Exemple de aplicare a tehnicilor de compilare dirijate de sintaxă

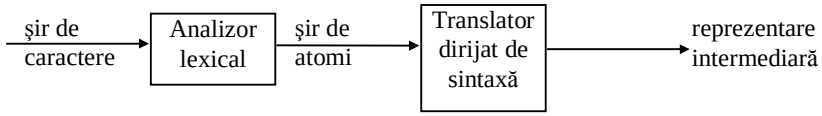
Primul exemplu este proiectarea şi realizarea unui program complet care să traducă expresii *infix* în forma *postfix*. Programul rezultat va reprezenta întreaga parte de *front-end* a unui compilator, adică va avea la intrare un şir de caractere reprezentând expresia iniţială şi va genera un cod intermediar. Tehnica de compilare de bază care va fi utilizată este o "traducere dirijată de sintaxă".

Exemplu: transformare "infix"→"postfix"

9–5+2→ 9 5–2+

Notaţia postfix prezintă avantajul că poate fi transformată direct în cod pentru un calculator abstract care utilizează o stivă pentru efectuarea calculelor. Se va construi un program simplu care tratează expresii constând din cifre şi operatorii + şi –.

Structura generală a programului compilator ce va fi proiectat este:



Translatorul dirijat de sintaxă va fi un analizor sintactic combinat cu un generator de cod intermediar. Analizorul lexical este foarte simplu întrucât fiecare caracter din intrare este un atom.

Definirea notaţiei postfix

Notaţia postfix pentru o expresie E poate fi definită inductiv astfel:

1. Dacă E este o variabilă sau o constantă atunci notaţia *postfix* pentru E este chiar E.
2. Dacă E este o expresie de forma E_1 **op** E_2, unde **op** este orice operator binar atunci notaţia postfix pentru E este E'_1 E'_2 **op**, unde E'_1 şi E'_2 sunt notaţiile postfix pentru E_1 şi respectiv E_2.
3. Dacă E este o expresie de forma (E_1) atunci notaţia postfix pentru E_1, este, de asemenea, şi notaţie postfix pentru E.

Observaţie: În notaţia postfix nu sunt necesare paranteze deoarece poziţia şi aritatea (numărul de argumente) pentru operatori conduc la un înţeles unic pentru expresia respectivă.

Exemplu: (9–5)+2 → 9 5–2+

9–(5+2) → 9 52+ –

Exemplu de translator pentru expresii simple

Specificarea iniţială a translatorului, ca schemă de traducere, este:

<expr>→	<expr>+<term>	{print ('+')}
<expr>→	<expr>–<term>	{print ('–')}
<expr>→	<term>	
<term>→	0	{print ('0')}
<term>→	1	{print ('1')}
....	....	....
<term>→	9	{print ('9')}

Gramatica inclusă în această schemă de traducere este recursivă la stânga, recursivitate care trebuie eliminată pentru a aplica translatarea descendentă recursivă. Tehnica de eliminare a recursivităţii de stânga este cea cunoscută, implicând însă şi includerea în producţii a acţiunilor semantice.

$$A \to A\alpha \mid A\beta \mid \gamma \quad \Rightarrow \begin{cases} A \to \gamma R \\ R \to \alpha R \mid \beta R \mid \varepsilon \end{cases}$$

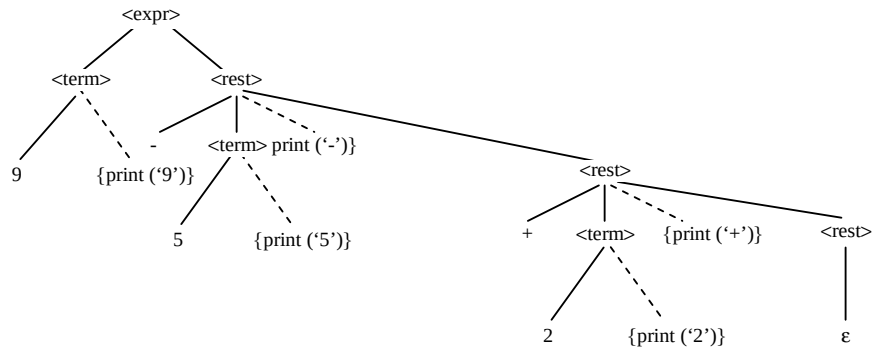
În exemplul nostru A=<expr>, α=+<term>{{print ('+')}

β= –<term>{print ('–')}, γ=<term>

Rezultă următoarea schemă de traducere transformată:

<expr>→	<term><rest>	
<rest>→	+<term>{print ('+')}<rest>	
	| –<term>{print ('–')}<rest>	
	| ε	
<term>→	0	{print ('0')}
....	....	....
<term>→	9	{print ('9')}

Traducerea efectivă a şirului "9–5+2" poate fi pusă în evidenţă prin traversarea *spre adâncime* a arborelui de mai jos:



Partea esențială a unui translator descendent recursiv pentru schema de traducere transformată o reprezintă funcțiile corespunzătoare celor trei neterminale: *<expr>*, *<term>*, *<rest>*, prezentate în continuare:

```
expr ()
{
    term (); rest ();
}
```

```
rest ()
        {
    if (simb= ='+') {
        corespunde ('+'); term (); putchar ('+'); rest ();
}
else    if (simb= ='−') {
        corespunde ('−'); term (); putchar ('−'); rest ();
        }
        else;
}
```

```
term ()
{
    if (isdigit (simb)) {
        putchar (simb); corespunde (simb);
    }
    else    eroare();
}
```

Funcția *corespunde* a fost prezentată în capitolele anterioare şi are rolul de a pune în corespondență un atom cu simbolul de anticipare şi de a avansa în şirul de intrare. În acest caz simplu fiecare atom este un singur caracter ⇒ funcția compară şi citeşte caractere.

Codul funcțiilor pentru un neterminal urmăreşte cu fidelitate conținutul părților drepte ale producțiilor şi conține câte o ramură distinctă pentru fiecare producție cu acel neterminal în partea stângă. Codul acestor funcții poate fi optimizat prin înlocuirea unor apeluri recursive prin iterații.

Când ultima instrucțiune într-un corp de procedură este un apel recursiv al aceleiaşi proceduri, apelul se numeşte **recursiv final**. Exemplu: apelurile lui *rest()* din funcția *rest()*.

Înlocuind *recursivitatea finală* prin iterație se poate creşte viteza programului. Pentru o procedură fără parametri un apel *recursiv final* poate fi înlocuit simplu, printr-un salt la începutul procedurii, ca mai jos:

```
rest ()
{
L : if (simb= ='+') {
        corespunde ('+'); termen (); putchar ('+'); goto L;
    }
    else    if (simb= ='−') {
                corespunde ('−'); termen (); putchar ('−'); goto L;
            }
            else;
}
```

Deoarece prin aceste modificări singurul apel la funcția *rest* rămâne cel din *expr*, cele două funcții pot fi cuplate într-una singură. În plus, ținând cont de faptul că în limbajul de programare C o instrucțiune oarecare "instr" poate fi executată repetat astfel:

```
        while (1) instr
```

ciclu din care se poate, eventual, ieşi cu break, codul final, comun, al funcțiilor *expr* şi *rest* devine:

```
expr ()
{
        term();
        while(1)
            if (simb= ='+') {
                corespunde ('+'); term (); putchar ('+');
            }
            else    if (simb= ='−') {
                        corespunde ('−'); term (); putchar ('−');
                    }
                    else break;
}
```

Programul C complet pentru translatorul proiectat este:

```
#include <ctype.h> /*încarcă fişierul cu predicatul isdigit*/
int simb;
main()
{
    simb=getchar();
    expr();
    putchar('\n'); /*adaugă un caracter newline*/
}
expr()
{
        term();
        while(1)
            if (simb= ='+') {
                corespunde ('+'); term (); putchar ('+');
            }
            else    if (simb= ='−') {
                        corespunde ('−'); term (); putchar ('−');
                    }
                    else break;
}
term ()
{
    if (isdigit (simb)) {
        putchar (simb); corespunde (simb);
    }
    else    eroare();
}
corespunde(t)
        int t;
{
        if (simb= =t)
            simb=getchar();
        else eroare();
}
eroare()
{
        printf ("eroare sintaxă \n"); /*tipărire mesaj de eroare*/
        exit (1); /*apoi opreşte*/
}
```


Încorporarea tabelei de simboluri

Tabela de simboluri (TS) este o structură de date utilizată pentru a înregistra informații despre diferite construcții ale limbajului sursă, în special despre identificatori (simboluri). În general, informațiile sunt determinate şi înregistrate în fazele de analiză ale compilatorului şi urmează să fie utilizate în fazele de sinteză, în scopul generării codului obiect. De exemplu, în timpul analizei lexicale, şirul de caractere sau lexema care formează un identificator este introdusă într-o intrare din TS. Fazele ulterioare ale compilatorului vor adăuga la această intrare alte informații: felul entității (exemplu: procedură, variabilă, etichetă etc.), tipul său, poziția în memorie etc. Faza de generare de cod va utiliza apoi aceste informații pentru a genera codul corect (de exemplu, pentru a memora sau a citi o variabilă).

Interfața cu tabela de simboluri

Interfața cu TS este asigurată de rutine care au ca roluri: salvarea şi regăsirea lexemelor. Când se înregistrează o lexemă se memorează totodată şi atomul asociat cu lexema respectivă. În exemplul care urmează, asupra TS se vor executa următoarele operații:

1) *inserează (s,t)*: returnează indexul noii intrări pentru şirul *s*, cu atomul *t*;
2) *caută (s)*: returnează indexul intrării pentru şirul *s*, sau *0*, dacă *s* nu este găsit în TS.

Analizorul lexical apelează operația *caută* pentru a determina dacă, în TS, există o intrare corespunzătoare lexemei respective. Dacă nu există o astfel de intrare, va apela operația *inserează* pentru a o crea. Formatul intrărilor în TS va fi cunoscut şi accesibil atât analizorului lexical cât şi celui sintactic.

Tratarea cuvintelor cheie

Rutinele de interfață cu TS de mai sus pot trata în aceeaşi manieră şi cuvintele cheie. Pentru exemplificare să considerăm atomii **div** şi **mod**, având lexemele “div” şi respectiv “mod”. Ele vor fi introduse în TS la inițializare, utilizând apelurile:

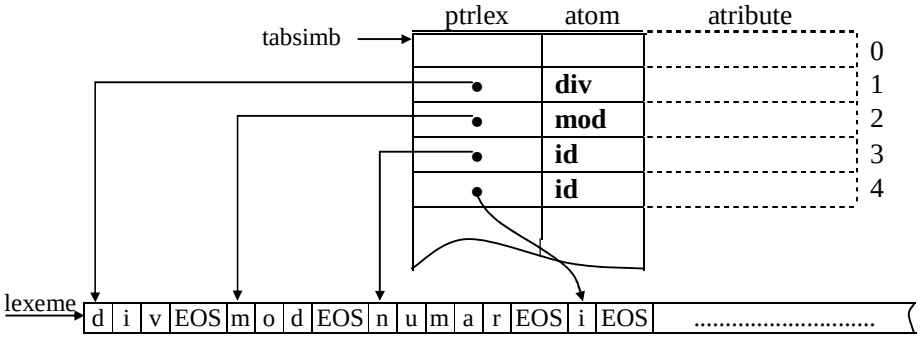
inserează (“div”,**div**); *inserează* (“mod”,**mod**);

Orice apel ulterior *caută* (“div”) returnează atomul **div**, ceea ce va asigura ca *div* să nu poată fi utilizat ca identificator obişnuit.

În acest mod, prin inițializarea corespunzătoare a TS, se poate trata orice colecție de cuvinte rezervate.

Implementarea tabelei de simboluri

Pentru implementarea practică a TS, s-a ales următoarea structură de date:



Pentru a nu limita lungimea identificatorilor, lexemele ce formează identificatorii sunt înregistrați într-un tablou distinct: *lexeme*. Şirul ce reprezintă un identificator se termină cu un caracter special de sfârşit de şir (EOS), care nu poate apărea în componența identificatorului. Câmpul *ptrlex* indică spre începutul lexemei corespunzătoare acelei intrări din TS. Valorile atributelor se pot reține în câmpuri suplimentare.

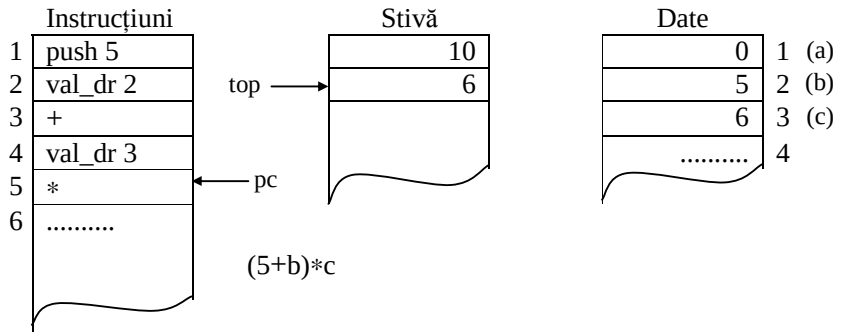
Intrarea cu indicele 0 a fost lăsată goală deoarece funcția *caută* returnează *zero* în cazul când nu găseşte identificatorul în TS.

În continuare, se prezintă pseudocodul corespunzător unui analizor lexical care tratează identificatorii în această manieră:

```
function anlex : integer;
   var tamplex : array [0 ... 100] of char;
      c : char; p : integer;
   begin
      loop begin
         *citeşte un caracter în c;
         if *c este spațiu sau tab then {nimic}
         else if *c este newline then nrline := nrline+1
            else if *c este o cifră then begin
                  *construieşte numărul în valatom;
                 return NUM
               end else if *c este o literă then begin
                        *introdu c împreună cu literele şi cifrele
                        următoare în tamplex;
                        p := caută (tamplex);
                        if p=0 then p := inserează (tamplex, ID);
                        valatom := p;
                        return tabsimb[p].atom
                     end else {atomul este un singur caracter} begin
                           valatom := NIMIC; {nu există atribut}
                           return ord(c)
                        end
      end {loop}
   end; {anlex}
```

Maşini stivă abstracte

Partea *front-end* a unui compilator construieşte o reprezentare intermediară a programului sursă din care, prin *back-end*, se generează programul destinaţie. În mod frecvent, reprezentarea intermediară este un cod abstract pentru o maşină virtuală care lucrează pe principiul stivei. Maşina are memorii separate pentru instrucţiuni (cod) şi respectiv pentru date iar valorile operanzilor şi operaţiilor aritmetice sunt întotdeauna în vârful stivei.



Starea maşinii-stivă după executarea primelor patru instrucţiuni

Pointerul *pc* indică instrucţiunea ce urmează să fie executată. Se consideră o listă redusă de instrucţiuni abstracte împărţite în trei clase:

- operaţii aritmetice între valori întregi;
- instrucţiuni pentru manipularea stivei;
- instrucţiuni pentru controlul fluxului programului.

Instrucţiuni aritmetice

Se presupune că toţi operatorii aritmetici acceptaţi sunt instrucţiuni ale maşinii abstracte. În aceste condiţii, codul virtual pentru o expresie aritmetică simulează evaluarea reprezentării *postfix* pentru acea expresie. Reprezentarea postfix este parcursă de la stânga la dreapta. La întâlnirea unui operand, valoarea lui se introduce în stivă iar la întâlnirea unui operator cu aritate *k*, cele *k* argumente ale sale sunt în vârful stivei, cu argumentul cel mai din dreapta situat chiar în vârf. Evaluarea aplică operatorul celor *k* argumente şi introduce rezultatul tot în stivă după ce, în prealabil, a extras operanzii. În final, în vârful stivei rămâne valoarea întregii expresii.

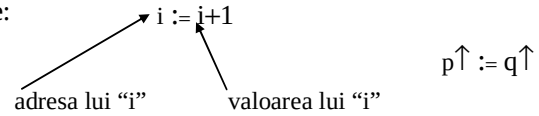
Pe lângă operaţiile aritmetice între valori întregi pot fi tratate în aceeaşi manieră şi operaţiile logice dacă se adoptă convenţia ca zero să-i corespundă lui *false* şi diferit de zero lui *true*. Trebuie remarcat că vor fi evaluaţi, indiferent de situaţie, ambii operanzi ai operatorilor **or** şi **and**.

Instrucţiuni pentru manipularea stivei

Semnificaţia diferită a unei variabile atunci când se află în stânga sau în dreapta operatorului de atribuire este exprimată prin termenii:

- *valoare-stânga*: reprezentând adresa locaţiei de memorie a variabilei;
- *valoare-dreapta*: reprezentând valoarea efectivă a variabilei (conţinutul locaţiei de memorie).

Exemple:



Rezultă de aici următoarele două coduri virtuale relative la variabile:

val_dr	l	– introdu în stivă conţinutul (valoarea) din locaţia *l*
val_st	l	– introdu în stivă adresa locaţiei *l*

La acestea se adaugă instrucţiunile virtuale pentru operaţiile obişnuite cu stiva:

push	v	– introdu în stivă valoarea *v*
pop		– extrage valoarea din vârful stivei

precum şi:

:=	*valoarea_dreaptă* din vârful stivei este plasată la adresa reprezentată de *valoarea_stângă* de sub ea, după care, amândouă valorile se extrag din stivă;
copy	introdu în stivă o copie a valorii din vârf.

Codurile abstracte definite până acum sunt suficiente pentru a traduce expresii pe baza reprezentării lor postfix, inclusiv instrucţiuni de atribuire.

Exemplu: a+b → val_dr a
val_dr b
+

Traducerea atribuirilor se face astfel:

- se introduce în stivă *valoarea_stângă* a identificatorului atribuit;
- se evaluează expresia;
- *valoarea_dreaptă* a expresiei (din vârful stivei) se atribuie identificatorului.

Exemplu: x := (a+4)div 5–b, se traduce prin:

val_st	x	push	5
val_dr	a	div	
push	4	val_dr	b
+		–	
		:=	

Formal, traducerea efectuată se poate exprima astfel: fiecărui neterminal i se asociază un atribut sintetizat *t* care primeşte ca valoare chiar codul obţinut prin traducerea acelui neterminal. Acest atribut este evaluat printr-o acţiune semantică ataşată producţiei.

Exemplu: <instr>→**id** := <expr>{<instr>**.**t := 'val_st' ‖ **id.**lexema ‖ <expr>**.**t ‖ ':='}

Instrucțiuni pentru controlul fluxului programului

Maşina abstractă execută instrucțiunile secvențial, fluxul secvențial putând fi întrerupt doar prin instrucțiuni de salt condiționat sau necondiționat. Acestea au următoarea formă generală:

<comandă> <destinație>

În ceea ce priveşte exprimarea destinației pot exista mai multe variante:

1) Se indică locația destinației (adresa ei);
2) Se specifică distanța relativă, pozitivă sau negativă a destinației, față de punctul respectiv din codul virtual;
3) Destinația este specificată simbolic, prin etichete.

Primele două variante pot fi completate cu posibilitatea suplimentară de a lua operandul (valoarea destinației) din vârful stivei. A treia variantă se poate aplica doar dacă maşina abstractă admite etichete şi are următoarele avantaje:

– simplifică generarea codului pentru salturi;
– permite modificări (inserări sau ştergeri de instrucțuini) în codul generat, de exemplu prin optimizări ulterioare.

Dacă se adoptă această a treia variantă, instrucțiunile virtuale pentru controlul fluxului sunt:

label	l	defineşte eticheta *l* în acel punct din programul virtual;
goto	l	salt necondiționat la linia de cod cu eticheta *l*;
gofalse	l	extrage valoarea din vârful stivei şi salt la eticheta *l* dacă ea este zero (false);
gotrue	l	extrage valoarea din vârful stivei şi salt la eticheta *l* dacă ea este diferită de zero (true);
halt		opreşte execuția programului virtual.

Cu aceste instrucțiuni virtuale se poate exprima în cod pentru maşina abstractă orice instrucțiune selectivă sau ciclică.

Exemple:

if <expr> **then** <instr>$_1$
se traduce prin: cod pentru <expr>
gofalse afară
cod pentru <instr>$_1$
label afară

while <expr> **do** <instr>$_1$
se traduce prin: label test
cod pentru <expr>
gofalse afară
cod pentru <instr>$_1$
goto test
label afară

În ceea ce priveşte crearea etichetelor, pot să apară confuzii dacă ele nu sunt unice. În acest scop trebuie prevăzut un mecanism care să furnizeze o etichetă unică de fiecare dată când este nevoie (de exemplu, o funcție notată *etichetă_nouă* care returnează câte o etichetă distinctă, de fiecare dată când este apelată). Cu aceste precizări, producția care generează instrucțiunea **if** şi acțiunile semantice asociate, sunt:

<instr>→**if** <expr> **then** <instr>$_1$ {afară := etichetă_nouă; <instr>**.**t := <expr>**.**t ‖ ’gofalse’ afară ‖ <instr>$_1$**.**t ‖ ’label’ afară}

Generarea codului virtual

În translatorul de expresii de la începutul capitolului, la generarea efectivă a traducerii s-a utilizat instrucțiunea *print*. În mod similar pot fi traduse şi instrucțiuni. Într-un caz mai general se poate prevedea o procedură, *emite*, care introduce codul generat într-un fişier de ieşire ce reprezintă intrarea maşinii abstracte. Luând în considerare o astfel de procedură, ST pentru instrucțiunile atribuire şi **if**, devine:

<instr> → **id** {emite (‘val_st’, **id.**lexema)}
:=
<expr> {emite (‘:=’)}

<instr> → **if**
<expr> {afară := etichetă_nouă; emite (‘gofalse’,afară)}
then
<instr>$_1$ {emite (‘label’,afară)}

Pseudocodul corespunzător traducerii celor două instrucțiuni este:

```
procedure instr;
   var afară : integer;
   begin
      if  simb=id then begin
          emite(‘val_st’,valatom); coresp(id);
          coresp(‘:=’); expr; emite(‘:=’) end
      else if simb=if then begin
              coresp(if); expr; afară := etichetă_nouă;
              emite(‘gofalse’,afară);
              coresp(‘then’); instr;
              emite(‘label’,afară);
              end else eroare;
end;
```