Principy překladačů

Překladač



Jakub Yaghob

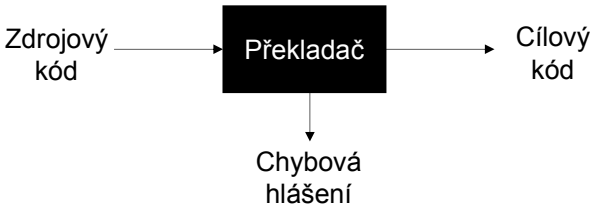
Literatura a slajdy

- Aho, Sethi, Ullman: Compilers - Principles, Techniques and Tools, Addison-Wesley 1986
- Chytil M.: Automaty a gramatiky, SNTL 1984
- Muchnick S.S.: Advanced compiler design and implementation, Morgan Kaufman Publishers 1997

- ftp://ulita.ms.mff.cuni.cz/predn/PP
- http://ulita.ms.mff.cuni.cz/pub/predn/PP/

Co je to překladač?

- Naivní představa
  - Černá krabička, která převádí (překládá) zdrojový kód do cílového kódu



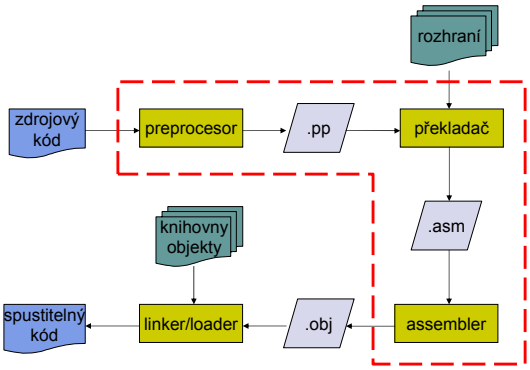
Co je to překladač? O něco formálněji

- Mějme vstupní jazyk L_{in} generovaný gramatikou G_{in}
- Dále mějme výstupní jazyk L_{out} generovaný gramatikou G_{out} nebo přijímaný automatem A_{out}
- Překladač je zobrazení $L_{in} \rightarrow L_{out}$, kde $\forall w_{in} \in L_{in}$ $\exists w_{out} \in L_{out}$. Pro $w_{in} \notin L_{in}$ zobrazení neexistuje

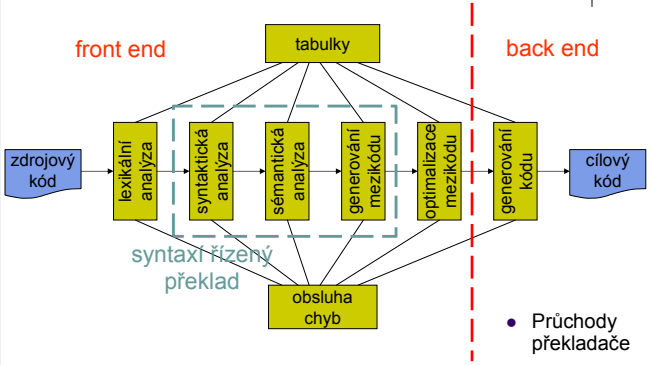
Příklady užití technik překladačů

- Strukturovaný nebo syntax-highlighting editor
- Pretty-printer
- Statické kontroly programu
  - LINT
- Interpretery
- Překladače modelovacích jazyků
  - Verilog, VHDL
- Dotazovací jazyky
  - SQL

Překlad programu



Fáze překladače



- Průchody překladače

Nástroje pro vývoj překladačů

- Generátory parserů
  - Produkují syntaktický analyzátor
  - Obvykle popis bezkontextovou gramatikou
  - Bison, Coco/R, ANTLR
- Generátory scannerů
  - Produkují lexikální analyzátor
  - Obvykle popis regulární gramatikou
  - Flex
- Generátory generátorů kódu
  - Produkují překlad pro každou instrukci mezikódu do cílového kódu
  - Model procesoru a jeho popis
  - Mono JIT

Principy překladačů

Mezikód



Jakub Yaghob

Mezikód

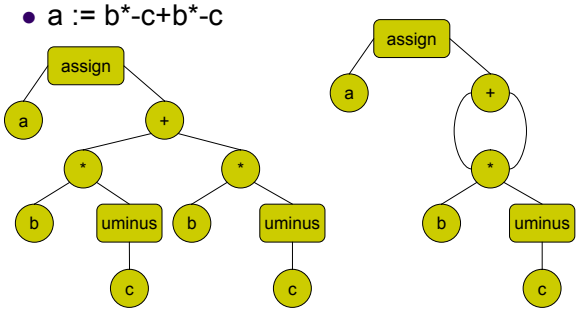
- Přechodná reprezentace zdrojového kódu
- Odděluje front end od back endu
- Výhody mezikódu
  - Různé back endy pro stejný vstupní jazyk – podpora různých procesorů
    - gcc
  - Různé front endy pro stejný výstupní jazyk – podpora více jazyků pro stejný stroj
    - .NET
  - Strojově nezávislé optimalizace nad mezikódem - HLO

Možné typy reprezentací

- Syntaktický strom
  - Možný i DAG
- Postfixový zápis
  - Linearizovaná reprezentace syntaktického stromu
  - Hrany grafu nejsou přímo v zápisu, dají se zrekonstruovat z pořadí a počtu operandů operátoru
- Tříadresový kód
  - Také linearizovaná reprezentace syntaktického stromu
  - Vypadá jako instrukce procesoru
  - x := y op z

Příklady – syntaktický strom a DAG

- a := b*-c+b*-c



Příklady – postfixový zápis a tříadresový kód

- a b c uminus * b c uminus * + assign

- t1 := -c
- t2 := b * t1
- t3 := -c
- t4 := b * t3
- t5 := t2 + t4
- a := t5

- t1 := -c
- t2 := b * t1
- t5 := t2 + t2
- a := t5

Operandy tříadresového kódu

- Jméno
  - Proměnná
  - Typ
  - Další možná jména
- Konstanta
  - Různé možné literály (celé číslo, řetězec, …)
- Přechodná proměnná
  - Generovány překladačem
  - Snadno si je lze představit jako registry CPU

Typy tříadresových instrukcí

- Binární aritmetické nebo logické operace
- Unární operace
- Přiřazení/kopie
- Nepodmíněný skok
- Podmíněný skok
- Mechanismus volání procedur/funkcí
  - Parametry, volání, návrat
- Indexace pole
- Adresové operátory
  - Vzetí adresy, dereference
- Deklarace

Implementace tříadresového kódu – čtveřice

- Záznam se čtyřmi položkami
  - *op*, *arg1*, *arg2*, *res*
- Některé *arg* (typicky *arg2*) nebo i *res* nemusí být pro některé typy tříadresových instrukcí využity
- Operandy jsou odkazy do tabulek symbolů

	op	arg1	arg2	res
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	:=	t5		a

Implementace tříadresového kódu – trojice



- Chceme se vyhnout generování přechodných jmen
- Záznam s třemi položkami
  - *op*, *arg1*, *arg2*
- Operandy jsou odkazy do tabulek symbolů (pro jména a konstanty) nebo odkaz na jinou trojici

	op	arg1	arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	a	(4)

Implementace tříadresového kódu – nepřímé trojice



- Jedno pole přímo s trojicemi
- Druhé pole s odkazy na trojici

	op	arg1	arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	a	(4)

	stmt
[0]	(0)
[1]	(1)
[2]	(2)
[3]	(3)
[4]	(4)
[5]	(5)

Srovnání implementací



- Čtveřice
  - Výhodnější pro optimalizování mezikódu
  - Lze s nimi snadno pohybovat, jména zůstávají stejná
- Trojice
  - Úspornější
  - Špatná manipulace při optimalizaci – přečíslování v celém mezikódu
- Nepřímé trojice
  - Lze s nimi dobře manipulovat při optimalizaci – změna pouze v druhém poli
  - Zhruba stejně paměťově náročné, jako čtveřice

Graf toku řízení



- Grafová reprezentace tříadresového kódu
- Uzly jsou výpočty
- Orientované hrany představují tok řízení
- Vhodný pro optimalizace, ale i pro reprezentaci

Základní blok



- Souvislá sekvence tříadresového kódu, do které tok řízení vstoupí na začátku a vystoupí na úplném konci a nikde jinde
- Algoritmus na rozsekání sekvence tříadresového kódu na základní bloky
  - Určení množiny vůdců
    - Začátek sekvence je vůdce
    - Každá instrukce mezikódu, která je cílem skoku, je vůdce
    - Každá instrukce mezikódu, která následuje těsně po instrukci skoku, je vůdce
  - Pro každého vůdce se základní blok sestává z vůdce a všech následujících tříadresových kódu až po dalšího vůdce (bez něj) nebo konec sekvence

Konstrukce grafu toku řízení

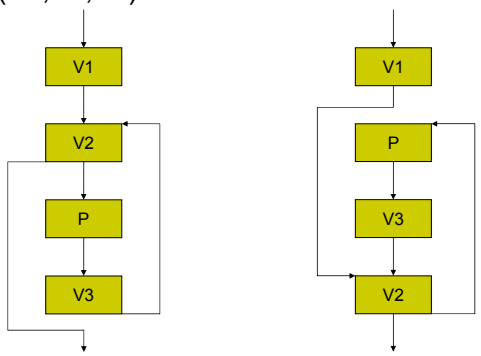


- Orientovaný graf
- Uzly jsou základní bloky
- Jeden ze základních bloků je počáteční – blok, jehož vůdce je první instrukcí mezikódu v sekvenci
- Mezi dvěma základními bloky B_1 a B_2 je orientovaná hrana z B_1 do B_2, pokud platí jedna z podmínek
  - Poslední příkaz B_1 je skok na vůdce B_2
  - B_2 následuje ihned za B_1 a B_1 nekončí nepodmíněným skokem

Příklad grafu toku řízení



- for(V1;V2;V3) P



Životnost proměnné



- Tříadresová instrukce x := y op z *definuje* x a *používá* y a z
- Říkáme, že proměnná je *živá* v daném bodě mezikódu, pokud tento bod leží na nějaké cestě začínající bodem, kde byla definována, do nějakého bodu, kde byla použita, přičemž po této cestě nejsou žádné další definice

Principy překladačů

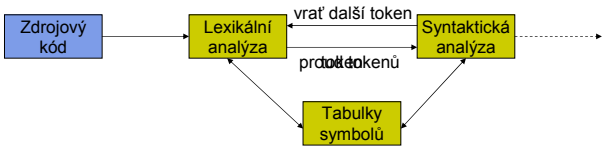
Lexikální a syntaktická analýza



Jakub Yaghob

Lexikální analýza

- Pokud jazyk podporuje nějaká makra, obvykle se řeší zde
- Zjednodušení návrhu
  - Syntaktická i lexikální analýza
- Specializace lexikálního scanneru
  - Zrychlení
- Zvětšení přenositelnosti
  - Přechod na jinou platformu přinese změny jen v lexikální analýze (EBCDIC)
- „Zvětšení“ výhledu gramatiky
  - LR(1) neznamená výhled dopředu na jeden znak



Pojmy lexikální analýzy

- Token
  - Výstup lexikální analýzy a vstup syntaktické analýzy
  - Na straně syntaktické analýzy se nazývá terminál
  - Množina řetězců, které produkují stejný token
- Pattern
  - Pravidla, která popisují množinu řetězců pro daný token
  - Obvykle se využívá regulárních výrazů
- Lexém, lexikální element
  - Sekvence znaků ve zdrojovém kódu, která odpovídá nějakému patternu nějakého tokenu
  - Některé lexémy nemají výstup jako token
    - Komentář
- Literál
  - Konstanta, má svoji hodnotu

Příklady

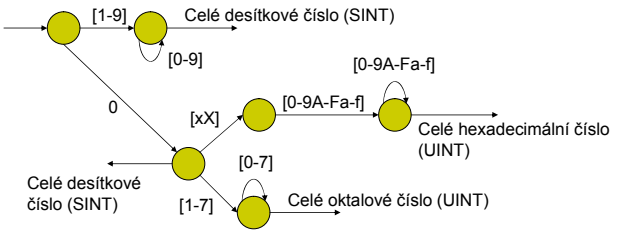
Token	Lexém	Regulární výraz
while	while	while
relop	<,<=,=,<>,>,>=	\<|\<=|=|\<>|>|>=
uint	0, 123	[0-9]+
	/* komentář */	\/* → cmt, <cmt>., <cmt>*\/

Problémy s lexikální analýzou

- Zarovnání na vstupní řádce
  - Některé jazyky mají zarovnání na řádce jako svoji syntaktickou konstrukci
    - Python, Flex
- Identifikátory
  - Identifikátory s mezerami
    - DO 5 I = 1.25
    - DO 5 I = 1,25
  - Klíčová slova jako identifikátory
- Kontextově závislé tokeny
  - Token závisí na jiných informacích
    - a*b;

Pozadí lexikální analýzy

- Patterny používají regulární výrazy → regulární jazyky → rozpoznávány konečnými automaty
- Restartování automatu po každém rozpoznaném tokenu
- Konečný automat pro celé číslo v C:



Atributy tokenů

- Pokud je token rozpoznáván více patterny nebo pokud je to literál
- Typicky jeden atribut, který upřesňuje token nebo udává hodnotu literálu
  - Token=relop, upřesnění=‘<=’
  - Token=uint, upřesnění=‘123’

Lexikální chyby

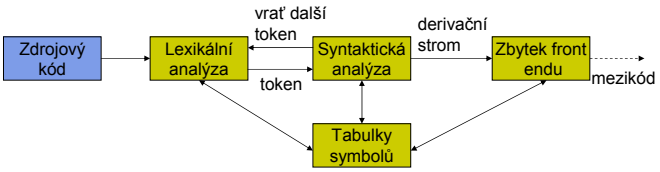
- Chyby, které nastanou v okamžiku, kdy konečný automat nemůže pokračovat dál a není v konečném stavu
  - Neznámý znak
  - Neukončený řetězec do konce řádky
- Zotavení
  - Ignorovat
  - Domyslet si chybějící znak(y)
- Překlep v klíčovém slově obvykle není lexikální chyba, ale vypadá jako identifikátor
- Může dost rozhodit syntaktickou analýzu

Bufferování vstupu

- Lexikální analýza zabírá 60-80% doby překladu
- Jedno z možných zrychlení: čtení vstupního souboru po blocích (bufferech) a práce automatu v paměti bufferu
- Potíže
  - Vložení souboru znamená „vnoření“ bufferu
    - #include

Syntaktická analýza

- Hlavní úkol
  - Rozpoznat, zda slovo na vstupu je slovem ze vstupního jazyka
  - Mluvíme o bezkontextových gramatikách a tudíž i o zásobníkových automatech
- Další důležité úkoly
  - Syntaxí řízený překlad řídí celý překladač
  - Stavba derivačního stromu



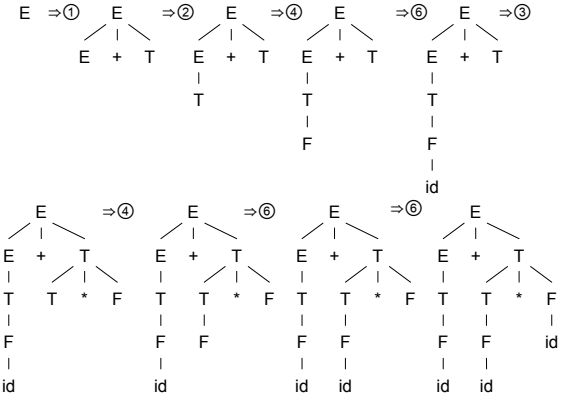
Pověstná gramatika

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

Derivační stromy

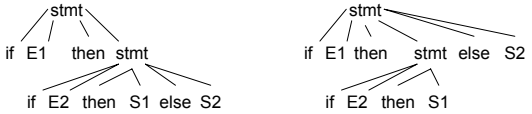
- Grafická reprezentace derivací použitím stromů
  - Uzly jsou neterminály i terminály
  - Hrany od neterminálu z levé strany pravidla na všechny symboly z pravé strany
- E ⇒① E+T ⇒② T+T ⇒④ F+T ⇒⑥ id+T ⇒③ id+T*F ⇒④ id+F*F ⇒⑥ id+id*F ⇒⑥ id+id*id

Příklad



Nejednoznačná gramatika

- Lze sestrojit různé derivační stromy pro stejné vstupní slovo
- Příklad ze života (dangling else):
  - stmt → **if** expr **then** stmt
    | **if** expr **then** stmt **else** stmt
    | **while** expr **do** stmt
    | **goto** num
  - Vstupní slovo: **if** E_1 **then if** E_2 **then** S_1 **else** S_2



Odstranění nejednoznačnosti

- Vyjasnit si, který derivační strom je ten správný
- V našem případě platí, že **else** se páruje s nejbližším „volným“ (bez **else**) **if**
- Idea: mezi **if** a **else** je vždy spárovaný příkaz
  - stmt → m_stmt
  - | u_stmt
  - m_stmt → **if** expr **then** m_stmt **else** m_stmt
    | **while** expr **do** m_stmt
    | **goto** num
  - u_stmt → **if** expr **then** stmt
    | **if** expr **then** m_stmt **else** u_stmt
    | **while** expr **do** u_stmt

Eliminace levé rekurze

- Gramatika je levě rekurzivní, pokud je tam neterminál A, pro který platí $A \Rightarrow^{+} A\alpha$ pro nějaký řetězec α
- Problém pro analýzu shora-dolů
- Jednoduchý návod pro $\beta\alpha^{m}$:
  - A → Aα
  - A → β
  - A → βA’
  - A’ → αA’
  - A’ → Λ

Odstranění levé rekurze na pověstné gramatice

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

1. E → TE'
2. E' → + TE'
3. E' → Λ
4. T → FT'
5. T' → * FT'
6. T' → Λ
7. F → (E)
8. F → id

Levá faktorizace

- Když není jasno, které ze dvou možných variant si vybrat
- Přepsat ekvivalentně gramatiku s tím, že odložíme rozhodnutí na pozdější dobu, až bude vidět, které z pravidel si vybrat
  - $A \rightarrow \alpha\beta_1$
  - $A \rightarrow \alpha\beta_2$
  - $A \rightarrow \alpha A'$
  - $A' \rightarrow \beta_1$
  - $A' \rightarrow \beta_2$

Jazykové konstrukce, které nejsou bezkontextové

- L_1={ wcw | w=(a|b)* }
  - Kontrola, zda identifikátor w je deklarován před použitím
- L_2={ $a^n b^m c^n d^m$ | n≥1, m≥1 }
  - Kontrola, zda počet parametrů v deklaraci funkce odpovídá počtu parametrů při volání funkce.
- L_3={ $a^n b^n c^n$ | n≥0 }
  - Problém „podtržítkování“ slova
    - a je znak, b je BS, c je podtržítko
  - (abc)* je regulární výraz

Operátory FIRST a FOLLOW – definice

- Pokud je α řetězec symbolů gramatiky, pak FIRST(α) je množina terminálů, kterými začínají všechny řetězce derivované z α. Pokud α může zderivovat na Λ, pak Λ je také ve FIRST(α)
- Definujme FOLLOW(A) pro neterminál A jako množinu terminálů, které se mohou vyskytovat těsně za A v nějakém řetězci, který vznikl derivací z počátečního neterminálu gramatiky (S ⇒* αAaβ, pro nějaká α a β). Pokud je A nejpravější symbol v nějakém přepisu, pak i $ je ve FOLLOW(A).

Konstrukce FIRST

- Konstrukce pro symbol gramatiky X
  - Pokud je X terminál, pak je FIRST(X)={X}
  - Pokud existuje přepisovací pravidlo X→Λ, pak přidej Λ do FIRST(X)
  - Pokud je X neterminál a $X \rightarrow Y_1 Y_2 \ldots Y_k$ je přepisovací pravidlo, pak přidej **a** do FIRST(X), pokud je **a** ve FIRST(Y_i) pro nějaké i a ∀ j<i platí, že Λ∈FIRST(Y_j). Pokud ∀ j je Λ∈FIRST(Y_j), pak přidej Λ do FIRST(X)
- Konstrukce pro řetězce
  - Pro řetězec $X_1 X_2 \ldots X_n$ je konstrukce FIRST podobná jako pro neterminál.

Konstrukce FOLLOW

- Konstrukce pro neterminál
  - Přidej $ do FOLLOW(S), pokud S je počáteční neterminál gramatiky a $ je značka pro EOS
  - Mějme přepisovací pravidlo A→αBβ. Pak přidej FIRST(β) do FOLLOW(B) kromě Λ
  - Mějme přepisovací pravidla A→αB nebo A→αBβ, kde Λ∈FIRST(β). Pak přidej vše z FOLLOW(A) do FOLLOW(B)

FIRST a FOLLOW – příklad s pověstnou gramatikou

- FIRST(E)={ (, id }
- FIRST(T)={ (, id }
- FIRST(F)={ (, id }
- FIRST(E')={ +, Λ }
- FIRST(T')={ *, Λ }

- FOLLOW(E)={), $ }
- FOLLOW(E')={), $ }
- FOLLOW(T)={ +,), $ }
- FOLLOW(T')={ +,), $ }
- FOLLOW(F)={ +, *,), $ }

Analýza shora dolu

- Pokus najít nejlevější derivaci pro vstupní řetězec
- Pokus zkonstruovat derivační strom pro daný vstup počínaje kořenem a přidáváním uzlů do stromu v preorderu
- Řešeno obvykle rekurzivním sestupem
  - Rekurzivní sestup pomocí procedur
- Nerekurzivní analýza s predikcí
  - Automat s explicitním zásobníkem
- Každé z těchto řešení má potíže s levou rekurzí v gramatice
- Dnes používáno v generátorech parserů
  - ANTLR, CocoR – LL(1) gramatiky s řešením konfliktů natažením výhledu na k

Rekurzivní sestup

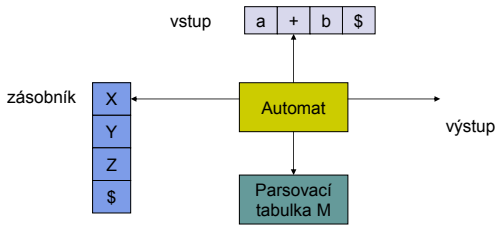
- Jedna procedura/funkce pro každý neterminál gramatiky
- Každá procedura dělá dvě věci
  - Rozhoduje se, které pravidlo budou použito na základě výhledu. Pravidlo s pravou stranou α bude použito, pokud je výhled ve FIRST(α). Je-li tam konflikt pro nějaký výhled mezi více pravými stranami, pak se tato gramatika nedá použít pro rekurzivní sestup. Pravidlo s Λ na pravé straně se použije tehdy, pokud výhled není ve FIRST žádné pravé strany.
  - Kód procedury kopíruje pravou stranu pravidla. Výskyt neterminálu znamená zavolání procedury neterminálu. Výskyt terminálu je kontrolován s výhledem, a pokud souhlasí, je přečten. Pokud na nějakém místě terminál nesouhlasí, došlo k chybě.

Rekurzivní sestup – příklad s pověstnou gramatikou

```
void match(token t) {
  if(lookahead==t)
    lookahead = nexttoken();
  else error();
}
void E(void) {
  T(); Eap();
}
void Eap(void) {
  if(lookahead=='+') {
    match('+'); T(); Eap();
  }
}
void T(void) {
  F(); Tap();
}
```

```
void Tap(void) {
  if(lookahead=='*') {
    match('*'); F(); Tap(); }
}
void F(void) {
  switch(lookahead) {
    case '(': match('('); E();
              match(')');break;
    case 'id':
       match('id'); break;
    default:
       error();
  }
}
```

Nerekurzivní analýza s predikcí – automat



- Parsovací tabulka M[A, a], kde A je neterminál a **a** je terminál
- Na zásobníku symboly gramatiky

Funkce automatu

- Počáteční konfigurace
  - Vstupní ukazatel ukazuje na začátek vstupu
  - Na zásobníku je počáteční neterminál gramatiky nad symbolem $
- V každém kroku se rozhoduji podle symbolu X na vrcholu zásobníku a terminálu **a**, který je právě na vstupu
  - Pokud je X=a=$, pak se parser s úspěchem zastaví
  - Pokud je X=a≠$, pak se vyzvedne X ze zásobníku a ukazatel vstupu se přesune o terminál dále
  - Je-li X neterminál, pak rozhodne položka M[X, a]. Pokud je tam přepisovací pravidlo, pak se nahradí na zásobníku X pravou stranou přepisovacího pravidla (s nejlevějším symbolem na vrcholu). Zároveň je generován výstup použití příslušného pravidla. Pokud je v tabulce **error**, pak se nahlásí chyba.

Konstrukce tabulky automatu

- Pro každé přepisovací pravidlo A→α gramatiky proveď následující kroky
  - Pro ∀ a∈FIRST(α) přidej A→α do M[A, a]
  - Pokud Λ∈FIRST(α), pak přidej A→α do M [A, b] ∀ b∈FOLLOW(A). Pokud navíc $∈FOLLOW(A), přidej A→α do M[A, $]
- Pro každé prázdné políčko M nastav **error**

Příklad konstrukce tabulky na pověstné gramatice

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→Λ	E'→Λ
T	T→FT'			T→FT'		
T'		T'→Λ	T'→*FT'		T'→Λ	T'→Λ
F	F→id			F→(E)		

Příklad funkce LL automatu na pověstné gramatice

Zásobník	Vstup	Výstup
$E	**id+id*id$**	
$E'T	**id+id*id$**	E→TE'
$E'T'F	**id+id*id$**	T→FT'
$E'T'**id**	**id+id*id$**	F→**id**
$E'T'	**+id*id$**	
$E'	**+id*id$**	T'→Λ
$E'T**+**	**+id*id$**	E'→**+**TE'
$E'T	**id*id$**	
$E'T'F	**id*id$**	T→FT'

Zásobník	Vstup	Výstup
$E'T'**id**	**id*id$**	F→**id**
$E'T'	***id$**	
$E'T'F*****	***id$**	T'→*FT'
$E'T'F	**id$**	
$E'T'**id**	**id$**	F→**id**
$E'T'	**$**	
$E'	**$**	T'→Λ
$	**$**	E'→Λ

LL(1) gramatika

- Bezkontextová gramatika G=(T,N,S,P) je LL(1) gramatika, pokud pro každá 2 pravidla A→α, A→β ∈ P, kde α≠β, a každé 2 levé větné formy uAγ, vAδ, kde u,v∈T* a γ,δ∈(T∪N)*, platí FIRST(αγ)∩FIRST(βδ)=∅.

Názvosloví gramatik

- PXY(k)
- X – směr čtení vstupu
  - V našem případě vždy L, tj. zleva doprava
- Y – druh derivace
  - L – levé derivace
  - R – pravé derivace
- P – prefix
  - Pro některé třídy gramatik ještě jemnější dělení na třídy
- k – výhled (lookahead)
  - Celé číslo, obvykle 1, ale také 0 nebo obecně k
- Příklady
  - LL(1), LR(0), LR(1), LL(k), SLR(1), LALR(1)

Rozšíření definic FIRST a FOLLOW na k

- Pokud je α řetězec symbolů gramatiky, pak $FIRST_k(\alpha)$ je množina slov terminálů o délce nejvýše k, kterými začínají všechny řetězce derivované z α. Pokud α může zderivovat na Λ, pak Λ je také ve $FIRST_k(\alpha)$.
- Definujme $FOLLOW_k(A)$ pro neterminál A jako množinu slov terminálů o délce nejvýše k, které se mohou vyskytovat těsně za A v nějakém řetězci, který vznikl derivací z počátečního neterminálu gramatiky ($S \Rightarrow^* \alpha A u \beta$, pro nějaká α a β). Pokud je A nejpravější symbol v nějakém přepisu, pak i $ je ve $FOLLOW_k(A)$.

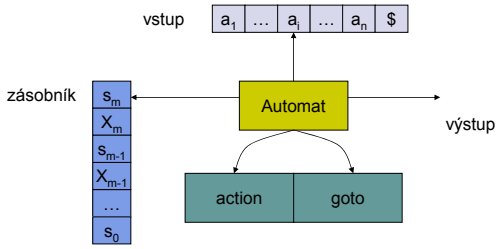
LL(k) gramatika

- Bezkontextová gramatika G=(T,N,S,P) je silná LL(k) gramatika pro k≥1, pokud pro každá 2 pravidla $A \to \alpha$, $A \to \beta \in P$, kde $\alpha \neq \beta$, a každé 2 levé větné formy $uA\gamma$, $vA\delta$, kde $u,v \in T^*$ a $\gamma,\delta \in (T \cup N)^*$, platí $FIRST_k(\alpha\gamma) \cap FIRST_k(\beta\delta) = \varnothing$.
- LL(k) (ne silná)
  - u=v, γ=δ

Analýza zdola nahoru

- Pokus najít pozpátku nejpravější derivaci pro vstupní řetězec
- Pokus zkonstruovat derivační strom pro daný vstup počínaje listy a stavěním zespodu až po kořen stromu.
- V redukčním kroku je podřetězec odpovídající pravé straně pravidla gramatiky nahrazen neterminálem z levé strany pravidla.
- Používáno známými generátory parserů
  - Bison – LALR(1), GLR(1)
- Výhody proti LL(1) parserům
  - Všechny programovací jazyky zapsatelné bezkontextovou gramatikou
  - Dá se implementovat stejně efektivně jako metody shora dolů
  - Třída rozpoznávaných jazyků LR(1) je vlastní nadmnožina LL(1)
- SLR(1), LR(1), LALR(1)

Automat pro LR parser



- s_i jsou stavy
  - Stav na vrcholu je aktuální stav automatu
- x_i jsou symboly gramatiky

Funkce LR automatu

- Počáteční konfigurace
  - Ukazatel vstupu na počátku vstupního slova
  - Na zásobníku je počáteční stav s_0
- V každém kroku podle s_m a a_i adresuji action[s_m, a_i]
  - Posun (shift) s, kde s je nový stav
    - Posune pásku o 1 terminál, na zásobník se přidá a_i a s
  - Redukce (reduction) podle pravidla gramatiky A→α
    - Zruší se ze zásobníku r=|α| dvojic (s_k, X_k), na zásobník se přidá A a goto[s_{m-r}, A] (s_{m-r} je stav, co zbyl na vrcholu zásobníku po odmazání)
    - Generuje výstup
  - Accept
    - Vstupní slovo je úspěšně rozpoznáno
    - Generuje výstup
  - Error
    - Vstupní slovo neodpovídá gramatice

Tabulky LR automatu pro pověstnou gramatiku

stav	action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Příklad funkce LR automatu na pověstné gramatice

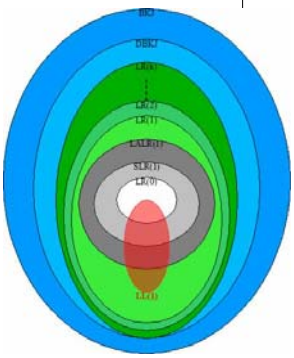
Zásobník	Vstup	Akce
0	**id+id*id$**	s5
0 **id** 5	**+id*id$**	r6: F→**id**
0 F 3	**+id*id$**	r4: T→F
0 T 2	**+id*id$**	r2: E→T
0 E 1	**+id*id$**	s6
0 E 1 + 6	**id*id$**	s5
0 E 1 + 6 **id** 5	***id$**	r6: F→**id**
0 E 1 + 6 F 3	***id$**	r4: T→F
0 E 1 + 6 T 9	***id$**	s7
0 E 1 + 6 T 9 * 7	**id$**	s5
0 E 1 + 6 T 9 * 7 **id** 5	**$**	r6: F→**id**
0 E 1 + 6 T 9 * 7 F 10	**$**	r3: T→T * F
0 E 1 + 6 T 9	**$**	r1: E→E + T
0 E 1	**$**	acc

LR(k) gramatika

- Bezkontextová gramatika G=(T,N,S,P) je LR(k) gramatika pro k≥1, pokud pro každá 2 pravidla A→α, A→β ∈ P, kde α≠β, a každé 2 pravé větné formy γAu, δAv, kde u,v∈T* a γ,δ∈(T∪N)*, platí $FIRST_k(u) \cap FIRST_k(v) = \emptyset$.

Síla gramatik

- Sjednocení všech LR(k) je DBKJ (deterministické BKJ)



Rozšíření gramatiky

- Mějme gramatiku G=(T,N,S,P). Rozšířením gramatiky G je gramatika G'=(T,N',S',P'), kde N'=N∪{S'}, P'=P∪{S'→S}
- Není třeba provádět, pokud S je na levé straně jednoho pravidla a není na žádné pravé straně pravidel
- Cílem je pomoci parseru s rozpoznáním konce parsování
- Pro pověstnou gramatiku:
  - S'→E

Otečkovaná pravidla

- Otečkované pravidlo gramatiky G je pravidlo, které má na pravé straně na nějaké pozici speciální symbol tečky
  - Speciální znamená, že stejné pravidlo s tečkou na různých pozicích na prave straně se chápe jako různá otečkovaná pravidla. Zároveň však tato tečka není terminálem ani neterminálem gramatiky
  - Otečkované pravidlo se také nazývá LR(0) položka
- Ukázka pro pravidlo E → E + T:

E → ♦E + T
E → E ♦+ T
E → E + ♦T
E → E + T♦

Operace uzávěru

- Mějme množinu otečkovaných pravidel I z gramatiky G. Definujme operaci CLOSURE(I) jako množinu otečkovaných pravidel zkonstruovaných z I následujícím postupem:
  - Přidej do CLOSURE(I) množinu I
  - ∀ A→α♦Bβ∈CLOSURE(I), kde B∈N, přidej ∀ B→γ∈P do CLOSURE(I) otečkované pravidlo B→♦γ, pokud tam ještě není. Toto opakuj tak dlouho, dokud přibývají pravidla do CLOSURE(I)

Příklad operace uzávěru na pověstné gramatice

- I={S'→♦E}
- CLOSURE(I)=
  - S'→ ♦E
  - E → ♦E + T
  - E → ♦T
  - T → ♦T * F
  - T → ♦F
  - F → ♦(E)
  - F → ♦id

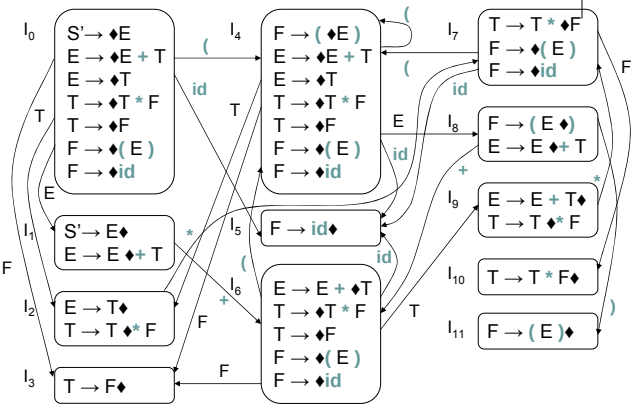
Operace přechodu

- Definujme operaci GOTO(I, X) pro množinu otečkovaných pravidel I a symbol gramatiky X jako uzávěr množiny všech pravidel A→αX♦β takových, že A→α♦Xβ∈I

Konstrukce kanonické kolekce množin LR(0) položek

- Mějme rozšířenou gramatiku G'=(T,N',S',P')
- Konstrukce kanonické kolekce C množin LR(0) položek:
  - Na počátku C={ CLOSURE({S'→♦S}) }
  - ∀ I∈C a ∀ X∈T∪N' takové, že GOTO(I, X)∉C ∧ GOTO(I, X)≠∅, přidej GOTO(I, X) do C. Toto opakuj, dokud něco přibývá do C

Konstrukce kanonické kolekce pro pověstnou gramatiku



Platné položky

- Otečkované pravidlo $A \rightarrow \beta_1 \blacklozenge \beta_2$ je platnou položkou pro schůdný (viable) prefix $\alpha\beta_1$, pokud $\exists$ pravá derivace $S' \Rightarrow^{+} \alpha A w \Rightarrow \alpha\beta_1\beta_2 w$
- Velká nápověda pro parser, zda provádět posun nebo redukci, pokud je na zásobníku $\alpha\beta_1$
- Základní věta LR parsování: Množina platných položek pro schůdný prefix γ je přesně množina položek dosažitelná z počátečního stavu přes cestu γ deterministickým konečným automatem zkonstruovaným z kanonické kolekce s přechody GOTO.

Konstrukce SLR(1) automatu

- Mějme rozšířenou gramatiku G'. Tabulky SLR(1) automatu se pak zkonstruují následujícím způsobem
  - Zkonstruuj kanonickou kolekci C LR(0) položek
  - Stav *i* je zkonstruován z I_i. Instrukce automatu pro stav *i* se určují podle následujících pravidel
    - $A \rightarrow \alpha \blacklozenge a\beta \in I_i, a \in T \wedge GOTO(I_i,a) = I_j$, pak action[i,a]=shift j
    - $A \rightarrow \alpha \blacklozenge \in I_i$, pak $\forall a \in FOLLOW(A) \wedge A \neq S'$ je action[i,a]=reduce $A \rightarrow \alpha$
    - $S' \rightarrow S \blacklozenge \in I_i$, pak action[i,$]=accept
  - Pokud v předchozím kroku nastal konflikt, gramatika není SLR(1) gramatikou a automat není možno zkonstruovat
  - Tabulka goto je pro stav *i* a $\forall A \in N'$: pokud $GOTO(I_i,A)=I_j$, pak goto [i,A]=j
  - Všechna nedefinovaná políčka se nastaví na error
  - Počáteční stav parseru je ten, jehož množina otečkovaných pravidel obsahuje $S' \rightarrow \blacklozenge S$

Skutečné LR(1) automaty

- Při konstrukci SLR(1) ve stavu *i* je nastaveno action[i,a] $\forall a \in FOLLOW(A)$ na redukci $A \rightarrow \alpha$, pokud $A \rightarrow \alpha \blacklozenge \in I_i$
- V některých situacích, kdy je *i* na vrcholu zásobníku, tak schůdný prefix βα je takový, že βA nemůže být následována terminálem **a** v žádné pravé větné formě. Tedy redukce podle $A \rightarrow \alpha$ je pro vstup a neplatná.
- Řešení: vložit do stavů více informace, abychom se vyhnuli neplatným redukcím.

LR(1) otečkované položky

- Tato informace navíc je uchována pro každé otečkované pravidlo ve stavu přidáním terminálu. Takové pravidlo pak má tvar $[A \rightarrow \alpha \blacklozenge \beta, a]$, kde $A \rightarrow \alpha\beta \in P$, $a \in T$. Otečkované pravidlo v tomto tvaru nazýváme LR(1) položka. Terminál **a** nazýváme výhled (lookahead).
  - Výhled nemá význam pro $A \rightarrow \alpha \blacklozenge \beta$, kde $\beta \neq \Lambda$
  - Redukce $A \rightarrow \alpha$ se provádí pouze tehdy, pokud $[A \rightarrow \alpha \blacklozenge, a] \in I_i$ pro aktuální stav *i* a terminál **a** na vstupu
  - Taková množina terminálů je $\subseteq FOLLOW(A)$
- LR(1) položka $[A \rightarrow \alpha \blacklozenge \beta, a]$ je platná pro schůdný prefix γ, pokud $\exists$ pravá derivace taková, že $S \Rightarrow^{+} \delta A w \Rightarrow \delta\alpha\beta w$, kde
  - $\gamma = \delta\alpha$
  - buď **a** je první symbol w nebo je $w = \Lambda$ a **a** je $

Upravená operace uzávěru pro LR(1) položky

- Mějme množinu LR(1) položek I pro gramatiku G. Definujme operaci CLOSURE1(I) jako množinu LR(1) položek zkonstruovaných z I následujícím postupem:
  - Přidej do CLOSURE1(I) množinu I
  - $\forall [A \rightarrow \alpha \blacklozenge B\beta, a] \in CLOSURE1(I)$, kde $B \in N$, přidej $\forall B \rightarrow \gamma \in P$ a $\forall b \in FIRST(\beta a)$ do CLOSURE1(I) LR(1) položku $[B \rightarrow \blacklozenge \gamma, b]$, pokud tam ještě není. Toto opakuj tak dlouho, dokud přibývají pravidla do CLOSURE1(I)

Upravená operace přechodu pro LR(1) položky

- Definujme operaci GOTO1(I, X) pro množinu LR(1) položek I a symbol gramatiky X jako CLOSURE1 množiny všech pravidel $[A \rightarrow \alpha X \blacklozenge \beta, a]$ takových, že $[A \rightarrow \alpha \blacklozenge X\beta, a] \in I$

Konstrukce kanonické kolekce LR(1) položek

- Mějme rozšířenou gramatiku G'=(T,N',S',P')
- Konstrukce kanonické kolekce C LR(1) položek:
  - Na počátku C={ CLOSURE1({[S'→♦S,$]}) }
  - $\forall I \in C$ a $\forall X \in T \cup N'$ takové, že $GOTO1(I, X) \notin C \wedge GOTO1(I, X) \neq \emptyset$, přidej GOTO1(I, X) do C. Toto opakuj, dokud něco přibývá do C

Příklad LR(1) gramatiky, která není SLR(1)



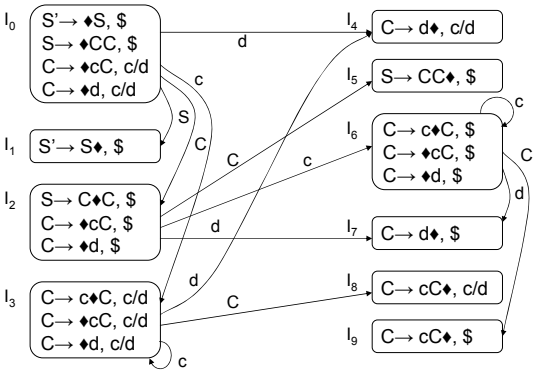
- S'→S
- S→CC
- C→cC
- C→d

Příklad konstrukce uzávěru pro LR(1) položky



- I={[S'→♦S,$]}
- CLOSURE1(I)=
  - S'→ ♦S, $ β=Λ,FIRST(β$)=FIRST($)={$}
  - S→ ♦CC, $ β=C,FIRST(C$)={c,d}
  - C→ ♦cC, c/d
  - C→ ♦d, c/d

Příklad konstrukce kanonické kolekce LR(1) položek



Konstrukce LR(1) parseru



- Mějme rozšířenou gramatiku G'. Tabulky LR(1) automatu se pak zkonstruují následujícím způsobem
  - Zkonstruuj kanonickou kolekci C LR(1) položek
  - Stav *i* je zkonstruován z I_i. Instrukce automatu pro stav *i* se určují podle následujících pravidel
    - [A→α♦aβ,b]∈I_i,a∈T ∧ GOTO1(I_i,a)=I_j, pak action[i,a]=shift j
    - [A→α♦,a]∈I_i ∧ A≠S', pak action[i,a]=reduce A→α
    - [S'→S♦,$]∈$I_i$, pak action[i,$]=accept
  - Pokud v předchozím kroku nastal konflikt, gramatika není LR(1) gramatikou a automat není možno zkonstruovat
  - Tabulka goto je pro stav *i* a ∀A∈N': pokud GOTO1(I_i,A)=I_j, pak goto [i,A]=j
  - Všechna nedefinovaná políčka se nastaví na error
  - Počáteční stav parseru je ten, jehož množina LR(1) položek obsahuje [S'→♦S,$]

LALR



- LALR=LookAhead-LR
- Běžně využívána v praxi
  - Bison
  - Většina programovacích jazyků se dá vyjádřit jako LALR
  - Tabulky parseru jsou významně menší než LR(1)
- SLR a LALR parsery mají stejný počet stavů, LR parsery mají větší počet stavů
  - Pro programovací jazyky řádově stovky stavů
  - LR(1) tabulky budou mít řádově tisíce stavů pro stejnou gramatiku

Jak zmenšit tabulky?



- Nápad: sloučit množiny se stejným jádrem do jedné množiny včetně sloučení GOTO1
  - Jádro: množina otečkovaných pravidel bez výhledu (LR(0) položky)
  - Při tomto slučování nemůže vzniknout shift/reduce konflikt
    - Mějme následující dvě konfliktní pravidla: [A→α♦,a] a [B→β♦aγ,b]
    - Jádra množin jsou stejná, takže v množině, kde je [A→α♦,a] musí být i [B→β♦aγ,c] pro nějaké **c**, takže už tam konflikt shift/reduce byl
  - Může vzniknout reduce/reduce konflikt

Snadná metoda konstrukce LALR(1) parseru



- Mějme rozšířenou gramatiku G'. Tabulky LALR(1) automatu se pak zkonstruují následujícím způsobem
  - Zkonstruuj kanonickou kolekci C LR(1) položek
  - Pro každé jádro v kolekci C najdi všechny množiny s tímto jádrem a nahraď je jejich sjednocením
  - Nechť C'={ J_0, J_1, …, J_m } je výsledná kolekce LR(1) pravidel
  - Tabulka action je zkonstruována pro C' stejně jako pro LR(1) parser
  - Pokud nastal konflikt, gramatika není LALR(1) gramatikou
  - Nechť množina J∈C' vznikla sjednocením několika položek I_i (J=I_1∪I_2∪…I_k). Pak jádra GOTO1(I_1,X), …, GOTO1(I_k,X) jsou stejná, protože i I_1, …, I_k mají stejná jádra. Nechť K je sjednocení všech množin položek, které mají stejné jádro jako goto(I_1,X). Pak GOTO1(J,X)=K
- Významná nevýhoda – je třeba zkonstruovat plné LR(1)

Principy překladačů

Sémantická analýza



Jakub Yaghob

Syntaxí řízené definice

- Každé symbol gramatiky má definovanou množinu atributů
  - Lze to chápat jako záznam
  - Atributy dvojího druhu
    - Syntetizované
    - Dědičné
  - Atributy mohou reprezentovat cokoliv
- Hodnoty atributů definovány sémantickými pravidly přidaných k přepisovacím pravidlům gramatiky
  - Podle grafu závislosti se určí pořadí vyhodnocení sémantických pravidel
  - Vyhodnocení sémantických pravidel určí hodnoty atributů

Druhy atributů

- Každé přepisovací pravidlo A→α má definovanou množinu sémantických pravidel ve formě $b=f(c_1,\ldots,c_k)$, kde f je funkce a c_i jsou atributy symbolů gramatiky daného přepisovacího pravidla
  - b je syntetizovaný atribut neterminálu A
  - b je dědičný atribut symbolu gramatiky z pravé strany daného přepisovacího pravidla

Atributová gramatika

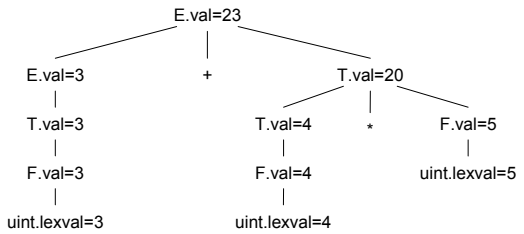
- Derivační strom s hodnotami atributů se nazývá anotovaný (obarvený) derivační strom
- Atributová gramatika je syntaxí řízená definice, kde funkce v sémantických pravidlech nemají vedlejší efekty

Atributová gramatika pro pověstnou gramatiku

1.	$E \rightarrow E_R + T$	$E.val = E_R.val + T.val$
2.	$E \rightarrow T$	$E.val = T.val$
3.	$T \rightarrow T_R * F$	$T.val = T_R.val * F.val$
4.	$T \rightarrow F$	$T.val = F.val$
5.	$F \rightarrow (E)$	$F.val = E.val$
6.	$F \rightarrow$ uint	$F.val =$ uint.lexval

Příklad anotovaného derivačního stromu

- 3+4*5



Syntetizované atributy

- Atribut neterminálu z levé strany přepisovacího pravidla je vypočten na základě atributů symbolů z pravé strany
- Běžně používány v praxi
  - Např. výpočet výrazu
- Atributová gramatika, která používá pouze syntetizované atributy, se nazývá S-atributová gramatika (čistě syntetizovaná atributová gramatika)
- S-atributová gramatika se snadno použije při analýze zdola-nahoru
  - Výpočet proběhne při redukci
  - Atributy pro symbol jsou na zásobníku parseru

Dědičné atributy

- Hodnota dědičného atributu je vypočtena na základě atributů otce nebo atributů symbolů na stejné úrovni v derivačním stromě
- Používají se k vyjádření závislosti nějaké syntaktické konstrukce na kontextu
  - Např. zda se identifikátor vyskytuje na levé nebo pravé straně přiřazení
- Vždy lze přepsat atributovou gramatiku tak, aby byla S-atributová

Příklad dědičných atributů

1.	$D \rightarrow T L$;	$L.in = T.typ$
2.	$T \rightarrow$ int	$T.typ =$ int
3.	$T \rightarrow$ double	$T.typ =$ double
4.	$L \rightarrow L_R$, id	$L_R.in = L.in$
		$id.typ = L.in$
5.	$L \rightarrow$ id	$id.typ = L.in$

Graf závislosti



- Konstrukce grafu závislosti
  - Uzly jsou vytvořeny pro každý atribut každého uzlu anotovaného derivačního stromu
  - Pro každé sémantické pravidlo $b=f(c_1,\ldots,c_k)$ sestroj orientovanou hranu z uzlu grafu závislosti pro c_i do uzlu pro b

Pořadí výpočtu



- Najdeme topologické třídění DAGu, který vznikl konstrukcí grafu závislosti
  - Pokud vznikne při konstrukci grafu závislosti kružnice, pak nelze určit pořadí výpočtu
  - Jakékoliv topologické setřídění je dobré

L-atributová gramatika



- Jinak také známa jako jednoduše zleva-doprava 1-průchodová gramatika
- Atributová gramatika je L-atributová, pokud $\forall$ dědičný atribut symbolu X_j z $A \rightarrow X_1 \ldots X_n$ závisí pouze na
  - Atributech symbolů $X_1,\ldots,X_{j-1}$ (vlevo od X_j)
  - Dědičných atributech A
- Lze využít pro výpočet atributů přímo při analýze shora-dolů
- Každá S-atributová gramatika je L-atributová

Průchody derivačním stromem



- Pokud atributová gramatika není L-atributová, není možné provést výpočet atributů přímo při parsování
- Je nutné samostatně zkonstruovat derivační strom a jako jedna z částí sémantické analýzy budou i průchody tímto derivačním stromem, které dopočítají atributy, které nešly spočítat během syntaxí řízeného překladu

Statická kontrola při překladu



- Kontrola typů
  - Použití operátorů na špatný typ
    - Násobení ukazatelů
- Kontrola toku řízení
  - Zda je změna toku řízení oprávněná
    - V C příkaz break, pokud nejsem ve switch nebo cyklu
- Kontrola jedinečnosti
  - Některé objekty mohou být definovány právě jednou
    - Návěští ve funkci, identifikátory globálních objektů
- Kontrola jmen
  - Některé konstrukty mají mít stejné jméno na svém začátku i na konci
    - Procedura v assembleru

Symbolické tabulky



- Tabulky, které stále rostou
  - Konstanty
- Tabulky, které přibývají/ubývají během překladu
  - Viditelnost identifikátorů v blocích
  - Jednoduchá implementace
    - Spojový seznam viditelností pro blok (zásobník)
  - Používaná implementace
    - „Obarvení“ identifikátorů jednoznačným číslem bloku

Chyby při překladu



- Překladač musí najít všechny chyby ve vstupním slově a nehlásit chyby, které tam nejsou
- Hlášení chyb
  - Hlásit srozumitelně a přesně
  - Zotavit se z chyb rychle a pokračovat v překladu
  - Nesmí znatelně zpomalovat překlad korektních vstupů
- Zavlečené chyby
  - Vzniknou nepřesným zotavením z předchozí chyby na místech, které původně chyby nebyly

Druhy chyb



- Lexikální chyby
  - Špatně definované lexikální elementy
    - Neukončený řetězec na konci řádku
  - Zotavení ignorováním
- Syntaktické chyby
  - Vstupní slovo není v jazyce definovaném gramatikou
    - Nespárované závorky
- Sémantické chyby
  - Odpovídají statickým kontrolám
    - Neexistující proměnná, špatný počet parametrů volání funkce, špatně použitý operátor na nějaký typ
- Logické chyby
  - Chyby programátora
    - Nekonečný cyklus, neinicializované proměnné

Zotavení ze syntaktických chyb

- Panický režim
  - Množina skeletálních (záchytných) symbolů
  - Při chybě se přeskakují všechny symboly, dokud se nenarazí na skeletální symbol
  - Pak se parser převede do známého stavu
- Úprava pravidel
  - Přidání, ubrání, nahrazení terminálu v přepisovacím pravidle
- Chybné přepisovací pravidlo
  - Zvětšení gramatiky o obvyklé chyby s přidáním chybového hlášení
    - Např. přiřazení v Pascalu

Principy překladačů

Generování mezikódu



Jakub Yaghob

Co už známe

- Typy instrukcí tříadresového kódu
- Implementace
  - Čtveřice
  - Trojice, nepřímé trojice
- Graf toku řízení
- Základní blok
- Životnost jména

Jak generovat mezikód v syntaxí řízeném překladu

- Přidáme ke každému symbolu gramatiky atributy obsahující informace o mezikódu
  - Umístění (place) – jméno objektu, který obsahuje hodnotu symbolu
  - Kód (code) – posloupnost tříadresových instrukcí, která „vypočítá“ symbol
  - Adresa do kódu (label) – absolutní nebo relativní adresa do tříadresových instrukcí

Příklad pro pověstnou gramatiku

$E \rightarrow E_R + T$	E.p = newtemp E.c = E_R.c | T.c | gen(E.p=E_R.p + T.p)
$E \rightarrow T$	E.p = T.p E.c = T.c
$T \rightarrow T_R * F$	T.p = newtemp T.c = T_R.c | F.c | gen(T.p = T_R.p * F.p)
$T \rightarrow F$	T.p = F.p T.c = F.c
$F \rightarrow (E)$	F.p = E.p F.c = E.c
$F \rightarrow id$	F.p = id.p F.c = ‘’

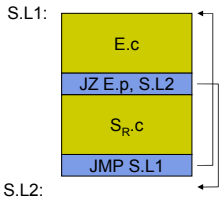
Příklad pro while (amatér)

- S→ **while** E **do** S_R

S.L1 = curradr

S.L2 = curradr + E.c.size + S_R.c.size + 2

S.c = E.c \| gen(JZ E.p, S.L2) \| S_R.c \| gen(JMP S.L1)



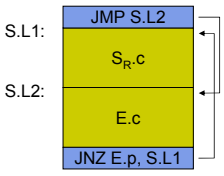
Příklad pro while (odborník)

- S→ **while** E **do** S_R

S.L1 = curradr + 1

S.L2 = curradr + S_R.c.size + 1

S.c = gen(JMP S.L2) \| S_R.c \| E.c \| gen(JNZ E.p, S.L1)



Deklarace

- Deklarace globálních objektů
  - Např. globální proměnné v C
- Deklarace lokálních objektů
  - Např. lokální proměnné v rámci funkce
  - Na začátku funkce x na začátku bloku x v průběhu bloku
- Přesun deklarací na „dobré“ místo
  - Globální na jednom předdefinovaném místě
  - Lokální na začátek funkce
    - Životnost proměnné v rámci funkce
- Určení velikosti objektu
- Určení offsetu ve struktuře pro její položky
  - Nemusí být řešeno v mezikódu

Přiřazení



- Konverze mezi typy
  - Obvykle explicitní
    - Mezikód nezná sémantiku vstupního jazyka
- Přístupy k položkám struktury
  - Využití vypočtených posunutí z deklarace
  - Přímý výpočet přes ukazatel a přičtenou konstantu
- Rozvinutí polí
  - Vícerozměrná pole se chápou jako jednorozměrná (stejně jako paměť) – obvykle rozvinutí řádka za řádkou
  - Pro jednorozměrné pole A[lb..ub] typu o šířce w
    - base + (i-lb) * w
    - i * w + (base – lb * w)
  - Dvojrozměrné pole $A[lb_1..ub_1,lb_2..ub_2]$
    - $base + ((i_1-lb_1) * (ub_2-lb_2+1) + i_2-lb_2) * w$

Booleovské výrazy



- Někdy nahrazení FALSE=0, TRUE=1 (nebo cokoliv !=0)
- Numerický (plný) výpočet
  - Všechny části výrazu vypočteny a vyhodnoceny booleovskou aritmetikou
  - Pascal
- Zkrácený výpočet
  - Jakmile je z výpočtu jasné, jaký bude výsledek, dále se nepočítá
    - Skoky ve výpočtu
  - C

Příkaz větvení (switch)



- Co potřebuji
  - Vyhodnotit výraz, podle kterého se větvím
  - Najít větev podle vypočtené hodnoty
  - Provést nalezenou větev nebo defaultní větev/nic
- Nalezení větve
  - Lineární sekvence podmínek
    - Velmi málo větví
  - Binární vyhledávání v tabulce [hodnota,vetevptr]
  - Binární strom podmínek
  - Tabulka ukazatelů indexovaná hodnotou
    - Velká hustota hodnot ve zkoumaném rozsahu

Backpatching



- Problém jednoprůchodového překladu: jak nastavit adresy skoků na ještě nepotkané objekty?
  - Popředný skok
  - Volání ještě nedefinované funkce v rámci jednoho modulu
    - Může být řešeno i linkerem
- Zapamatování si seznamu instrukcí mezikódu k ještě neznámému objektu
- Až daný objekt potkám, vyhodnotím jeho adresu v mezikódu
- Projdu seznam instrukcí a provedu opravu mezikódu

Volání procedur/funkcí



- Vyhodnocení parametrů
- Předání hodnot nebo referencí
  - V Pascalu „nevařené“ nebo „vařené“ parametry
- Svázání skutečných parametrů s formálními parametry
  - Pozičně
    - Může záležet na pořadí předávání parametrů
  - Jménem
- Návratová hodnota

Principy překladačů

Vysokoúrovňové optimalizace



Jakub Yaghob

Optimalizace



- Ideální případ – generovaný kód je stejně dobrý jako ručně psaný
  - Platí to dnes ještě?
- Skutečnost
  - Pouze v několika vymezených situacích
  - Obtížné
- Optimalizace
  - Transformace programu
  - Na rychlost nebo na velikost kódu
- Vysokoúrovňové optimalizace
  - Prováděné nad mezikódem

Kritéria pro transformace



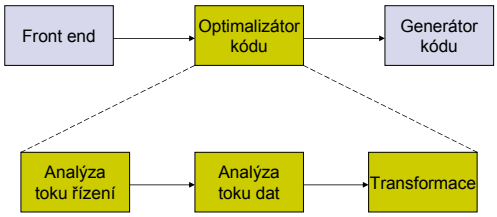
- Zachování smyslu programu
  - Výstup a chování programu nesmí být transformací změněny
- Zrychlení o měřitelnou velikost
  - Obvykle transformace zvyšující rychlost
  - Někdy optimalizace zpomalí, i když v průměrném případě zrychluje
- Transformace musí být hodna vynaložené námaze
  - Složitá/pomalá optimalizace s malým zrychlením není moc užitečná

Druhy optimalizací

- V rámci jednoho základního bloku
  - Nejsou žádné skoky
- V rámci jedné funkce
  - Pohyb kódu mezi bloky
  - Vznikají, zanikají základní bloky
- V rámci celého programu (interprocedurální)
  - Zrychlení volání mezi funkcemi
  - Provádí se ve fázi linkování zpožděným překladem

Organizace optimalizátoru

- Analýza toku řízení
  - Konstrukce grafu toku řízení
- Analýza toku dat
  - Výpočet životnosti proměnných



Lokální optimalizace

- V rámci jednoho základního bloku
  - Eliminace společného podvýrazu (common subexpression elimination)
  - Propagování kopií (copy propagation)
  - Eliminace mrtvého kódu (dead-code elimination)
    - x = y+z, a x již není nikde dál použitá
  - Výpočet konstantních výrazů (constant folding)
  - Algebraické transformace
    - x = x + 0, x = x * 1

Lokální eliminace společného podvýrazu

- a = b + c
- b = a – d
- c = b + c
- d = a – d

- a = b + c
- b = a – d
- c = b + c
- d = b

- Je tady schovaný problém
  - Pointer aliasing
  - ap = &a
  - b = a + c
  - *ap = d
  - e = a + c

Analýza toku dat

- „Obarvení“ uzlů grafu toku řízení (základních bloků) proměnnými, které jsou v tomto uzlu živé
  - Nemusí být živé v celém základním bloku
  - Uvnitř základního bloku se dopočítá přesná životnost
- Výpočet pořadí vyhodnocení
  - Topologické setřídění DAGu, který popisuje definici proměnné jako cíl hran uzlů použitých pro výpočet
- Modelování ukazatelů
  - Pointer aliasing
- Modelování volání funkcí
  - Interprocedurální optimalizace

Globální optimalizace

- Více základních bloků, možný pohyb kódu mezi bloky, příp. i změna grafu toku řízení
  - Eliminace společného podvýrazu (common subexpression elimination)
  - Propagování kopií (copy propagation)
  - Eliminace mrtvého kódu (dead-code elimination)
  - Optimalizace cyklů
    - Přesun invariantního kódu
    - Redukce síly operace
    - Odstranění indukční proměnné

Optimalizace cyklů – 1

- Přesun invariantního kódu
  - Výpočet, který se v cyklu nemění se předpočítá před cyklem

```
while(i<limit-5) S
```

```
t = limit – 5;
while(i<t) S
```

Optimalizace cyklů – 2

- Redukce síly operace
  - Typicky operace násobení převedená na sčítání
- Odstranění indukční proměnné
  - Stačí pouze jediná indukční proměnná cyklu
  - Obvykle se podaří zrušit při redukci

```
for(i=3;i<8;i+=3)
{
  j = i*2;
  a[j] = j;
}
```

```
for(t=6;t<16;t+=6)
  a[t] = t;
```

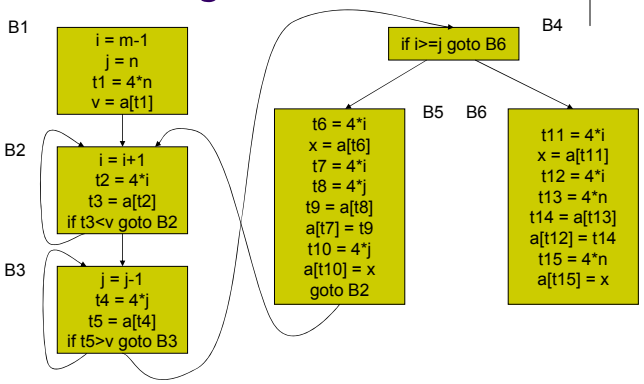
Příklad – kód v C

```
i=m-1; j=n; v=a[n];
for(;;) {
  do ++i; while(a[i]<v);
  do --j; while(a[j]>v);
  if(i>=j) break;
  x=a[i]; a[i]=a[j]; a[j]=x;
}
x=a[i]; a[i]=a[n]; a[n]=x;
```

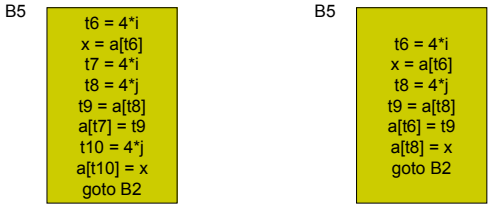
Příklad – třiadresový kód

1) i = m-1
2) j = n
3) t1 = 4*n
4) v = a[t1]
5) i = i+1
6) t2 = 4*i
7) t3 = a[t2]
8) if t3<v goto 5
9) j = j-1
10) t4 = 4*j
11) t5 = a[t4]
12) if t5>v goto 9
13) if i>=j goto 23
14) t6 = 4*i
15) x = a[t6]
16) t7 = 4*i
17) t8 = 4*j
18) t9 = a[t8]
19) a[t7] = t9
20) t10 = 4*j
21) a[t10] = x
22) goto 5
23) t11 = 4*i
24) x = a[t11]
25) t12 = 4*i
26) t13 = 4*n
27) t14 = a[t13]
28) a[t12] = t14
29) t15 = 4*n
30) a[t15] = x

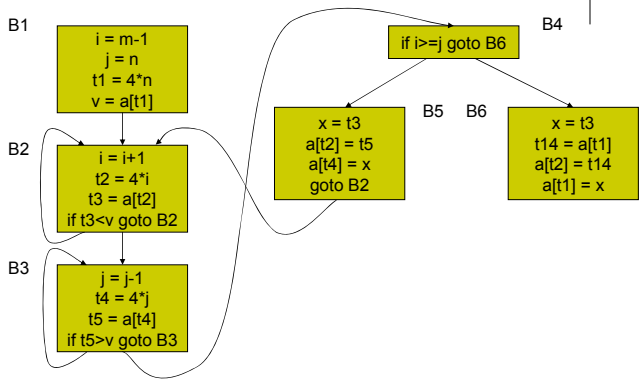
Příklad – graf toku řízení



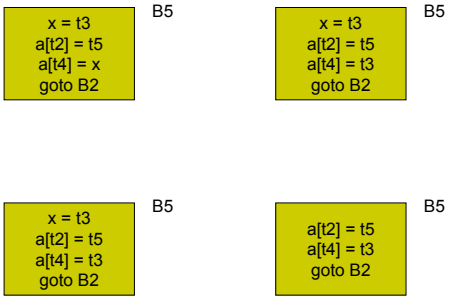
Příklad - LCSE



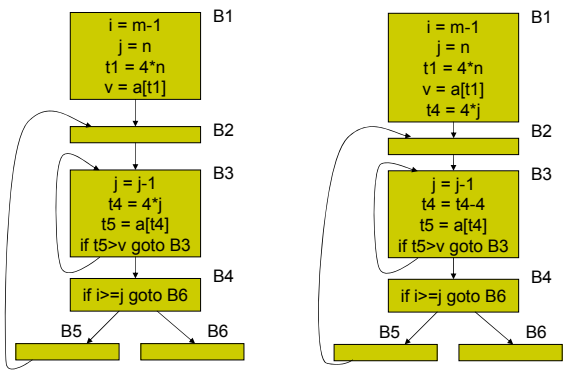
Příklad – GCSE



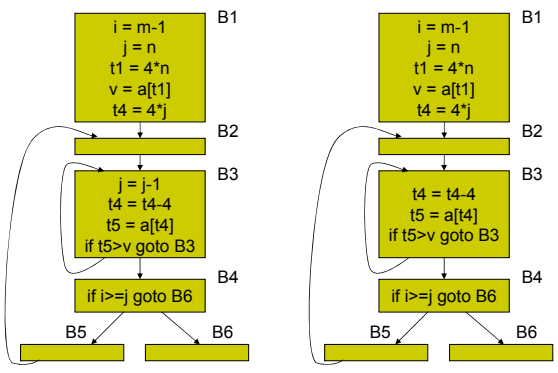
Příklad – copy propagation a dead-code elimination



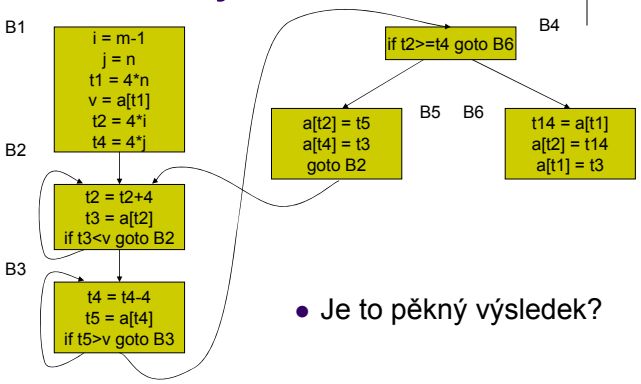
Příklad – redukce síly operace



Příklad – odstranění indukční proměnné

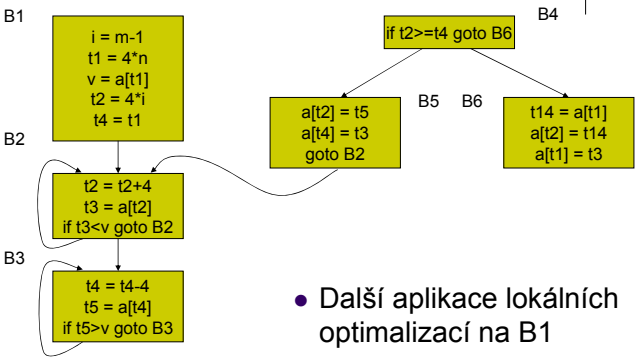


Příklad – výsledek



- Je to pěkný výsledek?

Příklad – pořádný výsledek



- Další aplikace lokálních optimalizací na B1

Paralelizace a vektorizace

- Paralelizace
  - Implicitní x explicitní
  - Části kódu, které lze vykonávat současně
  - Přednačítání proměnných do cache
- Vektorizace
  - Paralelizace výpočtu s použitím SIMD operací

Optimalizace řízená profilem (Profile Guided Optimization)

- Tři fáze
  - Instrumentace
    - Speciálně přeložený kód s voláním „sběrných“ funkcí na konci a začátku základních bloků
  - Profilování
    - Spouštění instrumentovaného programu s „typickými“ vstupy
    - Sběr statistických dat o průchodu kódu „sběrnými“ funkcemi
  - Optimalizace
    - Nasbíraná statistická data ovlivní optimalizaci
    - Využití při optimalizaci v mezikódu i kódu
    - Změření váhy cest v grafu toku řízení
    - Přidělování registrů

Principy překladačů

Architektury procesorů

Jakub Yaghob

Architektury procesorů

- Architektura procesoru představuje cílový jazyk
  - Platí pro překladače do kódu konkrétního procesoru
  - Ovlivňuje celý backend překladače, zejména generátor kódu
  - Může ovlivnit i tvar a instrukce mezikódu, nasazené druhy optimalizací nad mezikódem

Registry

- Nejrychlejší paměť
- Malý počet
  - x86 má 7 celočíselných 32-bitových
  - IA-64 má 128 celočíselných 64-bitových
- Různé druhy pro různé typy dat
  - Celočíselné, floatové, adresové, vektorové
- Různé druhy přístupu
  - Přímý
  - Zásobník (FPU)

Instrukční sada

- RISC
  - Jednoduché instrukce, malý počet (např. bez dělení)
- CISC
  - Složité instrukce, velký repertoár
- Load-Execute-Store
- Ortogonalita
  - x86 má neortogonální instrukční sadu

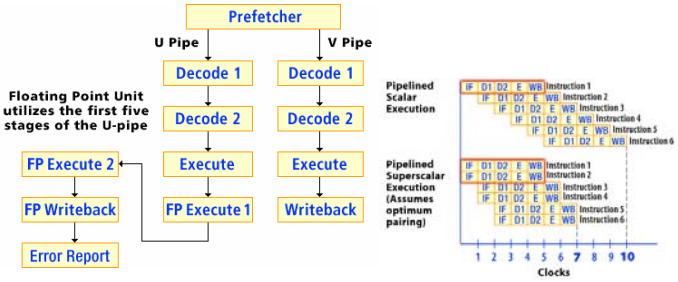
Pipelining

- Zahájení čtení a dekódování další instrukce před dokončením vykonávání předchozí instrukce
- Každá fáze a každá instrukce má svoje zpoždění (latency).
  - Problém při závislosti operandů (RAW)



Superscalarita

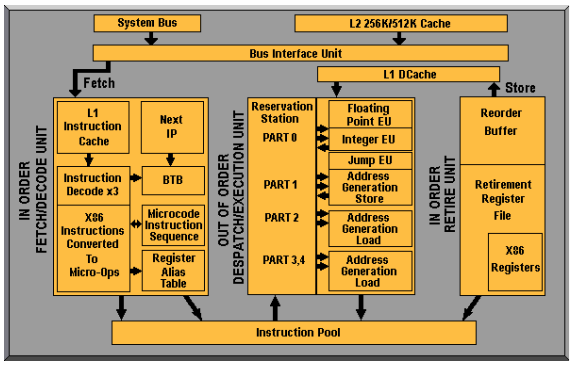
- Více stejných jednotek schopných paralelního vykonávání instrukcí



Out-of-Order Execution – 1 (vykonávání mimo pořadí)

- Načtení instrukce
- Zařazení do vyčkávací stanice
- Instrukce čeká na všechny svoje operandy
- Instrukce se vykoná ve své výkonné jednotce
- Výsledky se uchovají ve frontě
- Až se všechny starší instrukce zapíší do registrů, zapíše se výsledek této instrukce

Out-of-Order Execution – 2 (vykonávání mimo pořadí)



Predikce skoků

- Hluboké pipeliny mají problém, pokud podmíněný skok není proveden
- Dynamická predikce skoků
  - BTB
  - CPU podle historie (vzory nějaké hloubky) odhaduje, jestli skok bude proveden nebo nebude
  - Pokud není žádná historie, použije se statická predikce
- Statická predikce skoků
  - Bez nápovědy
    - Skok vpřed se neprovede, skok vzad se provede
  - S nápovědou
    - Překladač odhaduje pravděpodobnost skoku
- Delay slot

Spekulativní vykonávání

- Vykonávání kódu, který nemusí být zapotřebí
- Významná disproporce mezi rychlostí CPU a paměti
- Typické využití je značné předsunutí čtecích operací
- CPU provádí i odsouvání zápisových operací
- Paměťové bariéry
- Kódová spekulace
  - Kontrola, zda se provedla
- Datová spekulace
  - Předsunutí i v případě nevyjasněného pointer-aliasingu

SIMD instrukce

- Někdy také multimediální instrukce
- Celočíselné i floatové datové typy
- Vektorizace výpočtu už v mezikódu nebo až při generování kódu
- Značné zrychlení výpočtu
- Obtížná detekce možnosti vektorizovat

VLIW

- Kódování instrukcí
  - Šablony IA-64
- ILP (Instruction Level Parallelism)
  - Paralelismus zakódován přímo v instrukci
  - Závislosti v paralelní skupině
    - Zákaz RAW, WAW
    - Povolení WAR

Jednočipy



- Malý paměťový prostor
  - Optimalizace na velikost kódu i dat
    - Bitové proměnné
- Oddělené paměťové prostory
  - Kód a několik datových prostorů
  - Různý přístup
- Jednodušší pipeline, žádná superskalarita

Generování kódu



- Přidělování paměti
- Výběr instrukcí
- Přidělování registrů
- Instrukční scheduling
- Generování výsledku
  - Dohodnutý formát souboru
  - Objecty
    - Kód, data, relokace, externí a public symboly, ladící informace

Přidělování paměti



- Kód
  - Řešení skoků
- Statická oblast dat
  - Globální data
- Zásobník
  - Lokální proměnné a parametry funkcí
- Velikost datových typů
- Umístění v paměti
- Zarovnání dat
- Co bude v paměti a co v registrech

Výběr instrukcí



- 1:N
  - Z jedné instrukce mezikódu se generuje N instrukcí
  - Jednoduché, ale ne příliš dobrý kód
- M:N
  - Z M instrukcí mezikódu N instrukcí kódu
  - NP-úplný problém výběru
  - Heuristiky
- Více cest, jak stejnou věc vygenerovat
  - CISC, SIMD, VLIW
- Problémy s neortogonální sadou instrukcí

Přidělování registrů



- Co bude v paměti a co v registrech a v jakých
  - Typy registrů
- Problémy s neortogonální sadou instrukcí
- Obarvení grafu životnosti proměnných ve funkci
  - Graf je pouze pro daný typ proměnných (např. celočíselné)
  - Počet barev je počet registrů pro daný typ
  - Pokud nejde obarvit, odstraní se uzly, které nejsou moc „důležité“ a zkusí se to znovu
  - Není třeba najít ideální obarvení největšího počtu uzlu zadaným počtem barev, ale nějaké obarvení

Instrukční scheduling



- Přeházení instrukcí, aby zachovávalo sémantiku a nedocházelo k zastavení pipeline
  - Závislosti RAW, WAW, WAR
  - Sestrojení grafu závislosti dat
    - Orientovaný graf, kde hrana říká, která instrukce musí předcházet jinou instrukci, aby data byla k dispozici
    - Nalezení topologického třídění
  - Latence instrukcí, spekulace, superskalarita, out-of-order execution
  - Obvykle se provádí nad jedním základním blokem

Principy překladačů

Běhová podpora



Jakub Yaghob

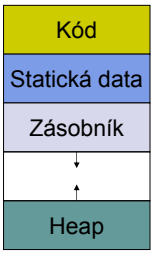
Běhová podpora

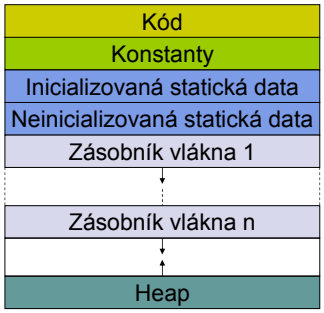


- Statická podpora jazyka
  - Překladač
  - Interface na knihovny
    - Hlavičkové soubory
- Dynamická podpora jazyka
  - Prostředí běhu programu
    - Rozdělení paměti
    - Stav paměti před spuštěním
    - Konstruktory, destruktory globálních objektů
  - Knihovny
  - Volací konvence

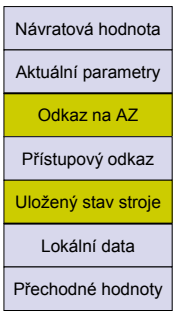
Rozdělení paměti

- Rozdělení paměti za běhu v procedurálních programech





Aktivační záznam

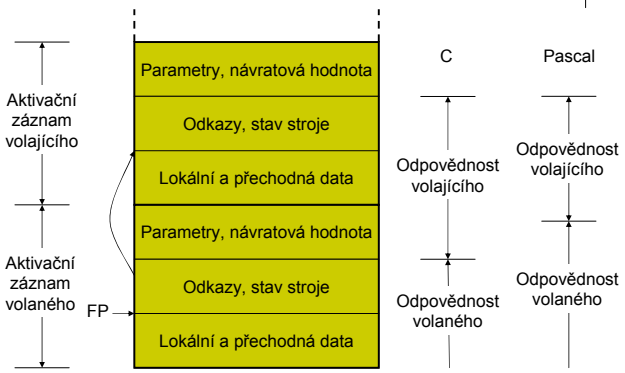


- Odkaz na AZ
  - AZ volající funkce
- Přístupový odkaz
  - Odkaz na data sémanticky nadřazených funkcí
- Uložený stav stroje
  - Návratová adresa do kódu
  - Registry

Volací konvence

- Volací konvence
  - Mandlování veřejných jmen
  - Mechanismus volání funkcí a procedur
    - Zodpovědnost za úklid
  - Způsob předávání parametrů
    - Registry, zásobník
    - Pořadí ukládání parametrů na zásobník
  - Způsob předávání návratové hodnoty funkcí
    - Registry, zásobník
  - Požadavky na uchování stavu registrů
    - Registry pro registrové proměnné

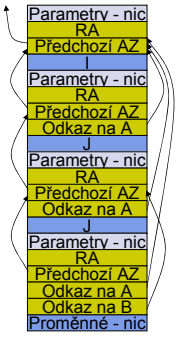
Mechanismus volání funkcí



Přístup k sémanticky nadřazeným proměnným

- Pascal
  - Vnořené funkce

```
procedure A;
var I:integer;
  procedure B;
  var J:integer;
      procedure C;
      begin …I+J… end;
  begin
      if I>0 B else C;
  end;
begin B end;
```



Předávání parametrů

- Předávání hodnotou
  - Aktuální hodnota parametru vypočtena a okopírována
  - Vstupní parametry, chová se jako lokální proměnná
  - C, „nevařené“ parametry v Pascalu
- Předávání referencí
  - Volající předá adresu na místo v paměti
  - Vstupně/výstupní parametry
  - & v C++, „vařené“ parametry v Pascalu
- Předávání jménem
  - Chová se jako makro – aktuální výraz se „inlinuje“ na místech použití

Dynamická paměť

- Alokační algoritmy
  - Spojité bloky volné velikosti
- Garbage collector
  - Implicitní dealokace
  - Čítače odkazů
  - Značkování
    - V jistém okamžiku se zastaví běh programu
    - Všechny ukazatele musí být známy včetně typů
    - Všechny bloky se označí jako nepoužité
    - Prochází se rekurzivně ukazatele a značkují se použité bloky
    - Nepoužité se odstraní
    - Lze i setřepat

Principy překladačů

Interpretované jazyky

Jakub Yaghob

Co to je?



- Zdrojový jazyk se nepřekládá do kódu skutečného procesoru, ale do kódu nějakého abstraktního stroje
- Interpret přeložený do kódu skutečného stroje simuluje zvolený abstraktní stroj

Proč?



- Není prostor pro překladač
  - 8-bitové počítače a Basic
- Přenositelnost
  - Stejný abstraktní stroj může běžet na různých OS i různých architekturách CPU
  - AS/400, Java
- Bezpečnost
  - Větší kontrola nad vykonávanými instrukcemi

Problémy



- Problém s rychlostí
  - Dá se řešit pomocí JIT (Just-In-Time compilation)
    - Pokud interpret narazí na kód abstraktního stroje, který ještě není přeložen, okamžitě ho přeloží na kód cílového stroje a uloží si ho vedle do své cache
- Problémy s přenositelností
  - Nevhodné změny v chování abstraktního stroje mohou přivodit problémy s přenositelností
    - Java
- Jak zvolit abstraktní stroj
  - Aby pokryl chování všech zdrojových jazyků
    - .NET

Dynamická paměť



- Pokud podporována, pak výhradně s garbage collectorem
  - Ukazatele jsou pod kontrolou
  - Snadnější programování
  - Rychlejší práce s dynamickou pamětí
    - Program obvykle nepotřebuje tolik paměti, aby GC vůbec musel zasahovat, takže se pouze souvisle alokuje
    - Simulátor abstraktního ale obvykle zabere více paměti, než by musel