

Modello AST: il Perl

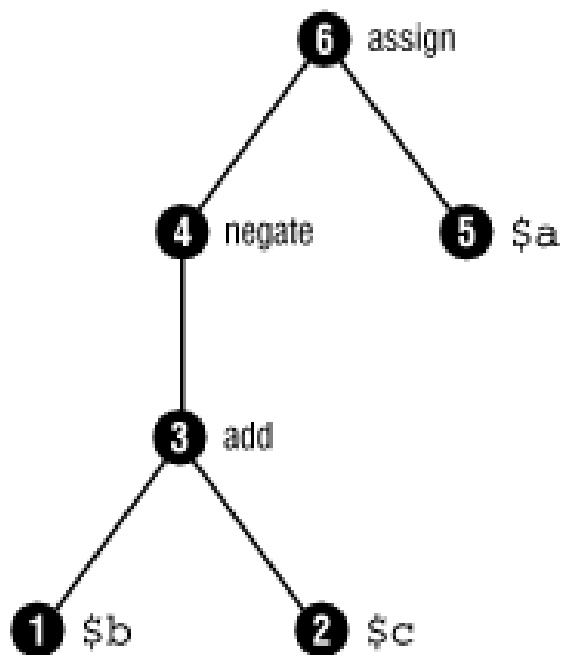
Ciclo di vita di un programma Perl

- Modello di esecuzione: **Abstract Syntax Tree**
 - Compilazione + interpretazione
- **Entrambe le fasi sono gestite dall'eseguibile Perl (*interprete Perl*)**
- **Compilatore**
 - Produce un AST
- **Interprete**
 - Prende in input un AST e lo esegue
 - Esegue **operazioni primitive del Perl (*opcode*)** secondo un ben preciso ordine
 - Interprete → esecutore di opcode

Regole di costruzione dell'AST

- La costruzione dell'AST deve soddisfare **due requisiti fondamentali**:
 - 1) **elencare gli opcode**
 - 2) **specificarne l'ordine di sequenza**
- L'AST è un **albero** in cui:
 - ogni **nodo non foglia** è un **opcode** interno del **Perl** (eseguibile in una istruzione dall'interprete)
 - ogni **nodo foglia** è un **operando**
 - **l'ordine di visita** indica la precedenza degli operatori: **visita in post-ordine** (prima il sottoalbero sinistro, poi quello destro, poi il nodo radice)

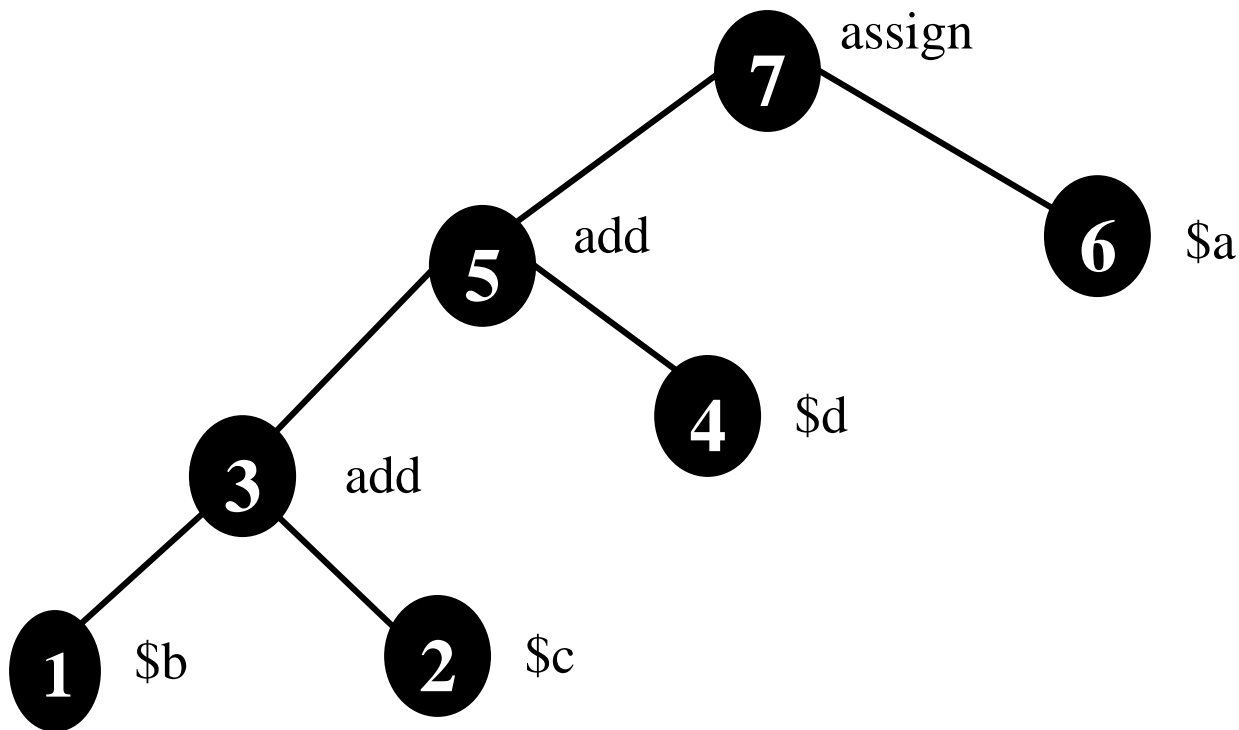
Esempio



AST associato allo statement
 $\$a = -(\$b + \$c)$

Altro esempio

- Statement: $\$a = (\$d + (\$b + \$c))$



Compilazione

Durante la fase di compilazione

- 1) Si effettua l'**analisi** del programma per determinare la presenza di eventuali **errori**
 - Analisi **lassicale, sintattica e semantica** (**type checking parziale**: nessun controllo di dichiarazione, identificazione di tipo limitata a valori costanti o già noti)
- 2) Può essere **eseguito codice**
 - Direttiva **use** per il caricamento dei moduli: è interpretata subito dopo la sua scansione
 - Blocchi di codice contrassegnati con particolari parole chiave (ad es. **BEGIN**) sono interpretati ed eseguiti subito dopo la loro scansione

Analisi

L'analisi avviene tramite l'uso di **tre moduli distinti**

- **Lexer:** identifica i token all'interno del codice sorgente (*lexical analyzer*)
- **Parser:** associa gruppi contigui di token a costrutti (espressioni, statement) in base alla grammatica del Perl (analisi sintattica e semantica)
- **Optimizer:** riordina e riduce i costrutti prodotti dal parser, con l'obiettivo di produrre sequenze di codice equivalenti più efficienti

NOTA: *nel Perl la fase di ottimizzazione viene inserita all'interno dell'analisi che produce l'AST*

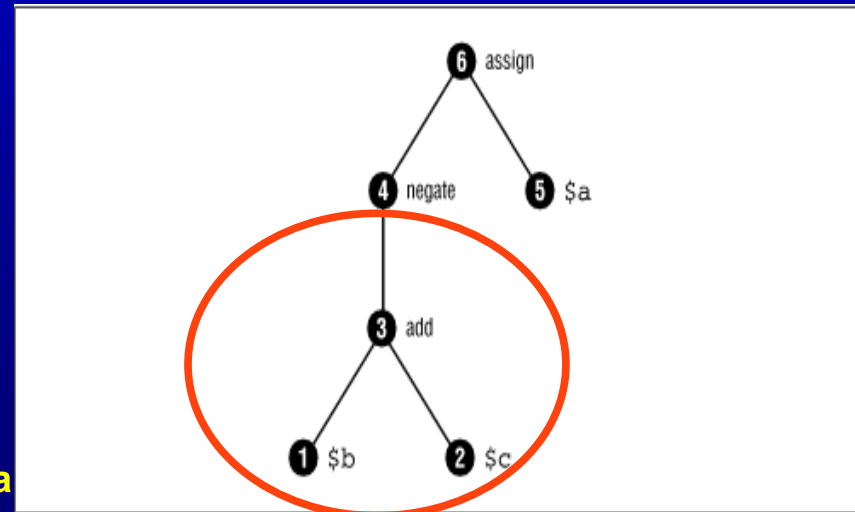
Analisi

- I moduli Parser e Optimizer sono responsabili della costruzione dell'AST e della sua ottimizzazione attraverso 3 fasi:
 - *Bottom-Up Parsing, Top-Down Optimizer, Peephole Optimizer*
- Non tutte le fasi sono sequenziali e consecutive: alcuni passi sono interallacciati
 - Es. l'optimizer a volte non può entrare in azione fino a quando il parser non ha raggiunto un certo punto nella costruzione dell'albero
 - Elaborazione di espressioni, blocchi o subroutine (parti dell'albero AST)

Bottom-Up Parsing

Fase Bottom-Up Parsing

- Il parser riceve in ingresso i token prodotti dal lexer dai quali costruisce l'AST
- Il parsing è di tipo “**bottom-up**” perché l'AST viene costruito partendo dalle foglie
 - $\$a = -(\$b + \$c)$
 - *Considero $\$b$ e $\$c$ (operandi) poi add che rappresenta l'opcode radice del primo sottoalbero di sinistra*



Bottom-Up Parsing

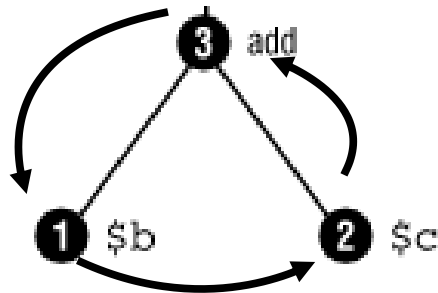
Fase Bottom-Up Parsing

- Dopo la costruzione di un nodo non foglia (opcode) – realizzazione di un sottoalbero - si verifica se la **semantica dell'opcode relativo è congruente** (ad es., numero corretto di parametri per una funzione o per un operatore)
- Poi si costruisce l'**ordine di visita dei nodi per quel sottoalbero**
- L'ordine di visita
 - Viene memorizzato nell'AST stesso (puntatori)
 - Memorizzazione necessaria per garantire **maggior efficienza all'interprete**

Ordine di visita

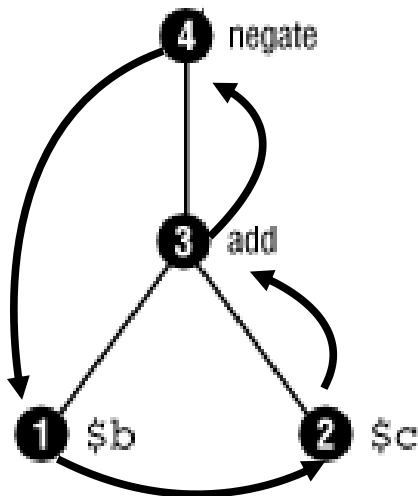
- Non appena un sottoalbero dell'AST è generato, viene creata una **struttura ciclica** che collega i nodi secondo l'ordinamento di esecuzione (**visita in post-ordine**) a partire dalla radice
 - Il nodo radice viene collegato al primo nodo da visitare, poi il primo nodo da visitare viene collegato al secondo nodo da visitare, e *così via*
 - Quando viene aggiunto un altro *sottoalbero AST al precedente*, il **ciclo** radice-primo nodo si spezza, e viene **ricostruito con il nuovo albero**
- In questo modo l'interprete potrà individuare il prossimo nodo da visitare in **maniera efficiente**, (**costo ($O(1)$)**) a partire dal nodo radice

Esempio



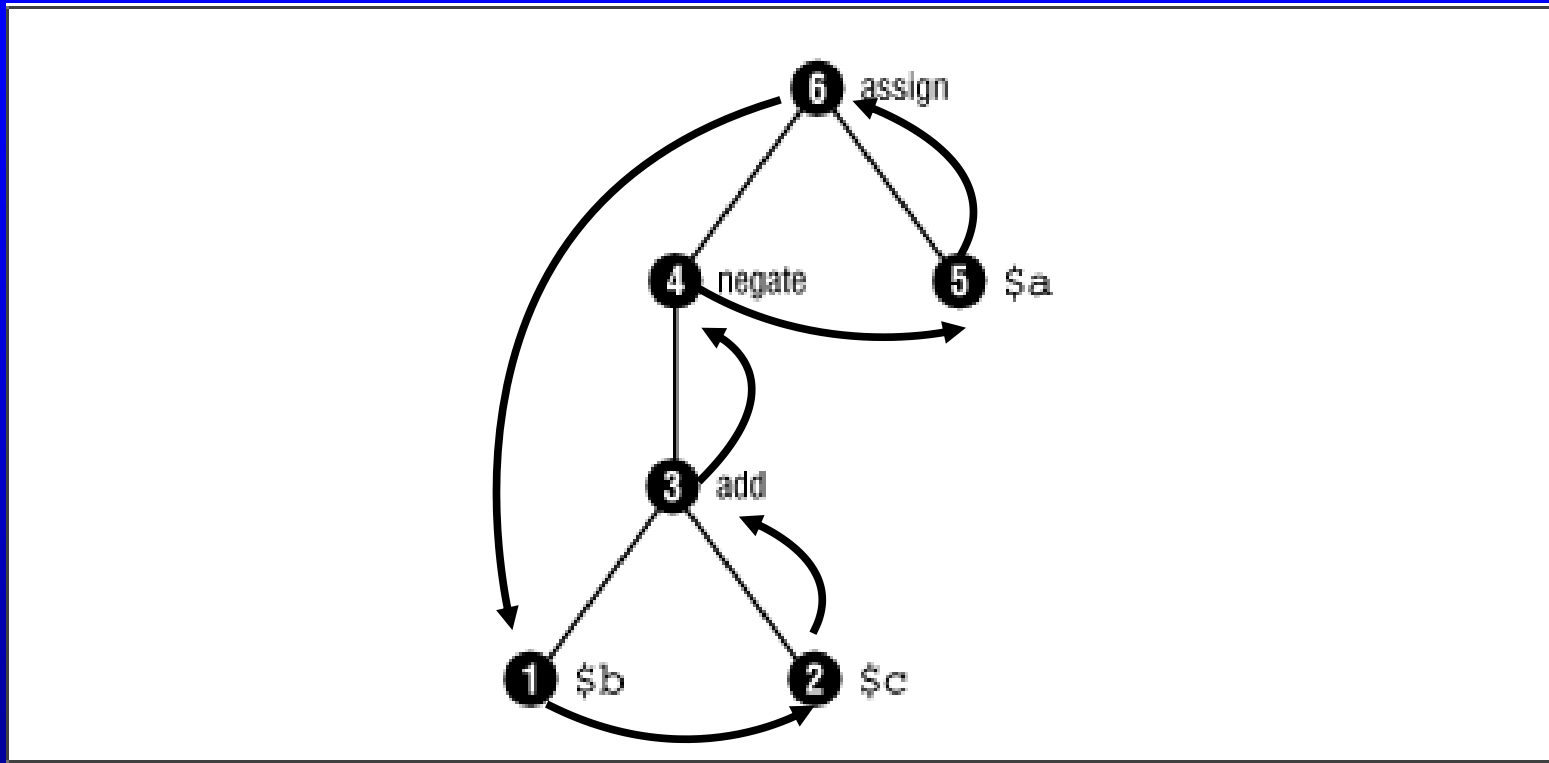
Costruzione dell'AST associato allo statement
 $\$a = -(\$b + \$c)$

Esempio



Costruzione dell'AST associato allo statement
 $\$a = -(\$b + \$c)$

Esempio



Costruzione dell'AST associato allo statement
 $\$a = -(\$b + \$c)$

Top-Down Optimizer

- Fase Top-down Optimizer
 - Fase interallacciata con la precedente
 - Non appena un sottoalbero dell'AST viene prodotto, viene scandito dalla radice alle foglie per eventuali ottimizzazioni
 - Si tratta di ottimizzazioni locali
 - Context propagation: una volta identificato il “contesto” di un nodo, esso viene propagato verso il basso ai nodi figli (top-down)
 - Es. identificazione del tipo del valore di ritorno di una funzione:
substr(foo(), 4, 5) → foo() restituisce una stringa
[sintassi: substr(expr, offset, length)]

Peephole Optimizer

- **Fase Peephole (*spioncino*) Optimizer**
 - Ottimizzazione su **porzioni di codice piccole**
 - Fase che inizia dopo la costruzione dell'AST
 - Il peephole optimizer percorre l'albero seguendo il **flusso di esecuzione** (segue la struttura dei next-opcode) e considera piccole porzioni di albero per cercare di **semplificare/ridurre gli opcode**
 - ES. Porzioni di codice che hanno uno spazio di memorizzazione locale per le variabili: singole subroutine

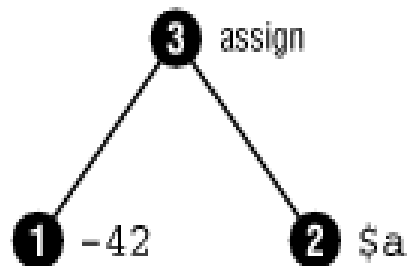
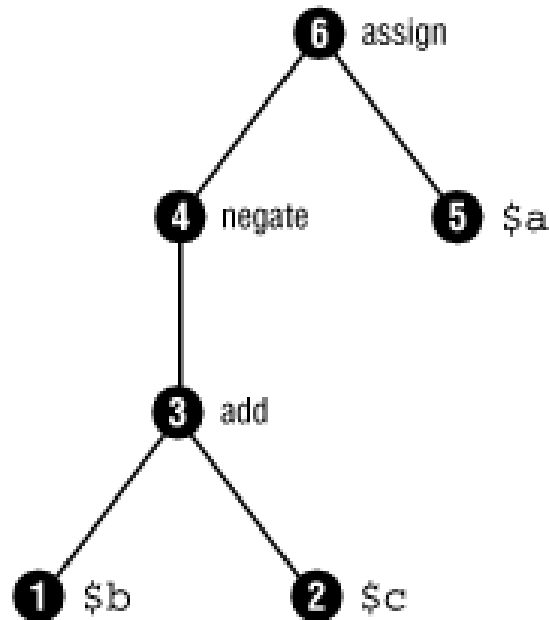
Peephole Optimizer

- **Fase Peephole (*spioncino*) Optimizer**
 - Il peephole optimizer può effettuare diversi tipi di ottimizzazione su piccole porzioni di codice
 - **Strength reduction**: rimpiazza operazioni “lente” con operazioni equivalenti e più veloci
 - Esempio:
mul ax,2 → shift_left ax
 - **Null sequences**: elimina operazioni inutili
 - Esempio:
mov r0, i; [...]; mov i, r0 → mov r0, i

Peephole Optimizer

- **Constant folding:** se trova operazioni con operandi tutti **costanti**, calcola l'espressione finale e sostituisce all'espressione di partenza il suo risultato
- Ad esempio, se nell'AST corrispondente a $a = - (b + c)$ si scopre che b e c sono due espressioni costanti tali che la loro somma sia sempre pari a 42, si sostituisce il valore costante, ottenendo il seguente **AST ottimizzato equivalente**

Peephole Optimizer



Peephole Optimizer

- **Combine Operations:** rimpiazza un insieme di operazioni con una operazione singola equivalente
- **Algebraic Laws:** usa leggi algebriche per semplificare o riordinare le istruzioni
 - $(a * 2) + (b * 2) \rightarrow (a + b) * 2$
- **Special Case Instructions:** usa istruzioni “speciali” progettate per specifici operandi
 - Stringhe, dotted names, ...

Interprete

- Quando la compilazione è **terminata**, l'AST viene **passato all'interprete** che esegue gli opcode dell'AST nella **sequenza specificata**
- L'interprete usa una **macchina basata su stack** per eseguire il codice
- L'idea di base è che ogni opcode manipola degli operandi, che vengono inseriti in uno stack secondo una modalità molto simile alla **notazione polacca inversa**
 - Se devo eseguire un calcolo su a e b:
Push a, Push b
Opcode (pop a, pop b, calcolo, push risultato)

Uso di stack

- L'interprete Perl crea **diversi stack**, fra cui:
 - **Operand stack**: memorizzazione degli operandi usati dagli operatori e dei risultati
 - **Save stack**: memorizzazione delle variabili il cui scope è stato alterato da altre
 - Es.: variabile globale / il cui valore è stato offuscato da una variabile locale / all'interno di una funzione
 - **Return stack**: memorizzazione degli indirizzi di ritorno delle funzioni

Compilazioni a run-time

- L'interprete Perl può effettuare **compilazioni di espressioni a run-time** attraverso la funzione **eval**

eval EXPR

```
$str = '$a++; $a + $b'; # str contiene due istruzioni
```

```
$a = 10; $b = 20;
```

```
$c = eval $str; # assegna a $c 31
```

- **eval** effettua una **compilazione** della espressione **prima** di passare alla sua **esecuzione**
- Usato per **motivi di sicurezza**: nel caso ad esempio in cui l'espressione contenga input dall'esterno, se ne deve controllare la validità prima di eseguirla
- In caso di errore: salta il resto dell'espressione e inserisce un messaggio di errore in **\$EVAL_ERROR**

Esempio runtime Perl

- Script perl_demo.pl con **subroutine speciali**

```
print    "start main running here\n";
die      "main now dying here\n";
die      "XXX: not reached\n";
END      { print "1st END: done running\n" }
CHECK    { print "1st CHECK: done compiling\n" }
INIT     { print "1st INIT: started running\n" }
END      { print "2nd END: done running\n" }
BEGIN    { print "1st BEGIN: still compiling\n" }
INIT     { print "2nd INIT: started running\n" }
BEGIN    { print "2nd BEGIN: still compiling\n" }
CHECK    { print "2nd CHECK: done compiling\n" }
END      { print "3rd END: done running\n" }
```


Subroutine speciali

- Richiamate (eseguite) in momenti specifici
 - *Durante la fase di compilazione (compile-time)*
 - *Durante la fase di esecuzione (run-time)*
- **Subroutine BEGIN:** eseguita (interpretata) all'inizio della fase di compilazione
 - Punto corretto per inizializzare/modificare l'ambiente prima che inizi la compilazione
- **Subroutine INIT:** eseguita all'inizio della fase di esecuzione del programma
 - Punto corretto per eventuali inizializzazioni di variabili, strutture dati, ...

Subroutine speciali

- **Subroutine CHECK:** eseguita alla fine della fase di compilazione del programma
 - Punto corretto per eventuali controlli sulla buona riuscita della compilazione (es. compilazioni condizionali)
 - A differenza di BEGIN e INIT, *l'ordine di esecuzione di CHECK multiple è inverso a quello delle chiamate*
- **Subroutine END:** eseguita alla fine della esecuzione del programma
 - Punto corretto per eseguire operazioni di chiusura (connessioni di rete, pulizia file, ...)
 - Eseguita anche in caso di istruzione die
 - Ordine di esecuzione inverso (come per CHECK)

Esecuzione di perl_demo.pl

1st BEGIN: still compiling
2nd BEGIN: still compiling
2nd CHECK: done compiling
1st CHECK: done compiling
1st INIT: started running
2nd INIT: started running
start main running here
main now dying here
3rd END: done running
2nd END: done running
1st END: done running

Output di
perl perl_demo.pl

Controllo sintattico

L'opzione **-c** permette di invocare soltanto il servizio di controllo sintattico del programma:

perl -c file.pl

- Non esegue, ma ci informa del risultato dell'analisi: *file.pl syntax OK*

1st BEGIN: still compiling

2nd BEGIN: still compiling

2nd CHECK: done compiling

1st CHECK: done compiling

perl_demo.pl syntax OK

Output di
perl -c perl_demo.pl

Opzione -MO=Concise

Si può anche ispezionare l'albero AST prodotto dalla fase di compilazione con l'opzione -MO=Concise

- Sintassi

- perl -MO=Concise file.pl
- perl -MO=Concise -e '\$a = \$b + 42'
- perl -MO=Concise, -exec -e '\$a = \$b + 42'

Opzione basic (default)

Opzione exec

- Due output (rendering) con diverse opzioni: -basic (default) e -exec

- -exec mostra gli opcode nell'ordine in cui sono eseguiti (visita in postordine dell'albero)
- -basic mostra la struttura dell'albero dalla radice alle foglie (visita in preordine dell'albero)

Output -exec

```
% perl -MO=Concise,-exec -e '$a = $b + 42'
```

```
1 <0> enter
```

```
2 <;> nextstate(main 1 -e:1) v
```

```
3 <#> gvsv[*b] s
```

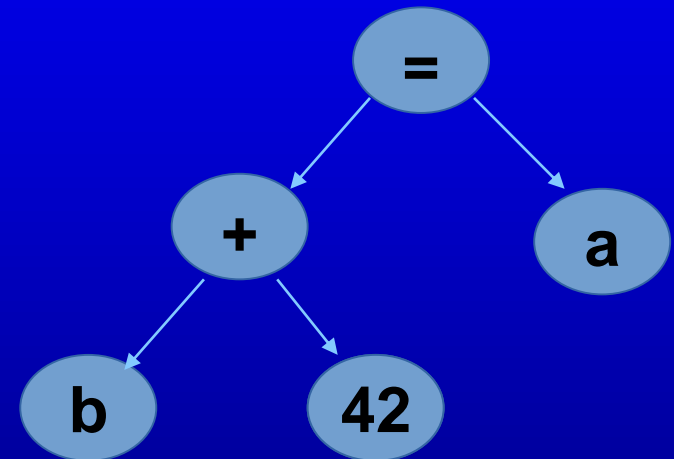
```
4 <$> const[IV 42] s
```

```
5 <2> add[t3] sK/2
```

```
6 <#> gvsv[*a] s
```

```
7 <2> sassign vKS/2
```

```
8 <@> leave[1 ref] vKP/REFC
```



Output layout

- **Prima colonna:** opcode sequence number (utili nel caso di loop)
- **Seconda colonna:** <x> tipo di opcode (<2> BINOP, <1> UNOP, ...)
- **Terza colonna:** nome opcode (es. add, assign) eventualmente seguito da informazioni specifiche tra parentesi
- **Quarta colonna:** opcode flags
- Tiene traccia di **ottimizzazioni** che tolgono opcode: gli opcode tolti compaiono con sequence number '-' (non eseguiti) e 'ex-opcodeName' come nome

Opzione -basic

- L'output mostra gli opcode come sono nell'albero offrendo un rendering top-down: visita in **preordine (radice, sottoalbero sinistro e sottoalbero destro)**
- Riflette il modo in cui lo stack può essere usato per **valutare le espressioni dal punto di vista semantico (top-down)**
 - Es: L'opcode add opera sui due termini che si trovano sul livello inferiore dell'albero
- La freccia nell'ultima colonna dell'output indica il **sequence number del prossimo opcode** da eseguire
 - Questa opzione permette di ricavare l'ordine di **esecuzione**

Output -basic

```
% perl -MO=Concise -e '$a = $b + 42'
8 <@> leave[1 ref] vKP/REFC ->(end)
1 <0> enter ->2
2 <;> nextstate(main 1 -e:1) v ->3
7 <2> sassign vKS/2 ->8
5 <2> add[t1] sK/2 ->6
- <1> ex-rv2sv sK/1 ->4
3 <$> gvsv(*b) s ->4
4 <$> const(IV 42) s ->5
- <1> ex-rv2sv sKRM*/1 ->7
6 <$> gvsv(*a) s ->7
```

Effetto del Peephole
Optimizer: opcode
rv2sv (più generico)
rimpiazzato
dal più efficiente
opcode gvsv