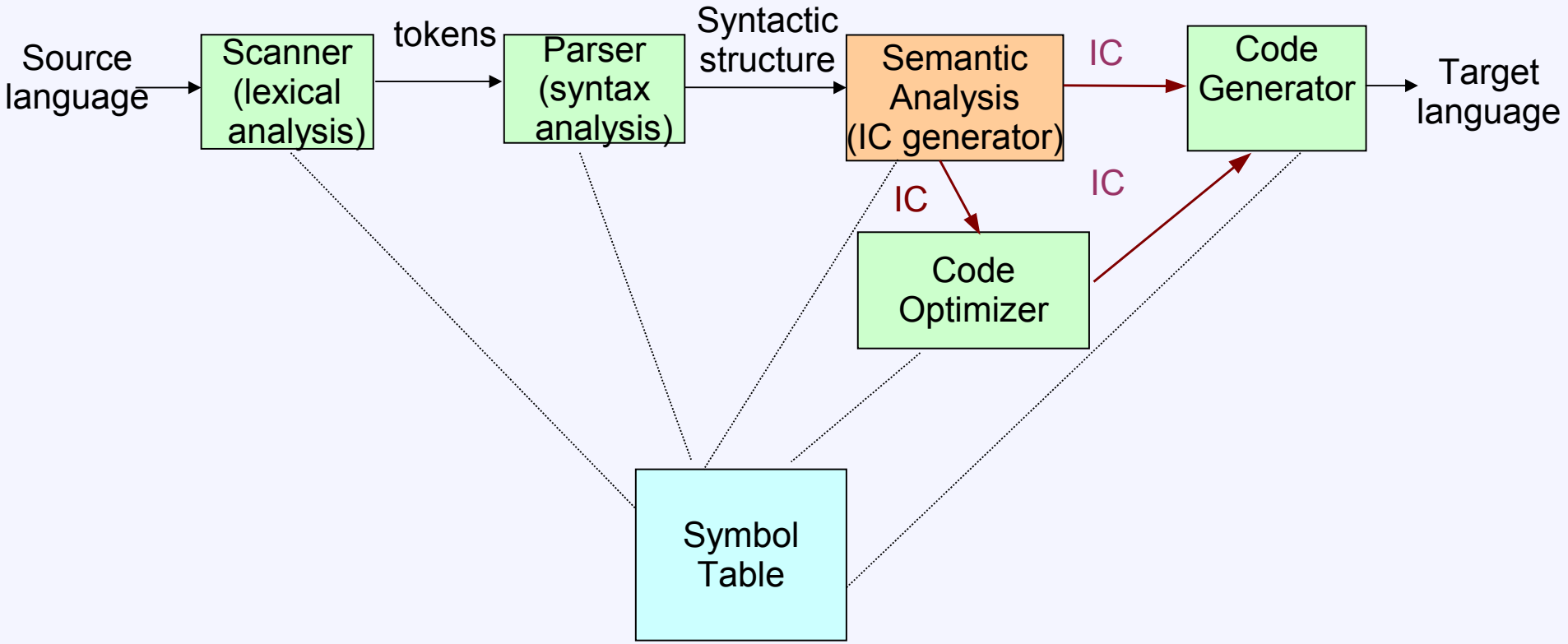
Generovanie do medzijazyka

Ján Šturc

Zima 2010

- Formy medzijazyka
- Generovanie výrazov
- Booleovské výrazy
- Príkazy
- Volania
- Spätné plátanie

Použitie medzijazyka v kompilátore



Poznámka:
Projekt uncol .

Formy medzijazyka

- Polská sufixová forma
  - Vynimočne len pre malé zariadenia. Neumožňuje prakticky žiadnu optimalizáciu. Pre bezprostredné vyhodnotenie na zásobníkovom automate (napr. kalkulačky).
- **Štvorice**
  - <operácia><1.operand><2.operand><výsledok>
  - Vlastne formát trojadresových inštrukcií
  - V súčasnosti najpoužívanejší tvar medzijazyka
- Trojice
  - Šetria 30% pamäte, neumožňujú však optimalizácie vyplývajúce z čiastočného usporiadania kódu. Na adresáciu výsledku sa adresuje trojica.
- Nepriame trojice
  - Kompromis. Tabuľka pre nepriamu adresáciu trojíc umožňuje ich preusporiadanie.

Inštrukcie trojadresového stroja

- Program je postupnosť záznamov tvaru:

  struct { oper: operation;
  address: operand1, operand2, result}

- Operácie:
  - aritmetické, logické
    - +,-,$_u$-,×,/, (real, integer, fixed), div, mod
    - ¬,∧,∨, ⊕, <<, >>, (všeliké shifty)
  - move (priradenie, zapamätanie, skoky)
    - **if** op1 (<,>,=,≠) 0 **then goto** op2;
    - Ak vieme adresovať všetky registre, skok je move to PC (program counter).
  - indirekcia a referencia (A[i], &a)
    - result:= ∗(∗op1+op2)
    - result:= &op2
- Nebudem v ďalšom príliš dodržovať formalizmus. Zápis v štýle publikácie programov.

„Dračí“ medzijazyk

1.	x := y *op* z	assignment
2.	x := *op* y	unary assignment
3.	x := y	copy
4.	goto L	unconditional jump
5.	if (x *relop* y) goto L	conditional jump
6.	param x	procedure call
7.	call p n	procedure call
8.	return y	procedure call
9.	x := y[i]	indexed assignment
10.	x[i] := y	indexed assignment

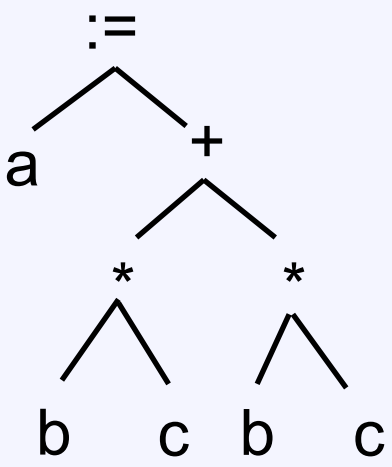
Projekt – interpretácia medzijazyka

- Príkazy medzijazyka, sú jednoduché príkazy, koré sa vyskytujú v nejakej podobe v každom imperatívnom jazyku (C, Pascal, Algol, …).
- Po dodaní „syntaktického cukru“ ich možno preložiť ľubovolným z uvedených jazykov.
- Volanie procedúr:
  - Ak vyšší jazyk ma rekurziu, je priamočiaré.
  - Programovací jazyk nemá rekurziu. Treba prekladať do volacích záznamov na zásobníku (viď „Podpora počas behu“).
- Adresácia:
  - Alokácia pamäte pri kompilácii, robíme alokáciu počas spracovávania deklarácii.
  - Ponechať symbolické mená a ponechať alokáciu pamäte na programovací jazyk (nevhodné, ak chceme skúsiť nejaké optimalizácie).

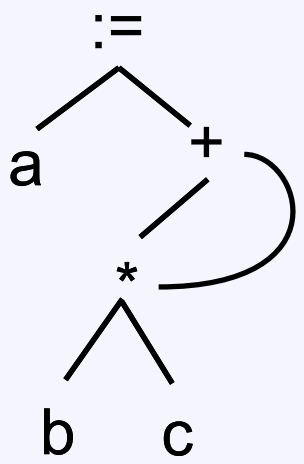
Rôzne reprezentácie

výraz: a := b * c + b * c
postfix: abc*bc*+ :=

strom výrazu:



dag výrazu:



trojadresový kód:

$t_1 := b * c$
$t_2 := b * c$
$t_3 := t_1 + t_2$
$a := t_3$

$t_1 := b * c$
~~$t_2 := t_1$~~
$t_3 := t_1 + t_2$ (t_2 → t_1)
$a := t_3$

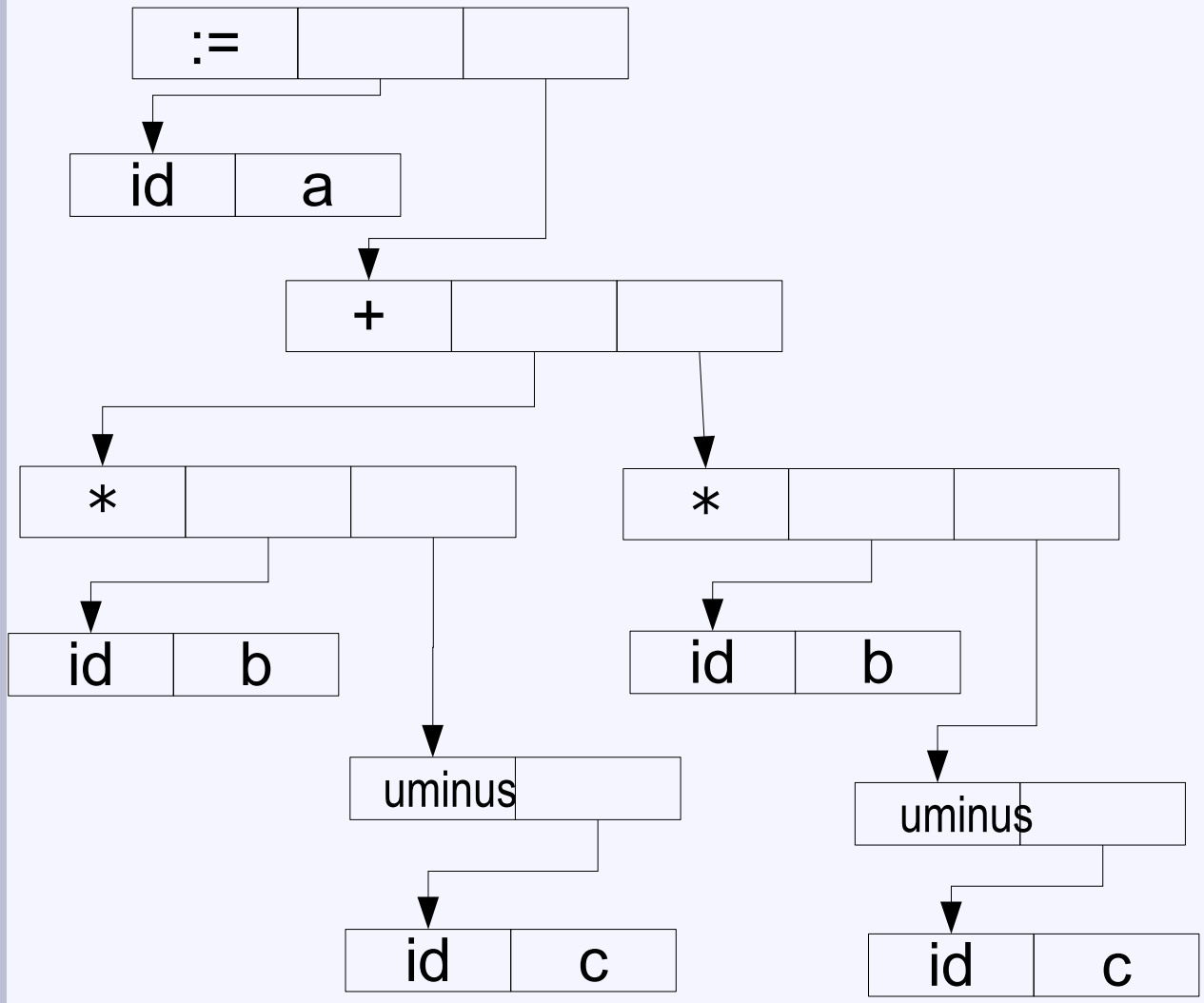
Generovanie stromov a dagov

- Grafická reprezentácia (binárne stromy), ktoré obsahujú:
  - binárne vnútorné uzly – nodes (operácia a dva smerníky na operandy)
  - unárne vnútorné uzly – unodes (operácia a smernik na operand)
  - listy – leaf obsahujú identifikátory a ich hodnoty
- Sémantické procedúry
  - mknode(op:operation, left, right: pointer):pointer
  - mkunode(op:operation, child: pointer):pointer
  - mkleaf("id", id.place:pointer):pointer
- Lineárna reprezentácia sú trojice (štvorice)
  - unode má druhý smerník prázdny
  - leaf má v prvej položke „identifikátor“
- Lineárnu reprezentáciu vytvoríme očíslovaním trojíc (štvoríc) pomocou post-order traverzovania.

Syntaxou riadený preklad

Syntax	Sémantika
$S \rightarrow$ **id** := E	S.nptr:= mknode(":=", mkleaf(**id**, **id**.place), E.nptr)
$E \rightarrow E_1 + E_2$	E.nptr:= mknode("+", E_1.nptr, E_2.nptr)
$E \rightarrow E_1 * E_2$	E.nptr:= mknode("$*$", E_1.nptr, E_2.nptr)
$E \rightarrow - E_1$	E.nptr:= mkunode("uminus", E_1.nptr)
$E \rightarrow (E_1)$	E.nptr:= E_1.nptr
$E \rightarrow$ **id**	E.nptr:= mkleaf(**id**, id.place)

Strom a:= b*(-c) + b*(-c)



0	id	b	
1	id	c	
2	u-	1	
3	*	0	2
4	id	b	
5	id	c	
6	u-	5	
7	*	4	6
8	+	3	7
9	id	a	
10	:=	9	8

Generovanie syntaxou riadeným prekladom

- Na generovanie trojadresového kódu použijeme syntaxou riadený preklad.
- Začneme jazykom pozostávajúcim z priradenia a výrazov.
- Príkaz priradenia S má jediný atribút **code** - vygenerovaný kód.
- Výrazy E majú dva atribúty:
  - **code** – úsek kódu zodpovedajúci výrazu
  - **place** – meno premennej, v ktorej je uložená hodnota výrazu E.
- Označenie: *gen(x ":=" y "+" z)* reprezentuje príkaz x:= y + z
- Označenie *<fragment>* || *expr* znamená zreťazenie fragmentu kódu s výrazom.

Syntaxou riadený preklad

S → **id** := E { S.code := E.code || gen(**id**.place ":=" E.place) }

E → E_1 + E_2 { E.place := newtemp;
E.code := E_1.code || E_2.code ||
gen(E.place ":=" E_1.place "+" E_2.place)}

E → E_1 * E_2 { E.place := newtemp();
E.code := E_1.code || E_2.code ||
gen(E.place ":=" E_1.place "*" E_2.place) }

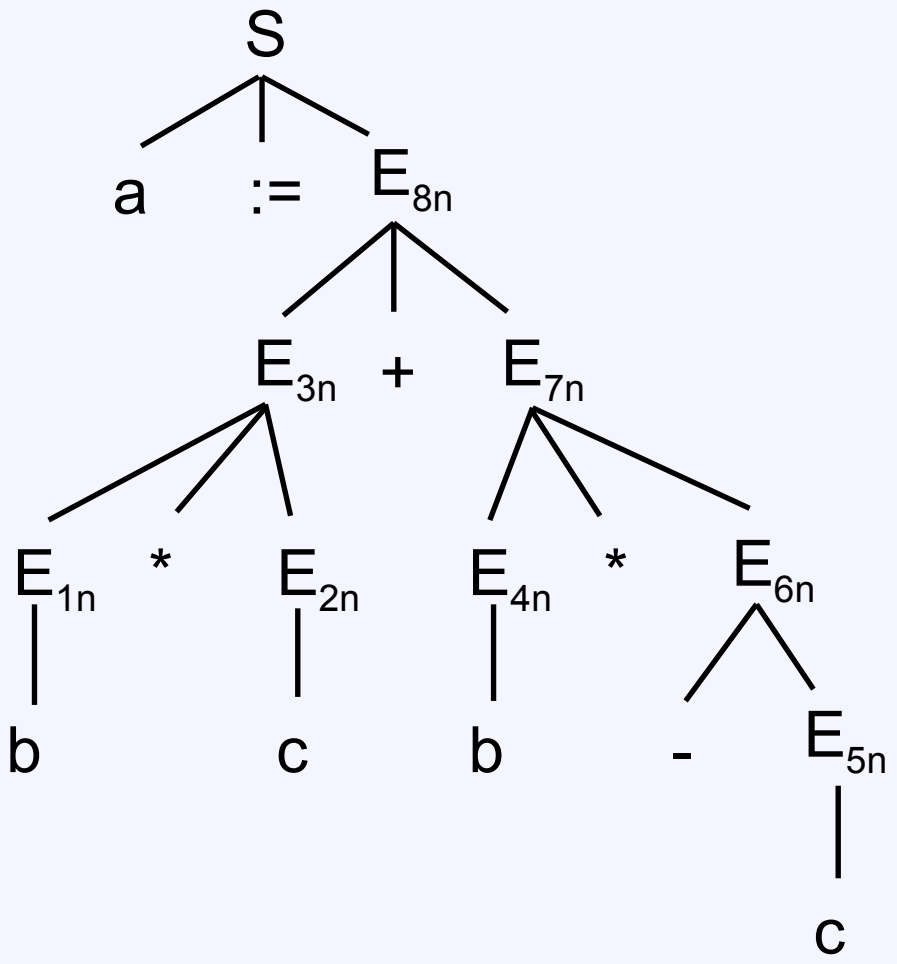
E → - E_1 { E.place := newtemp();
E.code := E_1.code ||
gen(E.place ":=" "uminus" E_1.place) }

E → (E_1) { E.place := newtemp();
E.code := E_1.code }

E → **id** { E.place = **id**.place;
E.code := "" }

Príklad – syntaktický strom

a:= b ∗ c + b ∗ -c

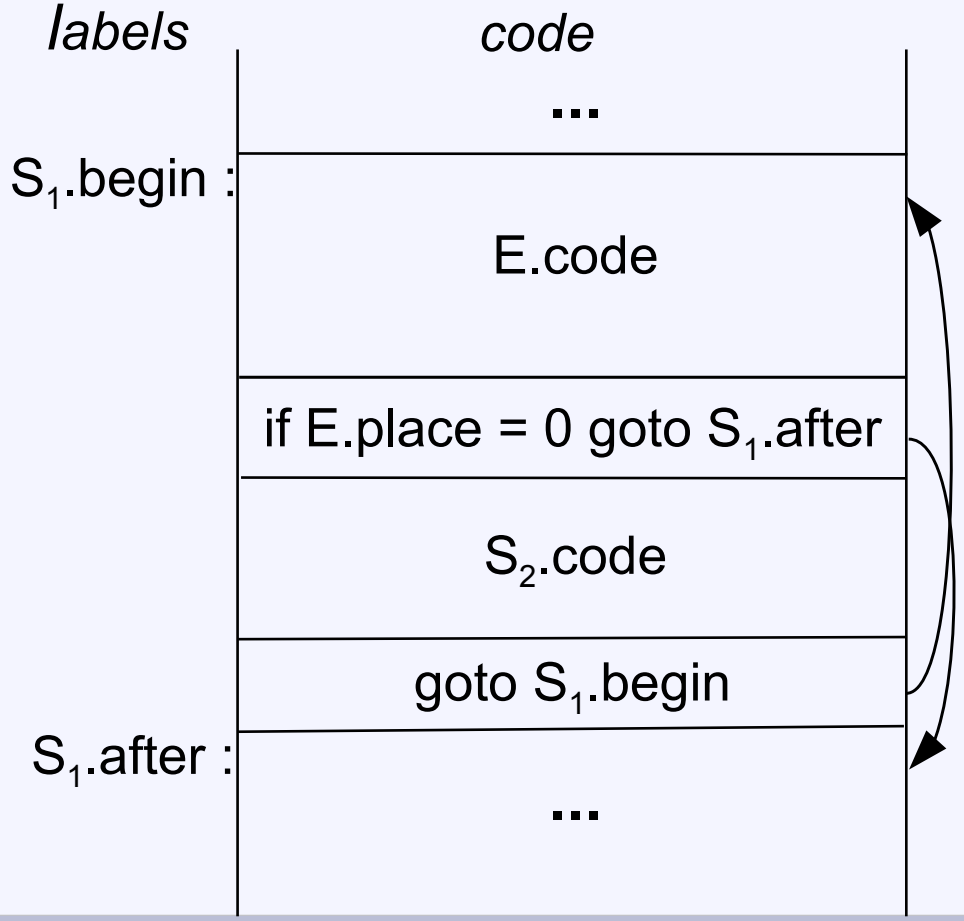


Príklad – generovanie kódu

	place	code
E_{1n}	b	
E_{2n}	c	
E_{3n}	t_1	E_{1n}.code || E_{2n}.code || t_1 := b * c
E_{4n}	b	
E_{5n}	c	E_{5n}.code || t_2 := $_u$- c
E_{6n}	t_2	E_{4n}.code || E_{6n}.code || t_3 := b * t_2
E_{7n}	t_3	E_{3n}.code || E_{7n}.code || t_4 := t_1 + t_3
E_{8n}	t_4	E_{8n}.code || a := t_4
S		

Tok riadenia

$S_1 \rightarrow$ **while** E **do** S_2 S_1.begin := newlabel;
S_1.after := newlabel;
S_1.code := gen(S_1.begin ":") || E.code ||
gen("if" E.place "= 0 goto" S_1.after)
|| S_2.code || gen("goto" S_1.begin) ||
gen(S_1.after ":")



Deklarácie

$P \rightarrow MD;B.$	
$M \rightarrow \varepsilon$	{ offset:=0 }
$D \rightarrow D;D$	
$D \rightarrow$ id:T	{ enter(id.name, T.type,ofset); offset:= offset + T.width }
$T \rightarrow$ **integer**	{ T.type:= integer; T.width:= 4 }
$T \rightarrow$ **real**	{ T.type:= real; T.width:= 8 }
$T \rightarrow$ **array**[**num**] **of** T_1	{ T.type:= array(**num**.val, T_1.type); T.width:= **num**.val $\times$ T_1.width }
$T \rightarrow \uparrow T_1$	{ T.type:= pointer(T_1.type); T.width:= 4 }

Pozn. Práca s tabuľkou symbolov je trochu komplikovanejšia. Treba zohľadniť „scope“ deklarácií.

„Dračie“ sémantické procedúry

1. mktable(previous: pointer): pointer
   Vytvára novú tabuľku symbolov. Argumentom je smerník na nadradenú tabuľku a vracia smernik na novo vytvorenú tabuľku.
2. enter(table:pointer, name:string, type:type, offset:int)
   Vytvára nový záznam v tabuľke symbolov pre identifikátor name.
3. addwidth(table:table, width:integer)
   Zaznamená kumulatívnu dĺžku všetkých položiek v hlavičke tabuľky symbolov.
4. enterproc(table:pointer, name:string, newtable:pointer)
   Vytvára novú položku pre vnorenú procedúru v danej tabuľke. Newtable ukazuje na novovytvorenú tabuľku symbolov pre túto procedúru.

Použitie

```
P  → M D                          { addwidth(top(tblptr), top(offset));
                                    pop(tblptr); pop(offset)                    }
M  →  ε                           { t:= mktable(nil);
                                    push(t, tblptr); push(0, offset)            }
D  →  D1 ; D2
D  →  proc id ; N D1; S    { t:= top(tblptr); addwidth(t, top(offset);
                                    pop(tblptr); pop(offset);
                                    enterproc(top(tblptr), id.name, t)          }
D  → id : T                { enter(top(tblptr), id.name, T.type, top(offset));
                                     top(offset):= top(offset) + T.width        }
N  →  ε                           { t:= mktable(top(tblptr));
                                    push(t, tbptr); push(0, offset)             }
T  → record L D end        { T.type:= record( top(tblptr));
                                     T.width:= top(offset);
                                     pop(tblptr); pop(offset);                  }
L  →  ε                           { t:= mktable(nil);
                                    push(t, tblptr); push(0, offset)            }
```

Príkaz priradenia

$S \rightarrow$ **id** := E { p:= lookup(**id**.name);
if p ≠ nil **then** gen(p ":=" E.place) **else** error }

$E \rightarrow E_1 + E_2$ { E.place := newtemp;
gen(E.place ":=" E_1.place "+" E_2.place) }

$E \rightarrow E_2 * E_3$ { E.place := newtemp;
gen(E.place ":=" E_1.place "∗" E_2.place) }

$E \rightarrow - E_1$ { E.place := newtemp;
gen(E_1.place ":= uminus" E_1.place }

$E \rightarrow (E_1)$ { E.place:= E_1.place }

$E \rightarrow$ **id** { p:= lookup(id.name);
if p ≠ nil **then** E.place := p **else** error }

Polia (arrays)

Polia ukladáme súvisle prvok po prvku: A[low:high] nech w je veľkosť jedneho prvku. Potom adresa i-tého prvku je:

$$base + (i - low)\times w = base - low\times w + i\times w = A[0] + i\times w$$

Zovšeobecnenie pre mnohorozmerné polia. Nech $n_j = high_j - low_j$ pre $j \le k$. Adresa prvku $A[i_1, i_2, \ldots, i_k]$ je:

$$A[0, 0, \ldots 0] + ((\ldots(i_1\times n_2+i_2)\times n_3+i_3)\ldots)\times n_k+i_k)\times w,$$

kde

$$A[0, 0, \ldots 0] = base - ((\ldots(low_1\times n_2+low_2)\times n_3+low_3)\ldots)\times n_k+low_k)\times w.$$

Základ base je offset prvé voľné miesto na zásobníku. Kvôli tomu musíme fiktívnu adresu nultého prvku počítať. Ak dovolíme len statické polia, stačí počas kompilácie. Označíme ju c(A).

Syntaxou riadený preklad

S → L := E { **if** L.offset = null **then** gen(L.place ":=" E.place)
else gen(L.place "[" L.offset "] :=" E.place) }

E → L { **if** L.offset = null **then** E.place = L.place
else E.place := newtemp;
gen(E.place ":=" L.place "[" L.offset "]") }

L → Elist] { L.place := newtemp; L.offset := newtemp;
gen(L.place ":=" c(Elist.array));
gen(L.offset ":=" Elist.place "*" width(Elist.array))}

L → id { L.place := **id**.place; L.offset := null }

Elist → $Elist_1$, E { t := newtemp; m := $Elist_1$.ndim + 1;
gen(t ":=" $Elist_1$.place "∗" limit($Elist_1$.array, m));
gen(t ":=" t "+" E.place); Elist.array := $Elist_1$.array;
Elist.place := t; Elist.ndim := m }

Elist → id [E { Elist.array := **id**.place;
Elist.place := E.place; Elist.ndim := 1 }

Vynutená konverzia (coercion)

Relatívne úplná semantika pravidla $E \rightarrow E_1 + E_2$

E.place:= newtemp; /* Radšej do každej vetvy ako druhý príkaz. */
if E_1.type = integer **and** E_2.type = integer **then**
    { E.type := integer; gen(E.place ":=" E_1.place "int +" E_2.place) }
else if E_1.type = real **and** E_2.type = real **then**
    { E.type := real; gen(E.place ":=" E_1.place "real +" E_2.place) }
else if E_1.type = integer **and** E_2.type = real **then**
    { E.type := real; u:= newtemp; gen(u ":=" "intoreal" E_1.place);
        gen(E.place ":=" u "real +" E_2.place) }
else if E_1.type = real **and** E_2.type = integer **then**
    { E.type := real; u:= newtemp; gen(u ":=" "intoreal" E_2.place);
      gen(E.place ":=" E_1.place "real +" u) }
else E.type := type_error;

Tok riadenia

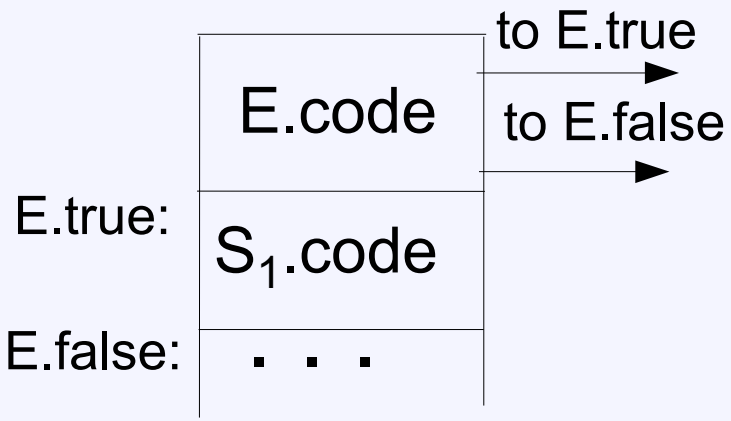
$S \rightarrow$ **if** E **then** S_1 { E.true:= newlabel; E.false:= S.next;
S_1.next:= S.next;
S.code:= E.code || gen(E.true ":") ||
S_1.code }

$S \rightarrow$ **if** E **then** S_1 **else** S_2 { E.true:= newlabel; E.false:= newlabel;
S_1.next:= S.next; S_1.next:= S.next;
S.code:= E.code || gen(E.true ":") ||
S_1.code || gen("go to" S.next)||
gen(E.false ":") || S_2.code }

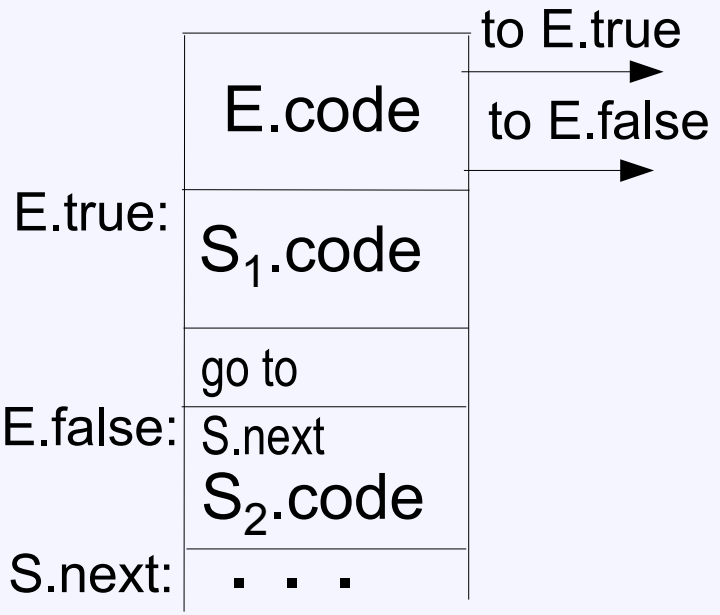
$S \rightarrow$ **while** E **do** S_1 { S.begin:= newlabel; E.true:= newlabel;
E.false:= S.next; S_1.next:= S.next;
S.code:= gen(S.begin ":") || E.code ||
gen(E.true ":") || S_1.code ||
gen("go to" S.begin) }

Grafické znázornenie

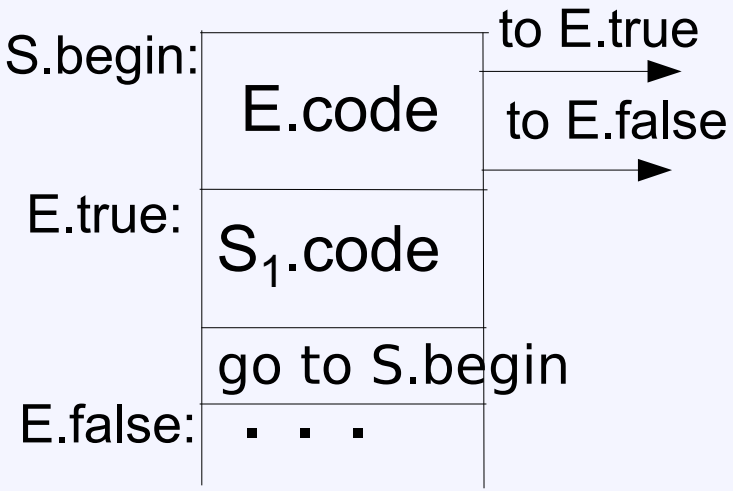
if then príkaz



if then else príkaz

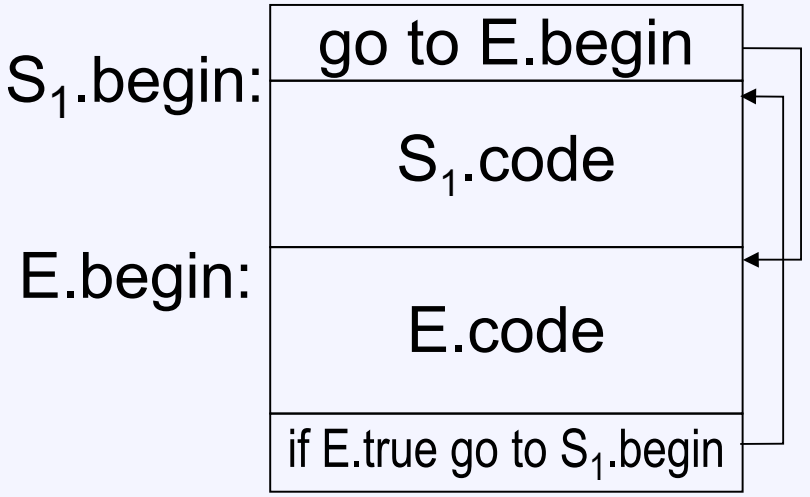


while príkaz



Iná šablóna pre while príkaz

$S \rightarrow$ **while** E **do** S_1 { S_1.begin:= newlabel; E.begin:= newlabel;
S.code:= gen("go to" S_1.begin) ||
S_1.code || E.code ||
gen("if E.true go to" S_1.begin) }



Pokiaľ boolovské výrazy počítame aritmetickým spôsobom, je táto šablóna vyhodnejšia.

Booleovské výrazy 1 – klasika

$E \rightarrow E_1$ **or** E_2 { E.place := newtemp;
gen(E.place ":=" E_1.place "or" E_2.place) }

$E \rightarrow E_1$ **and** E_2 { E.place := newtemp;
gen(E.place ":=" E_1.place "and" E_2.place) }

$E \rightarrow$ **not** E_1 { E.place := newtemp;
gen(E.place ":=" "not" E_1.place) }

$E \rightarrow (E_1)$ { E.place := E_1.place }

$E \rightarrow$ **true** { E.place := newtemp; gen(E.place ":=" "1") }

$E \rightarrow$ **false** { E.place := newtemp; gen(E.place ":=" "0") }

$E \rightarrow aE_1$ **relop** aE_2 { E.place := newtemp;
gen("if" aE_1.place **relop** aE_2.place "go to" +3);
gen(E.place ":=" "0"); gen("go to" +2);
gen(E.place ":=" "1") }

Boolovské výrazy 2 – skratkou

$E \rightarrow E_1$ **or** E_2 { E_1.true:= E.true; E_1.false:= newlabel;
E_2.true:= E.true;E_2.false:= E.false;
E.code:= E_1.code || gen(E_1.false ":")|| E_2.code }

$E \rightarrow E_1$ **and** E_2 { E_1.true:= newlabel; E_1.false:= E.false;
E_2.true:= E.true; E_2.false:= E.false;
E.code:= E_1.code || gen(E_1.true ":") || E_2.code }

$E \rightarrow$ **not** E_1 { E_1.true:= E.false; E_1.false:= E.true;
E.code:= E_1.code }

$E \rightarrow (E_1)$ { E_1.true:= E.true; E_1.false:= E.false;
E.code:= E_1.code }

$E \rightarrow$ **true** { gen("go to" E.true) }

$E \rightarrow$ **false** { gen("go to" E.false) }

$E \rightarrow aE_1$ **relop** aE_2 { E.code:= gen ("if" aE_1.place **relop** aE_2.place
"go to" E.true) || gen("go to" E.false) }

Príkaz case – šablóny

case E **of**
V_1: S_1
V_2: S_2
...
V_{n-1}: S_{n-1}
default: S_n
end

```
        t:= E
        go to test
L1:     S1.code
        go to next
        ...
Ln-1:   Sn-1.code
        go to next
Ln:     Sn.code
        go to next
test:   if t = V1 go to L1
        ...
        if t = Vn go to Ln
next:
```

```
        t:= E
        if t ≠ V1 go to L1
        S1.code
        go to next
L1:     if t ≠ V2 go to L2
        S2.code
        go to next
L2:
        ...
Ln-2:   if t ≠ Vn-1 go to Ln-1
        Sn-1.code
        go to next
Ln-1:   Sn.code
next:
```

Volanie procedúr

S → **call** **id** Alist { sgen(**id**.acr); count::= 0; save("returnaddress") }
Alist → (E Elist { save("param" E.place); count:=pcount + 1 }
Alist → ε { gen("call" count **id**.ref) }
Elist → ,E Elist { save("param" E.place); count:=pcount + 1 }
Elist →) { gen("call" count **id**.ref) }

Spätné plátanie – backpatching

- Najjednoduchší spôsob implementácie dva prechody
  - Vygenerovať (anotated) syntaktický strom
  - Generovať kód pomocou depth first traversingu tohto stromu
- Pri jednoprechodovom generovaní sú problémom dopredné skoky toku riadenia. Najjednoduchšie je nechať ciele skoku prázdne, poznamenať si ich do nejakého zoznamu a doplniť až keď príslušná adresa je vygenerovaná. Používame príkazy:
  - makelist(i:index_to_quad):pointer_to_list Vytvorí zoznam obsahujúci index i – poradové číslo štvorice a vráti smerník na tento zoznam.
  - merge(p_1, p_2:pointer_to_list):pointer_to_list Spojí dva zoznamy a vráti smerník na výsledný zoznam.
  - backpatch(p:pointer_to_list, a:index_to_quad) Dosadí adresu a do všetkých štvoríc v zozname, na ktorý ukazuje p.

Príklad 1 – boolovské výrazy

$E \rightarrow E_1$ **or** $M\ E_2$ { backpatch(E_1.falselist, Mquad);
E.truelist:= merge(E_1.truelist, E_2.truelist);
E.falselist:= E_2.falselist }

$E \rightarrow E_1$ **and** $M\ E_2$ { backpatch(E_1.truelist, Mquad);
E.truelist:= E_2.truelist;
E.falselist:= merge(E_1.falselist, E_2.falselist) }

$E \rightarrow$ **not** E_1 { E.truelist:= E_1.truelist; E.falselist:= E_1.falselist }

$E \rightarrow (E_1)$ { E.truelist:= E_1.truelist; E.falselist:= E_1.falselist }

$E \rightarrow$ **true** { E.truelist:= makelist(nextquad); gen("go to" _)}

$E \rightarrow$ **false** { E.falselist:= makelist(nextquad);gen("go to" _)}

$E \rightarrow aE_1$ **relop** aE_2 { E.truelist:= makelist(nextquad);
E.falselist:= makelist(nextquad+1);
gen("if" aE_1.place **relop** aE_2.place "go to" _);
gen("go to" _) }

$M \rightarrow \varepsilon$ {M.quad:= nextquad }

Príklad 2 – tok riadenia

$S \rightarrow$ **if** E **then** M_1 S_1 N **else** M_2 S_2
{ backpatch(E.truelist, M_1.quad);
backpatch(E.falselist, M_2.quad);
S.nextlist:= merge(S_1.nextlist, merge(N.nextlist, S_2.nextlist) }

$N \rightarrow \varepsilon$ { N.nextlist:= makelist(nextguad); gen("go to" _) }

$M \rightarrow \varepsilon$ { M.quad:= nextquad }

$S \rightarrow$ **if** E **then** M S_1 { backpatch(E.truelist, M.quad);
S.nextlist:= merge(E.falselist, S_1.nextlist) }

$S \rightarrow$ **while** M_1 E **do** M_2 S_1 { backpatch(S_1.nextlist, M_1.quad);
backpatch(E.truelist, M_2.quad);
S.nextlist:= E.falselist;
gen("go to" M_1.quad) }

Príklad 2 – dokončenie

$S \rightarrow$ **begin** L **end**	{ S.nextlist:= L.nextlist }
$S \rightarrow A$	{ S.nextlist:= **nill** }
$L \rightarrow L_1$ **;** M S	{ backpatch(L_1.nextlist, M.quad); L.nextlist:= S.nextlist }
$L \rightarrow S$	{ L.nextlist:= S.nextlist }