Fordprog tételsor részlet

Boncz Zoltán

March 23, 2006

Hibalistát a bonczz@gmail.com címre

A doksi az alábbi oldalakat tartalmazza:
23, 24, 26, 28, 29, 32, 36, 37, 38, 39, 40, 41
Az oldalhatárok a forráskódban szerepelnek: %-jellel körülbástyázva.

A jelölésben megprobáltam konzekvens maradni, amennyiben ez mégsem sikerült akkor az hibának minősül (gyk. hibalistára vele)

Amit nem tudtam elolvasni az általában lábjegyzetben szerepel, vagy kérdöjelek formájában jelenik meg.

(20. tétel) LALR(1) - elemző

Az SLR(1) elemző állapotszámánál az LR(1) elemző állapotszáma lényegesen magasabb. Cél az állapotok számának csökkentése úgy, hogy az elemezhető nyelvek halmaza az LR(1) nyelvekhez viszonyitva lényegesen ne csökkenjen. Az LR(1) elemek kanonikus halmazai között vannak olyanok, amelyekben ugyanolyan magú LR(1) elemek vannak. Egyesítsük ezeket, így kapjuk LALR(1)-et, ahol a halmazok száma az LR(0) kanonikus halmazok számával egyezik meg. A read fv. az egyesített halmazokra nem okozhat problémát, mert csak a magtól függ. léptetés/redukálás probléma nem fordulhat elő mert az eredeti grammatika nem lett volna LR(1). redukálás/redukálás probléma viszont előfordulhat. Ha az LR(1) elemek kanonikus halmazai: $I_0, I_1, \ldots I_m$, akkor az egyesített halmazok: $R_1, \ldots, R_n$ ($n \leq m$) Ezeket használva töltjük ki a táblázatot, pont úgy mint az LR(1)-nél.

Definíció. *Ha a G kiegészített grammatikára az LALR(1) elemző táblázatok kitötése konfliktus mentes, akkor G egy LALR(1) grammatika*

Az LALR(1) elemző táblázatok kitöltése nagyon munkaigényes, mert az összes LR(1)-elemet meg kell határozni. *Gyorsítás*: a nem I_0 LR(1) kanonikus halmazokban csak azokban az LR(1)-elemekben kezdődik a mag jobboldala ponttal, amelyek már egy, a kanonikus halmazban levő LR(1)-elemből a closure fv.-el származnak. így a kanonikus halmazokat nem kell összes elemük felsorolásával megadni.

Definíció. *I_0 kanonikus halmaz törzse legyen az* [S' $\rightarrow$ .S , #] *LR(1)-elem; nem I_0 kanonikus halmaz törzsébe pedig azok az LR(1)-elemei tartoznak, amelyekben a mag jobboldala nem ponttal kezdődik.*

Egy kanonikus halmazt tehát a törzsével is megadhatunk, hiszen abból az összes többi LR(1)-elem előállítható. A törzsből a léptetés és a redukció műveletek is meghatározhatók, így a táblázatok felépíthetők.

A törzs meghatározásának problémája:
Ha az LR(0)-elemek kanonikus halmazaira is definiáljuk a törzs fogalmát, akkor láthatjuk, hogy ezek a törzsek megegyeznek az LR(1) kanonikus halmazokból egyesítéssel létrehozott halmazok törzseivel (a magok azonosak). Ha a törzseket (LR(0)) kiegészítjük előreolvasási szimbólumokkal, akkor az egyesített LR(1) halmazok törzseit fogjuk mekapni. Az LR(0)-elemekre, ha ismerjük I_j-t akkor a $read(I_j, x)$ könnyen meghatározható. LR(1)-nél ez már bonyolultabb, mert ahhoz az előreolvasási szimbólumot is meg kell határozni, vagyis, hogy milyen a-ra lesz $[A \rightarrow X.\alpha\,,\, a] \in READ(R_j, x)$

Ha $\mu\delta \neq \varepsilon$ és $a \in FIRST_1(\mu\delta)$, akkor biztosan $[A \rightarrow X.\alpha\,,\, a]$
$\in READ(R_j, x)$ és azt mondjuk, hogy az 'a' előreolvasási szimbólum $READ(R_j, x)$-hez spontán generálható, ($[B \rightarrow \gamma.C\delta\,,\, b] \in R_j\,,\ C \rightarrow^* A\mu$ és $A \rightarrow X\alpha$) hiszen a 'b' szimbólumnak semmilyen szerepe nincs az új

előreolvasási szimbólum meghatározásában. Ha $\mu\delta = \varepsilon$, akkor $[A \to X.\alpha\,,\, b]$ lesz a $READ(R_j, x)$ eleme, azaz ebben az elemben a 'b' lesz az előreolvasási szimbólum, ekkor azt mondjuk, hogy a 'b' R_j-ből öröklődik a $READ(R_j, x)$-be. Ha adott a I_j törzse, akkor $\forall X$-re a $READ(R_j, X)$-hez az előreolvasási szimbólumok meghatározása:

- $\forall[B \to \gamma.\delta] \in I_j$-re határozzuk meg az $R'_j = CLOSURE([B \to \gamma\,,\, @])$ halmazt, ahol '@' egy dummy szimbólum
- Ha $[A \to \alpha.X\beta\,,\, a] \in R'_j$ és $a \neq @$
  akkor $[A \to \alpha X.\beta\,,\, a] \in READ(R_j, X)$ és az 'a' spontán generálható
- Ha $[A \to \alpha.X\beta\,,\, @] \in R'_j$, akkor $[A \to \alpha\beta\,,\, @] \in READ(R_j, X)$ és a @ öröklődik a $READ(R_j, X)$-be.

Ha már ismerjük a spontán generált és az öröklődő előreolvasási szimbólumokat, akkor az egyesített halmazok törzséhez meg tudjuk határozni az összes előreolvasási szimbólumot: egy táblázat tartalmazza az LR(0) elemek halmazainak a törzsét, az elemekhez tartozó spontán generált előreolvasási szimbólumokat és az R_0-hoz definiált #-ot. Ezután az előreolvasási szimbólumokat az öröklődések alapján átvisszük az összes lehetséges elemhez.

(22. tétel) Hibakezelési módszerek az LL(1) és SLR(1) elemzésekben

LL(1)
Hibaelfedési eljárás a rekurzív leszállás módszeréhez.
Alapprobléma: ha a szintaktikus elemző hibát detektált, akkor hogyan lehet az elemző programot és a szimbólumsorozatot újból szinkronizálni. Arra törekszünk, hogy minden procedúrahívás és procedúrából való kilépés szinkronizált legyen. A kihagyott szimbólumok száma a procedúrában meghívott procedúrák[1] számával egyenlő.
Minden szimbólumhoz meg tudjuk határozni a $FIRST_1(A)$ terminálisok halmazát, legyen ez 'starters'. SKIPTO(starters): az input sorozat szimbólumait átúgorja, amíg a starters halmazba tartozót nem talál. Hasonlóan a $FOLLOW_1(A)$ halmaz legyen 'followers'.
Okosítás: SKIPTO(starters+followers)
Okosítás: followers := followers $\cup FOLLOW_1(B)$,
ahol B az A-t hívó procedúra.

```
procedure A(followers);
  begin
    if not(akt_szimbólum in starters) then
      begin
        error(...);
```

[1] erdekesen hangzik

```
      skipto(starters + followers);
    end;
  if (akt_szimbólum in starters) then
    begin
      T(A)
      if not (akt_szimbólum in followers) then
        begin
          error(...)
          skipto(followers)
        end;
    end;
end;
```

SLR(1)
Hiba a *goto* táblázatból sosem származhat, az *action* táblázatban (az operátor–precedenciához hasonlóan) hibajelzéseket tehetünk és különböző típusú hibákhoz különböző tevékenységeket rendelhetünk. Egy egyszerű hibaelfedési módszer: vezessünk be egy *error* terminálist, majd válasszuk ki a legfontosabb nemterminálisokat (pl.: program, blokk, utasítás) és az ezekre vonatkozó szabályokat egészítsük ki egy $\mathsf{A} \rightarrow$ *error* alternatívával.
Ez alapján készítsük el a táblázatokat. Ha az elemző a táblázat alapján hibát talál, akkor a veremből (elempárokat) törölve visszalép ahhoz a legközelebbi állapothoz, amelyikből az *error* egy másik állapotba vezet. Ekkor az *error* szimbólumot a verembe teszi, az állapotot pedig a táblázat alapján határozza meg. Ha $\alpha \neq \varepsilon$, akkor az inputban is előrelép addig, amíg az input pointer az első szimbólumra nem mutat.

(23.tétel) Szemantikus elemzés veremmel

Szemantikus elemzés a szemantikus fa alapján történik. A szintaxis fa pontjaihoz olyan attribútumokat rendelünk, amelyek leírják az adott pont tulajdonságát. Ezeknek az attribútumoknak a meghatározása és konzisztenciájuk vizsgálata a szemantikus elemzés feladata.
A vizsgált tulajdonságok:

i) változók deklarációja, hatásköre, láthatósága
ii) többszörös deklaráció, deklaráció hiánya
iii) operátorok és operandusok közötti kompatibilitás
iv) procedúrák, tömbök paraméterei közötti kompatibilitás.

Az első kettő vizsgálat megoldható egy jól választott szimbólum-tábla kezeléssel.

Akciószimbólumok
Ha a szintaxisfa felépítésekor egy adott szabály esetében szemantikus elemzést is kell végezni, akkor a szabályokba 'akciószimbólumokat' írhatunk. '@Prog' esetén a 'Prog' szemantikus rutint kell végrehajtani. Ha a G grammatikát akciószimbólumokkal egészítjük ki, akkor TG fordítási grammatikát kapunk. Probléma, hogy a szemantikus rutinoknak nincs paraméterük. Mivel a szintaktikus elemzők vermet használnak, ezért a rutinok egymás közti paraméterátadásaira is vermet fogunk használni. A szemantikus verem lehet:

akcióvezérelt: a rutinok kezelik a verem push és pop műveleteit (vannak hátrányai)

elemzővezérelt: a szemantikus verem kezelése a szintaktikus verem kezelésével párhúzamos, az elemző egyszerre hajt bennük végre műveleteket.

Elemzővezérelt szemantikus verem [2]

Alulról felfelé elemzés (LR(1)):
Ha $\mathsf{A} \rightarrow \alpha$ szerinti redukció következik, akkor az α szimbólumai a szintaktikai veremben vannak, a szemantikai veremben pedig a szimbólumokhoz tartozó bejegyzések vannak. Ezek a bejegyzések csak akkor használhatóak egy rutin paramétereiként, ha kikerülnek a szemantikai veremből, vagyis redukciónál. Tehát alulról-felfelé elemzésnél rutin csak helyettesítési szabályhoz adható meg *(egyhez csak egy)*, végrehajtani pedig az adott szabály szerinti redukciónál lehet.
A szabály jobb oldalának szimbólumaihoz tartozó attribútumok címzése:
a pop művelet ne változtassa meg a verem tartalmát, csak az SP verem–mutatót mozgassa. Így a redukáláskor végrehajtandó pop műveletek után, de a push művelet előtt az attribútumokat $SP+1$, $SP+2$, ... módon tudjuk címezni és nem kell a kiolvasott értékeket külön tárolni.

Felülről lefelé elemzés (LL(1)):
Ha $\mathsf{A} \rightarrow \alpha$ szabályt alkalmazunk, akkor egy pop művelettel az A szimbólumot kivesszük a veremből, majd az α szimbólumait a verembe tesszük push-sal. Ha a két vermet párhúzamosan kezelnénk, akkor $\mathsf{A} \rightarrow \alpha$-nál elveszne az A-hoz tartozó szemantikus elem, amire később még szükség lehet. Amikor a szintaktikai verem tetején álló szimbólum megeggyezik a következő input szimbólummal, akkor is pop-ot kellene végrehajtani, de akkor is elvesztenénk az attribútumot.
Megoldás: egy $\mathsf{A} \rightarrow \mathsf{X}_1, \mathsf{X}_2, \ldots, \mathsf{X}_n$ szabály esetén egyik veremre se hajtsuk végre a pop műveletet. A szintaktikai veremben az A helyére írjunk egy EOP kódot, a szemantikus verembe pedig tegyük bele A-t(?)[3] Ezután már

[2] azt hiszem csak ezt vettuk
[3] sic

végrehajthatjuk a push műveleteket az X_i-kre, de az akciószimbólumokat csak a szintaktikai verembe tegyük be *(a másikba felesleges)*. A szemantikus verem kezelésére három pointert használunk: $\mathsf{A} \to \mathsf{X}_1, \ldots, \mathsf{X}_n$ szabály esetén left az A, right az X_n, top pedig az X_1 attribútumait tartalmazó mezőkre mutat. Az EOP szimbólum beírásakor mentsük el a pillanatnyi állapotot is EOP(left,right,top) formában.
Ha a szintaktikai verem tetején egy EOP van, akkor az $\mathsf{A} \to \mathsf{X}_1, \ldots, \mathsf{X}_n$ feldolgozása megtörtént, a szemantikus veremből ki kell törölni a top és right közötti attribútumokat.

(25. tétel) Rendezett attribútum fordítási grammatikák

A nyelv mondatain értelmezett nem feltétlenül véges attribútum függőségeket a szabályokhoz és a szimbólumokhoz kapcsolódó, az attribútomok közötti véges darabszámú függőségekre képezzük le. DS(X) legyen parciális rendezés ∀ X-re, ez határozza meg az attribútumok kiszámítási sorrendjét pl.: (X.a, X.b) ∈ DS(X)

Definíció. *Az $\mathcal{A}(X)$ egy $\mathcal{A}_1(X)$, $\mathcal{A}_2(X)$, …, $\mathcal{A}_{m(x)}(X)$ particionálását megengedett particionálásnak nevezzük, ha*

a) $\mathcal{A}_{m(x)}, \mathcal{A}_{m(x)-2}, \mathcal{A}_{m(x)-4}, \ldots \subseteq \varphi(X)$

b) $\mathcal{A}_{m(x)-1}, \mathcal{A}_{m(x)-3}, \mathcal{A}_{m(x)-5}, \ldots \subseteq \mu(X)$

Definíció. *Egy ATG partícionált, ha lokálisan aciklikus és minden X-hez létezik egy olyan $\mathcal{A}_1(X), \ldots, \mathcal{A}_n(x)(X)$ megengedett partícionálás, hogy az X attribútumai a partíciók sorrendjében meghatározhatók.*

Következmény: Ha egy ATG partícionált, akkor jóldefiniált $\mathcal{A}_i(\mathsf{X})$-et a DP(p) direkt függőségek figyelembevételével kell meghatározni (DP(p) a p-hez tartozó szem. fv.-ek halmaza) Mivel DP(p) tranzitív, ezért képezzük NDP(p) normalizált tranzitív lezárását:
NDP(p) = $\mathrm{DP}^+(\mathrm{p}) - \{(\mathsf{X}.a\,,\,\mathsf{X}.b) | \mathsf{X}.a\,,\,\mathsf{X}.b \in \mathrm{AF(p)}\}$ (AF(p)) a szemantikus fv.-ekben a baloldalak halmaza)
Legyen IDP(p) a p szabályra vonatkozó, IDS(X) az X attribútumai között fennálló indukált függőségek:

Definíció. *Egy ATG indukált függőségei:*

i) $\forall p \in P\text{-re } IDP(p) = NDP(p)$
ii) $\forall X : IDS(X) = (X.a\,,\,X.b) | \exists q : (X.a\,,\,X.b) \in IDP^+(q)$
iii) $\forall p : X_0 \to X_1 \ldots X_n : IDP(p) = IDP(p) \cup IDS(X_0) \cup \ldots \cup IDS(X_n)$
iv) A ii) és iii) pontot ismételjük amig van változás

Tétel. *Ha egy ATG partícionált akkor* $\forall p$ *és* $\forall X : IDP(p)$ *és IDS(X) gráfja körmentes. Ha* $(X.a\,,\, X.b) \in IDS(X)$*, akkor* $X.a \in \mathcal{A}(X_i)$ *és* $X.b \in \mathcal{A}(X_j)$ *és* $i \leq j$

A probléma az, hogy egy partícionálás további függőségeket indukál, hiszen két különböző partíció beli attr.-ra a paríciók automatikusan relációt írnak elő.

Definíció. *Legyen* $\mathcal{A}_1(X), \ldots, \mathcal{A}_{m(x)}(X)$ *megengedett partícionálás és* $p : X_0 \to X_1 \ldots, X_n \in DP(p)$ *kiterjesztett attribútum függőség*
$EDP(p) = IDP(p) \cup \{(X_i.a\,,\, X_i.b) \mid X_i.a \in \mathcal{A}_j(X_i)\,,\, X_i.b \in \mathcal{A}_k(X_i)\}$
ahol: $0 \leq i \leq n, j < k$

Tétel. *Egy ATG partícionált* $\iff$ $tp \in P : EDP(p)$ *gráf körmentes*

Definíció. *Egy ATG rendezett ATG, ha a következő, az* $\mathcal{A}$ *partícionálását végző algoritmussal partícionált grammatikát kapunk:*
$\forall$ *X: a partíciók legyenek* $\mathcal{A}_1(X), \ldots, \mathcal{A}_{m(x)}(X)$*, ahol* $\mathcal{A}_i(X) = T_{m-i+1}(X) - T_{m-i-1}(X)$ $(i = 1, \ldots, m(x))$*, ahol:* $T_{-1}(X) = T_0(X) = 0$*,* $m(X)$ *az a legkisebb olyan n index, amelyre* $T_{n-1}(X) \cup T_n(X) = \mathcal{A}(X)$ *és* $\forall k > 0$*:*

$$T_{2k-1}(X) = \{X.a \in \varphi(X) \mid ha\ (X.a,\ X.b) \in IDS(X),\ akkor\ X.b \in T_j(X),\ j \leq 2k-1\}$$

$$T_{2k}(X) = \{X.a \in \mu(X) \mid ha\ (X.a,\ X.b) \in IDS(X),\ akkor\ X.b \in T_j(X),\ j \leq 2k\}$$

Köv: Minden rendezett ATG partícionált!
Eredmény: Ha egy rendezett ATG-hez meghatározzuk $\mathcal{A}_i(X)$-eket, akkor DS(X) = $\{(X.a,\ X.b) \mid X.a \in \mathcal{A}_i(X),\ X.b \in \mathcal{A}_j(X),\ 1 \leq i,\ j \leq m(x),\ i \leq\ j\}$
Az attribútumok meghatározásának sorrendje független a nyelv mondataitól, csak a grammatikától függ!

(29.tétel) Prológus, epilógus, típusdeklarációk, változódeklarációk

A kódgeneráló egy olyan assembly nyelvű programot késszít, amelyben egy adat-, egy verem- és egy kódszegmens van. Az adatszegmensbe kerrülnek a program globális változói, a veremszegmensbe az alprogramok aktivációs rekordjai, a kódszegmensbe a lefordított utasítások.
A lefordítandó forrásnyelvű program:
`<program>` $\to \underline{program}$ `<id>` $\uparrow v$ @$Prologue(\downarrow v)$ `<block>` @Epilogue,
ahol @Prologue($\downarrow v$) az assembly program bevezető sorait generálja (v a neve), @Eplilogue pedig a végét.
A .DATA és a .CODE direktívák bárhol és többször is előfordulhatnak, az alprogramoktól függően. Most csak egy alprogrammal foglalkozunk

Típusdeklaráció:
<típusdeklaráció> → *type* <id> ↑n = <típusdefiníció> ↑d @TypeDecl (↓n, d) ahol: @TypeDecl végzi a szimbólumtábla kezelését: az n névhez egy bejegyzést készít és ehhez kapcsolja a d típusdeszkriptort.
Fontos probléma a típusok ekvivalenciája és kompatibilitása.

- két típus ekvivalenciájának vizsgálata a tipusdeszkriptorokra mutató pointerek azonosságának vizsgálata.
- egy kifejezésben egy operátor operandusainak kell kompatibilisnek lenniök.

Konstans deklaráció: pl.: constant real pi := 3.14;
<konstansdeklaráció> → *constant* <típus> ↑t <id> ↑n := <kifejezés> ↑s, v @ConstDecl (↓n, t, s, v)
ahol:

t: type,
n: name,
v: value,
s: az érték típusa,
@ConstDecl ellenőrzi t és s azonosságát és n,t,v,-vel a szimbólumtáblába ír (n EQU v; @ConstDecl)

Változódeklarációk:
A változók lehetnek statikusak (egész, valós), vagy dinamikusak (stringek, dinamikus tömbök). A dinamikus változók fordításakor csak a változó deszkriptora számára kell helyet foglalni.

statikus, predefinit típusú egyszerű változók:
<egyszerű_változó_deklaráció> → <típus>↑t, m <id> ↑n
@VarAlloc (↓n, m, ↑q) @VarDecl (↓n, t, m, q)
<típus> → *real* ↑t, m |
integer ↑t, m |
boolean ↑t, m |
character ↑t (<darabszám> ↑m)
ahol:

t: type,
m: méret byte-okban,
n: name,
@VarAlloc: helyet allokál a változónak

- ha globális változó, akkor az adatszegmensben
  n DB m DUP(?); @VarAlloc,
- ha lokális változó, akkor egy aktivációs rekordban
  n EQU WORD PTR[BP]−k ; @VarAlloc

@VarDecl: a változót a szimbólum táblába írja

felsorolás típusú változók: Intenzív szimbólumtábla kezelést igényel, mert az azonosító lista elemeit is fel kell írni a szimbólumtáblába.

intervallum típusú változók: Az intervallum határait kell tárolni.
<integer_exp>↑n : <integer_exp>↑s
@Lowerbound(↓s) . . <integer_exp>↑r @Upperbound(↓r)
@InterVarAlloc (↓$n, s, r, $↑$q$) @InterVarDecl (↓$n, t, s, r, q$)

tömbök: Sorfolytonosan szoktuk tárolni

- n-dimenziós tömb memóriacíme loc,
- i-edik dimenzió határai L(i) és U(i)

így az $(i_1, \ldots, i_n)$ *indexű tömbelem címe:* $loc + \sum_{j=1}^{n}(i_j - L(j))P(j)$
ahol: $P(j) = \prod_{k=j+1}^{n}(U(k) - L(k) + 1)$ $P(n) = m$
(m a tömbelem mérete byteban)
legyen: $RC = \sum_{j=1}^{n} L(j)P(j)$,
így a cím: $loc - RC + \sum_{j=1}^{n} i_j P(j)$
A tömb descriptora: n : dimenzió, $L(i)$, $U(i)$, $P(i)$, RC, loc
<tömb_deklaráció> → *array* @InitDim(↑i) <id> ↑n
[<sublist>↓i↑j] *of* <típus>↑t, m @ArrayAlloc(↓$n, t, m, j,$ ↑q)
@ArrayDecl(↓n, tm, jiq)
<sublist>↓i ↑j → <subscript>↓i ↑j | <subscript> ↓i ↑j
@echo(↓j ↑i) , sublist↓i ↑ j
<subscript>↓i ↑j → @Dimension(↓$i,$↑j) <integer_exp> ↑s
@Bounds(↓s, j)| <integer_exp> ↑r
@LowerBound(↓r, j) <int_exp> ↑s @UpperBound
ahol:

- @InitDim(↑i) i változót deklarál a dimenziónak
- @Dimension(↓$i,$↑j) $j \leftarrow i + 1$
- @Bounds(↓s, j) $L(j) \leftarrow 1$, $U(j) \leftarrow s$

rekordok: Az elemek offset címeinek meghatározása a fő feladat.
<rekord_deklaráció> → <id>↑ n : *record* @InitOffset(↑r)
<elem_lista>↓r ↑s @RecAlloc(↓$n, s,$ ↑q) @RecDecl *end record*
<elem_lista>↓r ↑s → <elem>↓r ↑s| <elem>↓r ↑s
@echo(↓s ↑r) , <elem_lista>↓r ↑s
<elem>↓r ↑s → <idn>↑e : <típus>↑t, m @RecElemDecl (↓$e, t, m, r,$ ↑s)

(30. tétel) Kifejezések, logikai kifejezések

Kifejezések: A kifejezéseket a következő példán tanulmányozzuk:

(1) E → TE',
(2) E' → +TE'/ ε,
(3) T → FT',
(4) T'→ ∗ FT'/ε,
(5) F → (E) | i

csak integerekkel (két byte-os) dolgozunk. A kifejezés eredményét az AX regiszterben tároljuk, a részeredményeket a run-time veremben. Az utolsó szabályban szereplő 'i' esetei:

1. 'i' egy egyszerű változó.
   $F \rightarrow (\underline{i}) \uparrow n$ @SearchVar($\downarrow n, \uparrow t, q$) @StLoadAX($\downarrow q$)
   ahol:

   t: type,
   n: azonosító,
   q: a deklarációban meghatározott címe
   @SearchVar: megkeresi az n szimbólumot a szimbólum-táblában, ellenőrzi a láthatóságot és visszaadja a típusát és a címét
   @StLoadAX: a változó értékét az AX regiszterbe töltő utasítást generálja: MOV,AX,q;
   globális változókra: MOV AX,n;
   lokális vált: MOV AX, WORDPTR[BP]$-k$;

2. 'i' egy konstans deklarációjában szereplő név MOV AX,n

3. 'i' egy konstans literál, akkor legyen a 'q' értéke az 'i'-t alkotó literál

4. 'i' egy tömb elem
   $F \rightarrow \underline{i} \uparrow n$ @InitDim($\uparrow i$) @SearchVar($\downarrow n, \uparrow t, d, q$) [`<sublist>` $\downarrow i, d \uparrow j$] @GenLabel($\uparrow r$) @ArrayRef($\downarrow t, d, q, j, r$)
   `<sublist>` $\downarrow i, d, \uparrow j \rightarrow$ `<subscript>`$\downarrow i, d \uparrow j$ | `<subscript>`$\downarrow i, d \uparrow j$ @echo($\downarrow i, \uparrow j$)`<sublist>`$\downarrow i, d, \uparrow j$
   `<subscript>`$\downarrow i, d, \uparrow j \rightarrow$ @Dimension($\downarrow i, \uparrow j$) `<integer_exp>` $\uparrow s$ @CallIndex($\downarrow j, s, d$)
   ahol:

   @InitDim, @Dimension: ahogy eddig,
   @SearchVar: visszadja a descriptort (d-ben)
   @CallIndex($\downarrow j, s, d$)**:** olyan utasításokat generál, amelyek az AX-ben lévő 's' értéket összeszorozzák a tömb descriptorában lévő P(j) értékkel:

```
MOV BX,P(j);
IMUL BX;
PUSH AX;
```

@ArrayRef: ellenőrzi a dimenziószámot és kódot generál:

```
     LEA BX,loc;
     SUB BX,RC;
     MOV CX,j
L_r: POP AX;
     ADD BX,AX;
     LOOP L_r;
     MOV AX,[BX]
```

@GenLabel: generálja az `L_r` címkét a LOOP-nak

5. A statikus rekord egy elemének címe: *rekord eleje+offset*
   Az 'E' kifejezést tovább vizsgálva, legyen:

   @StPushAX: `PUSH AX;`
   @StPopBX: `POP BX;`
   @StAddBX: `ADD AX,BX;`
   @StImulBX: `IMUL BX;`

   A kifejezést leíró grammatika:
   E → TE'
   E' → + @StPushAX T @StPopBX @StADDBX E' | ε
   T → FT'
   T' → * @StPushAX F @StPopBX @StImulBX T' | ε
   F → (E) | $\underline{i} \uparrow n$ @... *(i-től függ)*

Logikai kifejezések FALSE: 0, TRUE: nem 0; + és $*$ helyett *or* és *and* van, így kell @StAndBX, @StOrBX és @Negate: `NEG AX`
Megadható, hogy olyan kódot generáljon a fordításkor, amelyik optimalizálja a kiértékelést, ha lehet
E → @GenLabel($\uparrow s$) D @StJTrue($\downarrow s$) *or* E @StLabel($\downarrow s$) | D
D → @GenLabel($\uparrow r$) C @StJFalse($\downarrow r$) *and* D @StLabel($\downarrow r$) | C
C → *not* N@Negate | N
N → (E) | $\underline{i}$ *ahol:*

@GenLabel($\uparrow r$)**:** létrehoz egy `L_r` cimkét
@StLabel($\downarrow r$)**:** kiad egy "`L_r:`" sort
@StFalse($\downarrow r$)**:** `CMP AX,0; JNZ B+5` [4] `???????; JMP L_r`

[4] ez itt mi a pina...

(31. tétel) Utasítások

Az értékadó utasítás A legtöbb magasszintű programnyelvben nincs lehetőség egy változó címének elérésére, a szemantikus elemzőnek kell eldöntenie, hogy adott esetben a változó címére, vagy értékére hivatkozunk-e.
`<értékadó_utasítás>` $\rightarrow$ @SET($\uparrow r$) `<változó>` $\downarrow r \uparrow t$:= @RESET ($\downarrow r$)
`<kifejezés>`$\uparrow s$ @STORE($\downarrow r, t, s$)
ahol:

r: a változó címét, vagy értékét kell-e meghatározni
@STORE: ellenőrzi t és s típusát és típuskonvertál, ha kell

Az if és case utasítás
`<if_utasítás>` $\rightarrow$ *if* `<kifejezés>` *then* `<utasítás`$_1$`> <if_tail>`
`<if_tail>` $\rightarrow$ *else* `<utasítás`$_2$`>` *endif* | *endif*
Az *endif* bevezetésével megszűnik a nem-egyértelműség!
kódgen:
`<if_utasítás>` $\rightarrow$ *if* `<kifejezés>` @GenLabel($\uparrow r$) @StJFalse($\downarrow r$)
then `utasítás <if_tail>`
`<if_tail>` $\rightarrow$ *else* @GenLabel($\uparrow s$) @StJMP ($\downarrow s$) @StLabel($\downarrow r$)
`<utasítás>` *endif* @StLabel($\downarrow s$) | *endif* @StLabel($\downarrow r$)
A case fordítása lényegében megegyezik az if fordításával, de gondoskodni kell arról, hogy az egyes elágazások feltételeinek fordításakor rendelkezésre álljon a case utáni kifejezés (@SaveExp, @LoadExp, @ResetExp)

Ciklusutasítások

- **feltétel nélküli ciklus :**
  `<ciklusutasítás>` $\rightarrow$ *loop* @GenLabel ($\uparrow r$) @StLabel($\downarrow r$)
  `<utasítások>` *endloop*; @StJMP($\downarrow r$)

- **while-ciklus :**
  `<while_utasítás>` $\rightarrow$ `<while_head>` $\uparrow r, s$ *do* `<utasítás>`
  `<while_tail>` $\downarrow r, s$
  `<while_head>` $\uparrow r, s \rightarrow$ *while* @GenLabel($\uparrow r$) @StLabel($\downarrow r$)
  `<kifejezés>` @GenLabel($\uparrow s$) @StJFalse($\downarrow s$)
  `<while_tail>` $\downarrow r, s$ @StJMP($\downarrow r$) @StLabel($\downarrow s$)

- **for-ciklus :**
  nothing [5]

GOTO utasítás A goto fordítása a már ismert predefinit/postdefinit hivatkozások fordításához hasonló. A fordítás csak az őt deklaráló blokkban lehet.

[5] itt frankon nem all semmi!!

Az exit és a return utasítások

exit(break): a vezérlés a ciklus utáni első utasításra adódik
probléma: Hol van az exitet tartalmazó legbelső ciklus vége?
megoldás: Minden ciklushoz rendelünk egy logikai attr.-ot (e) Minden ciklus kezdődjön @ResetExit($\uparrow e$) *(jelentése e := FALSE)* és @GenLabel($\uparrow f$) rutinokkal ill. végződjön @CoudExit($\downarrow e, f$)-el[6] ami ha e = TRUE, akkor @StLabel ($\downarrow f$) *legyen:* `<exit_utasítás>`$\downarrow e, f \rightarrow$ *exit* @SetExit($\uparrow e$) @StJMP($\downarrow f$)

return: az alprogram végrehajtása befejeződik, fordítása az exit-hez hasonló.

Kivételkezelés: Ha kivétel lép fel, akkor a kivételkezelő handler kezdőcímére kell a vezérlést adni és biztosítani kell, hogy a vezérlés a kivételt kiváltó utasítást követő utasításra visszatérhessen.
Átviteli vektor: a handlerek címeit tartalmazza, a kivételek sorszámának sorrendjében. A forrásnyelvű programot úgy osztjuk tartományokra, hogy egy tartomány összes utasítására ugyanaz a kivételkezelés legyen érvényes. Ekkor egy tartomány-térkép segítségével a program minden pontján egyértelműen megadhatjuk, hogy az i-edik tartományban a j-edik kivételhez a $handler_{ij}$ tartozik.

[6] talan CondExit