Szemantikai elemzés, attribútumos nyelvtanok

Simon Balázs

BME IIT, 2010.

forrás: http://www.info.uni-karlsruhe.de/lehre/2007WS/uebau1/

Tartalom

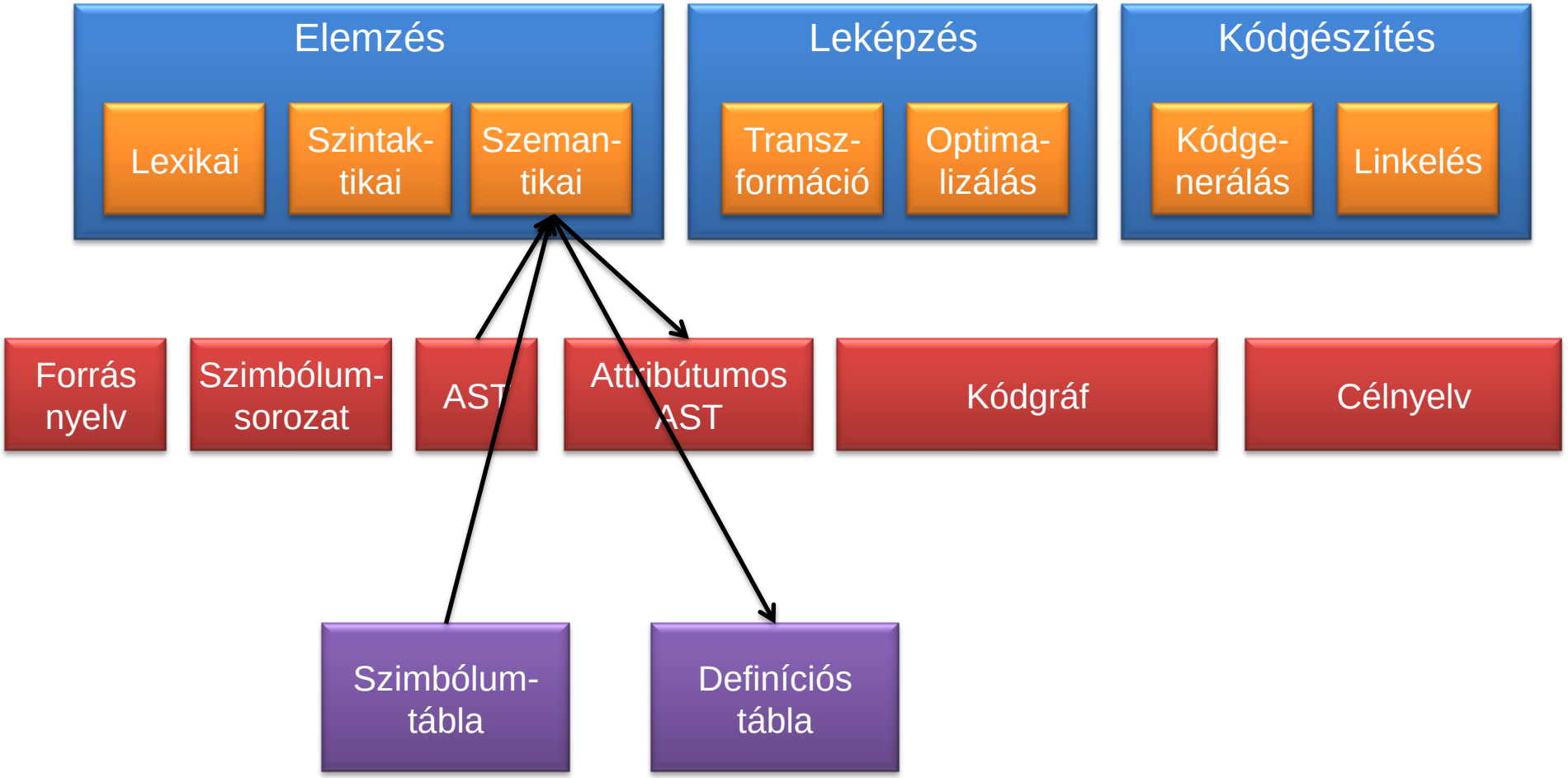
- Szemantikai elemzés
- Attribútumos nyelvtanok (AG)
- Attribútumok kiértékelésének sorrendje
- Névelemzés
- Típuselemzés

Szemantikai elemzés

Szemantikai elemzés

- AST elemzése
  - indok: a programnyelvek context-sensitive-ek
  - azonosítók (identifier) jelentésének meghatározása
- Név- és típusfeloldás
- Konzisztenciaellenőrzés
  - a programnyelv által meghatározott korlátok ellenőrzése
  - pl. statikus tömbök kiindexelése, konstansok, interfész függvényeinek implementációja, stb.
- Jelentés hozzárendelése
  - operátorok
  - definíciók és hivatkozások összerendelése (változók, függvények, stb.)

Szemantikai elemzés



Attribútumos nyelvtanok (AG)

Attribútumos nyelvtanok

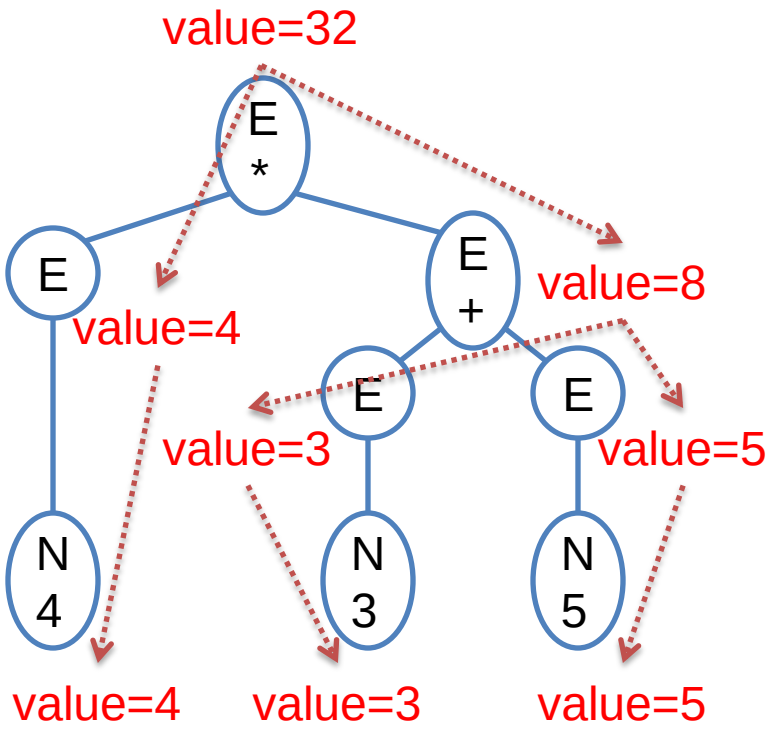
- Attributed grammars (AG)
- Miért van rá szükség?
  - a szemantikai ellenőrzőnek csak az AST áll rendelkezésére, amiből információt nyerhet
  - hatékony módszerek szükségesek a hierarchiában való mozgáshoz és számításhoz
  - a számítási szabályok általában erősen függenek az adott csúcs típusától
- Az AG-k erre adnak egységes megoldást
  - a számítási szabályok leírásától függetlenül különböző kiértékelési módszerek lehetségesek

Attribútumos nyelvtanok

- Az AST leírásához használt CF nyelvtanra épít
  - AST csúcsai: nemterminálisok, terminálisok
  - egy csúcs gyermekei: egy adott levezetési szabály jobb oldalán szereplő elemek
- A fa minden csúcsához tartoznak attribútumok (név-érték párok)
- Lehetnek előre definiált értékű attribútumok (pl. pozíció a forráskódban)
- Egy CF nyelvtani szabályban felhasznált elemek attribútumai között lehetnek függőségek
- Egy attribútum értéke egy levezetési szabály kontextusában a többi attribútum értékéből számítható
- Egy attribútumot pontosan egyszer kell kiszámolni, különben konzisztenciaproblémák lehetnek

Példa

- E → E+E
  - E[1].value = E[2].value + E[3].value;
- E → E*E
  - E[1].value = E[2].value * E[3].value;
- E → N
  - E.value = N.value;

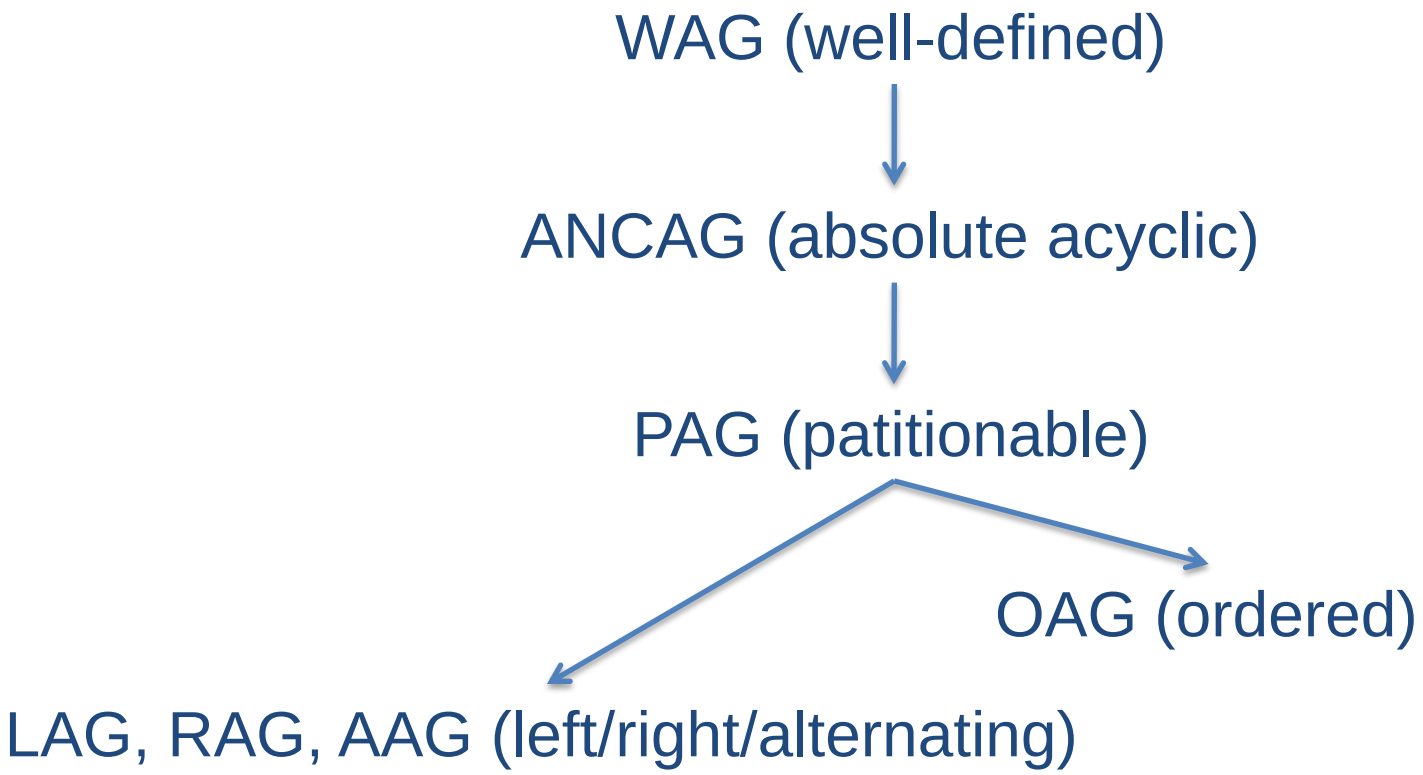


Attribútumok fajtái

- Szintetizált (synthetized) attribútum:
  - $A \rightarrow \alpha X \beta$
    - A.attribute = … X.attribute …
  - a levezetési szabály bal oldalán álló nemterminális attribútuma a jobb oldalon álló elemek attribútumaiból van számítva (lentről felfelé)
- Örökölt (inherited) attribútum:
  - $A \rightarrow \alpha X \beta$
    - X.attribute = … A.attribute …
  - a levezetési szabály jobb oldalán álló elem attribútuma a bal oldalon álló nemterminális attribútumai alapján van számítva (fentről lefelé)

Attribútumok kiértékelési sorrendje

- A sorrend a függőségek alapján dől el
- Kategóriák:



WAG

- Well-defined AG
- Jóldefiniált AG:
  - minden fára létezik hatékony módszer az attribútumok kiszámítására
  - vagyis: egy adott fánál minden attribútumot pontosan egy szabály definiál és nincs körkörös függőség az attribútumok között
- A WAG tulajdonság megléte csak exponenciális költségű algoritmussal vizsgálható.

ANCAG

- Absolute acyclic AG
- Abszolút aciklikus AG:
  - az összes fát figyelembe véve sincs körkörös függőség az attribútumok között
- Az ANCAG tulajdonság megléte polinom idejű algoritmussal ellenőrizhető.

PAG

- Partitionable AG
- Partícionálható AG:
  - a terminálisok és nemterminálisok attribútumai felosztható olyan halmazokra, hogy ezek kiértékelése a felosztás sorrendjében megtörténhet, függetlenül attól, hogy az adott terminális illetve nemterminális milyen szabályban fordul elő
- A PAG tulajdonság meglétének ellenőrzése NP-teljes feladat.

OAG

- Ordered AG
- Rendezett AG:
  - a lusta kiértékelés az attribútumok egy helyes partícionálását adja
- Minden PAG nyelvan új függőségek bevezetésével OAG-vá alakítható.
- Az OAG tulajdonság megléte polinom időben ellenőrizhető.

LAG

- Left AG
- Bal AG:
  - az attribútumok balról jobbra történő mélységi bejárással kiszámíthatók
- LAG(k): k db mélységi bejárás szükséges
- LAG(1): LL elemzés közben közvetlenül kiértékelhető, kézzel programozható

RAG

- Right AG
- Jobb AG:
  - az attribútumok jobbra balra történő mélységi bejárással kiszámíthatók
- RAG(k): k db mélységi bejárás szükséges
- RAG(1): kézzel programozható

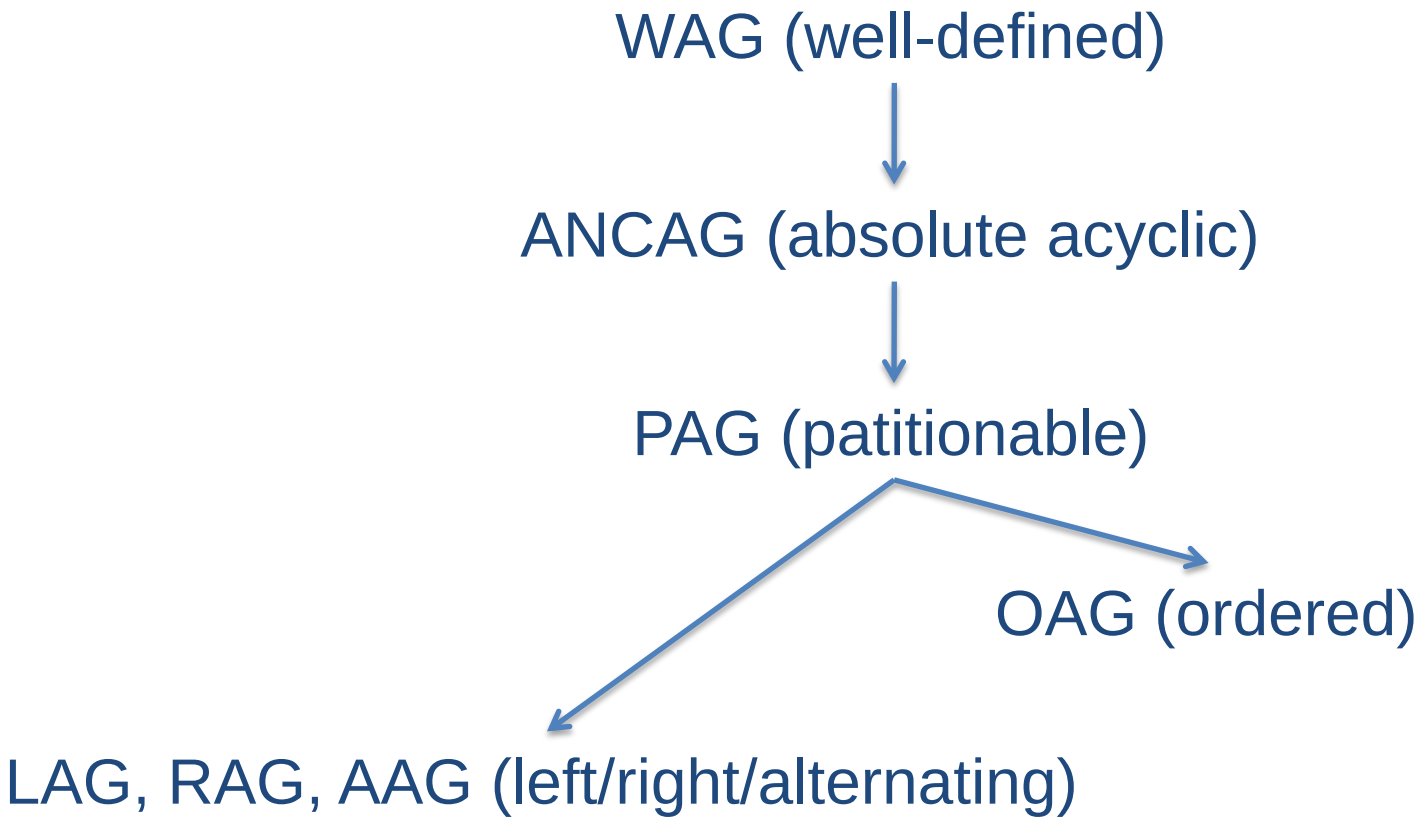
AAG

- Alternating AG
- Alternáló AG:
  - az attribútumok felváltva balról-jobbra és jobbról-balra történő mélységi bejárásokkal kiértékelhetők
- AAG(k): LAG(1)+RAG(1) k-szori végrehajtása
- Kézzel programozható
- Régen volt fontos: mágnesszalagok

Alkalmazás

- Egy egyfázisú fordítóhoz (szintaktikai+szemantikai elemzés és kódgenerálás azonnal) LAG(1) szükséges
  - pl. Züricher Pascal, Modula, Obera
- Majdnem minden nyelvre k = 2, 3 vagy 4
  - magasabb k általában csak néhány elemre lokálisan szükséges
  - az OAG ezért hatékonyabb:
    - megspórolja a sok bejárást
    - könnyebb tervezni is
- LAG(k)-ban illetve AAG(k)-ban való gondolkodás rossz fejlesztéshez vezet:
  - a k-t először túl kicsire választják (Pascal esetén k=4 szükséges)
  - a javításhoz újratervezés szükséges, hogy az attribútumokat a megfelelő csoportokba lehessen osztani

AG kiértékelési kategóriák



Névelemzés

Névelemzés

- Azonosítók definíciói
  - változók, függvények, paraméterek, stb.
    - felhasználó (programozó) által definiált
    - előre definiált (pl. Pascal: WriteLn() függvény)
- Azonosítók felhasználása
  - változók, konstansok, paraméterek, struktúrák/objektumok mezői, függvényhívások, metódushívások
  - speciális esetek: címkék, Pascal with-kifejezés
- Össze kell rendelni a felhasználást a definícióval a megfelelő érvényességi tartományban
- Névterek

Névterek

- Globális definíciók
- Modul/osztály/struktúra definíciók
- Helyi definíciók (lokális változók, paraméterek, blokkok)
- Egyéb

Alapséma

- block ::= declarations statements
  - statements.nameDefs = append(declarations.nameDefs, block.nameDefs)
- declarations ::= declarations ‘;’ declaration
  - declarations[1].nameDefs = append(declarations[2].nameDefs, declaration.nameDefs)
- declaration ::= name ‘:’ type
  - declaration.nameDefs = new Scope(name.symbol, type.typeUse)
- statements ::= statements ‘;’ statement
  - statements[2].nameDefs = statements[1].nameDefs
  - statement.nameDefs = statements[1].nameDefs
- statement ::= … variable …
  - variable.nameUse = statement.nameDefs.lookup(variable.symbol)

inherited

synthetized

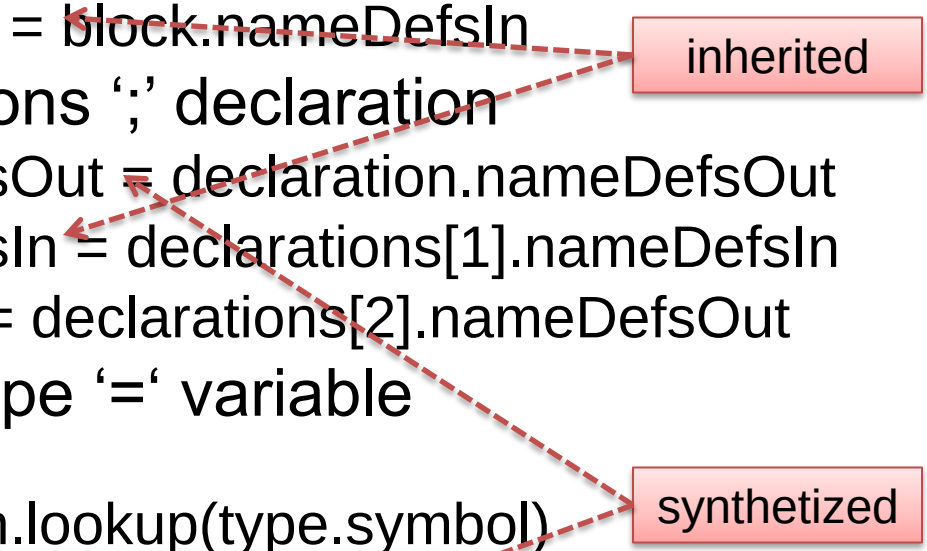
Honnan jön?

nameDef: inherited+synthetized

- Problémák:
  - egyszerre csak egyik fajta lehet
  - a definíciók hatóköre különbözhet, így használatkor ügyelni kell rá:
    - csak a megelőző definíciók használhatók (pl. Java: lokális változók inicializálása)
    - minden blokkban lévő definíció használható (pl. Java: ugyanazon osztályon belüli metódusok meghívása)
- Kétféle attribútum:
  - nameDefsIn (intherited)
    - vagy az adott blokk és környezete minden definíciója
    - vagy az adott blokkban lévő megelőző definíciók
    - vagy keverék
  - nameDefsOut (synthetized)

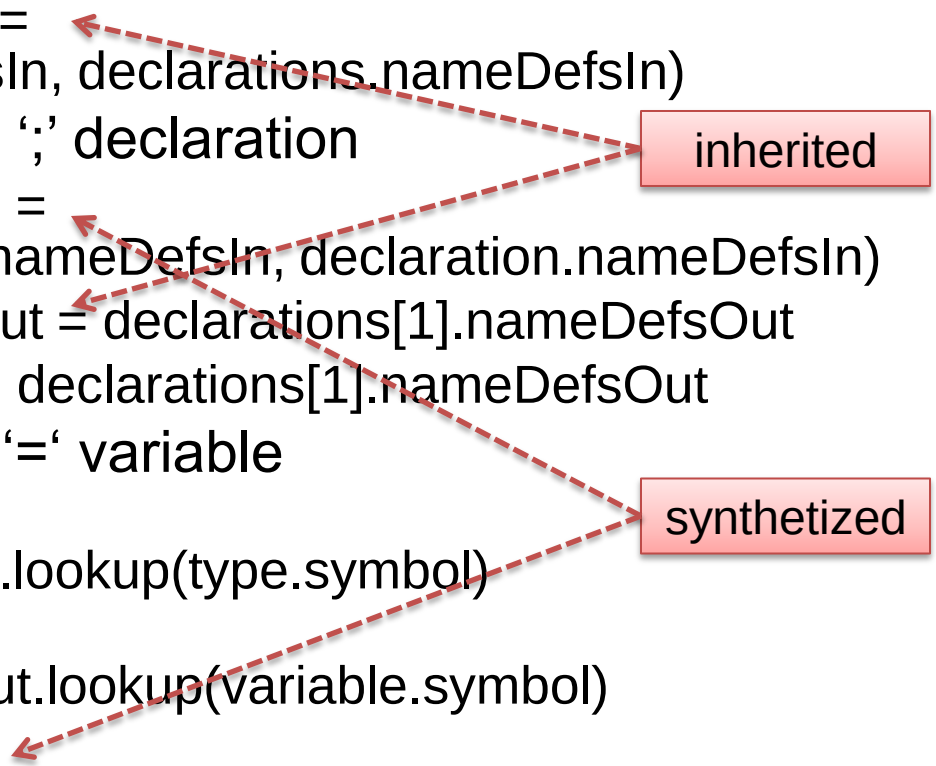
Alapséma: csak a megelőző definíciók

- block ::= declarations statements
  - statements.nameDefsIn = declarations.nameDefsOut
  - declarations.nameDefsIn = block.nameDefsIn
- declarations ::= declarations ‘;’ declaration
  - declarations[1].nameDefsOut = declaration.nameDefsOut
  - declarations[2].nameDefsIn = declarations[1].nameDefsIn
  - declaration.nameDefsIn = declarations[2].nameDefsOut
- declaration ::= name ‘:’ type ‘=‘ variable
  - type.typeUse = declaration.typeDefsIn.lookup(type.symbol)
  - variable.nameUse = declaration.nameDefsIn.lookup(variable.symbol)
  - declaration.nameDefsOut = append(new Scope(name.symbol, type.typeUse), declaration.nameDefsIn)



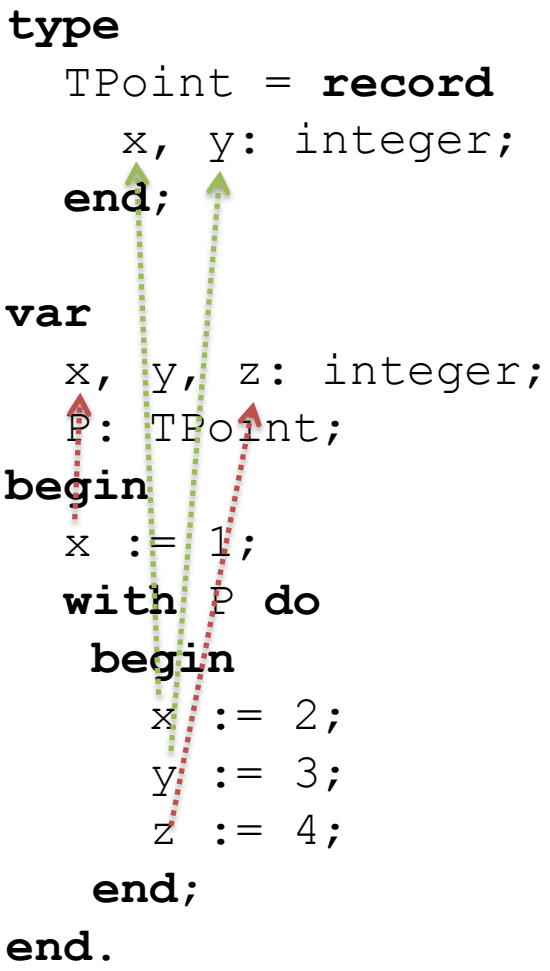
Alapséma: minden blokkban lévő definíció

- block ::= declarations statements
  - statements.nameDefsIn = declarations.nameDefsOut
  - declarations.nameDefsOut = append(block.nameDefsIn, declarations.nameDefsIn)
- declarations ::= declarations ‘;’ declaration
  - declarations[1].nameDefsIn = append(declarations[2].nameDefsIn, declaration.nameDefsIn)
  - declarations[2].nameDefsOut = declarations[1].nameDefsOut
  - declaration.nameDefsOut = declarations[1].nameDefsOut
- declaration ::= name ‘:’ type ‘=‘ variable
  - type.typeUse = declaration.typeDefsOut.lookup(type.symbol)
  - variable.nameUse = declaration.nameDefsOut.lookup(variable.symbol)
  - declaration.nameDefsIn = new Scope(name.symbol, type.typeUse)



Pascal with kulcsszó

- A környezeten kívül az adott típus névterét is meg kell nyitni
- with A, B do begin … x … end
  - a legutolsó definíció számít
- Keresési sorrend:
  - 1. struktúra névtere
  - 2. környezet
- Analógia:
  - osztályon belül függvény: lokális változók, paraméterek, attribútumok, ősosztályok attribútumai



Névelemzés eredménye

- Minden definícióra egy bejegyzés a definíciós táblában
- A felhasználásoknál hivatkozások a definíciókra
- Definíciós tábla:
  - a fordító „adatbázisa”
  - definíciós bejegyzések strukturálatlan halmaza
- Több tábla is lehetséges a névütközések elkerülésére:
  - attribútumok
  - metódusok

Előre definiált nevek

- 1. Ha a programnyelven leírhatók: bejegyzés a definíciós táblába
- 2. Különböző típusú paraméterekre generikus függvények (pl. abs):
  - ha a nyelven belül definiálható: 1. eset
  - egyébként: különleges esetként a fordítón belül generikus paraméterezés
- 3. Változó hosszúságú paraméterlisták:
  - ha a nyelven belül definiálható: 1. eset
  - egyébként:
    - a fordítón belül bevezetni a változó hosszú paraméterlistát
    - a szintaxisfa áttranszformálása véges hosszú paraméterlistából álló függvények hívására

Névelemzés összefoglalás

- Definíciók és hivatkozások összegyűjtése
- A hivatkozás maradandó: a transzformációs fázisban is szükség lesz rá
- A névelemzés függ a típuselemzéstől:
  - kvalifikált neveknél (pl. P.x) szükség van a kvalifikátor típusára
  - öröklésnél szükséges az ősosztályok attribútumainak és függvényeinek ismerete
  - genericitásnál szükséges a típusparaméter ismerete
  - a függvényekre való hivatkozás függhet a szignatúrától (overloading)
- Operátorazonosítás:
  - szignatúrafüggő függvények azonosítása (pl. egész vagy valós összeadás)

Típuselemzés

Típuselemzés

- Típus: objektumok megkülönböztetése az értékkészlet és a műveletek alapján
- Implicit típuskonverzió (pl. int → double)
- Típuselemzés feladata:
  - nevek, operandusok és kifejezések típusának meghatározása
    - névelemzéshez és operátorazonosításhoz
    - típuskonformitás ellenőrzéséhez
    - konzisztenciavizsgálathoz

Nyelvek típusossága

- Erősen típusos
  - statikus vagy dinamikus
  - pl. Pascal, Java, C#
  - általában bizonyos korlátozásokkal
  - funkcionális nyelvek típuskikövetkeztetéssel
- Gyengén típusos
  - pl. C, C++
    (pl. int logikai feltételként, pointerek cast-olása, stb.)
- Típusmentes
  - pl. gépi kód, néhány dinamikus nyelv

Egységes hozzáférés elve

- Az adatokhoz való hozzáférés legyen független az implementáció módjától
- Cél: a felhasználás és az implementáció szétválasztása
- Következmények:
  - változóhozzáférés és paraméter nélküli függvény hívása: tehát NEM f(), hanem f
  - tömb elemeinek elérése: a[i,j] helyett a(i,j), mint a Fortranban is
- Példák:
  - Pascal: paraméter nélküli függvények
  - C#: property
  - C++: operator overloading

Típusekvivalencia

- Névegyezés:
  - két típus megegyezik, ha ugyanazzal a típusdefinícióval definiálták őket
  - pl. modern OO nyelvek osztályai
- Struktúraegyezés:
  - két típus megegyezik, ha ugyanazzal a típuskonstruktorral, ugyanazokkal a típusargumentumokkal definiálták őket
  - pl. típus alias-ok, struktúrák, funkcionális nyelvek
  - vigyázat: a típusok lehetnek rekurzívak, a kifejezések kifejtve végtelenek is lehetnek
  - strukturális egyeztetés: nehéz

Típusattribútumok

- *a priori* típus: szintetizált (type)
- *a posteriori* típus: örökölt (expectedType)
- Közöttük: típuskompatibilitás vizsgálata
- A típusokat célszerű a definíciós táblába is bejegyezni a nevekhez:
  - lefoglalt memóriaméret szempontjából fontos
- Egyébként a névelemzés, operátorazonosítás, típuskompatibilitás és konzisztenciaellenőrzésen kívül nincs rá többé szükség
  - kivéve: dinamikus típusellenőrzés és polimorfizmus

Típuselemzés: operátorok

- 1. op1 τ op2 operandusai típusának meghatározása
- 2. τ lehetséges definícióinak meghatározása
- 3. a definíciók közül a megfelelő kiválasztása, vagy hibajelzés (ha nincs definíció vagy egynél több alternatíva van)
- 4. op1 és op2 elvárt típusának meghatározása
- 5. a típusok és elvárt típusok kompatibilitásának vizsgálata és feljegyzése attribútumként

Típuselemzés

- Két eset:
  - A. az operátorazonosítás csak az operandusok típusától függ (a legtöbb programnyelv ilyen)
  - B. az operátorazonosítás az eredmény típusától (pl. értékadás bal oldala) is függ (pl. Ada)
- A. eset:
  - 1-5. lépés lentről felfelé történő kiértékelése
- B. eset:
  - 1. és 2. lépés lentről felfelé (típusok halmazának meghatározása)
  - 3-5. lépések fentről lefelé (elvárt típus és operátor meghatározása: egyértelműnek kell lenniük)

Típusok konverziója

- Implicit típuskonverzió
  - pl. int → double
- Implicit dereferencia
  - pl. &int → int
- Implicit deprocedurálás
  - paraméter nélküli függvények

Tipikus típusellenőrző függvények

- getBaseType: Type → Type
  - visszaadja az alaptípust (pl. tömb)
- getContentType: Type → Type
  - visszaadja a tartalmazott típust (pl. referencia)
- equivalent: Type x Type → boolean
  - a két típus strukturálisan megegyezik
- coercible: Type x Type → boolean
  - az első típus a másikra konvertálható
- balance: Type x Type → Type
  - a két típus implicit dereferenciálással és deprocedurálással közös típusra hozható (típusegyeztetés)

Típusegyeztetés

- Két adott típus egy közös típusra hozása
- Alaptípusok:
  - egyszerű típusok, tömbök, rekordok, paraméteres függvények, stb.
  - egyeztetés: ha megegyeznek vagy implicit típuskonverzió lehetséges (pl. int → double)
- Származtatott típusok:
  - referenciák, paraméter nélküli függvények, pl. ref t, proc t, ref ref t, ref proc t, stb.
  - egyeztetés:
    - 1. alaptípus meghatározása
    - 2a. ha az alaptípusok egyeznek, minimális számú előtagot (ref, proc) kihúzni
    - 2b. ha az alaptípusok nem egyeznek, akkor egyiket a másikra konvertálni, vagy fordítva, vagy egy harmadik típusra, vagy hibajelzés

Típusegyeztetés példa

- Adottak az alábbiak:
  - `x: proc int;`
  - `y: ref proc int;`
  - `z: proc proc int;`
  - `x = expr ? y : z;`
- Kérdés: mi `y` és `z` egyeztetett típusa?
  - `getBaseType(y) = getBaseType(z) = int`
  - 2a. eset: `y`-ból egy `ref`, `z`-ből egy `proc` áthúzása
  - eredmény: `proc int`
  - vagyis: `y` referenciáját fel kell oldani, `z`-t pedig egy mélységig kell meghívni

Operátorazonosítás

- Az operátorazonosítás ekvivalens az azonos nevű, de különböző szignatúrájú függvények névanalízisével (overloading)

Típuselemzés példa

- ifStatement ::= ‘if’ ‘(‘ expr ‘)’ stmt ‘else’ stmt
  - expr.expectedType = Types.BOOLEAN_TYPE
- cmpExpr ::= expr smallerOp expr
  - cmpExpr.type = Types.BOOLEAN_TYPE
  - expr[1].expectedType = smallerOp.type
  - expr[2].expectedType = smallerOp.type
  - smallerOp.expectedType = balance(expr[1].type, expr[2].type)
- smallerOp ::= ‘<‘
  - smallerOp.type = smallerOp.expectedType
- expr
  - noTypeErrors = coercible(expr.type, expr.expectedType)

Szemantikai elemzés