PROGRAMOZÁSI NYELVEK SZINTAXISA ÉS SZEMANTIKÁJA, AZOK DEFINÍCIÓJA ÉS FORMALIZÁLÁSA

Farkas Ernő

Mint az közismert, a programozási nyelvek esetében szokás a nyelv szintaxisáról és szemantikájáról beszélni. Intuitíve azt mondjuk, hogy egy program szintaktikusan hibás, ha a leírásában olyan formai hibát vétettünk, amitől a nyelv fordítóprogramja nem fogadja el /hibajelzést ad/; és azt mondjuk szemantikusan hibás, ha mást hajt végre, mint amit várunk tőle. Mivel a program így vagy úgy, de mindkét esetben rossz, ez a körülmény alkalmas arra, hogy a szintaxis és a szemantika közötti alapvető különbségeket elfedje.

Kicsit pontosabban fogalmazva a fogalmakat így definiálhatjuk:

Szintaxisnak nevezzük a nyelv azon szabályait, amelyek alapján az adott nyelven helyes programokat írhatunk, vagy egy programról eldönthetjük, hogy helyes-e.

Szemantikának nevezzük a nyelv azon szabályait, amelyek megmondjak, hogy egy szintaktikusan helyes program milyen akciók végrehajtását jelenti.

A fentiekben egy nyelv szintaxisáról és szemantikájáról beszéltünk, de beszélhetünk egyetlen program szintaxisáról és szemantikájáról is, ez esetben arra gondolunk, hogy abban a konkrét esetben, hogyan érvényesülnek a fenti szabályok.

A definíciónak ezen a szintjén a szintaxis és szemantika között a következő különbségeket láthatjuk:

1. A szintaxisról lehet beszélnünk a szemantika ismerete nélkül, de fordítva lehetetlen.

2. A szintaktikus hibákról beszélhetünk önmagukban is, a szemantikus hibákról nem. Ez részletesebben a következőket jelenti. Ha ismerjük a szintaxis összes szabályát és egy konkrét programot, akkor megállapíthatjuk, hogy a program szintaktikusan, helyes vagy nem. Ha a szemantika összes szabályát ismerjük, akkor ebből meg nem tudjuk eldönteni, hogy a program szemantikusan helyes-e vagy nem, csak akkor, ha összehasonlítjuk azzal a feladattal, amit a programnak realizálni kellene. Ez viszont egy olyan külső fogalom, aminek a nyelvhez semmi köze.

3. A szintaxis statikus fogalom, a szemantika viszont dinamikus. Ezt úgy kell érteni, hogy egy programon önmagában ellenőrizni lehet a szintaktikus szabályok teljesülését, a program szemantikáját azt, hogy milyen akciók, milyen sorrendben mennek végbe, általában nem határozhatjuk meg csak a programból, hanem bizonyos bemenő adatokra is szükség van és ez a sorrend más és más bemenő adatoknál más és más lehet. Ezt az utolsó különbséget úgy lehetne kiküszöbölni, hogy a bemenő adatokat is a program részének tekintjük, azaz a szemantikát egy programból és egy adatból álló párokon értelmeznénk, mi azonban a továbbiakban nem ezt a megoldást választjuk.

Mielőtt tovább mennénk, foglalkozzunk egy kicsit azzal a kérdéssel, hogy mi a szintaxis és szemantika definiálásával a célunk. Ez a cél többrétű;

1. Össze akarjuk foglalni a szabályokat, amelyek alapján a célul kitűzött feladatot az adott nyelven megfogalmazhatjuk.

2. Azt akarjuk, hogy az így megfogalmazott feladatokat mindenki /emberek és számológépek/ megértse, és ugyanúgy értse.

3. Ezeket a szabályokat akarjuk felhasználni arra, hogy segítségével a nyelvet újabb és újabb gépekre implementáljuk.

Mint latjuk a probléma két irányban vetődik fel, egyrészt az emberek, másrészt a gépek irányában. Ezek közül egyelőre a gépek közötti kompatibilitás megteremtése látszik könnyebbnek, ugyanis az ember a felvetett problémákat az ő szokásos matematikai modelljébe képezi le, a gépek matematikai modellje viszont ettől különböző, „a gép más aritmetikával számol”. Ez a különbség egyelőre csak a felhasználók egy részénél tudatosul és csak igen kevesen képesek arra, hogy a szokásos matematikai modellről a gépi modellre áttérjenek, vagy közvetlenül arra fogalmazzanak.

Ennek az okai közé tartozik az is, hogy a gépi aritmetika fogalmai nincsenek teljesen tisztázva.

A másik kérdés, amivel foglalkoznunk kell, mielőtt tovább megyünk, egy terminológiai kérdés: mit nevezünk definíciónak és formalizálásnak. Mi a továbbiakban definíciónak nevezzük a következőt: feltételezzük, hogy vannak olyan egyszerű, mindenki szamara magától értetődő és mindenki szamara ugyanazt jelentő alapfogalmak, amelyek segítségével minden további fogalmat és állítást megmagyarázunk.

Formalisnak nevezünk egy definíciót akkor, ha a következő módon végeztük a definiálást:

1. **Megadunk egy jelölésrendszert, és megmondjuk, hogy a problémát az adott jelölésrendszerrel hogyan lehet leírni.**
2. **Az adott leíráson milyen formai változtatásokat kell vagy lehet elvégezni.**
3. **A kapott eredmény mit jelent.**

Nézzük meg például, hogyan alakul a szintaxis formalis definíciója a mondatszerkezetű grammatikák segítségével.

1. **Megadjuk a nyelv karakterkészletét és bizonyos segédszimbólumokat, továbbá olyan szabályokat, amelyek alapján, ha egy jelsorozat tartalmaz bizonyos részsorozatot, akkor ezt egy másik jelsorozattal helyettesíthetjük.**
2. **Képezzünk olyan jelsorozatokat, amelyek egy adott jelből levezethetők a fenti helyettesítésekkel, belőlük további jelsorozat már nem vezethető le, és nem tartalmaznak segédszimbólumot.**
3. **Az így definiált jelsorozatok lesznek a szintaktikusan helyes programok.**

Hasonlóan definiálható VDL-ben a nyelv szemantikája. Mindkét definíció kitűnően szemlélteti a formalizálás lényegét, de a későbbiekben láthatjuk, hogy a valóságot viszont nem adja vissza ilyen híven.

A formalis definíciónak az a jelentősége, hogy pontos és egyértelmű, ez azonban lényegében csak a 2. pontban foglaltakra vonatkozik, hiszen egy formalis leíráshoz és az abból kapott eredményekhez többféle jelentest is kapcsolhatunk.

Ezek után következzek a nyelv szintaxisának és szemantikájának pontos definíciója. Ez a definíció igen általános lesz, és éppen ezért az egyes konkrét esetekre nagyon keveset mond.

Legyen A a felhasználható karakterek halmaza.

X = F(A) Az A-ból képzett véges jelsorozatok halmaza.

Legyen I a lehetséges input adatok halmaza, 0 pedig a lehetséges output adatok halmaza.

Legyen Fio azon függvények halmaza, amelyek az inputadatokhoz az output adatokat rendelik.

L nyelvnek nevezzük X egy <u>rekurzív</u> részhalmazát.
Ekkor L szintaxisát X karakterisztikus függvénye adja meg.

$$\varphi(X) = \begin{cases} \text{igaz} & \text{ha } x \in L \\ \text{hamis} & \text{ha } x \notin L \end{cases}$$

Minden $x \in X$-re.

L szemantikáját pedig egy $\rho \epsilon [X \rightarrow Fio]$ leképezéssel adhatjuk meg.

$$\rho(x) = fio \qquad X \epsilon L \quad \text{és} \quad fio \in Fio.$$

Ha a fenti definíciót elfogadjuk, rögtön láthatjuk mennyivel egyszerűbb a szintaxis megadása a szemantika megadásánál. A szemantika megadásához ugyanis 3 dolgot kell ismernünk.

1. Azt, hogy milyen jelsorozat tekinthető egyáltalán programnak, azaz pontos szintaxist.
2. Azt, hogy milyen függvényeket kaphatunk értékül.
3. Végül, magát az összefüggést, ahogy a programhoz a függvényt hozzárendeljük.

Vizsgáljuk meg most azt, hogy mit mondhatunk az egyes pontokról külön-külön

A szintaxis formalizálásával kapcsolatban sok szakembernek az a véleménye, hogy az egy elintézett ügy: „hiszen minden programozási nyelv bizonyos értelemben CF nyelv, tehát a metanyelv jól leírja”. Mások viszont legalább ilyen joggal mondjak, hogy egyetlen programozási nyelv sem lehet CF. A helyzet ugyanis a következő a fordítóprogramok a szintaxis vizsgálatát két részre bontják, az egyik részben, amit szintaxis analízisnek neveznek, és amely a metanyelvvel jól leírható, azt vizsgálják, hogy a program milyen egységekre bomlik fel, ezek az egységek milyen további alegységekre és így tovább, lényegében nem tesznek mást, mint a program egyes részeit össze zárójelezik többszörös mélységben. Ezt a zárójelezést implicite beleértjük a programba, amikor megírjuk. Ezután a szintézis fázisában a programot a végrehajtás sorrendjében rendezzük át. Azaz először a legmélyebb szinten összezárójelezett részt kell kiszámítanunk, ha ez rendelkezésre áll, akkor a felette levő szinteket és így tovább. Igen ám, de nem elég megállapítani a programban szereplő elemek összetartozásának hierarchiáját, hanem meg kell állapítani, hogy a kifejezésben szereplő bizonyos alapvető elemekről, hogy a megfelelő helyen egy megfelelő elem áll-e, és a megfelelő formában áll-e. Általában elmondhatjuk, hogy a nyelvnek ezek az építőkövei bizonyos tulajdonságokkal kell, hogy rendelkezzenek, vagy nem szabad rendelkezniük vagy rendelkezhetnek. A fordítóprogramban a munka jelentős részét teszi ki annak a vizsgálata, hogy az adott kontextusban az adott elem milyen tulajdonságú, illetve hogy a kontextus hogyan változtatja meg a tulajdonsagait. Például igen sok nyelvben lehet többdimenziós tömböket deklarálni a deklaráció hatására feljegyzésre kerül, hogy hány dimenziós a tömb; és a tömb minden felhasználásánál ellenőriznünk kell, hogy ugyanolyan dimenziószámmal használtuk-e fel. Ezt metanyelvvel képtelenség leírni, bár maga ez a leírás is útmutatást nyújt arra, hogy hogyan lehetne az ilyen nyelvek szintaxisát jobban formalizálni. Elképzelhető persze olyan nyelv is, amelynél az egész zárójelezési eljárásnak, azaz a metanyelvnek nincs sok értelme. Képzeljük el, hogy a számológéppel bizonyos utasítássorozatot akarunk végrehajtatni, és ezt úgy írjuk le, hogy először leírjuk a műveleteket egymás után egy bizonyos szempontból, majd ugyanilyen sorrendben valamilyen más szempontból, és a fordítás során ezt a két listát elemenként összepárosítva dolgozzuk fel.

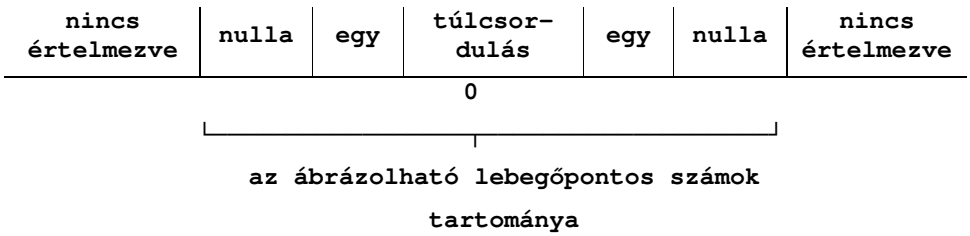
Egy másirányú bonyolult problémát vet föl az ALGOL 68, ahol az elemek tulajdonsagai szabjak meg a zárójelezés módját például olyan módon, hogy az operatorok prioritását az újradeklarálás segítségével megváltoztathatjuk.

Ez az egész felsorolás azt bizonyítja, hogy nem várhatjuk azt, hogy egy egységes minden nyelvre egyformán jó formalis szintaxist definiáló módszert találjunk, de azt várhatjuk, hogy minden egyes nyelvre vagy szerencsesebb esetben egész nyelvcsaládokra megtálaljuk a szintaxis formalizálásának módját.

Mivel a nyelvek bizonyos alapelemekből épülnek fel bizonyos összerakási szabályok segítségével a szintaxis definiálásának is ezt az utat kellene követni. Egy lehetséges út lenne például a metanyelvi leírás továbbfejlesztése olyan módon, hogy a nyelvet nem a karakterekig vezetnénk vissza, hanem bizonyos alapelemekre és formalizálnak az, hogy ezeknek milyen tulajdonságaik vannak, és azokat hogyan kell megvizsgálni.

A másik komoly feladat a szemantika definiálásának útján az, hogy megismerjük azokat a függvényeket, amelyeket egy-egy adott nyelven leírhatunk. A magas szintű nyelvek formalizmusa sok emberben kelti azt a téves hitet, hogy egy matematikailag jól leirt függvény számológépes kiszámításához nem kell mást tennie csupán a megfelelő képleteket, vagy algoritmusokat a nyelv szabályainak megfelelő formában leírni. Elfelejtik ugyanis, hogy a matematika valós és egész számokkal dolgozik, a gép viszont fixpontos és lebegőpontos számokkal. Néhány apró probléma ebből a témakörből: a műveletek nem asszociatívak a + (b + c) lehet, hogy túlcsordul és ugyanakkor (a + b) + c pedig nem, tehát a képletekben a zárójeleket más és más helyre helyezve a függvény értelmezési tartománya megváltozik. Ha egy szám létezik, nem biztos, hogy a negatívja is létezik, vagy a reciprokértéke is létezik, hiszen például az n jegyű bináris ábrázolás eseten 2n darab számot tudunk ábrázolni és ha ezt mind fel is használjuk, akkor világos, hogy nem lehet ugyanannyi negatív mint pozitív szám. Lebegőpontos számok eseten a szorzat akkor is nulla lehet, ha egyik tényezője sem nulla. A dolgot még súlyosbítja, hogy lebegőpontos számok esetében maga a lebegőpontos szám fogalma sem egyértelmű, hiszen például van kerekítéssel dolgozó aritmetika és letöréssel dolgozó aritmetika. Ez a különbség nemcsak azt okozhatja, hogy ugyanaz az algoritmus enyhen különböző eredményeket ad a két különböző üzemmódban, de szerencsétlen esetben ez a különbség szignifikáns is lehet, sőt az is előfordul, hogy az egyik üzemmódban az algoritmus működik a másikban pedig nem. Hasonló problémát vet fel az a kérdés, hogy mit csináljunk a legkisebb karakterisztikával még ábrázolható, de nem normalizálható számokkal.

Nagyon teoretikusan hangzanak, de gyakorlati szempontból fontosak a következő kérdések is: mit nevezzünk a lebegőpontos számok körében konvergenciának, mikor folytonos egy függvény, a legközönségesebb függvények polinom, törtfüggvények, iteratív úton előállított függvények folytonosak-e. Könnyen látható például, hogy $f(x) = 1/x^2 \cdot x \cdot x$ függvény a valós számok körében lényegében azonosan egy. Ugyanez a függvény a lebegőpontos számok körében valahogy így néz ki:



Azt mondhatjuk a függvények megadásáról, hogy tulajdonképpen két módszert képzelhetünk el. Az elsőt "matematikai megadás"-nak nevezhetjük, ilyenkor valamilyen matematikai módszer segítségével egész pontosan definiáljuk azt, hogy milyen objektumok a nyelv alapelemei, azokkal milyen tevékenységeket végezhetünk el, de nem foglalkozunk azzal a kérdéssel, hogy lehet-e az így definiált függvényeket számológépen megvalósítani, hogyan lehet, és hány féleképpen lehet. Ennek a módszernek az a lényege, hogy feltesszük azt, hogy az így adott definíció olyan pontos és olyan egyértelmű, hogy bármilyen realizációt választunk is ki az alap epilőelemekre, ennek a függvényekre semmilyen mellékhatása nem lesz. A másik módszert "gépi megadásnak" nevezhetjük, ilyenkor olyan adattípusokból és utasításokból indulunk ki, amelyek általában minden gépen léteznek, és ezekből építjük fel a függvényeinket. Bár ez a megoldás a gyakorlat szempontjából kézenfekvőbb az előzőnél, de egyértelmű függvény definíciókat csak úgy kaphatunk, ha az alapfogalmaknak pontos matematikai definícióját rögzítjük le.

Végül beszélnünk kell arról is, hogyan lehet a szemantikát definiáló ρ leképzést megadni. A három közül eddig ezen a területen tettek meg messze a legtöbb lépést. A ρ leképzés megadása két részből áll, egyrészt meg kell adnunk a nyelv alaputasításainak megfelelő függvényeket, másrészt meg kell adnunk, hogy az utasítások összekapcsolása a függvények milyen összekapcsolását jelenti.

Egy másik szempontból a szemantikát vagy interpreterrel vagy transzlatorral lehet megadni.

Interpreternek nevezünk egy olyan ρ^- függvényt, amely I x X-en van értelmezve és értékkészlete O. Ahol I a lehetséges input adatok halmaza O a lehetséges output adatok halmaza L pedig a nyelv. A $\rho^-(i,x)$ függvény pedig ($i \epsilon I$, $x \epsilon L$) pedig egy olyan leképzés, amely egy adott x programhoz és a hozzátartozó i adathoz a program végrehajtása során keletkező outputot rendeli.

A transzlator definíciója lényegében azonos a ρ leképzés definíciójával, azaz:

$$\rho(x) = fio \qquad X \epsilon L \qquad fio \in Fio.$$

Interpreter esetében egy konkrét X program szemantikáját kifejező fio függvényt úgy kapjuk meg, hogy a konkrét x-et behelyettesítjük a ρ^- interpreterbe az i inputot pedig végig futtatjuk az s halmazon.

Mind az interpreternek, mind a transzlátornak vannak előnyei a másikkal szemben. Az interpreter esetében nem kell definiálni az Fio függvényosztályt, a transzlator esetében pedig nem kell foglalkozni az input adatokkal.

Intuitíve úgy érezzük, hogy az interpreter jól definiálja a nyelv szemantikáját, de sokkal rosszabbul, kevésbé áttekinthetően az egyes konkrét programokét.

Mindkét esetben a szemantika megadása a ρ , illetve ρ^- megadásával történik. Hogy adhatjuk meg ezt a függvényt? Ezt a függvényt valamilyen nyelven Írjuk le. Ez a nyelv lehet valamilyen természetes nyelv pl. angol, valamilyen matematikai formalizmus, vagy valamilyen programozási nyelv, vagy ezek keveréke. A nyelv szemantikájának a formalizálásának esetében a definiáló nyelv mindig egy matematikai formalizmus vagy egy olyan programozási nyelv, amely a szemantikája nyilvánvalóbb, mint a definiálandó nyelve. A szemantika definiálásának esetében is érvényesül a kettősség: itt is beszélünk „matematikai" – és „gépi definícióról".

Transzlátorok esetében az Fio függvény osztályt is le kell írnunk ennek a leírásnak a nyelve lehet azonos is a ρ-t leíró nyelvvel de lehet különböző is.

A következőkben megpróbálom csoportosítani a szemantika definiálására irányuló módszereket:

Matematikai definíciók:

1. Bitenként leíró módszerek

Ezek a módszerek abból indulnak ki, mivel az egyes számológépek mindig más gépre nem jellemző utasításkészlettel, memóriakapacitással, címzési módokkal stb. rendelkeznek, először meg kell alkotni egy általános számológép modellt valamilyen Turing-gép, absztrakt automata vagy formalis rendszer formájában, és ezek után egy nyelv szemantikáját úgy adhatjuk meg, hogy a nyelv programjaihoz az absztrakt számológép programjait rendeljük hozzá. Mivel a leírás bitsorozatok vagy jelsorozatok transzformációjából áll az így adott leírás tokéletesen egyértelmű és rendszerint alkalmas arra, hogy matematikai tételeket bizonyítsunk ilyen módon. Gyakorlati felhasználásukra nem igen van példa, mért a használt formalizmus: tömeg emberi és gépi fogyasztásra egyaránt alkalmatlan. Megjegyzendő meg, hogy így definiált nyelvek, vagy nyelvi jelenségek inkább a matematikai példa illusztrálására szolgálnak, mint a gyakorlati programozás céljára.

2. Állapottranszformációs módszer

Az állapottranszformációs módszer a nyelvet utasításokból építi fel, az utasítás szemantikus tartalma egy absztrakt állapothalmazon végrehajtott transzformáció. Az utasításokból bizonyos kifejezéseket képezhetünk, azokból meg újabbakat, míg végül egy programot kapunk. Ez a módszer megmutatja, hogy hogyan építhetjük fel egy program állapottranszformációját az egyes utasítások állapottranszformációjából anélkül, hogy magukról az állapotokról vagy a transzformációkról valami konkrétat tudnánk. Ez a módszer inkább elméleti eredmények létrehozására alkalmas, ezek az eredmények azonban közeli kapcsolatban állnak a gyakorlattal.

Ezt a módszert alkalmazta Dana Scott és Christopher Strachey (2). Ez az írás azért is érdekes, mert bátran szakítottak sok olyan hagyománnyal, amely inkább akadályozta a munkát, mint segítette.

A következő módszer már félig a gépi módszerek közé tartozik.

3. A Lambdakalkulus és az applikatív nyelvek

A lambdakalkulus egy olyan matematikai formalizmus, amely annak leírására szolgál, hogy hogyan képezhetünk kifejezéseket, a kifejezésekből függvényeket, hogyan helyettesíthetünk ilyeneket egymásba, hogyan definiálhatjuk a kifejezésekben szereplő változók hatáskörét. Mint azt Church bebizonyította, a lambdakalkulus algoritmikusan egyenértékű a rekurzív függvényekkel.

A fentiek alapján a lambdakalkulus rendkívül alkalmasnak látszik a szemantika definiálására. Az alkalmazás során azonban problémák merültek fel, melyek megértéséhez újabb fogalmak bevezetésere van szükség.

Egy nyelv applikatív tulajdonsagainak nevezzük azokat a lehetőségeket, hogy kifejezéseket képezhetünk, függvényeket deklarálhatunk, ezeket alkalmazhatjuk kifejezésekben, és hogy kifejezéseket kiértékelhetünk.

A nyelv imperatív tulajdonsagainak nevezzük az utasítások sorrendjét, az ugrásokat, értékadásokat és függvények side-effectjeit.

A gépi kód teljesen imperatív jellegű nyelv. A magasabb szintű nyelvek között vannak teljesen imperatívak (iPLA) és teljesen applikativak (LISP 1.5) általában azonban a kétféle jellemző egyszerre tálalható meg.

Az imperatív jellemzők általában a programozási nyelv hatékonyságát, az applikatív tulajdonságok a kényelmes programozást segítik elő. A csak applikatív vonásokat tartalmazó nyelveket applikatív nyelveknek nevezzük.

Az applikatív nyelveket a lambdakalkulus „szintaktikusan cukrozott" változatának nevezik, ezt úgy kell érteni, hogy egy applikatív nyelv lényében azonos a lambdakalkulussal csak a leggyakoribb kifejezésekre egyszérűsítő jelöléseket vezetnek be.

Más a helyzet az imperatív jellemzőkkel. Egyrészt problematikus az utasítások sorrendjének, ugrásoknak a leírása, ezt azonban meg lehet oldani valahogy (ll, 3, 4, 5, 8) applikatív nyelven is.

A nagyobb probléma, hogy a legtöbb imperatív nyelvben lehetőség van arra, hogy egy mennyiségre több különböző néven is hivatkozzunk. Ezért, ha egy applikatív nyelven irt interpreterrel akarjuk a szemantikát definiálni, akkor ebbe a modellbe egy memóriamodellt is bele kell építeni, amely képes ezt a lehetőséget biztosítani.

Ugyancsak az imperatív tulajdonságok vetik fel azt a problémát, hogy a nyelv tisztán applikatív részei is imperatív vonásokat vesznek fel. Pl. egy függvény érteke függ az argumentumainak kiszámítási sorrendjétől.

Ehhez hasonlít az applikatív nyelveknek a problémája, amely viszont már nincs összefüggésben az imperatív vonásokkal, hogy a függvények paramétereit értek szerint vagy név szerint hívjuk-e.

Az applikatív függvények körében beszélhetünk név szerint és értek szerinti paraméterhívásról és ezeket a következőképpen értjük. Adott egy függvény, amelyet egy bizonyos helyen ki akarunk számolni, a függvényt egy kifejezés adja meg, amelyben a függvény paraméterei változóként szerepelnek. Pl.

$$f(x,y,z) = a^2x+by + z$$

Ha most ezt a függvényt egy olyan helyen akarjuk kiszámítani, amit aritmetikai kifejezésekkel adunk meg /amely lehet egyetlen változó is/, pl. f(u + v, v, uv), akkor értek szerinti hívás esetén először kiszámítjuk a kifejezések értékeit, majd ezek segítségével a függvénynek megfelelő kifejezést olyan módon, hogy a megfelelő változó helyen a megfelelő értékkel számolunk. Név szerinti hívás eseten a kiértékelést a függvénynek megfelelő kifejezés kiértékelésével kezdjük, és valahányszor egy paraméterre szükségünk van a kiértékelést felfüggesztjük, kiértékeljük a megfelelő kifejezést, és az így kapott értekkel folytatjuk a függvény kiértékelését. Mint azt Manna és McCharthy (13) bebizonyította, az érték és név szerinti hívás azonos eredményt ad ugyan, de a név szerinti híváskor függvénynek nagyobb az értelmezési tartománya általában, ugyanis ebben az esetben lehetséges, hogy valamelyik paraméter az adott helyen nem értelmezhető és így az összetett függvény sem érték szerinti hívás eseten, név szerinti hívás eseten viszont lehetséges, hogy azt a paramétert nem használjuk fel a függvény kiértékeléséhez.

A másik probléma, amely ilyenkor felmerül, szintén nem kapcsolódik szorosan az imperatív tulajdonságokhoz, de éppen az imperatív tulajdonságok applikatív leírásával kapcsolatban merül fel gyakran ez a probléma az önalkalmazás problémája.

Magasabb rendűnek nevezünk egy programozási nyelvet, ha függvényeken értelmezett vagy függvényértékű függvények is leírhatok benne. A probléma ott lép fel, ha a függvényeken definiált függvény önmagával azonos típusú függvényekre is alkalmazható. Ez esetben ugyanis a függvények értelmezési tartományára a

$$V = V \rightarrow V$$

rekurzív egyenlettel irható, ami ha → jelölést úgy értelmezzük, mint egy halmazból egy másikba történő összes lehetséges leképzést szamossági okokból lehetetlen eredményt kapunk. Nyilvánvaló tehát, hogy itt nem az összes lehetséges leképzésről van szó. Az ilyen függvényeknek az elméletet Dana Scott (12) alapozta meg algoritmuselméleti, halóelméleti és topológiai alapokon. Ezen alapuló újabb eredményeket találunk Manna és Vuillemin 14 munkájában.

Összefoglalva megállapíthatjuk, hogy igen sok kísérlet történt a szemantika applikatív nyelvű interpreterrel történő definiálására. Ezeknek nagyon jó összefoglalása található Reynolds (5) művében. Mindezek elméleti szempontból sok érdekeset tartalmaznak. Gyakorlati szempontból természetesen a VDL kiemelkedik a többi közül. (3,4).

Mindezeknek a kísérleteknek közös jellemzője azonban az, hogy nem a valóságos nyelvet interpretálják, hanem egy úgynevezett absztrakt szintaxist. Az absztrakt szintaxis lényegében a programnak egy olyan leírása, amely a nyelvben szereplő összes implicit információt explicite leírja. Az absztrakt szintaxis például minden változónál megmutatja annak összes lehetséges tulajdonságát vagy annak a hiányát megmutatja a kifejezések részeinek egybekapcsolódását stb. A konkrét program és az absztrakt szintaxis kapcsolatát egy transzlator hozza létre, de ezt egyik írás sem részletezi.

4. Az öndefiniáló módszer

A módszert először a LISP és más listakezelő nyelvek alkalmazták, a módszer lényege a következő.

Feltesszük, hogy egész pontosan és egyértelműen tudjuk definiálni a nyelv alapelemeit és bizonyos képzési szabályokat, ahogy bonyolultabb kifejezéseket felírhatunk ezekből. Ezek után a bonyolultabb elemeket az alapelemekből képzett kifejezésekkel írunk le, és a megbonyolultabbak az eddig definiáltak segítségével és így tovább.

A módszernek hármas előnye is van:

[1] A fokozatos felépítés lehetővé teszi a nyelv jó megértését.

[2] Lehetővé teszi a nyelv interpreterének a létrehozását bootstrapping útján.

[3] Mivel az így készült interpreter általában terjedelmes és lassú, ezt úgy javíthatjuk fokozatosan, hogy a bennük szereplő bonyolult függvényeket egyenként kicseréljük közvetlenül gépikódban írt változatukra.

5. A portabilitáson alapuló módszerek

Nagyon gyakori jelenség az, hogy egy nyelvnek a különböző gépeken alkalmazott változatai jelentős szintaktikai és szemantikai eltéréseket mutatnak. Egy nyelv két fordítóprogramját kompatibilisnek nevezzük, ha a két fordítóprogram ugyanazokat a programokat fogadja el helyesnek és a lefordított programok végrehajtása ugyanazt az eredményt szolgáltatja. A kompatibilitás nyilvánvalóan csak a nyelv szintaxisának és szemantikájának pontos definíciója alapján valósítható meg.

A nyelv egy lehetséges definíciója és egyben a kompatibilitás elérésének legbiztosabb útja az, ha írunk egy fordítóprogramot a nyelvre és ezt alkalmazzuk minden lehetséges gépre.

Egy program portabilitása az a tulajdonsaga, hogy könnyen alkalmazható minden gépen, illetve sok gépen. A portabilitást úgy érhetjük el legkönnyebben, ha egy portabilis /azaz sok gépre könnyen alkalmazható/ nyelven írjuk meg a programot.

Ha a nyelv fordítóprogramját egy portabilis nyelven írjuk meg, és ez a fordítóprogram egy portabilis tárgynyelvre fordít /a két nyelv lehet azonos, lehet különböző is/; akkor mind a fordítóprogramot, mind a lefordított programokat nagyon könnyű átültetni egy-egy konkrét gépre /rendszerint egyetlen makroprocesszálási fázis/.

A portabilis fordítóprogram nemcsak a gép szempontjából jelenti a szintaxis és szemantika definícióját. Az így nyert fordítóprogram ugyanis általában lassú és terjedelmes, ilyenkor a fordítóprogram kritikus részeit újraprogramozzák a gép nyelvén a program alapján.

Vizsgálatok folytak arra nézve, hogy milyen nyelv alkalmas leginkább portabilitásra. (15) A vizsgálat azt mutatta, hogy az assembler szintű nyelv a legalkalmasabb. A vizsgálat ugyanis azt mutatta, hogy a gépi kódok általában hasonlók egymáshoz, ha pedig nagyon eltérnek (ILLIAC IV, CDC STAR) akkor ez még a magasszintű nyelveken is erezhető.

A portabilis nyelveknek ez a szintje teszi lehetővé, hogy a portabilis nyelvek szemantikája majdnem triviális.

Mint a felsorolás is mutatja a szemantika definiálásának igen sok lehetséges útja van, és bár pillanatnyilag meg egyik út sem vezetett célhoz, valószínű, hogy a célhoz több úton is el fogunk jutni.

Összefoglalva tehát a következőket mondhatjuk a szintaxis és szemantika definiálásáról és formalizálásáról.

Mindaddig nem beszélhetünk komolyan a szemantika definiálásáról, amíg a szintaxisnak egy a jelenleginél sokkal pontosabb definíciója meg nem születik.

A számolásra orientált nyelvek szemantikáját nem definiálhatjuk közérthetően, addig amíg nem szerzünk tiszta képet a számológép aritmetikájának az alapvető tulajdonságairól.

A szintaxis és szemantika formalizálása több különböző úton lehetséges; egyes nyelveknek bizonyára az egyik, más nyelvnek a másik felel meg jobban. Elképzelhető, hogy a jelenlegi nyelvekkel kapcsolatban belső ellentmondások merülnek fel és ezért új nyelvek definiálása válik szükségessé.

IRODALOMJEGYZÉK

[1] C.Strachey: Varieties of programming Language. International Computing Symp. Venice 1972.

[2] Dana Scott and Christopher Statchey: Towards a Mathematical Semantics for Computer Languages. Oxford University Programming Research Group 1971*

[3] E. .Neuhold: The Formal Description of Programming Languages. IBM System Journal 1971. No2.

[4] P.Lucas, K.Walk: On. the Formal Description of PL/I. – Annual Review in Automatic Programming. 1969. Part 3.

[5] John C.Reynolds: Definitional Interpreters for Higher-Order Programming Languages. ACM Annual Conference 1972. Boston.

[6] J.McCarthy, Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. Com ACM 3 i960.

[7] W.M.Waite : Building a mobile programming system. The Computer Journal 1970.

[8] An Axiomatic Basis for Computer Programming C.A.R. Hoare Com. ACM 1969/10.

[9] P.J.Landin: A Correspondence between Algol 60 and Church's Lambda Notation. Part I, Part II. COM ACM 1965/2,3.

[10] D.Scott: Outline of a Mathematical theory of Computation. Oxford University Programming Research Groups 1970.

[11] Z.Manna, J.McCarthy: Properties of Programs and Partial Function logic Machine Intelligence vol 5.

[12] Z.Manna, J.Vuillemin: Fixpoint Approach to the Theory of Computation. Com. ACM. I972/7.

[13] P.J.Brown: Levels of Language for Portable Software. Com ACM 1972/12.