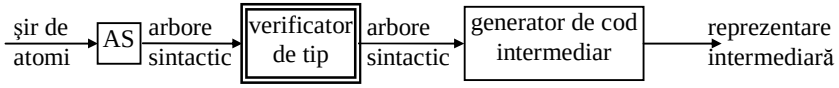
Verificarea tipurilor

Un compilator trebuie să asigure verificarea completă a programului sursă, adică atât respectarea convențiilor sintactice cât şi a celor semantice, corespunzătoare programului respectiv. Verificarea semantică realizată în timpul compilării se numeşte **verificare statică**, pentru a fi distinsă de cea **dinamică**, efectuată în timpul execuției programului obiect. Exemple de verificări statice sunt:

1. **Verificări de tip**. Un compilator trebuie să semnaleze eroare dacă operanzii sunt incompatibili cu operatorul aplicat; de exemplu, adunarea a două articole.
2. **Verificări de flux de control**. Instrucțiunilor care determină ca fluxul de control al programului să părăsească o anumită construcție trebuie să li se poată asocia o locație la care să se transfere controlul. De exemplu, instrucțiunea **break** din C determină părăsirea celui mai interior **while**, **for** sau **switch** şi se va semnala eroare dacă nu există o astfel de instrucțiune înconjurătoare.
3. **Verificări de unicitate**. Se efectuează în situațiile în care un obiect trebuie să fie definit sau declarat o singură dată. Exemple din Pascal:
   – identificatorii, în general;
   – etichetele, inclusiv cele dintr-o instrucțiune **case** (trebuie să fie distincte).
4. **Verificări legate de nume**. Există limbaje în care acelaşi nume trebuie să apară de două sau mai multe ori. De exemplu, în Ada, un ciclu **loop** sau **bloc** poate avea un nume care apare atât la începutul cât şi la sfârşitul construcției; compilatorul trebuie să verifice utilizarea aceluiaşi nume în ambele locuri.

După cum rezultă şi din exemplele de mai sus, majoritatea celorlalte verificări statice sunt de rutină şi pot fi implementate utilizând tehnicile din capitolul precedent. Unele dintre aceste verificări pot fi incluse în alte activități. De exemplu, la introducerea în TS a informațiilor despre un identificator, se poate verifica dacă el este unic declarat.

În multe compilatoare Pascal, verificările statice şi generarea de cod intermediar se combină cu AS într-o singură trecere. În cazul unor limbaje cu construcții mai complexe, cum sunt cele din Ada, este de preferat ca pentru verificări de tip să se prevadă o trecere separată, situată între AS şi generarea de cod intermediar, ca în figură.



Poziția unui verificator de tip

Un verificator de tip (VT) verifică dacă tipul unei construcții corespunde celui aşteptat de context. Exemple din Pascal:

– operatorul **mod** se aplică numai la operanzi întregi;
– operația de dereferire se aplică numai unui pointer;
– indexarea se face numai pentru tablouri;
– corespondența ca număr şi tip a parametrilor formali cu cei actuali la declararea şi respectiv la apelul procedurilor şi funcțiilor.

Informația de tip culeasă de un VT este necesară la generarea de cod. De exemplu, operatorul aritmetic + se aplică unor operanzi întregi sau reali şi chiar altor tipuri (mulțimi, şiruri de caractere). În acest caz trebuie luat în considerare contextul lui + pentru a determina sensul dorit şi pentru a genera codul corect. Un simbol care poate reprezenta diferite operații în contexte diferite se numeşte "supraîncărcat". Supraîncărcarea poate fi însoțită de constrângeri de tip (exemplu operanzii micşti la unele operații aritmetice), caz în care compilatorul generează un cod special de conversie a unui operand la tipul aşteptat de context.

O noțiune distinctă de cea de supraîncărcare este "polimorfismul" aplicat funcțiilor. Corpul unei funcții polimorfice poate fi executat cu argumente de tipuri diferite. De exemplu, definirea operațiilor asupra unei stive, indiferent de tipul elementelor din stivă.

Sisteme de tipuri

Proiectarea unui VT pentru un limbaj se bazează pe informații despre unele construcții sintactice ale limbajului şi anume:

– definițiile de tipuri;
– regulile pentru asignarea de tipuri unor construcții de limbaj.

Exemple de astfel de informații, necesare proiectantului de compilatoare, extrase din raportul Pascal şi respectiv din manualele de referință pentru C:

– "Dacă ambii operanzi ai operatorilor aritmetici de adunare şi înmulțire sunt de tip întreg, atunci rezultatul este de tip întreg".
– "Rezultatul operandului unar & este un pointer la obiectul referit de operand. Dacă tipul operandului este '...', atunci tipul rezultatului este 'pointer la ...'".

În cele două exemple este implicită ideea că fiecare expresie are un tip asociat cu ea. În plus, unele tipuri au structură; tipul "pointer la ..." este construit din tipul la care se referă "...".

Atât în Pascal cât şi în C tipurile sunt fie de bază, fie construite (definite) în program. Tipuri de bază sunt tipurile atomice fară structură internă din punctul de vedere al programatorului. Exemplu: tipurile logic, caracter, întreg şi real. Tipurile subdomeniu şi enumerare pot fi tratate şi ele ca tipuri de bază. Pascal permite unui programator să construiască noi tipuri din tipurile de bază şi din alte tipuri definite anterior. Exemple: tablouri, articole, mulțimi. În plus, se pot trata ca tipuri construite pointerii şi funcțiile.

Expresii de tip

Printr-o **expresie de tip** se indică tipul unei construcții de limbaj. O expresie de tip este fie un tip de bază fie este formată prin aplicarea unui operator numit **constructor de tip** asupra altor expresii de tip. Mulțimea de tipuri de bază și de constructori depinde de limbajul de verificat.

În continuare se utilizează următoarea definiție pentru expresii de tip (ET):

1. Un tip de bază este o ET. Se pot admite ca tipuri de bază cele din Pascal: *boolean*, *char*, *integer* și *real*. Se introduce, de asemenea, un tip de bază numit *tip_eroare* prin care se va semnala o eroare în timpul verificării tipurilor și un tip de bază denumit *vid*, semnificând "absența tipului", care permite verificarea instrucțiunilor.
2. Deoarece tipurile pot avea nume, rezultă că un nume de tip reprezintă, de asemenea, o ET.
3. Un constructor de tip aplicat unor ET este, de asemenea, o ET.

Ca și constructori se admit:

a) *Tablouri*. Dacă T este o ET, atunci *array*(I,T) este o ET reprezentând un tablou cu elemente de tip T și mulțimea de indici I. Foarte frecvent, I este un domeniu de întregi. De exemplu, declarația Pascal

```
var A: array[1 .. 10] of integer;
```

asociază lui A, ET *array*(1 .. 10, integer).

b) *Produse*. Dacă T_1 și T_2 sunt ET, atunci produsul lor cartezian $T_1 \times T_2$ este o ET. Presupunem că operandul $\times$ este asociativ la stânga.

c) *Articole*. Tipul unui articol (înregistrare) poate fi considerat ca produsul tipurilor câmpurilor sale cu diferența că la un articol câmpurile pot avea nume care vor apărea în ET. Verificarea tipului pentru înregistrări se poate face utilizând o ET formată prin aplicarea constructorului *record* unui produs ai cărui factori sunt alcătuiți din numele de câmpuri și tipurile asociate lor. De exemplu, fragmentele de program Pascal:

```
type tipsir = record
                adresa: integer;
                sir: array[1 .. 15] of char
              end;
var tablou : array[1 .. 101] of tipsir;
```

declară numele de tip *tipsir* pentru care se construiește următoarea ET:

record ((adresa×integer)×(sir×array(1 .. 15,char)))

și variabila *tablou*, ca fiind un tablou de înregistrări de acest tip.

d) *Pointeri*. Dacă T este o ET, atunci *pointer*(T) este o ET reprezentând tipul "pointer la un obiect de tip T". De exemplu, declarația Pascal:

var p: ↑ tipșir; declară variabila *p* cu tipul *pointer*(tipsir).

e) *Funcții*. Din punct de vedere matematic, o funcție pune în corespondență elementele unei mulțimi, *domeniul*, cu elementele altei mulțimi, *codomeniu*. Funcțiile într-un limbaj de programare, se pot trata ca punând în corespondență un **tip domeniu** D cu un **tip codomeniu** C. Tipul unei astfel de funcții va fi reprezentat prin ET: D→C. De exemplu, funcția intrinsecă **mod**, din Pascal, are tipul domeniu **int×int** și tipul codomeniu **int**. ⇒ET pentru funcția **mod** este:

int×int→int

Se presupune că operatorul $\times$ este mai prioritar decât operatorul $\rightarrow$ și, de asemenea, operatorul $\rightarrow$ esste asociativ la dreapta.

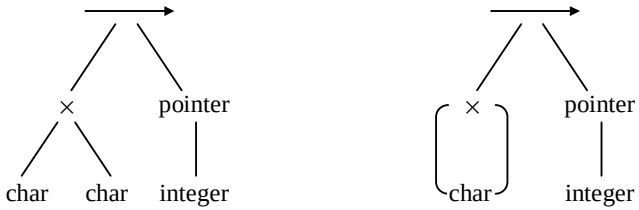
Alt exemplu: **function** f(a,b: char) : ↑ integer;

⇒ ET : char×char → *pointer*(integer)

Din motive de implementare, există deseori limitări asupra tipului returnat de o funcție; de exemplu, în mod frecvent, nu se pot returna tablouri sau alte funcții. Există însă și limbaje, între care un exemplu elocvent este LISP, care permit ca funcțiile să returneze obiecte de orice tip, inclusiv alte funcții. Exemplu:

$$g : \underbrace{(\text{integer} \rightarrow \text{integer})}_{\text{domeniu=funcție}} \rightarrow \underbrace{(\text{integer} \rightarrow \text{integer})}_{\text{codomeniu=tot o funcție}}$$

4. ET pot conține variabile ale căror valori sunt de asemenea, ET. O modalitate potrivită pentru reprezentarea unei ET este utilizarea unui graf. Pornind de la abordarea dirijată de sintaxă din capitolul precedent se poate construi un arbore sau un GOA pentru o ET, cu nodurile interioare pentru constructori de tip și frunzele pentru tipuri de bază, nume de tipuri și variabile de tip.



Arbore și GOA pentru char×char→pointer (integer)

Definirea unui sistem de tipuri

Un **sistem de tipuri** (ST) este o colecție de reguli care permit asignarea de ET diferitelor părți ale unui program. Un VT implementează întotdeauna un anumit sistem de tipuri. Sistemele de tipuri care urmează sunt specificate într-o modalitate dirijată de sintaxă. ⇒ pot fi uşor implementate utilizând tehnicile din capitolul precedent.

Sistemele de tipuri pot să fie diferite de la un limbaj la altul şi, de asemenea, pot să fie aceleaşi sau diferite în diverse compilatoare sau procesoare pentru acelaşi limbaj. De exemplu, în Pascal, tipul unui tablou include mulțimea de indici a tabloului ⇒o funcție cu argument tablou se poate aplica numai la tablouri cu acea mulțime de indici. Multe compilatoare Pascal, însă, permit ca mulțimea de indici să rămână nespecificată când tabloul este transmis ca argument. Deci, aceste compilatoare utilizează un sistem de tipuri diferit de cel din definiția limbajului Pascal. Similar, în sistemul UNIX, comanda **lint** examinează programele C pentru erori posibile, utilizând un sistem de tipuri mai detaliat decât cel implementat în compilatorul pentru C.

Verificarea statică şi dinamică a tipurilor

Verificarea realizată de un compilator se numeşte **statică** iar cea efectuată când se execută programul obiect se numeşte **dinamică**. În principiu, orice verificare de tip poate fi făcută dinamic, dacă în codul obiect se păstrează tipul unui element împreună cu valoarea acelui element.

Un sistem de tipuri se numeşte **consistent** dacă elimină necesitatea verificărilor dinamice pentru erorile de tip, garantând de la compilare (static) că aceste erori nu pot să apară la execuția programului. Dacă un astfel de sistem de tipuri asignează un tip≠*tip_eroare* unei părți de program atunci se garantează că nu pot să apară erori de tip când se rulează codul obiect corespunzător acelei părți de program. Un limbaj de programare este **puternic tipizat** dacă prin compilatorul său se poate garanta că programele pe care le acceptă se pot executa fără erori de tip.

În practică totuși, unele verificări pot fi făcute numai dinamic.

De exemplu: **var** tablou : **array**[0 .. 255] **of** char;
i:integer;

La utilizarea elementului tablou [i], compilatorul nu poate garanta, în general, că în timpul execuției, valoarea lui *i* se încadrează în domeniul 0 .. 255.

Specificarea unui verificator de tip, simplu

Se va descrie un VT pentru un limbaj simplu, în care tipul fiecărui identificator trebuie declarat înainte de utilizarea identificatorului respectiv. VT este o ST care sintetizează tipul fiecărei expresii din tipurile subexpresiilor componente. Se pot trata în această manieră tablouri, pointeri, instrucțiuni şi funcții.

Prezentarea limbajului

P→D; E
D→D; D | **id** : T | ε
T→**char** | **integer** | **array** [**num**] **of** T | ↑T
E→**literal** | **num** | **id** | E **mod** E | E[E] | E↑

Gramatica limbajului sursă

Programele generate de această gramatică, reprezentate prin neterminalul P, constau dintr-o secvență de declarații D, urmată de o singură expresie E. Un exemplu de program poate fi următorul:

cheie : integer;
cheie **mod** 1991

Limbajul are două tipuri de bază, *char* şi *integer*. Pentru semnalarea erorilor se va utiliza, de asemenea ca tip de bază, şi *tip_eroare*. Pentru simplitate, să presupunem că indicii tablourilor încep întotdeauna cu 1. De exemplu, "**array**[255] **of** char" conduce la ET: *array*(1 .. 255,char). Ca şi în Pascal, operatorul prefixat ↑ din declarații construieşte un tip pointer, aşa încât "↑integer" conduce la ET: *pointer*(integer).

Pe baza gramaticii se defineşte o ST care indică modul de stabilire a tipului unui identificator:

P→D; E
D→D; D
D→**id** : T {adaugătip (**id.**intrare, T**.**tip)}
D→ε
T→**char** {T**.**tip:=char}
T→**integer** {T**.**tip:=integer}
T→↑T_1 {T**.**tip:=pointer (T_1**.**tip)}
T→**array** [num] **of** T_1 {T**.**tip:=array (1 .. num.val,T_1**.**tip)}

Partea din ST care memorează tipul unui identificator

Acțiunea asociată cu producția D→**id** : T introduce un tip în intrarea TS pentru identificatorul respectiv. Ea este aplicată atributului sintetizat *intrare* (pointer în TS pentru **id**) şi unei expresii de tip reprezentată prin atributul sintetizat *tip* al neterminalului T.

Verificarea tipului expresiilor

În regulile care urmează, atributul sintetizat *tip* al neterminalului E furnizează expresia de tip asignată de sistemul de tipuri expresiei generate de E. Următoarele RS precizează că o constantă reprezentată prin atomul **literal** are tipul *char* iar una reprezentată prin **num** are tipul *integer*.

E→**literal** {E.tip:=char} E→**num** {E.tip:=integer}

Pentru a obține tipul înregistrat în intrarea *i* a TS se va apela o funcție *extrtip(i)*. De exemplu, dacă într-o expresie apare un identificator, tipul său declarat este extras şi asignat atributului **tip**, astfel:

E→**id** {E.tip:=extrtip (**id.**intrare)}

Cazul unei expresii formate prin aplicarea operatorului **mod**:

$E \rightarrow E_1$ **mod** E_2 {E.tip:=**if** E_1.tip=integer **and** E_2.tip=integer **then** integer **else** tip_eroare}

Cazul unui element de tablou $E_1[E_2]$:

$E \rightarrow E_1[E_2]$ {E.tip:=**if** E_2.tip=integer **and** E_1.tip=array(s,t) **then** t **else** tip_eroare}

Cazul unui pointer:

$E \rightarrow E_1\uparrow$ {E.tip:=**if** E_1.tip=pointer(t) **then** t **else** tip_eroare}

În mod analog, se pot adăuga producții şi reguli pentru alte tipuri şi operații în calculul expresiilor. De exemplu, prin introducerea producției T→**boolean** în gramatica limbajului sursă (prezentată anterior), identificatorii pot avea şi tipul boolean iar prin introducerea operatorilor relaționali şi a celor logici în producțiile pentru E s-ar permite şi construirea unor expresii de tip boolean.

Verificarea tipului instrucțiunilor

Deoarece instrucțiunile, privite ca şi construcții de limbaj nu au, în general, valori, lor li se poate atribui tipul de bază special numit *vid*. Dacă însă, într-o instrucțiune se detectează o eroare, tipul asignat va fi *tip_eroare*.

În continuare se consideră instrucțiunea de atribuire precum şi instrucțiunile **if** şi **while**. Instrucțiunile sunt separate prin **;**. Referitor la gramatica limbajului sursă (prezentată anterior), se schimbă producția pentru un program complet în **P→D;I**. Un program constă acum din declarații, urmate de instrucțiuni. Regulile pentru verificarea expresiilor rămân valabile deoarece instrucțiunile conțin expresii. La acestea se adaugă regulile specifice pentru verificarea instrucțiunilor date în figură.

I→**id**:=E{I.tip:=**if** id.tip=E.tip **then** vid **else** tip_eroare}

I→**if** E **then** I_1{I.tip:=**if** E.tip=boolean **then** I_1.tip **else** tip_eroare}

I→**while** E **do** I_1{I.tip:=**if** E.tip=boolean **then** I_1.tip **else** tip_eroare}

I→I_1; I_2{I.tip:=**if** I_1.tip=vid **and** I_2.tip=vid **then** vid **else** tip_eroare}

Schemă de traducere prin verificarea tipului instrucțiunilor

Prima regulă verifică dacă partea stângă şi partea dreaptă a unei instrucțiuni de atribuire au acelaşi tip. Construcția de limbaj este puternic tipizată şi nu conține constrângeri de tip (spre deosebire de Pascal, unde se permite conversia întreg→real). Erorile sunt propagate de ultima regulă deoarece o secvență de instrucțiuni are tipul *vid* dacă fiecare instrucțiune în parte are tipul *vid*. Un VT complet şi practic va trebui, desigur, să completeze aceste reguli aşa încât să se poată raporta şi locul erorilor de tip.

Verificarea tipurilor funcțiilor

Un apel de funcție, care este o expresie, se poate reda prin producția: **E→E(E).** Pentru a permite în declarații şi tipuri de funcții, regulile corespunzătoare neterminalului T trebuie completate cu următoarea producție:

$T \rightarrow T_1$'→'T_2 {T.tip:=T_1.tip→T_2.tip}

'→' este constructorul de funcție şi se deosebeşte de metasimbolul →.

Regula pentru verificarea tipului unui apel de funcție este:

$E \rightarrow E_1(E_1)$ {E.tip:=**if** E_2.tip=s **and** E_1.tip=s→t **then** t **else** tip_eroare}

Această regulă exprimă şi verificarea tipului argumentului.

Generalizarea la funcții cu mai multe argumente se poate face construind un tip produs, constând din argumentele respective care apoi să fie tratate împreună, ca un singur argument.

Exemplu declarația Pascal:

function f (**function** g (real): real; x: real): real

căreia i se poate asocia următoarea ET:

(real→real)×real→real