Code Generation

for a stack machine,

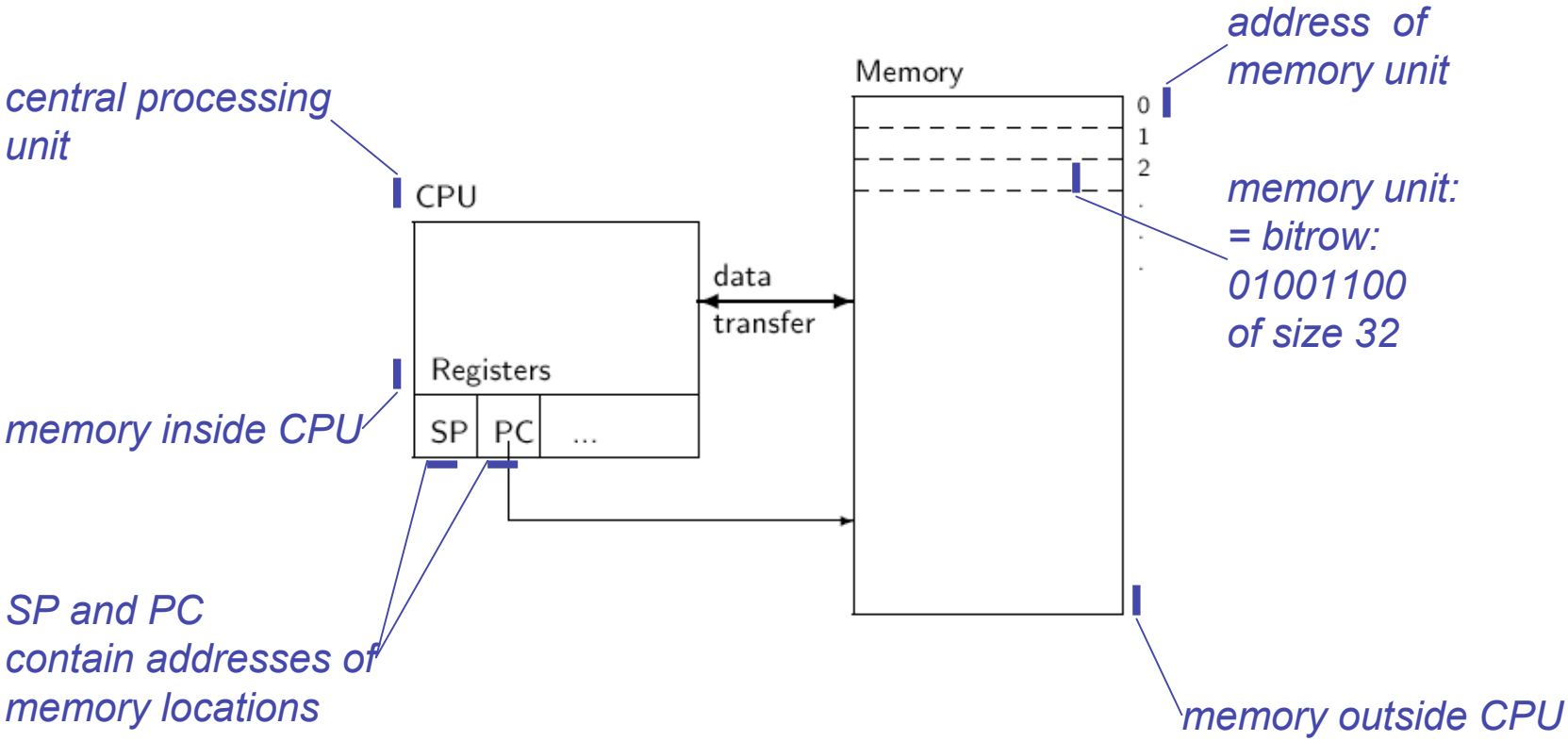
basic constructs

Content

- Machine model
  - Simple stack machine
- Translation
  - From simple (source) language (SL) to target language (machine)
- Language constructs (today)
  - Operator + operands
  - Conditional
  - Let
  - Assignment, sequence

Machine model

- CPU (processor) + memory



Memory

- Memory = array of memory units
- Address of memory unit = index into array
- Memory unit = array of bits (bitrow)

$$
\begin{aligned}
b &= b_{n-1} \ldots b_0 \\
value_{Int}(b) &= \sum_{i=0}^{n-1} b_i 2^i
\end{aligned}
$$

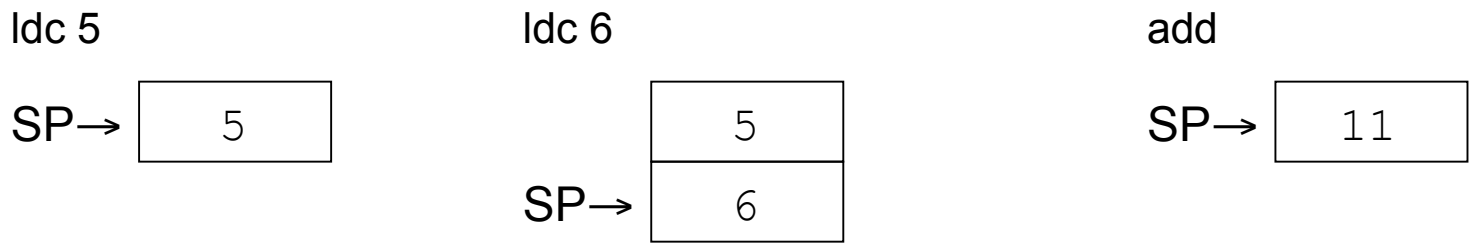
- Bitrow has meaning (for humans)

Registers

- Located inside CPU for fast access
- Programcounter (PC)

```
while ( ... )
{
  instruction = M[ PC ] ;
  PC += size of instruction ;
  execute instruction ;
}
```

- Stackpointer (SP)



Machine

- Executes instructions
  - Instructions are also encoded as bitrows
- Instructions are low level
  - E.g. copy a value from/to memory
  - Or perform addition
- High level constructs are absent
  - E.g. let, if, function
- But the building blocks are available

Assembly

- Machine is programmed using low level language, assembly language

```
                LDC   4
                LDC   5
                EQ                ; if 4 == 5
                BRF   elselabel   ; then
                LDC   6
                BRA   filabel
elselabel:      LDC   7           ; else
                LDC   8
                MUL
filabel:        ...               ; fi
```

label, name for location

instruction

arguments for instruction (inline)

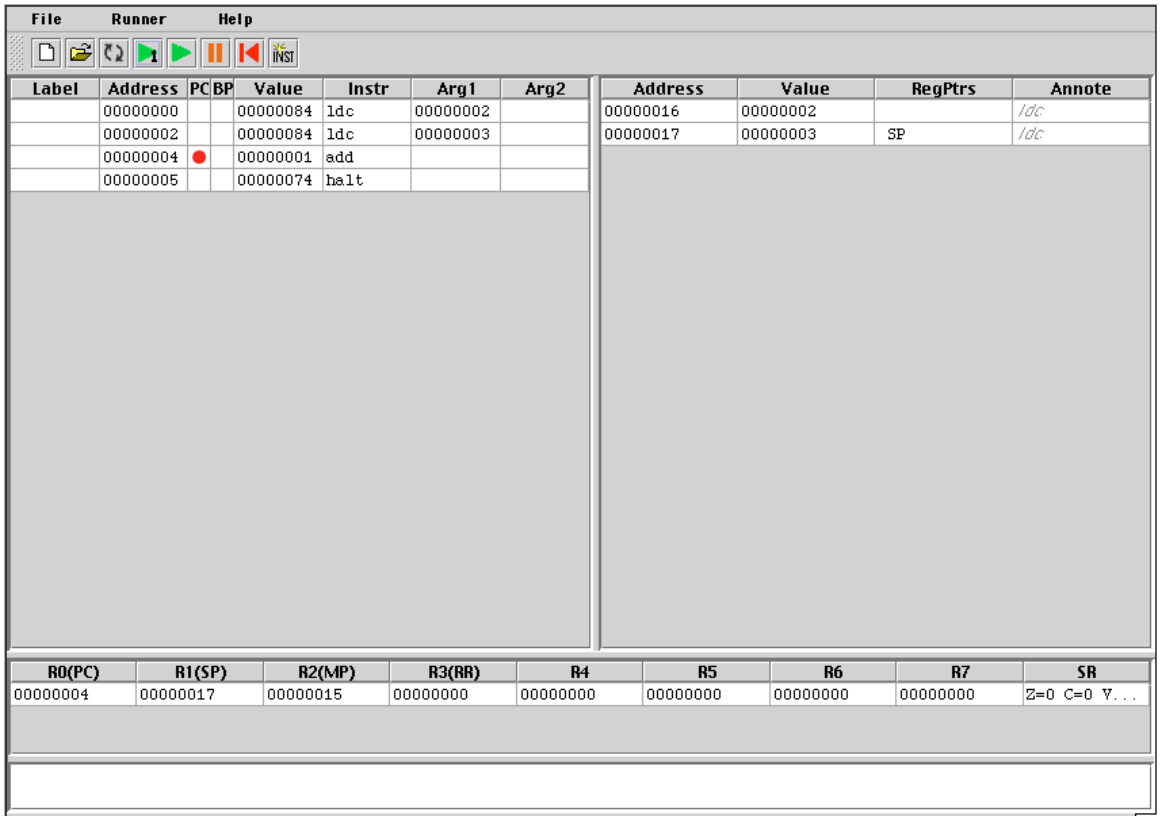
comment

Instruction categories

- Copying data
  - Onto the top of stack: load (ld) variants
  - From the top of stack: store (st) variants
- Arithmetic
  - Add (+), Sub (-), LT (<), ...
- Control
  - Changes the programcounter
  - Branch (br) or jump instructions

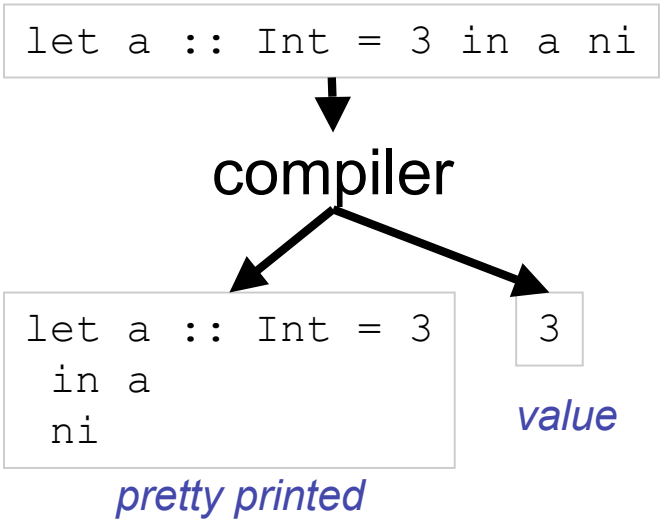
Simple Stack Machine

- Interprets machine instructions
- Shows content of memory

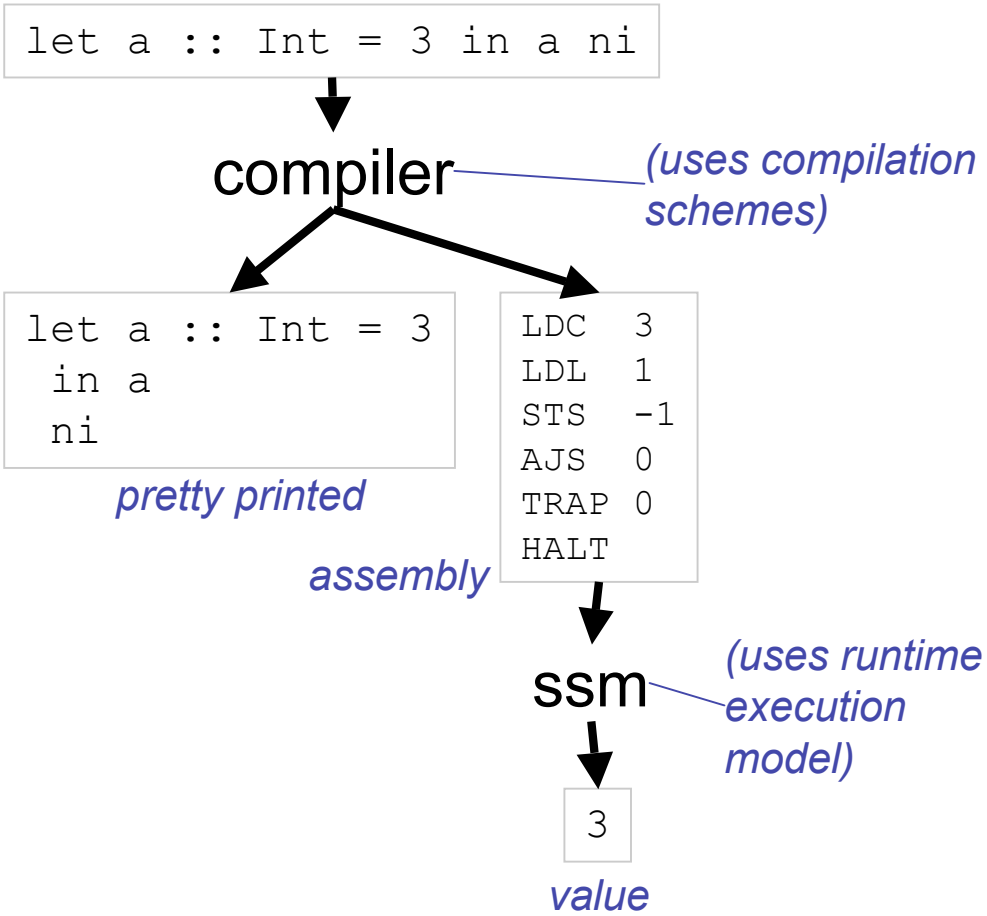


Overview

- Until now



- From now



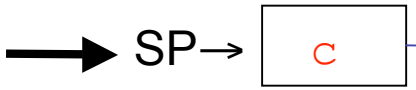
Constants

- Integer

- Boolean

load constant

- Runtime effect on stack

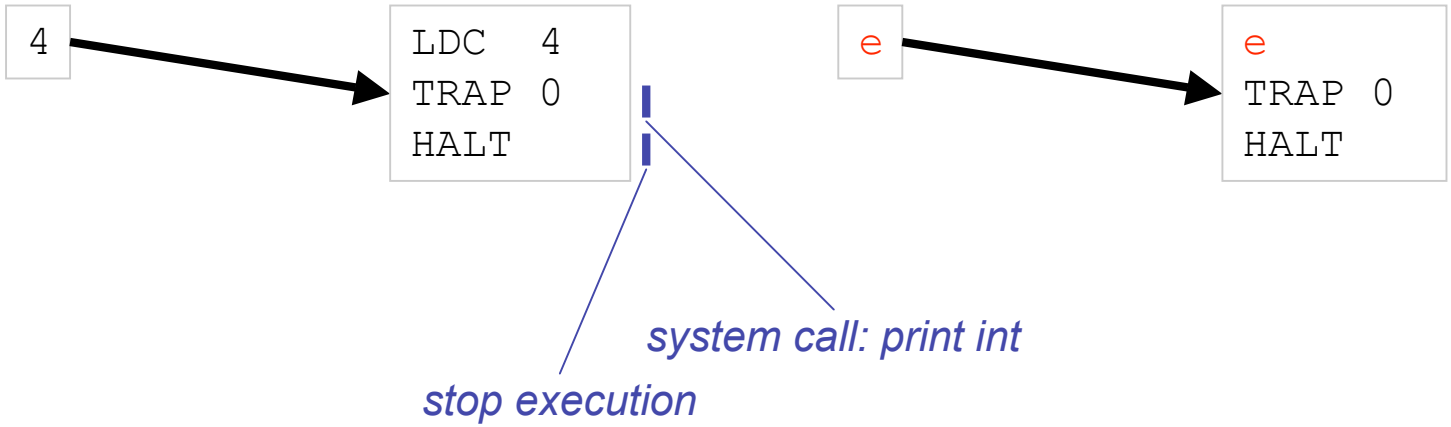


compilation scheme:

assumes effect on stack at execution time

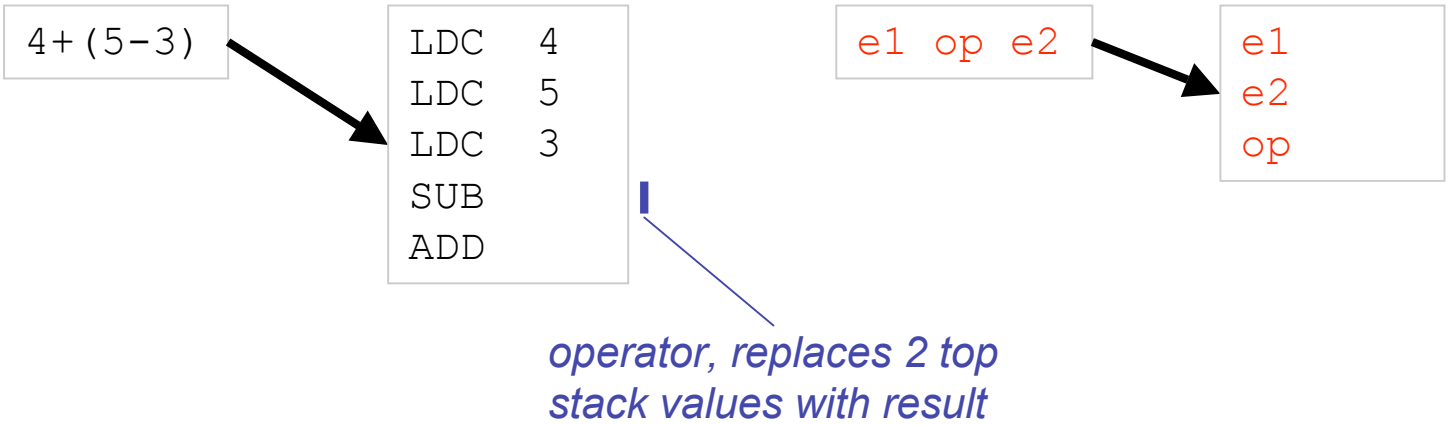
Program and output

- Program compilation scheme

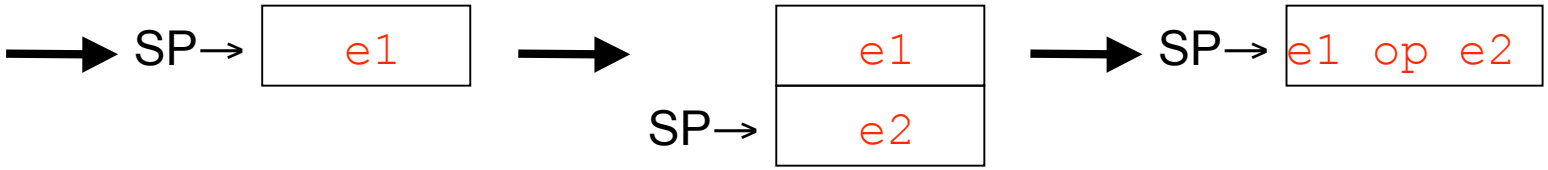


Operator and operands

- Operator compilation scheme



- Runtime effect on stack



Stack invariant

- All constructs take from the stack what they need
- And replace it by their result

Code generation

- Similar to pretty printed text
- Code for parent is obtained from the code for the children

Attribution

- Representation of instruction

```
data S
  | HALT
  | TRAP        Int
  | LDC         Int
  | ADD
  | SUB

type CodeGen = [S] -> [S]
```

why function composition?

- Attribution follows compilation scheme

```
SEM Expr [ | | code: CodeGen ]
  | Add   lhs.code = (@e1.code . @e2.code . (ADD:))
  | Sub   lhs.code = (@e1.code . @e2.code . (SUB:))
  | Val   lhs.code = (LDC @int:)
```

Delayed vs immediate list creation

- Composition + delayed list creation

```
type CodeGen = [S] -> [S]

| Add   lhs.code = (@e1.code . @e2.code . (ADD:))
| Val   lhs.code = (LDC int:)

final_code = ....
show ... (final_code [])
```

. delays creating the list until the last moment and then builds it cheaply (using :)

- List concatenation

```
type CodeGen = [S]

| Add   lhs.scode = (@e1.code ++ @e2.code ++ [ADD])
| Val   lhs.scode = [LDC @int]

final_scode = ....
show ... (final_scode)
```

++ on long lists is expensive, $O(n^2)$

Output

- Dumping on file

```
SEM Root [ || ppcode: PP_Doc ]
  | Root        lhs . ppscode   = ( vlist . map (text.show)
                                  . @expr.code
                                  . (TRAP 0:)
                                  . (HALT:)
                                  $ []
                                  )
main
    = do ...
         (codepp,exprpp,tp) <- parseIO (pRoot ()) tokens
         putStrLn ( (disp exprpp 300 ""))
         writeFile (basename ++ ".ssm") (disp codepp 80 "")
         putStrLn ("Generated code in file " ++ basename ++".ssm"))
```

Conditional expression

- If compilation scheme

```
if 3 < 4
then 5
else 6
fi
```

```
 LDC 3
 LDC 4
 LT
 BRF L1
 LDC 5
 BRA L2
L1:
 LDC 6
L2:
```

branch if false

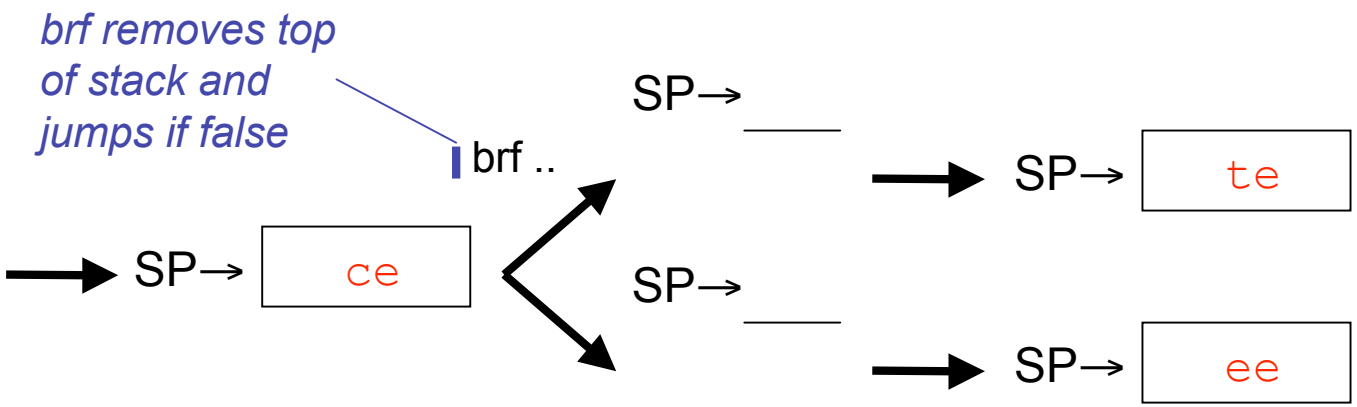
branch always

```
if ce
then te
else ee
fi
```

```
 ce
 BRF L1
 te
 BRA L2
L1:
 ee
L2:
```

Conditional expression

- Runtime effect on stack



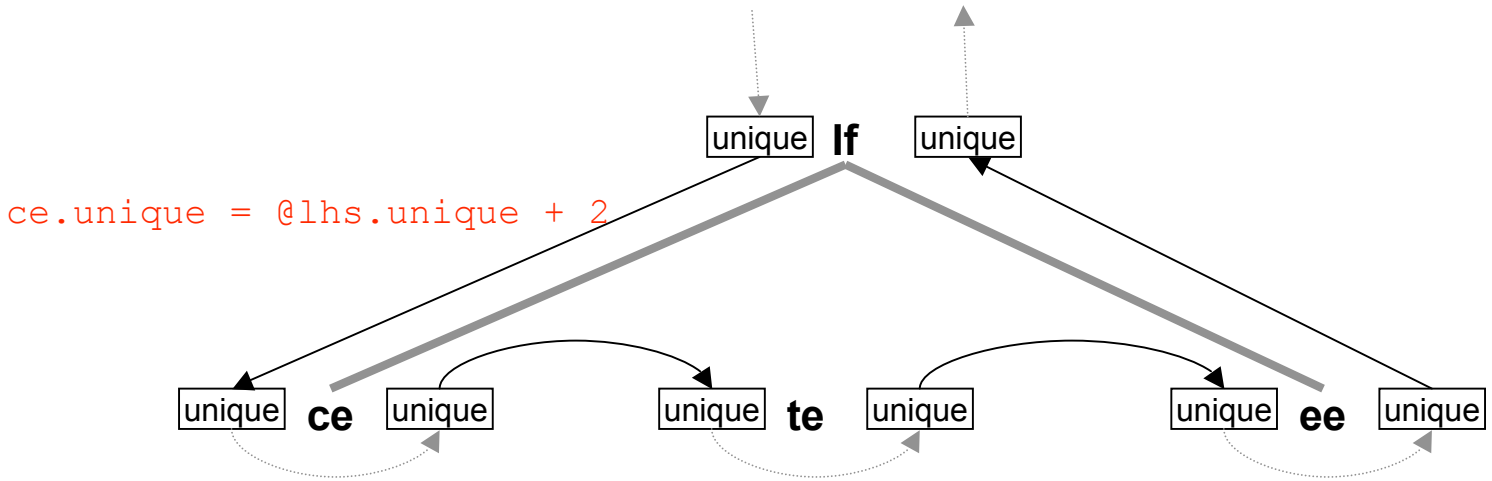
Attribution

```
SEM Expr
  | Add    lhs.code = (@e1.code . @e2.code . (ADD:))
  | Sub    lhs.code = (@e1.code . @e2.code . (SUB:))
  | LT     lhs.code = (@e1.code . @e2.code . (LT:))
  | GT     lhs.code = (@e1.code . @e2.code . (GT:))
  | Val    lhs.code = (LDC @int:)
  | If     lhs.code = ( @ce.code
                      . (BRF @lAftThen:)
                      . @te.code
                      . (BRA @lAftElse:)
                      . (DEFLABEL @lAftThen:)
                      . @ee.code
                      . (DEFLABEL @lAftElse:)
                      )
DATA S
  | ...
  | BRA          Label
  | BRF          Label
  | DEFLABEL     Label
```

```
 ce
 BRF L1
 te
 BRA L2
L1:
 ee
L2:
```

Labels

- Labels have to be unique
- Pass unique value around for label generation



- Take unique and pass (unique + 2)

Label generation

- Create label from unique

```
data Label    = Lab Int

instance Show Label
  where show (Lab l) = 'L' : (show l)

SEM Root
  | Root   expr.unique = 0

ATTR Expr Decl [ | unique: Int | ]

SEM Expr
  | If    loc.lAftThen  = Lab @lhs.unique
             .lAftElse  = Lab (@lhs.unique+1)
          ce.unique     = @lhs.unique + 2
```

Let expression

- Let compilation scheme

```
let a=3
 in a+a
 ni
```

→

```
LDC  3
LDS  0
LDS -1
ADD
STS -1
```

store on stack (relative to SP)

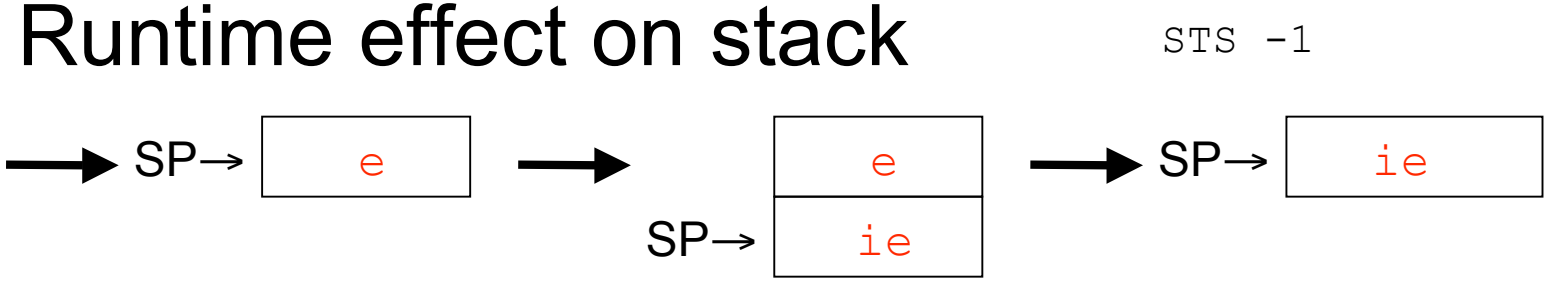
load from stack (relative to SP)

```
let i = e
 in ie
 ni
```

→

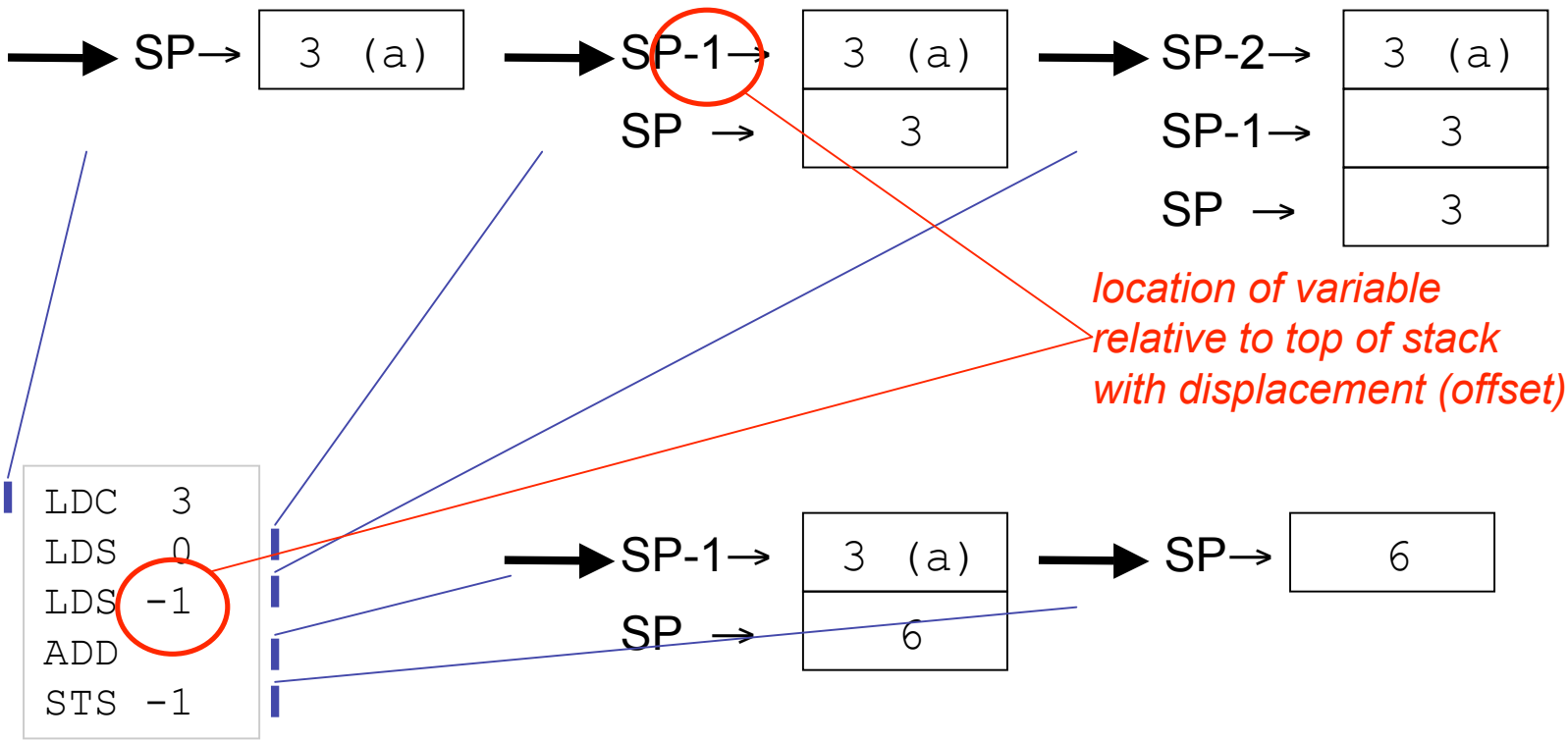
```
e
ie
STS -1
```

- Runtime effect on stack



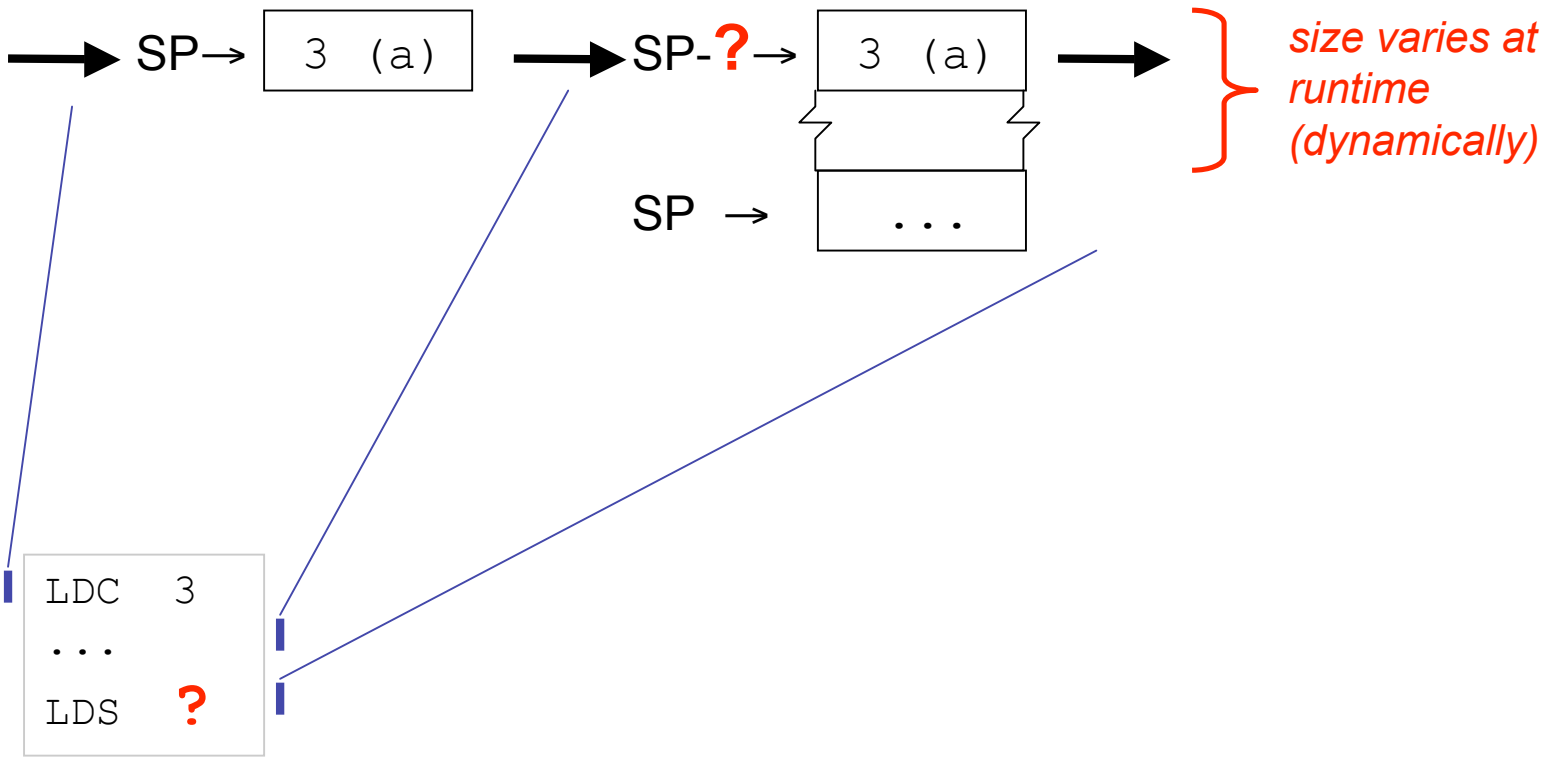
Access to variables

- Runtime effect on stack



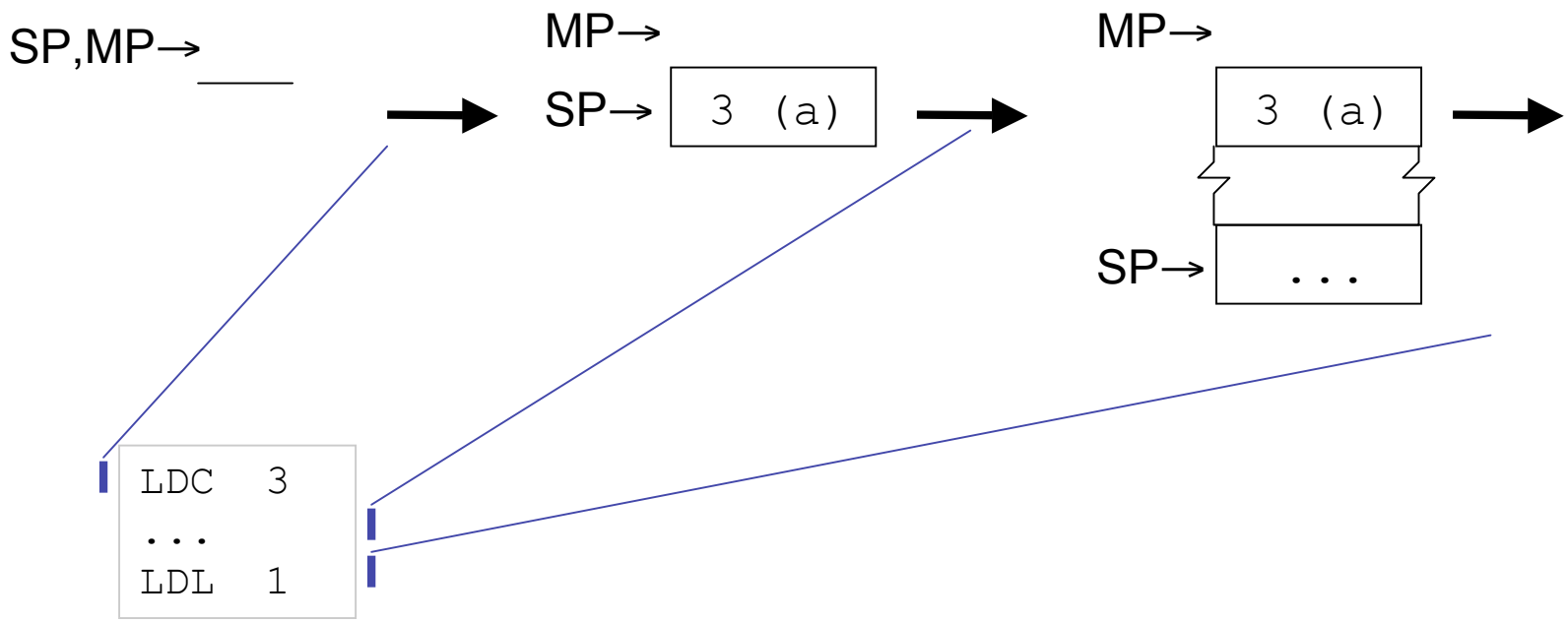
Access to variables

- Offset may be unknown (statically)



Access to variables

- Need to remember/use the 'base': Markpointer (MP) register



Let expression

- Let using MP

```
let a=3
 in a+a
 ni
```

```
LDC  3
LDL  1
LDL  1
ADD
STS -1
```

load local (relative to MP)

```
let a = 3
 in a + let b = 4
        in a + b
        ni
 ni
```

```
LDC  3
LDL  1
LDC  4
LDL  1
LDL  3
ADD
STS  -1
ADD
STS  -1
```

Attribution

```
type Displ     = Int
type IdInfo    = (Type,Displ)
type Env       = [(String,IdInfo)]

ATTR Expr [ env: Env  displ: Displ | | ]
ATTR Decl [ | displ: Displ  env: Env | ]

SEM Root
  | Root  expr.env   = []
              .displ = 0

SEM Expr
  | Ident loc.(msg,idtp,iddis)= env_lookup @lhs.env @string
  | Add   e2.displ            = @lhs.displ + 1
  | ...

SEM Decl
  | Decl  loc.(msg,env)       = env_add @lhs.env string @tp @displ
             .displ           = @lhs.displ + 1
          expr.displ          = @displ
          lhs.env             = @env
             .displ           = @displ
```

Attribution

```
ATTR Expr Decl [ | | code: CodeGen ]

SEM Expr
  | Add   lhs.code = (@e1.code . @e2.code . (ADD:))
  | ...
  | Val   lhs.code = (LDC @int:)
  | Ident lhs.code = (LDL @iddis:)
  | Let   lhs.code = @decl.code . @inexpr.code . (STS (-1):)
  | If    lhs.code = ...

SEM Decl
  | Decl  lhs.code = @expr.code
```

Side effect

- Side effect free: all expressions always return same value
- Side effect by changing memory location associated with variable
- By assigning a new value to it

```
let a :: Int = 3
 in a := a + 1
  ; a
 ni
```

sequence of expressions or assignments

assignment, puts right hand side in left hand side, leaving nothing on stack

Assignment expression

- Assignment compilation scheme

```
a := a+1
```

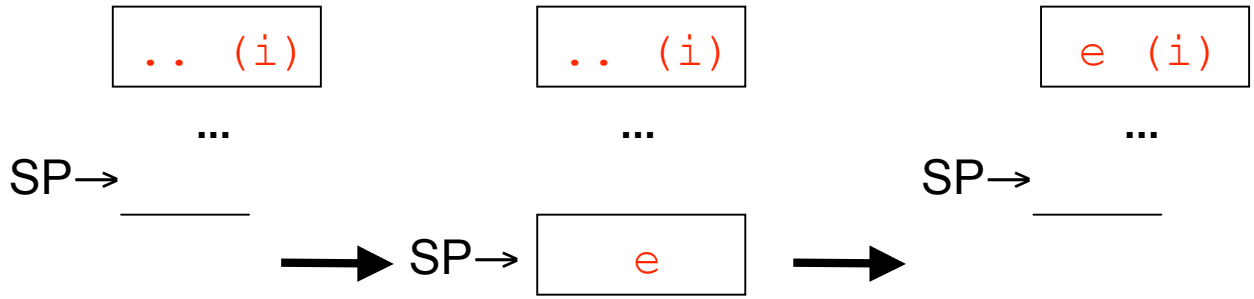
```
LDL  1
LDC  1
ADD
STL  1
```

store local (relative to MP)

```
i := e
```

```
e
STL i
```

- Runtime effect on stack



Sequence design alternatives

- ';' acts as operator throwing away left operand
- Allowable combinations for a ; b

```
a ; b
```

a \ b	x := e	e
x := e	return void	return b
e	remove a, return void	remove a, return b

Assigment design alternatives

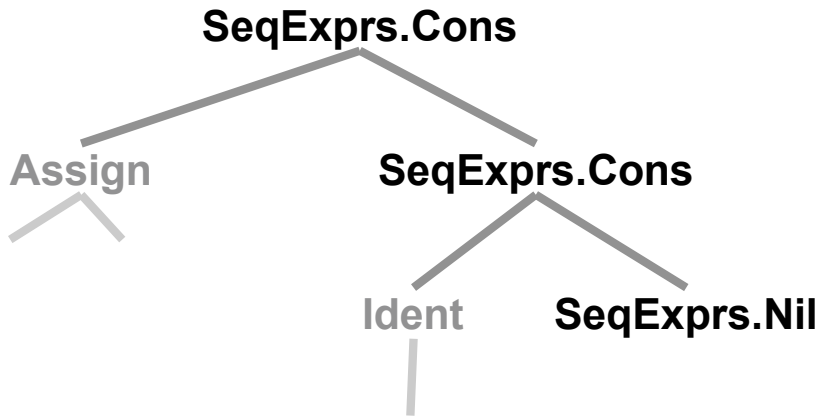
- What is the value of :=

```
x := (y:=e)
```

Sequence

- Common AG structure: List like

```
  a := a + 1
; a
```




```
DATA Expr
   | ...
   | Assign        String       Expr
   | Seq    exprs: Exprs
TYPE Exprs = [Expr]
```

Sequence attribution

- Many small changes, follow list structure

```
SEM SeqExprs [ || restps: Types ]
  | Cons   lhs.restps  = @expr.restp : @exprs.restps
  | Nil    lhs.restps  = []

SEM SeqExprs [ || pps: PP_Docs ]
  | Cons  lhs.pps = @expr.pp : @exprs.pps
  | Nil   lhs.pps = []

SEM SeqExprs [ || code: SCode ]
  | Nil   lhs.code   = id
  | Cons  loc.exprsz = codeSize @expr.restp
          lhs.code  = ( @expr.code
                      . ( if null @exprs.restps || @exprsz == 0
                          then id
                          else (AJS (-(@exprsz)):)
                        )
                      . @exprs.code
                      )
```

correcting the stack