

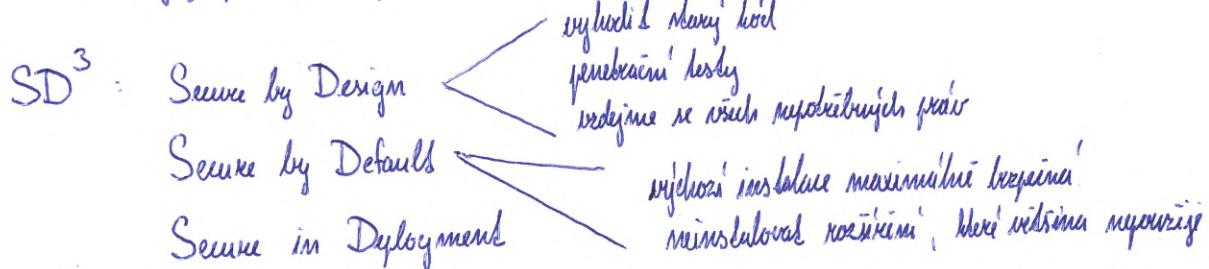
Základní bezpečnostní principy. Modelování bezpečnostních rizik, metodiky STRIDE a DREAD.

bezpečnostní
hrozba - pravina nežádoucího incideantu (proběhne autentizace hrubou silou)

závratitelnost - slabí místo, kde může být využití hrozby / hrozbami

Kvalita software - bezpečný & spolehlivý
 ↓
 chráni důvěryhodnost, integritu a dostupnost dat

- bezpečnost software se musí řešit už v návrhové fázi
 - aby byly operativní všechny autority!



datší pravidla: myslit jako útočník, minimalizace počtu uživatelů, microservices bezpečnost externích systémů jen zabezpečit, bezpečit sebe sami!
 ne security through obscurity

STRIDE: (model pro identifikaci bezpečnostních rizik)

Spoofting of Identity - podvádění identity

Tampering with data - neautorizovaná změna dat

Repudiation - popření transakce nežádoucím

Information disclosure - unik informací

Denial of service

Elevation of privilege - neautorizované využití oprávnění

možné řešení:

- logická autentizace, multifaktorová autentizace, ...
- autentizace, hashing, MAC, ...
- digitální podpis, časové dleky, ...
- autentizace, šifrování, ...
- filtrace, throttling, QoS, ...
- oprávnění uživatelských

modelování hrozeb

○	◦ proces	- jednotka spracování dat	[S,T,I,D,E]
=	◦ uživatel		[T,R,I,D]
---	◦ hranice	- počítač, adresní prostor, hranice divizny	
□	◦ interakce	- vstup do systému	[S,R]
→	◦ dok dat		[T,D,I,D]

- hodnocení hrozib = DREAD

- průměrná hodnoty z pěti kategorií, hrozba [0,10]

- Damage potential
- Reproducibility - jak snadno je kopirovat
- Exploitability - jak snadno je bítka
- Affected users - množství postižených uživatelů
- Discoverability

- vyplývají: CVSS ~~vs. CWSS~~ - více méně, méně hodnot (šířka)

CWSS - množství a komplikace nejčí

Přečtení bufferu, jeho chyby. Metody obrany, způsoby překonání obran.

- přípsání jednoho nebo více bytů za koncem bufferu
 - přípis návratové hodnoty → stack smashing (stack)
- přečtení na heap také nebezpečné!
 - můžeme přepsat informace aloacní funkce

Krátký:

DEP - Data execution prevention

- NX bit v poloze tabulky stránek
- k přečtení může dojít, ale daná stránka nelze spustit
- stačí se vymazat na bázi pomocí API
- return oriented programming (exploit)

ASLR - Address space layout randomization

- program umístí na náhodnou adresu
- (exploit) získá adresy pomocí nebezpečného printf()

kanárek - náhodně šroubované hodnoty (stack guard)

Safe stack/weird exception handling

- vyjímka se kontrolouje proti sevšem ručněkých handlerech
^(handles)
- pokud není nalezen, oharmály konec

(+ exploit context switch)

neporušit gets, ~~memp~~, strcpy, ...

Úrovně bezpečnostních oprávnění. Běh programu při nízkých oprávněních.

ACL - access control list

- obsahuje ACE (entry) - kód identifikující entitu (SID) ^a píšupová práva
- securable objects (kód objektu, který ne dle' zabezpečit)
- jen některé FS mají ACL = NTFS
- předí právidel je důležité!
 - jde se od objektu do průtahu, rámci kontrole kontroloují první

POSIX: getfacl, setfacl

- rozšíření standardního modelu práv

user: name: rwx

group: name: rwx

Access Token - popisuje bezpečnostní kontext procesu nebo vlastna

- každý uživatel má po přihlášení svůj token → jím spuštěné procesy mají svůj primární token
- všechny se spouštějí s touto hlavním tokenem
- dle' se vyměnit i spuštění s jiným tokenem (mají ho také služby OS)
- omezení tokenem → omezení některých práv
- OS používá SID uložené v tokenu při spuštění procesu

- každý program má dělat jen to, co má a nic méně → vzdál se ostatních práv
- pokud potřebujeme větší oprávnění, musíme ji spustit v procesu vedle
- výhradně kód napřednost

při píšupu vyšší práva:

- kvůli ALL: zapis na mikro, kam může pouze administrátor, zapis do registru HKEY_LOCAL_MACHINE
→ výběr mikro kam mohou zapisovat všichni (i registrov), rozvolnění ACL
- kvůli právem: → lze píšup uživatele do sloužícího než ho dělat administrátorem

- hledí LSA revokace (Local System Authority)
 - variabilní musí být ve skupině administrátorů
 - skupině lze povolit používání
- když ACE by mělo být vlivodárné → ostatní odstranit
- registrace v seřadišti používání a že všechny determinují úroveň přístupu používajou
- může mít běhu
- registrovat všechna SID v kontejneru a odstranit nespořitelné
 - smazání kontejneru

UAC - User Account Control

- pro kancelářské variabily je při přidávání vykresován token
- pro administrátory obvykle se restrikčními právy - když použijí explorer.exe
 - možnost elevace na administrátora

prompt for consent / prompt for credentials (consent.exe)

Application Information service - spouští consent.exe a kontroluje upřednostnění

- kontrolouje co je soubor načít
- + zlepšuje práva automaticky pro některé Microsoft soubory
- spouští prompt v reakci na některé akce
- pokud je soubor nečten, je do procesu naložený funkčnostní token a je povolen

Virtualizace pro starší aplikace

- povolit některé věci ve vlastním virtuálním prostředku

Bezpečnost databází. Útoky typu SQL Injection a obrana proti nim.

- všechny vstupy musí být prověřené
- pokud dodržujeme těch závažnějších opatření, můžeme problémům předejít
 - ne ale úniky (information disclosure - SELECT)
- je třeba sanitizovat vstupy
- mysql_query a mysqli_query umožňují pouze jednu query, pokud se ale dojde oběma

$$\text{DELETE FROM } \times \text{ WHERE jmeno = '1' OR '1'='1';}$$

- ! - o mediárech větší opatrností nelze vědět
- o důkladné kontrole vstupů - pozor hlavně na cípostroby
 - mysql_real_escape_string - PHP funkce pro escapování stringu
 - o používání placeholderů (prepared statements)
 - na místo parametru placeholdery, které se propíšou DB systému
 - mysql_real_escape_string není potřeba
 - + riskováme lepší efektivitu u spákování vykonávaných query
 - data v placeholderech může nahrazovat i databáse samotnou (DB procedure)
 - [QUOTENAME - bezpečné escapování]

Specializované nástroje

- SQL firewall - detekce pohamu & injection
- může vykazovat seznam povolených struktur

Nástroje testování vlastností

- [sqlmap]

Algoritmy přesného a přibližného vyhledávání.

Exact matching

- ▷ naivní (brute force) algoritmus - $O(mn)$ time
- ▷ Karp-Rabin - hledání pomocí hashovací funkce - $O(mn)$ time
- vypočítám hash vstupu (hledaného)
 - porovnávám se rolling hash → jednoduše srovnáním
 - pokud uželám stejně se všem substringy dílky m
 - můžu mít i false positives - musím ještě skontrolovat další element

příklad: base = 5 mod = 10

$$P = aba \rightarrow 0 \cdot 5^2 + 1 \cdot 5^1 + 0 \cdot 5^0 = 5$$

další element sčítám jako $(\text{prev} - x \cdot 5^2) \cdot 5 + y \cdot 5^0$

- ▷ Morris-Pratt algorithm - $O(n)$ time, $O(m)$ space

$$\beta'[i] = \beta[i-1] + 1$$

- reálnu neakceptuj symboly - v případě, že se neakceptuje, rukoum na hodnotu $\beta'[i]$
- hodnota říka "koliký symbol musí být po rukoum pod současným"

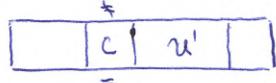
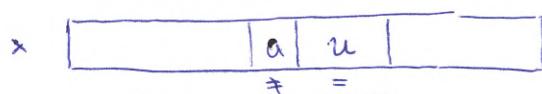
- ▷ Knuth-Morris-Pratt algorithmus

- vykreslím efektivnější rukoumávání'

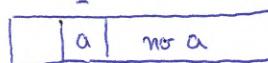
$\beta''[i]$ - ~~KMP~~ KMP funkce

$$\beta''[j] = j' , j'-1 \text{ je délka nejdélsího borderu } P[1 \dots j-1] \text{ kde je } P[j'] \neq P[j]$$

- ▷ Boyer-Moore algorithm



→ good suffix shift

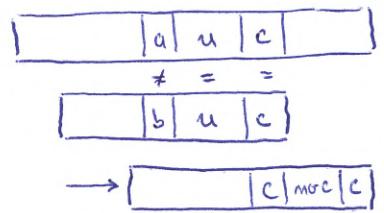


→ bad character shift

Boyer-Moore - Horzepool

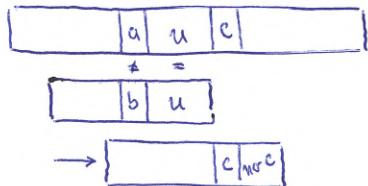
$O(nm)$

- BCS alle posledního symbolu daného patternu



Boyer - Moore - Sunday

- nejd symbol in text



[Zho - Tarhacha - používá dva symboly nášlo jehoho]

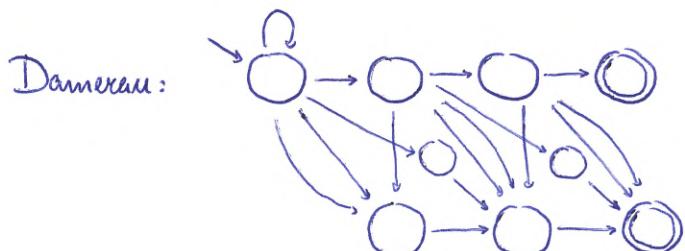
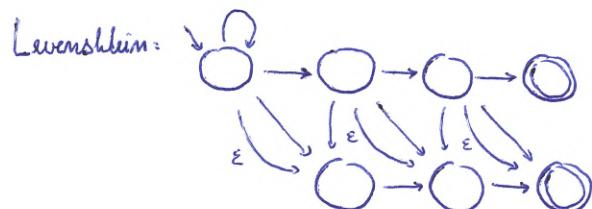
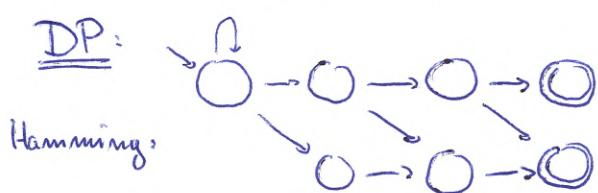
Approximate pattern Matching

- Edit distance - vzdálenost mezi dvěma řetězci

- Hamming - pouze operace replace
- Levenshtein (edit distance) - replace, delete, insert
 - dynamické programování

D	-	a	c	g	a	t
-	0	1	2	3	4	5
a	1	0 → ^m 1 → ^r 2 → ^m 3 → ^r 4 → ^m 5				
g	2	1 → ^d 1 → ^r 1 → ^m 2 → ^r 3 → ^m 3				
a	3	2 → ^d 2 → ^r 2 → ^m 1 → ^r 2 → ^m 2				
c	4	3 → ^d 2 → ^r 3 → ^m 2 → ^r 2 → ^m 2				
t	5	4 → ^d 3 → ^r 3 → ^m 3 → ^r 2 → ^m 2				(2)

- Weighted distance
- Needleman - Wunsch algorithm - rovnoměrná relevantní bioinformatické
 - ceny operací jsou stejné libovolnou - křížové řešení křížové dráhy'
 - + gap penalty - cena za mezery (operace insert a delete)
 - cena se maximizuje
- approximate string matching - hledání slova v řetězci se vzdálostí méně než k



- nové operace transpoze

[bit parallelism]

i 1 2 3 4 5 6 7 8 9 10 11 12 13 14

$p[i]$ a b a a b a b a a b a a b a

$\beta[i]$ 0 0 1 1 2 3 2 3 4 5 6 4 5 6

$\beta'[i]$ 0 1 1 2 2 3 4 3 4 5 6 7 5 6 7

$\beta''[i]$ 0 1 0 2 1 0 4 0 2 1 0 7 1 0 7

- KMP - hledáme na pozici $\beta'[i] = a \rightarrow$ pokud $p[a] \neq p[i] \rightarrow \beta''[i] = a$
- \rightarrow pokud $p[a] = p[i] \rightarrow \beta''[i] = \beta'[a]$

BMH:

$p = abaabaa$
6 5 4 3 2 1 0

C	BCS[c]
a	1
b	2
c	7

minimum je 1

a b c a b a @ b a @ b a a a

+ " " " "

a b a a b a a

a b a a b a a *

✓ a b a a b a a "

a b a a b a a *

✓ a b a a b a a "

Úplné indexování textu.

Lyndon decomposition

- primární slovo & osobi nejménší lexicografické rozložení

c b c b b a a b a a c a a c a - | c | b c | b | b | a a b a a c | a a a c | a

LZ-decomposition - používáno v LZ komprezii

Border Array - obsahuje délku nejdelšího borderu na každé pozici

- počet se dále rozšiřuje, stanou se $i = \beta$ (current)

→ počet menající pokračování, zapíšu 0

$$[\beta[i] = 0]$$

Suffix array - nové symboly $\$ \in \Sigma \subset \#$

string $T = \# a b a a b a b \$$

i	-1	0	1	2	3	4	5	6	7
T[i]	#	a	b	a	a	b	a	b	\$
SA[i]	7	2	5	0	3	6	1	4	-1
LCP[i]	0	0	1	2	3	0	1	2	0

starting pozice lexicograficky seřazených suffixů
nejdelší společné prefixy

LCP Search $x = aba$ - hledaný string

algoritmus: $d = -1$ $i = \lfloor (d+f)/2 \rfloor = 3$ - původní pole
 $f = 7$

$$L_3 = T[SA[3] \dots 13] = T[3 \dots 6] = abab$$

$$l = LCP(L_3, x) = 3 \rightarrow \text{match} - \text{nášti jiné}$$

- přes LCP hledám pozici 3 mohu najít ostatní výskyt

- musí být $\geq \text{len}(x)$

- pokud bych nematchoval, porovnávám $L_3[1] \neq x[1]$

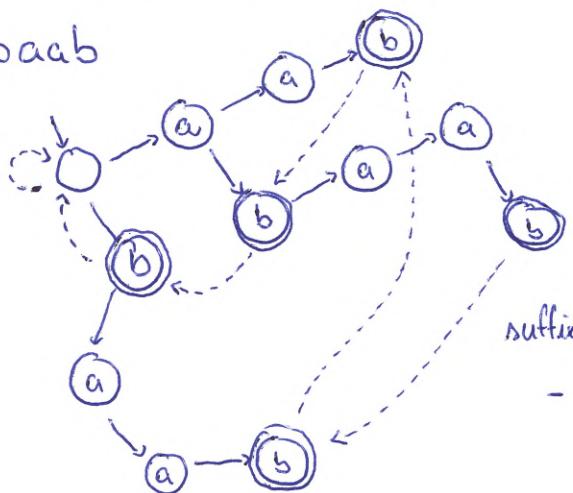
→ pokud je $x[1]$ menší, pokračuj v nížešší řadě ($f = i$)

→ v opačném případě se myšlím ($d = i$)

Suffix Tree

- strom všech možných suffixů
- stav pro každý možný morfismus, tabulka hravou všechny jednotlivé symboly

$T = abacab$



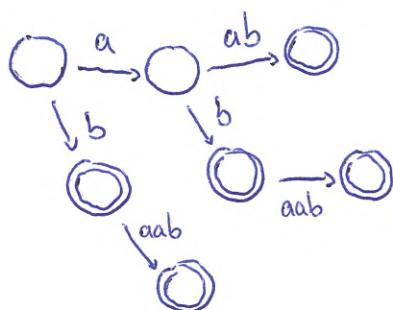
- uzel s několika vývěrami: fork

- koncové uzel vždy má koncového suffixu

suffixové linky

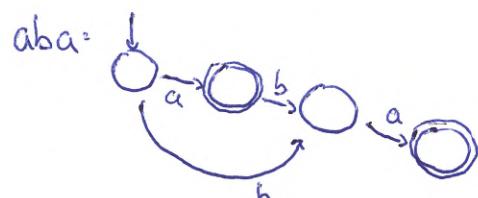
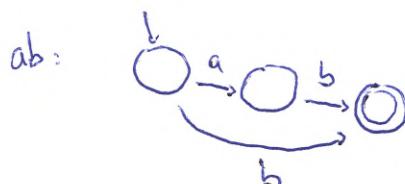
- mizí do učlu se suffixem o 1 krokům

Suffix Tree - strom s montinal nodes of degree 1 deleted

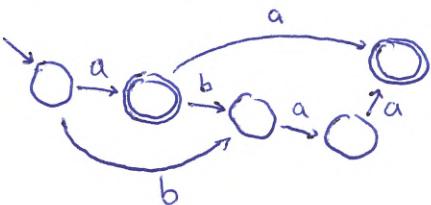


Suffix automaton

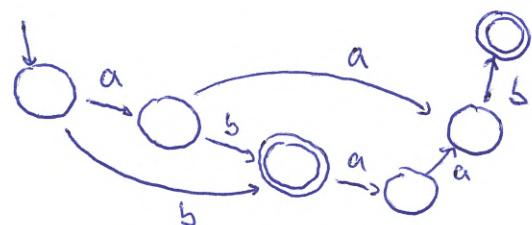
- minimální automatas přijímající všechny suffixy



abaa:



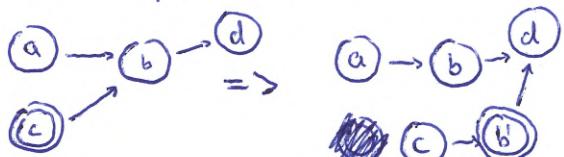
abaab:



- v každé vývěrce „vykouzlí koncový stav“ ze počátečního stavu

- pokud se sjede z každé vývěrce koncový stav \rightarrow de

- v opačném případě ji ruší fork



Compact suffix automaton
[implementace]

Succinct data structures.

Rank & Select

- ▷ rank(i) - #1 upo position i občas nutné $O(1)$
- ▷ select(i) - i-th #1 (position)

Reprezentace binárních stromů (posouze jeho vrátky)

- každým vrátkem máme polomky
- význam binárního stromu je, že první vrátek je 1, následující 0
→ indexují se vrátky po následkách

$$\text{left_child} = 2 \cdot \text{rank}(i)$$

$$\text{right_child}(i) = 2 \cdot \text{rank}(i) + 1$$

$$\text{parent}(i) = \text{select}\left(\left\lfloor \frac{i}{2} \right\rfloor\right)$$

- reprezentace má $2n+1$ bitů

general trees - každý vrátek reprezentuje $1^m 0$ kde m je počet polomku

Data structure - who knows exactly

- several levels (constant) of indirect access
- logarithmic blocks with superblocks and precalculated table

Wavelet Trees

- rozšíření rank & select na větší abecedu
 - více vektorů v jednom stromu výšky $O(\lg |\Sigma|)$
 - rank shora dolů - vždy používám následk jeho volby další query
 - select směrem zpět nahoru
- compressed suffix arrays
- každý level posuvina pruhu předchozího
 - srdce funkce SA_k budou řádky i v levelu $k+1$
 - lze nazvat neighborhood funkci Φ takovou, že $SA_k[\Phi_k(i)] = 1$
(i : $\Phi(i)$ obsahuje index pruhu jehož $SA = SA[i] + 1$)

Burrows - Wheeler & FM-Index

$$T = \text{abracadabra} \quad P = \text{bdabrab}$$

i	1	2	3	4	5	6	7	8	9	10	11
T[i]	a	b	r	a	c	a	d	a	b	r	a
SA	11	8	1	4	6	9	2	5	7	10	3
F	a	a	a	a	b	b	c	d	r	r	~T[SA[i]]
SA'	10	7	11	3	5	8	1	4	6	9	2
L	r	d	a	r	c	a	a	a	b	b	~T[SA'[i]] = BWT(T)

= první symboly lexicograficky seřazených řetězů

= poslední symboly lexicograficky seřazených řetězů

C	a	b	c	d	r	- kumulativní frekvence symbolů (první výskyt daného symbolu) vs F
	0	5	7	8	9	

FM-Index:

$$\text{Occ}(x, c, i) - \# počet výskytů c v prefixu X[1 \dots i]$$

$$Sp_i = C[c] + \text{Occ}(L, c, Sp_{i-1} - 1) + 1$$

$$Ep_i = C[c] + \text{Occ}(L, c, Ep_{i-1})$$

$$\text{Occ}_i = Ep_i - Sp_i + 1 - \# počet výskytů symbolů (družstva řetězce)$$

interval $[Sp_i, Ep_i]$ řídí index v SA → pozice ve řetězci

příklad: $Sp_0 = 1$

$Ep_0 = 11$

$\text{Occ}_0 = 11 - 1 + 1 = 11$ výskytů prvního řetězce

- pattern ještě odráží! $\rightarrow c = 2$

$$Sp_1 = 0 + 0 + 1 = 1 \quad \left. \right\} \text{interval } [1, 5]$$

$$Ep_1 = 0 + 5 = 5 \quad \left. \right\}$$

$c = r$

$$Sp_2 = 9 + 0 + 1 = 10 \quad \left. \right\} \text{interval } [10, 11]$$

$$Ep_2 = 9 + 2 = 11 \quad \left. \right\}$$

Lambda kalkul: definice pojmu, operaci, reprezentace čísel

Lambda calculus - ekvivalentní s Turingovým strojem

- nejmenší univerzální programovací jazyk na světě

$\langle \text{expr} \rangle ::= \langle \text{name} \rangle \mid \langle \text{function} \rangle \mid \langle \text{application} \rangle$

$\langle \text{expr} \rangle ::= (\langle \text{expr} \rangle)$ - libovolný výraz může být uvozovat

$\langle \text{name} \rangle ::= \text{constant} \mid \text{variable}$

$\langle \text{function} \rangle ::= \lambda \langle \text{name} \rangle : \langle \text{expr} \rangle$ jedinečné slovo

$\langle \text{application} \rangle ::= \langle \text{expr} \rangle \langle \text{expr} \rangle$

příklady: $\lambda x.x$ - identity function

$(\lambda x.x)y$ - aplikace funkce na proměnnou y

→ všechny x budou nahrazena hodnotou y (bude redukovat)

$$(\lambda x. + x 1) 4 \rightarrow (+ 4 1) \rightarrow 5$$

$$(\lambda x. + x x) 3 \rightarrow (+ 3 3) \rightarrow 6 \quad (\text{argumenty se nemusí vystýkat v řadě})$$

- argumentem mohou být jiné funkce

$$(\lambda f. f 4)(\lambda x. + x 1) \rightarrow (\lambda x. + x 1) 4 \rightarrow 5$$

$(\lambda x. + x y)$ free variable
bound variable

- při substituci nahrazení pouze volné v E
- v párce stejně pojmenované mohou mít problem
- α -redukcí = přejmenování všech proměnnel ve funkci

repräsentative ēisel: - formel funkt'

$$0 \equiv \lambda s z . \underbrace{z}_{\text{drei weitere Argumente}}$$

$$1 \equiv \lambda s z . s(z) \approx \text{suc(zero)}$$

$$2 \equiv \lambda s z . s(s(z)) \approx \text{suc}(\text{suc(zero)})$$

$$3 \equiv \lambda s z . s(s(s(z)))$$

→ successor function: $S \equiv \lambda x y x . y(w y x)$

$$\lambda y x . y((\lambda s z . z)y x) \rightarrow \lambda y x . y((\lambda z . z)x) \rightarrow \lambda y x . y(x) = 1$$

The Y combinator: funktion pro rekurri

$$\boxed{\lambda f . (\lambda x . f(x, x))(\lambda x . f(x, x))}$$

$$\text{TRUE: } \lambda x y . x \quad \text{FALSE: } \lambda x y . y \quad \text{NOT: } \lambda a . a (\lambda b c . c)(\lambda d e . d)$$

$$\text{AND: } \lambda a b . a b (\lambda x y . y) \rightarrow \lambda a b . a b \text{ FALSE}$$

$$\text{OR: } \lambda a b . a (\lambda x y . x) b \rightarrow \lambda a b . b \text{ TRUE } \cancel{y b}$$

$$\text{IS-ZERO } n : \lambda n . n \text{ FALSE NOT FALSE}$$

$$\text{GREATER-OR-EQUAL } n m : \lambda n m . \text{ IS-ZERO } (n P m)$$

↑ predecessor function

Δürkholz: verwandelt successor funktion a in a+b

$$3+5 \equiv 3 \text{ succ } 5$$

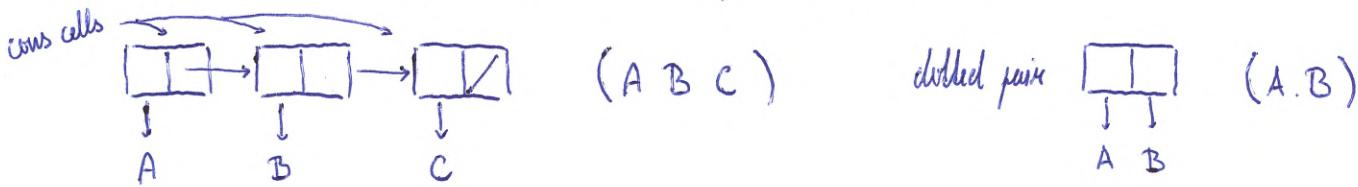
$$\underline{\text{maisobrem}}: \lambda a b c . a(b c)$$

Lisp: atomy, seznamy, funkce, cons buňky, rekurze, iteracní cykly, mapovací funkcionality, proměnné, strukturované datové typy, makra, realizace ne determinismu

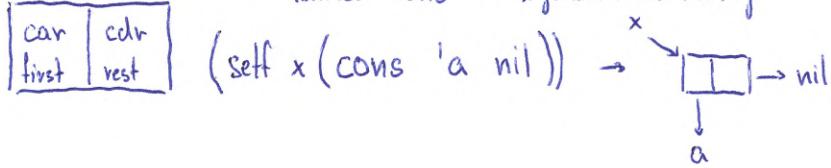
- všechno je atom nebo list

(čísla, proměnné, konstanty, stringy, ...)

- listy jsou reprezentovány rekurzivně - prázdný list () = NIL



- funkce cons - vytváří nové buňky



funkce v Lispu: (defun name (parameters) body)

(defun square (x) (* x x))

- všechny argumenty jsou evaluovány před zadáním funkce

→ funkce quote nebo ' konstanty'

funkcionality:

(cond (p1, e1)	(if predicate
(p2, e2)	consequent
...	alternative)
(pn, en))	

predikity:

(null s)	- T pro prázdný list	(length s), (nth s), (last s)
(number s)		(butlast s)
(listp s)	- T pro list s	

rekurze: (defun Fib (n)
 (if (< n 2) n
 (+ (Fib (- n 1)) (Fib (- n 2))))
))

- koncová rekurze - poslední operaci se funkcií
- výborná k počítači implementace → nového dletožného stack frame

(defun factorial (N)	(defun factorial (N)
(if (= N 0) 1	⇒ (factorial-aux N 1))
(* N (factorial (- N 1))))	(defun factorial-aux (N Acc)
	(if (= N 0) Acc (factorial-aux (- N 1) (* N Acc))))

(defvar *global* 36) - globální proměnné
 (defconstant big 100) - konstanty

arrays a vectors (vector 1 2 3)
 (make-array 5 :initial-element nil)

sequences (length '(1 2 3)) → 3 (reverse '(1 2 3)) → (3 2 1)
 (subseq '(1 2 3 4 5 6 7) 1 5) → (2 3 4 5)

copy-seq - kopíruje referenci (pole, vektory slouží)

(concatenate a b)

Hash tables

make-hash-table

(gethash 'foo *h*) = h['foo']
 (setf (gethash 'foo *h*) 'abcd) = h['foo'] = abcd

maphash

Stack - listy podporují push a pop operace

(push 'a x)

(pop x) → (*a)

- stejně tak mohu využít Set a Tree struktury

Mapovací funkcionality (funkce s funkcí jako argumentem)

(mapcar function list1 list2 ... listn)

- iteruje přes každý list a na všechny prvky volá danou funkci

- všechny elementy na i-tých pozicích jsou reprezentovány majetkovou

- končí když jeden list dojde (mapcar #'list '(1 2) '(5 8 10) '(9 9 9))

mapcan - spojuje výsledky do jediného listu → (((1 5 9) (2 8 9)))
→ (1 5 9 2 8 9)

maplist - pracuje na sub listech → (((((1 2) (5 8 10) (9 9 9)) ((2) (8 10) (9 9))))

mapc / mapcon / mapl - prací pouze list (list 1)

Iterace (dolist (x '(1 2 3)) (print x))

(dotimes (i 4) (print i))

(do ((i 0 (+ i))) (<= i 4)) (print i))

(loop body-form*)

Makros - makra jsou komplikovaná a expandovana'

→ makro expansion ~~at time~~ - v když ještě není přesunut k deklaraci

- jsou napsaná v Lispu

- mnoho implicitních operátorů jsou makros (looping, when, unless, ...)

(defmacro unless (condition &rest body)

~ (if (not condition) (progn , @body)))

debugování makr: macroexpand , macroexpand - n (expanduj)

- lze využít obou, jež leží funkce
 - (defun +1 (+ 1 x))
 - (defmacro +1 (x) `(+ 1 , x))
 - oběma musíme poslat argumenty
 - necheme expandovat parametry
→ chceme dynamicky vykračit funkce s danými argumenty
-
- makro se vykonává před vnitřním, může ale používat všechny funkce (i násé) → musí ji být:

(defmacro xreverse (x) (reverse x))

(xreverse (3 8 ~)) → 5

(choose '(1 2 3)) - random výběr

Functional programming: - využívá techniky, neměníme stav (pure functions)

- nemění řádně nějaké efekty
- neměníme původní vstupní argumenty funkce
- Lisp obsahuje i desstrukční funkce

Prolog: fakta, pravidla, dotoazy, aritmetika, způsob vyhodnocení dotazů, unifikace, řízení výpočtu, operator řízu

- postavení na first-order predicate logice
- program je postaven z Hornových klauzulí (axiomy)

facts, rules, queries

fakta: male (josef).

parent (josef, jan).

pravidla: mother (X,Y) :- parent (X,Y), female (X).

predecesor (X,Y) :- parent (X,Y).

predecesor (X,Y) :- parent (X,Z), predecesor (Z,Y).

→ rekurencií vyjádření vztahu

dotazy: ?- parent (josef, sarka). → yes

?- predecesor (X, sarka). → X = josef X = jana no

všechno jsou funkce: atomy, čísla, proměnné, struktury (funkce a list argumentů)

- atomy vždy lze rozepsat, proměnné first appearace

- Prolog většinou funguje DFS, proběh k domu může vyvážit stack (může ale i BFS)

→ prohlíží všechny dle kontextu operátorem řízu

Unifikace

- konstanta pouze se sebou
- proměnná s činnulicí, ale vymění ekvivalenci
- struktury jedna je funkce stejný (vždy počtu argumentů) & argumenty sedí

- na počtu argumentů záleží → lehčí by měly být proměnné

- řízku přečti může odpovídat nekoncovou rekureci

path (X,Y) :- path (X,Z), edge (Z,Y)

Aritmetika - půjčené pomocí is

, - konjunkce ; - disjunkce

fact (0,1).

fact (N,F) :- N > 0, N1 is N-1, fact (N1,F1), F is N * F1

Lists - [] - prázdný list

[1,2,3] - pevný list

| - separátor meziho prováděním a zbytek listu

[a | [b,c]]

member (Elm, [Elm | Rest]).

member (Elm, [X | Rest]) :- member (Elm, Rest)

append ([], F, F)

append ([A | L1], L2, [A | Rest]) :- append (L1, L2, Rest)

+ X, - Y, ?? Z - volání, využití a obj. argument

Operator nereg !

- všechny uspejí, neprovádí backtracking

- kamerají využití ostatních pravidel

marada (jana, X) :- plesat (X), !, fail

marada (jana, X) :- muz (X)

built-in predicates (some)

true, fail, not (Q)

integer (X), var (X), nonvar (X)

factorial example :

factorial (0,1).

factorial (N,F) :- N > 0,

N1 is N - 1,

factorial (N1,F1),

F is N * F1.

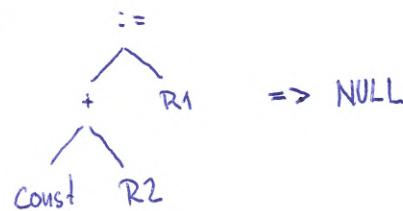
Generování cílového kódu poluvýčním syntaktického stromu a syntaxiřením překladem.

- generování optimálního kódu je NP-hard problem
 - + náviní na architekturu a jejich vlastnostech
- Naive approach - 1:N - N instrukcí je vygenerováno pro každou instrukci IR
 - simple and inefficient

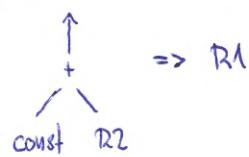
Conventional approach M:N → must use heuristic

- ↓
- IR is an AST - each target instruction has a corresponding tree pattern
 - we match parts of the tree until it is removed (reduced to Null)

st const (R2), R1



LD const (R2), R1



ADD R2, R1



- sometimes we can match such that it is not possible to reduce the tree
 - backtracking and new try
- try to use the cheapest patterns to cover the tree

Graham - Glaviville

- grammar rule for each machine instruction → LR(0) parser is constructed
- reductions by rule → given instruction is selected
- grammar is generally ~~ambiguous~~ and contains conflicts → heuristic

grammar is constructed: (prefix notation)

$R_2 \rightarrow := R_1 \quad \{ LD\ R_2, R_1 \}$

$R_1 \rightarrow + \text{ const } R_1 \quad \{ ADD \# \text{const } R_1 \}$

$\text{Null} \rightarrow := \text{const } R_2 \ R_1 \quad \{ ST \text{ const } (R_2), R_1 \}$

→ construct LR(0) parser

* of

→ handle conflicts such that minimum \checkmark instructions will be generated

- shift-reduce conflict → shift is preferred

- reduce-reduce conflict → biggest right-hand side is preferred

Dynamic programming - total cost of all patterns is to be minimal one

- produces optimal tiling (good for CISC CPUs)

→ bottom-up traversal - calculate prices of each subtree that can be matched

→ 2-traversal - generate target code → go all the way to root

Maximal match method

- tree top-down method

- tree pattern for each machine instruction → sort largest first

→ from root, cover whole tree (subtrees)

- good for RISC CPUs, not optimal tiling

SSA & Phi function

Single static assignment - each variable is always assigned just once

- Phi function - merger of values from different branches

Alokace registrů v generátorech kódu. Lokální optimalizace generovaného kódu v rámci základního bloku a v rámci funkce.

Register allocation

- registers are much faster than memory
→ compiler uses huge amount of virtual registers → those must be mapped
- solution using graph coloring (NP-hard problem)
- three states of temporary variables:
 - ▷ unallocated - not assigned yet
 - ▷ live - allocated and will be used in the future
 - ▷ dead
- two variables can share register only if they are not alive at the same moment

Register interference graph

- node for each variable - edge between those alive simultaneously
- create from the sets of living at each point of the program
- assign colors to registers → heuristic solution
 - k -colorable graph - can be colored with k colors
 - remove nodes with less than k neighbors
 - go back through stack and assign colors
- if we don't have enough space → register spilling (save into memory with load and store)
 - recompute the interference graph
 - try again
- try to spill temporaries with the most conflicts & try to avoid spilling in inner loops

Linear scan reallocation - simpler, faster, used by JITs (VM)

- based on live ranges of variables
- all variables in registers → if one must be spilled, choose the one with longest live range
- reserved registers for spilled variables

Optimizations

- performed over various levels of IR
 - ▷ local - inside BB [machine dependent × independent]
 - ▷ global - whole procedure
 - ▷ interprocedural - whole program
- " premature optimization is the root of all evil "
- 80% of execution time is spent executing 20% of the code
- good algorithms FTW

General optimization techniques

- ▷ strength reduction - e.g. shifts instead of multiplication
- ▷ common sub-expression elimination - compute common things just once
- ▷ constant propagation & constant folding
- ▷ dead code elimination
- ▷ code motion - move invariants from loops above them
- ▷ loop unrolling - do more steps within one iteration (with bigger steps)
- ▷ algebraic identities
- ▷ elimination of useless instructions
- + peephole optimization techniques

Generování kódu: HW závislé optimalizace, optimalizace pro instrukční pipelining

Pipelined processors

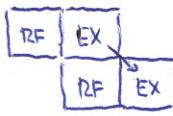


- each functional unit is independent \rightarrow multicycle design
- each tick one instruction executed (ideally)

\rightarrow pipeline hazards

- ① - instruction needs result of previous one \rightarrow stall

\rightarrow feed-forwarding



- some combinations can still result in stall



- ② - instruction reordering - we must respect data dependencies

- \triangleright read after write
- \triangleright write after read
- \triangleright write after write

Instruction scheduling \rightarrow create DAG between instructions representing dependencies

\rightarrow sorting the graph with any topological sort is ok

- choosing best order is NP-hard problem \rightarrow heuristics

- emit instruction that:
 - does not conflict with previous one

- is likely to conflict

- as far as possible from possible conflict

\rightarrow constraint scheduling DAG $O(n^2)$

on some platforms

\rightarrow emit instructions from candidate list or either NOP or inst satisfying at least last two rules

Dynamic scheduling

- modern CPUs (some) do scheduling instructions dynamically
 - complex technique in hardware
 - CPU sees only smaller group of instructions
 - sometimes can predict e.g. branches better than compiler could
- [Out-of-order & Speculative execution]

Entrópie zprávy (řádu 0 a vyšší), modelování. Statistické metody

- data redundancy removal
- short codes for common events

kompresce dat

Model: static, semi-adaptive, adaptive

Entrópie S - finite set of source units, C - finite set of codewords $f: S \rightarrow C^+$

entropy of unit x_i : $H_i = -\log_2 p_i$ - míra nejistoty?

$$\boxed{H_{\text{avg}}(S) = \sum_{i=1}^n p_i \log_2 p_i \text{ bits}} \quad - \text{průměrná entrópie} - \text{abeceda dilly m (velikost)}$$

entrópic vztahu!

entropy of source message $X = x_1 x_2 x_3 \dots x_k \in S^+$

$$\boxed{H(X) = -\sum_{j=1}^k \log_2 p_j} \quad - \text{k je délka zprávy } X$$

length of message X

$$L(X) = \sum_{j=1}^k d_j \text{ bits}$$

redundancy of code K for message X :

$$\boxed{R(X) = L(X) - H(X) = \sum_{j=1}^k (d_j + \log_2 p_j) \text{ bits}}$$

average length of a codeword:

$$\boxed{L_{\text{avg}}(K) = \sum_{i=1}^n d_i p_i \text{ bits}}$$

average redundancy of code K :

$$\boxed{R_{\text{avg}}(K) = L_{\text{avg}}(K) - H_{\text{avg}}(K)}$$

Empirical entropy > 0-th order ($T \in S^+$)

$$H_0(T) = - \sum_{w \in S} \frac{m_w^a}{n} \log_2 \frac{m_w^a}{n}$$

number of symbols a in
message T

> k -th order entropy

$$H_k(T) = \frac{1}{m} \sum_{w \in S^k} |w| H_0(w)$$

symbols following each other in T

$$0 \leq H_k(T) \leq H_{k-1}(T) \leq \dots \leq H_0(T) \leq \log_2 |S|$$

$$H_0 = T = \text{abbbaababbaaababa}$$

a	9/17
b	8/17

$$H_0 = - \left(\frac{9}{17} \cdot \log_2 \frac{9}{17} + \frac{8}{17} \cdot \log_2 \frac{8}{17} \right) = 0,9975$$

$$H_1 = w_T$$

a	babbaabb
b	bbaabaaa

$$\begin{aligned} H_1 &= \frac{1}{17} \cdot \left(8 \cdot H_0(\text{babbaabb}) + 8 \cdot H_0(\text{bbaabaaa}) \right) \\ &= \frac{1}{17} \cdot \left(8 \left(\frac{5}{8} \cdot \log_2 \frac{5}{8} + \frac{3}{8} \cdot \log_2 \frac{3}{8} \right) + \dots \right) = 0,8983 \end{aligned}$$

Shannon - Fano

- dělení intervalů pravděpodobnosti na poloviny (co nejvíce)
- nezávazující optimální kód, jednoduchý na implementaci

$$H_{\text{avg}} \leq L_{\text{avg}} \leq H_{\text{avg}} + 1$$

Huffman coding - optimální prefiksový kód

- konstrukce bottom-up \rightarrow vyřazení prioritní fronty

Sibling property - brády mají sourovnání (kromě root uzelu)

- uzel se dají řadit dle klesající pravděpodobnosti tak, že sourozenci budou v listu vedle sebe (levý uzel má liché pozici, pravý uzel má párovou pozici)

Huffman code \Leftrightarrow Sibling property

Kraft - MacMillan Inequality

$$\sum_{i=1}^n 2^{-L_i} \leq 1 \quad \begin{array}{l} \text{- kód je jednoznačný kód s délky kódů } L_1, \dots, L_n \\ \text{musí splňovat tuto podmínku} \end{array}$$

- pokud je suma větší než 1, kód může jehož značení dekódovat

Arithmetické coding

- kodujeme do intervalu rozděleného dle pravděpodobnosti jednotlivých symbolů
- konce se musí řešit EOF charakterem nebo explicitním dílkou
- hodnota bloku je entropie $H_{avg} = -\sum p(x_i) \log_2 p(x_i)$
- velmi dobrý pro adaptivní model

Celocíselná implementace:

$$\text{Low} = xxxx\ 00\dots \quad (0,0000)$$

$$\text{High} = yyyy\ 99\dots \quad (0,9999)$$

- stejná hodnota jsou poslány na systém a proměnné shiftování dolava

$$\text{Low} = \text{Low} + (\text{High} - \text{Low} + 1) \text{HighCP}(x) / \text{Total freq} - 1$$

$$\text{High} = \text{Low} + (\text{High} - \text{Low} + 1) \text{LowCP}(x) / \text{Total freq}$$

Underflow: ~~High~~^{Low} = 49xxxx & High = 50yyyy

$$\rightarrow \text{Low} = 4xxxx0, \text{High} = 5yyyy9, \text{counter}++$$

- první polohu obě Low [0] = High [0] = 4

→ output 4 a counter 9

→ v opakovaném případě 5 a můžu

Adaptive arithmetic coding

- syntetický strom - každý uzel symbol, frekvence a cumulative freq. levího podstromu

a₈, 19, 40

- pravděpodobnostní blízce ke kořenu

a₂, 12, 16

a₃, 12, 8

- strom balancovaný jeho hmoty (saki or poti)

/ \

/ \

Range encoding

- bere output ne jako binární číslo, ale jako číslo jimi lze
- menší počet normalizaci, rychlejší operace

Slovníkové metody komprese dat

- slovník - $D = (M, C)$
 - ↳ homogenní sestava řečnicí (frakci)
 - ↳ mapování M \rightarrow code words

LZ77 - sliding window method

- search buffer (thousands of bytes) |S|
- look-ahead buffer (hundreds of bytes) |L|

$\mapsto (i, j, a)$

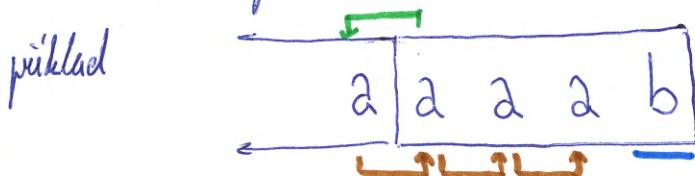
offset ↗
length ↗
need symbol
 $\lceil \log_2 S \rceil$ $\lceil \log_2 L \rceil$

$\left\{ \begin{array}{l} LZH - \text{Huffman coding} \end{array} \right.$

- výrazně zrychluje dekomprese než komprese

- offset - o kolik se vrátit repeat
- length - kolik jednom vypsal symbolů
 - může být i více než i
 - \rightarrow pokračují tím, co jsem sám vypsal

- další symbol



$(\underline{1}, \underline{3}, \underline{b})$

varianty:

LZSS

- search buffer v BST, look-ahead buffer circular queue
- dokon posílá dvě položky nebo nekomprimovatelný symbol
 - \rightarrow pro každých 8 jelen byly 2 flagy
- 2 byte break even point (polohu nejde komprimovat, output raw ...)

- Deflate
- huffman coding
 - výrobení složky na (i, j) , (a)
 - pro každý vlastní strom

- LZMA:
- podobné deflate
 - variable dictionary size
 - range encoding

- LZO - decompression speeds

- LZh - worse ratios but extremely fast

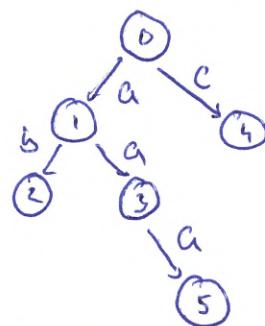
- LZ78
- growing dict (hie = partitioned tree)

$\hookrightarrow (i, a)$
where $i \uparrow$ next symbol

- index β hledaného symbolu

$$\left[\begin{array}{l} \beta(0) = 0 \quad \beta(1) = 1 \\ \beta(2i+j) = \beta(i)\beta(j) \end{array} \right]$$

aabaaacaaa



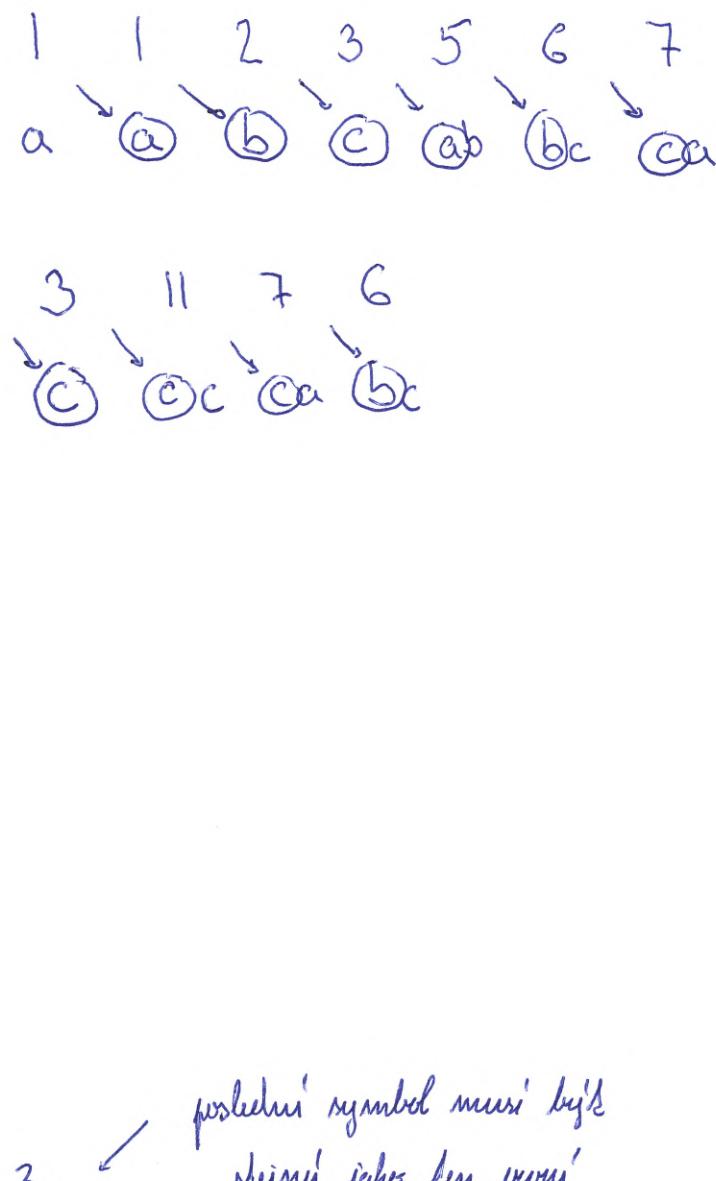
$(0,a)(1,b)(1,a)(0,c)(3,a)$

- LZW (i) - pointer to dictionary

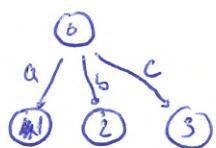
- všechny promínné inicitalizace od začátku
- poslední symbol fráze bude první symbol další fráze

increment string by:	add string to dict per phrase	add string to dict per input char
one char	LZW	LZ4
several chars	LZMW	LZAP

index	phase	coded
0	ϵ	ϵ
1	a	a
2	b	b
3	c	c
4	aa	1a
5	ab	1b
6	bc	2c
7	ca	3a
8	abb	5b
9	bcc	6c
10	cac	7c
11	cc	3?
12	ccc	11c
13	cab	7b



fungování:



- správný index, který málikuje výhled
- na něj připojím další symbol ve výhledu
- ten ale nevhodný! je pravdou pro další slovo

Kontextové metody komprese dat

PPM - Prediction by partial matching

- true for all found contexts
- each level for one context length

PPMA : escape symbol - pravděpodobnost $\frac{1}{1+m}$

PPMB : seeing twice is believing

PPMC : escape handled as others

PPM*

- dva různé kontexty, podle kterých se dří kódovat
- deterministický (vždy pokračují jiným symbol)
- stejná reprezentace kódový
- pokud nejsou, použijí neDeterministický (jako u PPM)
- deterministický kontext může být neomezený (délka)

DCA

- reakcioná slova, anti-dictionaries
- minimální reakce slov
 - × [1...n-1] a × [2...n] majou reakcioná slova

- komprese & dekomprese
 - output pouze to, co se nedaří rozložit do AD

- konstrukce AD → strom → suffix linky do min. reakcioných slov
- synchronizace ?

Verze

static

dynamic - 'jeden Schritt', bei synchronisierung, exceptions

almost antivord - geringe δ für viele, mehrere slava hörbar

- 2 pass

ACB - Associate Codes of Burrows-Wheeler

- unbounded context and content

→ pairs inserted into dictionary & sorted right to left (by ~~the~~ context)

- output (d, l, a)

first unmatched in lookahead buffer

distance between best context and content number of matched symbols

1 swissm

context is swissm

2 swissm

→ best match for it is between 2 & 3

3 swissm

(where would it appear sorted)

4 swissm

content is iss_is missing

5 swiss1m

→ best match is 4 symbols long at position 6

6 swi1ssm

→ (6-2, 4, :)

Varianten: - context matching at least k symbols

4 xx...x0zz...z0....

(4, b, b) length ($b - b$)

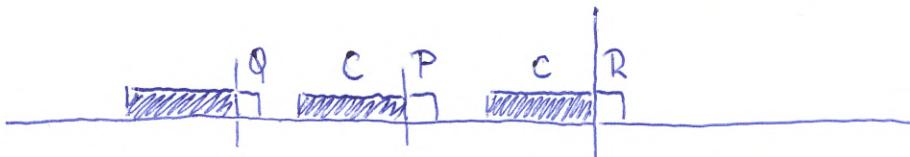
5 xx...x0zz...z1....

position mismatching bit

5 xx...x1....

Symbol ranking - next symbol is often easy to guess (3 or 4 ~~guesses~~ ^{guesses})

- context C of size n for symbol R → position of R in the list



Organizace a správa dynamické objektové paměti. Základní techniky

čištění: Mark & Sweep, Mark & Compact, Baker, Incremental

Mark & Sweep

Objekt tabule

- tabulka obsahuje adresy všech existujících objektů
- pro přístup používají index v tabulce nebo jiný identifikátor

- ⊕ - jednoduchý přístup k objektům
- minimální početnou adresu ještě maximální objekt

- ⊖ - časová náročnost přístupu
- GC musí bliskat i k tabulce
- volné místo po odstranění zahrnuje

Direct reference

- užitkové jméno na daný objekt
- ⊕ - rychlý výkon
- ⊖ - složitější přemístění objektu (GC)

Immediate value

- pro malé hodnoty je reprezentace objektu efektivní
- napsat přímo do místní adresy
- lag v největším měru → nutnost rozšířit celý adrese prostor
- (adresy jsou rozložené na celou možnou délku)

Garbage collection = automatická správa paměti

- ⊕ nemusíme se starat o paměť + chybky jako dangling pointer a memory leak mají různou
- ⊖ náročnější na paměť, CPU

Reference counting - obvious ...

- paměťově náročnější & nedekončuje cykly

Mark & Sweep - všechny objekty mají párku ne

- algoritmus prochází všechny objekty od koncového množiny (reservoir a registry) a dává ano
- nahoru odmírá všechno s ne
- reálně se může stát výjimka přeznačení

⊕ - paralelní menování (+1 bit)

- řeší výkonné rávosti

⊖ - program rozštaven při GC

- dechává k fragmentaci paměti

Mark & Compact - řeší fragmentaci

- markuje jeho předchozí

- compact průcházkou celého haldy cel mizkých k vysokým adresám
→ různé bloky písmena na první pozici adresy (a paměť je offsey)

- ve druhém průchodu aktualizuje adresy

Baker (copy constructor)

- hálka rozdělena na dvě části (s jednou se nepracuje)

- hdyž dojde mimořádné objekty se přehořívají do druhé polohy (u každého je používána)

- nové budou připraveny při dalším prohořívání

Incrementální algoritmy

- GC dostavá vždy haldy čas při kterém nutela kreska práce

→ program se nerozštavuje na celou dobu

Three color Mark & Sweep

- všechny adresy jsou bílé, hranici jsou žluté

- žluté adresy průchodem → všechny posouvat směrem na žluté a užl na černé
→ hdyž neexistují žluté, smazat všechny bílé

- aby se nemusal nové vykročit užl referenčovaný černým

→ černý a referenčovaný užl je oznámen jeho žlutý

Generační algoritmy

porozumění: - velkáma objektům máloho

- Nové připravují velmi dlouho

- staré velmi malé objekty referenčují méně

→ skupina pro nízku generaci, v horší jinak rychlý GC

Virtuální metody v objektově orientovaných jazycích a jejich implementace

optimalizace vyhledávacích metod.

Dynamic dispatch - zavolení vyhledávací metody na bázi programu

(nebo rozhodnutí v příložné komplikaci)

- např. u polymorfismu (v poli base class může být i child class)

Tabulka virtuálních metod (C++)

- hovězího býka obsahující virtuální metody má výplň tabulky
 - když vykreslím komplikacemi
 - obsahuje ukazatele na konkrétní implementaci metody
- hovězí objekt býky obsahuje ukazatel na takovou tabulku
 - k němu se vezme, co zavolat

- ② výpisu společné funkce pro sloužící hierarchie

Dynamic method lookup (Java, C#)

- ~~objekt~~ býka má tabulku pouze svých metod
- na bázi se v ní hledá daná metoda - pokud se najde, zavolá se
- v opakovaném půjčení kladeš v příložkách (až do kořna)
- opět se volá jen nad virtuálními metodami (C# : virtual, override)

Single & Multiple Dispatch

- single dispatch = blížeji předvídatelné v rámci podobnosti
 - známe jeho věc a ho býku objektu
- multiple dispatch - více metod se stejným názvem lšíší se v typech argumentů
 - nejdřív se o přetížování využívá při komplikaci

Optimalizace vyhledávacích metod

- globální cache - hash tabe s klíčem (třída, metoda) a hodnotou address metody
- povýšená velikost

Inline cache, Polymorphic inline cache

- většina metod na stejném místě hledu (call sites) ji počítají
- místo vyhledávání se tam dříve uloží konkrétní metoda
- inline působí v hledu
- v případě, kdy nevhoduje, dříve se lookup
- PIC na jednom místě obsahuje více variant (malý místek)
 - různé varianty mají pouze volání (jednotky)
- call sites: monomorfické (IC), polymorfické (PIC), megamorfické

Devirtualizace

- vyřízení virtualizace při komplaci
 - nevhodný virtualizace buňky
- velmi náročné členování

Bytekód, typické instrukce, struktura, implementace interpretu bytekódu, Just-In-Time překlad

- instrukční sada navržená pro rychlé spracování interpretorem
 - strojový kód může být
→ rychlosť
 - strojový kód návislý na architektúru
→ portability
- schémata bytu marshalling/předávání parametrů

typické instrukce:

- ▷ práce se rámcem (push, pop)
- ▷ aritmetické a logické (add, negate, less-than, ...)
- ▷ volání metod a metody (invoke, call)
- ▷ skoky a podmíněné skoky (jump, jump-if-true)
- ▷ vyhodnocení objektů & konverze typů

→ reprezentace interpretuem (Python, Perl)

- mohou kompilovat sami interpret, jindy před tím manuálně (Java)
- případně ne interpretuje první strojový kód

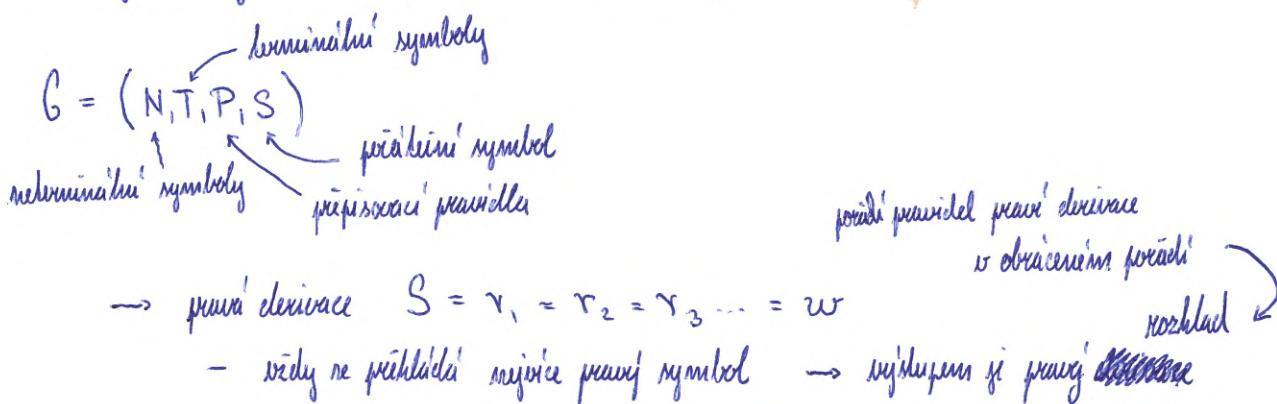
⊕ rychlosť vývoje, kompatibilita, správa paměti (např. možnost přesouvat bloky paměti → řízení fragmentace)
(jednoznačnost implementace interpretu vs kompilátorem)

JIT

- dynamicky překlad kódu na běhu programu (na strojový kód)
- přeložené části se ukládají do cache
- překládají se části často používané a mimořádné (memoriál programu)
 - můžou být známé měkké proměnné, díky kterým budeme lítě optimizovat

LR(0), SLR(k), LALR(k) a LR(k) syntaktická analýza

LR - gramatiky → pravý rozklad gramatiky



redukční automat - na rejdoru je předpona pravé větve formy
→ spolu s napředovaným výsledkem celá pravá větva forma

$$r_i = \alpha A a_j a_{j+1} \dots a_n : A \in N, \alpha \in (NUT)^*, a_j a_{j+1} \dots a_n \in T^*$$

transitions: $\delta(q, a, \varepsilon) = \{(q, a)\} \quad - \text{shift}$

$\delta(q, \varepsilon, \alpha) = \{(q, A) \mid A \rightarrow \alpha \in P\} \quad - \text{reduce}$

$\delta(q, \varepsilon, \#S) = \{(q, \varepsilon)\} \quad - \text{accept}$

minoviny: FIRST - pro každý pravidlo → jaký první symbol může být po jeho nahrazení

$$S \rightarrow \varepsilon \rightarrow \varepsilon$$

$$S \rightarrow aA \rightarrow a$$

$$S \rightarrow Ab \rightarrow \text{FIRST}(A) \quad \& \quad b \text{ pokud FIRST}(A) obsahuje \varepsilon$$

FOLLOW - pro každý nel. symbol - co se může objevit na něm

$$S \rightarrow Ab \rightarrow A: b$$

$$A \rightarrow B \rightarrow B: \text{FOLLOW}(A)$$

$$S \rightarrow X \rightarrow X: \varepsilon$$

$$\text{FOLLOW}(A) = S \rightarrow^* \alpha A \beta, \alpha \in \text{FIRST}(\beta)$$

LR(0) - parser

- nejjednodušší parser, který rozchádají na základě symbolů na vstupu (doprávou prohlížených)
- rozchádají se pouze na základě zásobníku \leftarrow strong LR - udržuje historii, pouze svého zásobníku
weak LR - pouze vlastní historii
- parser pouze k rozchádání LR automat (charakteristický automat)
- každý stav reprezentuje jeden stav zásobníku
- ve větvech (listech) jsou redukce ~ jinak konflikt

• restaurování parseru: LR(0) položky

\rightarrow součinné s rozšířenou Beckhoffovou gramatikou

$$G' = (N \cup \{S'\}, T, P \cup \{S' \rightarrow S\}, S')$$

\rightarrow restaurování množiny položek

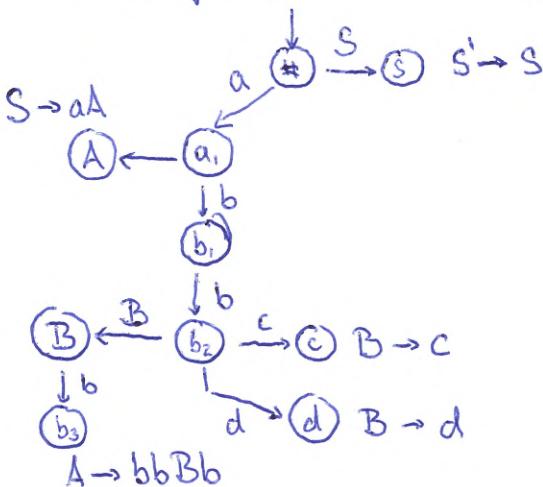
$$\Rightarrow \text{první maximální } \sigma \# := \{S' \rightarrow .S\}$$

- kartačka množina ($=$ stav charakteristického automatu) se jmenuje dle symbolu před tekoucím
- kartačka znaků, kde se momentální nacházející se čárka
- větve množiny: pokud je $\#$ před N symbolom, expanduj (přidám) i všechna pravidla nazývající N
- \rightarrow opakuji dokud nemám všechny

příklad:
 1: $S' \rightarrow S$
 2: $S \rightarrow aA$
 3: $A \rightarrow bbBb$
 4: $B \rightarrow c \quad | \quad d$

$\# = \{ S' \rightarrow .S, \quad S \rightarrow .aA \}$	$S \Rightarrow \{ S' \rightarrow S. \}$
$a_1 = \{ S \rightarrow a.A, \quad A \rightarrow .bbBb \}$	$A = \{ S \rightarrow aA. \}$
$B = \{ A \rightarrow bbB.b \}$	$b_1 = \{ A \rightarrow b.bBb \}$
$b_2 = \{ A \rightarrow bb.Bb. \}$	$b_3 = \{ A \rightarrow bbB.b. \}$

charakteristický automat:



- gramatika není silná - existují více b množin
- při čárce na zásobník uložil jsem symboly dle pojmenování stupně!
- redukuji $S' \rightarrow S$ koncem stupně (pokud není vstup)

- na rozdílku mohu mít všechny symboly ze jmen stupňů

#	P	
S	Accept	- v tomto stavu je redukce pravidlem $S \rightarrow S$, což je ekvivalentní půjči'
a,	P	
A	R(2)	příklad: Input = abbdB $(\#, abbdB, \epsilon) \vdash (\#a, bbdB) \vdash (\#a, b_1, bdb, \epsilon) \vdash$ $\vdash (\#a, b_1, b_2, db, \epsilon) \vdash (\#a, b, b_2 d, b, \epsilon) \vdash (\#a, b, b_2 B, b, 5) \vdash$ $\vdash (\#a, b, b_2 B b_3, \epsilon, 5) \vdash (\#a, A, \epsilon, 53) \vdash (\#S, \epsilon, 532) \vdash$ $\vdash (\text{accept}) \checkmark$
b, b ₁	P	
b ₂	P	
B	P	
b ₃	R(3)	
c	R(4)	
d	R(5)	

SLR(k) - parser

- občas máme konflikty mezi reduce, když LR(0) nemůžeme
- rozděluji na rozdíl FIRST & FOLLOW množin pro dané symboly (stavky)
 - množiny se svaří z původních pravidel = sváří momentální kontext

1	$E' \rightarrow E$	$s T \rightarrow F$
2	$E \rightarrow E + T$	$s F \rightarrow (E)$
3	$E \rightarrow T$	$s F \rightarrow \omega$
4	$T \rightarrow T * F$	

- LR(0) položky obsahují kontexty:

$$E_i = \{ E' \rightarrow E_i., E \rightarrow E_i. + T \}$$

$$\text{FOLLOW}(E') = \{ \epsilon \} \quad \text{FIRST}(+T) = \{ + \}$$

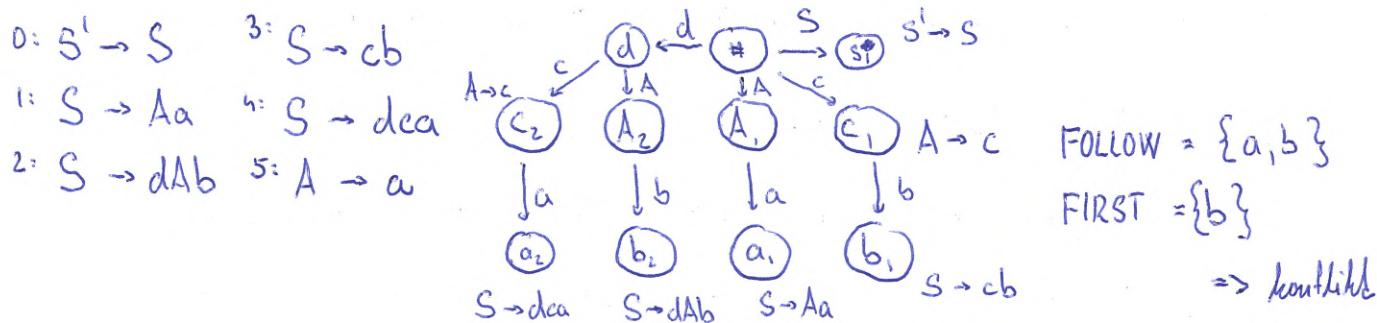
→ máme dvě jednotlivé množiny, můžeme se deterministicky rozdělit

- rozhledová tabulka má sloupec pro každý T symbol
 - myslím redukcii a symboly ve FOLLOW daného stavu (když mají v grámu přechod)

F	a	*	()	ϵ	+ předchozí přechod
#	P		P			
E,		P			accept	
T,	R(2)	P	R(2)	R(2)		
....						

LALR (k) - parser

- řešení problémů s nedifinovanými FIRST & FOLLOW
→ u podstatě již definuje pro každý stav svůj set FOLLOW



- používáme body LR(k) položky

$[A \rightarrow \alpha \cdot \beta, w]$ $w \in T^*$ - dopadnuť pořízený řetězec

$$\Rightarrow C_1 = \{[S \rightarrow c \cdot b, \epsilon], [A \rightarrow c \cdot, a]\} \quad C_2 = \{[S \rightarrow d \cdot c \cdot a, \epsilon], [A \rightarrow c \cdot, b]\}$$

- dopadnuť pořízené řetězce mohu opět využívat k automatu
- vzhledem již další symbol na expandovaném rečením

LR(k) - parser

- řešit kontakty redukce-redukce (najprve ne všechny)?
- LALR ~~jej~~ spolu s jinou nejprve jdeštěm (a tedy výhody)
→ LR(k) je spojovat nebudou
- velmi máložádá velikost - až $2^{|G|}$ množin

Formální a atributovaný překlad významu LR analyzátorem.

Formální příklad:

- překladové gramatiky → mísí i výstupní symboly
- vstupní a výstupní gramatiky (spolu nejsou překladovou gramatikou)

Překladová gramatika $PG = (N, T, D, R, S)$

$$A \rightarrow \alpha\beta \quad \alpha \in (N \cup T)^*, \beta \in D^*$$

→ výstupní symboly jsou vždy na konci pravidla (poslední')

- mísí pravidlo rozděluji mísíme rovnou překladač

druhá možnost → tabulka redukční pravidla → výstup (substituce)

LR(k) překladové gramatiky

- rozšíření na výstup i při přesunu

$$A \rightarrow \alpha x \beta \quad x \in D^*, \alpha \in T, \alpha, \beta \in (N \cup T \cup D)^*$$

R - translation gramatika - D jsou na konci nebo vždy téměř před T

+ získání neobsahující překladový kontextu

$$\Rightarrow \begin{bmatrix} A \rightarrow \alpha a \beta, x, u \\ \text{výstup} \end{bmatrix} \text{ výstup} \quad \text{pokud } x \neq y \text{ a } FIRST_k(a\beta u) \cap FIRST_k(b\delta v) \neq \emptyset$$

$[B \rightarrow r.b\delta, y, v]$

- mísí masakr pouze u přesunu, u redukce jsou pravidla farsa'

| (to se stane pokud $a = b \in T$ - tj na nějž T mám na výstupu kromě D)

Překladové gramatiky s LR(k) vstupní gramatikou

- mísíme se gramatiku transformoval na LR(k) překladovou

- gramatiky jsou ekvivalentní, pokud $Z(PG_1) = Z(PG_2)$ (~~$T_1 = T_2 \wedge D_1 = D_2$~~)

- jestliže charakteristické gramatiky G_1 a G_2 jsou ekvivalentní a $T_1 = T_2$, $D_1 = D_2$,

tak gramatiky $Z(PG_1)$ a $Z(PG_2)$ jsou také ekvivalentní

některé transformace:

- $A \rightarrow \alpha\alpha \otimes \beta \Rightarrow A \rightarrow \alpha \otimes \alpha\beta$
- $A \rightarrow \alpha\beta r \Rightarrow A \rightarrow \alpha A' r$
 $A' \rightarrow \beta$

pohlcení slva

$$\circ A \rightarrow \alpha\beta \otimes r \Rightarrow A \rightarrow \alpha A' r$$

$A' \rightarrow \beta \otimes$

pohlcení sprava

$$\circ A \rightarrow \alpha \otimes \beta r \Rightarrow A \rightarrow \alpha A' r$$

$A' \rightarrow \otimes \beta$

vláčení ϵ -pravilla

- $A \rightarrow \alpha \otimes r \Rightarrow A \rightarrow \alpha A' r$
 $A' \rightarrow \otimes$

- jedna ze pouze o pohlcení kde β je prázdná
- vstupní gramatika může mít i $LR(k)$
(pouze halo transformace)

příklad

- pravilla s levou rekurencí jsou problematická (ale pouze pro ~~vstupní gramatiku~~)

$$A \rightarrow yAby \quad A \rightarrow A'Aby \quad - \text{toto ale můži } LR(k) \text{ pro následující } k$$
$$A \rightarrow ax \quad \Rightarrow \quad A' \rightarrow y$$
$$A \rightarrow ax$$

◦ rekursivní

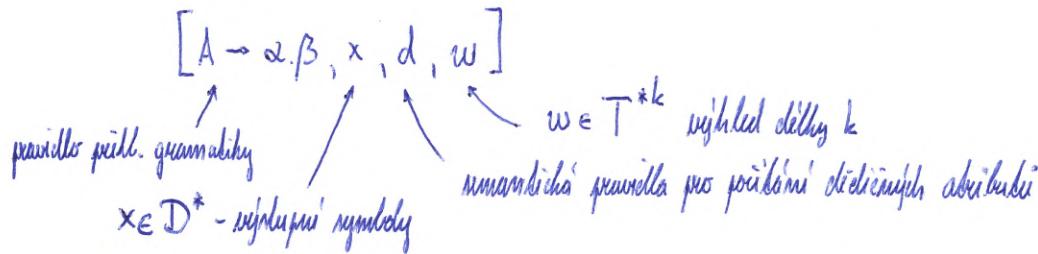
$$A \rightarrow \alpha x B \beta \quad A \rightarrow \alpha A' \beta$$
$$B \rightarrow \delta_1 | \delta_2 | \dots | \delta_n \quad \Rightarrow \quad A' \rightarrow x \delta_1 | x \delta_2 | \dots | x \delta_n$$

- rekursivní ale nebude na předchozí případ také fungovat
→ rekurenci nevyužili
- rekursivní fungují pouze pokud vstupní gramatika nemá symbol před leví rekurenciím symbolem
- ne všechny gramatiky lze transformovat
 - např. posloupný řečík na proti sebe stojící možnost

LR - atributované gramatiky

- umožňují i výčetné atributy

atributovaná LR(k) poloha pro ATG = (TG, A, V, F)



semantický konflikt:

$[A \rightarrow \alpha \beta, x_1, d_1, \underline{w}]$

$[B \rightarrow \gamma \delta, x_2, d_2, v]$

smyslné výsledky sem. pravidla

pokud $d_1 + d_2 \wedge \beta, \delta \in (N \cup D)^* \cup \{\epsilon\}$

nebo $x_1 \neq x_2 \wedge$

$\text{FIRST}_L(\beta u) \cap \text{FIRST}_L(\delta v) \neq \emptyset$

nevýznačené postupy na výstup (jako příkladový konflikt)

→ parsing table ještě obsahuje matice s sem. pravidla

S - abstraktní gramatika

- vstupem je posfixní gramatika
- nekonvenční symboly mají pouze symbolizované atributy
- dělící se majou priority v sémantických pravidlech

založeno na předních a následních slv.

založeno na abstraktech potomků

modifikace LR - parseru:

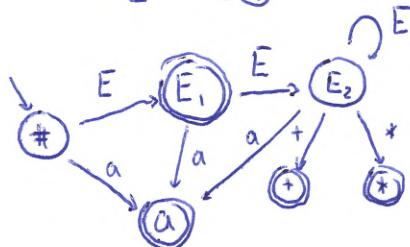
- vstupní symbol je uložen na záložním místě svých symbolizovaných atributů
- při redukcích se vypočítají dělící se atributy výsledných symbolů k pravé straně pravidla
 - i se svými atributy jsou předány na výsledek
- spojují se své sl. atributy N na levé straně a uloží na záložní

?
 alternativné: atribut počítáního symb. S je upřesňující atribut
 → výsledek je hodnota upřesňujícího atributu v konci stranu

příklad: $E \rightarrow \oplus EE^+$

$E \rightarrow \otimes EE^*$

$E \rightarrow a @$



$$\# = \{E' \rightarrow E, E \rightarrow .EE^+, E \rightarrow .EE^*, E \rightarrow .a\}$$

$$a = \{E \rightarrow a.\}$$

$$E_1 = \{E \rightarrow EE_1., E \rightarrow EE_1^*, E \rightarrow EE_1^+, E \rightarrow EE_1^*\}$$

$$+ = \{E \rightarrow EE_2., E \rightarrow EE_2^*, E \rightarrow EE_2^+\}$$

$$* = \{E \rightarrow EE_2^*, E \rightarrow EE_2^+\}$$

$$E_2 = \{E \rightarrow EE_2., E \rightarrow EE_2^*, E \rightarrow EE_2^+\}$$

$$E \rightarrow E.E^+, E \rightarrow E.E^*, E \rightarrow .EE^+, E \rightarrow .EE^*, E \rightarrow .a\}$$

$$TR_1: E^0 \rightarrow E'E^2+$$

$$E^0_P = f_3(\oplus, E'_P, E^2_P)$$

$$TR_2: E^0 \rightarrow E'E^2^*$$

$$E^0_P = f_3(\otimes, E'_P, E^2_P)$$

$$TR_3: E^0 \rightarrow a$$

$$E^0_P = @$$

$$(\#, aa+a*, \epsilon) \vdash (\#a, a+a*, \epsilon) \xrightarrow{R_3} (\#E_1(a), a+a*, \epsilon)$$

$$\vdash (\#E_1(a)a, +a*, \epsilon) \xrightarrow{R_3} (\#E_1(a)E_2(a), +a*, \epsilon) \vdash (\#E_1(a)E_2(a)+, a*, \epsilon) \xrightarrow{R_1} (\#E_1(+aa), a*, \epsilon) \vdash$$

$$\vdash (\#E_1(+aa)a, +a*, \epsilon) \xrightarrow{R_3} (\#E_1(+aa)E_2(a), +a*, \epsilon) \vdash (\#E_1(+aa)E_2(a)*, \epsilon, \epsilon) \xrightarrow{R_2} (\#E_1(*+aaa), \epsilon, \epsilon) \vdash$$

$$\vdash \text{accept}$$

	a	+	*	E
#	sh			
a	R(3)	R(3)	R(3)	R(3)
+	R(1)	R(1)	R(1)	R(1)
*	R(2)	R(2)	R(2)	R(2)
E ₁	sh			Acc
E ₂	sh	sh	sh	

$$1) E \rightarrow \oplus EE+ \quad E.p = f_2(\oplus, E_{1,p}, E_{2,p}) \quad 0) E' \rightarrow E$$

$$2) E \rightarrow \odot EE* \quad E.p = f_3(\odot, E_{1,p}, E_{2,p})$$

$$3) E \rightarrow a@ \quad E.p = @$$

$$+ = \{ E \rightarrow EE+ \}$$

$$* = \{ E \rightarrow EE* \}$$

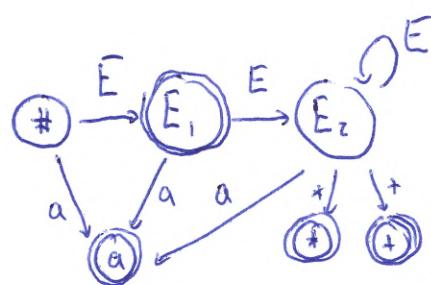
$$\# = \{ E' \rightarrow .E, \\ E \rightarrow .EE+, \\ E \rightarrow .EE*, \\ E \rightarrow .a \}$$

$$E_1 = \{ E' \rightarrow E. \}$$

$$\cancel{E_1} = \{ E \rightarrow E.E+, \\ E \rightarrow E.E*, \\ E \rightarrow .EE+, \\ E \rightarrow .EE*, \\ E \rightarrow .a \}$$

$$E_2 = \{ E \rightarrow EE.+ ,$$

$$E \rightarrow EE.+ , \\ E \rightarrow E.E+, \\ E \rightarrow E.E*, \\ E \rightarrow .EE+, \\ E \rightarrow .EE*, \\ E \rightarrow .a \}$$



$$a = \{ E \rightarrow a. \}$$

	a	+	*	ϵ
#	sh			
a	R(3)	R(3)	R(3)	R(3)
E ₁	sh		acc	
E ₂	sh	sh	sh	
+	R(1)	R(1)	R(1)	R(1)
*	R(2)	R(2)	R(2)	R(2)

$$\begin{aligned}
 & (\#, aaaa+*, E) \vdash (\#a, aa+*, \epsilon) \vdash \\
 & \vdash (\#E_1(a), aa+*, \epsilon) \vdash (\#E_1(a)a, a+*, \epsilon) \vdash (\#E_1(a)E_2(a), a+*, \epsilon) \\
 & \vdash (\#E_1(a)E_2(a)a, +*, \epsilon) \vdash (\#E_1(a)E_2(a)E_2(a), +*, \epsilon) \vdash \\
 & \vdash (\#E_1(a)E_2(a)E_2(a)+, *, \epsilon) \vdash (\#E_1(a)E_2(+aa), *, \epsilon) \vdash \\
 & \vdash (\#E_1(a)E_2(+aa)*, \epsilon, \epsilon) \vdash (\#E_1(*a+aa), \epsilon, \epsilon) \vdash acc
 \end{aligned}$$

