előtt feldolgozásra kerül minden egyes fordításkor. Több feladata 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).

[4] Az előfordító működéséről a *C* nyelv specifikációjában (lásd 0) olvashat részletesebben.

2.2.1. Az include direktíva

Az **include** direktíva az előfordító leggyakrabban használt utasítása. A fordító megkeresi a programozó által megadott fájlt és annak a tartalmát 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. Azonban 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 *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 az **#ifndef** direktívával, ha nem, akkor a **#define**-al megtesszük azt:

```
#ifndef PERSON_H
#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;
};
```

2.2.2. Konstansok definiálása

Lehetőségünk van egyszerű konstansokat is definiálni a preprocesszor segítségével. Ezek a konstansok nem azonosak a *C* nyelvben lévőkkel, 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 következőkben már csak kódrészleteket fogunk közölni, és megmutatjuk, 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.

2.2.3. Feltételes fordítás

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

2.2.4. Makrók

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

A *makró*k használatával a programozónak rengeteg lehetősége nyílik arra, hogy olyan dolgokat is automatizálni tudjon, amit a nyelv szintaxisával, sokkal bővebben kellene kifejteni. Szintaxisa nagyon hasonlít a 2.2.2-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 a legnagyobbal. A szintaxis hasonló a konstansoknál megismertnél azzal a különbséggel, hogy zárójelben felsoroljuk a makró formális paramétereit is:

```
// így definiálhatjuk a MAX makrót
#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
```

2.3. Template metaprogramozás a C++ nyelvben

A *C++* nyelv egyik nagy újítása volt a sablonok bevezetése, aminek egy érdekes melléktermé lett az úgynevezett *template metaprogramming* (lásd [18] és [19]). A sablonok nagy hatást gyakoroltak a későbbi imperatív nyelvekre (ilyen a *Java, C#, Eiffel* vagy akár a *TypeScript* is), de ott inkább a generikus típusok támogatása lett a mérvadó, ami nem annyira kifejező, mint a *C++*-ban.

2.3.1. Sablonok működése a C++-ban

A sablonok alapötlete az, hogy az osztályainkat, illetve függvényeinket elláthatjuk típusparaméterekkel, ezzel általánosítva azoknak a működését, illetve a típusparaméterek helyett, még fordítási időben kiszámítható típusokkal is paraméterezhetjük őket.

C++-ban a függvényeket és osztályokat is el lehet látni típusparaméterekkel. Mindegyik esetében egy **`template<[típusparaméterek]>`** prefixxel kell megjelölni őket, és utána már a típusparaméter nevével tudunk rájuk hivatkozni. A következő példában egy **`max`** függvényt definiálunk, ami típustól függően visszatér a nagyobbik objektum értékével.

```
// a T jelzi a függvény típusparaméterét
template<typename T>
// ezután már T-ként hivatkozhatunk rájuk
T max(T a, T b) {
  return a > b ? a : b;
}
```

A sablonoknál nem lehet explicit megszorításokat tenni a típusparaméterekre, fordítás közben derül ki (implicite), hogy milyen elvárásoknak kell teljesülniük nekik.

A *Java*-ban, vagy a *C#*-ban látott generikus típusoknál látott megoldás helyett, itt csak használatkor példányosulnak a sablonok és akkor végzi el a megfelelő ellenőrzéseket a fordító. Az előző példából kiindulva a **`max<int>(1, 2)`** függvényhíváskor a következő kód generálódik a háttérben:

```
// egyértelmű esetekben a fordító képes kikövetkeztetni a
// típusparaméter értékét, azaz max(1, 2)-őt is használhattunk
// volna
int max(int a, int b) {
  return a > b ? a : b;
}
```

Az implicit típuskonverzió miatt problémák léphetnek fel a **`max(1, 1.2)`** hívás esetében, ugyanis felmerül a kérdés, hogy mi legyen a **`T`** paraméter értéke: **`int`** vagy **`double`**? Ezt úgy tudjuk elkerülni, ha mindig **`max<double>(1, 1.2)`** formában használjuk, vagy még egy új típusparamétert adunk hozzá:

```
template<typename T, typename S>
T max(T a, S b) {
  return a > b ? a : b;
}
```

Az új típusparaméter hozzáadásával még mindig kétséges, hogy milyen típusú eredménnyel térjen vissza a függvényünk?

Egy újabb megoldás lehet az, hogy egy harmadikat is hozzáadunk ezzel biztosítva, hogy biztosan jó típusú eredményt kapunk vissza. Figyeljük meg, hogy az új R típusparamétert a paraméterlista elejére tettük, csupán azért, hogy használatkor csak ennek az értékét kelljen megadni:

```
template<typename R, typename T, typename S>
R max(T a, S b) {
  return a > b ? a : b;
}
```

Használata nagyon egyszerű, explicite csak a visszatérési érték típusát szükséges megadnunk, azaz csak a **`max<double>(1, 1.2)`**-őt kell meghívni.

Egy másik fontos művelet a sablonokkal, az a sablonok specializációja, azaz megmondhatjuk a fordítónak, hogy egyes típusok esetében nem kell legenerálnia a sablont, hanem használja annak a specializált változatát, amit mi már deklaráltunk.

2.3.2. Faktoriális kiszámítása template metaprogramozással

Egy klasszikus példán keresztül fogom bemutatni a *template metaprogramozását* a *C++*-nak, ugyanis faktoriálist fogunk számolni fordítási időben.

A faktoriálist a funkcionális nyelveknél gyakran használt módszerrel, a rekurzióval fogjuk kiszámolni. Egy **`Factorial`** nevű *template* osztályt fogunk készíteni, aminek egy statikus adattagja a **`value`** lesz, ami fordítási időben a következőt kapja értékül:

```
static const int value = N * Factorial<N - 1>::value;
```

A *template paramétere* az osztálynak nem egy típust fog várni, hanem egy **`int`** konstanst, amire **`N`**-el fogunk hivatkozni.

```
template <int N> struct Factorial {
  static const int value = N * Factorial<N - 1>::value;
};
```

Mint minden rekurziónál, itt is kell lennie egy megállási feltételnél. Ebben az esetben a `0`-nál fogunk megállni, ami egy *specializált template osztály* lesz. Itt a **`value`** adattag értéke `1` lesz:

```
template<> struct Factorial<0> {
  static const int value = 1;
};
```

Használatánál egyszerűen a típusparaméternek meg kell adni, hogy, hogy mely érték faktoriálisát szeretnénk kiszámolni, amit a **`value`** fog értékül megkapni:

```
int factorial = Factorial<4>::value;   // factorial == 24
```

2.4. 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 operációs 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, 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.

2.4.1. JavaScript nyelvi alapjai, érdekességei

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

Függvények a nyelvben

Az egyik legfontosabb és legtöbbet használt lehetőségünk maguk a függvények, amik a nyelvben objektumokként vannak definiálva, ezért bármilyen változónak értékül adhatjuk őket, illetve akár más függvények aktuális paramétereként is használhatjuk őket. A **max** nevű változónak egy függvényt fogunk értékül adni:

```
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:

```
// definiáljunk 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);
```

Érthető módon, 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 mondhatjuk meg, hogy a függvény **this** paramétere éppen melyik objektumra mutasson[5].

```
// 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)
```

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

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, kivételt képeznek ez alól, a vezérlési szerkezetek és a primitív típusok (**number**, **string**, **boolean**, **undefined**). Az objektumok (**Object**) több olyan tulajdonsággal is rendelkeznek, amire a későbbiekben szükségünk lehet.

Az egyik ilyen tulajdonság 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, ahol 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.

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 technikáknak az, hogy ugyanúgy működnek más dinamikusan típusos szkript nyelvek esetében is (úgymint *Python* (lásd [14]) vagy *Ruby* (lásd [15])). További tervezési mintákról lehet olvasni a [12] könyvben, amik ezeket a metaprogramozási technikákat használják fel az implementációhoz.

2.5. Metaprogramozás Scala nyelven

A *Scala*[6] 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. 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, 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 implementációja a nyelvnek *.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.

[6] A *Scala* nyelv részletes specifikációjáról a [4]-ben olvashat bővebben.

2.5.1. Scala makrókról általánosságban

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

Martin sokáig szkeptikus volt a projekt sikerével kapcsolatban, illetve meg kellett győzni őt arról is, hogy a 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 *Scala*-ban a makrók nem a forráskód szövegén, hanem a szintaktikus elemző által generált absztrakt szintaxisfán (AST) végeznek 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.

2.5.2. Függvény makrók

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

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

A függvény makrók nagyon hasonlítanak az egyszerű függvényekre, kivéve a függvénytörzsük, hiszen ott hívjuk meg a **`macro`** kulcsszóval a makrónk implementációját. Használatuk teljesen természetes, ugyanúgy kell meghívni őket, 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 az ahhoz tartozó implementációt, átadva neki az aktuális paraméterek absztrakt szintaxisfáját[7]. 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, csupán 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 fordítóprogram a fordítás alatt gyűjtött össze.

A gyakorlatban a generált szintaxisfák a **scala.reflect.api.Trees** nevű trait[8]-ből származnak.Vegyük például az előző programkódot és nézzük meg, hogy milyen fa generálódik belőle:

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

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

```
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 implementációjának a paraméterei 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))
```

Ezután 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 a kifejezés aktuális értékét. A példából kiindulva a következő jelenik meg a képernyőn:

```
val x = 10
debug(x + 1 > 0)
```

A fenti kódrészlet az alábbi kódot generálja nekünk és a képernyőre a következő íródik ki: **Test.this.x.+(1).>(0) = true**.

```
println("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 kódgenerálást. 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.

2.5.3. Generikus függvény makrók

A nyelv típusrendszere lehetővé teszi számunkra, hogy generikus típusokat is definiálhassunk a *Scala*-ban. Tegyük fel, hogy felmerül az igény arra, hogy a **debug** makrónkat

kifejezések közepén is szeretnénk használni. Így azonban a visszatérési értéke már nem lehet **`Unit`**[9] és **`Any`**[10] 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:

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

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

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

2.5.4. Sztringek interpolációja

Még mielőtt megismerkednénk a kvázi literálokkal (lásd 2.5.5), nézzük meg, hogy hogyan működnek a *sztring interpolációk* (*string interpolation*). Szigorú értelemben véve nem tartoznak a metaprogramozás eszközeihez, viszont alkalmasak lehetnek saját deklaratív nyelvek implementálására a *Scala*-ba.

Az sztringek interpolációjára azért volt szükség, hogy formázott karaktersorozatokat tudjunk használni nagyon kényelmesen. Az egyik legelterjedtebb interpolátor az **s** függvény, amivel a literálon belül tudunk kifejezéseket definiálni.

```
val value = "World"
// $ operátorral tudjuk elérni a külső azonosítókat
println(s"Hello $value!")
```

Nézzük meg közelebbről, hogy mi történik valójában és hogyan dolgozza fel a fordító ezeket a sztring literálokat. A háttérben a fordító automatikusan átalakítja a literálokat **StringContext** objektumokká, majd azon fogja meghívni az **s** függvényt, ami majd aktuális paraméterként megkapja a sztringben lévő speciális tokeneket. *Scala* kóddal leírva a következőről van szó:

```
// amint azt látni lehet a StringContext objektumon belül
// hívja meg az s függvényt a fordító
// s"Hello $value!" ==
new StringContext("Hello ", "!").s(value)
```

A legszebb az egészben, hogy ezek a formázott sztringek erősen típusosak tudnak lenni, azaz a fordító jelez, ha esetleg nem megfelelő típusú értéket adtunk át az interpolátornak.

Saját interpolátor készítése Scala-ban

Most pedig nézzük meg, hogyan készíthetünk magunknak saját sztring interpolátorokat, ugyanis pár egyszerű nyelvi szerkezet kihasználásával, akár ezt is megtehetjük. Tegyük fel, hogy olyan interpolátort szeretnénk készíteni, ami kiírja a konzolra a karaktersorozatban lévő dinamikusan átadott lebegőpontos számokat. Ha nem azt ad meg, akkor fordítási hibát kapunk.

A megvalósítása az osztálynak nagyon egyszerű lesz, egy implicit osztályt fogunk definiálni, aminek lesz egy **`log`** nevezetű függvénye. Az implicit osztály a **`StringContext`** típusú objektumok esetén fog példányosulni és ezzel érjük el, hogy implementálni tudjuk a saját interpolátorunkat.

`LogInterpolation` néven fogjuk definiálni az osztályt, aminek majd az elsődleges konstruktoron keresztül fogja a fordító átadni a **`StringContext`** típusú objektumot:

```
implicit class LogInterpolation(context: StringContext) {
  // a log függvényt úgy definiáljuk, hogy csak lebegőpontos
  // számokat kaphasson meg aktuális paraméteréül
  def log(args: Float*) : StringContext = {
    for (arg <- args)
      println(arg)
    context
  }
}
```

Használata hasonló a formázott karaktersorozathoz, azzal a különbséggel, hogy **`log`**-ot használunk:

```
println(log"${10 + 1} + ${13 + 7}")
```

Fordítás közben a fordító megpróbálja visszakeresni, hogy mely osztályban található a **`log`** függvény. Miután megtalálta és látja, hogy a **`LogInterpolation`** nevű implicit osztályban van definiálva, akkor a **`StringContext`** objektumot becsomagolja egy

`LogInterpolation` példányosításába és meghívja rajta a **`log`** függvényt. Látható, hogy az egész csupán egy szintaktikai könnyítés a programozók számára:

```
new LogInterpolation(new StringContext(" + "))
    .log(10 + 1, 13 + 7)
```

2.5.5. Kvázi literálok használata a kódgenerációhoz

Amikor makrókat használunk elvárás az, hogy valamilyen módon dinamikusan tudjuk generálni szintaxisfát. Ezt sokféleképpen meg tudjuk tenni, használhatjuk a reflexiót (ami sajnos csak futási időben fog végrehajtódni), a beépített típusokat, amik az absztrakt szintaxisfát reprezentálják, vagy a már eddig bemutatott **`reify`** makrót.

Számos lehetőség közül válogathatunk, de nem nehéz belátni azt, hogy a szintaxisfák felépítése manuálisan hosszadalmas és sokszor kínkeserves munka, nem is említve, hogy könnyen olvashatatlanná tudja tenni a forráskódot.

Az előbbi problémán segítséget tud nyújtani a **`reify`** makró, de ennek a megoldásnak a fő gyengesége az, hogy csak és kizárólag típusellenőrzött fákon működik. Belefuthatunk egyes esetekbe, amikor olyan nyelvi elemeket generálunk, amik önmagukban szintaktikailag helytelenek lennének, de egy kontextusba helyezve már értelmet nyernek és azokat a fordító már képes lenne értelmezni. Egy másik gyengeségük az eddig felsorolt technikáknak, hogy körülményesen tudjuk a már meglévő szintaxisfákat elemeire bontani és mintaellenőrzés alá vetni.

Ezeket a hátrányokat küszöböli ki a *kvázi literálok* (*quasiquote*) használata a nyelvben és próbál egyfajta megoldást nyújtani a programozók számára. Nagyon hasonlítanak a *Boo* nyelvben bemutatott nyelvi konstrukcióhoz (lásd 2.6.3), de itt nem különálló szintaktikai elemként, hanem sztring interpolációk segítségével implementálták.

A manuálisan felépített szintaxisfákkal ellentétben, sokkal egyszerűbb a kvázi literálokat olvasni, karbantartani és szerkeszteni. Ugyanúgy tudjuk őket definiálni, mint

az egyszerű sztring literálokat, azzal a különbséggel, hogy egy **`q`** prefixxel kell őket megjelölni:

```
q"val result = callFunction($args)"
```

Ezután a háttérben a következő forráskód generálódik:

```
new StringContext("val result = callFunction(", ")").q(args)
```

Ha a külvilágból szeretnénk valamilyen dinamikus adatot beilleszteni, akkor egyszerűen a **`$név`** szintaktikai elemet kell használnunk az interpolációnál, majd a fordító a külvilágból próbálja meg behelyettesíteni a megfelelő értéket.

Jóformán bármit be tudunk helyettesíteni, amire szükségünk van. Lehetőségünk van azonosítókat, típusneveket megadni, de akár annotációkat, módosítókat, szimbólumokat is, azaz szinte mindent, amivel szintaxisfát tudunk generálni.

```
// mintaillesztéssel kiválogatjuk
// az "a" és "b" változókba az 1 és 2-őt
val q"$a + $b" = q"1 + 2"
```

Ha egyszerre több szintaxisfabeli elemet szeretnénk beilleszteni a kvázi literálba, akkor kötelezően meg kell jelölnünk `..` operátorral a **`$név`** típusú referenciákat. Ekkor a fordító tudni fogja, hogy nemcsak egy elemet adtak meg.

2.5.6. 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*[11] kiterjesztésen keresztül.

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

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.

Készítsünk egy **`Person`** osztályt, aminek két adattagja lesz: a **`name`** a személy nevét, az **`age`** pedig a személy életkorát reprezentálja:

```
case class Person(name: String, age: Int)
```

[11] 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.

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 következő sor azt mutatja, hogy a p objektumból kinyerjük a **name** és az **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:

```
class Person(val name: String, val age: Int)
```

Egy *object* is generálódik az osztályhoz **Person** néven. Az **apply** függvényen keresztül is tudjuk példányosítani az osztályt, míg az **unapply** a mintaillesztésnél hívódik meg:

```
object Person {
    def apply(name: String, age: Int) = new Person(name, age)
    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.), de nekünk elég csak ennyit tudni.

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 klaszszikus osztályokból tudunk *extractor object*-eket generálni.

A makró elkészítéséhez a 2.5.5. alfejezetben bemutatott kvázi literálokat fogjuk használni, ami a kódgenerálásban fog segítséget nyújtani számunkra. Az annotált elemeket az **`annottees`** formális paraméteren keresztül tudjuk majd elérni a makrón belül.

```
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))
```

Eddig csak technikai dolgokat végeztünk el, most viszont definiáljuk az absztrakt szintaxisfát (AST) a *quasiquote* segítségével. A generálás során az eredeti osztályt is újra elkészítjük azzal a különbséggel, hogy az elsődleges konstruktor láthatóságát **`protected`**-re állítjuk:

```
        val tree = q"""
          class $name protected (..$params)
                      extends ..$parents { ..$body }
```

Majd elkészítjük hozzá az *extractor objektumot*:

```
          object $termName {
```

Így mostmár mint *case class*-t lehet példányosítani és a mintaillesztéshez szükséges **`unapply`** függvényt is legeneráljuk:

```
            def apply(..$params) = new $name(..$parameterNames)
            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)
      }
    }
  }
}
```

2.5.7. Makró csomagok

A 2.5.2. alfejezetben látott függvény makrókkal számos felhasználási esetet meg lehet valósítani, de vannak olyan helyzetek, amikor egy feladatot összetettebb módon kell megoldanunk. Ekkor jöhetnek szóba a *makró csomagok* (*macro bundles*), amik a makrókat egy osztályba csomagolják, biztosítva azt, hogy a makró ugyanazt a kontextust osztják meg egymás között. Ezt a nyelvi lehetőséget a *Scala* 2.11-es verziója már támogatja.

Két alkalom van, amikor ezeket a csomagokat érdemes használni. Az első akkor, amikor a bonyolultabbakat szeretnénk valamilyen módon modularizálni, így téve őket sokkal karbantarthatóvá. A második esetben pedig akkor használhatjuk, amikor szeretnénk szétválasztani a makrók implementációját és a kisegítő metódusokat.

A következő példa a [3] hivatalos dokumentációból van és egy remek példát szolgáltat arra, hogy hogyan is tudjuk használni ezt az új nyelvi lehetőséget.

```
// importáljuk a szükséges csomagokat
import scala.reflect.macros.blackbox.Context
```

A következőkben egy **Impl** nevű osztályt fogunk készíteni a makrók implementációjához. Itt láthatjuk, hogy az elsődleges konstruktoron keresztül fogja megkapni az osztály a makróknak szükséges kontextust, amit a **c** objektum reprezentál:

```
class Impl(val c: Context) {
```

Definiáljuk a **mono** makrónkat lehet látni, hogy már az első paramétert el kell hagynunk, ugyanis a kontextust az osztályon keresztül fogja megkapni:

```
  def mono = c.literalUnit
  // definiáljuk a második makrónkat is
  def poly[T: c.WeakTypeTag] =
    c.literal(c.weakTypeOf[T].toString)
}
```

A **Macros** nevű *object*-ből érjük el az implementációban definiált makrókat:

```
object Macros {
  // a mono-val hivatkozunk az implementált mono-ra
  // itt a szintaxis megegyezik
  // a függvény makróknál látottakkal
  def mono = macro Impl.mono
  // és a poly-val az implemenált poly-ra
  def poly[T] = macro Impl.poly[T]
}
```

2.6. Metaprogramozás Boo nyelven

A *Boo* egy objektum-orientált, statikusan típusos, általános célú programozási nyelv *Microsoft .NET* és *Mono* keretrendszerekre. Maga a *Python* nyelv szintaxisa ihlette, 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.

2.6.1. Boo szintaktikus makrók

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

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

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

Az assert makró

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

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

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

```
assert <kifejezés>
```

Ha ehhez az utasításhoz ér a fordító, akkor az alábbi szintaxisfával fog visszatérni:

```
unless (<kifejezés>):
    raise Boo.AssertionFailedException('(<kifejezés)')
```

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

```
assert <kifejezés>, <üzenet>
```

Most pedig 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 a *.NET* keretrendszer ezen funkcióját? 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 annak az 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:

```
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.

2.6.2. Makrók definiálása

Makrókat egyszerűbben is definiálhatunk a nyelvben. Szintaxisuk rendkívül hasonlít a függvényekéhez, azzal a különbséggel, hogy a **`macro`** kulcsszót kell használnunk. Ugyanúgy lehetnek formális paraméterei, és használatának módja is megegyezik a klaszszikus függvényekével, azzal a különbséggel, hogy a makrók már fordítási időben végrehajtódnak. Egy nagyon egyszerű makrót a következőképpen lehet definiálni:

```
macro hello(str as string):
  // quasiquote használata, ezzel a szintaxisfával térünk vissza
  yield [|
    Console.WriteLine("Hello " + $(str) + "!");
  |]
```

Használni ugyanúgy lehet, mint a függvényeket, azaz egyszerűen csak a nevével hivatkozva, meghívjuk azt:

```
hello("World")
```

A fenti makróhívás a következő kód szintaxisfáját generálja le fordítási időben:

```
Console.WriteLine("Hello " + "World" + "!");
```

A konzolon pedig a következő eredmény jelenik meg: **`Hello World!`**

2.6.3. Kvázi literálok használata a nyelvben

A *Boo* objektum-orientált megoldása a makrók használatára egy nagyon elegáns megoldás, viszont felhasználói oldalról sokszor időigényes és bonyolult lehet szintaxisfákat dinamikusan generálni objektumok segítségével. Erre a problémára a nyelv tervezője egy sokkal kézre állóbb megoldást választott, a *kvázi literálok* (*quasiquote*) használatát.

Ez a nyelvi szerkezet nagyon hasonlít a *Scala*-ban bemutatott kvázi literálokhoz (lásd 2.5.5.), csupán a szintaxisukban van különbség. *Boo*-ban a literálokat `[|` … `|]` zárójelek között definiáljuk és a benne lévő értékek dinamikusan legenerálásra kerülnek a fordítás alatt. Tegyük fel, hogy egy olyan makrót akarunk készíteni, ami ellenőrzi egy objektumról, hogy az **`null`** értékű vagy sem:

```
// checkNull makró definiálása
macro checkNull:
  // a parameter nevű változóba eltároljuk
  // az első aktuális paramétert
  parameter = checkNull.Arguments[0]
```

Ezután legeneráljuk a szükséges szintaxisfát a kvázi literál segítségével:

```
  yield [|
    // egy elágazást generálunk mégpedig úgy,
    // hogy a feltételben beleírjuk, hogy mely
    // objektum értékét ellenőrizzük
    if ($(parameter) == null):
      // láthatjuk, hogy az objektum nevét is ki tudtuk olvasni
      message = string.Format("{0} cannot be null!",
                              $(parameter.ToString()))
      raise ArgumentNullException(message,
                                  $(parameter.ToString()))
  |]
name = "Hello World!"
```

Ahogy arról már volt szó, hasonlóan használhatjuk, mintha egy függvényt hívnánk meg:

```
checkNull name
```

A fenti kódrészlethez ez a kódhoz tartozó szintaxisfa fog generálódni:

```
if (name == null):
  message = string.Format("{0} cannot be null!", "name")
  raise ArgumentNullException(message, "name")
```

A **yield** kulcsszó ebben az esetben nagyon hasonlít a *C#*-ban megtalálhatóéhoz, azzal a különbséggel, hogy egy elem értéke helyett egy szintaxisfát fog eredményül adni, amikor ehhez az utasításhoz ér. A literált egy változóba is eltárolhatjuk és így dekomponálhatjuk a nagyobb szintaxisfákat, ami növeli a kód olvashatóságát.

2.6.4. 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. Az eddig bemutatott lehetőségeket fogjuk felhasználni, hogy el tudjuk készíteni.

A megvalósítás rövid és lényegretörő lesz, csupán azt szeretnénk, hogy generáljon nekünk egy már meglévő osztályból egy saját egyke osztályt, aminek az **`GetInstance`** függvénye visszatér ennek a típusnak az egyetlen példányosított objektumával.

```
// egyke makró definiálása
macro singleton:
  // lekérjük az aktuális paramétereket
  args = singleton.Arguments
  // az első paraméter a makró neve lesz
  name = args[0]
  // a második paraméter pedig
  // a generált statikus osztályunk ősosztálya
  baseClass = args[1]
```

A **`yield`** kifejezés használatával visszatérünk a generált osztályunkkal:

```
  yield [|
    // statikus osztály
    static class $(name)($(baseClass)):
      private static _instance as $(baseClass) = $(baseClass)()
      public static def GetInstance() as $(baseClass):
       return $(name)._instance
  |]
```

Használata a következőmódon történik:

```
singleton PersonSingleton, Person
person = PersonSingleton.GetInstance()
person2 = PersonSingleton.GetInstance()
Console.WriteLine(person == person2)  // Kimenet: igaz
```

2.7. 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 *.NET* keretrendszer, és azon belül a *C#* nyelv, további eszközöket is nyújt a metaprogramozáshoz, amiről bővebben a [10] könyvben olvashat részletesebben.

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. 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*.

2.7.1. T4 direktívák

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

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

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

T4 sablon direktíva

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

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

Az egyik legfontosabb attribútumon, a **`language`**-en 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 a publikus (**`public`**) és internál (**`internal`**) választhatunk.

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.

2.7.2. Szöveg blokkok

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

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

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

2.7.3. Vezérlő blokkok

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

Alapértelmezett vezérlő blokkok

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

A vezérlő blokkok közé zárt szöveg blokkok az adott vezérlési szerkezet szemantikája alapján működik. Ez azt jelenti, hogy egy elágazás igaz ágában található szöveg blokk akkor fog megjelenni a kimeneti fájlban, amikor az elágazás feltétele teljesül 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** ősosztályból származik közvetlenül. Ezt 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); #>`**.

2.8. Összefoglalás

Amint az látható a metaprogramozás egyáltalán nem az új kor vívmánya, hanem már több évtizede vannak olyan eszközök, amivel kihasználhatjuk az előnyeit és hátrányait. A tendencia azt mutatja, hogy egyre nagyobb hangsúlyt kapnak a fordítási időben kiszámítható metaprogramok, amivel a futásidőn lehet javítani, illetve a különböző hibákat már el lehet kapni.

Az itt bemutatott nyelveknek és technológiák, mellett még számtalan másik eszköz támogatja a metaprogramozás valamilyen szinten.

Érdemes megemlíteni a *Microsoft* által már évek óta fejlett *Roslyn*[12] projektét, aminek a célja, hogy a *C#* és *Visual Basic .NET* fordítóit teljesen újraírták felügyelt környezetben és a *Roslyn* segítségével különböző szolgáltatásokat adjanak a fordításhoz. Segítségével, mint külső kiterjesztésként, újabb szintaktikai és szemantikai elemeket lehet majd implementálni mindkét programozási nyelvhez.

Ha kicsit kilépünk az imperatív nyelvek világából, és a funkcionális paradigma felé vesszük az utunkat, akkor a *Rascal*[13]-al jó közelebbről megismerkedni. A *Rascal* programozási nyelvet pontosan a metaprogramozáshoz készítették, amiben nagyon könnyű nyelvtanokat definiálni, illetve szintaxisfákat bejárni és feldolgozni.

[12] A projekt weboldala: *https://roslyn.codeplex.com/*
[13] A nyelv weboldala: *http://www.rascal-mpl.org/*

3. Metaprogramozást támogató programozási nyelv tervezése

Ebben a fejezetben részletesebben meg fogjuk nézni, hogy hogyan is működnek a fordítóprogramok és a fordítás mely részeinél lehet implementálni a metaprogramozás eszközeit.

Továbbá ismertetésre kerül egy matematikai modell, ami definiálja mit is jelentenek a makrók, szelektorok, szintaxisfa transzformációk és ezek alapján fogjuk megnézni, hogy az implementáció során milyen problémák merülhetnek fel.

Legvégén pedig szó esik majd arról, hogy a gyakorlatban milyen feladatokat lehet megoldani a szintaxisfa transzformációk segítségével.

3.1. A fordítóprogramokról általában

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

Ha a lexikális elemző befejezte a működését, akkor a fordító átlép a következő fázisba a szintaktikus elemzésbe. A szintaktikus elemző feladat, hogy a neki átadott token sorozatokból és egy környezetfüggetlen nyelvtan segítségével szintaxis fát építsen. Ebben a fázisban még nincs lehetőség kiszűrni az olyan hibákat, mint pl. a típusok helytelen használata, 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. 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*) kód kerül generálásra, 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.

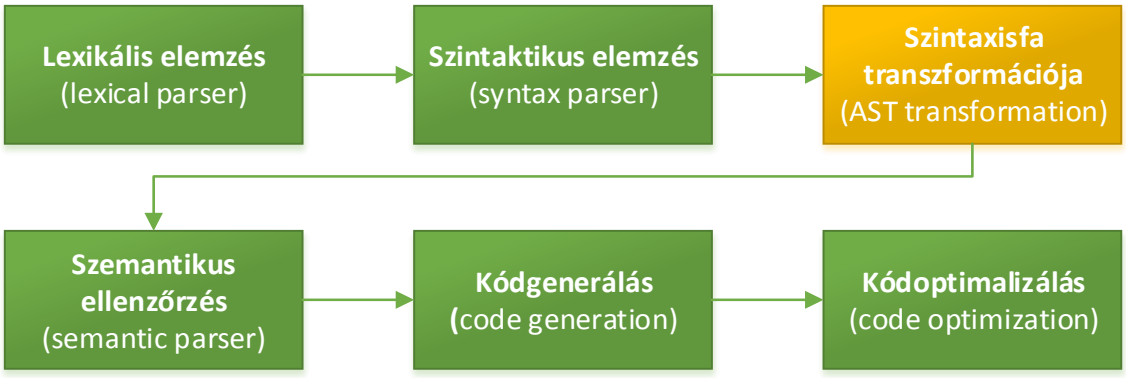
Nem feltétlenül kell azonban alacsonyszintű kódot generálnia a fordítónak. Jó példák tudnak lenni erre a *CoffeeScript, Dart* vagy *TypeScript*, melyek mindegyike *JavaScript* kódot generál, így téve lehetővé a böngészők számára, hogy ezeken a nyelveken írt programokat értelmezni tudják. *Haskell* esetében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

3.2. 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, ami miatt 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 után fogjuk elvégezni *makró*k segítségével, ahogy azt az 1. ábra is mutatja. Ezek a *makró*k a nyelv részei, nagyon hasonlóak a függvényekhez, attól eltekintve, hogy aktuális paraméterül az absztrakt szintaxisfát kapják és fordítási időben képesek végrehajtódni.



1. ábra – A fordítás folyamata kiegészítve a szintaxisfa transzformációjával

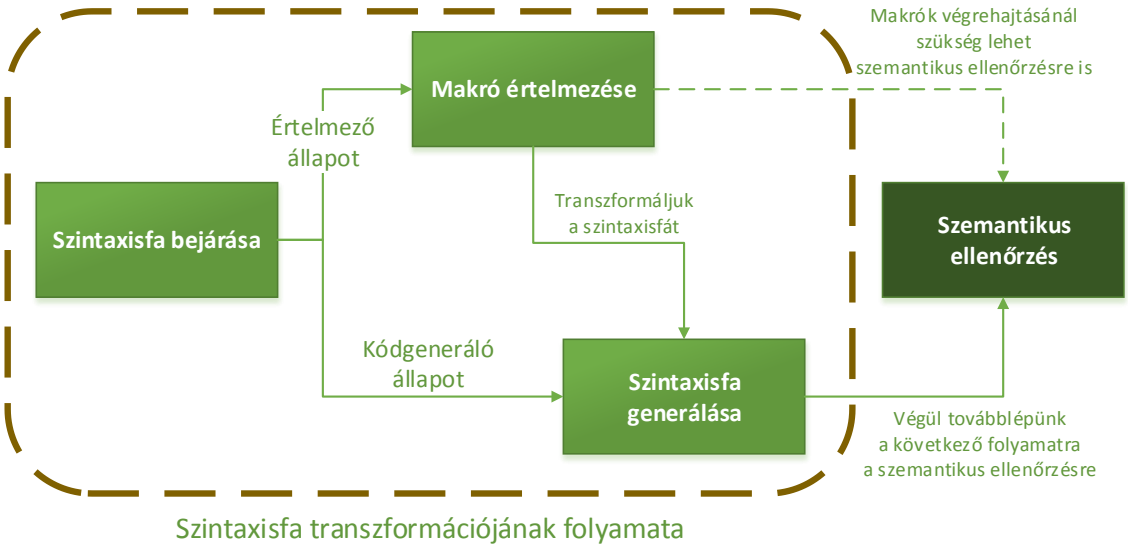
A fordítóprogramunknak képesnek kell lennie fordítási időben a programozási nyelv segítségével definiált *makró*kat értelmezni és végrehajtani. Ehhez az szükséges, hogy két állapotban kell tudnia futnia: *értelmezőként* (*interpreter*) és *kódgenerálóként*.

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

Az itt látható 2. ábra az eddig tárgyalt folyamatot mutatja be. Itt feltesszük azt, hogy a szintaktikai elemző a lexikális elemekből már elkészített egy szintaxisfát és már ezt kapja meg a szintaxisfa transzformációt végző komponens. Ezután újra bejárjuk a fát, úgy hogy ha makróhoz ér a bejárás, akkor értelmező állapotba lép a fordító, míg minden más esetben úgy generáljuk a szintaxisfát, hogy azt már a szemantikus elemző is fel tudja dolgozni.

Meglepőnek tűnhet, de a makrók értelmezése és a szemantikus elemző között is látható egy kapcsolat. Erre akkor lehet szükség, amikor a makró még nincs futtatható állapotban, azaz őt is fel kell tudni dolgozni.

Végül, miután végrehajtódtak a transzformációk, elkészült a végleges szintaxisfa, átadhatjuk a vezérlést a szemantikus elemzőnek.



2. ábra – Szintaxisfa transzformációjának részletesebb folyamata

Természetesen ez egy elméleti folyamat, a gyakorlatban egyes folyamatokat össze lehetne vonni, esetleg másokat redukálni, hogy a fordítás sokkal hatékonyabb és gyorsabb lehessen, de elméleti szempontból így lehet elképzelni a szintaktikus elemek generálását fordítási időben.

3.3. Metaprogramozást támogató eszközök a nyelvben

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

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

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

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

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

3.4. Metaprogramozás matematikai modellje

A következőkben egy egyszerű matematikai modellt fogunk definiálni a nyelvben bevezetett metaprogramozás használatához és ezzel fogjuk szemléltetni, hogy milyen problémák merülhetnek fel az implementáció közben. Továbbiakban a metaprogramozás alatt a szintaxisfa transzformációk sorozatát fogjuk érteni és ebből a szemszögből fogjuk körüljárni a témát.

3.4.1. Szintaxisfa definíciója

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

A csúcsok gyerek elemei között definiálhatunk valamiféle sorrendiséget is, hiszen a legtöbbször fontos, hogy milyen sorrendben dolgozzuk fel őket. A sorrendiséget meghatározó függvényt a következőképpen fogjuk definiálni:

$$\mathrm{O}(v) \in \mathbb{N} \text{ és } \forall v_1, v_2 \in V\colon \mathrm{O}(v_1) \neq \mathrm{O}(v_2).$$

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

3.4.2. Jól definiált szintaxisfa

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

3.4.3. Szintaxisfa részfája

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

3.4.4. Szintaxisfa komplementere

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

3.4.5. Két szintaxisfa uniója

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

3.4.6. Két szintaxisfa metszete

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

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

3.4.7. Két szintaxisfa különbsége

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

3.4.8. 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 $T = (V, E)$ szintaxisfa, akkor $T \cup T = T$ és $T \cap T = T$.

Bizonyítás: A tételt a 3.4.5 és a 3.4.6 definíciók alapján fogjuk belátni:

$$\mathrm{T} \cup \mathrm{T} = (\mathrm{V} \cup \mathrm{V}, \mathrm{E} \cup \mathrm{E}) = (\mathrm{V}, \mathrm{E}) = \mathrm{T}$$
$$\mathrm{T} \cap \mathrm{T} = (\mathrm{V} \cap \mathrm{V}, \mathrm{E} \cap \mathrm{E}) = (\mathrm{V}, \mathrm{E}) = \mathrm{T}.$$

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

Tétel (szintaxisfa unió- és metszetképzésének kommutativitása)

Ha $T = (V, E)$ és $T' = (V', E')$ szintaxisfák, akkor $T \cup T' = T' \cup T$ és $T \cap T' = T' \cap T$.

Bizonyítás: A tételt a 3.4.5 és a 3.4.6 definíciók alapján fogjuk belátni:

$$\mathrm{T} \cup \mathrm{T}' = (\mathrm{V} \cup \mathrm{V}', \mathrm{E} \cup \mathrm{E}') = (\mathrm{V}' \cup \mathrm{V}, \mathrm{E}' \cup \mathrm{E}) = \mathrm{T}' \cup \mathrm{T}$$
$$\mathrm{T} \cap \mathrm{T}' = (\mathrm{V} \cap \mathrm{V}', \mathrm{E} \cap \mathrm{E}') = (\mathrm{V}' \cap \mathrm{V}, \mathrm{E}' \cap \mathrm{E}) = \mathrm{T}' \cap \mathrm{T}.$$

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

Tétel (szintaxisfa unió- és metszetképzésének asszociativitása)

Ha $T = (V, E)$, $T' = (V', E')$ és $T'' = (V'', E'')$ szintaxisfák, akkor $(T \cup T') \cup T'' = T \cup (T' \cup T'')$ és $(T \cap T') \cap T'' = T \cap (T' \cap T'')$.

Bizonyítás: A tételt a 3.4.5 és a 3.4.6 definíciók alapján fogjuk belátni:

$$(\mathrm{T} \cup \mathrm{T}') \cup \mathrm{T}'' = \big((\mathrm{V} \cup \mathrm{V}') \cup \mathrm{V}'', (\mathrm{E} \cup \mathrm{E}') \cup \mathrm{E}''\big) =$$
$$= \big(\mathrm{V} \cup (\mathrm{V}' \cup \mathrm{V}''), \mathrm{E} \cup (\mathrm{E}' \cup \mathrm{E}'')\big) = \mathrm{T} \cup (\mathrm{T}' \cup \mathrm{T}'')$$

$$(\mathrm{T} \cap \mathrm{T}') \cap \mathrm{T}'' = \big((\mathrm{V} \cap \mathrm{V}') \cap \mathrm{V}'', (\mathrm{E} \cap \mathrm{E}') \cap \mathrm{E}''\big) =$$
$$= \big(\mathrm{V} \cap (\mathrm{V}' \cap \mathrm{V}''), \mathrm{E} \cap (\mathrm{E}' \cap \mathrm{E}'')\big) = \mathrm{T} \cap (\mathrm{T}' \cap \mathrm{T}'').$$

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

Tétel (szintaxisfa unió- és metszetképzésének disztributivitása)

Ha $T = (V, E), T' = (V', E')$ és $T'' = (V'', E'')$ szintaxisfák, akkor $(T \cup T') \cap T'' = (T \cap T'') \cup (T' \cap T'')$ és $(T \cap T') \cup T'' = (T \cup T'') \cap (T' \cup T'')$.

Bizonyítás: A tételt a 3.4.5 és a 3.4.6 definíciók alapján fogjuk belátni:

$$(T \cup T') \cap T'' = \big((\mathrm{V} \cup \mathrm{V}') \cap \mathrm{V}'', (\mathrm{E} \cup \mathrm{E}') \cap \mathrm{E}''\big) =$$
$$= \big((V \cap V'') \cup (\mathrm{V}' \cap \mathrm{V}''), (E \cap E'') \cup (\mathrm{E}' \cap \mathrm{E}'')\big) = (T \cap T'') \cup (T' \cap T'')$$

$$(T \cap T') \cup T'' = \big((\mathrm{V} \cap \mathrm{V}') \cup \mathrm{V}'', (\mathrm{E} \cap \mathrm{E}') \cup \mathrm{E}''\big) =$$
$$= \big((V \cup V'') \cap (\mathrm{V}' \cup \mathrm{V}''), (E \cup E'') \cap (\mathrm{E}' \cup \mathrm{E}'')\big) = (T \cup T'') \cap (T' \cup T'')$$

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

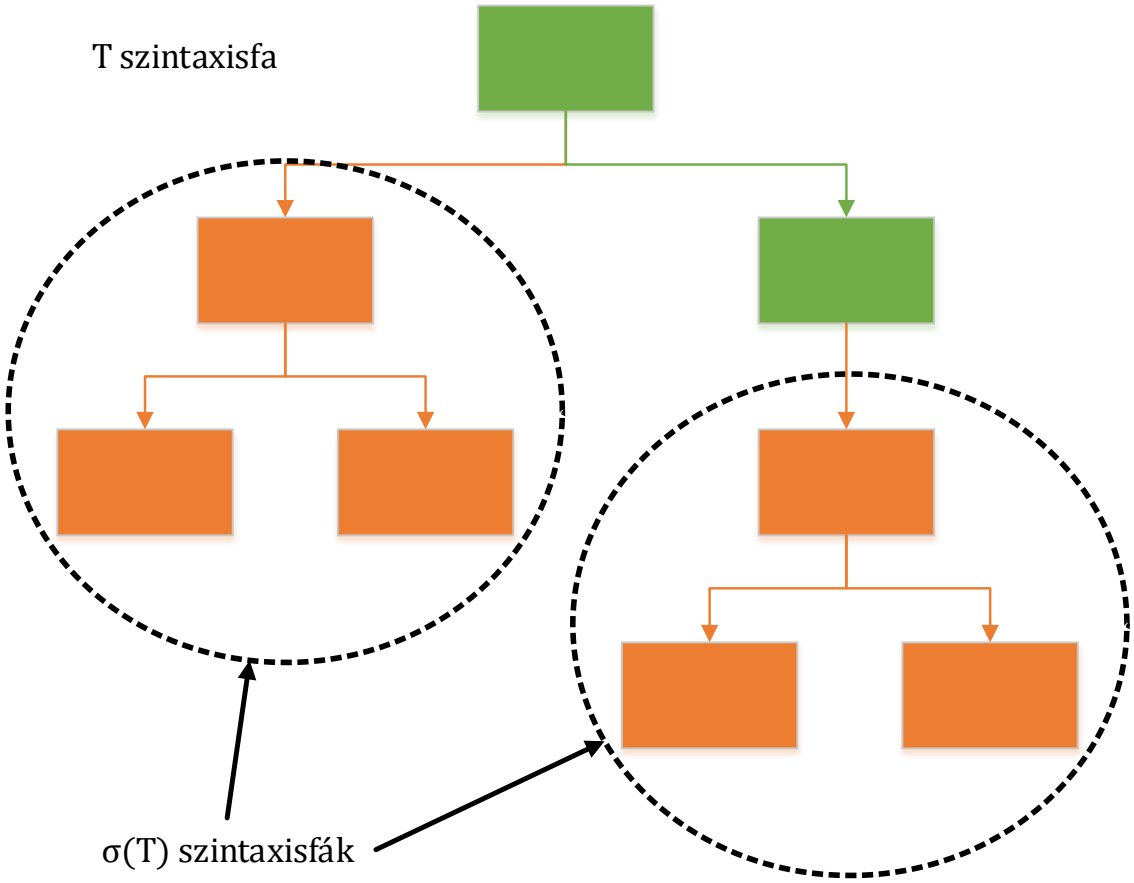
3.4.9. Szelektor definíciója

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

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

Feltesszük továbbá azt is, hogy a T' jól definiált szintaxisfa. Az 3. ábra mutatja, hogy hogyan is kell elképzelni a szelektorokat. Ebben az esetben a σ szelektor kijelöli a narancssárga színű csúcsokat a szintaxisfában.

Az *összes szelektorok halmazát* Σ-val fogjuk jelölni. *Identikus szelektornak* fogjuk nevezni az $\sigma_{id}(T) \coloneqq \{T\}$ szelektort.



3. ábra – $\sigma(T)$ szelekor működése

3.4.10. Makró definíciója

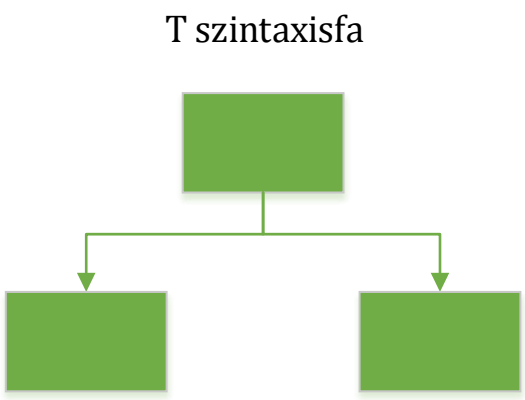
Makróknak nevezzük azokat a $\mu: AST \rightarrow AST$ leképezéseket, ahol

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

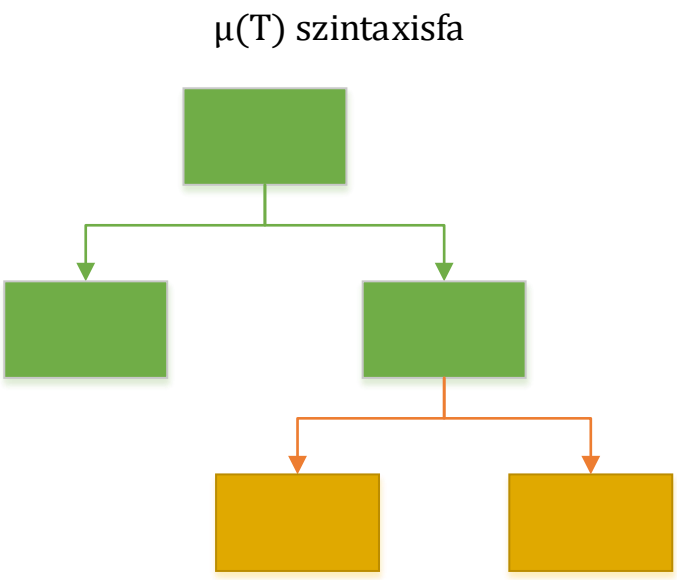
Feltesszük még továbbá azt is, hogy T és T' jól definiált szintaxisfák. Az *összes makrók halmazát* M-vel fogjuk jelölni. A $\mu_{id}(T) \coloneqq T$ makrót *identikus makrónak* fogjuk nevezni.

A 4. a) és b) ábra mutatja, hogy hogyan is lehetne elképzelni a makrókat. Definíció alapján látható, hogy valójában arról van szó, hogy a makró bemenetként kap valamilyen

szintaxisfát és azon végez műveleteket. A megfogalmazás alapján bármilyen manipulációt elvégezhetünk, amint azt látni lehet az 4. ábrán is, ahol a fa jobb oldali ágához hozzácsatoltunk két további elemet.



4. a) ábra – Az eredeti T szintaxisfa



4. b) ábra – A $\mu(T) = T'$ szintaxisfa

3.4.11. Szintaxisfa transzformációjának definíciója

Legyen $T = (V, E) \in AST$ egy jól definiált szintaxisfa, $\mu \in$ M egy makró és a hozzá tartozó $\sigma \in \Sigma$ szelektor. Egy $\tau_{\mu,\sigma}: AST \times AST$ relációt a *szintaxisfa transzformációjának* nevezzük, ahol

$$\bar{T} \coloneqq \left\{ \left(V', E' \cup \left(n, \rho(t) \right) \right) \middle| \, t \in \sigma(T) \wedge t = (V', E') \wedge \left(n, \rho(t) \right) \in E \right\}$$

$$\bar{\bar{T}} \coloneqq \left\{ \left(V', E' \cup \left(n, \rho\left(\mu(t) \right) \right) \right) \middle| \, t \in \sigma(T) \wedge \mu(t) = (V', E') \wedge \left(n, \rho(t) \right) \in E \right\}$$

$$\tau_{\mu,\sigma}(T) \coloneqq (T \setminus \bar{T}) \cup \left(\bigcup_{t \in \bar{\bar{T}}} t \right).$$

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

Elsőre bonyolultnak tűnhet a transzformáció definíciója, de valójában csak technikai szinten van megfogalmazva. Az alapötlet az, hogy lecsatoljuk az eredeti szintaxisfáról a σ által kiválasztott részfákat, ami a $(T \setminus \bar{T})$ rész a definícióban. Ezután pedig hozzácsatoljuk a redukált fához a transzformált részfákat a $(\bigcup_{t \in \bar{\bar{T}}} t)$ művelettel.

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

3.4.12. Metaprogramozás definíciója

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

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

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

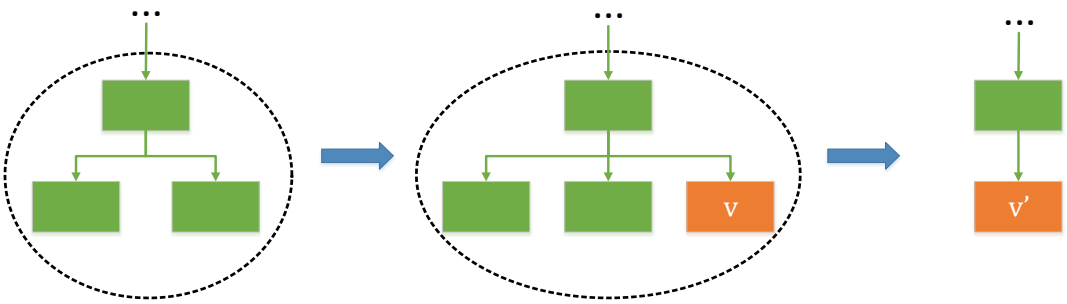
3.4.13. Tétel (szintaxisfa transzformációi nem cserélhetőek fel)

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

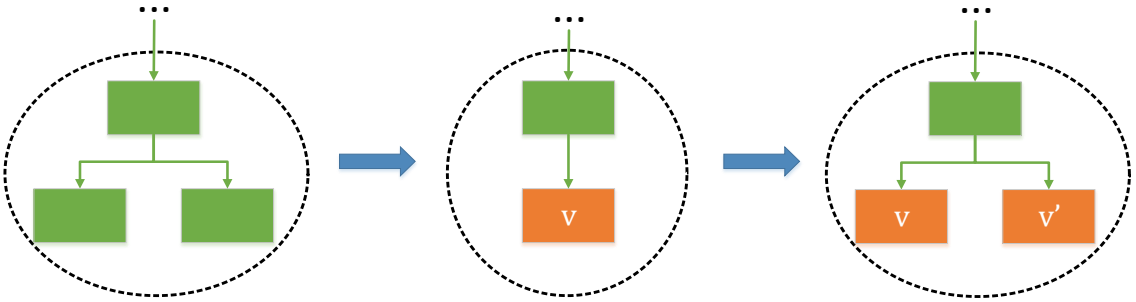
$$\tau_{\mu,\sigma}({\tau'}_{\mu',\sigma'}(T)) \neq {\tau'}_{\mu',\sigma'}(\tau_{\mu,\sigma}(T)).$$

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

A bizonyítás alapötlete az lesz, hogy ha bármilyen programozási nyelv által jól definiált szintaxisfát veszünk is, annak biztosan lesz legalább egy olyan szintaktikai szabálya, ami úgy van reprezentálva a fában, hogy több gyermekcsúcs kapcsolódik hozzá. A bizonyítás könnyebb megértésében a 5. a) és b) ábra nyújt segítséget.



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



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

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

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

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

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

Ezután már csak a két transzformációs leképezést kell megadnunk:

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

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

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

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

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

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

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

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

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

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

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

3.5. Egyszerű imperatív nyelv definiálása

Ebben a fejezetben EBNF[14] segítségével egy metaprogramozásra alkalmas imperatív nyelvet fogunk definiálni formálisan. A nyelv célja, hogy demonstrálja a szintaxisfa transzformációja közben fellépő lehetséges hibákat.

Olyan nyelvi elemeket fogunk meghatározni, ami minden imperatív nyelvben megtalálható (szekvencia, elágazás, ciklus stb.). A következőképpen definiáljuk a nyelvünket:

⟨fordítási egység⟩ ≔ $\{\{$⟨utasítás⟩$\}_1 |$⟨függvény⟩$\}_0$

⟨utasítás⟩ ≔ ⟨üres utasítás⟩|⟨értékadás⟩|⟨függvényhívás⟩|⟨elágazás⟩|⟨ciklus⟩

⟨üres utasítás⟩ ≔ **skip**

⟨elágazás⟩ ≔ **if** (⟨kifejezés⟩) **then** $\{$⟨utasítás⟩$\}_1$ **endif**

⟨ciklus⟩ ≔ **loop** (⟨kifejezés⟩) **do** $\{$⟨utasítás⟩$\}_1$ **endloop**

⟨értékadás⟩ ≔ ⟨bal érték⟩ **≔** ⟨jobb érték⟩

⟨függvényhívás⟩ ≔ ⟨azonosító⟩(⟨aktuális paraméterek⟩)

⟨aktuális paraméterek⟩ ≔ ⟨kifejezés⟩$\{$**,** ⟨kifejezés⟩$\}_0$

[14] (extended) Backus-Naur forma (lásd [17]), amit 2-es típusú Chomsky-féle nyelvek formális definiálásra szoktak használni

⟨kifejezés⟩ ≔ ⟨konstans⟩|⟨azonosító⟩|⟨unáris kifejezés⟩|⟨bináris kifejezés⟩

⟨unáris kifejezés⟩ ≔ (**not** | + | −) ⟨kifejezés⟩

$$\langle \text{bináris kifejezés} \rangle \coloneqq \langle \text{kifejezés} \rangle \begin{pmatrix} \langle \text{numerikus op.} \rangle \\ \langle \text{relációs op.} \rangle \\ \langle \text{logikai op.} \rangle \end{pmatrix} \langle \text{kifejezés} \rangle$$

⟨relációs op.⟩ ≔ < | > | ≤ | ≥ | = | ≠

⟨numerikus op.⟩ ≔ + | − | ∗ | **div**

⟨logikai op.⟩ ≔ **and** | **or**

⟨konstans⟩ ≔ ⟨szám⟩|⟨szöveg⟩

⟨szám⟩ ≔ $\{0|\dots|9\}_1$

⟨szöveg⟩ ≔ "$\{a|\dots|z\}_0$"

⟨azonosító⟩ ≔ $\{a|\dots|z\}_1$

A nyelvben nem lesz lehetőség arra, hogy makrókat és szelektorokat definiáljunk, így csak a függvények szintaktikai szabályait kell megadnunk és kiegészítjük még az ⟨utasítás⟩ szabályt egy új opcióval:

⟨függvény⟩ ≔ **function** ⟨azonosító⟩(⟨formális paraméterek⟩)
$\{\langle \text{utasítás} \rangle\}_1$
endfunction

⟨formális paraméterek⟩ ≔ ⟨azonosító⟩ $\{,\langle \text{azonosító} \rangle\}_0$

⟨visszatérés⟩ ≔ **return** ⟨kifejezés⟩

⟨utasítás⟩ ≔ ⟨üres utasítás⟩|⟨értékadás⟩|⟨függvényhívás⟩
|⟨elágazás⟩|⟨ciklus⟩|⟨visszatérés⟩

Megjegyzés: Az operátorok precedenciája és asszociativitása megegyezik a matematikában használatos műveletekével, továbbá a nyelv szemantikája az imperatív nyelvekével felel meg, de számunkra most nem releváns a formális szemantika a metaprogramozás szempontjából, ugyanis a szintaxisfákat transzformáljuk.

3.5.1. Egyszerű imperatív nyelv jól definiált szintaxisfája

Legyen $T_{imp} = (V, E) \in AST$ szintaxisfa. Azt mondjuk, hogy T_{imp} az *egyszerű imperatív nyelv alapján szintaxisfa*, ha

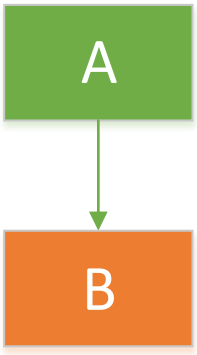
$$(v : \text{címke}) \in V, \text{ahol címke} \in (T \cup N)$$
$$T - \text{terminális ábécé}$$
$$N - \text{nyelvtani jelek ábécéje.}$$

Feltesszük azt is, hogy a fa levélcsúcsainak a címkéje csak a terminális ábécéből kerülhetnek ki, és a fa gyökerének a címkéje a ⟨fordítási egység⟩.

$$\text{levél}(T_{imp}) = \{v \in V \mid \nexists v' \in V : (v, v') \in E\}$$
$$\forall (v : \text{címke}) \in \text{levél}(T_{imp}) : \text{címke} \in T$$
$$\forall (v : \text{címke}) \in V \setminus \text{levél}(T_{imp}) : \text{címke} \in N$$
$$(v : \langle fordítási\ egység \rangle) = \rho(T_{imp}).$$

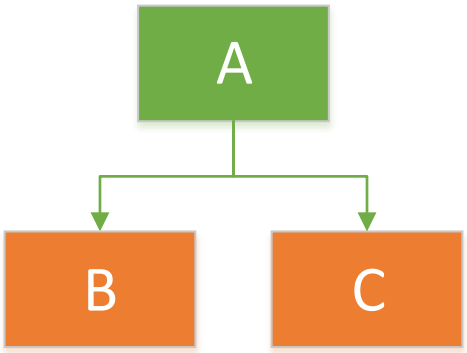
Az egyszerű imperatív nyelv alapján felépített szintaxisfát *jól definiáltnak* fogjuk nevezni, ha a következő feltételek igazak rá:

- $\langle A \rangle ::= \langle B \rangle$, ahol $(A, B \in N)$ szabály esetén a $\forall (v : \langle A \rangle) \in V : \big((v : \langle A \rangle), (v' : \langle B \rangle)\big) \in E$ feltétel igaz. A 6. ábra mutatja, hogy hogyan is néz ki a memóriában a szintaxisfa.



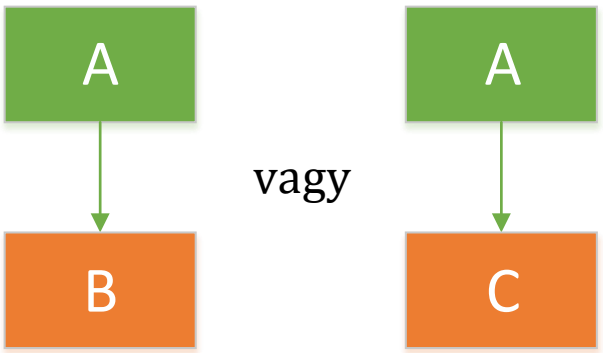
6. ábra – $\langle A \rangle ::= \langle B \rangle$ nyelvtani szabály esete

- $\langle A \rangle \coloneqq \langle B \rangle \langle C \rangle$, ahol $(A, B, C \in N)$ szabály esetén a $\forall (v : \langle A \rangle) \in V : \big((v : \langle A \rangle), (v' : \langle B \rangle)\big) \in E \wedge \big((v : \langle A \rangle), (v' : \langle C \rangle)\big) \in E$ feltétel igaz. Az 7. ábra mutatja, hogy hogyan is néz ki a memóriában a szintaxisfa.



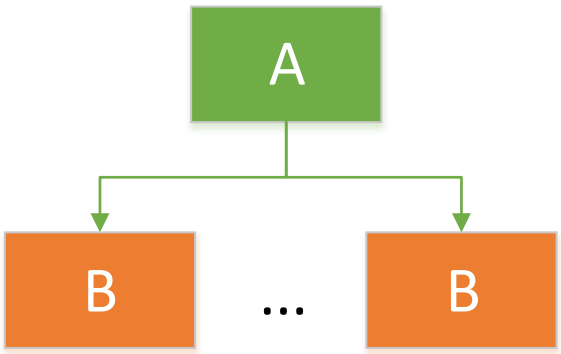
7. ábra – $\langle A \rangle \coloneqq \langle B \rangle \langle C \rangle$ nyelvtani szabály esete

- $\langle A \rangle \coloneqq \langle B \rangle | \langle C \rangle$, ahol $(A, B, C \in N)$ szabály esetén a $\forall (v : \langle A \rangle) \in V : \big((v : \langle A \rangle), (v' : \langle B \rangle)\big) \in E \vee \big((v : \langle A \rangle), (v' : \langle C \rangle)\big) \in E$ feltétel igaz. A 8. ábra mutatja, hogy hogyan is néz ki a memóriában a szintaxisfa.



8. ábra – $\langle A \rangle \coloneqq \langle B \rangle | \langle C \rangle$ nyelvtani szabály esete

- $\langle A \rangle \coloneqq \{\langle B \rangle\}_m^n$, ahol $(A, B \in N)$ szabály esetén a $\forall (v : \langle A \rangle) \in V : k \in \mathbb{N} \wedge m \leq k \leq n \wedge \forall i \in \{1, \dots, k\} : \big((v : \langle A \rangle), (v_i : \langle B \rangle)\big) \in E$ feltétel igaz. A 9. ábra mutatja, hogy hogyan is néz ki a memóriában a szintaxisfa.



9. ábra – $\langle A \rangle ::= \{\langle B \rangle\}_m^n$ nyelvtani szabály esete

3.6. Szelekciós stratégiák

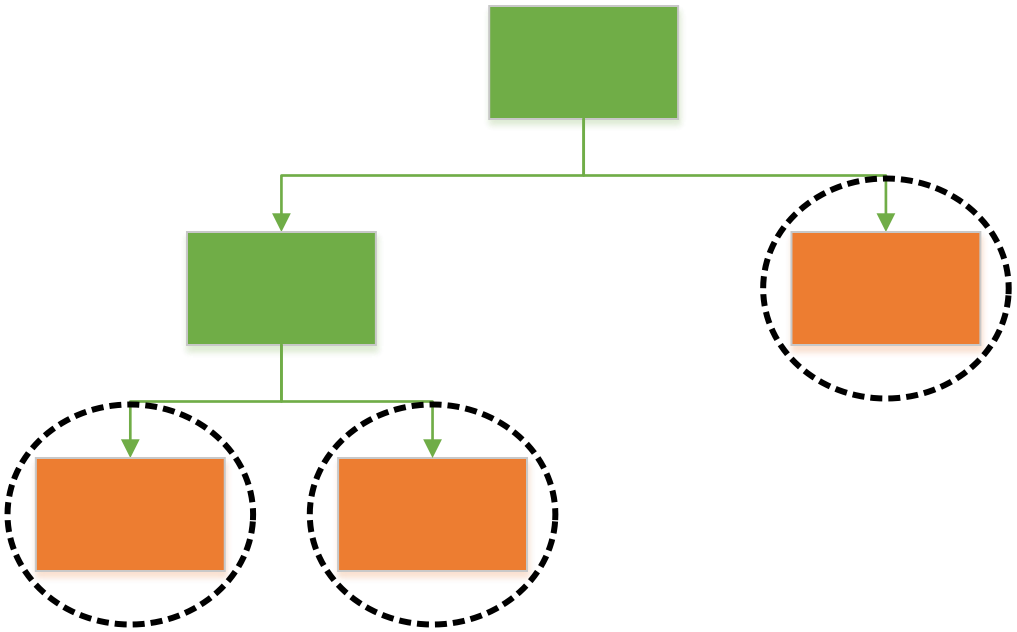
Az első probléma, amit az implementáció során felmerülhet az az, hogy milyen szelekciós stratégiákat használjunk a szintaxisfa transzformációja során.

Ha megnézzük a *szintaxisfa transzformációjának definícióját* (lásd 3.4.11), 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 μ makró által visszaadott fát „vissza kell csatolni” az eredeti szintaxisfához és így egyáltalán nem mindegy, hogy milyen sorrendben tesszük ezt meg.

3.6.1. Diszjunkt részfák esete

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

Tegyük fel, hogy T az eredeti szintaxisfánk és $\sigma: AST \to 2^{AST}$ egy szelektor, amire az igaz, hogy $\sigma(T) \neq \emptyset$ és $\forall T_i, T_j \in \sigma(T): T_i \cap T_j = \omega$. Továbbá legyen $H ::= \sigma(T) = \{T_1, \dots, T_n\}$ halmaz. A 10. ábra az ilyen esetekre mutat egy példát.



10. 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.

Gyakorlati példa a diszjunkt részfák esetére

A gyakorlatban a szintaxisfa levélcsúcsainak kiválasztásánál fordulhat elő, pl. amikor a konstansokkal vagy azonosítókkal szeretnénk valamilyen műveletet végezni.

Tegyük fel, hogy a $\sigma(T) = \{(v : \langle\text{szöveg}\rangle) \in V\}$, azaz a sztringliterálokat választja ki a szelektorunk és lesz egy $\mu \in \mathrm{M}$ makrónk, ami a kiválasztott literálokat kicseréli egy `log` nevű függvény hívásával, ami annyit fog tenni, hogy naplózza azok értékét és visszatér a neki átadott aktuális paraméter értékével. Az alábbi egyszerű imperatív nyelven írt kódot így fogja a $\tau_{\mu,\sigma}$ transzformálni:

```
print("Program futásának kezdete")
i := 0
loop (i < 10) do
  print(toString(i) + " sor: ")
  i := i + 1
end loop
print("Program futásának vége.")
```

A σ szelektor a kódban aláhúzással jelölt szintaxisfabeli elemeket jelöli ki. Lássuk, hogy mi történik velük, ha végrehajtjuk rajtuk a μ makrót:

$\mu($`"Program futásának kezdete"`$) =$ `log("Program futásának kezdete")`

$\mu($`" sor: "`$) =$ `log(" sor: ")`

$\mu($`"Program futásának vége."`$) =$ `log("Program futásának vége.")`

A $\tau_{\mu,\sigma}$ transzformációt végrehajtva a teljes szintaxisfán, az alábbi eredményt kapjuk[15]:

```
print(log("Program futásának kezdete"))
i := 0
loop (i < 10) do
  print(toString(i) + log(" sor: "))
  i := i + 1
end loop
print(log("Program futásának vége."))
```

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.

[15] Az aláhúzott kódrészletek a μ makró által megváltoztatott szintaxisfabeli elemeket reprezentálják.

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 férhetnek hozzá, globálishoz nem. Ha mégis meg szeretnénk engedni, akkor lehetőséget kell biztosítanunk a nyelvünk használóinak ahhoz, hogy az egyes erőforrásokat valamilyen módszerrel tudják zárolni ezeket a közös erőforrásokat.

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 ezt nem tudjuk megtenni, 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 σ szelektor által kiválasztott részfából. Ilyen esetben semmiképp sem lehet párhuzamosítani, ugyanis nem garantált, hogy a σ 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 gyerek elemeket is
   foreach (child in children(node))
      visit(child, selector(node))
   end foreach
end procedure
// a szintaxisfa gyökerétől kezdjük a keresést
visit(root(AST), false)
```

3.6.2. Egymást tartalmazó részfák esete

Olyan szelektorokat fogunk vizsgálni, amik egy adott T 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.

Gyakorlati példa az egymást tartalmazó részfák esetére

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. Definiáljuk a következőképpen a $\sigma \in \Sigma$ szelektorunkat:

$$\sigma(T) = \{(v : \langle\text{elágazás}\rangle) \in V\}.$$

Szeretnénk futtatás közben tudni, hogy a programunk mikor lépett be egy elágazásba, és mikor lépett ki belőle. Ezt úgy fogjuk megtenni, hogy minden elágazás törzsében meghívunk egy **`log`** nevű függvényt valamilyen üzenettel. A $\mu \in \mathrm{M}$ makró ezt az esetet fogalmazza meg a következőképpen:

μ`(if ([kifejezés]) then [törzs] end if)` $:=$
$:=$ `if ([kifejezés]) then`
`log("Belépés az elágazásba")`
`[törzs]`
`log("Kilépés az elágazásból")`
`end if`
`[kifejezés]` $\subset T \wedge \rho($`[kifejezés]`$) = (v : \langle$kifejezés$\rangle) \in V$
`[törzs]` $\subset T \wedge \rho($`[törzs]`$) = (v : \langle$utasítás$\rangle) \in V.$

A következő példa azt szemlélteti, hogy hogyan is működik a $\mu \in \mathrm{M}$ makrónk a gyakorlatban.

```
if (age > 0 and age < 18) then
  if (age <= 14) then
    if (age < 7) then
      result := "kisgyerek"
    end if
    if (age >= 7) then
      result := "gyerek"
    end if
  end if
  result := "tinédzser"
end if
```

A $\sigma \in \Sigma$ szelektor a példában az elágazásokat választja ki számunkra, majd azokat kapja meg a $\mu \in \mathrm{M}$ makró. A következő példa azt mutatja, hogy mi történik az egyik kiválasztott elágazással:

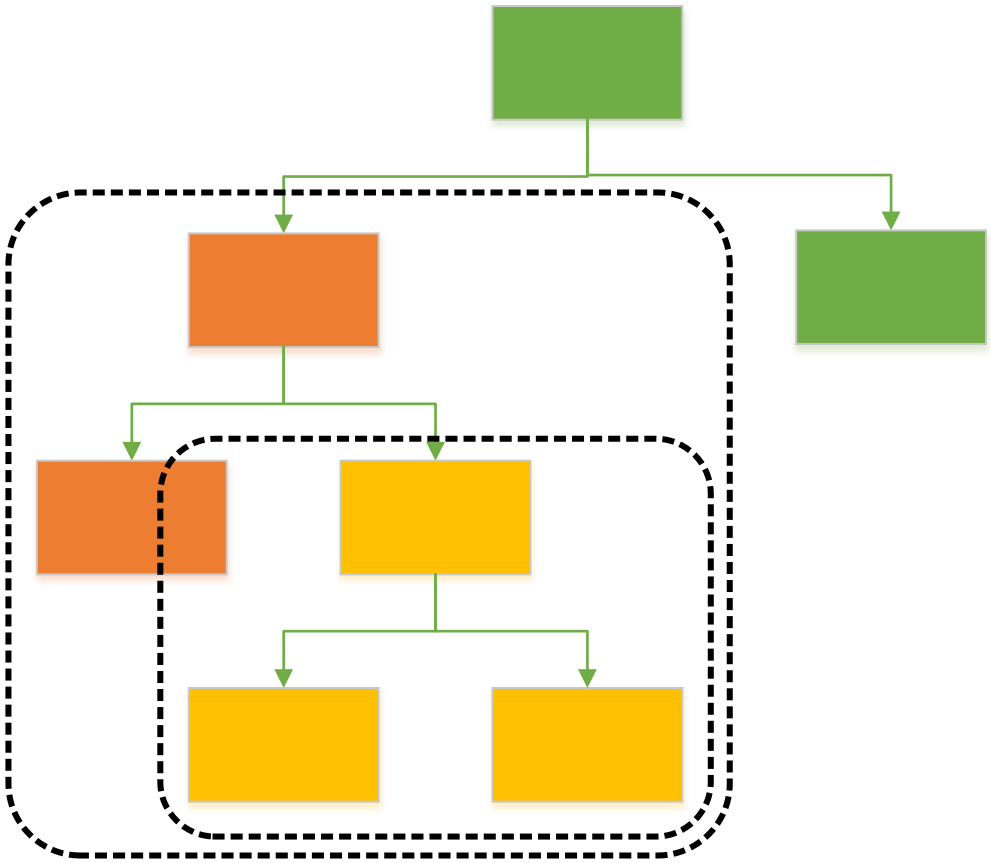
```
μ(if (age < 7) then result := "gyerek" end if) :=
  := if (age < 7) then
       log("Belépés az elágazásba")
       result := "gyerek"
       log("Kilépés az elágazásból")
     end if.
```

A $\tau_{\mu,\sigma}$ transzformáció végrehajtása után a következő szintaxisfát kapjuk eredményül. Az aláhúzott szintaktikai elemekkel bővült az eredeti alkalmazásunk:

```
if (age > 0 and age < 18) then
  log("Belépés az elágazásba")
  if (age <= 14) then
    log("Belépés az elágazásba")
    if (age < 7) then
      log("Belépés az elágazásba")
      result := "kisgyerek"
      log("Kilépés az elágazásból")
    end if
    if (age >= 7) then
      log("Belépés az elágazásba")
      result := "gyerek"
      log("Kilépés az elágazásból")
    end if
    log("Kilépés az elágazásból")
  end if
  result := "tinédzser"
  log("Kilépés az elágazásból")
end if
```

Formális megfogalmazása az egymást tartalmazó részfák esetére

Kicsit formálisabban a következőről van szó: tegyük fel, hogy T az eredeti szintaxisfánk és $\sigma: AST \rightarrow 2^{AST}$ egy szelektor, amire az igaz, hogy $\sigma(T) \neq \emptyset$ és $\exists T_i, T_j \in \sigma(T): T_i \subset T_j$. Mivel a σ szelektor egy halmazzal tér vissza, így csak olyan esetekre vagyunk kíváncsiak, amik egymásnak a valódi részfái. Az 11. ábra egy ilyen lehetséges esetet szemléltet.



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

A 3.6.1. 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 olyanokat 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.

Kiindulva az előző gyakorlati példából, tegyük fel, hogy a $\mu \in \mathrm{M}$ makróval nem kiegészíteni akarjuk az elágazásokat, hanem a törzsüket egy üres utasítással akarjuk helyettesíteni. Formalizálva a következőt szeretnénk elvárni tőle:

$$\mu(\texttt{if ([kifejezés]) then [törzs] end if}) \coloneqq$$
$$\coloneqq \texttt{if ([kifejezés]) then skip end if}$$
$$[\texttt{kifejezés}] \subset T \wedge \rho([\texttt{kifejezés}]) = (v : \langle \text{kifejezés} \rangle) \in V$$
$$[\texttt{törzs}] \subset T \wedge \rho([\texttt{törzs}]) = (v : \langle \text{utasítás} \rangle) \in V.$$

Ebben az esetben azonban a $\tau_{\mu,\sigma}$ transzformáció nemdeterminisztikus lesz, ugyanis nem mindegy, hogy milyen sorrendben dolgozza fel a részfákat. Ha a legnagyobb csúcsszámút vesszük alapul, akkor a gyerekelemeit (amik elágazások) fogjuk helyettesíteni az üres utasítással és utána már a σ által kiválasztott részfák már nem lesznek részei a transzformált szintaxisfának.

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

Részfák feldolgozásának stratégiája (definíció)

Legyen $T \in AST$ jól definiált szintaxisfa és hozzá egy $\tau_{\mu,\sigma}$ szintaxisfa transzformáció. Az $f: AST \to \mathbb{N}$ függvényt *részfák feldolgozási stratégiájának* fogjuk nevezni, ha az igaz rá, hogy $\sigma(T) = \{T_1, \dots, T_n\}$ esetében

$$\overline{\overline{T_i}} \coloneqq \left(V_{T_i}, E_{T_i} \cup \left(n, \rho\left(\mu(T_i)\right)\right)\right) \text{ amire igaz, hogy } f(T_i) < f(T_{i+1}) \wedge \left(n, \rho(t)\right) \in E$$
$$(i \in \{1, \dots, n-1\})$$

$$\tau_{\mu,\sigma}(T) \coloneqq (T \setminus \bar{T}) \cup \left(\bigcup_{i=1}^{n} \overline{\overline{T_i}}\right).$$

Stratégia (legkisebb részfa feldolgozása először)

A stratégia lényege, hogy mindig azokat a részfákat dolgozzuk fel előbb, amik 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 $f: AST \to \mathbb{N}$ függvényt az alábbi módon definiáljuk:

$$f(T) \coloneqq |V|, \text{ ahol } T = (V, E) \in AST.$$

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

$$\mu(T) \coloneqq \begin{cases} \omega, & \sigma(T) = \emptyset \\ T, & \sigma(T) \neq \emptyset. \end{cases}$$

A gyakorlatban a kiválasztott részfákra a μ 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:

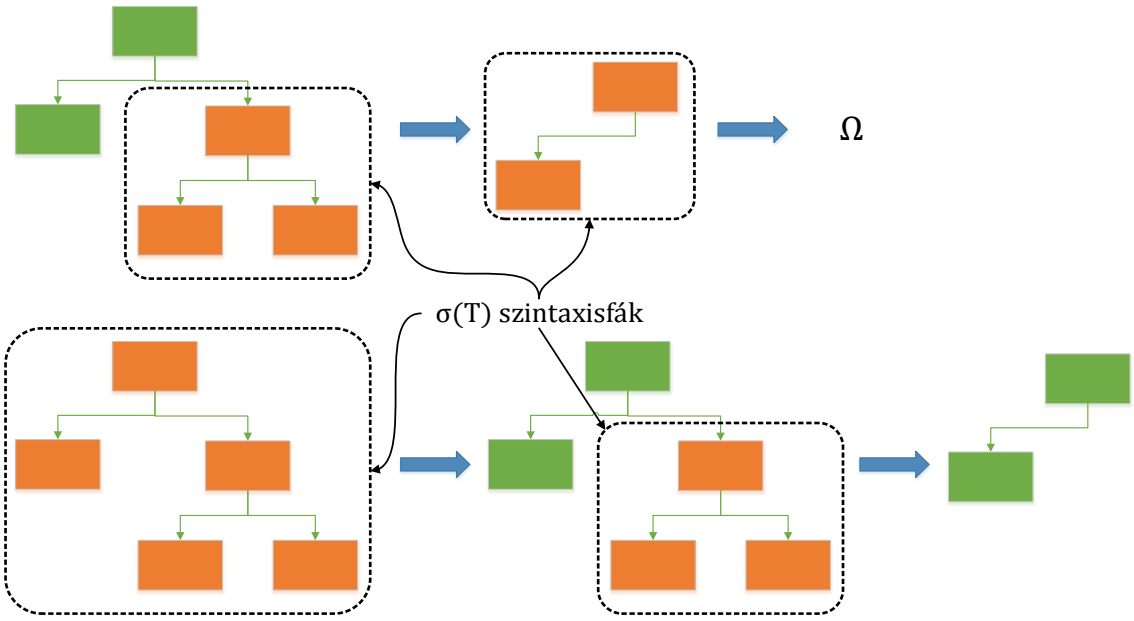
$$\sigma(T) \coloneqq \{T_1, \dots, T_n\}, \text{ ahol } f(T_i) < f(T_{i+1}) \ (i \in \{1, \dots, n\})$$
$$\sigma_i(T) \coloneqq \{T_i\} \quad (i \in \{1, \dots, n\})$$
$$\tau_i(T) \coloneqq \tau_{\mu,\sigma_i}(T)$$
$$T' = (\tau_n \circ \dots \circ \tau_1)(T).$$

A 3.4.13. 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.

A 12. ábra egy példát mutat be erre vonatkozóan. Tegyük fel, hogy a σ szelektor olyan részfákat választ ki, amik gyökerének van gyermeke (azaz a csúcsszámuk több mint egy).

A μ makró pedig csak akkor törli a kiválasztott részfát, ha gyökércsúcs gyermekei levélcsúcsok, különben helybenhagyja azt. Látható, hogy a felső sorban a kisebbik részfát dolgozzuk fel először, és csak utána jönne az egész fa (ugyanis azt is kiválasztásra került), aminek eredményéül egy üres fát kapunk.

Az alsó sorban először a nagyobb csúcsszámú fa kerül feldolgozásra, a μ makró ezt helybenhagyja, majd törli a másik kiválasztott részfát, aminek eredménye az lesz, hogy kaptunk egy két csúcsszámú fát.



12. 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 μ 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 σ 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 $f: AST \to \mathbb{N}$ függvény:

$$f(T') \coloneqq |V| - |V'|, \text{ ahol } T = (V, E) \text{ az eredeti szintaxisfa és } T' = (V', E').$$

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 $T_1 \subset T_2$ 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 σ szelektornak, akkor kiválasztjuk azt és transzformáljuk a μ 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 3.4.9) alapján úgy tűnhet, hogy először meghatározzuk az eredeti $T \in AST$ 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 T szintaxisfát a memóriában, hanem a makrót mindig a már transzformált fára alkalmazzuk, akkor előfordulhat olyan helyzet, hogy az előzőleg kiválasztott $T_i \in \sigma(T)$ 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 megoldást 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 máshogy 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.

3.7. Implicit makrók végrehajtásának sorrendje

A 3.4.13 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 3.6.2 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.

3.7.1. 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.

3.7.2. Transzformációk végrehajtása szelektorok specialitásai sorrendjében

Az előző alfejezetben (lásd 3.7.1) 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.[16]

Azt mondjuk, hogy a $\sigma \in \Sigma$ szelektor *szigorúan speciálisabb* egy másik $\sigma' \in \Sigma$ szelektortól, ha igaz rá, hogy $\forall T \in AST : |\sigma(T)| < |\sigma'(T)|$, azaz a σ bármilyen szintaxisfa

[16] 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.

esetében kevesebb részfát választ ki, mint a σ'. *Gyengén speciálisabbnak* mondjuk, ha $\forall T \in AST : |\sigma(T)| \leq |\sigma'(T)|$.

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.

3.7.3. 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 3.7.1-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.

3.7.4. 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.

3.7.5. 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 vegyítve próbáljuk meg őket használni.

A definiálás sorrendjén alapuló módszer (lásd 3.7.1) mellé könnyedén implementálni lehet a prioritáson alapuló stratégiát (lásd 3.7.3). 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 3.7.1) a függőségeken alapuló stratégiával (lásd 3.7.4) 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 is.

Bármilyen végrehajtási stratégiát is választunk, fontos, hogy a felhasználó mindig tisztában legyen azzal, hogy milyen kódot fog generálni a fordító. Jó módszer lehet az,

hogy úgy implementáljuk a fordítóprogramot és a fejlesztési környezetet, hogy a programozó bármikor meg tudja tekinteni, hogy pontosan milyen kódot generált a fordítóprogram.

3.8. Makrók végrehajtása transzformált szintaxisfákra

A metaprogramozás definíciója (lásd 3.4.12) azt mondja, hogy a transzformációkat szekvenciálisan hajtsuk végre, és 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.

3.9. 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.

3.10. Szintaxisfa transzformáció alkalmazásai

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

Ebben az alfejezetben a transzformációkat még általánosan fogjuk tárgyalni, azaz nem foglalkozunk azzal, hogy a makrók végrehajtása hogyan van megvalósítva a fordítási folyamatban. A megvalósítás történhet külső kiegészítésként (*explicit módon*) is, amivel a fordítóhoz, mint külső komponenst csatlakoztatunk, vagy megjelenhet a feldolgozandó forráskódban (*implicit módon*) is, azaz már fordítás közben a nyelv saját értelmezőjével dolgozunk fel (mint egy szkript nyelvet).

3.10.1. Design by Contract

A *Design by Contract* (*DbC*) egy szoftverfejlesztési megközelítés, aminek az alapelve az, hogy a szoftverfejlesztők fogalmazzák meg formálisan a feladot (elő- és utófeltétel, illetve invariánsok segítségével), adjunk precíz és verifikálható specifikációt minden szoftverkomponensről és az absztrakt adattípusokat egészítsük ki a megfelelő elő-, utófeltétellel és invariánssal. Ezekre a specifikációkra, mint kontraktusokra hivatkozunk, innen ered a módszer neve is.

Ezt az *Eiffel* programozási nyelv tervezője *Bertrand Meyer* fogalmazta meg és implementálta először, de mára más nyelveknél is megjelent a lehetőség, hogy ilyen típusú szerződéseket határozzunk meg a komponenseink számára. A [13] cikkben részletesen kifejti Meyer, hogy hogyan lehet alkalmazni a gyakorlatban a *DbC*-t.

Az egyik ilyen a Microsoft által fejlesztett *Code Contracts*, ami az *Eiffel* nyelvtől eltérően nemcsak futási időben értékeli ki a feltételeket, hanem ha tudja, akkor már fordítási időben megpróbálja.

Nemcsak az *Eiffel* támogatja nyelvi szinten a kontraktusok definiálását, hanem maga az *Oxygene* is, ami a *Delphi* nyelv *.NET* alapú kiterjesztésének felel meg.

Ha megnézzük közelebbről, hogy hogyan működnek a kontraktusok az *Eiffel*-ben, akkor azt láthatjuk, hogy a metódusok lefutása előtt és után futásidőben értékeli ki a neki megadott elő- és utófeltételeket. Ha nem teljesül valamelyik feltétel, akkor az alkalmazás futása alatt hibát kapunk. Invariánsokat is definiálhatunk a ciklusok esetében, illetve osztályszinten is, amik garantálják, hogy az adott objektumunk állapota mindig ki fogja elégíteni a feltételt.

Ez valójában szintaktikai cukorka, hiszen a *DbC*-t bármilyen imperatív nyelven meg lehet fogalmazni elágazások segítségével.

Amivel még kiegészíthetjük a *DbC* adta előnyöket, az a statikus kódanalízis eszközei, azaz a feltételeket már megpróbálhatjuk a forráskód fordítása alatt is ellenőrizni, ha arra van lehetőség.

Előfeltételek implementálása szintaxisfa transzformációval

Lássuk, hogy hogyan lehetne megfogalmazni szintaxisfa transzformációk segítségével az elő- és utófeltételeket. Tegyük fel, hogy csak függvények/eljárások esetében szeretnénk explicit definiálni őket, mégpedig úgy, hogy valamilyen annotációval látjuk el őket. Ha találunk ilyen függvénydeklarációkat, akkor legeneráljuk a szükséges elágazásokat, ha viszont nem, akkor egyszerűen helyben hagyjuk őket. Formalizálva a következőképpen néznek ki:

$\sigma(T) \coloneqq \{(v : \langle\text{függvény}\rangle) \in V\}$

$\boldsymbol{\mu}$(`@pre_condition([feltétel])`
`function [azonosító]([függvényparaméterek]) begin`
`  [törzs]`
`endfunction`) $\coloneqq$

```
function [azonosító]([függvényparaméterek]) begin
  if (not [feltétel]) then
    error("Előfeltétel megsértése")
  endif
  [törzs]
endfunction.
```

A gyakorlatban a $\mu \in \mathrm{M}$ makró végrehajtása során statikus kódanalízist is végezhetünk, ha megvan hozzá a megfelelő mennyiségű információnk. Viszont, hogy ezt implementálni is tudjuk, ahhoz már szükségünk lenne pár szemantikus információra, mint például a feltételeknek átadott kifejezések, használt változók, esetleg különböző objektumok típusára. Szerencsére a legtöbb esetben, aránylag könnyen ki lehet következtetni az adott nyelvek esetében.

Utófeltételek implementálása szintaxisfa transzformációval

Az utófeltételeknél kicsit bonyolódik a helyzetünk, hiszen ha a programozási nyelv szabályai megengedik, hogy a bárhol kiléphetünk a metódusból[17], akkor nem elég csak a törzs végére beszúrni egy elágazást, mivel az az utasítás lehetséges, hogy nem fog lefutni.

Ha a szintaktikai szabályok megengedik, hogy a metódus törzsén belül is definiálhassunk egy újabb metódust (aminek a szignatúrája megegyezik az eredeti metódussal), akkor nagyon egyszerű dolgunk van, mivel elég csak a törzset bemásolnunk abba. Ha ezzel megvagyunk, akkor meghívjuk ezt az új metódust (ha van valamilyen visszatérési értéke, akkor eltároljuk azt), majd utána már probléma nélkül leellenőrizhetjük az utófeltételeket.

Azonban ha nem engedik meg a szabályok a metóduson belüli metódusok definiálását, akkor a globális hatókörben kell definiálnunk őket. Vigyáznunk kell ilyenkor, hogy a felhasználó még véletlenül se hívhassa meg ezeket az eljárásokat, ezért valamilyen egyedi nevet kell tudnunk neki generálni. Generálhatjuk véletlenszerűen, esetleg valamilyen extremális karakter segítségével, ami nincs megengedve alapértelmezetten vagy valamilyen szabály szerint.

A következő szelektor és makró az első megoldásra mutat egy lehetséges megoldást. Az eredeti függvényhez generálunk egy **`_[azonosító]_body`** nevű segédfüggvényt, ami a törzset reprezentálja. Nemcsak az függvényparaméterek értékét, hanem a külső függőségeket[18] is megkapja aktuális paraméterként.

[17] Azaz függvényből vagy eljárásból.
[18] A külső függőségek lehetnek globális változók, vagy külső hatókörben definiált objektum értéke

$\sigma(T) \coloneqq \{(v : \langle \text{függvény} \rangle) \in V\}$

$\boldsymbol{\mu}$(`@post_condition([feltétel])`
`function [azonosító]([függvényparaméterek]) begin`
`  [törzs]`
`endfunction`) $\coloneqq$

```
function _[azonosító]_body([függvényparaméterek],
                           [külső függőségek])
begin
  [törzs]
endfunction
function [azonosító]([függvényparaméterek]) begin
  [lokális változók deklarálása]
  result := _[azonosító]_body([függvényparaméterek],
                              [külső függőségek])
  if (not [feltétel]) then
    error("Utófeltétel megsértése")
  endif
  return result
endfunction.
```

Egyetlen hátránya mindkét megoldásnak, hogy a lokális változókra nem szabhatunk meg feltételt, ugyanis a hatókörük már nem az eredeti metódus belseje. Ezt egy kevés trükközéssel ki lehet kerülni, ugyanis megtehetjük azt, hogy az összes lokális változót definiálhatjuk az eredeti metódus elején, aktuális paraméterként átadhatjuk a generált metódusnak referenciák szerint, majd a legvégén már képesek vagyunk őket is bevonni az utófeltétel ellenőrzésébe.

Ha az előző megoldások közül egyik sem ideális, akkor egy kicsivel komolyabb kihívás elé fogunk állni. Először is fel kell kutatnunk a metódus lehetséges kilépési pontjait. Függvények esetében a visszatérési értéket egy külön változóban el is kell tárolnunk.

Ezután törölnünk kell a kilépési pontokat, és úgy kell transzformálnunk a fát, hogy az utána következő utasítások már biztosan ne fussanak le. Ezt elágazások definiálásával érhetjük el úgy, hogy nyilvántartjuk egy logikai változóban, hogy kiléptünk-e már és ennek az értékét ellenőrizzük állandóan. Legvégül pedig elvégezzük a törzs legvégén az utófeltételek ellenőrzését, és ha szükséges, akkor visszatérünk a megfelelő eredménnyel.

Optimalizációként elágazások helyett használhatnánk **`goto`** utasításokat is (a kódgenerálásnál is hasonlóan oldják meg a fordítóprogramok az *Assembly* nyelv szintjén), ezt a nyelvi lehetőséget azonban a legtöbb programozási nyelv nem támogatja.

Ciklus invariáns implementálása szintaxisfa transzformációval

Ciklus invariáns ellenőrzését hasonlóképpen meg tudjuk fogalmazni, mint az elő- és utófeltételekét. Itt is mérlegelnünk kell, hogy milyen nyelvi elemek állnak rendelkezésünkre az implementációhoz, ugyanis nagyon hasonló problémákkal állunk szemben.

Azonban nemcsak a metódus kilépési pontjaira kell ügyelnünk, hanem ellenőriznünk kell azt is, hogy milyen nyelvi szerkezetek segítségével tudunk kilépni a ciklus törzséből (mint pl.: **`break`** és **`continue`**).

Az alapelvünk ugyanaz lesz, azaz kiszervezzük a ciklus törzsét egy függvénybe. Itt azonban már a visszatérési értéknél nyilván kell tartanunk azt is, hogy hogyan léptünk ki a ciklusból, ugyanis ennek megfelelően kell reagálnunk.

A következőkben tegyük fel, hogy az imperatív nyelvünk rendelkezik **`break`** utasítással is.

$\sigma(T) \coloneqq \{(v : \langle \text{ciklus} \rangle) \in V\}$

```
μ(@invariant([feltétel])
  loop ([ciklus feltétel]) do
    [ciklus törzse]
  endloop) :=
loop ([ciklus feltétel]) do
  if (not [feltétel]) then
    error("Nem teljesült a ciklus invariáns!")
  endif
  (type, result) := loop_function([törzs])
  if (not [feltétel]) then
    error("Nem teljesült a ciklus invariáns!")
  endif
  if (out_type == "return") then
    return out_result;
  endif
  if (out_type == "break") then
    break
  endif
endloop
```

Ebben az esetben a **loop_function** egy olyan explicit makró[19] lesz, ami a ciklus törzséből készít egy új törzsöt, de nem teljesen ugyanazt, mivel a **return** és **break** vezérlési szerkezeteket kicseréljük értékadásokra, ahol nyilvántartjuk azt is, hogy milyen típusú volt az adott kilépési pont. Így egy rendezett párral fog visszatérni a függvény, aminek az első eleme a kilépési pont típusát, míg a második annak az értékét tárolja. A **loop_function**-t felfoghatjuk egy másik makrónak.

[19] Az explicit makró egy olyan függvényszerű hívás, ami aktuális paraméterként elfogadhat vezérlési szerkezeteket is

A következő μ' makró azt szemlélteti, hogy milyen változtatásokat hajt végre az előbb említett két szerkezeten:

μ'(`break`) ≔ `return ("break", 0)`

μ'(`return [kifejezés]`) ≔ `return ("return", [kifejezés])`

A μ' makrót a `loop_function` függvény törzsének belsejében hajtjuk végre a `break` és `return` utasításokra.

3.10.2. Domain-Specific Language (DSL) definiálása

Sokszor nehéz imperatív nyelven konfigurálni egy olyan objektumot, amit sokkal kifejezőbben és olvashatóbban meg lehetne fogalmazni deklaratívan is. Elég csak a weboldalakra gondolni, hiszen a megjelenítést *HTML* és *CSS* nyelven definiáljuk, míg az üzleti logikát *JavaScript*-ben.

Jó lenne egy olyan nyelvi lehetőség az imperatív nyelveken belül, hogy már fordítási időben képes lenne különböző deklaratív (más akár bármilyen) nyelven írt kódrészleteket is feldolgozni és lefordítania az adott nyelvvé. Nagy előnye lenne, hogy még futás előtt fel tudnánk deríteni a fordítási hibákat, nem beszélve arról, hogy maga a feldolgozás sem futásidőben zajlana.

Saját *DSL*-eket ugyanúgy képesek lennénk szintaxisfa transzformációkkal létrehozni, hiszen a $\mu \in \mathrm{M}$ makróba kellene implementálni az adott nyelv fordítóját és a kódgenerálás helyett szintaxisfát készítenénk, amit a μ adna vissza eredményül.

Új szintaktikai elemet se lenne szükséges bevezetnünk, hiszen valamilyen sztring literálhoz hasonló nyelvi elembe be tudnánk csomagolni, és valamilyen annotáció vagy függvényhívás segítségével fel tudná ismerni a $\sigma \in \Sigma$ szelektor, hogy a fa mely részeit kell értelmeznie a *DSL* fordítójának.

Az már a nyelv tervezőjének a feladata lenne, hogy a felhasználót mennyire tájékoztatja arról, hogy mi történik a valójában a háttérben. Ezt alapos specifikációval, esetleg valamilyen névkonvenció bevezetésével meg lehetne tenni.

Tegyük fel, hogy a $\sigma \in \Sigma$ makrónkat az alábbi módon definiáljuk:

$$\sigma(\mathrm{T}) \coloneqq \left\{ (v : \langle \text{függvényhívás} \rangle) \in V \;\middle|\; \begin{array}{l} v = \texttt{[azonosító]([szöveg])} \wedge \\ (\texttt{[szöveg]} : \langle \text{szöveg} \rangle) \in V \end{array} \right\}.$$

Azaz csak azokra a szintaxisfabeli elemekre vagyunk kíváncsiak, amik függvényhívásokhoz hasonlítanak és az aktuális paraméterük valamilyen szövegliterál. Ezután már csak annyi a dolgunk, hogy elvégezzük a transzformációt, azaz a μ makróba implementált DSL nyelv fordítója legenerálja az adott nyelv szintaxisára a neki átadott kódot: $\mu(\texttt{[azonosító]([szöveg])}) \coloneqq DSL(\texttt{[szöveg]})$, ahol $DSL : \text{Szöveg} \to AST$ képező függvény.

Ahogy látható, az ilyen programozási nyelvekbe implementált eszközök segítségével sokat fejlődhet a kód karbantartása, olvashatósága, úgy hogy nem kell gyökeresen megváltoztatnunk a nyelv szintaktikai szabályait ahhoz, hogy használni.

Másik előnye lehet az is, hogy a fejlesztési környezet is képes lehet támogatni ezeket a nyelvi funkciókat, amivel sokkal gyorsabbá és biztonságosabbá válhat a fejlesztés is.

3.10.3. Refaktorálás szintaxisfa transzformációval

Adja magát a lehetőség, hogy miért is ne használhatnánk a szintaxisfa transzformációját refaktorálásra? A metaprogramozás nagyon hasonlít a refaktorálás folyamatára (lásd [16]), azzal a különbséggel, hogy magát a forráskódot transzformáljuk és nem az alkalmazás futását.

Ha olyan programozási nyelvet terveznénk, ami képes feldolgozni és manipulálni a saját szintaxisfáját, akkor egy *REPL* (*read-eval-print-loop*) környezet segítségével nem is lenne szükség külső fejlesztőkörnyezeti támogatottságra.

Kihívást jelenthet az is, hogy vannak olyan felhasználói esetek, amikor a transzformációnál szükség lehet különböző szemantikai információkra is. Ilyen lehet egy egyszerűnek tűnő azonosító átnevezése, esetleg annak a biztonságos törlése (azaz csak azokat az előfordulásokat töröljük, amik szemantikailag megegyeznek) vagy függvények kiemelése stb.

Nézzünk egy egyszerű példát, hogy hogyan lehetne átnevezni egy függvény azonosítóját. Tegyük fel, hogy a függvények nevéhez egy **`_function`** szuffixet szeretnénk konkatenálni.

$$\sigma\left(\underbrace{(V,E)}_{T\text{ szintaxisfa}}\right) \coloneqq \{\,(v:\langle\text{azonosító}\rangle) \in V \mid \exists(v':\langle\text{függvény}\rangle) \in V : (v',v) \in E\}$$

$\mu($**`function [azonosító]([függvény paraméterek])`**
`  [törzs]`
`endfunction`$) \coloneqq$

```
function [azonosító]_function([függvény paraméterek])
  [törzs]
endfunction
```

Vegyük észre, hogy ez még nem elég, ugyanis nem elég a függvények deklarációját megváltoztatni, hanem ugyanígy a függvényhívásoknál is el kell végeznünk az átnevezést. Ezért egy másik transzformációt is definiálni fogunk a következőképpen:

$$\sigma'(T) \coloneqq \{\,(v:\langle\text{azonosító}\rangle) \in V \mid \exists(v':\langle\text{függvényhívás}\rangle) \in V : (v',v) \in E\}$$

$\mu($**`[azonosító]([aktuális függvényparaméterek]`**$) \coloneqq$

```
[azonosító]_function([aktuális függvényparaméterek]
```

Esetünkben az imperatív nyelv szintaktikai szabályai miatt nem volt szükség szemantikai információkra, de ez általában nincs így, ezért lokális szinten mindenképpen valamilyen szemantikai ellenőrzést végre kell hajtanunk.

3.10.4. Tesztelés makrókkal (mockolás)

Napjainkra egyre elterjedtebbé vált a tesztelés vezérelt szoftverfejlesztés (*test-driven development*), aminek az alapelve, hogy először megírjuk a teszt eseteinket, lefuttatjuk őket, javítjuk a hibákat úgy, hogy minden tesztünk lefutása sikeres legyen, majd azután refaktoráljuk a kódbázisunkat.

Amikor egy-egy objektumot tesztelünk, akkor általában egység- és integrációs teszteket írunk, amivel meghatározzuk, hogy milyen viselkedést várunk el azoktól. Ezeknek az objektumoknak a használatához általában valamilyen függőségre van szüksége, hogy megfelelően tudjon működni.

Integrációs tesztek esetében a függőséggel való kapcsolatát is teszteljük, de egységteszteknél fontos, hogy semmilyen függőség ne tudja befolyásolni a teszt eredményét. Ezt a viselkedést úgy tudjuk elérni, hogy megpróbáljuk szimulálni ezeknek az objektumoknak a viselkedését és ezzel elérhetjük, hogy a tesztelés alatt álló típusunk, funkciónk biztosan a megfelelően működhessen. Erre egy általános megoldás, ha úgynevezett *mock*-okat[20] használunk a teszteléseinkhez.

Mock-okat írhatunk saját magunk is, de mára minden fontosabb programozási nyelvhez készültek könyvtárak, amik leveszik a felhasználó válláról a terhet és automatikusan el tudják készíteni a szimulált objektumokat. Viszont eddig ezek a *mock*-ok automatikusan csak futási időben tudtak elkészülni, míg szintaxisfa transzformáció segítségével ezt a lépést már fordítási időben is meg lehetne oldani, ezzel redukálva a tesztelés idejét.

A következő példánkban fel kell tennünk, hogy az imperatív nyelvünk támogatja az új objektumok deklarálását és ezeknek lehetnek tagfüggvényei is. Ennek megfelelően egészítsük ki a szintaktikai szabályrendszerünket a következő *EBNF* szabályokkal:

[20] Illetve *stub*-okat is létrehozhatunk. A tesztelés világában sokszor keverik a két fogalmat, pedig nem árt tudni, hogy a kettő nem ugyanaz. A *stub*-ok kevesebb funkcióval rendelkeznek, mint a *mock*-ok. A *mock*-ok képesek azt is figyelembe venni, hogy a tesztelés alatt lévő objektum vajon jól használta-e a mockolt objektumot (jó sorrendben hívta-e meg a függőség függvényeit, ha igen, akkor hányszor hívta meg őket stb.), míg a *stub*-ok erre nem képesek.

⟨objektum deklaráció⟩ ≔ **member** ⟨azonosító⟩.⟨azonosító⟩ **≔** ⟨tagfüggvény⟩

⟨tagfüggvény⟩ ≔ **function** (⟨formális paraméterek⟩)

{⟨utasítás⟩}$_1$

endfunction

⟨objektum adattag⟩ ≔ ⟨azonosító⟩{.⟨azonosító⟩{(⟨aktuális paraméterek⟩)}$_0^1$}$_0$

⟨objektum példányosítása⟩ ≔ **new** ⟨azonosító⟩

Az objektumok deklarációja úgy fog működni, hogy a típushoz pont (**.**) operátorral lehet deklarálni az új adattagokat (a szemantikája meg fog egyezni a *JavaScript*-ben látottal).

Tegyük fel, van egy **`Person`** nevű típusunk, aminek három tulajdonsága lesz (**`firstName`**, **`lastName`**, **`age`**) és egy **`name`** tagfüggvénye.

```
member Person.firstName = ""
member Person.lastName = ""
member Person.age = 0
member Person.name = function()
  return this.firstName + this.lastName
endfunction
```

Szeretnénk egy olyan makrót definiálni, ami az **`Person`** objektum deklarációjából fog készíteni egy olyan *stub*-ot, ami a **`name`** függvénynek szimulálja a viselkedését:

σ(T) ≔

$$\coloneqq \left\{ (v : \langle\text{objektum deklaráció}\rangle) \in V \;\middle|\; v = \begin{pmatrix} \textbf{member}\ \text{Person.name} \\ \coloneqq\ \langle\text{tagfüggvény}\rangle \end{pmatrix} \in V \right\}$$

μ(**`member Person.name:=[tagfüggvény]`**) ≔

```
   member Person.name:=[tagfüggvény]
   member PersonStub.name:= function([tagfüggvény par.])
     return "Gipsz Jabab";
   endfunction
```

Amint az látható a μ makró egy új **`PersonStub`** nevű típust hoz létre fordítási időben, így már nem függünk a **`Person`**-tól.

3.11. Összefoglalás

Ebben a fejezetben megnéztük a matematika eszközeivel, hogy mit is értünk metaprogramozás alatt és milyen problémák merülhetnek fel körülötte. Láttuk, hogy csupán arról van szó, hogy a szintaxisfánkat szeretnénk még a szemantikus ellenőrzés előtt valamilyen módon transzformálni, majd ezek elvégzése után a transzformált fából előállítani a kívánt alkalmazásunkat.

Fontos tételként említettük továbbá azt is, hogy a transzformációk sorrendjét nem szabad megváltoztatni, ugyanis könnyen lehetséges az, hogy nem a várt programot kapjuk eredményül. Ez alapján különböző végrehajtási stratégiák lettek bemutatva, hogy hogyan és milyen körülmények között lehetne elérni azt, hogy a fordítóprogramunk determinisztikus maradhasson.

Legvégül pedig megtekintettünk pár programozási technikát, paradigmát, amiket ki lehet váltani a metaprogramozés és azonbelül is a szintaxisfa transzformáció segítségével.

4. Megvalósítás és alkalmazás

A 3. fejezetben bemutattam, hogy milyen problémákba futhatunk a tervezés közben, ha a szintaxisfa transzformációs eszközöket szeretnénk implementálni a programozási nyelvünkbe. Jelen fejezet célja pedig az lesz, hogy az általam tervezett *MetaCode*[21] programozási nyelv milyen metaprogramozási lehetőségeket kínál a fejlesztők számára.

Az implementációnál az volt a cél, hogy ezeket a szintaxisfa transzformációkat lehessen szemléltetni és megmutatni, hogy már ennyire egyszerű programozási nyelv esetében is milyen mértékben ki lehet bővíteni akár új paradigmákkal.

Az alapötlet az volt, hogy egy *JavaScript* tudásához hasonló programozási nyelvet készüljön és a szintaxisfa transzformációk szelektorait a *CSS* nyelvben látott módon lehessen definiálni. A *Scala*-ban lévő megoldások (lásd 2.5) helyett, itt úgynevezett implicit makrókat is lehet definiálni, amik automatikusan végrehajtódnak a fordítás során.

4.1. A technológia kiválasztása

Egy programozási nyelv implementációjánál több dolgot is figyelembe kell vennünk, mint pl.: hogy milyen környezetben akarunk vele fejleszteni, milyen igényeknek kell megfelelnie, illetve milyen már meglévő szolgáltatásokat akarunk újra felhasználni.

4.1.1. Platformfüggetlenség

Fontos, hogy megfelelően válasszuk ki az fejlesztői eszközöket, hiszen egy ilyen projekt esetében hosszútávon szoktunk gondolkodni, azaz szem előtt kell tartani azt is, hogy minél kevesebb legyen a külső függőség, és ha van, akkor azok hosszútávon támogatva legyenek a készítői által.

[21] A projekt megtalálható a *https://github.com/szabototo89/MetaCode* link alatt, illetve a mellékelt DVD-n is. Az `install` mappában a telepítője, míg a `src`-ban a forráskódja.

Először meg kell határoznunk, hogy milyen környezetben fogják használni a programozási nyelvünket. A választásom a *.NET keretrendszerre* esett, ami jelenleg *Windows* környezeten érhető el, de a *Mono* projekt[22] segítségével más operációs rendszerekre (*Linux, Debian, Mac OS X* stb.) is könnyedén ki lehet terjeszteni.

A keretrendszert több mint 40 programozási nyelv támogatja, ezért a fordító egyes komponenseit, akár különböző nyelvek segítségével is implementálni lehet. A választás a *C#*-ra esett, mivel támogatja az objektum-orientált programozási paradigmát és jelenleg is a legnépszerűbb és legtámogatottabb a *.NET* fejlesztők körében.

4.1.2. Fordítógeneráló eszközök

Több fordítógeneráló eszköz áll a programozó rendelkezésére, de kétségtelen, hogy az egyik legjobban támogatott és legtöbbet használt maga az *ANTLR*. A választás azért erre esett, mivel különböző nyelvekre (többek között *C#*-ra is) képes legenerálni a lexikális és szintaktikus elemzőt, illetve egyik nagy újítása, hogy támogatja az adaptív LL(*)[23] elemzést.

Az új verzióban bevezetett újításoknak köszönhetően, egyszerűbben és sokkal gyorsabban tudunk nyelvi szabályokat definiálni és felépítéséből adódóan elegánsan szétválasztja az implementációt a szabályrendszer kódjától, ezzel téve sokkal karbantarthatóbbá a projektünket. A 4.0-ás verzióban bevezetett újdonságokról a [7] könyvben olvashat részletesebben.

Elsődlegesen *Java*-ra generálja az elemző kódját, de lehetőségünk van *C#* nyelvet is választani. További előnye, hogy a *Visual Studio* fejlesztőkörnyezethez létezik egy *ANTLR Language Support* nevű kiterjesztése is, ami teljes támogatást nyújt a fejlesztéshez.

[22] *http://www.mono-project.com/*

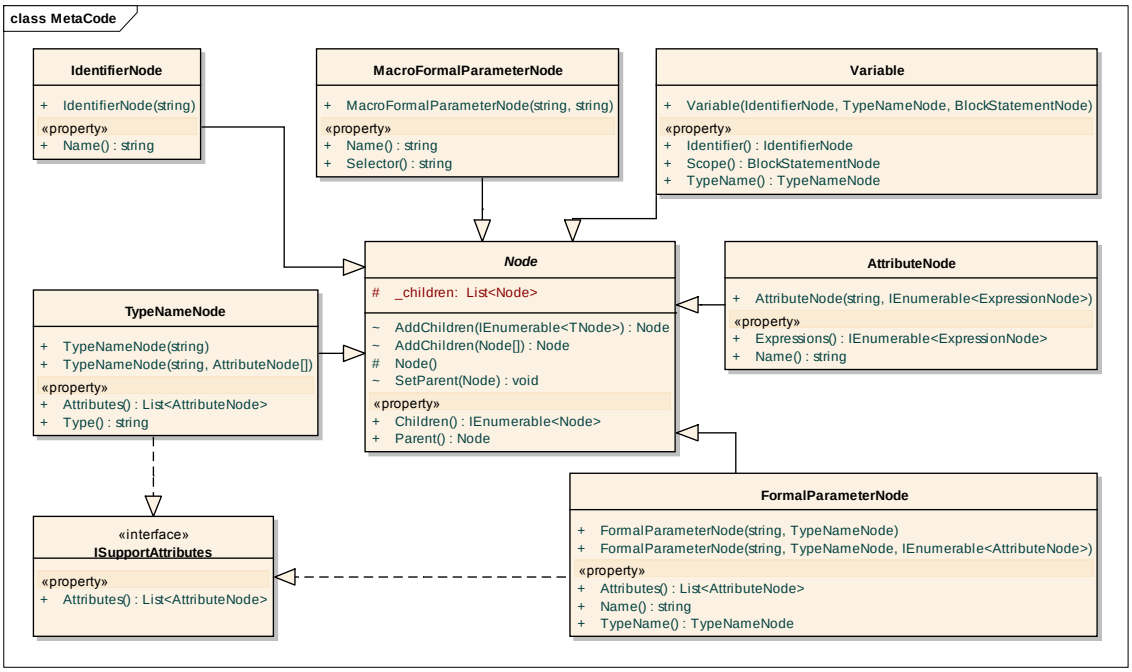
[23] Adaptive LL(*) egy új elemző stratégia, amit Terence Parr fejlesztett az ANTLR legújabb verziójához. Egyszerre vegyíti az LL(k) elemzők egyszerűségét, hatékonyságát és kiszámíthatóságát a GLR-típusú stratégiák erejével. Részletesebben a [6] cikkben olvashat róla.

4.2. A nyelv implementációja

A nyelv implementációjánál fontos volt, hogy függetleníteni tudjuk a szintaxisfa reprezentációját a fordítógeneráló eszköztől. Egyik előnye, hogy így bármikor le lehet cserélni az *ANTLR*-t valami más technológiára (akár saját megvalósításra is), és mivel saját magunk fogjuk szintaxisfát transzformálni, ezért saját megoldással kellett előállni a szintaxisfa reprezentációját illetően is.

4.2.1. Szintaxisfa reprezentációja

Ennek megfelelően a **`MetaCode.Compiler.AbstractSyntaxTree.Node`** osztályból származnak az összes absztrakt szintaxisfa eleme, mint pl. az elágazást, attribútumot vagy akár azonosítót reprezentáló csúcsok osztályai. A szintaxisfa mutabilis módon lett megvalósítva, azaz a csúcsok belső állapotát kívülről is meg lehet változtatni. A 13. ábra mutatja a **`Node`** osztály és annak néhány leszármazottjának a kapcsolatát.



13. ábra – Az absztrakt szintaxisfa őscsúcsának és néhány elemének az osztálydiagramja

A szintaxisfa csúcsai úgy lettek megvalósítva, hogy minden egyes objektum, pontosan tudja, hogy mik a gyerek elemei (`Children` tulajdonság) és mi a szülőeleme (`Parent` tulajdonság). Ezzel a fa bejárását akár teljesen általános meg tudjuk írni, anélkül, hogy tudnánk éppen milyen típusú csúcsnál járunk.

4.2.2. Absztrakt szintaxisfa bejárásáért felelős osztályok

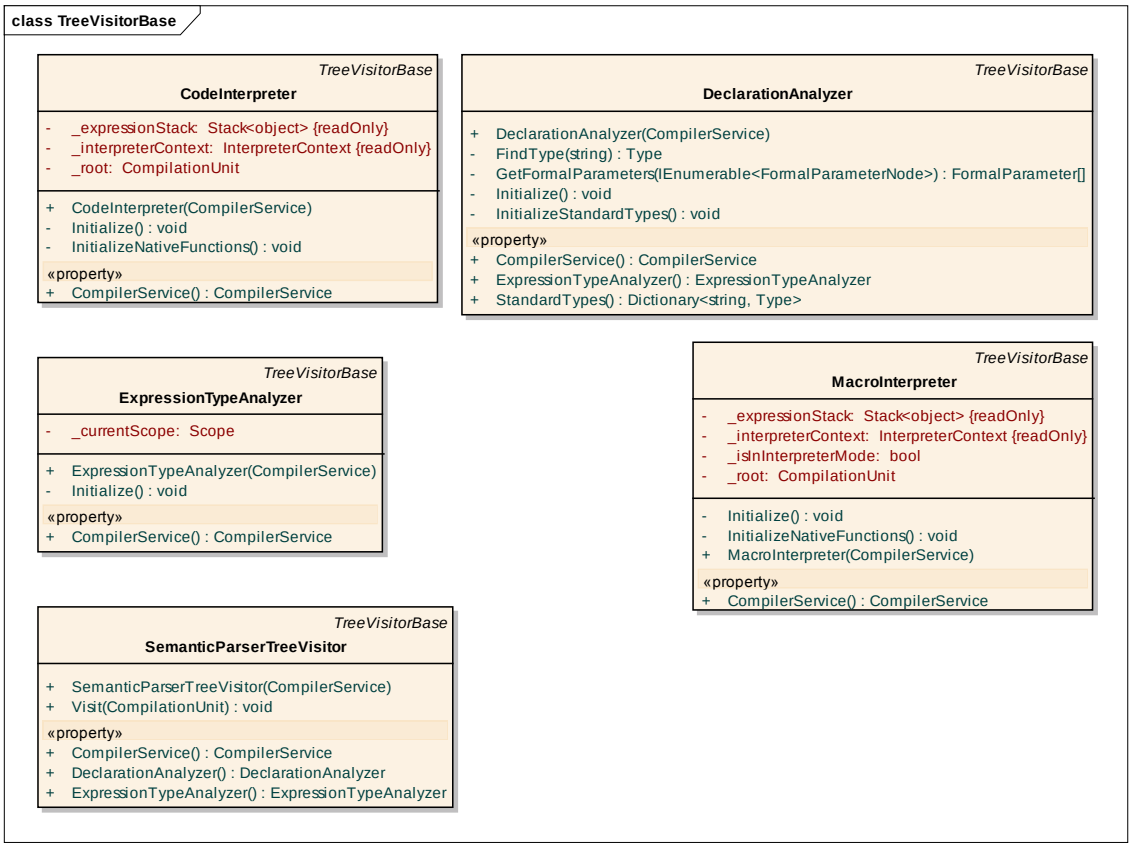
Nem elég reprezentálnunk a memóriában a szintaxisfánkat, be is kell ezeket a csúcsokat járnunk, ugyanis így fogjuk implementálni a szemantikus ellenőrzőt (mint pl.: típusok megfelelő használata), illetve interpretálni a felhasználó által írt kódot.

Ennél azonban sokkal többet fogunk tenni, ugyanis a szintaxisfa transzformációkat is végre kell tudnia hajtania a fordítónak. A transzformációk egy nagyon egyszerű elven fognak működni, egyszerűen a bejárás közben interpretáljuk a szintaxisfát (közben természetesen elvégezzük a szemantikus ellenőrzést) és végrehajtjuk az adott makrót.

A makrók végrehajtása közben módosulhat a szintaxisfánk, ezért nem érdemes a makrón kívüli részeket szemantikusan ellenőrizni, hiszen egyáltalán nem biztos, hogy a végleges alkalmazásnál szerepet fognak játszani.

A fordítás legvégén, ugyanúgy, mint a makrók esetében interpretáljuk a szintaxisfát, a makrók kivételével. A fordító úgy lett megtervezve, hogy ne csak interpretáltan lehessen végrehajtani a kódot, hanem képes legyen kódot is generálni.

Minden bejárónak egy általánosan elkészítetett `TreeVisitorBase<TResult>` generikus osztályból kell, hogy származzon. A 14. ábra a `TreeVisitorBase` osztályból leszármazó bejárókat mutatja, amit a fordítás alatt használunk.



14. ábra – TreeVisitorBase osztályt megvalósító bejárók

- **`MacroInterpreter`** osztály: Ez az osztály felelős a szintaxisfa transzformációk végrehajtásáért.
- **`SemanticParserTreeVisitor`** osztály: A szemantikus ellenőrzéseket ez az osztály végzi. Bár a makrók „nyers” (azaz szemantikus ellenőrzés nélküli) szintaxisfákon végeznek műveleteket, de a makrókat a végrehajtásuk alatt szemantikusan ellenőrizni kell, ezért nemcsak a transzformációk után, hanem közben is használjuk.
- **`ExpressionTypeAnalyzer`** osztály: Kifejezések típusának kikövetkeztetéséért felelős osztály, amit a szemantikus ellenőrzések alatt veszünk igénybe.
- **`DeclarationAnalyzer`** osztály: Egy adminisztratív jellegű osztály, ugyanis a deklarációkat tartja számon, segítségével tudja ellenőrizni a

`SemanticParserTreeVisitor`, hogy egyes függvények, változók deklarálva vannak-e.

- `CodeInterpreter` osztály: Nagyon hasonlít a `MacroInterpreter`-hez, azzal a különbséggel, hogy nem a makrókat, hanem a szintaxisfa többi részét hajtja végre.

4.2.3. MetaCode nyelv implementációja

A *MetaCode* tervezése alatt a cél az volt, hogy egy kis tudású imperatív nyelv készüljön el, ami támogatja a szintaxisfa transzformációját. Ezt szem előtt tartva a *JavaScript*-hez hasonló tudással felvértezett nyelv készítése lett célul kitűzve. A nyelv szintaxisáról a 1. táblázatban lehet olvasni.

MetaCode nyelv szintaxisa	
Nyelvi szerkezet formája	**Példa**
üres utasítás	
`skip;`	`do skip; end;`
értékadás	
`[változó neve] = [jobb érték];`	`a = 10;`
szekvencia	

`[attribútumok]`
`do`
`  [utasítás]`
`end;`
	`@not-null do`
	`  a := 10;`
	`end;`

elágazás	

[attribútumok]
if ([logikai kifejezés]) then
  [utasítások]
else
  [utasítások]
end;
	if (a > 4)
	  print("True!");
	else
	  print(„False!”);
	end;

while ciklus	

[attribútumok]
while ([logikai kifejezés])
  [utasítás]
	while (i < 10) do
	  i := i + 1;
	end;

foreach ciklus	

[attribútumok]
foreach (var [azonosító] in
[iterálandó kifejezés])
  [utasítások]
	foreach (var item in [1,2,3])
	  print(item);

tömbkifejezés	
’[’ [kifejezés],...,[kifejezés] ’]’	[1,2,3,4,5]
TreeSelector lekérdezés	
{ [részfa lekérdezése] }	{ if > sequence }

függvény definíciója	

[attribútumok]
function [függvény neve]([függvényparaméterek]) : [függvény visszatérési típusa] do
  [utasítások]
end;
	function abs(a: number): number
	do
	  if (a > 0) return a;
	  else return –a;
	end;

makró definiálása	

[attribútumok]
macro [név]([paraméter] : [TreeSelector kifejezés]) do
  [utasítások]
end;
	macro dummy(tree: { * }) do
	  skip;
	end;

attribútumok definiálása	
[attribútumok] attribute @[attribútum neve]([paraméter])	attribute @id(name: string)
objektumok definiálása	

[attribútumok]
object [objektum neve]
  [adattagok]
end;
	object Person
	  age: number;
	  name: string;
	end;

1. táblázat – MetaCode nyelv szintaxisát leíró táblázat

4.2.4. Szelektorok implementációja

A szelektorok nagyon fontos elemei a *MetaCode*-nak, hiszen kényelmes megoldást nyújtanak a szintaxisfa részfáinak a kiválasztására. Az alapötlet az volt, hogy definiáljunk

egy deklaratív nyelvet, amivel nagyon könnyen tudjuk leírni, hogy milyen típusú részfákon szeretnénk transzformációkat végrehajtani.

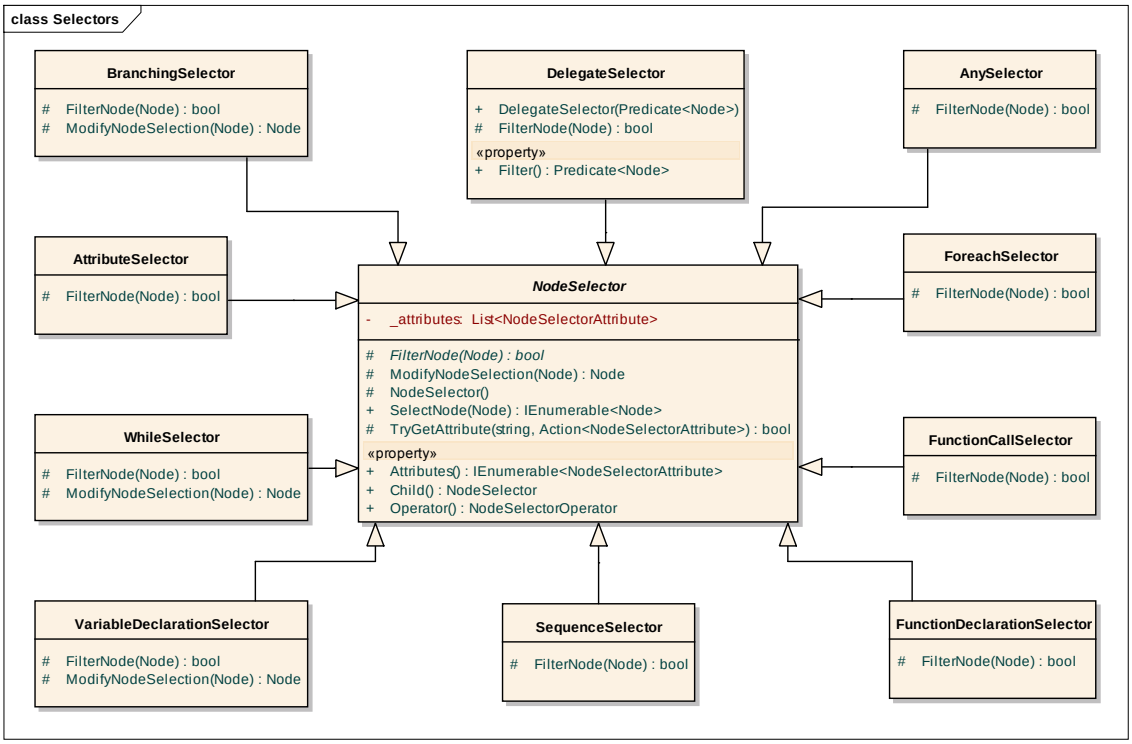
Ha alaposan végigtekintünk, hogy milyen megoldások születtek eddig az informatika világában, akkor szinte felkínálja magát a *CSS* nyelv, ugyanis ugyanazokkal a tulajdonságokkal bír, mint amit mi is szeretnénk implementálni. A *CSS* nyelv a szelektorai segítségével egy adott fa reprezentáción (ami egy *DOM*, azaz *Document Object Model* szokott lenni) kiválasztja a szükséges részfákat és azokon elvégzi a felhasználó által definiált értékeket. A *CSS* nyelv specifikációjáról a [8] link alatt olvashat róla részletesebben.

A *jQuery* alapelve is ez, ugyanis egy *CSS* szelektor segítségével kiválaszthatjuk a megfelelő részfákat és azokon valamilyen *DOM* manipulációs műveletet végezhetünk. Ezen az úton indul el a *MetaCode* szelektorainak az ötlete is.

Szelektorok implementációja

A *MetaCode*-ba implementálásra került egy *CSS*-hez hasonló nyelv, amivel deklaratívan tudjuk lekérdezni a szintaxisfánk egyes részeit. Minden nyelvi elemet (deklarációt, vezérlési szerkezetet stb.) el lehet érni valamilyen név segítségével, majd a szelekció típusának megadása után újabb elemet lehet kiválasztani. A fordító automatikusan átadja makróknak a szelektorok által kijelölt részfákat, ezért sincs szükség explicit hívásra.

A szelektorok implementálása két részből áll. Az elsőben egy nyelvet kellett definiálni, ami alapján feldolgozásra kerül, hogy milyen szelektort példányosítson a fordító a memóriában. A második rész maga a memóriában lévő reprezentáció. Minden szelektor (azaz nyelvi elemeket kijelölő objektum) a `NodeSelector` nevű absztrakt osztályból származik. A szelektororok osztálydiagramját a 15. ábra szemlélteti.



15. ábra – Szelektorok megvalósítása a MetaCode projektben

Minden szelektornak csak egy gyermeke lehet (azaz lényegében össze vannak láncolva lineárisan), amit a **`Child`** nevű tulajdonság tart számon. A **`FilterNode`** metódus dönti el egy adott részfáról, hogy megfelel-e a szelektor által definiált feltételeknek. A **`SelectNode`** metódus azokkal a részfákkal tér vissza, amik átmennek az ellenőrzésen.

Vannak olyan esetek, amikor egy adott nyelvi elemnek egy részét szeretnénk kijelölni (pl. elágazásnál csak a feltétel kifejezését, ciklusnál csak a törzsét). Ilyenkor lehet felüldefiniálni a **`ModifyNodeSelection`** függvényt, ami a szelektor által kijelölt részfa egyik csúcsával tér vissza.

Szelektorok lekérdezőnyelve

A *TreeSelector*, azaz a szelektorok lekérdezőnyelvének, szintaxisa nagyon hasonlít az előbb említett *CSS* nyelvéhez, azzal a különbséggel, hogy nem *HTML*, hanem a

MetaCode nyelvi elemeit lehet velük kijelölni. Funkcionalitásában azonban egy az egyben megegyezik, azaz ugyanúgy vannak kiválasztó operátorai, tudunk tulajdonságok (azaz attribútumok) alapján is keresni, és egyszerre több részfát is kiválaszthatunk. Az alább a *TreeSelector* nyelv szintaktikai szabályai olvashatóak *ANTLR*[24] nyelven definiálva:

```
grammar TreeSelector;
init       : '{' selectors '}';
selectors  : selector (',' selector)*;
selector   : statement (OPERATOR? selector)?;
attribute  : '[' ID ('=' ID)? ']';
OPERATOR   : '>' | '+';
statement  : ID attribute*;
ID         : ('@'? (LETTER|'_') (LETTER|'_'|'-'|[0-9])*) | '*';
fragment LETTER : [a-zA-Z];
WHITESPACE : [ \t]+ -> skip;
NEWLINE    : '\r'? '\n' -> skip;
```

Az 2. táblázat foglalja össze, hogy milyen szelektorokat használhatunk a makrók definiálásánál és azoknak milyen attribútumai lehetnek.

A TreeSelector nyelv szelektorjai			
Szelektor jelölése	**Attribútumai**	**Leírás**	**Példa**
`*`	nincs	A legáltalánosabb szelektor, ami egy adott csúcs összes gyermekét kiválasztja.	`if > *` – Kiválasztja az elágazás összes gyermekét.

[24] Az olvasó az *ANTLR* nyelv részletes specifikációjáról a [7] könyvben és 0 helyen olvashat

A TreeSelector nyelv szelektorjai			
Szelektor jelölése	**Attribútumai**	**Leírás**	**Példa**
`sequence`	**`length`**: a szekvenciában lévő utasítások számára lehet megadni egy szűrési feltételt.	Kiválasztja az összes szekvenciát egy adott részfában.	**`sequence[length=5]`** – Kiválasztja azokat a szekvenciákat, amiben pontosan 5 utasítás lett definiálva.
`if`	**`condition`**: az elágazás feltételével tér vissza. **`true-statement`**: az elágazás igaz ágával tér vissza. **`false-statement`**: az elágazás hamis ágával tér vissza.	Ezzel a szelektorral tudjuk lekérdezni a forráskódban definiált elágazásokat. Ha az elágazás egyes elemeire vagyunk kíváncsiak, akkor finomhangolhatjuk azzal, hogy az attribútumait használjuk.	**`sequence > if[condition]`** – Kiválasztjuk összes olyan elágazás feltételét, amik egy szekvencián belül lettek definiálva.
`while`	**`condition`**: a ciklus feltételével tér vissza. **`body`**: a ciklus törzsével tér vissza.	Ezzel a szelektorokkal tudjuk lekérdezni a **`while`** típusú ciklusainkat a forráskódból. Ugyanúgy, mint az elágazásnál, itt is finomhangolhatjuk a lekérdezést az attribútumokkal.	**`if > while[body]`** – Lekérdezzük az összes olyan ciklus törzsét, ami egy elágazáson belül van definiálva.

A TreeSelector nyelv szelektorjai			
Szelektor jelölése	**Attribútumai**	**Leírás**	**Példa**
foreach	**loop-variable:** A ciklusváltozóval tér vissza. **loop-variable-type:** A ciklusváltozó típusával tér vissza. **iterable:** az iterálás alatt álló kifejezéssel tér vissza. **body:** a ciklus törzsével tér vissza.	Ezzel a szelektorral tudjuk lekérdezni a **foreach** típusú csúcsokat a fában. Attribútumokkal finomhangolhatjuk a lekérdezést.	**foreach[loop-variable] –** Visszatér a foreach-ben lévő változók nevével.
variable	**identifier:** a változó nevével tér vissza, illetve ha megadunk valami nevet, akkor az olyan nevű változókkal tér vissza. **initial-value**: a változó iniciálizálási értékével tér vissza. **type**: a változó típusával tér vissza.	Ezzel a szelektorral a változók deklarációját lehet lekérdezni. Lehet szűrni név, típus alapján és az attribútumok segítségével a deklaráció egyes részeit is lekérdezhetjük.	**variable[identifier=index]** – Visszatér az **index** nevezetű változókat.

A TreeSelector nyelv szelektorjai			
Szelektor jelölése	**Attribútumai**	**Leírás**	**Példa**
`function-call`	**`name`**: a függvényhívás nevére lehet szűrni ezzel a változóval .	Ezzel a szelektorral a függvényhívásos kifejezésekre lehet szűrni.	**`function-call[name=max]`** – Kiválasztjuk a **`max`** nevű függvényhívásokat.
`attribute`	**`name:`** az attribútumok nevére lehet vele szűrni.	Ezzel a szelektorral lehet lekérdezni az attribútum deklarációkat.	**`attribute[name=not_null]:`** Visszatér a **`not_null`** nevű attribútumok deklarációjával.

2. táblázat – A MetaCode nyelvben lévő szelektorok

4.2.5. MetaCode nyelvi példa

Lássunk egy példát arra, hogy hogyan is működik a *MetaCode* a gyakorlatban. Első példában még nem fogjuk használni a makrókat, mint nyelvi lehetőségeket, egyszerűen megnézzük, hogy hogyan tudjuk felírni az összegzés tételét.

```
// deklaráljuk a szükséges változókat
var i : number = 0;
var n : number = 100;
var sum : number = 0;
// while ciklussal összeadjuk
// a 0-tól 100-ig a számok kétszeresét
while (i < n) do
  sum = sum + 2 * i;
  i = i + 1;
end;
```

Tegyük fel, hogy szeretnénk minden **while** ciklus törzsét kiegészíteni két utasítással[25] (**prependTo** és **appendTo**), amik azt jelzik, hogy kezdődött el a ciklus törzsének a végrehajtása, míg a másik, hogy mikor fejeződött be. Ezt úgy fogjuk elérni, hogy definiálunk rá egy makrót[26]:

```
// A LogLoopsMacro kiválasztja a globális hatókörben lévő
// elágazosok igaz ágát választjuk ki
macro LogLoopsMacro(tree: { * > whileBody }) do
      // hozzácsatoljuk a debug függvényhívást
      // a szekvencia végéhez
      appendTo(
         functionCall('debug', [str('Ciklus vége ...')]), tree
      );
      // hozzácsatoljuk a debug függvényhívást
      // a szekvencia elejéhez
      prependTo(
            // a függvényhívást a functionCall
            // függvénnyel állítjuk elő
            // ez automatikusan visszatér annak
            // a szintaxisfájával első paraméterként
            // átadjuk neki a függvény nevét,
            // míg másodikként az aktuális paramétereket
            functionCall('debug',
                        [str('Ciklus kezdete ...')]), tree
      );
end;
```

[25] Mind a **prependTo**, mind pedig az **appendTo** egy-egy natív függvény, amik a fordítóban lettek definiálva. Feladatuk, hogy hozzácsatolnak új gyerekelemeket egy aktuális paraméterként átadott absztrakt szintaxisfának.

[26] A makrókat ugyanabban a fordítási egységben kell deklarálnunk.

Ha elindítjuk a fordítást, akkor a szintaxisfa transzformációnk az előbb bemutatott **while** ciklusból a következő kódot fogja generálni nekünk (természetesen a kód többi részét helyben fogja hagyni nekünk):

```
while (i < n) do
  debug("Ciklus kezdete ...");
  sum = sum + 2 * i;
  i = i + 1;
  debug("Ciklus vége ...");
end;
```

4.2.6. Szintaxisfa transzformációs függvények

Ebben az alfejezetben megnézzük, hogy milyen beépített transzformációs függvények állnak a rendelkezésünkre. Ezeket a függvényeket csakis a makrókon belül lehet használni.

- **ast([kódrészlet])**: Az **ast** függvény egy kódrészlet szintaxisfává való generálásában nyújt segítséget. Aktuális paraméterként csupán egy sztring literált kell átadni neki és annak a szintaxisfák sorozatával fog visszatérni. A következő példa is ezt szemlélteti, ahol egy **min** függvényhívás szintaxisfáját készítjük el:
  ```
  var tree : any = ast('min(a,b)');
  ```
- **convertToString([szintaxisfa])**: A konvertálás nemcsak egy oldalú lehet, hiszen a már meglévő szintaxisfát is vissza tudjuk alakítani szöveggé. Erre szolgál a **convertToString** függvény, ami paraméterül a fát várja. A következő példa azt mutatja, hogy a függvényhívást visszakonvertáljuk szöveggé:
  ```
  var call: string = convertToString(ast('min(a,b'));
  ```
- **find([részfa], [szelektor])**: Többször előfordulhat, hogy egy adott részfán belül szeretnénk újabb részfákat kiválasztani. Ezt a célt szolgálja a **find** függvény, aminek első paraméteréül átadva egy részfát, másodiknak

pedig egy szelektort visszatér a kiválasztott részfákkal. A következő példa esetében kiválasztjuk a **tree** szintaxisfa közvetlen elágazásait:

```
var subtree : any = find(tree, '{ * > if}');
```

- **detach([részfa])**: Előfordulhat, hogy nemcsak kibővíteni akarjuk az adott részfát, hanem leválasztani valamelyik gyerekelemét. Erre szolgál a detach függvény, ami lecsatolja a szintaxisfából az adott részfát. A lecsatolt részfa nem törlődik, azaz könnyedén megtehetjük azt, hogy áthelyezzük egy másik helyre azt. A következő kódrészlet azt mutatja be, hogy a lecsatolással töröljük az adott részfában lévő közvetlen elágazásokat:

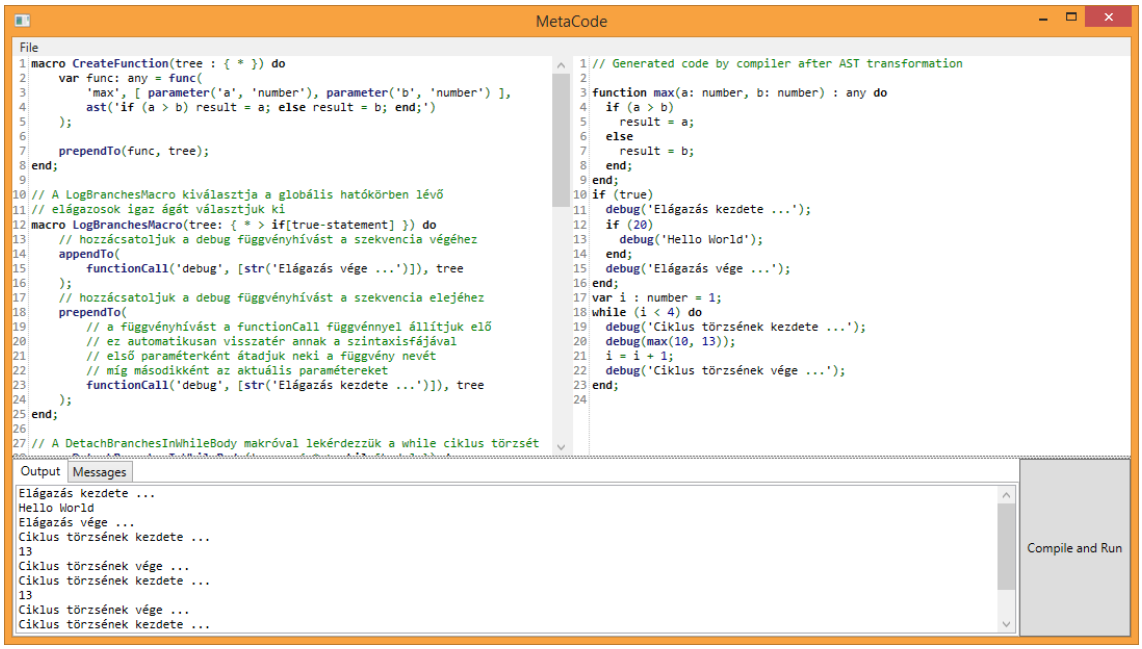
  ```
  detach(find(tree, '{ * > if }'));
  ```

- **appendTo([mit],[hova]) / prependTo([mit],[hova])**: Ha elkészítettünk egy új szintaxisfát, vagy lecsatoltunk egy előzőt, akkor vissza is csatolhatjuk az eredeti fához azokat. Ezt a legkönnyebben az **appendTo** és **prependTo** függvényekkel tudjuk elérni. Az **appendTo** a szekvencia legvégére, míg a **prependTo** a legelejére szúrja be az új utasításokat.
  Megjegyzés: Kifejezések esetében nem lehet őket használni.

Ezekkel a függvényekkel már kényelmesen tudjuk manipulálni a szintaxisfánkat és bonyolult makrókat is könnyedén létre tudunk hozni.

4.2.7. A MetaCode kódszerkesztő használata

A fordító egy fejlesztőkörnyezetbe lett implementálva a kényelmesebb használat érdekében. A felület két részre van osztva: a bal oldalon lehet szerkeszteni a tényleges forráskódot, míg a jobb oldalon a fordító által generált kódot lehet találni. A 16. ábra szemlélteti a program működését.



16. ábra – A MetaCode kódszerkesztője

A fordítás és a futtatás a *Compile and Run* gomb megnyomásával történik, a fordítási folyamatot alul a *Messages* fül megnyomásával lehet követni. Az *Output* résznél azok a szöveges információk jelennek meg, amiket a **`debug`** paranccsal írtunk ki a konzolra.

A *File* menüpont alatt lehet újat létrehozni (*New*), az eddig szerkesztetett elmenteni (*Save*) vagy egy már meglévő megnyitni szerkesztésre (*Open*).

4.3. Összefoglalás

Ebben a fejezetben megnéztük, hogy milyen módon lehetne implementálni a szintaxisfa transzformációkat egy nagyon egyszerű imperatív nyelvbe. Mint az látható volt az alapelvet könnyen meg lehet valósítani, egyetlen problémája a hatékony megvalósítása lehet, hiszen egy bonyolultabb nyelv esetében a fordítási idő, akár a sokszorosára is nőhet.

Megnéztünk tovább egy olyan megoldást, amivel a transzformációkért felelős szelektorokat sokkal egyszerűbben és kényelmesebben tudjuk definiálni. Kritikus pontja a

metaprogramozásnak az, hogy a nyelvi eszközöket minél felhasználóbarátabban tudjuk megtervezni, azért, hogy a programozó sokkal hatékonyabban tudja kihasználni a benne rejlő lehetőségeket. A *TreeSelector* segít abban, hogy sokkal kézre állóbb legyen a szűrési feltételek meghatározása, ami azt eredményezi, hogy olvashatóbb, karbantarthatóbb kódot kapunk eredményül.

A *MetaCode*-ban rejlő lehetőségek szinte végtelenek, azonban mindenképpen el kell gondolkodni azon, hogy megfelelően legyenek implementálva a nyelvi eszközök. Ha túlságosan elengedjük a programozó kezét, akkor ő könnyen áteshet a ló túlsó oldalára, azaz mindent fordítási időben akar elvégezni. Fontos, hogy különböző megkötéseket, szabályokat fektessünk le a nyelvvel szemben, elérve azt, hogy a metaprogramozás hatékony és biztonságos paradigma legyen mindenki kezében.

5. Összefoglalás

Az informatikai világ ilyen szintű fejlődésével maguknak a szoftverfejlesztőknek is lépést kell tudniuk tartani. A felhasználói elvárások és igények folytonos változását nem elég követni, hanem reagálni is kell rájuk. Ezért elengedhetetlen az, hogy a szoftverfejlesztés egyes metódusait megpróbáljuk automatizálni, levenni a felelősséget a programozó válláról és valamilyen technológiával kiváltani azt.

A metaprogramozás ezt a feladatot hivatott megkönnyíteni azzal, hogy megpróbálunk olyan alkalmazásokat készíteni, amik újabb és újabb programokat tudnak előállítani akár felügyelet nélkül. A tendencia azt mutatja, hogy a fordítóprogramok tervezői kutatják, hogyan lehetne implementálni a metaprogramozásból ismert makrókat, szintaxisfa transzformációkat, kódgenerálást és ezeket milyen módon lehetne biztonságosan alkalmazni.

Ilyen programozási nyelv pl.: a *Scala*, ahol egyszerre több projekt is fut párhuzamosan és folyamatosan emelik be a sztenderd könyvtárba a már végleges megoldásokat. A Microsoft a *Roslyn* projekt[27] keretein belül a metaprogramozást, mint nyelvi szerviz szolgáltatásként szeretné megvalósítani és a *C#*-ban teljesen újraírt fordítóját lehet majd saját kiegészítőkkel bővíteni.

Diplomamunkámban a 2. fejezetben kitértem arra, hogy jelenleg milyen lehetőségek alapján lehet kihasználni a metaprogramozás egyes eszközeit. Láthattuk, hogy nem újkeletű irányzatról van szó, hiszen már a *C* előfordítóját is ennek tekinthető, de igazán az elmúlt pár évben jelentek meg olyan megoldások, amiktől igazán népszerű lehet. Egyes nyelvek, úgymint a *Boo*, a szintaxisfák manipulációját mint objektumok manipulációját fogják fel és ezen keresztül alakíthatjuk a saját forráskódunkat, míg a *Scala* ennél továbbmegy és az erősen típusos fák építésével próbálja meg kiszámíthatóbbá tenni a makrók használatát.

[27] Részletesebben a projekt weboldalán lehet róla olvasni: *https://roslyn.codeplex.com/*

A 3. fejezetben a metaprogramozást egy matematikai modellként definiáltuk és ennek segítségével elemeztük, hogy egy fordítóprogram implementációjánál milyen hibákba futhatunk és ezeket milyen módon tudjuk orvosolni. Megnéztük továbbá azt is, hogy milyen programozási paradigmákat, módszereket lehet helyettesíteni a metaprogramozás segítségével.

A 4. fejezetben egy saját imperatív nyelv került bemutatásra, ami a *CSS* szelektorokhoz nagyon hasonló deklarációs nyelv segítségével képes kijelölni részfákat a teljes absztrakt szintaxisfából és ezeket makrók segítségével képes manipulálni fordítási időben. A nyelv továbbfejlesztése számos irányban történhet, ugyanis ki lehetne egészíteni egy teljesen objektum-orientált nyelvvé és a makrók segítségével támogathatná a multiparadigma programozást. A *TreeSelector* nevű nyelv kifejezőerejét is lehetne bővíteni azzal, hogy más típusú szelektor operátor használatát támogatná, vagy akár megengedhetné azt is, hogy maga a felhasználó definiálhasson újakat a már meglévők mellé.

A diplomamunkám célja az volt, hogy bemutassam, hogyan lehet olyan eszközöket tervezni és implementálni egy imperatív nyelvbe, amik támogatják a szintaxisfa transzformációját fordítási időben a makrók segítségével és ezeket milyen módon lehet felhasználni a gyakorlatban.

6. Irodalomjegyzék

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

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

[3] Macro bundles,
http://docs.scala-lang.org/overviews/macros/bundles.html,
2014. május 04.

[4] The Scala Language Specification Version 2.9,
http://www.scala-lang.org/docu/files/ScalaReference.pdf,
2014. május 10.

[5] Denys Shabalin, Eugene Burmako, Martin Odersky: Quasiquotes for Scala, a Technical Report,
http://infoscience.epfl.ch/record/185242/files/QuasiquotesForScala.pdf,
2013. március

[6] Terence Parr, Sam Harwell, Kathleen Fisher: Adaptive LL(*) Parsing: The Power of Dynamic Analysis,
http://www.antlr.org/papers/allstar-techreport.pdf,
2014. március 24.

[7] Terence Parr: The Definitive ANTLR 4 Reference,
The Pragmatic Programmers,
2013. január 15.

[8] Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification,
http://www.w3.org/TR/CSS2/,
2014. május 10.

[9] Programming languages – C,
http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf,
2014. május 10.

[10] Kevin Hazzard, Jason Bock: Metaprogramming in .NET,
Manning Publications,
2013. január 7.

[11] ANTLR v4.0 Specification,
https://theantlrguy.atlassian.net/wiki/display/ANTLR4/ANTLR+4+Documentation,
2014. május 10.

[12] Douglas Crockford: Javascript The Good Parts,
O'Reilly Media / Yahoo Press,
2008. május

[13] Bertrand Meyer: Applying „Design by Contract",
http://dl.acm.org/citation.cfm?id=619797,
1992. október

[14] Mark Summerfield: Python in Practice,
Addison-Wesley Professional,
2013. augusztus 29.

[15] Russ Olsen: Design Patterns in Ruby,
Addison-Wesley Professional,
2007. december 20.

[16] Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts: Refactoring: Improving the Design of Existing Code,
Addison-Wesley Professional,
1999. július 8.

[17] R.S. Scowen: Extended BNF – A generic base standard,
http://www.cl.cam.ac.uk/~mgk25/iso-14977-paper.pdf,
1998. szeptember 17.

[18] Sinkovics Ábel, Porkoláb Zoltán: Expression C++ Template Metaprograms as Lambda Expressions,
http://abel.sinkovics.hu/download.php?fn=lambda.pdf,
2009. június

[19] David Abrahams, Aleksey Gurtovoy: C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond,
Addison-Wesley Professional,
2004. december 20.

[20] Laurence Tratt: Domain Specific Language Implementation via Compile-Time Meta-Programming,
http://dl.acm.org/citation.cfm?id=1391958,
2008. október

[21] Robert D. Cameron, M. Robert Ito: Grammar-Based Definition of Metaprogramming Systems,
http://dl.acm.org/citation.cfm?id=357235,
1984. január