

## Implementace Prologu (pokračování)

### Interpret – Základní princip

1. Vyber redukovaný literál („první“, tj. nejlevější literál cíle)
2. Lineárním průchodem od začátku databáze najdi klauzuli, jejíž hlava má stejný funktor a stejný počet argumentů jako redukovaný literál
3. V případě nalezení klauzule založ bod volby procedury
4. Založ dále okolí první klauzule (velikost odvozena od počtu lokálních proměnných v klauzuli)
5. Proved' unifikaci literálu a hlavy klauzule
6. Úspěch  $\Rightarrow$  přidej všechny literály klauzule k cíli („doleva“, tj. na místo redukovaného literálu).  
Tělo prázdné  $\Rightarrow$  výpočet se s úspěchem vrací do klauzule, jejíž adresa je v aktuálním okolí.
7. Neúspěch unifikace  $\Rightarrow$  z bodu volby se obnoví stav a pokračuje se v hledání další vhodné klauzule v databázi.
8. Pokud není nalezena odpovídající klauzule, výpočet se vrací na předchozí bod volby (krátí se lokální i globální zásobník).
9. Výpočet končí neúspěchem: neexistuje již bod volby, k němuž by se výpočet mohl vrátit.
10. Výpočet končí úspěchem, jsou-li úspěšně redukovány všechny literály v cíli.

## Interpret Prologu

Základní principy:

- klauzule uloženy jako termy
- **programová databáze**
  - pro uložení klauzulí
  - má charakter haldy
    - umožňuje modifikovatelnost prologovských programů za běhu (assert)
- klauzule zřetězeny podle pořadí načtení
  - triviální zřetězení

Vyhodnocení dotazu: volání procedur řízené unifikací

### Interpret – vlastnosti

- Lokální i globální zásobník
  - při dopředném výpočtu roste
  - při zpětném výpočtu se zmenšuje

Lokální zásobník se může zmenšit při dopředném úspěšném výpočtu deterministické procedury.

- Unifikace argumentů hlavy – obecný unifikační algoritmus  
Současně poznačí adresy instanciovanych proměnných na stopu.

- „Interpret“:

```
interpret(Query, Vars) :- call(Query), success(Query, Vars).  
interpret(_,_) :- failure.
```

- dotaz vsazen do kontextu této speciální nedeterministické procedury
- tato procedura odpovídá za korektní reakci systému v případě úspěchu i neúspěchu

## Optimalizace: Indexace

- Zřetězení klauzulí podle pořadí načtení velmi neefektivní
- Provázání klauzulí se stejným funktorem a aritou hlavy (tvoří jednu **proceduru**)
  - tj., **indexace procedur**
- Hash tabulka pro vyhledání první klauzule
- Možno rozhodnout (parciálně) determinismus procedury

## Indexace argumentů

$a(1) :- q(1).$

$a(a) :- b(X).$

$a([A|T]) :- c(A,T).$

- Obecně nedeterministická
- Při volání s alespoň částečně instanciováním argumentem vždy deterministická (pouze jedna klauzule může uspět)
- **Indexace podle prvního argumentu**

Základní typy zřetězení:

- podle pořadí klauzulí (aktuální argument je volná proměnná)
- dle konstant (aktuální je argument konstanta)
- formální argument je seznam (aktuální argument je seznam)
- dle struktur (aktuální argument je struktura)

## Indexace argumentů II

- Složitější indexační techniky
  - podle všech argumentů
  - podle nejvíce diskriminujícího argumentu
  - kombinace argumentů (indexové techniky z databází)
    - zejména pro přístup k faktům

## Tail Recursion Optimization, TRO

Iterace prováděna pomocí rekurze  $\Rightarrow$  lineární paměťová náročnost cyklů

**Optimalizace koncové rekurze (*Tail Recursion Optimisation*), TRO:**

Okolí se odstraní **před** rekurzivním voláním posledního literálu klauzule, pokud je klauzule resp. její volání deterministické.

Řízení se nemusí vracet:

- v případě úspěchu se rovnou pokračuje
- v případě neúspěchu se vrací na předchozí bod volby („nad“ aktuální klauzulí)
  - aktuální klauzule nemá dle předpokladu bod volby

Rekurzivně volaná klauzule může být volána přímo z kontextu volající klauzule.

## TRO – příklad

Program:

```
append([], L, L).
```

```
append([A|X], L, [A|Y]) :- append(X, L, Y).
```

Dotaz:

```
?- append([a,b,c], [x], L).
```

append volán rekurzivně 4krát

- bez TRO: 4 okolí, lineární paměťová náročnost
- s TRO: 1 okolí, konstatní paměťová náročnost

## Optimalizace posledního volání

TRO pouze speciální případ

obecně **optimalizace posledního volání** (*Last Call Optimization*), LCO

Okolí (před redukcí posledního literálu)

odstraňováno vždy, když leží na vrcholu zásobníku.

### Nutné úpravy interpretu

- disciplína směřování ukazatelů
  - vždy „mladší“ ukazuje na „starší“ („mladší“ budou odstraněny dříve)
  - z lokálního do globálního zásobníku

vyhneme se vzniku „visících odkazů“ při předčasném odstranění okolí

- „globalizace“ lokálních proměnných: lokální proměnné posledního literálu
  - nutno přesunout na globální zásobník
  - pouze pro neinstanciované proměnné

## Překlad

## Překlad

- Motivace:
  - dosažení vyšší míry optimalizace
  - kompaktní kód
  - částečná nezávislost na hardware
- Etapy překladu:
  1. zdrojový text ⇒ kód abstraktního počítače
  2. kód abstraktního počítače ⇒ kód (instrukce) cílového počítače
- Výhody:
  - snazší přenos jazyka (nutno přepsat jen druhou část)
  - kód abstraktního počítače možno navrhnout s ohledem na jednoduchost překladu; prostor pro strojově nezávislou optimalizaci
- Překlad Prologu založen na principu existence abstraktního počítače  
V dalším se věnujeme jeho odvození a vlastnostem

## Parciální vyhodnocení

- Jak navrhnout **Warrenův abstraktní počítač**?
  - prostřednictvím parciálního vyhodnocení
- **Parciální vyhodnocení**
  - forma zpracování programu, tzv. transformace na úrovni zdrojového kódu
  - dosažení známých hodnot vstupních parametrů a vyhodnocení všech operací nad nimi
    - příklad: vyhodnocení aritmetických výrazů nad konstantami

## Parciální vyhodnocení – příklad

$a(X,Y) :- b(X), c(X,Y).$        $a(X,Y) :- b(Y), c(Y,X).$

$b(1).$     $b(2).$     $b(3).$     $b(4).$

$c(1,2).$     $c(1,3).$     $c(1,4).$     $c(2,3).$     $c(2,4).$     $c(3,4).$

Dotaz ?-  $a(2,Z).$

Ize společně s uvedeným programem parciálně vyhodnotit na nový program

$a'(3).$     $a'(4).$     $a'(1).$

a nový dotaz

?-  $a'(Z).$

Je evidentní, že dotaz nad parciálně vyhodnoceným programem bude zpracován mnohem rychleji (efektivněji) než v případě původního programu.

## Parciální vyhodnocení II

**Konstrukce překladače:** parciálním vyhodnocením interpretu

**Problémy:**

- příliš složitá operace
  - vyhodnocení se musí provést vždy znovu pro každý nový program
- výsledný program příliš rozsáhlý
- nedostatečná dekompozice
  - zejména při použití zdrojového jazyka jako implementačního jazyka interpretu

**Vhodnější:**

- využití („ručního“) parciálního vyhodnocení pro návrh abstraktního počítače
  1. nalezení operací zdrojového jazyka, které lze dekomponovat do jednodušších operací
  2. dekomponujeme tak dlouho, až jsou výsledné operace dostatečně jednoduché nebo již neexistují informace pro parciální vyhodnocení

## Parciální vyhodnocení Prologu

Cílová operace: **unifikace**.   Důvod:

- řízení výpočtu poměrně podrobně i v interpretu
- unifikace v interpretu atomickou operací
- unifikace v interpretu nahrazuje řadu podstatně jednodušších operací (testy, přiřazení, předání a převzetí parametrů . . .)
- většina unifikací nevyžaduje obecnou unifikaci a lze je nahradit jednoduššími operacemi

**Zviditelnění unifikace: transformací zdrojového programu**

- termíny reprezentujeme kopírováním struktur na globálním zásobníku
- parametry procedur jsou vždy umístěny na globální zásobník (predikátem put/2) a předávány jsou pouze adresy
- formálním parametrem procedury jsou pouze volné proměnné, které se v hlavě vyskytují pouze jednou
- všechny unifikace jsou explicitně zachyceny voláním predikátu uni fy/2

## Explicitní unifikace

Příklad: `append/3` s explicitní unifikací:

```
append(A1, A2, A3) :- unify(A1, []),      | append([], L, L).
                        unify(A2, L),      |
                        unify(A3, L).      |
append(A1, A2, A3) :- unify(A1, [A|X]),   | append([A|X], L, [A|Y]) :-
                        unify(A2, L),      |
                        unify(A3, [A|Y]),  |
                        put(X, B1),        | append(X, L, Y).
                        put(L, B2),        |
                        put(Y, B3),        |
                        append(B1, B2, B3). |
```

Cíl: parciálně vyhodnotit predikáty `unify/2` a `put/2`

## unify pro volnou proměnnou

`unify(A, T)` unifikuje term na adrese `A` (aktuální parametr) s termem `T` (formální parametr). Podle hodnoty `T` mohou nastat následující 4 případy:

1) `T` je volná proměnná: výsledkem je instanciac

```
unify(A, T) :- var(T),
              ( var(A), create_var(A)
              ; true ),
              T := $addr$(A).
```

Disjunkce garantuje, že `A` je korektní adresa na globálním zásobníku: nutný run-time test, tedy nelze využít při parc. překladu. Lze proto přepsat na

```
unify(A, T) :- var(T),
              unify_var(A, T).
```

kde `unify_var/2` vloží do `T` odkaz nebo založí novou proměnnou.

## Pomocné termy a predikáty

- term `$addr$(A)` – odkaz na objekt s adresou `A`
- predikát `i_s_addr(P, V)` – je-li `P` ve tvaru `$addr$(A)`, pak `V` se unifikuje s hodnotou slova na adrese `A` (jinak predikát selže)
- predikát `:=(X, T)` – přiřadí volné proměnné `X` term `T`;  
`X` musí být volná proměnná.
- predikát `repres(A, Tag, V)` – uloží do proměnné `Tag` příznak a do proměnné `V` hodnotu slova na adrese `A`.  
`A` musí být adresa na globálním zásobníku,  
`Tag` i `V` musí být volné proměnné.
  - příznak: informace o struktuře součástí objektu  
volná proměnná `FREE`, konstanta `CONST`, celé číslo `INT`, odkaz `REF`, složený term `FUNCT`
- je-li `A` adresa a `i` celočíselná konstanta, pak výraz `A+i` reprezentuje adresu o `i` slov vyšší (ukazatelová aritmetika)

## unify pro konstantu

2) `T` je konstanta: výsledkem je test nebo přiřazení

```
unify(A, T) :-
  atomic(T),
  ( ( var(A), create_var(A), instantiate_const(A, T) )
  ; ( repres(A, Tag, Value), Tag == 'FREE', instantiate_const(A, T)
      ; Tag == 'CONST', Value == T )
  ).
```

kde `instantiate_const/2` uloží do slova s adresou `A` hodnotu `T`.

Opět možno přepsat do kompaktního tvaru

```
unify(A, T) :-
  atomic(T),
  unify_const(A, T).
```

kde `unify_const/2` provede příslušný test nebo přiřazení.

## unify pro složený term

3) T je složený term: dvoufázové zpracování, v první fázi test nebo založení funktoru, v druhé rekurzivní unifikace argumentů

```
unify(A,T) :-  
    struct(T),  
    functor(T,F,N),  
    unify_struct(F,N,A),  
    T =.. [_|T1],  
    unify_args(T1,A+1).
```

Predikát unify\_struct/3 je analogický výše použitým predikátům unify\_var/2 a unify\_const/2.

```
Druhá fáze: unify_args([],_).  
            unify_args([T|T1], A) :-  
                unify(A,T),  
                unify_args(T1,A+1).
```

## unify pro odkaz

4) T je odkazem: nutno použít obecnou unifikaci (není žádná informace pro parciální vyhodnocení)

```
unify(A,T) :-  
    is_addr(T,P),  
    unification(A,P).
```

## put

Parametry procedur jsou vždy umístěny na globální zásobník predikátem put/2 a předávány jsou pouze adresy.

Predikát put/2 je jednodušší (nikdy nepotřebuje unifikaci)

```
put(T,B) :-  
    is_addr(T,B).           % T je odkaz  
  
put(T,B) :-  
    var(T),                 % T je proměnná  
    create_var(B),  
    T := $addr$(B).  
  
put(T,B) :-  
    atomic(T),             % T je konstanta  
    create_const(B,T).  
  
put(T,B) :-  
    struct(T),             % T je struktura  
    create_struct(B,T).
```

## První klauzule append/3

Parciální vyhodnocení první klauzule programu append/3

```
append(A1, A2, A3) :- unify(A1,[]),      | append([],L,L).  
                      unify(A2,L),      |  
                      unify(A3,L).      |
```

upraví

unify(A1,[]) na unify\_const(A1,[])

unify(A2,L) na L:=\$addr\$(A2)

unify(A3,L) na is\_addr(L,T), unification(T,A3)

posloupnost L:=\$addr\$(A2), is\_addr(L,T) odpovídá přejmenování T na A2

⇒ není nutné vytvářet novou proměnnout T

⇒ stačí provést unification(A2,A3)

## Výsledný tvar append/3

```
append(A1, A2, A3) :-
    unify_const(A1, []),
    unification(A2,A3).
append(A1, A2, A3) :-
    unify_struct('.',2,A1),
    unify_var(A,A1+1),
    unify_var(X,A1+2),
    unify_var(L,A2),
    unify_struct('.',2,A3),
    unification(A1+1,A3+1),
    unify_var(Y,A3+2),

    append(A1+2,A2,A3+2).

append(A1, A2, A3) :-
    unify(A1, []),
    unify(A2,L), unify(A3,L).
append(A1, A2, A3) :-
    unify(A1, [A|X]),
    unify(A2,L),
    unify(A3, [A|Y]),

    put(X,B1), put(L,B2), put(Y,B3),
    append(B1,B2,B3).
```

Většina původních unifikací převedena na jednodušší operace;  
unifikace v posledním kroku je nezbytná (důsledkem dvojího výskytu proměnné)

## Jiný příklad

```
a(c,s(f),d,X) :- g(X).
```

Procedurální pseudokód (testy a přiřazení) a kód abstraktního počítače:

```
procedure a(X,Y,Z,A) is
    if ( X == 'c' &&
        ( is_struct(Y,'s',1) &&
          first_arg(Y) == 'f' ) &&
        Z == 'd' )
    then
        call g(A)
    else
        call fail
    end procedure

a(A1, A2, A3, A4) :-
    unify_const(c,A1),
    unify_struct(s,1,A2),
    unify_const(f,A2+1),
    unify_const(d,A3),
    unify_var(A,A4),
    g(A4).
```

tj. posloupnost testů jako v procedurálním jazyce

Vyzkoušejte si: delete(X, [Y|T], [Y|T1]) :- delete(X, T, T1).

## Warrenův abstraktní počítač, WAM I.

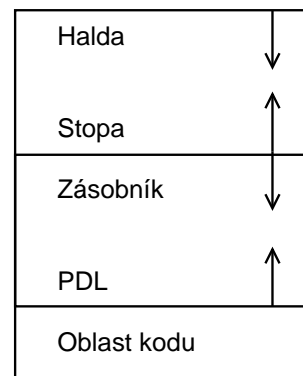
Navržen D.H.D. Warrenem v roce 1983, modifikace do druhé poloviny 80. let

Datové oblasti:

- **Oblast kódu** (programová databáze)
  - separátní oblasti pro uživatelský kód (modifikovatelný) a vestavěné predikáty (nemění se)
  - obsahuje rovněž všechny statické objekty (texty atomů a funktorů apod.)
- **Lokální zásobník (Stack)**
- **Stopa (Trail)**
- **Globální zásobník n. halda(Heap)**
- **Pomocný zásobník (Push Down List, PDL)**
  - pracovní paměť abstraktního počítače
  - použitý v unifikaci, syntaktické analýze apod.

## Rozmístění datových oblastí

- Příklad konfigurace



- Halda i lokální zásobník musí růst stejným směrem
  - lze jednoduše porovnat stáří dvou proměnných srovnáním adres využívá se při zabránění vzniku visících odkazů

## Registry WAMu

- **Stavové registry:**
  - P čítač adres (Program counter)
  - CP adresa návratu (Continuation Pointer)
  - E ukazatel na nejmladší okolí (Environment)
  - B ukazatel na nejmladší bod volby (Backtrack point)
  - TR vrchol stopy (TRail)
  - H vrchol haldy (Heap)
  - HB vrchol haldy v okamžiku založení posledního bodu volby (Heap on Backtrack point)
  - S ukazatel, používaný při analýze složených termů (Structure pointer)
  - CUT ukazatel na bod volby, na který se řezem zařízne zásobník
- **Argumentové registry:** A1, A2, . . . (při předávání parametrů n. pracovní registry)
- **Registry pro lokální proměnné:** Y1, Y2, . . .
  - abstraktní znázornění lok. proměnných na zásobníku

## Typy instrukcí WAMu

- **put instrukce** – příprava argumentů před voláním podcíle
  - žádná z těchto instrukcí nevolá obecný unifikáčnı algoritmus
- **get instrukce** – unifikace aktuálních a formálních parametrů
  - vykonávají činnost analogickou instrukcım uni fy u parc. vyhodnocení
  - obecná unifikace pouze při get\_value
- **unify instrukce** – zpracování složených termů
  - jednoargumentové instrukce, používají registr S jako druhý argument
  - počáteční hodnota S je odkaz na 1. argument
  - volání instrukce uni fy zvětší hodnotu S o jedničku
  - obecná unifikace pouze při uni fy\_value a uni fy\_local\_value
- **Indexační instrukce** – indexace klauzulí a manipulace s body volby
- **Instrukce řízení běhu** – předávání řízení a explicitní manipulace s okolím

## Instrukce put a get: příklad

Příklad: a(X,Y,Z) :- b(f,X,Y,Z).

```
get_var      A1,A5
get_var      A2,A6
get_var      A3,A7
put_const    A1,f
put_value    A2,A5
put_value    A3,A6
put_value    A4,A7
execute      b/4
```

## Instrukce WAMu

get instrukce	put instrukce	unify instrukce
get_var Ai,Y	put_var Ai,Y	unify_var Y
get_value Ai,Y	put_value Ai,Y	unify_value Y
get_const Ai,C	put_unsafe_value Ai,Y	unify_local_value Y
get_nil Ai	put_const Ai,C	unify_const C
get_struct Ai,F/N	put_nil Ai	unify_nil
get_list Ai	put_struct Ai,F/N	unify_void N
	put_list Ai	

instrukce řízení	indexační instrukce
allocate	try_me_else Next try Next
deallocate	retry_me_else Next retry Next
call Proc/N,A	trust_me_else fail trust fail
execute Proc/N	
proceed	cut_last switch_on_term Var,Const,List,Struct
	save_cut Y switch_on_const Table
	load_cut Y switch_on_struct Table



# Instrukce unify, get, put

- Větší počet typů objektů
  - rozlišeny atomy, čísla, nil ≡ prázdný seznam, seznam speciální druh složeného termu
- unify\_void umožní přeskočit anonymních proměnné ve složených termech
- put\_unsafe\_value pro optimalizaci práce s lokálními proměnnými při TRO
  - a(X) :- b(X,Y), !, a(Y).  
při TRO nesmí být lokální proměnné posledního literálu (Y) na lokálním zásobníku
  - kompilátor může všechny **nebezpečné (unsafe)** výskyty lok. proměnných detekovat při překladu (jsou to poslední výskyty lok. proměnných) a generuje složitější instrukce put\_unsafe\_value, které provádějí test umístění
- unify\_local\_value kvůli TRO jako put\_unsafe\_value
  - a(X) :- d(X), b(s(Y),X). objekt přístupný přes Y opět nesmí být na lok. zásobníku doba života s/1 může být delší než doba života okolí na něž se Y odkazuje
  - unify\_local\_value testují umístění a pokud nutné přesouvají objekty na haldu

# WAM – indexace

- Provázání klauzulí: instrukce XX\_me\_else:
  - první klauzule: try\_me\_else; založí bod volby
  - poslední klauzule: trust\_me\_else; zruší nejmladší bod volby
  - ostatní klauzule: retry\_me\_else; znovu použije nejmladší bod volby po neúspěchu
- Provázání podmnožiny klauzulí (podle argumentu):
  - try
  - retry
  - trust
- „Rozskokové” instrukce (dle typu a hodnoty argumentu):
  - switch\_on\_term Var, Const, List, Struct  
výpočet následuje uvedeným návěstím podle typu prvního argumentu
  - switch\_on\_YY: hashovací tabulka pro konkrétní typ (konstanta, struktura)

# Příklad indexace instrukcí

```

Proceduře
a(atom) :- body1.
a(1) :- body2.
a(2) :- body3.

a([X|Y]) :- body4.
a([X|Y]) :- body5.
a(s(N)) :- body6.
a(f(N)) :- body7.

```

## odpovídají instrukce

```

a:   switch_on_term L1, L2, L3, L4   L5a: body2
L2:  switch_on_const atom :L1a      L6:  retry_me_else L7
      1 :L5a                        L6a: body3
      2 :L6a                        L7:  retry_me_else L8
L3:  try L7a                        L7a: body4
      trust L8a                      L8:  retry_me_else L9
L4:  switch_on_struct s/1 :L9a      L8a: body5
      f/1 :L10a                      L9:  retry_me_else L10
L1:  try_me_else L5                 L9a: body6
L1a: body1                          L10: trust_me_else fail
L5:  retry_me_else L6               L10a: body7

```

# WAM – řízení výpočtu

- execute Proc: ekvivalentní příkazu goto
  - proceed: zpracování faktů
  - allocate: alokuje okolí (pro některé klauzule netřeba, proto explicitně generováno)
  - deallocate: uvolní okolí (je-li to možné, tedy leží-li na vrcholu zásobníku)
  - call Proc,N: zavolá Proc, N udává počet lok. proměnných (odpovídá velikosti zásobníku)  
Možná optimalizace: vhodným uspořádáním proměnných lze dosáhnout postupného zkracování lokálního zásobníku
- ```

a(A,B,C,D) :- b(D), c(A,C), d(B), e(A), f.
generujeme instrukce
allocate
call b/1,4
call c/2,3
call d/1,2
call e/1,1
deallocate
execute f/0

```

## WAM – řez

Implementace řezu (opakování): odstranění bodů volby mezi současným vrcholem zásobníku a bodem volby procedury, která řez vyvolala (včetně bodu volby procedury s řezem)

Indexační instrukce znemožňují v době překladu rozhodnout, zda bude alokován bod volby

- příklad:  $?- a(X)$ . může být nedeterministické, ale  $?- a(1)$ . může být deterministické

cut\_last: B := CUT      save\_cut Y: Y := CUT      load\_cut Y: B := Y

Hodnota registru B je uchovávána v registru CUT instrukcemi call a execute.

Je-li řez prvním predikátem klauzule, použije se rovnou cut\_last. V opačném případě se použije jako první instrukce save\_cut Y a v místě skutečného volání řezu se použije load\_cut Y.

Příklad:  $a(X,Z) :- b(X), !, c(Z)$ .

$a(2,Z) :- !, c(Z)$ .

$a(X,Z) :- d(X,Z)$ .      odpovídá

save\_cut Y2; get A2,Y1; call b/1,2; load\_cut Y2; put Y1,A1; execute c/1

get\_const A1,2; cut\_last; put A2,A1; execute c/1

execute d/2

## WAM – optimalizace

1. Indexace klauzulí

2. Generování optimální posloupnosti instrukcí WAMu

3. Odstranění redundancí při generování cílového kódu.

- Příklad:  $a(X,Y,Z) :- b(f,X,Y,Z)$ .

naivní kód (vytvoří kompilátor pracující striktně zleva doprava) vs.

optimalizovaný kód (počet registrů a tedy i počet instrukcí/přesunů v paměti snížen):

|           |       |  |           |       |
|-----------|-------|--|-----------|-------|
| get_var   | A1,A5 |  | get_var   | A3,A4 |
| get_var   | A2,A6 |  | get_var   | A2,A3 |
| get_var   | A3,A7 |  | get_var   | A1,A2 |
| put_const | A1,f  |  | put_const | A1,f  |
| put_value | A2,A5 |  | execute   | b/4   |
| put_value | A3,A6 |  |           |       |
| put_value | A4,A7 |  |           |       |
| execute   | b/4   |  |           |       |