MŰEGYETEM 1782

SZAKDOLGOZAT-FELADAT

Smikál Csanád (T5OHA2)
szigorló villamosmérnök hallgató részére

Magasszintű tesztspecifikációs nyelv PIL tesztekhez

A modern gépjárművek biztonságtechnikai és kényelmi funkcióinak megvalósításában, környezetvédelmi jellemzőinek javításában egyre jelentősebb szerepet kapnak a számítástechnikai megoldások. Ma egy prémium személyautó gyártójának közel száz elektronikus vezérlőegységből (ECU) és számos fedélzeti kommunikációs sínből kell kialakítani egy megbízhatóan működő elosztott rendszert, amely komoly algoritmus- és kommunikációtervezési, illetve munkaszervezési kihívást jelent.

Az alkalmazott vezőrlőegységek komplexitásának növekedésével egyre fontosabbá válik a több szintű, automatikus szoftver- és rendszertesztelés megvalósítása. Az egyik első szint, ahol már a teljes szoftver tesztelhető, a célkontrolleren az úgynevezett processor-in-the-loop teszt (PIL) teszt szint. Itt a célkontroller egy szimulált környezetbe kerül, és a külső eszközök (szenzorok, kommunikációs busz, stb.) helyettesítésre kerülnek szimulált eszközökkel. A piacon több ilyen rendszer is elérhető, de áruk és az igényelt speciális tudás gátat szab az elterjedésüknek.

Jelen feladat célja egy egyszerűen kezelhető, kis költséggel megvalósítható PIL rendszer egy részének megvalósítása. Már rendelkezésre áll egy hardver, melyen egy 32 bites kontroller látja el a környezetszimulációs feladatokat, de ezt még ki kell egészíteni egy szoftver környezettel, illetve egy magas szintű tesztleíró nyelvvel. A feladat megoldása az alábbi lépésekre bontható:

- Tipikus tesztlépések felmérése
- A tesztnyelv megtervezése, figyelembe véve az egyszerű tanulhatóságot
- A tesztnyelvhez egy fejlesztőkörnyezet létrehozása
- A tesztspecifikáció futtatókörnyezetének megvalósítása a mikrovezérlőn
- Specifikáció fordítóprogram írása (a magas szintű leírásból futtatható kód készítésére)

A feladat megvalósításához szükséges eszközöket a ThyssenKrupp Presta Hungary Kft. biztosítja.

Tanszéki konzulens: Dr. Sujbert László docens
Külső konzulens: Dr. Balogh András (ThyssenKrupp Presta Hungary Kft.)

Budapest, 2016. október 7.

.....
Dr. Dabóczi Tamás
tanszékvezető

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

1117 Budapest, Magyar Tudósok krt. 2. I. ép. I.E.444.
Telefon: 463-2057, Fax: 463-4112
http://www.mit.bme.hu • e-mail: mitadm@mit.bme.hu

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Smikál Csanád

MAGAS SZINTŰ TESZTSPECIFIKÁCIÓS NYELV PIL TESZTEKHEZ

KONZULENS

Dr. Sujbert László

Dr. Balogh András

(ThyssenKrupp Presta Hungary kft.)

BUDAPEST, 2016

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott **Smikál Csanád**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2016. 12. 9.

.....
Smikál Csanád

Összefoglaló

Jelen dolgozat célja egy olcsó, könnyen megvalósítható PIL tesztelő egység egy részének létrehozása. Ehhez először leírom, hogy mit is jelent ez, mire használják, és hogy az ipar mely területén lehet hasznos. Felmérem az ehhez kapcsolódó igényeket, és tisztázom a projekt lépéseit. Ezek után részletezem azokat, kezdve a magas szintű nyelvtan tervezési lépéseivel. Itt alakítom ki a nyelvtan szintaktikáját, valamint a különböző speciális utasítások lehetővé tételére is sor kerül.

A nyelvtan kialakítása után a projekt egy igen jelentős részét kitevő, futtatható kód generálás leírása következik. Itt részletezem, hogy a létrehozott nyelven megírt programot hogyan lehet egy, a kontroller számára is értelmezhető, hexadecimális számsorrá alakítani. Ehhez szükséges egy utasításkészlet definiálása, valamint a konkrét fordító mechanizmus létrehozása.

Ahhoz, hogy ez ténylegesen futtatásra kerüljön a céleszközön, szükség van egy kommunikációs modulra is, amelynek segítségével a generált kód letölthető a kontrollerre. Az 5-ös fejezetben ennek leírásával foglalkozom.

Végezetül betekintést nyújtok a projekt azon részébe, amely megvalósítására idő hiányában nem került sor. Szót ejtek majd a bővítési-, valamint módosítási lehetőségekről, és végezetül összegzem az elvégzett feladatokat.

Abstract

This thesis aims the creation of one part of a module, which can be used for writing and running PIL tests. First the definition of PIL testing is described and the part of the industry in which it can be used. Then the related demands are estimated and the steps of the project are clarified. After that the details of the solution, starting with the construction of the high level programming language are defined. In this part the syntax of the grammar and the specific commands are constructed.

The creation of the grammar is followed by the immensely significant description of code generation. The program – written in the created language – is transformed into hexadecimal number sequence, which is interpretable to the controller. This process is detailed in chapter 4. For this the definition of a set of commands, and the concrete mechanism of the translation is necessary.

Without a proper communication module, the hexadecimal byte code cannot be downloaded to the target consequently the execution of the program is impossible. The development of this module is described in chapter 5.

Finally those parts of the project, which were not created in lack of time are presented. In this part the expansion and modification possibilities are mentioned and the performed work is summarized.

1 Bevezetés

A technológiai fejlődés következtében egyre több használati eszközünket látjuk el programozható technológiával. Okosodnak a telefonjaink, az óráink, televízióink, sőt akár a teljes lakásunk. Ebből az autók is kiveszik részüket, egyre több olyan funkció kerül bennük megvalósításra, amelyek megfelelő működéséhez elektromos vezérlőegységekre (Electronic Control Unit – ECU) van szükség. Ezeknek – hogy az általuk alkotott rendszer hibamentes működését garantálni lehessen – komoly teszteken kell átesniük. A teljes szoftver célprocesszoron való tesztelésének egyik módja a Processor in the Loop (PIL) teszt. Egy ilyen PIL rendszer egy részének megvalósításába vágtam bele szakdolgozatom elkészítésének kezdetén. A feladatom az volt, hogy a rendelkezésre álló kontrolleren hozzak létre egy szoftver környezetet, amellyel az általa végzendő feladatokat irányítani lehet.

2 Az elvégzendő feladat ismertetése

A feladat tehát egy PIL teszteket végrehajtani képes rendszer egy részének megvalósítása volt. Ez egy olyan tesztelési metódus, amely során az elkészített szoftver már a célkontrolleren fut, amelynek környezete szimulálva van. A szimulált környezet azt jelenti, hogy állítani lehet például a különböző (analóg és digitális) be- és kimenetek értékeit, a kommunikációs buszon érkező adatokat, a rajta lévő szenzorok jeleit, tehát gyakorlatilag mindent, ami a processzor környezetéről információt szolgáltat, ezáltal lehetővé téve, hogy egy általunk meghatározott helyzetbe kerüljön [2].

Ezekre a feladatokra több szempont alapján lehet kiválasztani a felhasználni kívánt programozási nyelvet. Fontos, hogy – szinte – bárki számára könnyen tanulható legyen. Ez alapján tehát valamilyen magas szintű nyelvet érdemes használni, hogy a felhasználónak ne feltétlenül kelljen mélyre ható programozási ismeretekkel rendelkeznie. Az itt megemlíthető nyelvek (pl. Python, JAVA) azonban igencsak erőforrás igényesek, a tesztek végrehajtásáért felelős kontroller képességei pedig korlátozottak, így ezek támogatására nem alkalmas. Ebből a szempontból megközelítve a kérdést, a C nyelv tűnhet megfelelő választásnak. Képes az összes említett igény kielégítésére, a rendelkezésre álló erőforrásokkal kezelhető, azonban alacsonyabb szintű annál, hogy könnyen tanulható legyen.

Össze kell tehát hozni a két elvárást. A használandó nyelv legyen magas szintű, ugyanakkor kis erőforrás igényű. Ezen feltételeknek egy olyan nyelv tehet eleget, amely csak egy konkrét feladatra specializálódik (esetünkben PIL tesztek írására), így nem igényel nagy utasításkészletet, ezáltal sok erőforrást, azonban a felépítése és szintaktikája úgy van kialakítva, hogy használatához ne kelljen nagy programozási tapasztalattal rendelkezni. Mivel ilyen még nem létezik, létre kellett hoznom, hogy a tesztelést a lehető legnagyobb mértékben optimalizáljam.

2.1 Elvárások a nyelvtan felé

A létrehozandó nyelvnek tehát PIL tesztek írására kell alkalmasnak lennie. A minimálisnál több funkciót nem érdemes beletenni, hiszen a rendelkezésre álló erőforrás mennyiség véges. Ennek alapján a lehető legkisebb számú parancsot kell beleépíteni, amikkel azonban a megírni kívánt tesztek kivétel nélkül kielégíthetőek.

Először is, vegyük a specifikus utasításokat. A felhasználás során el kell érni a már említett (analóg és digitális) be- és kimeneteket, A/D átalakítókat, kommunikációs buszt, szenzorokat, stb. Ezekre az egy-egy speciális paranccsal való hivatkozás lenne a legjobb. Kelleni fog tehát egy csoport, amely az ezekhez szükséges utasításokat tartalmazza. Ez segít abban, hogy a felhasználó a lehető legegyszerűbb módon kezelni tudja a kontroller perifériáit.

Az ezen kívül létrehozandó elemek már kevésbé specifikusak, azonban a működés során létfontosságúnak bizonyulnak. Ide tartoznak azok a részek, amelyek másik, már létező nyelvekben definiálásra kerültek, így célszerű lenne azokból átemelni a szükséges utasításokat. Nagyon fontos azonban, hogy csak a megfelelő működéshez elengedhetetlen elemeket vegyük át, hiszen nem akarunk kihasználatlan részeket fölöslegesen belerakni a nyelvbe. Az egyik legfontosabb, és a könnyű felhasználást talán leginkább segítő ilyen átemelés a szintaktika volt. Az ugyanis a C és JAVA nyelvekhez nagyon hasonló, így nem kell újat megszokni.

A tesztelés során egészen biztosan szükségünk lesz függvényekre, azokat tehát mindenképpen implementálni kell a nyelvben. Ezek argumentummal történő hívása is elérhető kell, hogy legyen, amit a változók bevezetésével célszerű megoldani. A változókat lehet használni a különböző perifériákról beolvasásra került adatok tárolására is. Biztosan szükség lesz arra, hogy ezeket lehessen módosítani, így legalább a négy alapművelet implementálása elengedhetetlen.

Egy programozási nyelv sem létezhet cikluskezelés nélkül, természetesen nekem is lehetővé kellett ezt tenni. Itt vált nagyon fontossá az, hogy ezek más nyelvekből már ismerős szintaktikával kerüljenek megvalósításra. A ciklusokhoz szorosan kapcsolódik a feltételes utasításvégrehajtás, amely szintén helyet kapott a nyelvben.

Ezen feltételek kielégítésével tehát egy olyan komplex nyelvet kapunk, amely nem csak eleget tesz a bármely programozási nyelv felé támasztott minimális elvárásoknak, de olyan speciális utasításokat is tartalmaz, amellyel a PIL tesztekhez szükséges környezetszimulációs feladatok elláthatóak. Emellett kicsi az erőforrás igénye, ugyanakkor magas szintű, lehetővé téve a könnyen tanulhatóságot.

2.2 A fordító mechanizmus

Miután a nyelv készen van, egy fordító programot is kell hozzá írni, hiszen a létrehozott nyelv közvetlenül nem értelmezhető a kontroller számára. Egy olyan modulra van tehát szükség, ami a magas szintű nyelven megírt programból egy gépi kódhoz hasonló számsorozatot generál, melyet a későbbiekben a célprocesszorra írt program értelmezni tud.

A fordításhoz feltétlenül szükséges egy olyan utasításkészlet, amely elemeiből a nyelvben létrehozott összes parancs felépíthető. Nevezzük ezeket az elemeket byte kód utasításoknak, hogy a nyelv utasításaitól könnyebben megkülönböztethetőek legyenek. A megfelelő utasításkészlet létrehozása után tehát kell egy program, ami végiglépked az adott programon, értelmezi azt, és a megfelelő byte kód utasítások egymás után fűzésével létrehoz egy olyan sorozatot, amelyet a kontrolleren futó program értelmezni tud. Ez a köztes állapot ember által nehezen értelmezhető, ugyanis hexadecimális számokból áll, azonban a PC és a céleszköz közötti kommunikációt megkönnyíti, hiszen könnyen kezelhető a letöltő modul számára

2.3 A letöltő modul

A fordító által előállított gépi kód célhoz való eljuttatására létre kell hozni egy letöltő modult is. Ennek a feladata abból áll, hogy a generált kódot beolvasva megfelelő üzeneteket állítson elő, amit el tud küldeni a kontrollernek. Az üzenetküldés során ismernie kell a cél IP címét, így érdemes kialakítani egy kis felhasználói felületet, amelyen lehetőség van ennek megadására.

Fontos, hogy a kommunikáció mind a két irányba működjön, hiszen bár a lényegi adatot a PC felől küldjük, visszafelé is érkezik csomag, amely azt hivatott jelezni, hogy a kód letöltésre került. Ez azért fontos, mert ennek hiányában újra lehet küldeni az üzenetet, valamint esetleges hibák jelzésére is alkalmas lehet.

2.4 Feldolgozás a kontrolleren

A sikeres letöltést követően a magas szintű nyelven írt programból generált gépi kód már elérhető a kontrolleren. Létre kell tehát hozni egy olyan környezetet, amely ezt értelmezi, és futtatja. A korábban definiált byte kód utasításkészletet mindenképpen

változtatás nélkül kell itt is létrehozni, ez alapján értelmezhetőek ugyanis az egyes parancsok.

Ez bizonyos szempontból a fordító ellentéte, ugyanis „visszafordítja” a gépi kódot. A korábban említett speciális utasításokhoz itt kell olyan feldolgozó függvényeket létrehozni, amelyek végrehajtják a magas szintű nyelven csak egy-egy szóval leírt parancsokat. El kell tehát érni a különböző perifériákat, szenzorokat, A/D átalakítókat, stb. Mivel itt már a kontroller erőforrásait használjuk, ügyelni kell arra, hogy az elérhető korlátokon belül maradjunk.

A fejezetben leírtak alapján összeáll egy olyan rendszer, amely lehetővé teszi, hogy mélyebb programozási ismeretek nélkül is képes lehessen valaki működő környezetszimulációs teszteket létrehozni, csupán a megalkotott magas szintű nyelv elsajátításával.

3 A nyelvtan felépítése

A létrehozni kívánt nyelv megalkotásához létre kellett hozni a nyelvtant, ami a betartandó szabályokat tartalmazza. Ennek elkészítéséhez az Xtext nevű Eclipse kiegészítőt használtam. Ez a nyílt forrású eszköz nyelvtanok leírására alkalmas. Azért volt jó választás, mert a megírt nyelvtanból nem csak egy absztrakt szintaxis fát hoz létre, hanem az abban definiált entitások alapján egy JAVA osztályokból álló modellt is generál, ami később, a fordító modul megírásánál nagyon fontos szempont volt. További pozitívuma, hogy biztosít egy Eclipse alapú fejlesztőkörnyezetet is, a maga minden előnyével. Így kapunk hozzá rögtön szintaxis-ellenőrzést, színes szövegként vannak kiemelve a nyelvtanban definiált parancsok, és még lehetne sorolni az általa nyújtott lehetőségeket [1]. Annyi volt tehát a dolgom, hogy az Xtext szintaktikájának megfelelően írjak egy nyelvtant, ezek után automatikusan generálódik belőle egy Ecore modell, létrejön egy ANTLR elemző, fejlesztőkörnyezet, így ezek megfelelő módon való használatával a feladat végrehajtható volt.

3.1 Az alapok

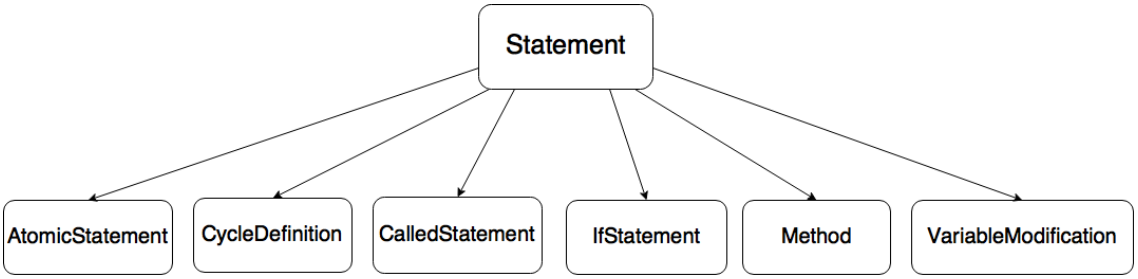
A fentebb leírt követelmények kielégítéséhez kellett tehát nyelvtant terveznem. Az alap koncepció az volt, hogy a nyelv álljon függvényekből, amelyek törzsét pedig utasítások alkotják. Az utasítások nem egyszerre kerültek implementálásra, hanem minden felmerülő igény esetén újabbakkal bővítettem a már meglévő listát. Ennek a módszernek nagy előnye, hogy a működést folyamatosan lehet tesztelni, így ha felmerül valami probléma, akkor lehet tudni, hogy melyik utasítás lehet érte felelős.

A változók hozzáadása tűnt a legcélszerűbb folytatásnak. Kezdetben csak egy fajta, egész számokat tartalmazó változók definiálására volt lehetőség. A későbbiekben ezt még kiegészítettem a logikai igaz/hamis értéket tartalmazó *boolean* típussal is. Ennek szükségessége a ciklus kezelésnél merült fel, ahol a logikai érték alapján való döntéshozás elkerülhetetlen, így – mivel a háttérben mindenképpen használni kellett – szintén elérhetővé tettem a nyelvtanban. Az említett ciklus kezelés egészen pontosan két fajta ciklust jelent, mégpedig a programozás világából ismerős *for* és *while* ciklusokat implementáltam, a maguk minden tulajdonságával.

Említettem, hogy a változók létrehozásának lehetőségével kezdtem a nyelvtan bővítését. Ez viszont szinte magában foglalja azok változtatásának szükségességét is, hiszen anélkül csak konstansokról beszélnénk. Ehhez lehetővé kellett tenni a változókon történő műveletvégzést is. A négy alap műveleten kívül (összeadás, kivonás, szorzás, osztás) még a bitenkénti jobbra- és balra shiftelést definiáltam. A változókhoz tartozik még az a kiegészítés, miszerint a létrehozott speciális utasítások, és a definiált függvények argumentumában is használhatónak kellett lenniük. Ez magában a nyelvtanban csak egy szintaktikai probléma, azonban a funkcionalitás és a kód feldolgozás során igen fontos kérdés.

3.2 Az utasításlista kiterjesztése

Az igények felmérése után kezdődhetett a nyelvtan elemeinek konkrét felvétele. Ez nem egyszerre zajlott, hanem folytonos bővítéssel, az új elemek tesztelésével, valamint a tesztek során felfedezett hibák javításával. A bővítések során bizonyos logika szerint jártam el, amely alapján a felvett utasítások és lehetőségek csoportokba sorolhatóak, ami a könnyebb átláthatóság szempontjából igen fontos lehet.



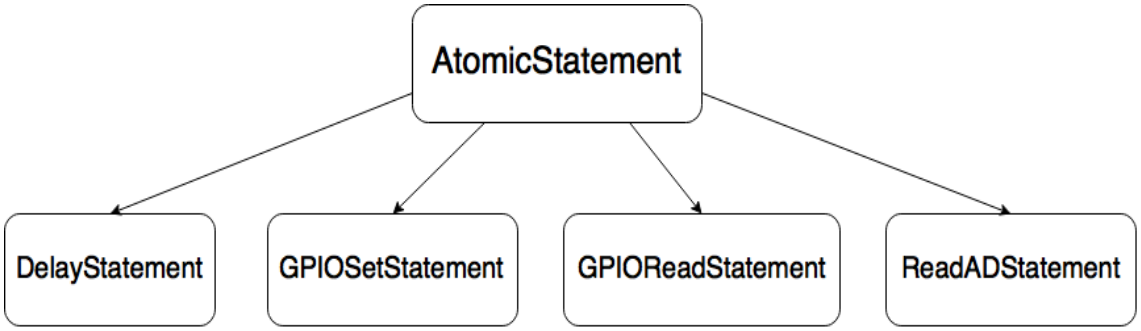
3.1 ábra: Az utasítások csoportosítása

A 3.1-es ábrán látható csoportokba osztva az utasításokat, könnyebben kezelhetőek, mintha mindegyiket közvetlenül a Statement ősből származtattam volna. Természetesen ezek még csak csoportok, nem konkrét parancsokat jelölnek, mégis fontos, hogy a felépítés logikáját követve vegyük végig az egyes csoportokat és tulajdonságaikat.

3.2.1 Alap utasítások

A 2-es fejezetben leírt specifikus szükségletek közül először a digitális be- és kimenetek megvalósítására koncentráltam. Ehhez hoztam létre az *AtomicStatement* és

CalledStatement csoportokat. A kettő gyakorlatilag ugyan azokat az utasításokat tartalmazza, annyi különbséggel, hogy míg az előbbinél konkrét számokat várnak argumentumként, addig utóbbinál változókkal lehet meghívni őket.



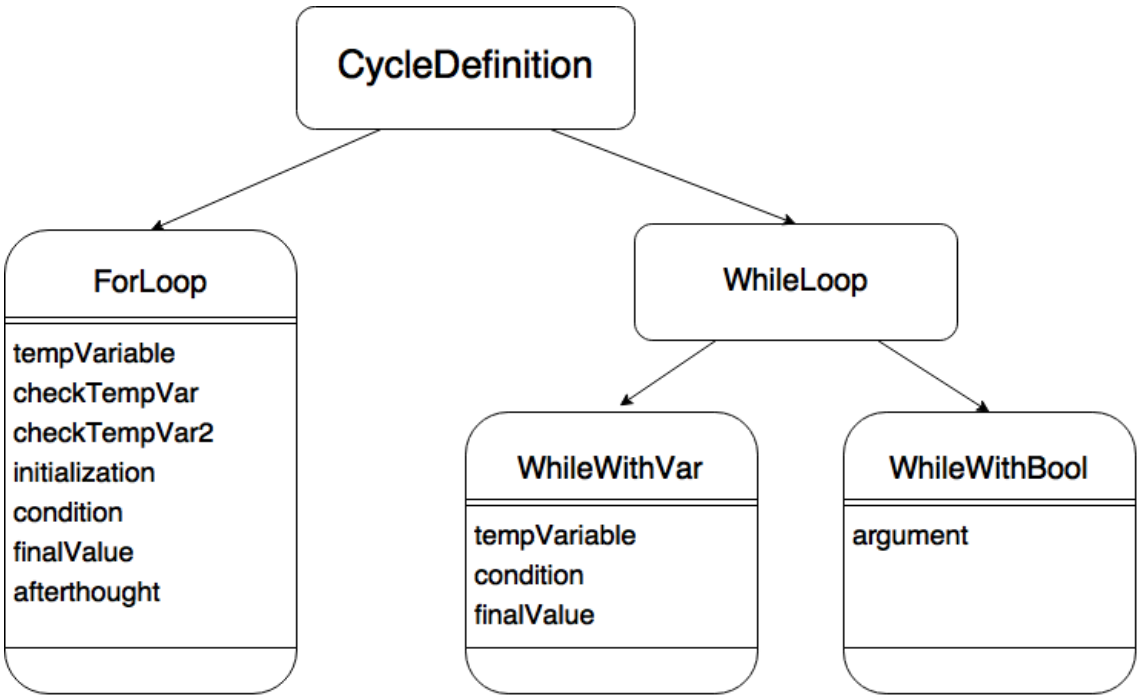
3.2 ábra: Az alap utasítások

Azért fontos külön entitásként kezelni a változóval hívott utasításokat a konkrét számmal hívott párjuktól, mert máskülönben nem lenne egyértelmű, hogy az adott parancs egyes tulajdonságaihoz mit várunk, hogy a felhasználó megadjon. Vegyük példának a késleltetést megszabó utasítást (*DelayStatement*). Alap esetben az egyetlen tulajdonsága az értéke, amit egy szám beírásával tud megadni a felhasználó. Ha azonban egy változó értékét akarjuk átadni, akkor annak nevét kell megadni, és az érték mező ez esetben már nem egy szám, hanem egy karaktersorozat lesz.

A 3.2-es ábrán láthatóak a jelenleg implementált specifikus utasítások. Ezekkel lehet kezelni a digitális be- és kimeneteket (*GPIOSetStatement*, *GPIOReadStatement*), az A/D átalakítókat (*ReadADStatement*), valamint késleltetést is be lehet állítani a programnak (*DelayStatement*). Mindegyik itt említett utasításnak van egy változóval meghívható párja is, a fentebb leírt okok miatt.

3.2.2 Ciklusok, feltételkezelés

Ciklusok közül a több programozási nyelvből is ismerős *for* és *while* került megvalósításra. Természetesen lett volna lehetőség hátultesztelő ciklus létrehozására is, azonban a cél az volt, hogy minél előbb legyen egy működő nyelvtan, így mivel a felhasználási lehetőségeket nem csökkenti, ezt nem raktam bele. A nyelvtan azonban bármikor kiegészíthető vele, így a későbbiekben, ha esetleg kényelmi okokból szükség lenne rá, bele lehet tenni.

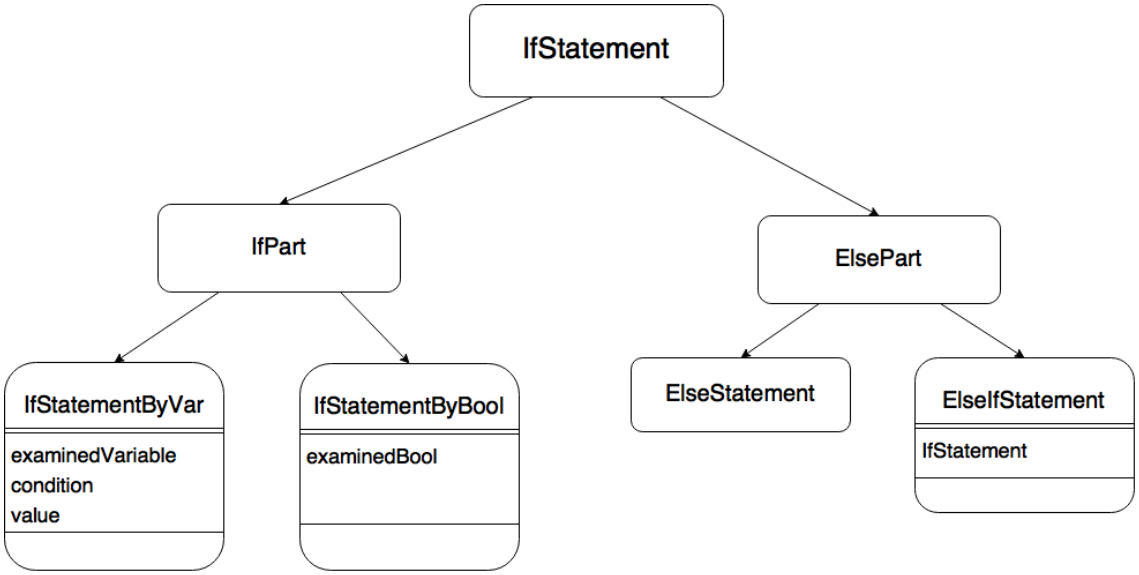


3.3 ábra: A ciklusok csoportosítása

A ciklusok működésében nincs eltérés az egyéb nyelvekben definiált testvéreiktől. Felépítésükben is megegyeznek, csupán némi magyarázattal szolgálnék arra, hogy mit jelentenek az egyes mezők. Mint ismert a *for* ciklus egy értékadással kezdődik, így az argumentum első része (*tempVariable*) az itt használt változó. Ennek értéke lesz összehasonlítva a *finalValue*-ként megadott értékkel. A másodikként és harmadikként leírt *checkTempVar* mezők hibadetektálásra szolgálnak. A 3.3-as fejezetben részletezett módon ezek segítségével lehet ellenőrizni, hogy mind a három helyen ugyanaz a változó lett-e megadva. Szintén az értékadáshoz tartozik az *initialization* mező, ami a kezdeti értéket jelenti. A *condition* azt a feltételt jelöli, amelynek beteljesüléséig végre fognak hajtódni a ciklus törzsében található utasítások. Végezetül az *afterthought* mező jelöli azt a műveletet, amely a ciklus minden lefutása után elvégzésre kerül. Fontos még, hogy a *for* ciklus ciklusvégének megadható két speciális operátor is (’++’ és ’--’), melyek az eggyel történő növelést-, illetve csökkentést hivatottak prezentálni. A *while* ciklus felépítése láthatóan ehhez nagyon hasonló.

A feltétel kezelő utasításnál és a *while* ciklusnál is látható, hogy alapvetően két csoportba soroljuk őket. Ez annak köszönhető, hogy kétféle szintaktikával adhatóak meg. Az egyik a *for* ciklusnál is definiált, *Condition* mezőbe írható feltétel, amely egy

változót hasonlít egy megadott értékhez. A másik pedig a *boolean* változók segítségével leírt feltétel, amely során az adott változó logikai értékét vizsgáljuk.

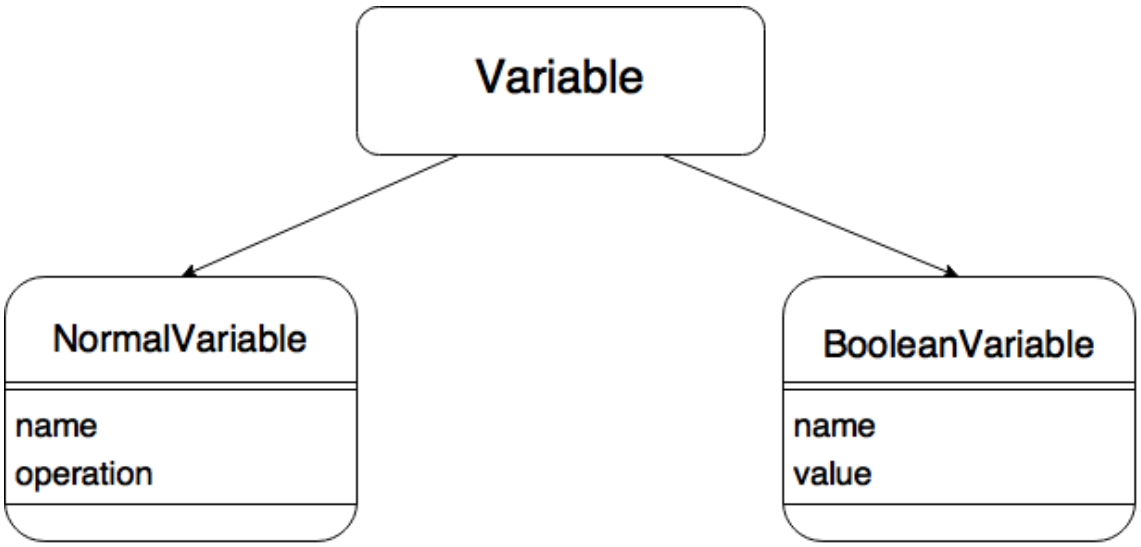


3.4 ábra: Az *if* utasítás felépítése

Az *if* utasítás érdekessége még, hogy az *ElsePart*-ban megadható, hogy a feltétel nem teljesülése esetén milyen utasításokat hajtson végre, valamint ez után fűzhető még egy feltétel. Ezzel gyakorlatilag létrehozható egy különböző eseteket kezelő, és ezekre más-más utasításokat leíró szerkezet. A megvalósítás során így maga az utasítás két részre lett osztva. Az *ifPart* egy egyszerű, korábban leírásra került feltétel kezelő utasítás. Az ez után következő *elsePart* viszont csak egy lehetőség, nem kötelezően felhasználandó. Ennek is két típusa van, attól függően, hogy akarunk-e még egy feltételt kezelni (*ElseStatement* és *ElseIfStatement*). Ha ugyanis akarunk, akkor az *'else'* kulcsszó után jöhet egy újabb *IfStatement*, amely ismét két résszel rendelkezik, amely végén szintén meg lehet adni egy újabbat, és így tovább, végtelen számú feltétel létrehozható egymás után fűzve.

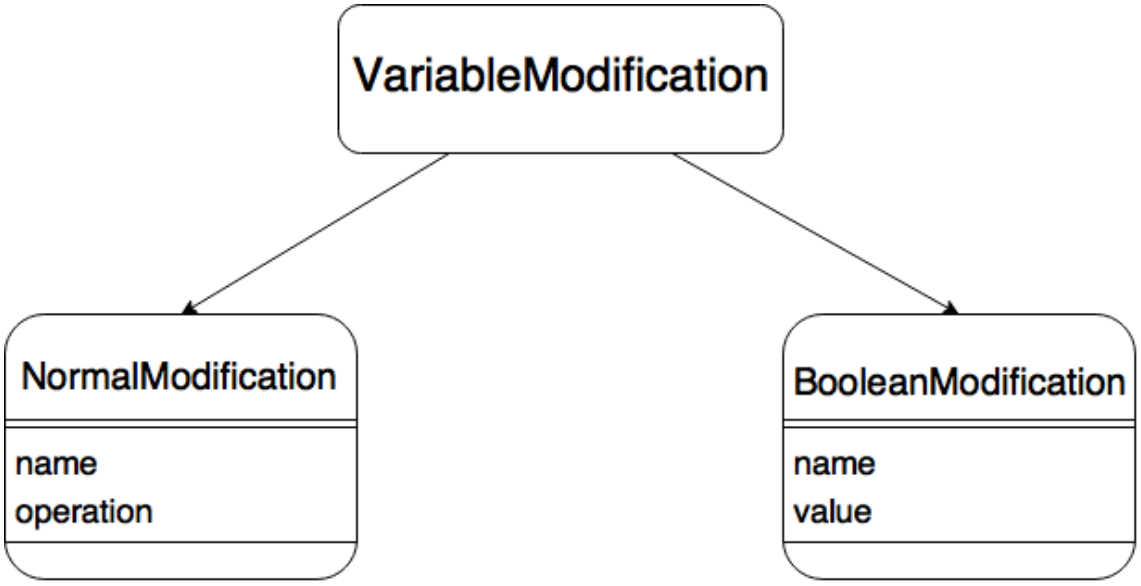
3.2.3 Változók definiálása és módosítása, műveletkezelés

Ahogy fentebb említettem, a nyelvtanban két fajta változó definiálására van lehetőség. Az egyikkel egész számokat tudunk tárolni, valamint hivatkozni rájuk. Ezeket nevezzük a nyelvben *NormalVariable*-nek, és használatukhoz a 'var' kulcsszó után a nevük, majd egy egyenlőségjel után egy műveleteket tartalmazó kifejezés segítségével kezdeti értékük adható meg. Definiáláskor kötelező az értékadás, de ez később természetesen változtatható.



3.5 ábra: A változók felépítése

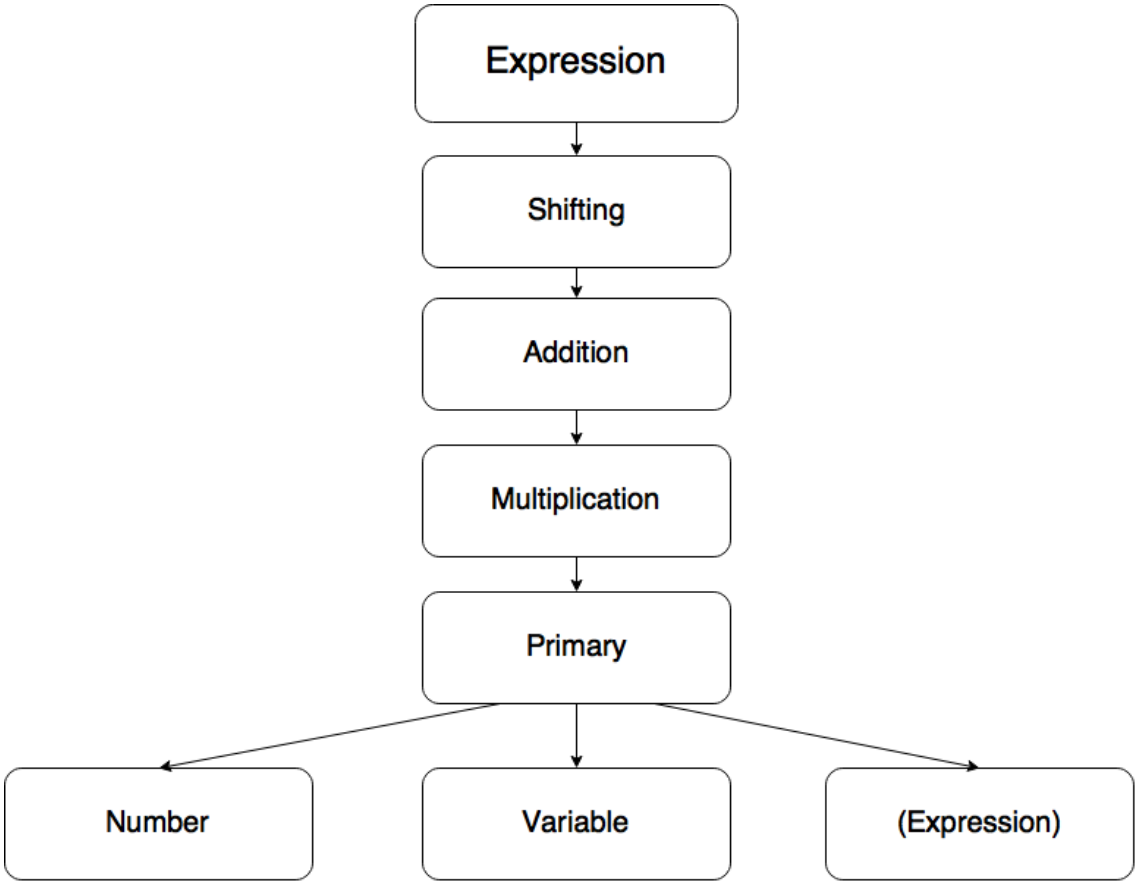
A másik fajta változótípus csak egy logikai érték tárolására alkalmas, neve *BooleanVariable* és a ’boolean’ kulcsszó után lehet megadni a nevét, majd egy egyenlőségjel után az értékét, ami igaz, vagy hamis lehet (’true’ / ’false’). Ezek a változók igazából csak feltételt kezelő utasításoknál (’if’), valamint a *while* ciklus paramétereként használhatóak, műveleteket végezni sem velük, sem rajtuk nem lehet, valamint függvények hívásakor sem használhatóak. Ellentétben a korábban említett *NormalVariable* típusokkal mindez megengedett. A változók módosítása a megszokott szintaktika szerint történhet (pl.: $variable = 1 + 2$), ahol az egyenlőségjel jobb oldalán állhatnak számok, de változók is.



3.6 ábra: A változók módosításának altípusai

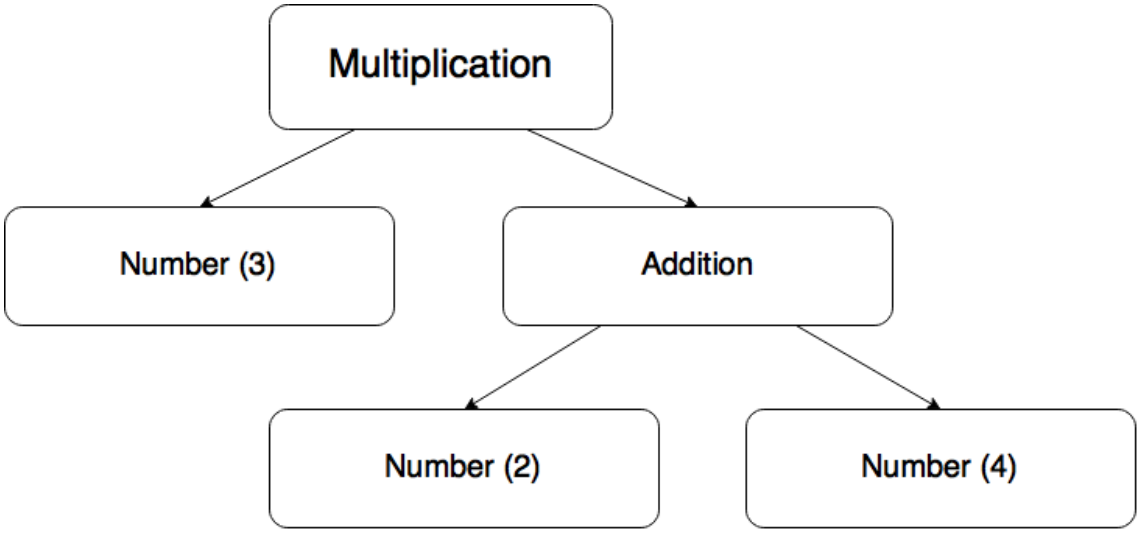
Érdekes kérdés lehet, hogy ha a nyelvtan csak egész számokat kezel a változókban, akkor mi a helyzet az osztás műveletével, ami mint tudjuk, az egész számok halmazára nem zárt művelet. Ezt a látszólagos ellentmondást viszonylag könnyű feloldani. A megírt kód ugyanis a célprocesszoron fog futni, ami pedig képes a tört számok kezelésére, így ha egy változó értéke nem egész lenne egy művelet elvégzése után, nem lesz semmi probléma, hiszen a változókra csak egy relatív címmel fog hivatkozni a program. A szabály tehát annyi, hogy műveletekben jelenleg nem lehet tört számokat használni. Ez a bővítés is megoldható természetesen, ám mind a nyelvtan kialakítása, mind pedig a kódgenerálás során viszonylag sok plusz munkát okozott volna, így ebben a verzióban ezt nem tettem elérhetővé.

A műveletek sorrendjének meghatározása szintén egy érdekes feladatnak bizonyult, hiszen azon kívül, hogy definiálni kellett egy hierarchiát a műveletek között, a zárójelezést is támogatni kellett, ami pedig ezt felül tudja írni. Mindemellett pedig figyelni kellett arra is, hogy a nyelvtan ne legyen balrekurzív, hiszen az Xtext által használt ANTLR azt nem tudja kezelni.



3.7 ábra: A műveletek felépítése

A változók módosításához létrehozott *NormalModification* entitás *operation* mezejébe egy *Expression* kerül. Ennek felépítését szemlélteti a 3.7-es ábra. Mint látható, ez alapvetően shiftelést tartalmaz, mint a legalacsonyabb rendű műveletet. A shiftelés a beírt műveleti jeltől függően történhet jobbra, vagy balra. Ez után a következő művelet az összeadás (és kivonás), itt is a beírt műveleti jel dönt arról, hogy melyiket végezzük. A legmagasabb szinten pedig a szorzás (és osztás) áll. Ezen műveletek jobb- és bal oldalát pedig egy általam létrehozott *Primary* típus képezheti. Ez lehet egy szám, egy változó neve, vagy pedig egy zárójelbe tett *Expression*, ami azt jelenti, hogy az egész kifejezés kezdődhet elölről, így végtelen számú műveletet megengedve. Az, hogy az építkezést a legalacsonyabb szintű művelettel kezdtem, azt eredményezi, hogy ha a fordító elkezd végiglépkedni a szintaxis fán, akkor először a legmagasabb szintű művelettel fog találkozni. A 3.8-as ábra szemlélteti, hogy a fordító hogyan dolgozza fel a következő műveletet: $3 * (2 + 4)$.



3.8 ábra: A műveletek fordító általi értelmezése

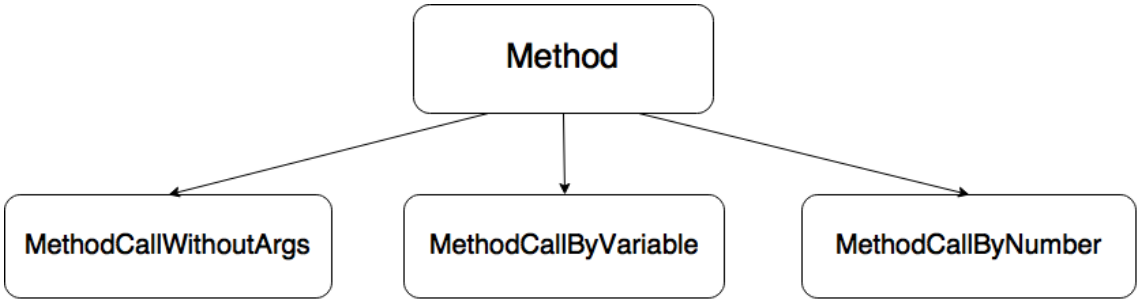
A fentebb vázolt értékadási procedúra természetesen csak *NormalVariable* esetén működik, egy logikai értéket tartalmazó *BooleanVariable*-re értelme sincs alkalmazni. Épp ezért a *VariableModification* típus is kettéválasztható egy *NormalModification*-re, aminek működését fentebb leírtam, illetve egy *BooleanModification*-re, ami pedig egyszerűen egy egyenlőségjel utáni értékadást foglal magában, ahol a lehetséges értékek az igaz és a hamis (true/false).

Változó módosítás esetén azonban észben kell tartani, hogy az egyes függvényekben létrehozott változók lokálisnak számítanak, így egy függvényen belül csak az ott létrehozott- és globális változókat lehet használni.

3.2.4 Függvényhívás

A függvényhívás utasításának definiálásánál egy dologra kellett odafigyelni, mégpedig arra, hogy nem mindegy, hogy az argumentumok helyére konkrét számokat, vagy pedig változók nevét írjuk, amikor meg akarunk hívni egy korábban létrehozott függvényt. Ezt a két esetet különválasztva a *Method* típus ketté bontható, egy *MethodCallByVariable* és egy *MethodCallByNumber* altípusokra. Az elsőnél a függvény neve utáni zárójelek közé karaktersorozatot vár a nyelvtan, míg a másodiknál decimális számokat. A kódgenerálás során előkerült problémák hatására azonban bevezettem még egy típust, ahol a függvény argumentumok nélkül kerül meghívásra. Ez nem feltétlenül lenne szükséges, de a fordító megírásánál könnyebbséget okozott.

Fontos megemlíteni, hogy maga a fejlesztőkörnyezet nem fogja figyelmeztetni a felhasználót, ha például egy létre nem hozott (esetleg másik függvényben létrehozott) változót akar berakni egy függvényhívás argumentumába, vagy esetleg egy még nem definiált függvényt szeretne meghívni. Ez a kódgenerátor dolga, ami viszont meg is teszi, és ilyen esetben nem generál kódot, nehogy hibás értékek kerülhessenek letöltésre.



3.9 ábra: A függvényhívás típusai

3.3 Kódvalidáció, hibakezelés a nyelvtanban

Az Xtext a felhasználó rendelkezésére állít egy kód validátor osztályt, amely feladata a definiált nyelven megírt kód átvizsgálása, és annak eldöntése, hogy minden az előre definiált szabályok alapján lett-e megírva. Itt ugyanis létrehozhatóak saját függvények, amelyekkel új szabályokat tudunk alkotni. Fontos még, hogy ez a validáció nem kíván mentést, így még a kódgenerálás előtt ellenőrzésre kerül a megírt program.

Én itt két saját szabályt definiáltam. Az egyik a ledek állításáért felelős *GPIOSet* utasítás érték mezejét vizsgálja. Annak ugyanis csak 0 és 1 értékek adhatóak. Ha ettől eltérő bemenetet kap, akkor pirossal aláhúzza a nem megfelelő kódrészletet, és egy hibaüzenetben leírja, hogy mi az, ami miatt nem kerülhet lefordításra.

A másik a *for* ciklusoknál említett futó változó ellenőrzés. Ott ugyanis az argumentum mindhárom részében meg kell adni a vizsgálni kívánt változó nevét. Ha ez a három nem egyezik meg, akkor szintén hibát jelez, hiszen nem egyértelmű, hogy melyik az, amelyiket valóban használni szeretné a kód írója. Erre természetesen a fordító is figyelhetne, de ez egy sokkal felhasználóbarátabb megoldás, hiszen rögtön észrevehető, hogy hol a hiba. Nem mellékes szempont továbbá az sem, hogy így biztosak lehetünk abban, hogy az elmentett (és így a kódgenerátornak átadott) kódban ilyen hiba nem fordul elő, így a fordítónak elég csak a legelső változót vizsgálnia.

3.4 A nyelvtanban definiált szabályok

A fentiek alapján létrehozott nyelvtannak vannak sajátosságai, amik bizonyos korlátok közé szorítják a felhasználót. Az első kettő, amikről még nem esett szó, szintaktikai szabályok, amelyek szintén jól ismertek lehetnek más programozási nyelvekből. Az első, hogy minden sor végét pontosvessző karakternek kell lezárnia, a második pedig, hogy a függvények, ciklusok és a feltételkezelő utasítás (illetve minden olyan utasítás, amely tartalmazhat további parancsokat) után kell állnia egy kapcsos zárójelpárnak, és ezek között kell megírni a függvény, vagy ciklus törzsét.

A következő szabály arról szól, hogy pl. egy függvényen belül előbb kell a változókat definiálni, és csak utána jöhetnek az utasítások. Természetesen megoldható lenne, hogy felváltva is lehessen használni ezeket, ám a nyelvtan funkcionalitását nem korlátozza, a kódgenerálás procedúráját viszont könnyebbé teszi, így a kényelmi szempontoktól eltekintve fölösleges változtatás lenne.

Az itt megemlített korlátozások a nyelvtan implementációjából fakadnak, így rájuk az Xtext által létrehozott fejlesztőkörnyezet figyelmeztet, tehát ezekre még a kódgenerálás előtt fény derül. Van néhány olyan szabály, amit viszont csak akkor lehet megvizsgálni, amikor már egy elkészült programot nézünk végig, így azok a kódgenerátor osztályban kerülnek ellenőrzésre. Ezekre tehát nem maga a fejlesztőkörnyezet fogja figyelmeztetni a felhasználót, hanem a mentés után meghívásra

kerülő kódgenerátor modul által dobott kivételek szövegei nyújthatnak némi információt az esetlegesen szükséges javításokhoz.

4 Kódgenerálás a nyelvtanhoz

A kódgenerátor osztály felel azért, hogy a felhasználó által magas szintű nyelven megírt programból egy, a céleszköz számára értelmezhető, hexadecimális számokból álló kód készüljön. Ehhez szükség volt egy utasításkészletre, amelynek segítségével a fentebb leírt magas szintű nyelven írt program lefordítható egy, a gépi kódhoz hasonló utasítássorozatra. A teljesség igénye nélkül, szükség volt például ugró utasításra (ciklusokhoz és függvényhíváshoz), feltételes utasításokra (szintén például a ciklusokhoz), valamint az alap utasítások végrehajtásához is definiálni kellett néhányat. A teljes lista elérhető a Függelékekben.

4.1 Az alapkoncepció

Hogy ne keverjük össze a nyelvtanban definiált utasításokat, és a kódgenerátorban létrehozottakat, utóbbiakra a későbbiekben túlnyomó részt byte kód utasításokként fogok hivatkozni, hiszen az ő segítségükkel generálódik byte kód a felhasználó által írt programból.

A lényeg tehát az volt, hogy a magas szintű kódból egy ilyen byte kód utasításokból álló sorozatot hozzak létre, amin a kontroller majd végig tud lépkedni, és végre tudja hajtani őket egyesével. A végiglépkedést elősegítendő, minden utasításnak rendelkeznie kell egy saját címmel. Ez egy 4 byte-os hexadecimális szám, amellyel egy adott parancsra hivatkozni lehet. Egy ugrásnál például ezt a címet megadva lehet elérni az ugrás célját. Ezen kívül minden utasítás típus rendelkezik egy egyedi operációs kóddal, ami alapján azonosítható, hogy milyen feladatot kell elvégeznie a céleszköznek, valamint utasításonként eltérő mennyiségű argumentumuk is lehet. Ezekről pontos információ a Függelékekben található.

A byte kód utasításoknak egy *ByteCodeStatement* nevű absztrakt osztály lett létrehozva, amiből az egyes utasítások származtathatóak, a fenti közös tulajdonságok azonban már ebben a közös ősosztályban definiálásra kerültek. Az ezekből álló listának pedig egy *ByteCodeModule* nevű osztály lett megfeleltetve, ami rendelkezik egy kódgeneráló függvénnyel, így tulajdonképpen a megfelelő byte kód utasításokat egymás

után rakva csak meg kell hívni ezt a függvényt és már generálódik is a megfelelő hexadecimális kód. A megfelelő utasítások megfelelő sorrendben történő egymás után helyezése, valamint az egyes argumentumok beállítása viszont már nem mindig volt ennyire triviális feladat.

A működés elve tehát úgy alakul, hogy az Xtext által generált osztályok segítségével lekérdezzük a program függvényeit, valamint ezek változóit és utasításait, majd ezekhez külön-külön kódot generálunk. Fontos, hogy előbb mindig a változók jönnek, és a kódgenerálás során el is mentjük őket, hiszen tudnunk kell, hogy melyek azok a változók, amikre később hivatkozni lehet. A függvényeken sorban megyünk végig, így azt már előre ki lehet kötni, hogy egy függvényt csak akkor tudunk meghívni egy másikból, ha már előtte az definiálva lett. Ebből következik persze, hogy a *main* függvény lesz utoljára létrehozva a programban, mégis ezt kellene a generált byte kód legelejére tenni, hiszen ennek a végrehajtását végzi el majd a kontroller, a többi csak akkor kerül végrehajtásra, ha innen meghívjuk. Ennek megoldására a Függvényhívás c. fejezetben részletesen kitérek majd.

4.2 Az utasítások kezelése

4.2.1 Alap utasítások

A nyelvtan definiálása közben ide sorolt utasítások annyira speciálisak, hogy mindenképpen létre kellett hozni nekik egy-egy külön byte kód utasítást is. Ezek argumentumai viszont elég egyértelműek. A fentebb definiált tulajdonságokat kell átadni, így például késleltetés esetén annak hosszát, vagy GPIO állítás esetén a pin azonosítóját és a beállítani kívánt értéket, stb. Ezekkel tehát különösebben sok dolgom nem volt. Kivéve a *GPIOReadStatement* és *ReadADStatement* esetében, ezeknél ugyanis egy változóba rakjuk bele a kiolvasott értékeket, így a byte kódban szerepelnie kell egy változó módosítását jelző utasításnak is, a maga minden tulajdonságával. Erről azonban később, a változók módosításánál részletesebben is lesz szó.

Ide tartoznak még továbbá a *CalledStatement* típusba tartozó utasítások, amiknél az argumentum viszont nem egy konstans szám, hanem egy változó. Itt annyi kiegészítést kell tenni az előzőekhez képest, hogy a bennük hivatkozott változót meg kell vizsgálni, hogy már korábban definiálva lett-e. Ha ugyanis nem, akkor hibát fog jelezni a fordító, amelyben tudatja a felhasználót arról, hogy nem talál létrehozott

változót a megadott névvel. Ez egy általam eszközölt biztonsági intézkedés, amelyből még több is szóba fog kerülni a későbbi fejezetek során. Ezek célja az, hogy csak egyértelmű és hibátlan generált kód esetén készüljön file, amit át lehet küldeni a kontrollerre, egyéb esetben hibát jelezve szakadjon meg a kódgenerálás végrehajtása.

4.2.2 Ciklusok, feltételkezelés

A két megvalósított ciklus egymáshoz igen hasonló, így nincs értelme külön tárgyalni őket. Különbség a *for* ciklus futó változó kezelésében van csak, így arra majd külön kitérek. Ami viszont mind a két esetben azonos, hogy mindkettő elöltesztelő fajta. Ez alapján a byte kódban egy feltételes ugró utasítással kell kezdődniük. Ennek meg kell adni a változó címét, amit vizsgálunk, az értéket, amihez hasonlítjuk, a kettő között fennálló relációt, valamint azt a címet, ahová akkor kell ugrani, ha a reláció nem teljesül. Ha ugyanis teljesül, akkor az adott ciklus törzsében leírt utasítások fognak végrehajtódni. Azt, hogy mi lesz ez a cím, egy ofszet kalkulátor függvény segítségével lehet kiszámítani. Ez működésében nagyon hasonlít a kódgenerátor függvényre, a szükséges utasításokat azonban nem gyűjti modulba, csupán a hosszukat összeadogatva kiszámolja, hogy hol lesz vége az adott utasítás blokknak, mielőtt a valódi kódgenerálásra sor kerülne. Az argumentum többi része már közvetlenül a felhasználó által megírt kódból kinyerhető, így azok beállításával különösebb feladat nincs. A reláció – a műveleti kódhoz hasonlóan – egy 1 byte-os hexadecimális szám, amely konstansként definiálva van mindkét oldalon, így a megfelelő helyre írva értelmezni tudja majd a feldolgozó egység is.

Miután a feltételes ugró utasítás részeit beállítottuk és hozzáadtuk a modulhoz, el kell menteni az ő címét is, hiszen miután egy rekurzív függvényhívással legeneráltuk a ciklusban található változókhoz és utasításokhoz tartozó kódot, ide fog kelleni visszaugrani – immár egy feltétel nélküli, mindenképpen végrehajtásra kerülő ugró utasítással. A *while* ciklus ezzel el is készült, hiszen ott a futó változót a benne lévő utasítások egyike kell, hogy változtassa. Ha ez nem történik meg, akkor elképzelhető, hogy végtelen ciklusba kerül a program, ám erre a felhasználónak kell figyelnie. *For* ciklus esetén azonban meg kell adni, hogy hogyan módosuljon a futó változó a ciklus minden lefutása után. Ez egy értékadás szintaktikájának megfelelő módon történhet, valamint definiálva van a ’++’ és ’--’ operátor is. Tehát az ennek megfelelő műveletet még el kell végezni, ennek byte kódja a feltétel nélküli ugró utasítás elé kell, hogy

kerüljön, hiszen ez a feltétel vizsgálat előtt kell, hogy végrehajtódjon. A műveletek kódgenerálásáról a 4.2.3-as fejezetben írok bővebben. Szintén oda tartozik az utolsó hátralévő feladat a ciklusok kódgenerálásából. Ez pedig a *for* ciklus futó változójának az argumentumban megadott kezdeti érték beállítása. Ezt ugyanis még az első feltétel vizsgálat előtt végre kell hajtani.

A fentiekkel megegyező elven történik a feltételt kezelő *if* utasításhoz való kódgenerálás is. Itt azonban akad még egy további teendő, mégpedig az *else* ágak kezelése. A feltételes ugró utasítás beállítása után meg kell vizsgálni, hogy a megadott utasítás *ElsePart* részének meg lett-e adva valami, és ha igen, akkor van-e utána további feltétel. Ezzel a két kérdéssel eldönthető, hogy tovább lehet-e lépni a következő utasításra, vagy az előző lépéseket ismételve további byte kód generálás szükséges.

4.2.3 Változók definiálása és módosítása, műveletkezelés

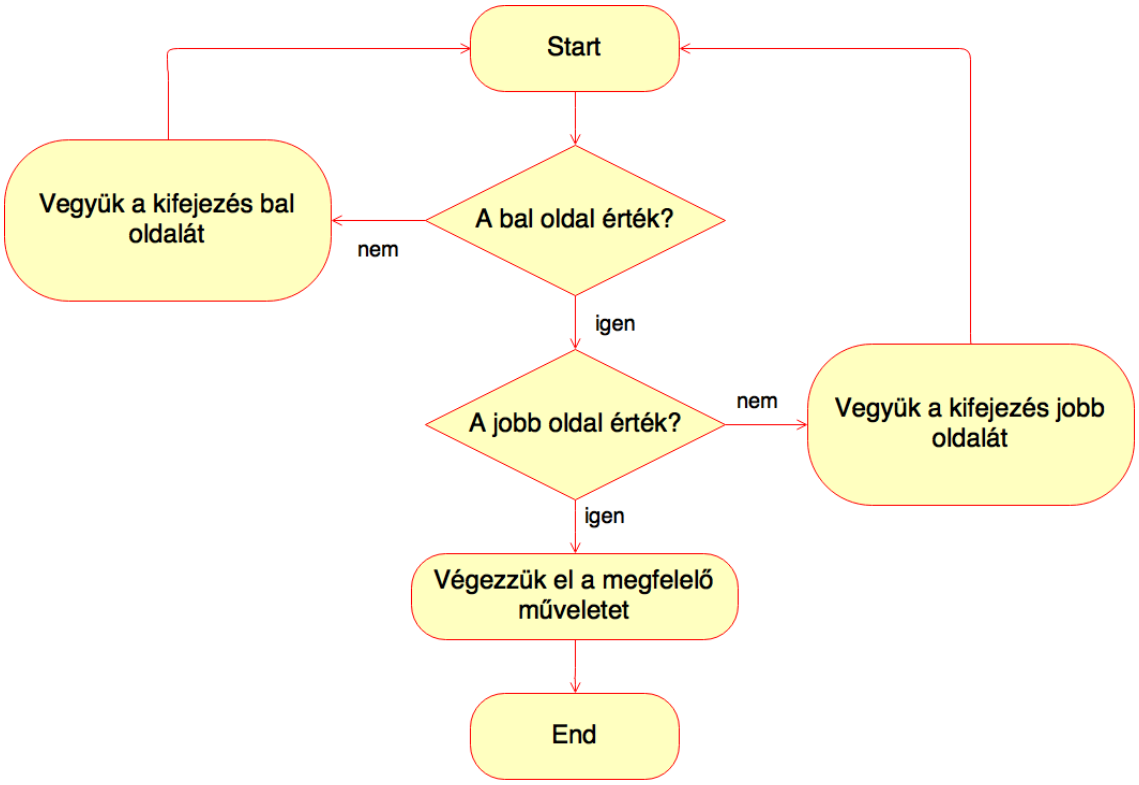
A műveletkezelés része a változók definiálásának és módosításának is, így célszerű azzal kezdeni. A nyelvtan felépítésénél leírtak alapján felépülő műveletek nagy előnye, hogy egy összetett kifejezést mindig a matematikai szabályok szerint utolsóként végrehajtandó művelettel kezd kibontani az elemző, és az elsőként végrehajtandó művelettel fejezi be, ezt a 3.8-as ábra jól szemlélteti.

Ezek alapján tehát meg kell vizsgálni ezen fa legalsó elemeit, amelyek már mindenképpen számértékek lesznek (a valóságban ezek lehetnek változók is, de az egyszerűség kedvéért maradjunk csak a számoknál). Ha ezek megvannak, akkor az első művelet már el is végezhető. Ez után egyet visszalépve vizsgálható a következő, amely azonban már nem feltétlen két megadott szám között kerül végrehajtásra, hanem az egyik operandus lehet az előző művelet eredménye is. Ezt úgy tettem lehetővé, hogy minden művelet eredményét elmentettem egy regiszterbe, amelyekre sorszámukkal lehet hivatkozni. Ez 0 és 127 között van, így ha ennél több részeredmény tárolására lenne szükség, akkor a fordító jelezni fogja, hogy elfogyott a rendelkezésre álló erőforrás-állomány.

Ilyen módon végiglépkedve a fenti fán, minden művelethez szükséges byte kód utasítás hozzáadható a modulhoz. Annak megkülönböztetésére, hogy az adott műveletet két szám, vagy regiszter (vagy esetleg változó) között kell-e végrehajtani, minden művelet típusú byte kód utasítás tartalmaz egy olyan mezőt, ahová egy előre definiált konstans értéket vár. Ez a kilenc féle konstans pedig egyértelműen megmutatja a

végrehajtó egység számára, hogy a művelet argumentumaként kapott hexadecimális szám értéket, címet, vagy regiszter sorszámot jelöl-e.

Ezzel a metódussal tehát feldolgozhatóak az összetett műveletek is. Az erre használt függvény egyszerűsített felépítését szemlélteti a 4.1-es ábra.



4.1 ábra: A művelet-értelmező függvény működése

A műveletekhez tartozó kód generálása tehát megtörtént, azonban még mindig nem egyértelmű, hogy ettől hogyan fog módosulni a változók értéke. Mivel a *BooleanVariable* típusra nincsenek műveletek definiálva, foglalkozzunk most a *NormalVariable* típussal. Az ezek definiálására létrehozott byte kód utasítás csak egy számot vár. Ide kell megadni a legutoljára végrehajtott művelet eredményét tartalmazó regiszter sorszámát. Ha csak egy értékadásról van szó, akkor csak egy értékadó utasítást küldünk, majd az ott megadott regiszter sorszámát adjuk a definiáló parancsnak is. Változók módosítása esetén nagyon hasonló a helyzet, annyi különbséggel, hogy a módosító byte kód utasításnak nyilván tudnia kell azt is, hogy melyik változót szeretnénk módosítani. Ezen információk beállítása után már módosítható a kívánt változó értéke.

Boolean változók esetén sokkal egyszerűbb a helyzet. Itt ugyanis nem szükséges regisztereket használni, hiszen csupán kétféle értéket vehetnek fel. Ezek szintén meg

vannak adva konstansként, így a feladat egy definiáló és egy módosító utasítással a fentiek szerint elvégezhető.

Itt érdemes megemlíteni a *ReadGPIO* és *ReadAD* speciális utasításokat is. Ezeknél ugyanis a beolvasott értékeket szintén egy változóba helyezzük el. A különbség csak annyi, hogy a használni kívánt regiszter értékét nem közvetlenül, vagy műveleteken keresztül állítjuk, hanem egy beolvasó paranccsal. Ennek megadjuk az elérni kívánt I/O periféria, vagy A/D átalakító azonosítóját, valamint azon regiszter sorszámát, ahova az értéket menteni akarjuk, és innentől pedig már a módosító utasítással állítható az adott változó értéke. Változókat definiálni ezzel a módszerrel nem lehet, hiszen a nyelvtan szintaktikája azt nem teszi lehetővé.

4.2.4 Függvényhívás

A nyelvtan kikötéseinél már említettem, hogy csak olyan függvényt lehet meghívni, ami már korábban definiálva lett. Ennek az oka, hogy a program függvényeinek egymás után, sorban generálunk kódot. Ha tehát egy olyan függvényt szeretnénk meghívni, ami még nem lett definiálva, akkor nem lehetne tudni, hogy mi a kezdőcíme, vagyis hogy hova kell majd ugrani. Ebből az is következik, hogy érdemes a main függvényt utoljára implementálni (ez ugyanis mindenképpen szükséges, hiányában a fordító hibát fog jelezni), ugyanis az ez után létrehozott függvényeket nem lehet meghívni, így létezésük értelmét veszti. Ennek következményeképpen viszont a main függvényhez generálódna utoljára kód, ami nem jó, hiszen ezt akarjuk végrehajtani, ennek a kezdőcíme kellene, hogy legyen a program kezdőcíme is (egészen pontosan, a globális változók utasításai megelőzik). Ennek áthidalására hoztam létre egy ofszet generátor függvényt. Ez szintén sorra veszi a függvény változóit és utasításait, de kódot nem generál hozzá, csupán kiszámolja, hogy mekkora eltolást eredményez az adott függvény a címképzésben. Ezt a main függvényre meghívva meg is kapjuk az először implementált függvény kezdőcímét, és innentől kezdve pedig már automatikusan történik az aktuális cím növelése.

Az így beállított eltolás után tehát már végig lehet lépkedni a program függvényein. Minden esetben elmentjük az épp vizsgált függvény kezdőcímét, hogy később, ha meg akarjuk hívni, akkor tudjunk rá hivatkozni. Ha ugyanis egy *Method* típus fordítására kerül sor, akkor többek között egy feltétel nélküli ugró utasítást is hozzá kell adni a modulhoz, aminek argumentumában a megfelelő függvény kezdőcíme

található. Ezen kívül – hogy a kért függvény végrehajtása után vissza is tudjon ugrani a program – el kell menteni azt a címet, ahonnan a program futásának folytatódnia kell majd. Ezt az adott cím verembe helyezésével lehet elérni. Ha ugyanis a meghívott függvény végére érünk, akkor a verem tetején lévő értéket kivéve már tudni is lehet, hogy hova kell visszaugrani ahhoz, hogy megszakítás nélkül, a megfelelő helyről folytatódjon a program futása. Ehhez azonban nem az épp aktuális címet kell elmenteni, hiszen a függvényeknek lehet argumentuma, amelyet szintén be kell helyezni a verembe, valamint ezek után egy ugró utasítás is következik, majd csak ez után jön az a cím, ahova vissza kell majd térni. Az argumentumok elmentése szintén a már említett verembe helyezéssel történik, annak megjelölésével, hogy az adott argumentum helyére egy konstans számot, vagy egy változót írtunk, így értéket, vagy címet mentünk el. Fontos még, hogy fordított sorrendben rakjuk bele a verembe ezeket az értékeket, hiszen az utolsóként belekerült elem lesz először kiszedve, tehát az első argumentumot kell utoljára hozzáadni, a visszatérési címet viszont legelőször. Ilyenkor természetesen a fordító megvizsgálja, hogy megfelelő mennyiségű argumentummal került-e meghívásra az adott függvény, és változóval történő hívás esetén azt is, hogy az esetleg nem *boolean* típus-e. Az ezekkel történő függvényhívás ugyanis nem engedélyezett a nyelvben.

Az argumentumok beolvasása során a veremből sorban kivételre kerülő értékeket belerakjuk egy változóba, és innentől már az adott argumentumra való hivatkozás erre a változóra fog mutatni. Így tehát minden függvény, ami argumentummal rendelkezik, és meghívható egy másik függvény által, annyi változó definiálással és értékadással kezdődik, ahány argumentuma az adott függvénynek van.

A függvényhívás tehát három részre bontható. Először egy verembe kell menteni a visszaugrásra kijelölt címet, valamint azokat a címeket/értékeket, amelyeket a függvény hívására használunk. Ez után jöhet a feltétel nélküli ugró utasítás, aminek célját a függvények kezdőcímeit tartalmazó listából lehet kiolvasni. Valamint ide sorolható még a visszaugrás is. Ez csupán annyit jelent, hogy a veremből kivesszük a legfelső értéket (ez minden esetben, amikor ilyen utasítás kerül sorra, a megfelelő cím lesz), majd az így kapott címre ugrunk. Ez utóbbi procedúra a *main* kivételével minden függvény végén megtörténik, hiszen azon kívül minden esetben folytatni kell a program végrehajtását. Ha azonban a *main* függvény végére érünk, akkor egy speciális, a lefutás

végét jelző parancs következik, amiből egyértelműen kiderül, hogy a program végére értünk.

4.3 A kódgenerálás során definiált szabályok

Korábban említettem néhány példát, amelyek a fordítás során hibát okozhatnak, aminek következtében pedig nem áll elő egy byte kódot tartalmazó file. Ezen hibák mindegyike tehát megszakítja a fordító program futását, amelyre azért van szükség, hogy semmiképpen ne generálódhasson esetlegesen hibás kód. Ha ugyanis egy ilyen hiba észlelése esetén csak figyelmeztetnénk a felhasználót, az nem jelentene teljes körű védelmet egy rossz fordításból fakadó meghibásodásért. Így viszont ezt a felelősséget nem a felhasználóra hárítottam, hanem én magam oldottam meg.

Több ide tartozó szabályt említettem már, de vegyük akkor most sorra, hogy milyen hibaüzenetek is fordulhatnak elő. Az első, és talán legfontosabb szabály, amelyre a fejlesztőkörnyezet nem figyelmeztet rögtön, az az, hogy csak olyan függvény kerülhet meghívásra, ami már a kódban korábban implementálva lett. Hasonlóan fontos, hogy a változókat sem lehet akárhogyan használni. Egy függvényen belül definiált változó csak az adott függvényben használható. A globális változók adnak lehetőséget arra, hogy egy-egy értékre az egész programot átfogóan lehessen hivatkozni. Ide tartozik még, hogy *boolean* típusú változókkal nem lehet függvényt hívni, sem őket műveletekben használni.

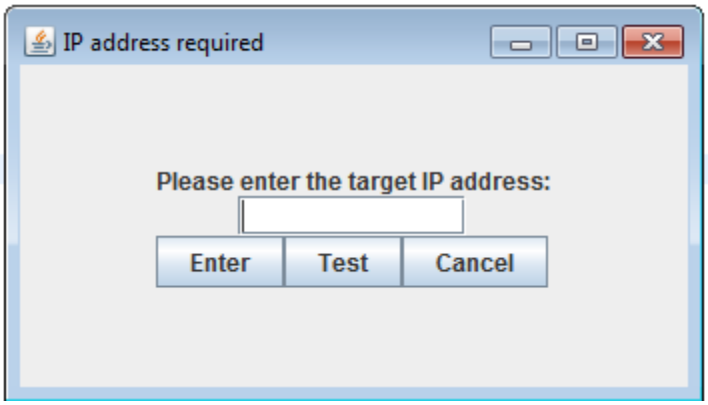
Egy kicsit kilóg a sorból az utolsó szabály, ugyanis ennek létezése nem teljes mértékben tudatos döntésen alapszik. Ez ugyanis arról szól, hogy egy függvény implementációja során, amennyiben az rendelkezik argumentumokkal, akkor azok nevei nem egyezhetnek meg sem korábban létrehozott, sem a későbbiekben definiálásra váró változók nevével. Ez egy tesztelés során észlelt hiba egy megoldása. Amikor ugyanis ez a szabály nem volt lekezelve, az általam választott változó kezelési megoldás miatt az azonos nevű változók és argumentumok egy kicsit összekeveredtek. Ennek javítása viszont több időt vett volna igénybe, így ezt az utat választottam. Természetesen van megoldásom a problémára, csak a jelenlegi verzióban az még nem elérhető.

5 Az üzenetküldő plug-in

Miután készen voltam a futtatható byte kód generálásával, következhetett a feladat talán legegyszerűbb része. Az önálló labor projektem keretein belül elkészítettem egy működő kommunikációt, amely UDP protokollt használva a PC részéről üzenetek küldésére képes, a kontrolleren pedig ezek fogadása, valamint egy sikeres vételről való visszajelzés küldése volt megoldott. Ezt felhasználva tehát csak annyit kellett elérnem, hogy a generált, byte kódot tartalmazó file-ból beolvasva az értékeket, csináljak belőle egy UDP üzenetet, majd ezt átadjam a már meglévő modulnak.

Az adatok beolvasásához a JAVA nyelv minden szükséges eszközt biztosít, így nem igényelt saját ötleteket. Talán annyit érdemes megemlíteni, hogy a byte kódot tartalmazó file a kód elemzését és megértését elősegítendő, el van látva néhány kommenttel is. Ezek azonban mindig gondosan egy ';' karakter után következnek, így ezek sorok végéről való levágása nem okozott gondot. A másik ide sorolható feladat a hexadecimális értékek decimálissá alakítása volt, ezt szintén nem tárgyalnám részletekbe menően.

Az üzenetküldést minél egyszerűbbé téve, azt egy Eclipse-ben létrehozott plug-in-nal oldottam meg. Ennek köszönhetően a felhasználónak csak jobb egérgombbal kell kattintania a byte kódot tartalmazó file-ra, ahol megjelenik a 'PIL test' opció, azon belül pedig a 'download to target'. Ezt kiválasztva egy általam létrehozott kis felhasználói felület fogadja, ahol a cél eszköz IP címét megadva letöltésre kerül a kód. Itt található egy 'Test' feliratú gomb is, amely a tesztelés során volt hasznos. Erre kattintva ugyanis az üzenet előáll, de nem kerül letöltésre, csupán megtekinthető annak felépítése. Így anélkül tudtam tesztelni az üzenet küldését, hogy fizikailag csatlakoztatnom kellett volna a PC-met a kontrollerhez.



5.1 ábra: A létrehozott felhasználói felület

Fontos megjegyezni, hogy az üzenet, a konkrét utasítások byte kódjain kívül tartalmazza saját maga hosszát, valamint azt is, hogy a letöltésre kerülő programban hány változó definícióra lesz szükség. Ezeknek a feldolgozás során lesz jelentőségük.

6 Összefoglalás, további lehetőségek

Miután a futtatható kód megérkezett az eszközre, az üzenet feldolgozása következett. Mint említettem, az első pár byte csak hasznos információkat tartalmaz, a konkrét kódhoz nem tartozik. Az üzenet hosszára azért volt szükség, mert így a feldolgozó egység tudja, hogy mire számítson, hogy mekkora memória területet kell lefoglalnia magának a kódnak. A változók számának megadása is pontosan ezt a célt szolgálja. A programon végiglépkedve ugyanis csak azt látja a feldolgozó, hogy jön egy újabb változó definíció, amelynek helyet kell foglalni a memóriában. Ha azonban előre megadjuk neki, hogy hány változót szeretnénk létrehozni majd, akkor előre tud foglalni egy akkora területet, és ebben az esetben akár az is lehetséges, hogy mint egy tömb elemeit kezeljük a változókat, sorszámuk szerint.

A szakdolgozatom készítése során tehát létrehoztam egy magas szintű nyelvet, amelynek segítségével PIL tesztekhez használt processzorok környezetét lehet szimulálni. A nyelvtan több helyen kiegészítésre szorul különböző specifikus utasításokkal, de jelen állapotában is működőképes. Ehhez készítettem egy fordítót, amely gépi kóddá alakítja a magas szintű nyelven megírt programkódot. Egy kommunikációs modul segítségével lehetőség van ezen kód célkontrollerre való letöltésére, ami után már csak annak feldolgozása van hátra. Itt kell értelmezni a létrehozott utasításkészletet, és megoldani a tényleges környezetszimulációs feladatokat. Ennek létrehozására idő hiányában azonban nem került sor. Az elképzeléseim alapján azonban igen hasonlóan működne a byte kód generáláshoz, csak fordítva. A beérkező utasításokon lépkedne tehát végig, azokat egyesével végrehajtva. Ezek a kiegészítések a jövőben viszonylag kis ráfordítással elvégezhetőek.

Több helyen is említettem, hogy a nyelvtan is úgy lett kialakítva, hogy rugalmasan bővíthető, módosítható legyen. Jelenleg biztosan nem teljes, hiszen az utasításlista nem terjed ki minden szenzorra, így nem lehet egy teljes környezetet szimulálni vele. Ez azonban szintén könnyedén megoldható. A nyelvtan módosítása esetén természetesen a megfelelő helyeken a kódgenerátor is változtatásokra szorul. Itt elképzelhető nagyobb volumenű beavatkozás szükségessége is, hiszen mint említettem, például a változók kezelése nem biztos, hogy a legmegfelelőbb módon történik. Ennek oka, hogy a *boolean* típus nem egyszerre került bele a nyelvtanba az egész számokat

tartalmazó változótípussal. Ennek következtében viszonylag sok problémám akadt vele, mire minden – nyilvánvaló és rejtett – problémát kiküszöböltem.

Az utasításlista kiterjesztésén túl bővíteni lehet a nyelvet például a tört számok kezelésével is, amennyiben erre szükség lesz. A függvények argumentumában történő műveletvégzés igénye már felmerült, természetesen ez is megoldható. Korábban említésre került, hogy függvények hívásánál kényelmesebb lenne, ha felváltva lehetne írni értékeket és változókat is az argumentumba. Ezeken kívül még a változó definiálások és az utasítások felváltva használhatóságának szükségét éreztem esetlegesen fontosnak, ami szintén nem követeli meg a nyelvtan teljes átszervezését. Ezek alapján látható, hogy az általam elkészített nyelv egyik nagy előnye, hogy a felhasználás során esetlegesen felmerülő igények kielégítésére mindenféle nagyobb erőforrások megmozgatása nélkül is van lehetőség.

7 Köszönetnyilvánítás

Szeretném megköszönni konzulenseimnek a témaválasztási lehetőséget, valamint az ezzel kapcsolatos segítséget, támogatást. Szintén köszönöm a ThyssenKrupp Presta Hungary kft. támogatását, hogy biztosította a szakdolgozathoz szükséges eszközöket.

Rövidítések jegyzéke

PIL – Processor in the Loop

ECU – Electronic Control Unit

GPIO – General Purpose Input/Output

ANTLR – Another Tool for Language Recognition

UDP – User Datagram Protocol

Irodalomjegyzék

[1] Xtext dokumentáció (http://www.eclipse.org/Xtext/documentation/index.html)

[2] Eckard Bringmann, Andreas Krämer - Model-based Testing of Automotive Systems (http://www.win.tue.nl/~mvdbrand/courses/sse/0809/papers/MBT.pdf)

Függelékek

Utasítás neve	Operációs kód	Argumetum(ok)	Leírás
NOP	0x00	-	Üres utasítás
JUMP	0x01	Cél címe	Feltétel nélküli ugrás
DELAY	0x02	Késleltetés hossza	Késleltetés megadása
GPIOSET	0x03	Pin ID, érték	GPIO állítás
GPIOREAD	0x04	Pin ID	GPIO olvasás
READ_AD	0x05	A/D ID	A/D átalakító olvasás
PUSH_VALUE	0x06	Tárolni kívánt érték	Érték stack-re mentése
PUSH_ADDRESS	0x07	Tárolni kívánt cím	Cím stack-re mentése
POP	0x08	-	Olvasás stack-ről
VARIABLE_ DEFINITION	0x09	Változó relatív címe	Változó definiálás
VARIABLE_ MODIFICATION	0x0A	Változó rel. címe, regiszter száma	Változó értékének módosítása
BOOLEAN_ DEFINITION	0x0B	Változó rel. címe, értéke	Boolean változó definiálás
BOOLEAN_ MODIFICATION	0x0C	Változó rel. címe, értéke	Boolean változó módosítás
POINTER_ DEFINITION	0x0D	Változó rel. címe	Argumentum definíció
IF_BY_VARIABLE	0x0E	Változó rel. címe, feltétel, érték, elérendő cím	Változóval megadott *if* feltétel
IF_BY_BOOLEAN	0x0F	Változó rel. címe, elérendő cím	Boolean értékkel megadott *if* feltétel
DELAY_BY_ VARIABLE	0x10	Változó rel. címe	Változóval megadott késleltető utasítás
GPIOSET_BY_VARIABLE	0x11	Változó rel. címe, érték	Változóval megadott GPIO állító utasítás
END_OF_PROGRAM	0x12	-	Program végét jelző utasítás
SHIFT_LEFT	0x30	Jobb oldali érték, bal oldali érték, regiszter száma,	Shiftelés balra

Utasítás neve	Operációs kód	Argumetum(ok)	Leírás
		módosító	
SHIFT_RIGHT	0x31	Jobb oldali érték, bal oldali érték, regiszter száma, módosító	Shiftelés jobbra
PLUS	0x32	Jobb oldali érték, bal oldali érték, regiszter száma, módosító	Összeadás
MINUS	0x33	Jobb oldali érték, bal oldali érték, regiszter száma, módosító	Kivonás
MULTI	0x34	Jobb oldali érték, bal oldali érték, regiszter száma, módosító	Szorzás
DIVIDE	0x35	Jobb oldali érték, bal oldali érték, regiszter száma, módosító	Osztás
SET_VALUE	0x36	Érték, regiszter száma	Regiszter értékének állítása
SET_VALUE_BY_VAR	0x37	Változó rel. címe, regiszter száma	Regiszter értékének állítása változóval

1. táblázat: A definiált byte kód utasításkészlet