Překladače

Jiří Dobeš, s využitím verze Luďka Bártka

20. dubna 1995

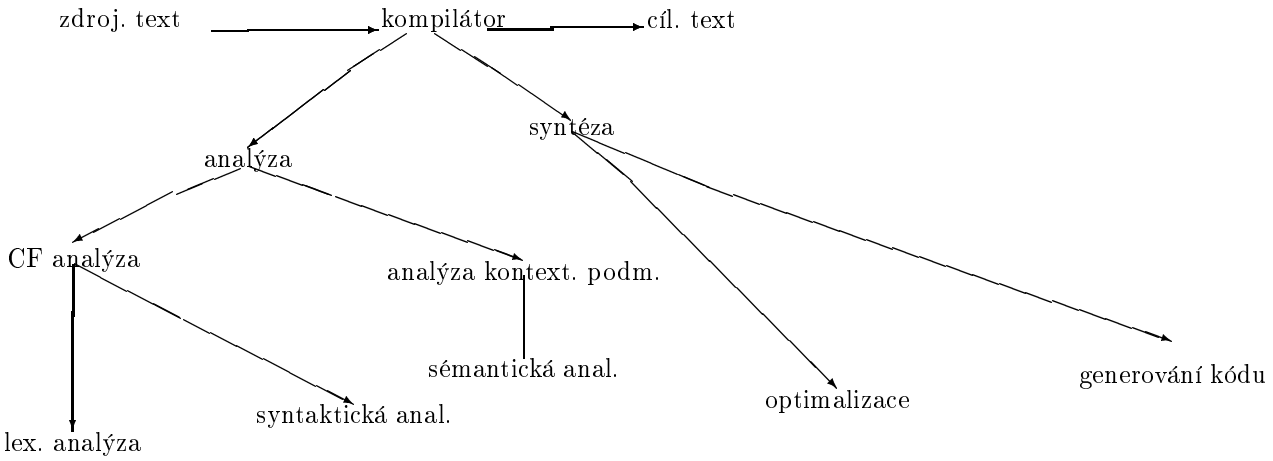
Obsah

1 Úvod

prog. jazyk:

- syntaxe – CFG + kontextové podmínky
- sémantika – formální sémantiky:
  - axiomatické (statická sém.)
  - operační(abstrkt. stroj)
  - denotační(dynamická sémantika)
- pragmatika (na co je určen,..)

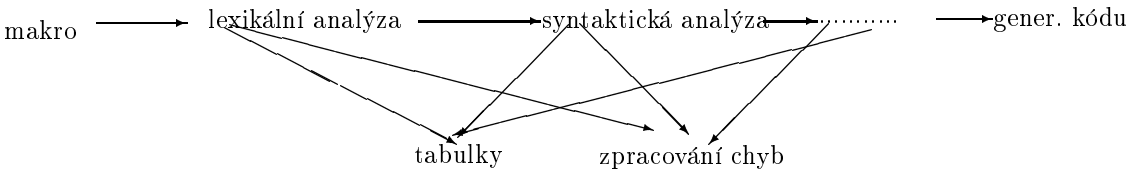
statický - vztahuje se k době kompilace
dynamický - běh programu
text zdroj → kompilátor → cíl. text
kompilátor je robustní program
na vstupu je vstupní podmínka $\varphi(x)$
na výstupu dá také zprávu o chybách
robustní programy - reagují na data, která nesplňují vstupní podmínku.



Obrázek 1: Celková činnost kompilátoru

2 Lexikální analýza

⇒ rozdělit analýzu syntaxe – LA
– SA
⇒ zdroj. text → LA →



Obrázek 2: front - end kompilátoru

zvyšuje celkovou efektivitu (regulární části)
regulární fragment - lexikální položka
Příklad 2.1
Lexikální položky: identifikátor, konstanta, klíčová slova, operátory.
Obecně pod lexikální položkou rozumíme dvojici
(syntaktická kategorie, data)
data je buď hodnota nebo pointer v závislosti na kategorii
(IDSYM,↑ tab_symbolů) identifikátor
(IDSYM, 3)

Je dána CFG G. Napsat gramatiku G_0 takovou,že napíšeme pro každý regulární fragment regulární gramatiku
$G_1, ..., G_n, G_i = (..., S_i)$
Vznikne $G_0 = (\quad, S_i)$
v G_0 se objeví jako terminály kořeny S_i
Ke každé gramatice G_i lze přistoupit několika způsoby:
G_i ... kon. automat A_i $T(A_i) = L(G_i)$
- ručně napsat G_i
- ručně automat
- nástroj lex pro všechna i

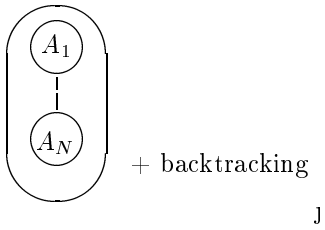
Dvě základní strategie analýzy
1. A_i vytvořím sjednocení $\cup_i A_i$, které převedu na deterministický konečný automat LA. Při ručním vytváření lze využít "princip nejdelší shody" automat musí dát i zprávu o jakou kategorii jde.
2. Dostanu od syntaktického analyzátoru zprávu, co se očekává a provedu příslušnou analýzu. Tedy spustí se jeden z automatů A_i

Lexikální automat

Strategie:

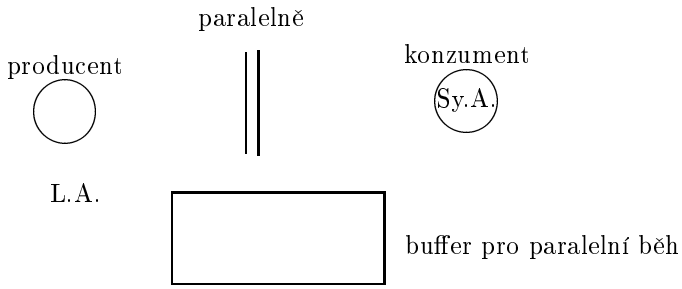
Lemma 2.1

1. přímá:
LA pracuje přímo $\Leftrightarrow^{def}$ zdrojový text a ukazatel do zdrojáku určí typ lexikální položky a aktualizují ukazatel.
2. nepřímo $\Leftrightarrow^{def}$ zdrojový text, ukazatel a typ lexikální položky určí,zda ano/ne a aktualizuje ukazatel.

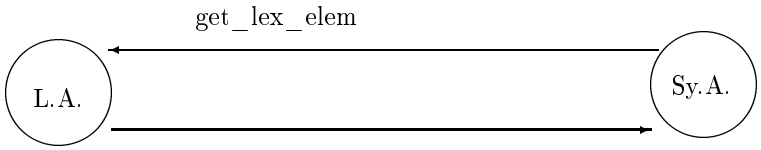


Obrázek 3: Struktura lexikálního automatu

2.1 Vztah mezi LA a SyA



Obrázek 4: Vztah mezi LA a Sya



Obrázek 5: Možná činnost lex. analyzátoru

Procedurou get_lex_elem se vyvolá lexikální analyzátor a ten po zpracování vrátí výsledek.

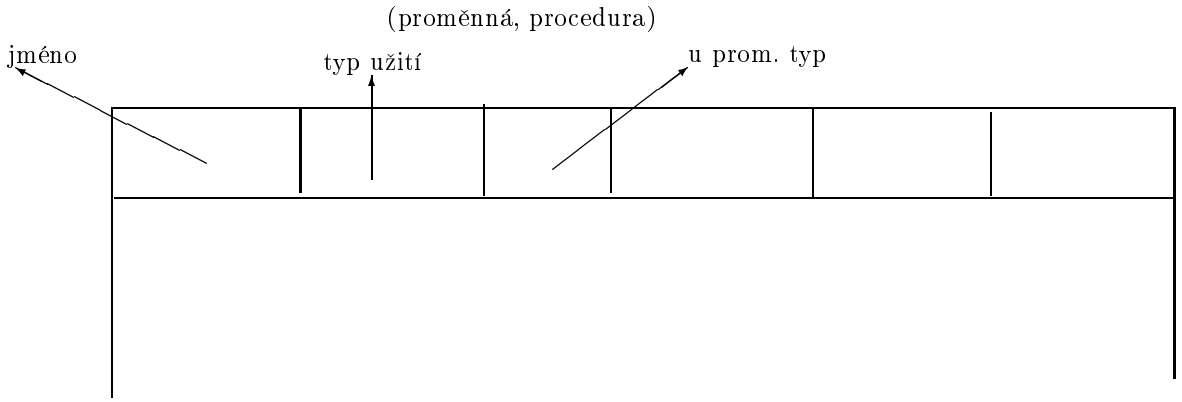
2.2 Reprezentace lexikálních položek

TYPE typy_lex_pol = (IDSYM,CONSTSYM,...
BEGINSYM,ENDSYM,...
PLUSSYM,MINUSSYM,...
GESYM,LTSYM,...
ASSIGNSYM,....., NULLSYM)
NULLSYM nemusí být vždy definován, jestliže se pošle syntaktickému analyzátoru, znamená to, že v L.A. došlo k chybě.
lex_pol = RECORD synt_kat : typy_lex_pol

Obrázek 6: Další varianta činnosti lexikálního analyzátoru

```
CASE typy_lex_pol
   IDSYM : int
```

Mohou být i další položky, např. pointr, kde se vyskytl poprvé.
další úkol L.A.: naplnění tabulek (tabulka identifikátorů, tabulka konstant)

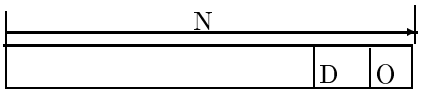


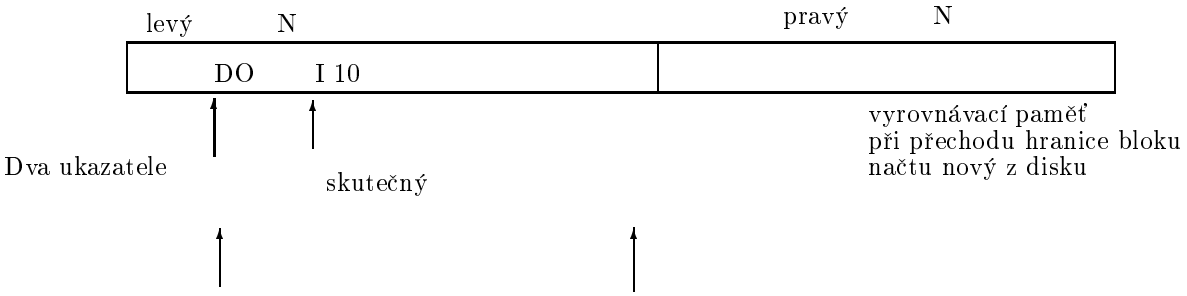
Obrázek 7: Tabulka identifikátorů

Položkou tabulky může být u jména:
přímo jméno (omezená velikost) nebo ukazatel na haldu (tabulka jmen, prostor pro řetězy), kde mohou být libovolné délky; Tabulka může být různě implementována (organizace dat).

Příklad 2.2
FORTRAN:
DO I 10 I= 1,10 cyklus for
DO I 10 I= 1.10 přiřazení
Lexikální analyzátor musí číst až za konec lexikální položky, aby rozlišil o jakou položku se jedná.

Máme implementovat procedury:
– čti_znak()
– vrať_znak()
pro čtení z disku (chceme číst lexikálníjednotku).
Mezi disk (text) a L.A. dáme modul čtení zdrojového textu
Obecný tvar cyklu:



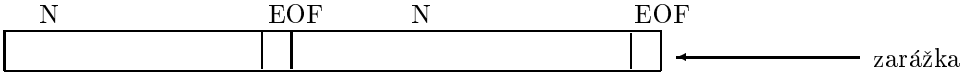


Obrázek 8: čtení z disku

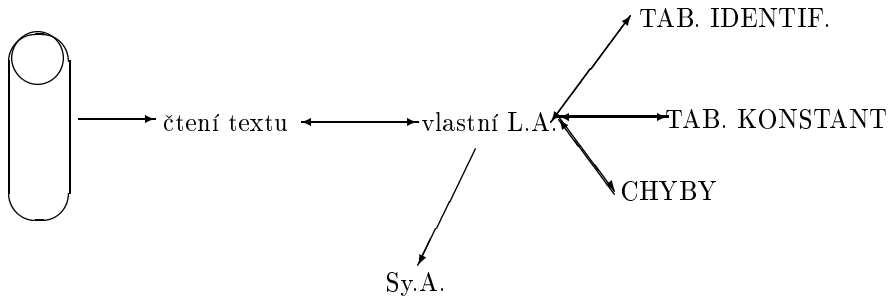
WHILE NOT EOF DO *rozpoznávej položky*

Blok délky N - není vhodné používat

zefektivnění: nebudeme testovat obojí(konec bloku i souboru)
nahradíme konec bloku značkou **EOF**, testujeme pak jen jednou, šlo-li o konec bloku či souboru



2.3 Lexikální Analyzátor



LEX[1]
nástroj na psaní lexikálních analyzátorů

[1] viz skripta str. 30 - 34

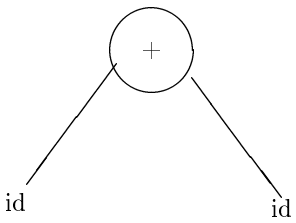
TABULKY
–I lex. analýza
–II analýza jmen a rozsahů[2]
I – tabulky konstant, jmen, identifikátorů
Organizace tabulky identifikátorů
– nesetříděné
– setříděné – klíčová slova (setříděné dopředu)
– AVL stromy (pro identifikátory)

Příklad 2.3
FORTRAN
délka identifikátorů 1...6 znaků
pro každou délku zvláštní strom
– hešovací tabulky
Tabulka konstant
– abstraktní datový typ, který vyváží operace nad konstantami
V době překladu je možné třeba spočítat
např. : 2 * PI na dostatečný počet míst -vlastní procedura

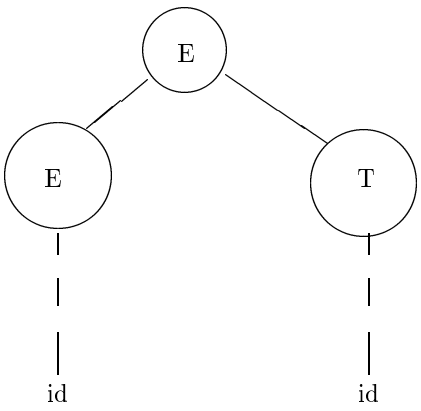
3 Syntaktická analýza

Vstup: řetěz lexikálních položek
Výstup: syntaktický strom

Příklad 3.1
(id,4) (addsym,_) (id,2)



derivační strom



Obrázek 9: Syntaktický a derivační strom

$$vraz \;=\; vraz \;\times\; vraz \;+^{sjednoc.}\; i$$

Proč syntaktický strom:
– kratší než derivační⇒ méně paměti
– využívá se v dalších fázích překladu (typová kontrola)
Úkol Sy.A.:

[2] viz sémantická analýza

– detekce, zda vstup je bez syntaktických chyb
IF ano THEN vytvoří syntaktický strom (pro sém. anal.)
ELSE zpracování a zotavení z chyby

Obrázek 10: Syntaktický strom

Syntaktická Analýza $\rightarrow_{pedvsynt.strom}$ Semantické Analýze
Přiblížení realitě:

1. Syntaktický strom může být implementován (kódování) – tj. mezikód (může mít tvar):
   - postfixový kód syntakt. stromu, prefix, infix (nepoužívá se).
   - čtveřice (n- tice s explicitně pojmenovaným výsledkem)
   - trojice (n- tice s implicitně pojmenovaným výsledkem)
   - nepřímé trojice
   - strom

Příklad 3.2

Ukázka trojic:
$E_1 \rightarrow E_2 + T$ $(+, E_2, T, E_1)$
může nést sdruženou informaci (např. typ, uložení,...)
každý mezivýsledek se uloží do pomocné proměnné

1 : (+,a,b) //trojice se musí pojmenovávat

2. Interaktivní kooperace mezi Sy.A. a Se.A. Syntaktický analyzátor nikdy nevytváří celý strom, který by dával Se.A.
   - položková spojení (“ vytvoř list ”)
   - strukturní spojení (“ vytvoř uzel ”)

Vstupní gramatika:

$$\begin{array}{lcl} E & \rightarrow & E + T|T \\ T & \rightarrow & T \times F|F \\ F & \rightarrow & id|(E) \end{array}$$

strategicky významná pravidla (kdy se dává informace Se.A.)

$$\begin{array}{lll} E \rightarrow E + T & \&struct + (+, \quad , \quad) \\ ?E \rightarrow T^3 & & \\ T \rightarrow T \times F & \&struct \times (\times, \quad , \quad) \\ ?T \rightarrow F & & \\ F \rightarrow id & \%pol(id)^4 & \\ F \rightarrow (E) & & \end{array}$$

položková spojení /řetězce začínající%
strukturní spojení /řetězce začínající&
Informace se předává až po analýze celé pravé strany.
Pořád je to CFG.
Máme terminály vstupní(vstupní gramatika) a výstupní gramatiku:

$$\begin{array}{lcll} E & \to & ET & \&struct+ \\ T & \to & TF & \&struct\times \\ F & \to & id & \%pol \end{array}$$

– Motivace: překladové gramatiky, syntaxí řízené překladové schéma.
Do gramatiky jsme si pouze poznamenali, kdy se má Se.A. informovat, ale nemáme postiženo o čem;
Přidáme procedury provádějící:

- Vytvoř nelist (pro & struct+)
- Proveď typovou kontrolu.

Budeme definovat rozšíření CFG's : atributové gramatiky:
(CFG + procedury, funkce spojené s pravidly - s každým pravidlem je spojena řada procedur) A.G. specifikuje, co se má udělat a kdy.
– rozšíření překladové CFG.
S každým uzlem (tj. symbolem gramatiky) sdružíme informaci(atribut) "nptr" (ukazatel na uzel).

Příklad 3.3
Atributová gramatika:

$$E_0 \to E_1 + T$$

funkce df_list(op, spoj) //spoj = ukazatel do tabulek symbolů
– dostane lex. položku (op, spoj), vytvoří list a vrací ukazatel na tento list
df_uzel(op, levý, pravý)
– vytvoří vnitřní uzel s následníky levý a pravý, vrací ukazatel na tento uzel
Atributová gramatika:

1. $E_0 \to E_1 + T$ $E_0.nptr :=$ df_uzel(PLUS,E_1.nptr,T.nptr)
2. $E \to T$ $E.nptr :=$ T.nptr

z hlediska překladačů dochází pouze k přenosu informace
šíření informace zdola nahoru

3. $T_0 \to T_1 \times F$ $T_0.nptr :=$ df_uzel(TIMES,$T_1.nptr, F.nptr$)
4. $T \to F$ $T.nptr := F.nptr$

přenos informace se ani nemusí psát

5. $F \to i$ $F.nptr :=$ df_list(IDSYM,spoj)

//spoj – uk. do tabulky symbolů
i je lexikální položka

6. $F \to (E)$ $F.nptr :=$ E.nptr

je z hlediska sémantiky jednoduché pravidlo
Není řečeno, kdy se má funkce provést, je pouze k tomuto pravidlu přiřazena.

Dodatek k překladové gramatice: máme také ukazatel nptr
přidá se : & struct = df_uzel (PLUS,E_1.nptr,T.nptr)
sami musíme dát do E_0
Protože &struct je v pravidle, je dáno, kdy se provede funkce(atributová gramatika k překladové)

Definice 3.1
Překladová gramatika je postfixová $\Leftrightarrow$ každá pravá strana je tvaru

$$(vstupn\ \ abeceda \cup neterminln\ \ abeceda)^*.vstupn\ \ abeceda^*$$

Příklad 3.4

$$S_0 \to if \;\; BE \diamond then \;\; S_1 \diamond \;\; else \;\; S_2 \diamond fi^5$$

Před vyhodnocováním S je třeba přerušit analýzu, vygenerovat podmíněný skok, pak analyzovat S_1, přerušení- skok, analyzovat S_2, konec příkazu, resp. nový příkaz.

Pravidlo překladové gramatiky:

$$S_0 \;\; \to if \;\; BE \;\; \underline{IFJUMP} \;\; then \;\; S_1 \;\; \underline{JUMP} \;\; else \;\; S_2 \;\; \underline{C} \;\; fi$$

Není postfixové schéma překladu!

Přepis na postfixový překlad:

$$S_0 \;\; \to \;\; if \;\; BE \;\; M^6 \;\; thenS_1M'N^7 \;\; else \;\; S_2 \;\; fi$$

$$M \;\; \to \epsilon^8 \text{ a za ním vyvolání sémantické fce.}$$

3.1 Překladové CFGs

Definice 3.2 PŘEKLADOVÉ CFG

Překladová CFG (P_CFG) je CFG G=$(N, \Sigma \cup \Delta, P, S)$, kde všechny symboly mají tentýž význam jako v CFG a navíc $\Sigma \cap \Delta = 0$. Místo věta se někdy říká charakteristická věta.

Definice 3.3

Dále def. dva homomorfismy h_{in}, h_{out} :

$$\begin{array}{rcl} h_{in} & : & N \cup \Sigma \cup \Delta \;\; \to N \cup \Sigma \;\; , \\ h_{out} & : & N \cup \Sigma \cup \Delta \;\; \to N \cup \Delta \end{array}$$

$$klademe \quad h_{in}(X) = \begin{cases} X, & x \in N \cup \Sigma \\ \epsilon, & jinak \end{cases}$$

$$h_{out}(X) = \begin{cases} X, & x \in N \cup \Delta \\ \epsilon, & jinak \end{cases}$$

h_{in}, h_{out} rozšíříme na řetězy.

Je li w věta, pak $h_{in}(w)$ nazveme vstupní věta, $h_{out}(w)$ výstupní věta.

Překlad generovaný P_CFG G:

$$\{< h_{in}(w), h_{out}(w) > \mid \;\; w \;\; je \;\; vta \;\; v \;\; G\}$$

Deterministický překlad, jednoznačný – podmínka

$$A \to \alpha, A \to \beta \qquad \alpha \neq \beta \Rightarrow h_{in}(\alpha) \neq h_{in}(\beta)$$

Definice 3.4

Je li dána P_CFG $G = (N, \Sigma \cup \Delta, P, S)$

pak gramatiku $G_{in} = (N, \Sigma, P_{in}, S) pro P_{in} = \{A \to h_{in}(\alpha) | A \to \alpha \in P\}$

nazveme vstupní gramatikou (analogicky výstupní gramatika).

Vstupní a výstupní gramatiky charakterizují překlad. Pořadí neterminálů je ve všech pravidlech stejné.

---

[5] jedno pravidlo gramatiky
[6] pomocný terminál
[7] pomocný terminál
[8] nesmí se dostat do syntaxe

3.2 Překladová schémata

Definice 3.5 PŘEKLADOVÉ SCHÉMA

Překladové schéma (PS) je systém $(N, \Sigma, \Delta, P, S)$, kde jednotlivé položky jsou:
N - neterminály
Σ - abeceda vstupních symbolů
Δ - abeceda výstupních symbolů
S - kořen schématu
P - množina pravidel tvaru $A \to \alpha, \beta$[9], kde $\alpha \in (N \cup \Sigma)^*$, $\beta \in (N \cup \Delta)^*$ [10]

(Permutace je identitou $\Rightarrow$ PS se nazývá jednoduché.)

Definice 3.6

Překladový zásobníkový automat *PZA* $(Q, \Sigma, \Gamma, \Delta, \delta, q_0, Z_0, F)$, kde Δ je výstupní abeceda ,

$$\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma^* \to Q \times \Gamma^* \times \Delta^*$$

Věta 3.7

Ke každé P_CFG existuje PZA takový, že jimi generované překlady jsou totožné.

Důkaz: pro G $= (N, \Sigma \cup \Delta, P, S)$klademe PZA $(\{q\}, \Sigma, N \cup \Sigma \cup \Delta, \Delta, \delta, q, S, 0)$ [11] kde

δ :	$\delta(q, \epsilon, A)$	obsahuje	(q, α, ϵ)je li	$A \to \alpha$
	$\delta(q, a, a)$	=	$\{(q, \epsilon, \epsilon)\}$pro	$a \in \Sigma$
	$\delta(q, \epsilon, b)$	=	$\{(q, \epsilon, b)\}$	$b \in \Delta$//napíše na výstupní pásku

□

Věta 3.8

Ke každému jednoduchému překladovému schématu existuje překladová gramatika a naopak.[12]

3.3 Atributové gramatiky

[13]

A.G. ... jedna z komponent bude CFG G=(N, Σ, P, S), kde
– $\forall X \in N \cup \Sigma$ je množina atributů A(X)

Příklad 3.5

atributem pro X může být : typ, datová oblast, ...

Budeme značit $a \in A(X)$: ozn. X.a
– s každým pravidlem

$$p \in Pp : X_0 \to X_1 \ldots X_n$$

je sdružena množina sémantických funkcí [14] R(p)

$$R(p) = \{X_i.a^{15} p := f(X_j.b, \ldots, X_k.c) | i, j, k \in < 0, n >\}$$

– speciální typ sémantických fcí. jsou bool. fce, na které se budeme odkazovat jako na množinu bool. podmínek B(p)

$$B(p) = \{B(X_j.b, \ldots, X_k.c) | j, k \in < 0, n >\}$$

Obrázek 11: dědičné a syntetizované atributy

Syntetizované atributy : hodnota atributu levé strany se počítá z hodnot atributů pravé strany.
Ozn:
dědičné atributy —od symbolu X I(X)
syntetické atributy S(X)

Definice 3.9 ATRIBUTOVÁ GRAMATIKA
Atributová gramatika je čtveřice AG=(G,A,R,B), kde G=(N,Σ,P,S) je CFG, A= $\cup_{x \in N \cup \Sigma} A(x)$, kde A(x) je konečná množina atributů symbolu X.
$R = \cup_{p \in P} R(p)$, kde R(p) je konečná množina sémantických fcí pro pravidlo p.
$B = \cup B(p)$, kde B(p) je konečná množina podmínek pro pravidlo p.
Pro $X \neq Y$ platí$A(X) \cap A(Y) = 0$

Pro každý výskyt uzlu X v syntaktickém (nebo derivačním) stromu libovolné dané věty z L(G) platí , že pro určení hodnoty libovolného atributu $a \in A(X)$ (tj. X.a) je aplikovaná nejvýše jedna sémantická fce.

$$\forall p \in P p = X_0 \to X_1 \dots X_n$$

nechť AF(p) je množina definujících výskytů atributů,

$$AF(p) = \{x_i.a | x_i.a := f(\dots) \in R(p)\}$$

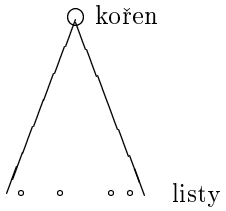
atribut X.a se nazývá *syntetizovaný* $\Leftrightarrow^{def} \exists p : p \equiv X \to X \quad \in P \wedge X.a \in AF(p)$.
X.a se nazývá *dědičný* $\Leftrightarrow^{def} \exists p : p \equiv Y \to uXv \wedge X.a \in AF(p)$
Pro daný symbol $X \in N \cup \Sigma$ značme množinu všech jeho syntetizovaných atributů S(X) a množinu dědičných atributů I(X).
Lze dokázat:

$$S(X) \cap I(X) = 0 \text{ pro libovolné X.}$$

Zatím jsme nevyloučili (ale nechceme), aby kořen měl dědičné atributy a listy syntetické.

Definice 3.10 A
G se nazývá *úplná* $\Leftrightarrow$ pro určení hodnoty lib. atributu je aplikovatelná právě jedna sémantická fce a všechny atributy se dají vyhodnotit, tj. jsou zadány všechny

[11] přijímá prázdným zásobníkem
[12] kapitola 4, 4.22 a zvláště 4.5 NASTUDOVAT
[13] ve skriptech: překladové atributové gramatiky(tj. atributová gramatika pro překladovou gram. Zde pro normální CFG)
[14] sém. pravidel
[15] atribut příslušný symboluX_i v tomto pravidle



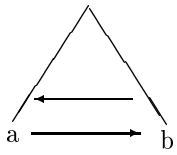
Obrázek 12: Dědičné a syntetizované atributy

dědičné atributy kořene, syntetické atributy listů a "každý další atribut lze spočítat ".[16]

Příklad 3.6

- Syntetizovaný atribut : typ - je v listu[17]
- Dědičný atribut kořene : od které adresy se přiděluje paměť pro přeložený kód.

Dále jsme nevyloučili definice kruhem.



Obrázek 13: Definice kruhem

Definice 3.11
AG je *dobře definovaná* $\Leftrightarrow^{def}$ pro libovolnou větu lze v jejím syntaktickém stromu vyhodnotit všechny atributy.

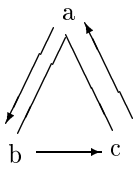
AG neobsahuje cyklické závislosti mezi atributy.

Příklad 3.7
2 druhy cyklických závislostí :

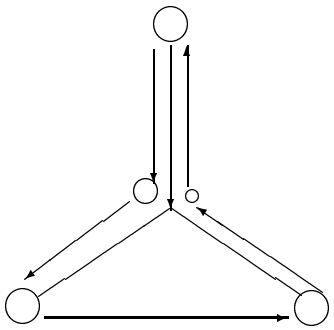
- v rámci jednoho pravidla (lokální je lépe identifikovatelná)
- globální cykly

Cykly způsobí, že atributy nevyhodnotíme.

[16] věc návrhu
[17] hodnota se vezme v tabulkách



Obrázek 14: Lokální cyklus



Obrázek 15: Globální cyklus

Příklad 3.8

Dána CFG:

$S \to E$
$E \to E + T | T$
$T \to T \times F | F$
$F \to c$[18]$|(E)$

Cíl: vyhodnotit konstantní výrazy.

1. navrhneme atributy

typ atributu val

tj. máme {S.val, E.val, T.val, F.val, c.num} = A ($=\cup_{X \in N \cup \Sigma} S(X)$)

budou to syntetizované atributy

Pravidla:

0. $S \to E$ $\{S.val := E.val\}$[19]

- 1. $E_0 \to E_1 + T$ $\{E_0.val := seti(E_1.val, T.val)\}$
- 2. $E \to T$ $\{E.val := T.val\}$
- 3. $T_0 \to T_1 \times F$ $\{T_0.val := mult(T_1.val, F.val)\}$
- 4. $F \to (E)$
- 5. $F \to c$ $\{F.val := c.num\}$[20]

$3 + 4 \times 9$ zdrojový text

Převede se na :

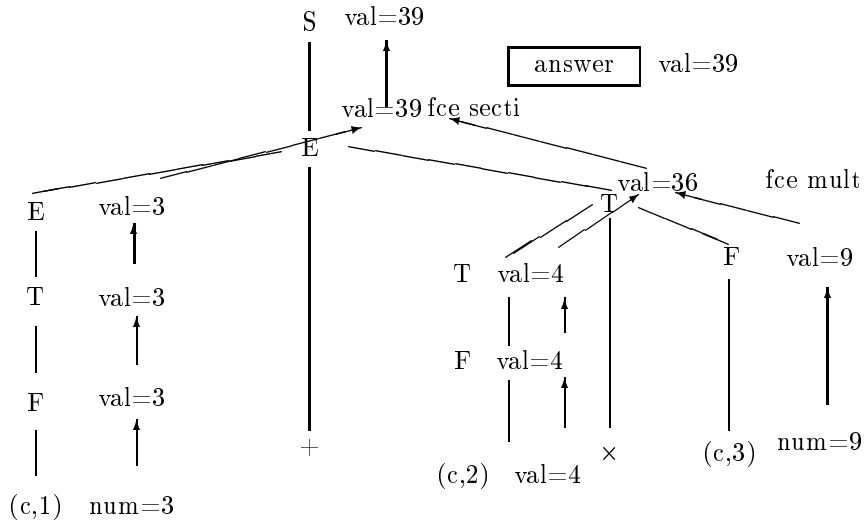
$(c, 1)$ $PLUSSYMB(c, 2)$ $MULTSYMB$ $(c, 3$[21]$)$

[18] nějaká konstantní hodnota
[19] u jednoduchých pravidel se obvykle nepíše
[20] může být také napsáno F.val:= obsah 2. komponenty lex. položky c (přes tabulku)
[21] ukazatel do tabulky

tabulka:

3
4
9

Průchod zdola nahoru — atributový strom



Obrázek 16: Atributový strom

Ze syntaktické do sémantické analýzy se snažíme předávat pouze sémanticky významné uzly (syntaktický strom).

Vytvoření překladové gramatiky:

- $0.S \to E$ *answer*
- atribut answer.val' ... I(answer)
- $0.S \to E$ $\{S.val := E.val, answer.val := S.val\}$
- $0.S \to E$ $\{answer.val := E.val\}$[22]
- Změna stromu - přibude neterminál s hodnotou 39

3.4 Implementace a uložení atributů

– atribut nahradíme globální proměnou

– atribut lze implementovat pomocí zásobníku [23]

V sémantické analýze jsou globálními proměnnými tabulky.

[22] lépe napsané

[23] pokud nelze nahradit globální proměnou

4 Sémantická analýza

obsahuje

- analýza jmen a rozsahů(tabulky, jejich organizace)

  (environment – deklarační okolí)

  implementace tabulek pro blokově strukturované jazyky, modulární prog. jazyky - export, import

- typová analýza (typové kontroly, šíření typů+ typové konverze)

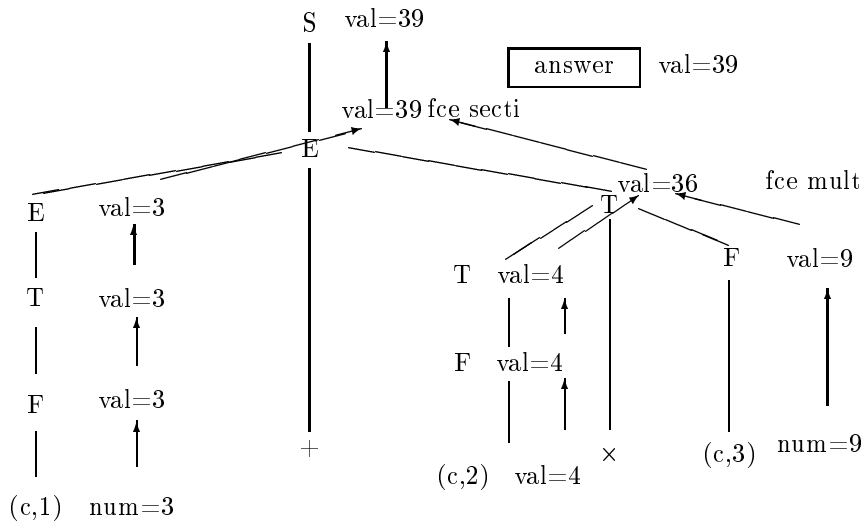
- generování mezikódu

  organizace paměti - přidělování, uvolňování+ vlastní mezikód

4.1 Analýza jmen a rozsahů aneb tabulky

Blokově strukturované programovací jazyky a tabulky [24]
Typy rozsahu:

- uzavřený
- otevřený
- aktuální



Obrázek 17: Bloková struktura

Pravidla viditelnosti:

- vidíme do aktuálního rozsahu
- a vidíme do otevřených.

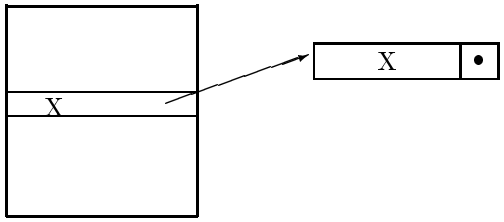
Tabulky:
2 přístupy:

[24] skripta str. 123 – 126

a) všechno v jedné tabulce

b) co rozsah, to zvláštní tabulka[25]

ad a) Jedna hash tabulka:
Při kolizi je vhodné přidávat do kolizních řetězců přes hlavu



Přijde y, hešuje se na stejné místo

Obrázek 18: hašování

kam zařadit y?:

- za x - není vhodné
- před x (na začátek seznamu) – kolizní řetězec se pak chová jako zásobník.

Jak implementovat blokovou strukturu? :

– s každým jménem je uložena identifikace rozsahu

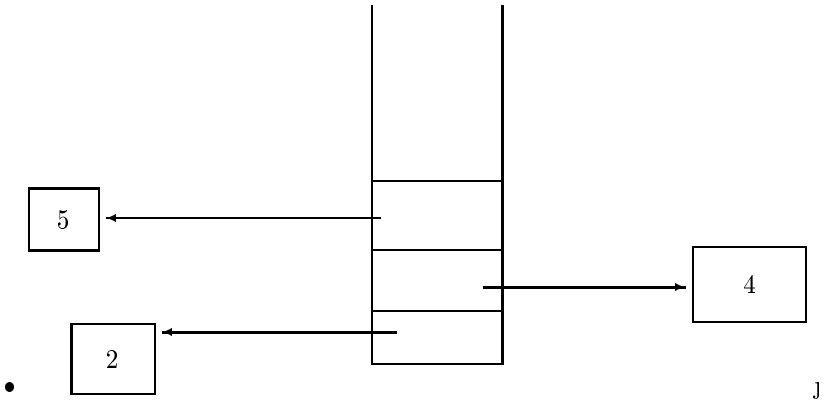
Uzavření rozsahu – odstranit všechny proměnné , které mají dané číslo rozsahu.
Strategie všechno v jedné tabulce – rychlá, trochu náročná na programování, ale je nevhodná pro jazyky umožňující separátní kompilaci.
ad b) Co rozsah, to zvláštní tabulka
Při vnořené deklaraci

- jsou tabulky spojeny ukazateli do řetězce.
- nebo ve zvláštním zásobníku[26] máme ukazatele na tabulky, jak se otevíraly.

Implementace:

1. Máme zásobník, kam se ukládají proměnné v nových blocích
   a *scope- stack*[27]
   – uloženy ukazatele na první položku daného rozsahu v zásobníku.
2. Důležitý je zásobník rozsahů, který obsahuje ukazatele na jednotlivé tabulky (jsou umístěny libovolně).
   Výhoda : dají se vhodně organizovat.
   - tabulky jsou zřetězeny
   - všechny tabulky na jednom zásobníku, druhý zásobník rozsahů obsahuje ukazatele do tohoto prvního zásobníku.

[25] vhodné pro separátní kompilaci
[26] zásobník rozsahů
[27] může být také implicitně v zásobníku - každá proměnná má položku identifikace rozsahu

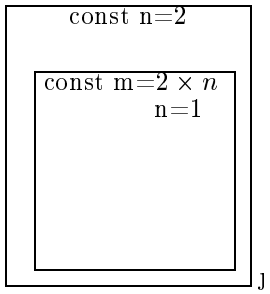


Obrázek 19: Organizace tabulek

4.2 Změna pravidel viditelnosti

(= rozšíření)
– "deklarace podél textu"(along the text)

Příklad 4.1



Obrázek 20: deklarace podél textu

z hlediska Pascalu je to špatně napsaný program stylem "podél textu"

– najde se deklarace **n** v nadřazených blocích[28]

Skupiny změn pravidel:

– řízení viditelnosti (export, import)

– změny viditelnosti (př. WITH jméno recordu)

– implicitní deklarace

– overloading (vícenásobné výskyty jmen)

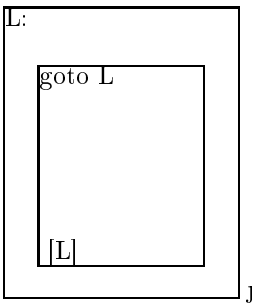
[28] porušena viditelnost

4.2.1 Implicitní deklarace

Příklad 4.2

implicitní deklarace návěští

jazyk, kde není povinnost deklarovat návěští Zde nemohu říci, co to L je. Může se vyskytovat také v tomto bloku. Nevím to, dokud se neuzavře tento rozsah. Problém se řeší pomocí techniky "*back-patching*". Nevyřešené goto (všechny výskyty) zřetězím, narazím-li na deklarace L v bloku — doplním všem v seznamu adresu, jinak po opuštění rozsahu doplním adresu z nadřazeného bloku.

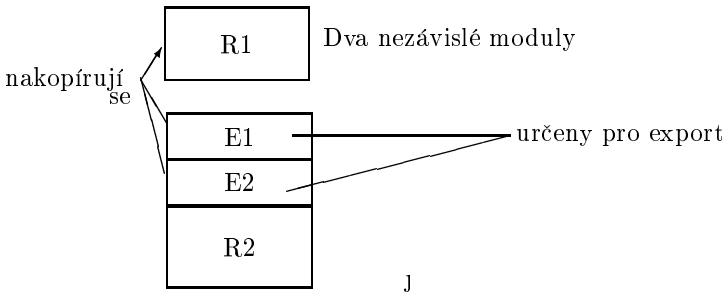


Obrázek 21: použití backpatchingu

Mimo to některé jazyky umožňují implicitní deklarace proměnných (BASIC), což může způsobovat těžko hledatelné chyby.

4.2.2 Řízení viditelnosti (export,import)

1. duplikace příslušného rozsahu (části rozsahu)



Obrázek 22: export jmen

Nevýhoda: větší nároky na paměť
Výhoda : stejné vyhledávání

2. bez duplikace
je potřeba doplnit "příznaky"[29] a "spoje"
+ změněná pravidla pro vyhledávání

[29] např., že toto jméno je určeno pro export

Nevýhoda: náročnější na implementaci
může být časově pomalejší

4.2.3 Záznamy (record) a jejich položky

Algoritmus 4.1

1. a,r record
   1.1. A:record
      1.1.1. A:real;
      1.1.2. c:bool;
   1.2. end;
2. end;

R.A.A
R.C někde je možné , nedochází-li k nejednoznačnostem jako např. v Cobolu
ad duplikace:

Pro každý typ záznamu se zřídí tabulka symbolů (σ_o nejde na zásobník, je to atribut záznamu R.x – ověří se, zda je R záznam.

Je výhodnější mít je uspořádány ve stromu nebo setříděné než jako hashovací tabulku.

ad bez duplikace:
„čísla záznamů“

každá proměnná by měla číslo záznamu = 0 (implicitně nebo explicitně

0 = normální proměnná (typu int nebo záznam) položky záznamu mají číslo k závisející na hloubce zanoření záznamu.

0	R	r
k	A	l
k	X	
k	A	
k	C	

Hledání je zde pochopitelně pozměněné:

Příklad 4.3
hledá se R – hledá se R, které má k rovno 0 R.X – hledá se X, které má k rovno 1 atd.
obyč. prom. Z – hledá se Z, které má 0

4.2.4 Pravidla pro export

umožňují :[některá] jména v lokálním rozsahu mají být viditelná mimo tento rozsah (MODULA - 2, Ada, Pascal – prog. jednotka Unit).

Příklad 4.4

Algoritmus 4.2

Module Int-Stack
  Export Push,Pop;[30] [31]
  const Max = 1000;
  var

[30] je viditelné
[31] uvedeno co se vyváží, někde musí být uvedena i signatura (parametry)

```
        Stack:ARRAY...
        TOP:INT;
    Proc Push;
    Proc Pop;
    (* Tělo modulu *)
```

Jak korektně zpracovat *export-pravidla*
typy exportu:

– jednorázový (jednoúrovňový)

– víceúrovňový (to co dovezu, mohu dále vyvézt).

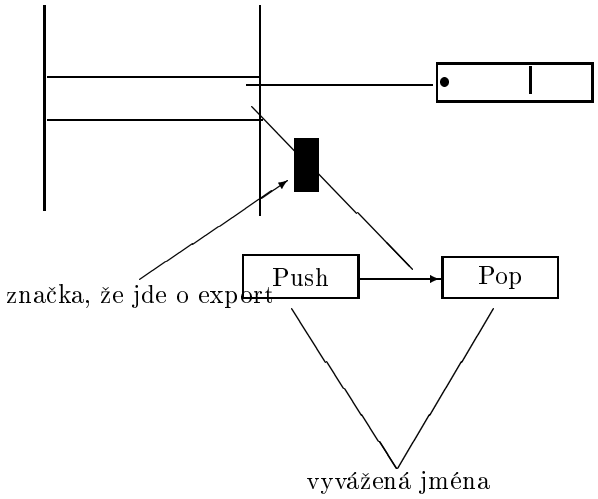
Způsob jak "dovézt" do požadovaného rozsahu:

- duplikací
- referencí.

ad Duplikace:
při otevírání tabulky

– příznak "export"

při uzavírání — najdou se všechny položky tabulky s příznakem a přesunou se kam je třeba
jednorázový — při přenosu se příznak vymaže
ad Reference:
vše v jedné tabulce:



Obrázek 23:

Při zpracování tohoto rozsahu značku neberu v úvahu při opuštění- před značkou jsou lokální proměnné rozsahu, které odstraním, nevymažu to, co je za značkou (i když má stejné číslo rozsahu).

Exportovaná jména budou na konci kolizních řetězců (v tabulce) nebo u kořene stromu.

4.2.5 Pravidla pro import

– jsou svázána s exportem

– dovážet je dovoleno pouze to co lze někde vyvážet

Protějšek musí obsahovat klauzuli EXPORT nebo export může být implicitní(v Pascalu : interface . . .).

Rozsah je klasifikován jako

- importující(př. procedury)
- neimportující(modul, unit, package)

USES dovoz všeho
IMPORT jen některé
je nutné dát pozor na jednoznačnost (kolize jmen)

4.2.6 Změněná pravidla pro hledání, pascalské WITH

"co rozsah to tabulka"

– každý(typ) záznam má svou tabulku
na vrchol dám tabulku nebo pointer na tabulku pro záznam, při END se ze zásobníku daná tabulka (pointer) odstraní

– technika "čísel záznamů"

1. kopírováním
pro daný záznam zkopíruji všechny položky záznamu, které mají číslo záznamu podle čísla v záznamu (hlavním)
– nároky na paměť
2. bez kopírování
zřídím zásobník záznamů, ukládám čísla záznamů; mám - li A : prohledávám zásobník (od vrchu), ke kterému záznamu patří, a není-li v záznamech hledá se jako normální proměnná (mimo zásobník záznamů)

Možné konflikty:
use $p_1, \ldots, p_n$ - seznam unitů, ze kterých se dováží všechna jména. Obvykle dojde-li ke kolizi mezi jmény v unitech - nedoveze se nic nebo se ohlásí chyba.

1. duplikace - vše co se v unitech nabízí, se doveze
(– nároky na paměť , možné zpomalení při kopírování)
2. bez duplikace - musí se projít všechny unity, aby se odstranily možnosti kolizí
(– náročné na čas)
3. kompromis: při nalezení jména, které není známé , vyhledá se v unitech a řádek tabulky se zduplikuje
(selektivní import).

4.3 Typová analýza

Typy v prog. jazycích - viz principy prog. jazyků.

Úkol typového analyzátoru (T.A.):

U každého listu (synt. stromu) máme v tabulce atribut typu (z analýzy jmen a rozsahů a " vlastní typová analýza")

Pro každý operátor *op* arity *n* musí být dána tabulka s typy výsledků na základě typu argumentů:

$$op_n : \times_1^n typ \rightarrow typ$$

Může být typ $\perp$ (výsledek operace nedefinován).

Úkolem T.A. je tedy určit typy vnitřních uzlů (identifikace operátorů).

Jako výsledek typové kontroly dostanu:

– $\perp$

– typ výsledku (u složitějších jazyků množinu možných typů)

S typem souvisí přetížení operátorů a převádění typů (typové transformace).

Ad přetížení operátorů:

Pokud je množina - typ se určí z kontextu přes dědičné atributy - pokud není průnik jednoprvkový - chyba

Ad typové tra - ce:

Jazyk tyto konverze může, ale i nemusí umožňovat. Speciální typ : void $\neq \perp$ nemá typ, nenastane typová chyba

Programovací jazyk může být z hlediska typů hodnocen jako:

- statický silně typovaný (jazyk) – všechny typy se určí při překladu,
- s**typový systémem sound** – v době kompilace nelze určit všechny typy, ale lze jednoznačně určit, že při běhu nenastane typová chyba,
- dynamický - ostatní,musí se vygenerovat kód na typovou kontrolu za běhu.

Typ:

- statický - všechny atributy typu jsou známy v době překladu (znám paměťové nároky na uložení proměnných tohoto typu, u strukturovaných typů znám i přístup ke komponentám). V principu mohou být ve statické oblasti paměti.
- dynamický (např. není přesně určena některá z komponent, uložen na zásobníku.
- silně dynamické typy (souvisí s dobou životnosti proměnných). Příkazy mohou dynamicky změnit velikost typu (např. pole), uloženy musí být na haldě.

Příklad 4.5

dynamické typy: array[1...10] of T – typ, který neznám
array[m...n] of integer – n, m nejsou konstanty, ale proměnné

4.3.1 Typový systém

Typový výraz : (druh konstrukce v prog. jazyce)
(viz skripta kapitola 7.1)

1. Základní typ je typový výraz (T.V.).
2. Jméno typu je typový výraz.
3. (a) pole: Je-li I ordinální typ a T typový výraz, pak array(I,T) je T.V.
   (b) Jsou-li T1, T2 T.V., pak $T1 \times T2$ je T.V.
   (c) záznam: Jsou-li $n_1 \ldots n_k$ jména a $T_1, \ldots T_k$ T.V., pak

$$record(X_1^k(n_i, T_i))$$

je T.V

(d) ukazatel : Je-li T T.V., pak i pointer(T) je T.V.

(e) funkce: Jsou-li $T, T_1, \ldots, T_k$ T.V., pak i

$$X_i^k T_i \to T$$

je T.V.

4. Proměnná α nabývající hodnot z množiny typových výrazů je T.V.

Příklad 4.6
kd: $list\_of(\alpha) \to \alpha$
Př. jazyka umožňující4. – ML

Příklad 4.7
CFG:[32]
$P \to D$[33]$; S$[34]
$D \to D; D | id : T$[35]
$T \to char | int | array\ \ [num]\ \ of\ \ T | \uparrow T | T' \to' T$[36]
$E \to literal | num | id | E$[37]$mod\ \ E | E[E] | E \uparrow | E(E)$
$S \to id := E | if\ \ E\ \ then\ \ S | while\ \ E\ \ do\ \ S | S; S$

Statická sémantika typů. Označení :
wft ... dobře utvořený typ.
wfd ... dobře utvořená deklarace.
wfe ... dobře utvořený výraz.
wfs ... dobře utvořený příkaz.
wft:
wft(char)
wft(int) //základní typy jsou wft
Indukční krok: $wft(T_1) \Rightarrow wft(\uparrow T_1)$ (ukazatel)
$wft(T_1) \Rightarrow wft(array[num]T_1)$(pole)
$wft(T_1) \wedge \ \ wft(T_2) \Rightarrow wft(T_1 \to T_2$ (funkce)
Statická sémantika deklarací
$wft(T) \Rightarrow wfd(id : T)$
$wfd(D_1) \wedge wfd(D_2) \Rightarrow wfd(D_1; D_2)$
Statická sémantika výrazů
wfe(literal)
wfe(num)
wfe(id)
⋮
Přidání typu boolean:

$$Be := \ \ tt | ff | E_1 < E_2 | \ldots | BorB | \ldots$$

Statická sémantika příkazů:
$wfe(E) \ \ \Rightarrow \ \ wfs(id := E)$
.
.
$wfe(E) \wedge wfs(S) \ \ \Rightarrow \ \ wfs(if\ \ E\ \ thenS)$

[32] daná CFG představuje pouze syntaktická pravidla
[33] deklarace
[34] příkaz
[35] typ
[36] funkce
[37] výraz

Operační sémantika
Každému neterminálu se přiřadí atribut type (např. D.type)
terminály: mají také type – syntetizovaný atribut
daný v tabulce

– zapsat do tabulky (zprac. D)

– vyhledat v tabulce (jinak)

//zbytek skripta obr 7.2
Typová kontrola pro mod (atributová gramatika):
$E_0 \rightarrow \ E_1 \ \ mod \ \ E_2$
$\{E_0.type :=$
$if \ \ E_1.type = int \ \ and \ \ E_2.type = int$
$then \ \ int \ \ else \ \ type\_error\}$
pro funkci:
$E \rightarrow E_1(E_2)$
$\{E.type = (E_2.type == s \&\& E_1.type == s \rightarrow t \ \ ? \ \ t \ \ : \ \ type\_error\}$

$S \rightarrow id = E$
$\{S.type = (id.type == E.type^{38} \ \ ? \ \ void \ \ : \ \ type\_error\}$

$S \rightarrow if \ \ E \ \ then \ \ S_1$
$\{S.type = (E.type == int \ \ ? \ \ S_1.type \ \ :\}$

while .....totéž
$S \rightarrow S_1; S_2$
$\{S.type = (S_1.type == void \ \ \&\& \ \ S_2 == void \ \ ? \ \ void \ \ : \ \ type\_error\}$

Jak interpretovat ekvivalenci typů? viz Principy programovacích jazyků (typová ekvivalence).

- textová ekvivalence:
  - ⋆ strukturní ekvivalence
  - ⋆ ekvivalence jménem

Možnosti:"="

- jen na ekvivalentních typech
- kompatibilní typy (není symetrická; typ a podtyp)
- převoditelné (koercibilní) (př. int na float – když to jazyk dovoluje); také není symetrické.

4.3.2 Přetížení operátorů (a funkcí), identifikace operátorů (overloading)

Př.: +
jméno( ) může být prvek pole nebo vyvolání funkce. Musí se jednoznačně"rozřešit" (s pomocí kontextu)
+: pomocí typových operátorů (šíří se "zdola nahoru")
( ): ("shora dolů")
Je možná kombinace obojího.

[38] problém typové ekvivalence

Příklad 4.8

V jazyku Ada:

$* : int \times int \rightarrow int$ (infixový operátor násobení) a definice funkce:

function "*" (i,j:int) return complex;

function "*" (i,j:complex) return complex;

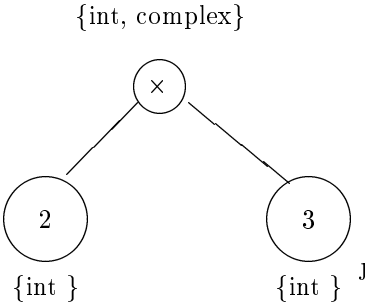
tím jsme dodefinovali :

$: int \times int \rightarrow complex$

$complex \times complex \rightarrow complex$

2,3,5 implicitně integer

z prom. typu complex



Obrázek 24:

Typ musí být jednoznačně určen z kontextu:

$(2 * 3) * 5$ bude int

$(2 * 3) * z$ bude complex

Kdyby to nešlo z kontextu rozlišit – má se ohlásit chyba.

Atribut .type – může být množina typů, nutno použít syntetizovaných i dědičných atributů.

4.3.3 Typová kontrola u polymorfních funkcí

Příklad 4.9

(ML)

fun $length(lptr) =$ if $null(lptr)$ then 0 else $length(tl(lptr)) + 1$

Volání v programu:

`length([ 'sun','snow','true'])` dává délku seznamu s libovolným typem položek

`length([10,9,8])`

Typová kontrola + typ inference (určení typu – polytyp, monotyp)

Chceme dospět k typu funkce:

$$\forall \alpha.list(\alpha) \rightarrow integer$$[39]

– odvodí se z deklarace

V programu může mít funkce různé typové instance:

např. : list(integer) → integer

Řešíme problém, zda existuje taková substituce, která učiní 2 typy ekvivalentní (např. strukturně).

Příklad 4.10 CFG

$P \rightarrow D; E$

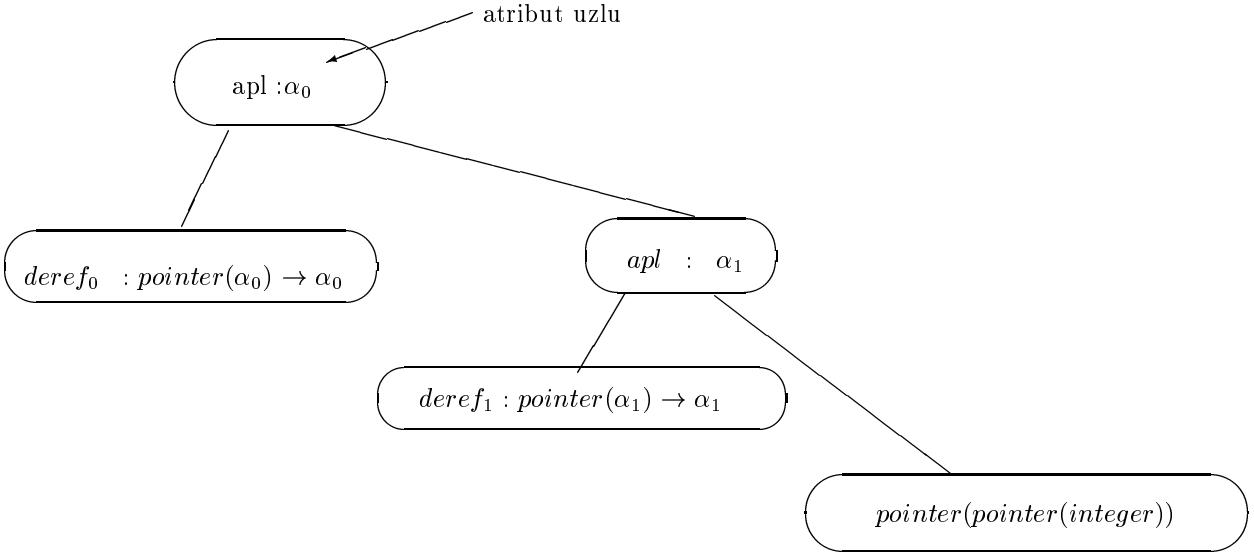
$D \rightarrow D; D \mid id : Q$

$Q \rightarrow \forall\ typovpromnn.Q \mid T$

[39] nejobecnější typ (genericky)

$T \rightarrow \ zkladn\_typ|typovpromnn|(T)|unr\_konstruktor(T)|T' \rightarrow' T|T \times T$
$typovpromnn \ \rightarrow \alpha|\beta|\ldots$
$unr\_konstruktor \rightarrow \ list|hd|tl|pointer\ldots$
$zkl\_typ \rightarrow \ integer|char|\ldots$
$E \rightarrow \ E(E)|E,E|id$

Deklarace v programu:
$deref: \ \forall\alpha.pointer(\alpha) \rightarrow \alpha;$
$q: \ pointer(pointer(integer))$
$deref(deref(q))$
Provedeme typovou kontrolu. K výrazu existuje syntaktický strom:



Obrázek 25: Syntaktický strom

Odlišnosti:

1. Různé výskyty polymorfních f-cí nemusí mít argumenty téhož typu.
2. Pracujeme s typovými proměnnými ⇒ musíme redefinovat ekvivalence typů.

   Příklad 4.11

   $E_1 \ : \ s \rightarrow s' \qquad je \ \ aplikovn \ \ E_2 \ : t$
   V imperativním případě: ? jsou ekvivalentní typy s a t
   polymorfní: s, t unifikovat – zjistit, zda existuje substituce taková, která udělá s a t strukturně ekvivalentní.

3. potřebujeme mechanismus, který bude uchovávat výsledky unifikací, protože dál se pracuje s touto informací.

Pomocné funkce pro dále uvedenou atributovou gramatiku:

1. $fresh(t)$ všechny proměnné vázané ∀ nahradí instancí nové dosud nepoužité typové proměnné , vrací pointer na uzel reprezentující výsledný typový výraz

2. $unify(m,n)$ unifikuje typové výrazy m,n (tj. výrazy na než ukazují pointry m,n), jako vedlejší efekt uchovává seznam substitucí,které unifikovaly m,n.
   Výsledek:
   - fail – výraz špatně, typová chyba
   - access

$E \rightarrow \ E_1(E_2)$
$\{p := vytvo\_list(nov\_typ\_promnn);$
$unify(E_1.type, vytvo\_uzel('\rightarrow', E_2.type, p); E.type := p\}$
$E \rightarrow E_1, E_2$
$\{E.type := \ vytvo\_uzel('x', E_1.type, E_2.type)\}$
$E \rightarrow \ id$
$\{E.type := fresh(id.type)\}$

Příklad 4.12 (
length — viz dříve) length : β
lptr : γ
if : $\forall\alpha.bool \ \times \alpha \times \alpha \rightarrow \alpha$
null : $\forall\alpha.list(\alpha) \rightarrow bool$
tl : $\forall\alpha.list(\alpha) \rightarrow list(\alpha)$
0 : int
1 : int
$+ : int \times int \rightarrow int$
match : $\forall\alpha.\alpha \times \alpha \rightarrow \alpha$
kontrola deklarace:
match(length(lptr), IF(null(lptr), 0, length(tl(lptr)) + 1))
výsledek: $list(\alpha) \rightarrow \ int$

Příklad 4.13

$$funmap(f,n) = \underline{if} null(m) \underline{then} nil$$
$$\underline{else} cons(f(hd^{40}(m)), map(f, tl(m)))$$

$$cons : \forall\alpha \cdot \alpha \times list(\alpha) \rightarrow \ list(\alpha)$$

Úkoly:

1. najít (poly)typ map.
2. tok je zákl. typ (řetězec znaků)
   tokl : tok list (seznam řetězců)
   len : tok → int
   sqroot : int → real
   určit typy funkce map ve výrazu:

$$map(sqroot, map(len, tokl))$$

[40] head

5 Organizace a přidělování paměti

statický	**dynamický**
deklarace procedury	aktivace procedury
deklarace proměnných	vazba jména na um. v paměti
scope	lifetime
okolí	stav

Co je zapotřebí řešit ve zdrojovém jazyce:

- proměnné : statické × dynamické
- typy: statické × dynamické (nutno uchovat deskriptor — tedy strukturu popisující typ,nebo ukazatel na tuto strukturu)
  U silně dynamických typů se během lifetime mění deskriptor.
- vnořování procedur a funkcí(pravidla viditelnosti → lokální a globální proměnné)
- rekurzivní procedury
- způsob korespondence mezi skutečnými a formálními parametry + jaké parametry jsou přípustné a co jsou vracené funkční hodnoty
- paralelismus
  - ko- procedury
  - synchronizace
  - komunikace
    - posílání zpráv
    - sdílené proměnné

5.1 Aktivační záznam (procedury/ funkce)

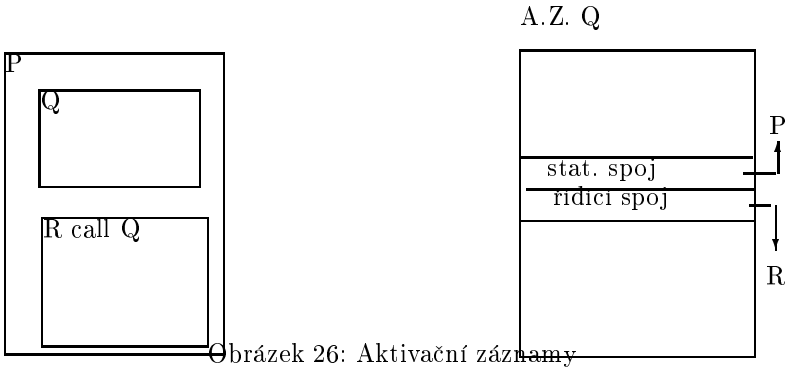
Jde o úsek paměti obsahující informace pro provedení procedury.

pomocné proměnné
lokálnídata
oblast úschovy . . . , návratová adresa, . . .
statický spoj procedura nadřazená proceduře ve zdrojáku (závisí na jazyku – nepovinné)
řídící spoj (dynamický spoj) – ukazatel na aktivující proceduru
skutečné parametry
[vracená funkční hodnota]
display může být umístěn i jinde

ad Lokální data + jejich uložení

A.Z. – má pevnou, z doby kompilace známou, za běhu již neměnnou, délku. Nevýhoda : objekty dynamického typu nelze uchovávat v A.Z. ⇒ v A.Z. je "jen" deskriptor tohoto dynamického objektu.

1. Základní typy: na jednotlivých architekturách zaberou různý počet bytů.
2. Strukturované typy:



Obrázek 26: Aktivační záznamy

- pole
- záznam

Pole:

Dimenze 1:

a[1]
.
.
.
a[10]

Dimenze 2:

- po řádcích [41] na konstantní část + část obsahující i a j
- po sloupcích [42]
- vektorování– ne jako

  a[0,0]

  a[n,m]

  ale jako vektor ukazatelů na řádky
  nevýhody: mrhání pamětí
  Výhody:
  - ⋆ není-li koprocesor, násobení je mnohem delší než sčítání, zde není nutno násobit v adresovací f-ci.
  - ⋆ když jsou pole velká(matice není celá v paměti; alà stránkování).

Poznámka 5.1

Ukládání řídkých matic neorganizovat jako 2 dim. pole, najít vhodnější uspořádání. Většina nenulových prvků může být soustředěna okolo diagonály.

3. Řetězce:
   - pevná délka
   - proměnná délka

[41] a[i][j]=báze + n * i + j; vhodné rozdělit
[42] adresace analogicky jako po řádcích

   - ⋆ stanovena maximální délka
   - ⋆ není stanovena max. délka – v A.Z. je uložen deskriptor

délka	ukazatel na řetězec

4. ad pole:

   - dynamické – ne v aktivačním záznamu.
     Řešení: do A.Z se umístí deskriptor, vlastní pole někde jinde.
     - ⋆ dynamického typu – na zásobník
     - ⋆ silně dyn. typu – na haldu

   Deskriptor pro pole A[$l_1..u_1, \ldots, l_n..u_n$] of T :

(jinak označujeme jako informační vektor)

l_1	u_1	d_1
.	.	.
.	.	.
.	.	.
l_n	u_n	d_n
n		
adr A[$I_1, \ldots, I_n$]		
konst. část		

$A[i, j, \ldots, n]$ = báze + konst. část (nezávisí na indexech) + variabilní část $(i, j, \ldots, n)$

5.2 Korespondence mezi formálními a skutečnými parametry

1. volání referencí(adresou) (Ada : inout)
   vymezí se v A.Z. tolik, kolik stačí na úschovu pro pointer.

2. volání hodnotou (Ada : in)
   vymezí se tolik paměti v A.Z., kolik potřebujeme na uložení hodnoty parametru.

3. volání výsledkem (Ada : out)
   uschová se adresa parametru, vymezí se lokální umístění (nelze použít jako vstupní), před ukončením se zkopíruje na adresu parametru.

4. volání hodnotou-výsledkem – kombinace 2) a 3)
   (lze použít pro implementaci inout (ad 1)).

5. volání jménem (hlavně ve funkcionálních jazycích, vyhodnocuje se zleva!) = textová substituce — musí se dodat procedura, která vypočte adresu skutečného parametru.

6. parametr procedura — předání adresy procedury.

5.3 Datové oblasti

Obsazení paměti: datové oblasti jsou (vzhledem k době života — lifetime)

- statické — zřízena před započetí výpočtu, konec při skončení výpočtu programu.
- dynamické — vznikají a zanikají opakovaně během výpočtu.

Nutno definovat procedury:

PŘIDĚL (in velikost,out prostor)

UVOLNI (in prostor)

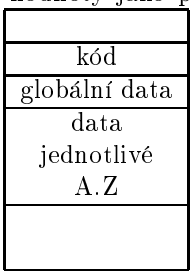
pro přidělení, resp. uvolnění oblasti.

Adresování `báze D.O. + offset`. Báze identifikuje datovou oblast. U statických D.O. mohou být absolutní adresy.

5.3.1 Statická organizace paměti (statická alokace)

Vazby se v době výpočtu nemění(vazby mezi jménem a adresou).

Možno je generovat kód v absolutních adresách. Při opakovaných vstupech do funkce můžou mít lokální data stejné hodnoty jako při posledním opuštění této funkce. Není zde statický a řídící spoj.



5.4 Dynamické přidělování paměti

pomocí :

5.4.1 Zásobník

A) Zásobník

1. Simple stack – bez dynamických typů, dynamických proměných, je vnořování procedur, s viditelností globálních proměnných

Kód při vstupu do procedury
Kód při vystupu z procedury

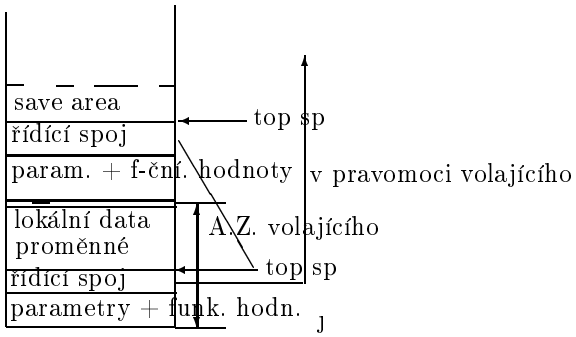
1. při vstupu do procedury : – alokace paměti pro A.Z. volané procedury + naplnění A.Z odpovídajícími informacemi + úschova registrů

2. při výstupu : – obnova registrů + dealokace paměti

Obecně přijímaný princip
if |AZ|*známá z doby kompilace*
then *na "libovůli" implementátora*
else *řídící spoj [statický spoj] doprostřed A.Z.*
vstupní volací posloupnost:
volající:

- vyhodnotí parametry (+ předá hodnoty)
- uloží návratovou adresu
- uloží"starou hodnotu" top_sp do řídícího spoje volaného
- aktualizace top_sp

volaný:



Obrázek 27: Aktivační záznam

- úschova registrů (+ dalších informací)
- inicializace svých lokálních proměnných + začne se provádět kód

výstupní volací posloupnost:
volaný:

- uloží parametry volané výsledkem
- aktualizace top_sp, obnova registrů+ předání řízení volajícímu

volající:

- vyzvedne funkční hodnotu + parametry

A) Zásobník

2. Simple stack s dynamickými datovými typy

A) Zásobník

2.1. Simple stack, který uchovává hodnotu mezi voláními dané procedury jako deklarátor static v céčku. Proměnné jsou dynamického typu ⇒ jsou všechny na haldě.

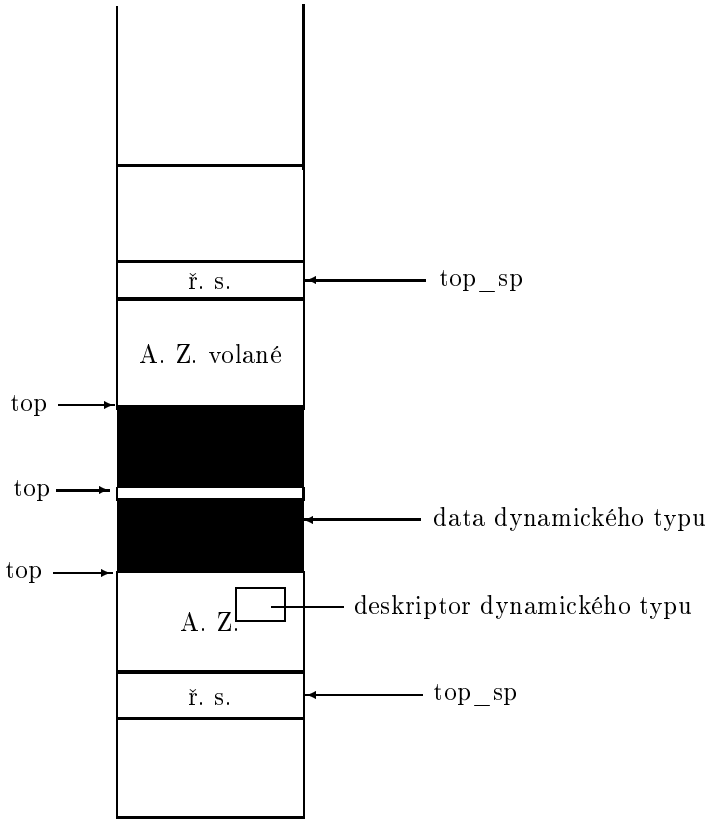
A) Zásobník

3. Stack s vnořovanými procedurami, nelokální data, která nejsou na úrovni hlavního programu a nejsou statická.

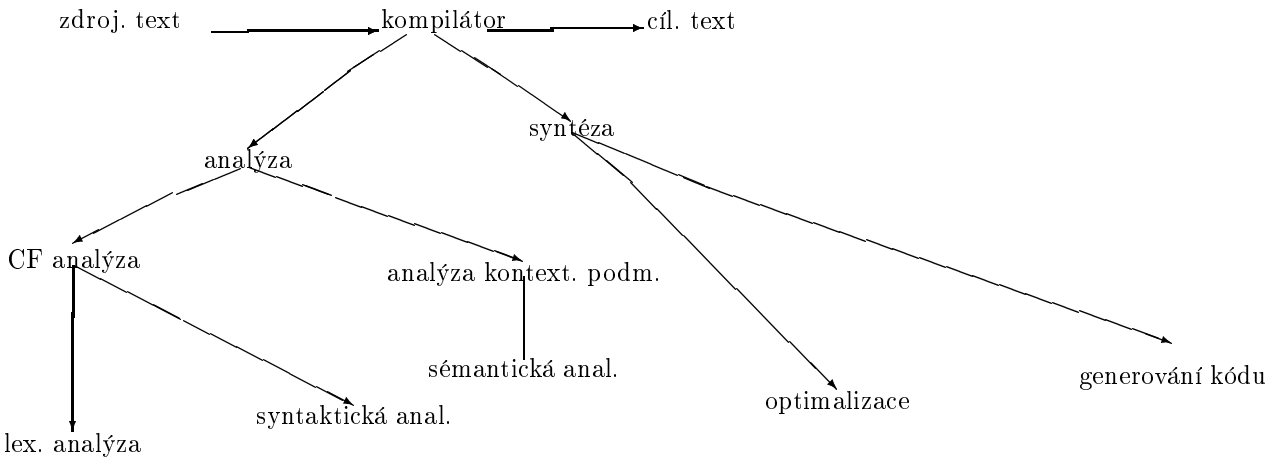
Globální / lokální proměnné Řízeno buď staticky nebo dynamicky. Dynamicky jen u interpretovaných funkcionálních jazyků. Je nutné při programování dodržovat zásady viditelnosti tak, jak jsou nadefinovány v jazyce..

STATICKY NADŘAZENÝ A.Z odpovídá poslední aktivaci podpprogramu, kam je daný podprogram vnořen. Úkolem je samozřejmě zpřístupnit nelokální data. Používají se dvě metody:

- statický spoj — ukazatel na přímo nadřazený AZ,
- display — vektor ukazatelů na začátky všech staticky nadřazených AZ, může volitelně obsahovat i ukazatel na aktuální AZ. Kam s ním ?



Obrázek 28: Zásobník



1. Jen jeden display, tedy globální. Poslední v poli ukazatelů je ten, který odkazuje na příslušný právě volané proceduře. Může být buď v statické datové oblasti nebo v po sobě jdoucích registrech (omezeno jejich počtem).
2. Do aktivačního záznamu, který se pak skládá ze dvou částí: displaye a vlastního AZ podle předchozí definice.

Poznámka 5.2 P

omocí aktivačních záznamů se řeší i bloky v procedurách. Vnořený blok jde na vrchol zásobníku aktivačních záznamů, i když to není AZ se všemi příslušnými částmi (nemusí mít display, lokální data jemu příslušného AZ jsou i pro něj lokální)

5.4.2 Procedury jako formální parametry

Organizace paměti pomocí zásobníku.

- pokud není vnořování procedur :
  formální param → skut: adresa procedury
- else if statický spoj : formální parametr → dvojice – adresa procedury + odovídající statický spoj.

Příklad 5.1

if display : kde je display

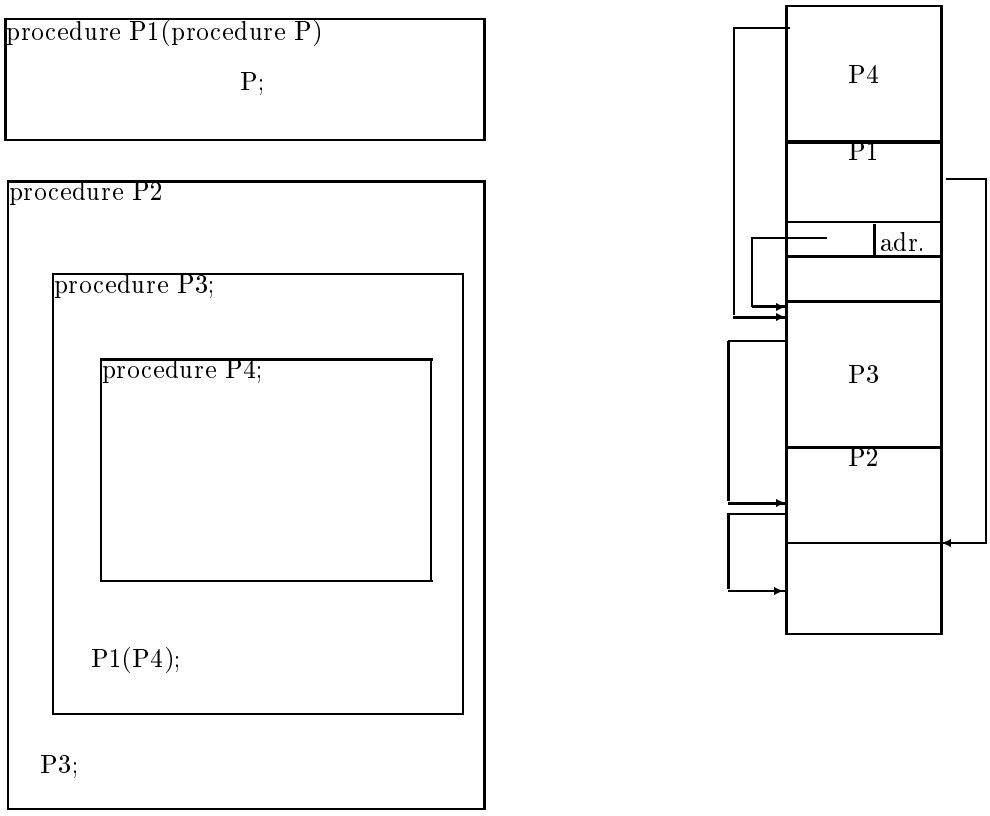
- mimo A.Z.
  adresa, odpovídající statický spoj + číslo úrovně
- v A.Z.
  adresa + zkopírovaná část displaye

5.4.3 Organizace paměti pomocí haldy

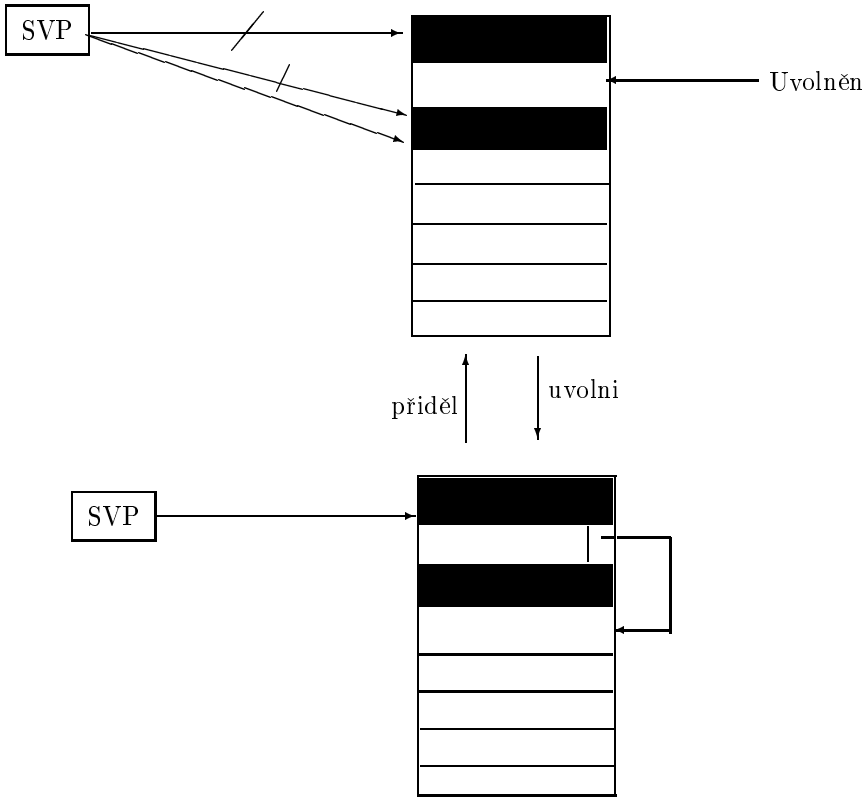
- dynamické "proměnné "(silně dynamické typy) – imperativní jazyky
- funkcionální jazyky (LISP,. . . ,ML,. . .)
- Prolog - like

Halda (heap) – souvislý úsek paměti

- přidělení paměti – počáteční
- uvolnění + příprava pro nové použití
- fragmentace paměti ⇒ scelování paměti
- A) přidělují se prvky pevné délky LISP.
  B) přidělují se prvky proměnné délky.



Obrázek 29: Procedury



Obrázek 30: Přidělování paměti

ad A)

- Počáteční přidělování paměti
- Uvolnění a příprava pro nové použití:
  - ⋆ Seznam volného prostoru

Uvolnění paměti:

1. Řízené programátorem:
   - mark, release — viz starší verze pascalu,
   - allocate (new), dispose (delete). dispose označí konkrétní prvek jako volný.

   Poznámka 5.3 SMETÍ VERSUS VÁZNOUCÍ REFERENCE

   Na jeden prvek ukazují dva ukazatele a pomocí jednoho z nich tento prvek uvolním.

   Například:
   alloc(P); alloc(Q);
   Q:=P; { Pak *Q — smetí}
   dispose(P); {Pak Q – váznoucí reference}
   {Po další alokaci paměti se může stát, že se přidělí blok na nějž ukazuje Q ! }

   Jak rozpoznat smetí? čítače referencí — každý blok obsahuje čítač ukazatelů na něj ukazujících, což je pamětově náročné. Navíc se to komplikuje cyklickými datovými strukturami.

2. Garbage collection — sběr smetí. Jde o to, které bloky jsou potenciálně „volné“. Hledají se aktivní prvky na haldě, tj. ty na které je ukazováno:
   - z vnějšku haldy,
   - z jiných aktivních prvků.

   Všechny bloky na haldě musí mít pomocný 1 bit — volný/obsazený (čistící bit).

Garbage collection :

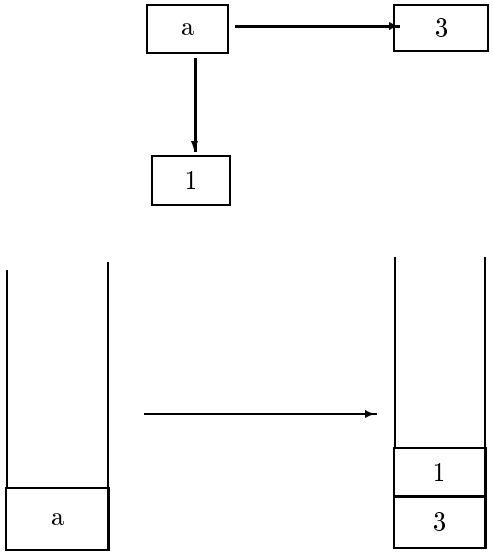
Algoritmus 5.1 V
šechny čistící bity := volný
průchod lesem (graf pointrů do haldy) a u dosažitelných prvků nastavíme čistící bit na obsazený.

Implementace průchodu lesem:

1. Použít zásobník (uložím všechny ukazatele z vnějšku na haldu – jeden po druhém)

   zpracování pointru:
   - odstranění pointru ze zásobníku a nahrazení prvky, na které ukazuje

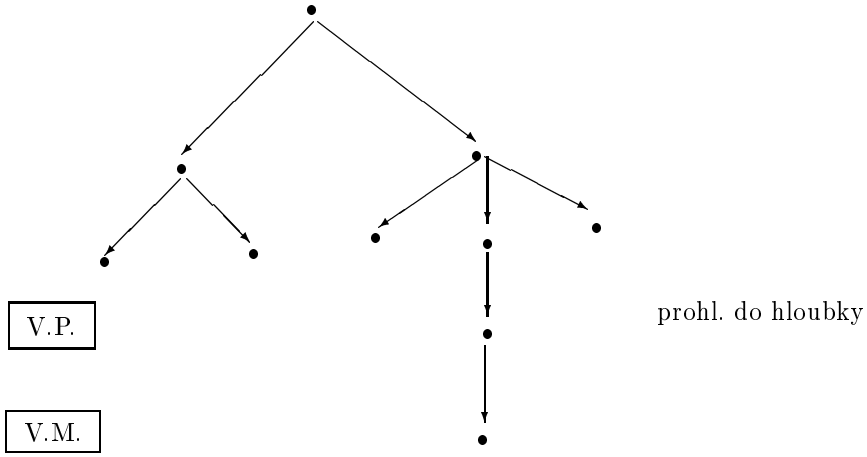
   Celý proces se opakuje, skončí až je zásobník prázdný, pak dám další pointr z vnějšku.
   Problém: nemusí být dostatek místa pro zásobník.



Obrázek 31: Garbage collection

WARNING! V době čištění paměti nikdo nesmí na haldu — nutno ji uzamknout.

2. Obracení ukazatelů
   V.P. – ukazatel na předcházející prvek
   V.M. – ukazatel na momentální prvek
   Prohledávání do hloubky – při průchodu obracím pointry
   složitost O(2n) (každý prvek procházím 2×)



Obrázek 32: Obracení ukazatelů

Poznámka 5.4 G
arbage collection — hlavně funkcionální jazyky + logické prog. Protože, hledání pointrů v aktivních prvcích na haldě je v imperativních jazycích problémem .

ad B) (Přidělování paměti v prvích proměnné délky)

- počáteční přidělování — zřejmé,
- uvolnění a znovupoužití:
  - ⋆ S.V.P. (Seznam Volného Prostoru)
    `Přiděl(100)`
    Strategie přidělování:
    - first-fit — vezmu první větší a z něj odkrojím požadovanou část,
    - best-fit — hledám optimální (nejmenší větší), ale tato strategie je ve svém důsledku horší. Vznikají malé "odkrojky".

Jak organizovat S.V.P. ? Fragmentace → scelovat:

a) částečné scelování (průběžné),

b) úplné scelování.

ad a) Než zařadím uvolňovaný blok do S.V.P. zeptám se, zda není volný předchozí nebo následující prvek. Je-li pak je spojím.

Organizace SVP

1. Seznam volného prostoru uspořádán dle adres (vzestupně)

   (a)

velikost

   (b) tabulka volných bloků

2. SVP bude dvousměrný
   volný:

+	velikost
Pred	Nasl
+	velikost

   obsazený:

-	velikost
-	velikost

3. Buddy systém — rychlejší než 2. Přidělují se bloky délky 2^n, pro každé n je udržován zvláštní S.V.P SVP[n].

   Přidělení — prvku 2^n:

   - if exist.
   - neexist.
     2^{n+1} – if exist. rozděl na 2.
     (a) přidělím
     (b) zařadím dané bloky do SVP[n]

Uvolňování prvku velikosti 2^n :

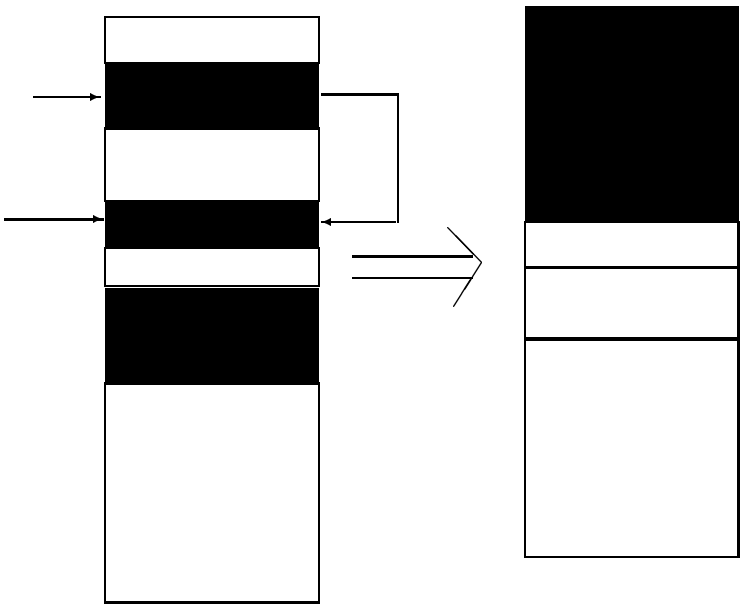
nezajímá	0/1	0 . . . 0

XOR

0 . . . 0	1	0 . . . 0

=

– adresa vhodného souseda; je-li volný ⇒ scelím a jdu na SVP[n+1] atd.



Obrázek 33:

Úplné scelování

1. Označit volné /obsazené prvky,
2. Setřepání.

Podmínky: znát umístění ukazatelů(uvnitř aktivních prvků na haldě) a uzamčít haldu.

Informace v prvcích na haldě:

- čistící bit,
- velikost prvku,
- scelovací pointer, který obsahuje adresu, na kterou bude blok přesunut.

1. (a) nastavení všech čistících bitů na volný
   (b) detekce obs./volný – značkovací algoritmus (průchod "stromy")
2. Nastavení scelovacích pointrů
   P,Q:pom_pointer;/*sekvenční průchod haldou*/
   inic.:P=Q=zač.haldy;
   next: if(P→blok == obsazeny)
   {scel.pointer=Q;

$Q = Q + sizeof(*P);\}$
$P = P + sizeof(*P);$
goto next;

3. Aktualizace ukazatelů (stromový průchod)
   Procházíme řetěz ukazatelů a každý ukazatel na blok je přestaven na hodnotu, kterou udává scelovací ukazatel tohoto bloku.
4. Přesun bloků(sekv. průchod)
   Každý blok je posunut na adresu, kterou udává jeho scelovací pointer.
5. Odemčení haldy.

6 Generování mezikódu

Podrobně viz skripta strana 147– 171.

Mezikód tvoří přechod mezi frontendem a backendem návrhu kompilátoru. Jde o hranici, na které začíná strojová závislost překladače.

Formy mezikódu:

- tříadresový kód,
- čtyřadresový kód,
- stromová reprezentace, (možná je i reprezentace zásobníkem)

My se budeme dále zabývat tříadresovým nebo čtyřadresovým kódem. Ty mají tvar:
čtveřice: $< opertor, op1, op2, vsledek >$,
trojice: $< opertor, op1, op2 >$, přičemž výsledek je odkazován z pořadí instrukce.

6.1 Zpracování deklarací

Je to takzvaný implicitní kód. Zabývá se plněním tabulek symbolů a definováním aktivačních záznamů procedur (viz tabulky symbolů a aktivační záznamy). Odkazuji na skripta, kde je popsáno několik příkladů gramatiky a mezikódu pro typový analyzátor.

6.2 Vnořování procedur

Týká se jazyků s blokovou strukturou. Úkolem mezikódu je, při zpracování vnořovaných procedur, udržovat informaci o rozsahu v tabulce. Jsou možné dva způsoby:

- čísla rozsahů,
- co rozsah, to tabulka.

Pro práci s tabulkami je nutná minimálně tato sada funkcí:

`procedure entry(table, name, type, offset)` uložení identifikátoru do některé tabulky.

`function mk_table (previous)` zřídí novou tabulku, vrátí pointr na tuto novou tabulku, previous je ukazatel na statisticky nadřazenou tabulku.

`procedure enterproc (table, name, type, offset, newtable)` – do tabulky přidá odkaz na novou tabulku pro proceduru.

Pro každou tabulku potřebujeme zvlášť udržovat offset, proto jsou nutné dva zásobníky:

1. tab_ptr — obsahuje ukazatele na tabulky nadřazených procedur,
2. offset — konkrétní offsety.

Pro tyto zásobníky jsou implementovány běžné zásobníkové operace.

Přiřazovací příkazy a výrazy se zpracovávají pomocí dočasných proměnných. Ty jsou poskytovány funkcí *newtemp*, která vrátí offset do aktuální tabulky na proměnnou příslušného typu. Pro překladač je ideální používat pro každý mezivýsledek dočasnou proměnnou, i když je to náročné na paměť.

Adresování prvků polí bylo rozebíráno v kapitole věnované ukládání prvků polí.

6.3 Backpatching

Během zpracování řídících příkazů nebo při zkráceném vyhodnocování booleovských výrazů je nutno generovat odskoky na kód, který ještě není vygenerován (tedy dopředu). Takové odskoky se řeší pomocí symbolických jmen návěští. Při víceprůchodovém překladu se mohou během jednoho průchodu vygenerovat symbolická jména a při druhém průchodu se těmto jménům přiřadí kokrétní adresa.

Zpracovávame-li však mezikód jednoprůchodově, musíme tyto případy řešit takzvaným backpatchingem, česky snad něco jako „zpětné sešívání“. Metoda spočívá v tom, že necháme adresu návěští, které odkazuje dopředu na zatím nevygenerovaný kód, prázdnou a udržujeme seznam adres, ze kterých se na takovéto návěští odkazuje. V okamžiku, kdy dosáhneme místa obsahujícího definici návěští, zpětně doplníme jeho adresu do všech míst uvedených v seznamu.

Při implementacemi použijeme tuto sadu funkcí:
`makelist(i)` vytvoří nový seznam obsahující pouze index i, vrací ukazatel na takto vytvořený seznam.
`merge(p,q)` spojí seznamy, určené pointry p a q do jednoho a vrací pointr na něj.
`backpatch(p,i)` vloží i jako adresu návěští do všech příkazů obsažených v seznamu určeném pointrem p.

Pro ilustraci odkazuji na skripta od strany 165, kde jsou uvedeny oba příklady použití, jak pro booleovské výrazy, tak pro řídící příkazy.

7 Optimalizace

Optimalizace chápeme jakožto vylepšení programu z hlediska rychlosti výpočtu nebo zmenšení paměťových nároků.

- Optimalizace rozlišujeme na strojově nezávislé (na úrovni mezikódu), a na strojově závislé, které využívají specifika architektury a instrukčního souboru.

  Další dělení optimalizací je na

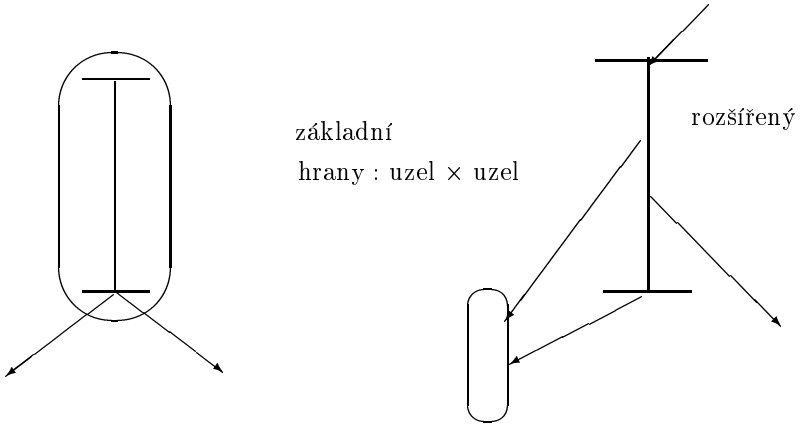
  - ⋆ lokální— na úrovni lineárních úseků kódu
    - ◦ eliminace společných podvýrazů,
    - ◦ odstranění zbytečných přiřazení,
    - ◦ redukce počtu pomocných proměnných.
- globální optimalizace (graf toku řízení G.T.Ř.)
  - ⋆ zbytečné výpočty přes všechny cesty
  - ⋆ detekce co optimalizovat
  - ⋆ tra-ce na ekvivalentní program

  ad detekce)
- konstrukce G.T.Ř. (analýza toku řízení)
- vlastní detekce (např. nalezení společných podvýrazů) (analýza toku dat)

ad tra-ce)
{ omezení se na tra-ce, o kterých je dokázáno, že jsou bezpečné (tj. zachovávají ekvivalenci)} = sada tra-cí zachovávajících ekvivalenci.

7.1 Optimalizace lineárních úseků kódu

lineární úsek kódu = blok



Obrázek 34:

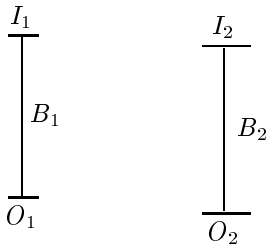
B=(P,I,O)

P = posloupnost příkazů tříadresového kódu

I = množina vstupních proměnných
y=X/C //X,C – vstupní proměnné

O = množina výstupních proměnných (analýza živých proměnných – *LIVE* problém)

? Jsou B_1 a B_2 ekvivalentní:



Obrázek 35:

P=1;. . . n;
Hodnota proměnné v čase i ($0 \leq i \leq n$):
v čase 0:

$$h_0(A) = A \quad if \;\; A \in I \quad A_i \leftarrow \Theta B_{i_1} \ldots B_{i_r}$$

$$h_i(A) =' \Theta' \bullet h_{i-1}(B_{i_1}) \ldots h_{i-1}(B_{i_r})$$

Hodnota bloku se pak rovná množině hodnot výstupních proměnných h(B).

Cena bloku: libovolná f-ce, která bloku přiřadí hodnotu

$$B \to N$$

Optimální blok: blok s minimální cenou, ekvivalentní s daným blokem.

7.1.1 Základní transformace na blocích

:

T1: Eliminace zbytečných příkazů (nedostupné z počátečního příkazu, nebo ty příkazy, které počítají hodnoty, jež se nepoužijí při výpočtu hodnot výstupních proměnných.

Příklad 7.1

C=A+B; D=C×A; C=A-B; D= C×A;

$$\Rightarrow^{T1}$$

C=A-B;D=C×A;
O={D}

T2: Eliminace nadbytečného výpočtu (eliminace společných podvýrazů) C=A+B

C=A+B	
D=A*C	D=A*C
E=A+B	[E=C]
F=E*D	F=C*D

T3: Výměna pořadí příkazů

T4: Přejmenování proměnných
(všechna levostranná jména různá)

Věta 7.1 O optimalizaci (lineárních úseků)
Nechť B je blok a c přiměřené kritérium ceny bloku. Nechť

$$B_1 \text{ je } : B \Rightarrow^{T_1,T_2*} B_1 \text{a nelze to již opakovat}$$

$$B_2 \text{ je } : B_1 \Rightarrow^{T_4} B_2.$$

Pak existujíB_3 a B_4 takové , že

$$B_2 \Leftrightarrow^{T_3*} B_3 \text{ a } B_3 \Leftrightarrow^{T_4*} B_4$$

a B_4 je optimální k B vzhledem k c.

Poznámka 7.2 Minimalizace LOAD a STORE

Algoritmus 7.1 Procedure OPT

1. L=0;
2. Nechť S je první příkaz takový, že všechny příkazy, které používají levostrannou proměnnou z S, jsou už v seznamu. Přidej S do L (neexistuje-li END).
3. Nechť S je poslední příkaz přidaný do L. Jestliže 1. operand $\notin I$ a S_1 je první příkaz, který má na L- side tento operand, pak $L+ = S_1$, opakuj 3, jinak goto 2.

L je reverse optimálního bloku.

7.2 Programy s cykly

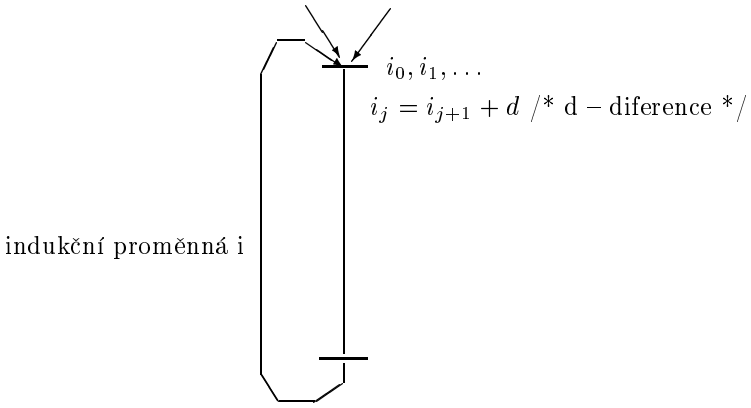
1. transformacece,
2. detekce, kdy tra-ce aplikovat (analýza toku dat ← analýza toku řízení).

ad 1. použití tra-cí T1,T2:

- Nahrazení výpočtu v době běhu výpočtem v době kompilace, např. (2*PI*...) (šíření konstant, detekce konstant, a následné dosazení).
- Rozvinutí cyklů – ušetří se příkazy skoků.
- Slučování cyklů — např. při dosazování hodnot do matic stejných rozměrů.
- Čištění cyklů (přesun kódu — „vytýkání“ kódu před začátek cyklu)

```
        beg:  A+B+I^43          T1=A+B
Př.:          goto beg    beg:  T1+i
                                goto beg
```

- Indukční proměnné (váží se k cyklům majícím jeden vstup):
  - ⋆ eliminace indukčních proměnných,
  - ⋆ redukce složitosti operátorů u indukčních proměnných.
- Inline procedury.



Obrázek 36:

díky lineární závislosti lze eliminovat některé indukční proměnné

redukce složitosti:

```
                I=1;                    I=1;
          A:    T=I*5;                  T=5;
                B(I)=C(T);  ⇒     A:    B(I)=C(T);
                I=I+1;                  T=T+10;
                goto A                  I=I+1;
                                        goto A
```

ad 2. Detekce:

1. Analýza toku řízení→ Graf toku řízení
   !detekce možnosti T1,T2!
   ⇓
   Analýza toku řízení

detekce T1: – problém živých proměnných nebo také **LIVE problem**

Proměnná i je živá v uzlu u (G.T.Ř.) jestliže existuje cesta z u do nějakého uzlu v taková, že hodnota i spočtená ($\exists zptn$ uk. v G.T.Ř.) v u je ve v použita.

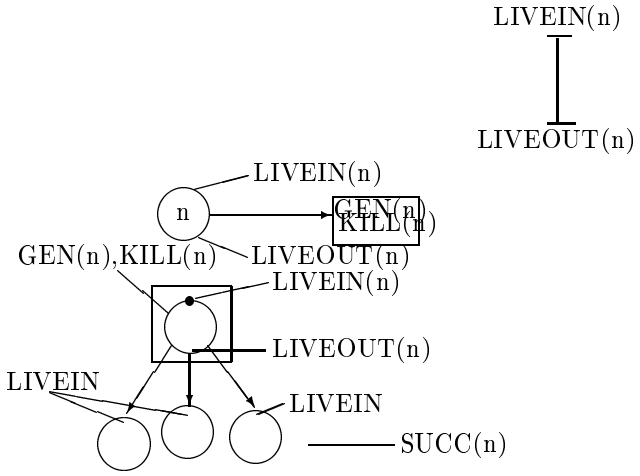
Lokální informace použitá v uzlu Grafu toku řízení. Proměnná je v uzlu

– definována KILL(N),

– referencována GEN(N),

– nic.

Graf toku řízení je reprezentace vnitřního tvaru programu. Je to orientovaný graf, jehož vrcholy příslušejí posloupnostem příkazů, které se nazývají základní bloky. Hrany grafu toku řízení pak reprezentují relaci následnosti z hlediska potencionálního přenosu řízení mezi základními bloky. Základní blok je posloupnost po sobě následujících příkazů, jejich provádění může začít pouze prvním příkazem a neobsahuje, mimo posledního příkazu, příkaz větvení, skoku nebo zastavení výpočtu. Odkazuji na skripta str. 173.



Obrázek 37:

$$LIVEIN(N) = LIVEOUT(N) \cap \neg KILL(N) \cup GEN(N)$$

$$LIVEOUT(N) = \cup_{m \in SUCC(N)} LIVEIN(m)$$

$$LIVEOUT(N) = \cup_{m \in SUCC(N)} (LIVEOUT(m) \cap \neg KILL(m) \cup GEN(m)$$

$X_i = f(x)$ – lze algoritmizovat (řešení= pevný bod f)

1. 1...n postorder
2. pracovní seznam

$O(n^4)$

$\forall$ – dopředný (AVAIL – problém dostupných výrazů– T2)

GEN(n) – výrazy, jejichž hodnota je spočtena

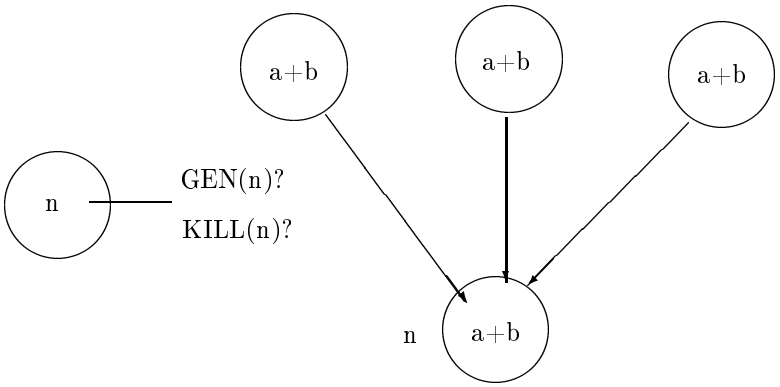
KILL(n) – výrazy obsahující proměnnou, jejíž hodnota se změnila

$$AVAILOUT(n) = AVAILIN(n) \cap \neg KILL(n) \cup GEN(n)$$

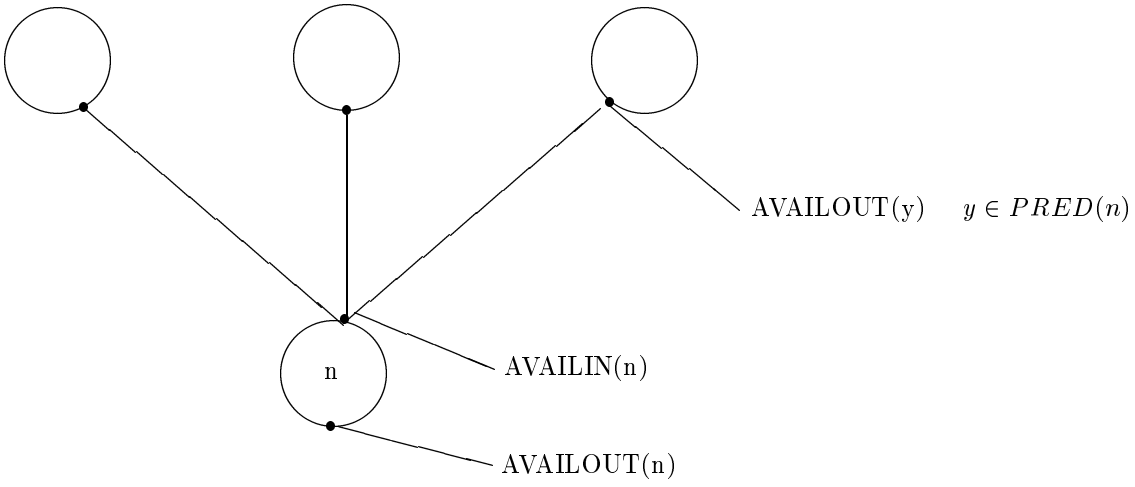
$$AVAILIN(n) = \cap_{x \in PRED(n)} AVAILOUT(x)$$

řešení– maximální pevný bod

inicializace :



Obrázek 38:



Obrázek 39:

- AVAILOUT=GEN
- AVAILIN(vstup_uzlu)=0

8 Generování cílového kódu

viz. skripta 207 – konec