Osvrt na kompajlere

- U toku kompajliranja mogu da se izdvoje **prolazi**
  - to su delovi procesa kompajliranja koji obuhvataju
    - čitanje ulazne datoteke
      - una primer sa izvornim programom i
    - pisanje izlazne datoteke
      - na primer sa ciljnim programom
- Na osnovu broja prolaza, kompajleri mogu biti:
  - jednoprolazni
  - dvoprolazni
- Broj faza koje obuhvata jedan prolaz se menja od kompajlera do kompajlera

Osvrt na kompajlere

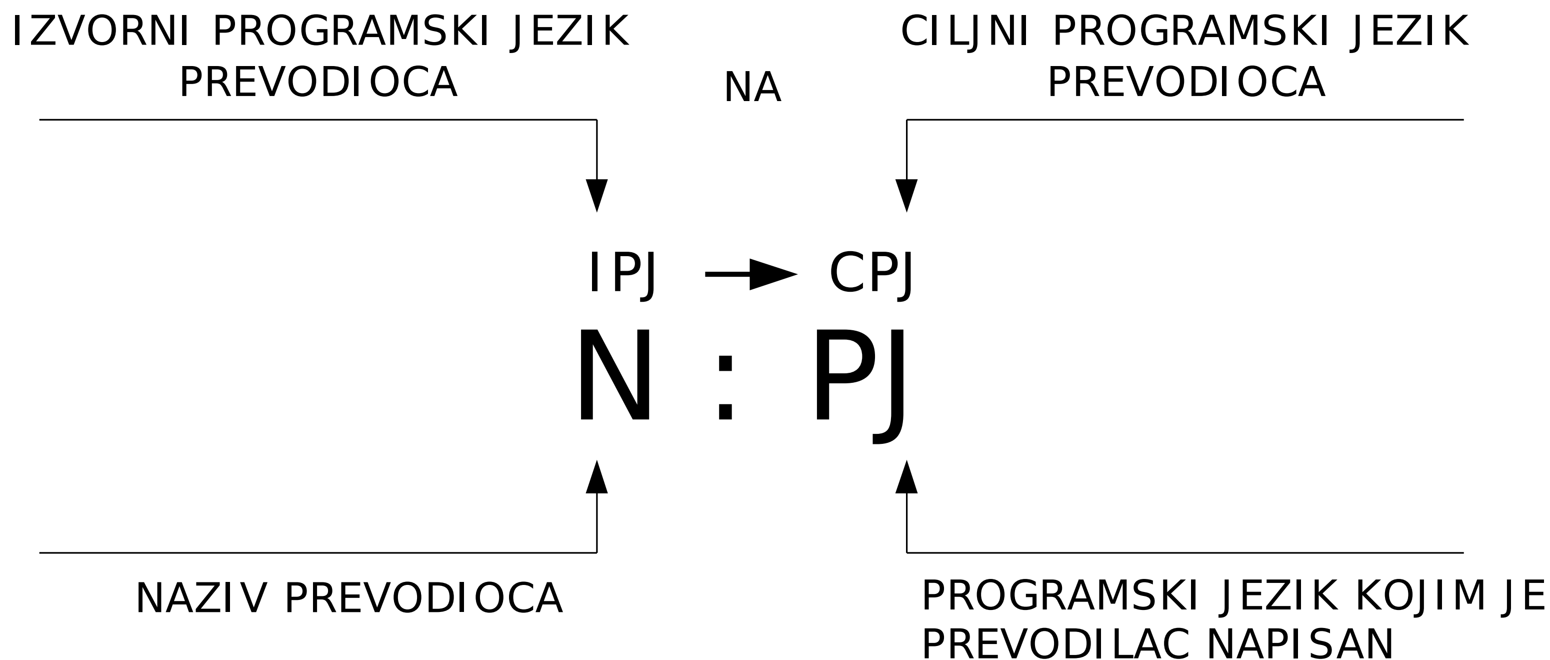
- Jednoprolazni kompajler
  - sve faze u jednom prolazu
  - karakterističan za neke *Pascal* kompajlere
    - njihov ciljni jezik je međukod, nazvan *P-code* zbog svojstva portabilnosti sa računara na računar
    - portabilnost P-koda se zasniva na pravljenju interpretera P-koda za razne računare
- Dvoprolazni kompajler
  - prvi prolaz odgovara prednjem modulu kompajlera, a
  - drugi prolaz odgovara zadnjem modulu kompajlera
  - karakterističan za *C* kompajlere

Osvrt na kompajlere

- Osobine kompajlera
  - brzina prevođenja (manje prolaza – veća brzina prevođenja)
  - kvalitet izgenerisanog koda
    - brzina izvršavanja
    - iskorišćenost resursa (memorija, pristup disku, baterija)
  - dobra dijagnostika grešaka
  - dobra optimizacija
  - prenosivost
    - na novi ciljni jezik – *retargetability*
    - na novi računar – *rehostability*
  - lakoća održavanja
  - *bug-free*
  - mora generisati ispravan mašinski kod

Osvrt na kompajlere

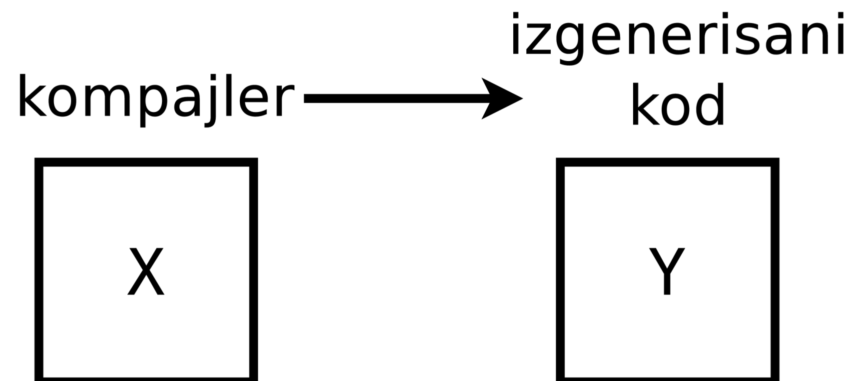
- Kompajler karakterišu ne samo njegov izvorni i ciljni jezik, nego i njegov implementacioni jezik
- Oznaka kompajlera (programskog prevodioca):



- Prevodilac N je napisan programskim jezikom PJ i prevodi sa izvornog programskog jezika IPJ na ciljni programski jezik CPJ

Osvrt na kompajlere

- Kompajleri čiji ciljni jezik ne pripada računaru na kome se oni izvršavaju se nazivaju **kros-kompajleri** *(cross-compiler)*
  - *embedded* sistem, uređaj sa ograničenim resursima
  - platforma na kojoj nije moguće kompajlirati
    - mikrokontroleri koji nemaju OS
- Pravljenje kompajlera se značajno olakšava ako se oslanja na postojeće kompajlere i kros-kompajlere



X i Y su različite arhitekture

Osvrt na kompajlere

- Postupak u kome kompajler kompilira "sam sebe" ili neku svoju verziju ili drugi kompajler *(**bootstraping**)* je uobičajen u toku pravljenja kompajlera za novi računar ili za usavršavanje kompajlera za isti računar

Interpreter

- Program koji oponaša hipotetski računar kome odgovara međukod
- Realizacija interpretera podrazumeva postojanje struktura podataka koje opisuju *run-time* stanje programa:
  - vrednosti koje se pojavljuju u programu
  - *activation record* (ekvivalent stek frejmu) za svaku pozvanu funkciju
    - veze između lokalnih promenljivih
    - pokazivač na pozivajući *activation record* (*dynamic link*)
    - pokazivač na *lexically-enclosing activation record* (*static link*; kod programskih jezika koji dozvoljavaju funkcije u funkciji, ovo je pokazivač na *activation record* funkcije koja je na vrhu hijerarhije)

Interpreter

- Hipotetski računar koga oponaša interpreter je često računar sa stek arhitekturom
- Primer hipotetskog računara: *JVM* (*Java Virtual Machine*)
- Posmatraće se podskup *JVM* označen skraćenicom *IJVM* (*integer JVM*)

IJVM memorijski model

- *Java Virtual Machine*
  - stek-orijentisana virtuelna mašina
    - nema registara
    - nego se operandi svih naredbi uzimaju sa posebnog steka operanada
- Memorija *IJVM* se može posmatrati kao niz od 4GB ili kao niz od 1G reči, gde svaka reč sadrži 4 bajta
- *Java Virtual Machine* ne adresira direktno
  - već preko nekoliko implicitnih adresa (pokazivača)
  - *IJVM* naredbe mogu pristupiti memoriji samo indeksiranjem preko ovih pokazivača

IJVM memorijski model

- Memorija *IJVM* je podeljena u 4 dela:
  1. *Constant Pool*
     - *IJVM* program ne može da ga menja
     - ovaj deo memorije sadrži konstante, stringove i pokazivače na druge delove memorije koji mogu biti referencirani
     - puni se kada program dospe u memoriju i kasnije se ne može menjati
     - postoji implicitni registar *CPP* koji sadrži adresu prve reči iz *constant pool*

IJVM memorijski model

2. *Local Variable Frame*

- prilikom svakog poziva metode, alocira se prostor za smeštanje promenljivih korišćenih tokom životnog veka poziva
- taj prostor se zove frejm lokalnih promenljivih
- na početku frejma nalaze se argumenti sa kojima je metoda pozvana
- frejm lokalnih promenljivih ne sadrži stek operanada (on je zaseban)
- postoji implicitni registar koji sadrži adresu prve lokacije frejma lokalnih promenljivih
- ovaj registar se zove *LV*

IJVM memorijski model

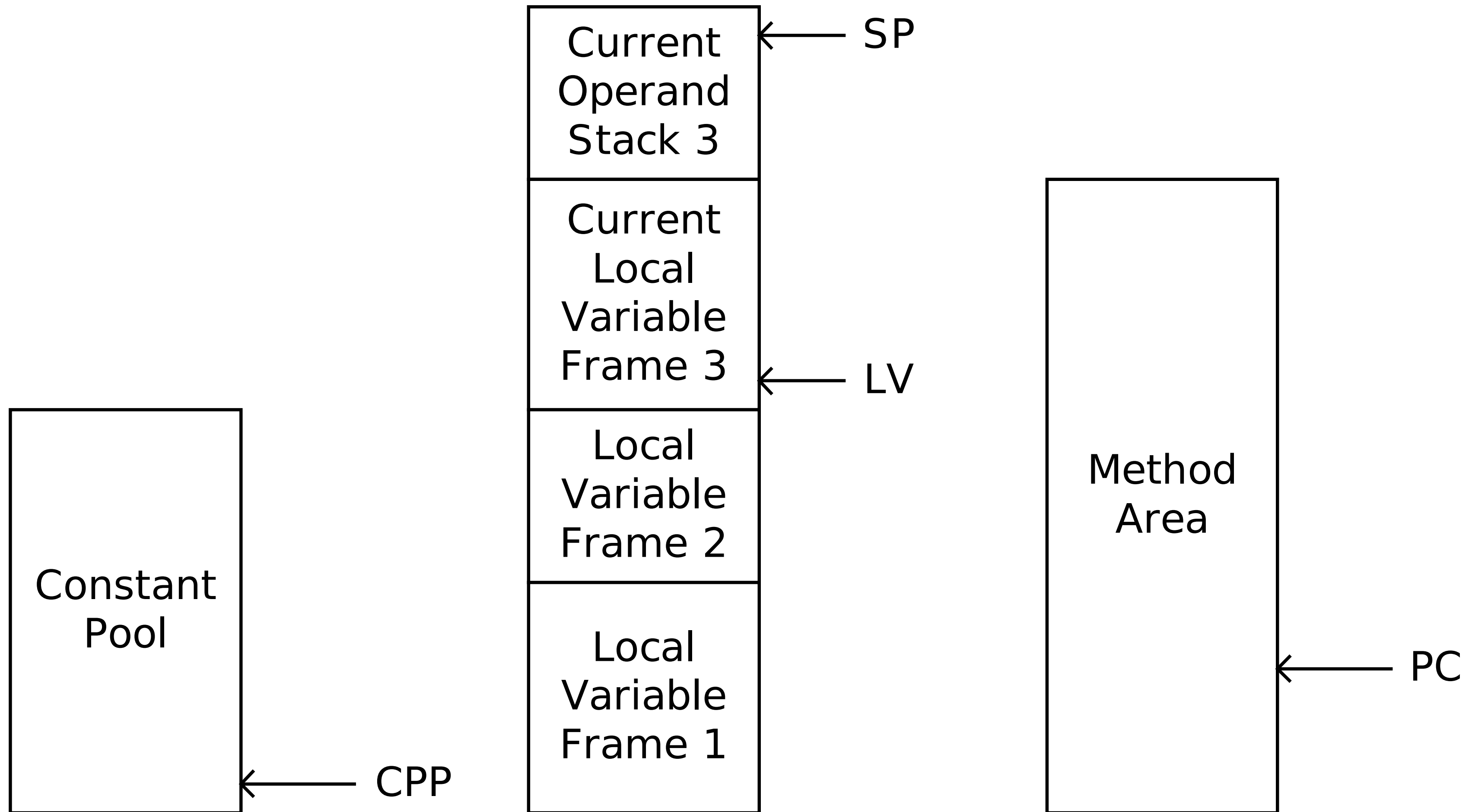
3. *Operand Stack*
   - Java prevodilac garantuje da stek frejm neće preći određenu veličinu, jer veličinu steka izračunava unapred
   - prostor za stek operanada se alocira tačno iznad frejma lokalnih promenljivih
   - postoji implicitni registar koji sadrži adresu poslednje reči na steku
   - pokazivač *SP* se menja tokom izvršavanja metode jer se operandi smeštaju na stek i skidaju sa njega

IJVM memorijski model

4. *Method Area*
   - postoji još i deo memorije koji sadrži program
   - implicitni registar koji sadrži adresu naredbe koja će biti sledeća izvršavana se zove *Program Counter* ili *PC*
   - za razliku od ostalih regiona memorije, *Method Area* se tretira kao niz bajtova

- Treba zapaziti razliku između registara:
  - registri *CPP*, *LV* i *SP* pokazuju na reči i predstavljaju ofset po broju reči
  - nasuprot tome, *PC* sadrži bajt adresu, pa tako i sva aritmetika nad njim menja adresu za broj bajtova - a ne za broj reči

IJVM memorijski model



Skup naredbi za *IJVM*

- Skup naredbi za *IJVM* je dat u tabeli
  - svaka naredba se sastoji od koda operacije i ponekad operanda
  - prva kolona prikazuje heksadecimalni kod instrukcije
    - *byte code*
  - u drugoj koloni je naveden mnemonik asemblerskog jezika
  - treća kolona sadrži kratak opis naredbe

Skup naredbi za *IJVM*

Hex	Mnemonik	Značenje
0x10	BIPUSH byte	smešta *byte* na vrh steka
0x13	LDC_W index	smešta konstantu iz *constant pool* na vrh steka
0x15	ILOAD varnum	smešta lokalnu promenljivu *varnum* na vrh steka
0x36	ISTORE varnum	skida reč sa vrha steka i smešta je u lokalnu promenljivu *varnum*
0x57	POP	briše reč sa vrha steka
0x59	DUP	smešta reč sa vrha steka na vrh steka
0x5F	SWAP	zamenjuje dve reči na vrhu steka
0x60	IADD	skida dve reči sa vrha steka, sabere ih i rezultat smesti na vrh steka
0x64	ISUB	skida dve reči sa vrha steka, oduzme ih i rezultat smesti na vrh steka
0x68	IMUL	skida dve reči sa vrha steka, pomnoži ih i rezultat smesti na vrh steka
0x6C	IDIV	skida dve reči sa vrha steka, podeli ih i rezultat smesti na vrh steka

Skup naredbi za *IJVM*

Hex	Mnemonik	Značenje
0x7E	IAND	skida dve reči sa vrha steka, napravi njihov Boolean AND i rezultat smesti na vrh steka
0x80	IOR	skida dve reči sa vrha steka, napravi njihov Boolean OR i rezultat smesti na vrh steka
0x84	IINC varnum const	dodaje konstantu lokalnoj promenljivoj
0x99	IFEQ offset	skida reč sa vrha steka i ako je 0 usmerava nastavak izvršavanja na naredbu *offset*
0x9B	IFLT offset	skida reč sa vrha steka i ako je manja od 0 usmerava nastavak izvršavanja na naredbu *offset*
0x9F	IF_ICMPEQ offset	skida dve reči sa vrha steka i ako su iste usmerava nastavak izvršavanja na naredbu *offset*
0xA7	GOTO offset	usmerava nastavak izvršavanja na naredbu *offset*
0xAC	IRETURN	povratak iz metode sa celobrojnom vrednošću
0xB6	INVOKE VIRTUAL disp	poziva metodu
0x00	NOP	prazna operacija

Primer

Java program:

```
i = j + k;
if (i == 3)
  k = 0;
else
  j = j – 1;
```

odgovarajući Java asemblerski program:

```
    ILOAD j          // i = j + k;
    ILOAD k
    IADD
    ISTORE i
    ILOAD i          // if (i == 3)
    BIPUSH 3
    IF_ICMPEQ L1
    ILOAD j           // j = j - 1
    BIPUSH 1
    ISUB
    ISTORE j
    GOTO L2
L1: BIPUSH 0          // k = 0
    ISTORE k
L2:
```

Primer

Java program:

```
m = (a + b) * c - -d;
```

odgovarajući Java asemblerski program:

```
ILOAD a        // a + b
ILOAD b
IADD
ILOAD c        // (a + b) * c
IMUL
BIPUSH 0       // -d
ILOAD d
ISUB
ISUB
ISTORE m
```

Osvrt na gramatike

- Formalna gramatika se sastoji od:
  - pravila (*lhs* → *rhs*)
  - pojmovi (neterminalni simboli)
  - simboli (terminalni simboli)
  - startni pojam
- Formalna gramatika definiše formalni jezik
  - kao skup svih nizova simbola koji mogu biti konstruisani primenom pravila gramatike
  - *lhs* može biti zamenjen *rhs*, i obrnuto

Osvrt na gramatike

- *Chomsky* hijerarhija gramatika
  - tip0 - **opšte gramatike**
  - tip1 - **kontekstno zavisne gramatike**
    - *non-context-free grammars*
  - tip2 - **kontekstno nezavisne gramatike**
    - *context-free grammars*
  - tip3 - **regularne gramatike**
    - *regular grammars*

  - tip2 i tip3 se najčešće koriste
  - tip2 za parsere
  - tip3 za skenere

Osvrt na gramatike

- Regularne gramatike se u praksi izražavaju regularnim izrazima
  - pomoću njih se lako i koncizno izražavaju leksička pravila
  - iz njih se automatski generišu konačni automati (skeneri)

Osvrt na gramatike

- Pored determinističkih postoje i **nedeterministički konačni automati**
  - kod njih iz nekog čvora može izlaziti više spojnica labeliranih istim znakovima
  - imaju manje čvorova od determinističkih
  - cena: među spojnicama koje su labelirane istim znakovima ne mogu unapred da odaberu spojnicu koja vodi ka prepoznavanju pravog simbola
  - neuspeh u prepoznavanju simbola ne znači obavezno grešku, nego podrazumeva **vraćanje skeniranja u natrag** (*backtracking*) radi pokušaja sa nekom od preostalih alternativnih spojnica
  - zbog toga su nedeterministički automati u proseku sporiji od determinističkih

Osvrt na gramatike

- Kontekstno nezavisne gramatike su uvedene radi sintaksne analize
- Iz skupa kontekstno nezavisnih gramatika izdvajaju se dva podskupa:
  - LL(1) gramatike koje su prilagođene silaznom parsiranju i
  - LR(1) gramatike koje su prilagođene uzlaznom parsiranju

LR(1) gramatike

- LR(1) gramatike
  - ***L****eft-to right scanning of the input, for constructing a* ***R****ightmost derivation in reverse*
    - čitaju ulazni tekst s leva na desno
    - prave desna izvođenja na gore
  - ***1*** znači da koriste 1 *lookahead* token za donošenje odluka u toku parsiranja
  - nemaju desnih rekurzija
  - mogu biti dvosmislene
  - pogodne za *bootom-up* parsiranje
  - pogodne za automatsko generisanje parsera

LR(1) gramatike

- LR parseri
  - mogu prepoznati praktično sve jezike opisane pomoću kontekstno nezavisnih gramatika
    - pokrivaju veći broj gramatika od LL parsera
    - generalniji su od LL parsera
  - imaju efikasnu implementaciju
    - koriste najgeneralniji "*non backtracking shift-reduce parsing*" metod
    - otkrivaju sintaksne greške u najmanjem mogućem broju koraka
  - nisu pogodni za ručno pravljenje,
  - mogu biti automatski generisani pomoću generatora LR parsera

LR(1) gramatike

- Generatori LR parsera
  - mogu da ukažu na dvosmislenosti gramatike
  - glavni problem je pravljenje tabele akcija i prelaza
  - algoritmi za pravljenje tabele akcija i prelaza:
    - *Simple LR* (SLR)
      - lak za implementaciju
      - ali primenljiv za najmanji broj LR gramatika
    - *Canonical LR*
      - najteži za implementaciju
      - ali primenljiv za najveći broj LR gramatika
    - *Look-ahead LR* (LALR)
      - srednji po težini implementacije
      - srednji po mogućnostima primene
      - koristi se u Bison-u

LL(1) gramatike

- LL(1) gramatike
  - ***L****eft–to-right scanning of the input, producing a* ***L****eftmost derivation*
    - čitaju ulazni tekst s leva na desno
    - prave leva izvođenja
  - ***1*** znači da koriste 1 *lookahead* token za donošenje odluka u toku parsiranja
  - nemaju levih rekurzija
  - nisu dvosmislene
  - pogodne za silazno (*top-down*) parsiranje
  - pogodne za ručno pravljenje kompajlera
    - primenom predvidivog rekurzivnog spuštanja
  - nisu pogodne za automatsko generisanje parsera

Silazno parsiranje

- Silazno parsiranje proverava da li je ulazni niz simbola u skladu sa datom gramatikom tako što proverava da li gramatika dozvoljava
  - da prvi simbol iz ulaznog niza bude na prvom mestu, a
  - da drugi simbol iz ulaznog niza sledi iza prvog simbola, itd.
- Postupak provere se pojednostavljuje ako se za svaki pojam gramatike uvede odgovarajući **sintaksni potprogram** koji proverava da li je **posmatrani simbol** (iz ulaznog niza) prihvatljiv sa stanovišta dotičnog pojma
- Podrazumeva se da sintaksni potprogram jednog pojma poziva sintaksne potprograme drugih pojmova da bi zajedno proverili ispravnost ulaznog niza simbola
- Zahvaljujući saradnji sintaksnih potprograma, silazno parsiranje započinje pozivom sintaksnog potprograma polaznog pojma
  - iz njega se pozivaju sintaksni potprogrami pojmova koji se pominju sa desne strane pravila polaznog pojma itd.

Silazno parsiranje

- Ovakvo parsiranje se naziva **rekurzivno spuštanje** (*recursive-descent parsing*)
  - jer pozivi sintaksnih potprograma dovode do spuštanja niz stablo parsiranja
  - pri tome su neizbežni rekurzivni pozivi sintaksnih potprograma zbog rekurzivne prirode konekstno nezavisnih gramatika
  - na primer, sintaksni potprogram pojma *`statement`* poziva sintaksni potprogram pojma *`compound_statement`*, a on, posredstvom sintaksnog potprograma pojma *`statement_list`* poziva sintaksni potprogram pojma *`statement`*
  - za rekurzivno spuštanje je zgodno da se na osnovu posmatranog simbola uvek jednoznačno može odrediti sintaksni potprogram koji se sledeći poziva
    - ovakvo rekurzivno spuštanje se naziva **predvidivim** (*predictive recursive-descent parsing*)

Silazno parsiranje

- Kod predvidivog rekurzivnog spuštanja ne dolazi do vraćanja parsiranja unatrag (*backtracking*)
  - jer je u svakom sintaksnom potprogramu (na svakom koraku parsiranja) unapred određeno koji simboli su prihvatljivi, a za svaki od njih je obezbeđeno da jednoznačno usmerava nastavak parsiranja
- Predvidivo rekurzivno spuštanje zahteva posebne gramatike
- Jedno svojstvo ovakvih gramatika je
  - da se svi početni simboli međusobno razlikuju
  - početni simboli nekog pojma obrazuju njegov **početni skup simbola** (*FIRST*)

Silazno parsiranje

- Na primer, početni skup simbola pojma *`type`*

  `type` → **`"int" | "unsigned"`**

  sačinjavaju simboli **`int`** i **`unsigned`**
- Kod određivanja početnog skupa simbola treba konsultovati:
  - desne strane pravila koja opisuju izvođenja iz datog pojma
  - ako desne strane ovih pravila započinju novim pojmovima, tada se moraju konsultovati i desne strane pravila koja opisuju izvođenja iz novih pojmova i tako redom
  - na primer, početni skup simbola pojma *`variable`*:

    `variable` → ***`type`*** **`identifier`**

    sačinjavaju simboli **`int`** i **`unsigned`**.

FIRST skup

- *FIRST* skup nekog pojma gramatike je skup simbola kojima započinju desne strane alternativnih pravila tog pojma
- Algoritam određivanja *FIRST* skupa za pojam X:
  - ako postoji izvođenje **X =>* ε**, dodati **ε** u *FIRST(X)*
  - za pravilo **X → Y1 Y2 … Yk**
    - sve simbole iz skupa *FIRST(Y1)* dodati u *FIRST(X)*
    - ako postoji izvođenje **Y1 =>* ε** onda sve simbole iz skupa *FIRST(Y2)* dodati u *FIRST(X)*;
      - ukoliko je potrebno, isti postupak primenjivati na **Y2, ... Yk**

Primer *FIRST* skupa

- primer gramatike izraza:

```
expression    →   term expression1
expression1   →   "+" term expression1
              |   ε
term   →   factor term1
term1  →   "*" factor term1
       |   ε
factor →   "(" expression ")"
       |   id
```

FIRST(expression) = *FIRST(term)* = *FIRST(factor)* = { (, id }

FIRST(expression1) = { +, ε }

FIRST(term1) = { *, ε }

Primer *FIRST* skupa

- primer gramatike iskaza:

```
statement
 → if_statement
 → return_statement

if_statement
  → “if” “(” rel_exp “)” statement

return_statement
  → “return” num_exp “;”
```

FIRST(statement) = { if, return }
FIRST(if_statement) = { if }
FIRST(return_statement) = { return }

Silazno parsiranje

- Jedinstvenost simbola iz početnog skupa simbola nekog pojma obezbeđuje jednoznačno usmeravanje predvidivog rekurzivnog spuštanja

Silazno parsiranje

- Iz ograničenja da simboli iz početnog skupa simbola nekog pojma moraju biti jedinstveni sledi da
  - leva rekurzija u pravilima i identični počeci desnih strana alternativnih pravila istog pojma nisu prihvatljivi,
  - jer narušavaju jedinstvenost simbola iz početnog skupa simbola nekog pojma
- Levu rekurziju sadrži pravilo pojma ***variable_list***:

  variable_list → ***variable*** **";"**

  | ***variable_list variable*** **";"**

  Ako je "**int**" posmatrani simbol, tada on u sintaksnom potprogramu prethodnog pojma ne upućuje jednoznačno na izbor jedne od dve alternative (jer su početni simboli ovih alternativa identični).

  Međutim, takva dilema nestaje ako se pravila pojma ***variable_list*** preformulišu (*elimination of left recursion*):

Silazno parsiranje

```
variable_list  → variable ";" variable_list1
variable_list1 → variable ";" variable_list1
                | ε
```

jer je, nakon uvođenja novog pojma *variable_list1*, preostala samo jedna alternativa u pravilu pojma *variable_list*

- Identične početke desnih strana sadrže pravila pojma *if*:

```
if → "if" "(" log_exp ")" statement
if → "if" "(" log_exp ")" statement "else" statement
```

Silazno parsiranje

- Ako je "**`if`**" posmatrani simbol u sintaksnom pravilu prethodnog pojma, tada on ne upućuje jednoznačno na izbor jedne od dve alternative (jer su početni simboli ovih alternativa identični). Međutim, takva dilema nestaje, ako se pravila pojma *`if`* preformulišu (*left factoring*):

```
if   → "if" "(" log_exp ")" statement else
else → "else" statement
     |   ε
```

jer je, nakon uvođenja novog pojma *`else`*, preostala samo jedna alternativa u pravilu pojma *`if`*

Silazno parsiranje

- Pored početnog skupa simbola, za svaki pojam je važan i **sledbenički skup simbola** (*FOLLOW*)
  - njega obrazuju simbol kraja datoteke i
  - simboli koji mogu da slede iza dotičnog pojma
- Sledbenički skup olakšava oporavak od grešaka u toku kompajliranja, jer omogućuje da se,
  - nakon otkrivanja pogrešnog simbola u okviru sintaksnog potprograma nekog pojma,
  - ignorišu se simboli koji se u ulaznom nizu nalaze između pogrešnog simbola i simbola koji pripada sledbeničkom skupu ovog pojma *(panic mode error-recovery)*
  - kada se koristi na prethodni način, sledbenički skup se naziva i **sinhronizacioni skup**, a njegovi elementi **sinhronizacioni simboli**
  - oni omogućuju nastavak kompajliranja nakon otkrivanja greške s ciljem da se u jednoj kompilaciji otkrije što više grešaka

FOLLOW skup

- *FOLLOW* skup nekog pojma X je skup simbola koji se mogu u pravilima pojaviti odmah nakon pojma X
- Algoritam određivanja *FOLLOW* skupa za pojam X:
  - Ako je X polazni pojam gramatike, dodati simbol $ (*End Of File* oznaka) u *FOLLOW(X)*
  - Za svako pravilo u kome se X nalazi sa desne strane:
    - U pravilima u kojima se iza X nalaze drugi pojmovi i/ili simboli, na primer

      $$Y \rightarrow \alpha X \beta$$

      (gde α i β mogu biti više pojmova ili simbola) dodati sve simbole iz *FIRST*(β) osim ε u *FOLLOW(X).* Ako ε postoji u β, tada dodati *FOLLOW(Y)* u *FOLLOW(X)*
    - U pravilima u kojima se X nalazi na poslednjem mestu, na primer

      $$Y \rightarrow \alpha X$$

      dodati *FOLLOW(Y)* u *FOLLOW(X)*

Primer *FOLLOW* skupa

- primer gramatike izraza:

```
expression    →   term expression1
expression1   →   "+" term expression1
              |   ε
term   →   factor term1
term1  →   "*" factor term1
       |   ε
factor →   "(" expression ")"
       |   id
```

FOLLOW(expression) = *FOLLOW(expression1*) = {), $ }

FOLLOW(term) = *FOLLOW(term1*) = { +,), $ }

FOLLOW(factor) = { +, *,), $ }

Primer *FOLLOW* skupa

- primer gramatike izraza:

```
num_exp
 → exp arop num_exp
arop
 → “+”
 → “-”
exp
 → constant
 → identifier
 → “(” num_exp “)”
```

FOLLOW(num_exp) = {), $ }
FOLLOW(arop) = { constant, identifier, (}
FOLLOW(exp) = { +, - }

Primer LL(1) gramatike – *Pascal* gramatika

- *Pascal* gramatika
- Skener se može predstaviti dijagramom prelaza
  - implementira se **`switch`** iskazom
  - čije **`case`** naredbe odgovaraju pojedinim stanjima skenera
- Parser
  - za svaki pojam posebna funkcija
  - jednoznačno određen poziv funkcije zbog jednoznačnosti FIRST skupa

```
program absolute;

procedure abs(x : int);
  begin
    if x < 0
    then abs := -x
    else abs := x
  end

begin
  abs(-8)
end
```

Literatura

- Aho A.V., Sethi R., Ullman J.D., "*Compilers – principles, techniques, and tools*", Addison-Wesley, Reading, Massachusetts, 1986 (***Dragon's book***) (drugo izdanje: 2006)
- Michael L. Scott, "*Programming Language Pragmatics*", Morgan Kaufmann, San Francisco, 2000
- Torben {\AE}gidius Mogensen, "*Basics of Compiler Design*", lulu.com, Copenhagen, DK, 2009
- Levine, J., "*flex&bison*", O'Reilly Media, 2009
- M. E. Lesk and E. Schmidt, "*Lex - A Lexical Analyzer Generator*", AT&T Bell Laboratories, New Jersey, 1975
- Stephen C. Johnson, "*Yacc: Yet Another Compiler-Compiler*", AT&T Bell Laboratories, New Jersey, 1975
- Kulenović A., "*Osnove projektovanja kompajlera*", Svetlost, Sarajevo, Jugoslavija, 1991
- Muchnick S.S., "*Advanced compiler design & implementation*", Morgan Kaufmann, San Francisco, 1997