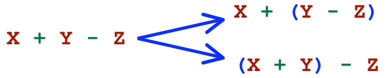
Infix / Postfix Notation

Consider Binary Operators

Infix Notation: `operand operator operand`

Can be ambiguous!

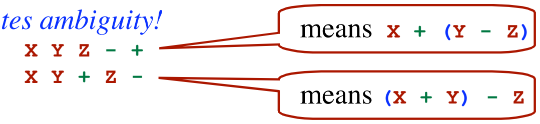


```
X + Y - Z
X + (Y - Z)
(X + Y) - Z
```

Need rules of precedence, associativity, parentheses.

Postfix Notation: `operand operand operator`

Eliminates ambiguity!



```
X Y Z - +
X Y + Z -
```

means `X + (Y - Z)`

means `(X + Y) - Z`

Assumption: *No confusion about how many operands an operator requires.*

binary- versus unary-

Infix: `X + -(Y - Z)`   `X + (Y - -Z)`

Postfix: $X\ Y\ Z\ -_{bin}\ -_{un}\ +$ $X\ Y\ Z\ -_{un}\ -_{bin}\ +$

Converting Expressions to Postfix

Let E be an infix expression.
Define POSTFIX(E) to be the same expression in postfix.
(Ignore unary operators.)

- If E is a variable or constant...
  then POSTFIX (E) = E

- If E is of the form E_1 op E_2 ...
  then POSTFIX (E_1 op E_2) = POSTFIX (E_1) || POSTFIX (E_2) || op

- If E is of the form (E_1) ...
  then POSTFIX ((E_1)) = POSTFIX (E_1)

String concatenation

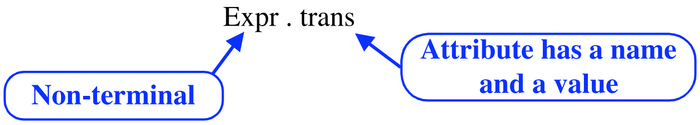
Syntax-Directed Translation

Problem/Goal:

Translate infix expressions into postfix.
The input is described by a CFG.

Approach:

Start with the grammar.
Each production is augmented with *semantic rules*.
Each non-terminal has an associated *attribute*.



Semantic rules added to grammar productions
...tell how to compute the attributes' values.

```
trans = "a b + c +";
```

An Example Attribute Grammar

CFG Grammar

```
Expr  → Expr + Term
      → Expr - Term
      → Term
Term  → ID
```

Terminals:

"+", "-", ID
Token attribute: ID.svalue

Non-terminals:

Expr
Term

Attributes:

Expr.t
Term.s

Attribute Values:

Strings, e.g., "x y + z -"
Concatenation: ||

Attribute Grammar

Expr → Expr **+** Term

Expr → Expr **-** Term

Expr → Term

Term → ID

Attribute Grammar

Expr_0 → Expr_1 **+** Term

Expr_0 → Expr_1 **-** Term

Expr_0 → Term

Term → ID

Subscripts added
...to tell different
non-terminals apart

Attribute Grammar

$\text{Expr}_0 \rightarrow \text{Expr}_1 + \text{Term}$	$\text{Expr}_0\text{.t} = \text{Expr}_1\text{.t} \mid\mid \text{Term.s} \mid\mid$ "+";
$\text{Expr}_0 \rightarrow \text{Expr}_1 - \text{Term}$	$\text{Expr}_0\text{.t} = \text{Expr}_1\text{.t} \mid\mid \text{Term.s} \mid\mid$ "-";
$\text{Expr}_0 \rightarrow \text{Term}$	$\text{Expr}_0\text{.t} = \text{Term.s};$
$\text{Term} \rightarrow \text{ID}$	Term.s = ID.svalue;

Example

Expr_0	$\rightarrow \text{Expr}_1 + \text{Term}$
Expr_0	$\rightarrow \text{Expr}_1 - \text{Term}$
Expr_0	$\rightarrow \text{Term}$
Term	$\rightarrow$ ID

Translate: "**X - Y + W**"

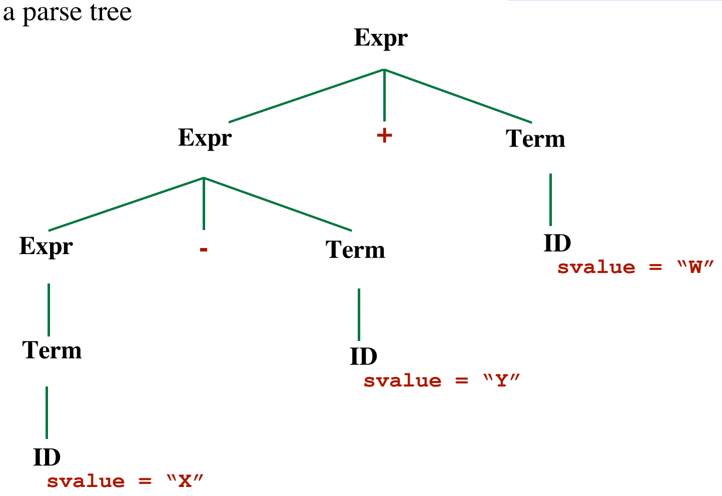
Step 1: Find a parse tree

Example

$\text{Expr}_0 \rightarrow \text{Expr}_1 + \text{Term}$
$\text{Expr}_0 \rightarrow \text{Expr}_1 - \text{Term}$
$\text{Expr}_0 \rightarrow \text{Term}$
$\text{Term} \rightarrow \text{ID}$

Translate: "**X - Y + W**"

Step 1: Find a parse tree



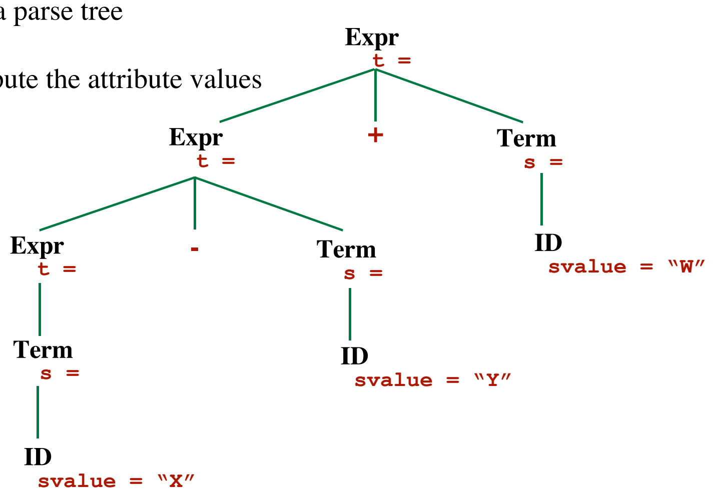
Example

$\text{Expr}_0 \rightarrow \text{Expr}_1 + \text{Term}$
$\text{Expr}_0 \rightarrow \text{Expr}_1 - \text{Term}$
$\text{Expr}_0 \rightarrow \text{Term}$
$\text{Term} \rightarrow \text{ID}$

Translate: "**X - Y + W**"

Step 1: Find a parse tree

Step 2: Compute the attribute values



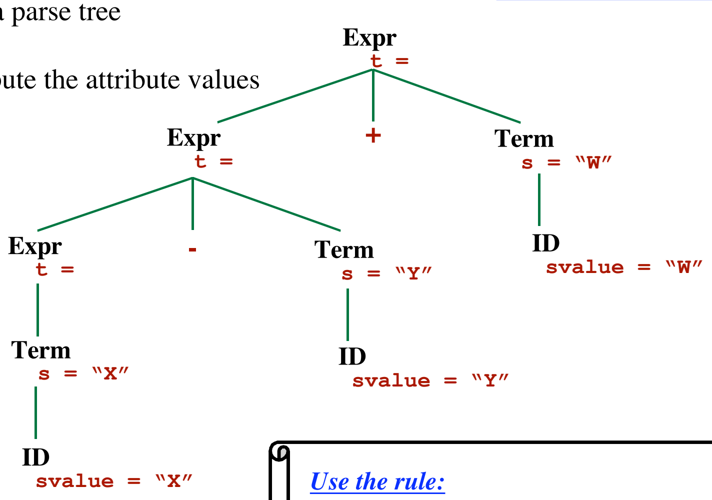
Example

$Expr_0 \rightarrow Expr_1 + Term$
$Expr_0 \rightarrow Expr_1 - Term$
$Expr_0 \rightarrow Term$
$Term \rightarrow ID$

Translate: "**X - Y + W**"

Step 1: Find a parse tree

Step 2: Compute the attribute values



Use the rule:
Term → ID `Term.s = ID.svalue;`

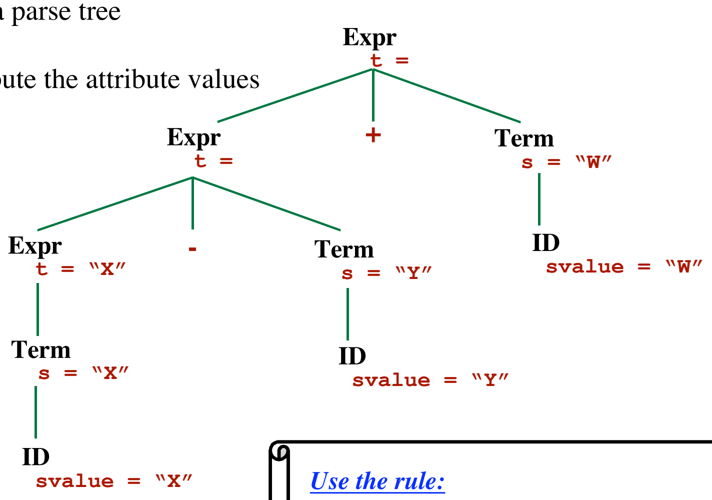
Example

$Expr_0 \rightarrow Expr_1 + Term$
$Expr_0 \rightarrow Expr_1 - Term$
$Expr_0 \rightarrow Term$
$Term \rightarrow ID$

Translate: "**X - Y + W**"

Step 1: Find a parse tree

Step 2: Compute the attribute values



Use the rule:
$Expr_0$ → Term `Expr0.t = Term.s;`

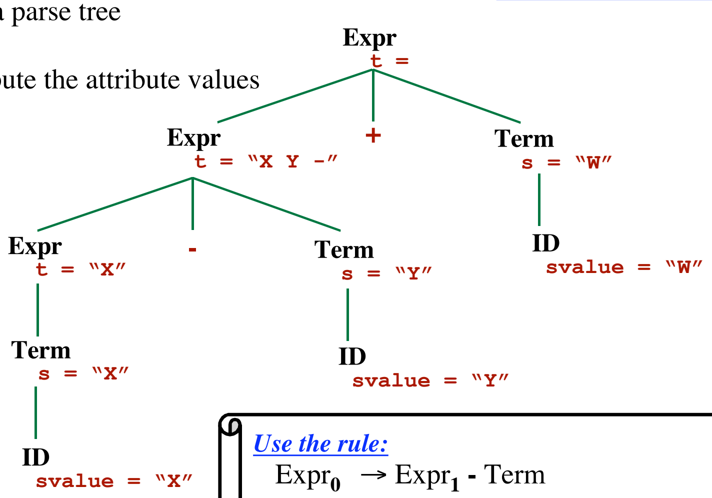
Example

$Expr_0$	→ $Expr_1$ **+** Term
$Expr_0$	→ $Expr_1$ **-** Term
$Expr_0$	→ Term
Term	→ ID

Translate: "**X - Y + W**"

Step 1: Find a parse tree

Step 2: Compute the attribute values



Use the rule:

$Expr_0$ → $Expr_1$ **-** Term

`Expr`$_0$`.t = Expr`$_1$`.t || Term.s || "-";`

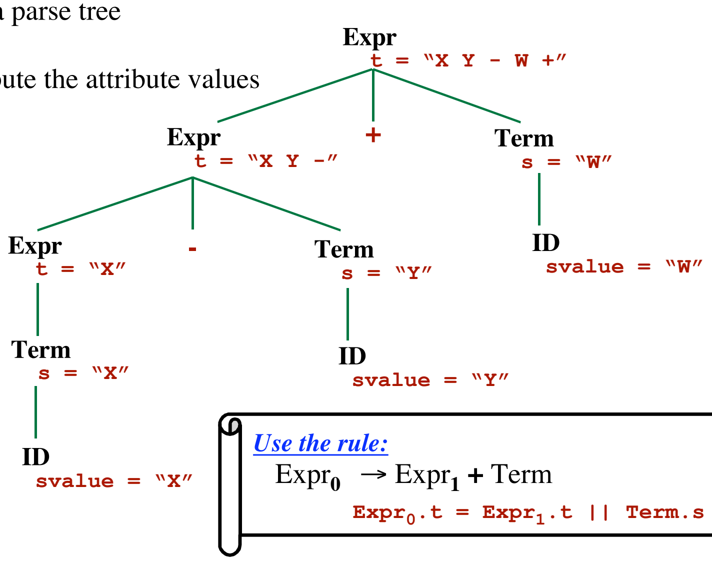
Example

$Expr_0$	→ $Expr_1$ **+** Term
$Expr_0$	→ $Expr_1$ **-** Term
$Expr_0$	→ Term
Term	→ ID

Translate: "**X - Y + W**"

Step 1: Find a parse tree

Step 2: Compute the attribute values



Use the rule:

$Expr_0$ → $Expr_1$ **+** Term

`Expr`$_0$`.t = Expr`$_1$`.t || Term.s || "+";`

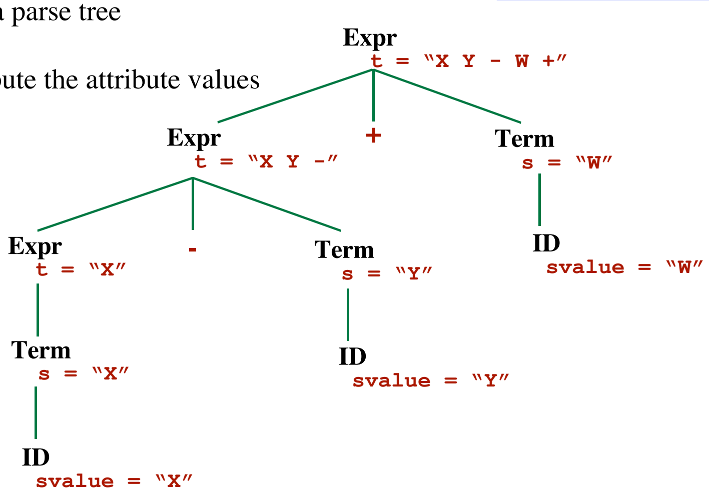
Example

Expr_0	→ Expr_1 + Term
Expr_0	→ Expr_1 - Term
Expr_0	→ Term
Term	→ ID

Translate: "**X - Y + W**"

Step 1: Find a parse tree

Step 2: Compute the attribute values



Synthesized v. Inherited

Synthesized Attributes

(see previous example)

Compute the attributes bottom-up

From leaves toward root

Example Semantic Rule:

Expr_0 → Expr_1 - Term `Expr`$_0$`.t = Expr`$_1$`.t || Term.s || "-";`

All rules compute the attribute of the left-hand side

... as a function of the attributes from the right-hand side.

X → A B C `X.t = f(A.t, B.t, C.t);`

Information flows *up the tree*.

A *Bottom-Up* Approach

Inherited Attributes

Information flows *down the tree*

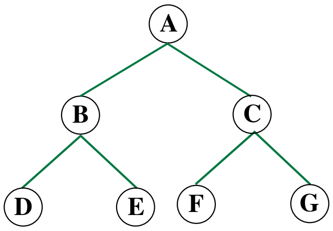
Example:

X → A B C `B.t = f(X.t);`

A *Top-Down* Approach

Depth-First Traversal

```
function Visit (N: Node)
  for each child of N do
    Visit (child)
  endFor
  "process" N
endFunction
```



Sythesized Attributes

Evaluate children first

Then move up the tree

... and take care of parents' attributes

Translator Schemes

Embed semantic actions into grammar rules.

Example

X → A `{ print("+") }` B `{ print(".") }` C

Enclose actions in braces `{ ... }`

Arbitrary code (e.g., Java statements)

Translator Schemes

Embed semantic actions into grammar rules.

Example

X → A `{ print("+") }` B `{ print(".") }` C

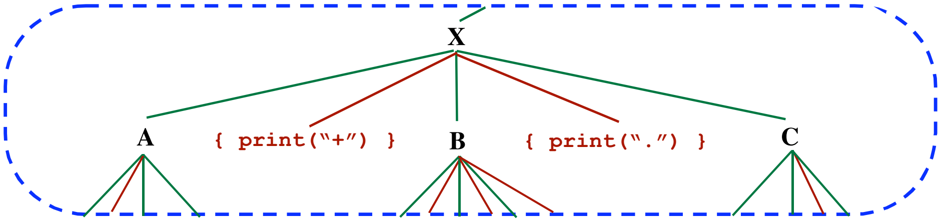
Enclose actions in braces `{ ... }`

Arbitrary code (e.g., Java statements)

How to execute?

Step 1: Construct a parse tree

Add the actions to the parse tree



Translator Schemes

Embed semantic actions into grammar rules.

Example

X → A `{ print("+") }` B `{ print(".") }` C

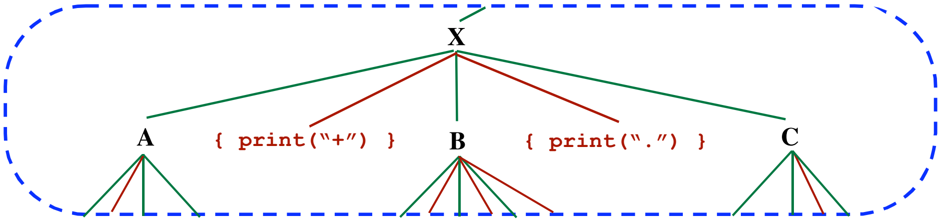
Enclose actions in braces `{ ... }`

Arbitrary code (e.g., Java statements)

How to execute?

Step 1: Construct a parse tree

Add the actions to the parse tree



Step 2: Perform a depth-first traversal

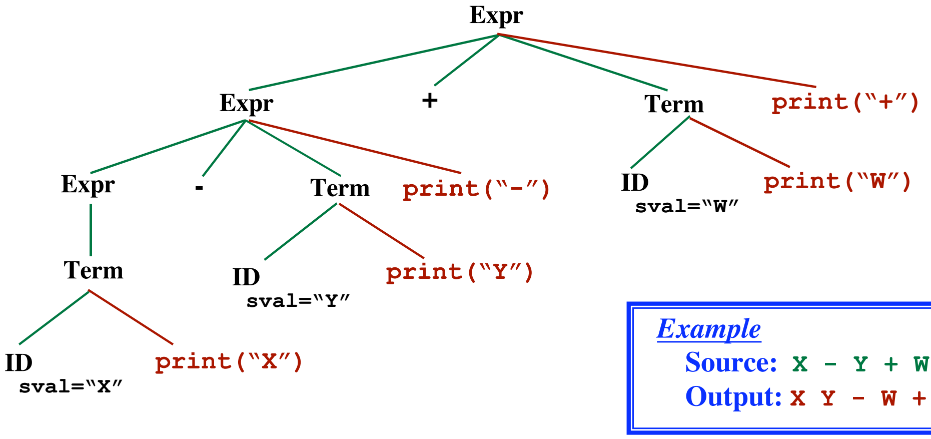
Execute actions as they are encountered in traversal

Example: Convert Infix Expressions to Postfix

Expr → Expr **+** Term `{ print("+") }`
Expr → Expr **-** Term `{ print("-") }`
Expr → Term
Term → ID `{ print(ID.svalue) }`

Example: Convert Infix Expressions to Postfix

Expr → Expr **+** Term `{ print("+") }`
Expr → Expr **-** Term `{ print("-") }`
Expr → Term
Term → ID `{ print(ID.svalue) }`



Example
Source: `X - Y + W`
Output: `X Y - W +`

Assume we have a translator scheme...
Assume we have a parser...

Can we execute the actions while we do the parsing?

Assume we have a translator scheme...
Assume we have a parser...

Can we execute the actions while we do the parsing?

Depth-first traversal → Recursive descent parser!

Assume we have a translator scheme...
Assume we have a parser...
Can we execute the actions while we do the parsing?

Depth-first traversal → Recursive descent parser!

Example:

Expr → Expr **+** Term `{ print("+") }`
→ Expr **-** Term `{ print("-") }`
→ Term
Term → ID `{ print(ID.svalue) }`

Assume we have a translator scheme...
Assume we have a parser...
Can we execute the actions while we do the parsing?

Depth-first traversal → Recursive descent parser!

Example:

Expr → Expr **+** Term `{ print("+") }`
→ Expr **-** Term `{ print("-") }`
→ Term
Term → ID `{ print(ID.svalue) }`

First, we'll need to eliminate left-recursion:

Expr → Term Rest
Rest → **+** Term `{ print("+") }` Rest
→ **-** Term `{ print("-") }` Rest
→ ε
Term → ID `{ print(ID.svalue) }`

Implementation

```
Expr  → Term Rest
Rest  → + Term { print("+") } Rest
      → - Term { print("-") } Rest
      → ε
Term  → ID { print(ID.svalue) }
```

```
function ParseExpr ()
  ParseTerm ()
  ParseRest ()
endFunction
```

Implementation

```
Expr  → Term Rest
Rest  → + Term { print("+") } Rest
      → - Term { print("-") } Rest
      → ε
Term  → ID { print(ID.svalue) }
```

```
function ParseTerm ()
  if nextToken == ID then
    s = token.svalue
    MustHave (ID)
    print (s)
  else
    Error "Expecting ID"
  endIf
endFunction
```

Implementation

```
Expr  → Term Rest
Rest  → + Term { print("+") } Rest
      → - Term { print("-") } Rest
      → ε
Term  → ID { print(ID.svalue) }
```

```
function ParseRest ()
  if nextToken == '+' then
    MustHave ('+')
    ParseTerm ()
    print ("+")
    ParseRest ()
  elseIf nextToken == '-' then
    MustHave ('-')
    ParseTerm ()
    print ("-")
    ParseRest ()
  else
    // Epsilon -- do nothing
  endIf
endFunction
```

Repeating...

```
function ParseExpr ()
  ParseTerm ()
  ParseRest ()
endFunction


function ParseRest ()
  if nextToken == '+' then
    MustHave ('+')
    ParseTerm ()
    print ("+")
    ParseRest ()
  elseIf nextToken == '-' then
    MustHave ('-')
    ParseTerm ()
    print ("-")
    ParseRest ()
  else
    // Epsilon -- do nothing
  endIf
endFunction
```

Optimizing “Tail Recursion”

```
function ParseExpr ()
  ParseTerm ()
  ParseRest ()
endFunction


function ParseRest ()
  if nextToken == ‘+’ then
    MustHave (‘+’)
    ParseTerm ()
    print (“+”)
    ParseRest ()
  elseIf nextToken == ‘-’ then
    MustHave (‘-’)
    ParseTerm ()
    print (“-”)
    ParseRest ()
  else
    // Epsilon -- do nothing
  endIf
endFunction
```

```
function ParseRest ()
  while true
    if nextToken == ‘+’ then
      MustHave (‘+’)
      ParseTerm ()
      print (“+”)
    elseIf nextToken == ‘-’ then
      MustHave (‘-’)
      ParseTerm ()
      print (“-”)
    else
      return
    endIf
  endWhile
endFunction
```

In-Lining...

```
function ParseExpr ()
  ParseTerm ()
  ParseRest ()
endFunction


function ParseRest ()
  if nextToken == ‘+’ then
    MustHave (‘+’)
    ParseTerm ()
    print (“+”)
    ParseRest ()
  elseIf nextToken == ‘-’ then
    MustHave (‘-’)
    ParseTerm ()
    print (“-”)
    ParseRest ()
  else
    // Epsilon -- do nothing
  endIf
endFunction
```

```
function ParseExpr ()
  ParseTerm ()
  while true
    if nextToken == ‘+’ then
      MustHave (‘+’)
      ParseTerm ()
      print (“+”)
    elseIf nextToken == ‘-’ then
      MustHave (‘-’)
      ParseTerm ()
      print (“-”)
    else
      return
    endIf
  endWhile
endFunction
```

Generating Target Code

Output of compiler

- Assembly code (e.g., SPARC code)
- Machine code (e.g., 0x3b4E0F0F...)
- “Bytecodes” (e.g., PUSH, POP, GOTO, ...)

Generating Target Code

Output of compiler

- Assembly code (e.g., SPARC code)
- Machine code (e.g., 0x3b4E0F0F...)
- “Bytecodes” (e.g., PUSH, POP, GOTO, ...)

Bytecodes

Higher level than machine-specific code.
Example: Java Bytecodes
Software to execute the instructions
    Interpreter (aka: “Virtual Machine”, “Emulator”)
Or translate the bytecodes into machine-specific code
    “Just-in-time” compilers

Generating Target Code

Output of compiler

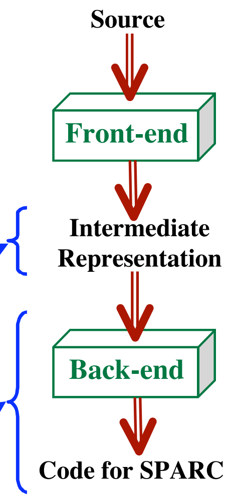
- Assembly code (e.g., SPARC code)
- Machine code (e.g., 0x3b4E0F0F...)
- "Bytecodes" (e.g., PUSH, POP, GOTO, ...)

Bytecodes

Higher level than machine-specific code.
Example: Java Bytecodes
Software to execute the instructions
  Interpreter (aka: "Virtual Machine", "Emulator")
Or translate the bytecodes into machine-specific code
  "Just-in-time" compilers

Abstract Stack Machine

A virtual machine architecture
Based on a stack
  Can be used as intermediate code



Code for Abstract Stack Machine

Replace with back-end For Intel, Power-PC, ...

Abstract Stack Machine

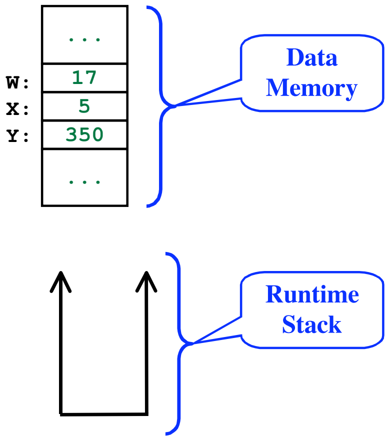
- Limited / simple instruction set
- No registers
- Stack of data values
- Program memory
- Data memory

(3 + (6 - X))

3 6 X - +

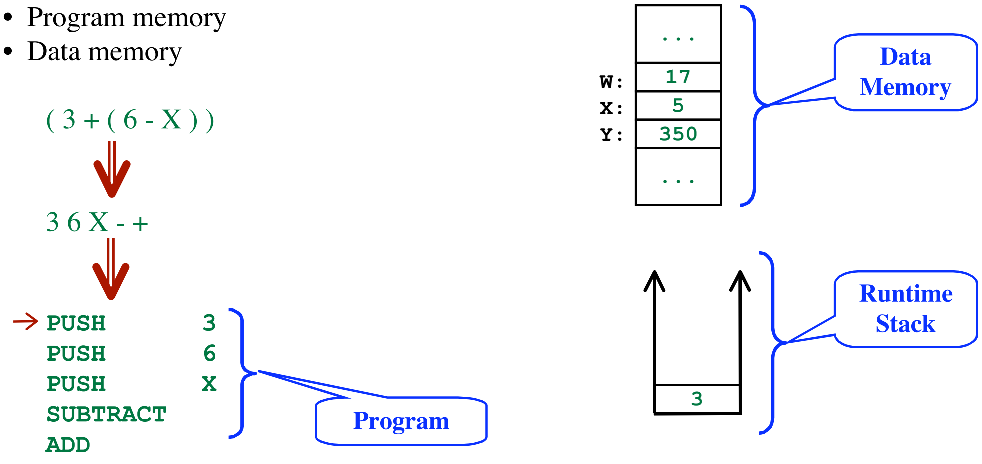
```
PUSH      3
PUSH      6
PUSH      X
SUBTRACT
ADD
```

Program



Abstract Stack Machine

- Limited / simple instruction set
- No registers
- Stack of data values
- Program memory
- Data memory



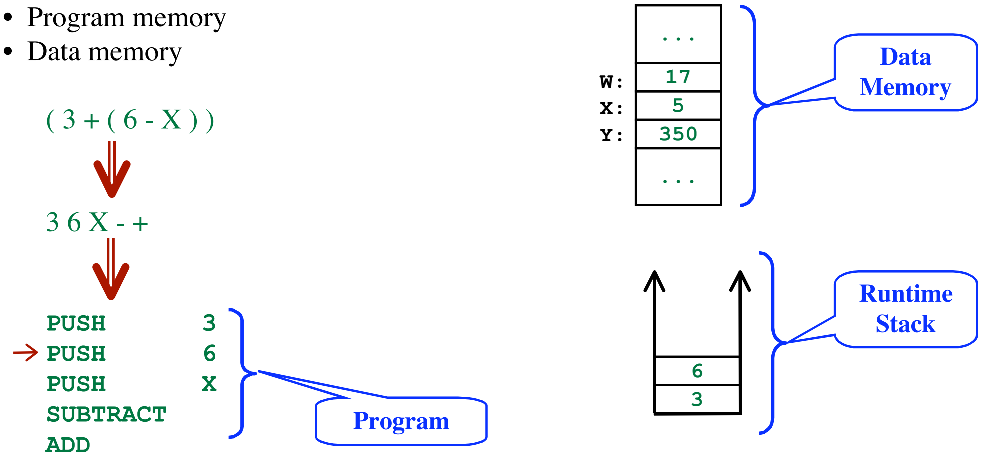
(3 + (6 - X))

3 6 X - +

```
→ PUSH       3
  PUSH       6
  PUSH       X
  SUBTRACT
  ADD
```

Abstract Stack Machine

- Limited / simple instruction set
- No registers
- Stack of data values
- Program memory
- Data memory



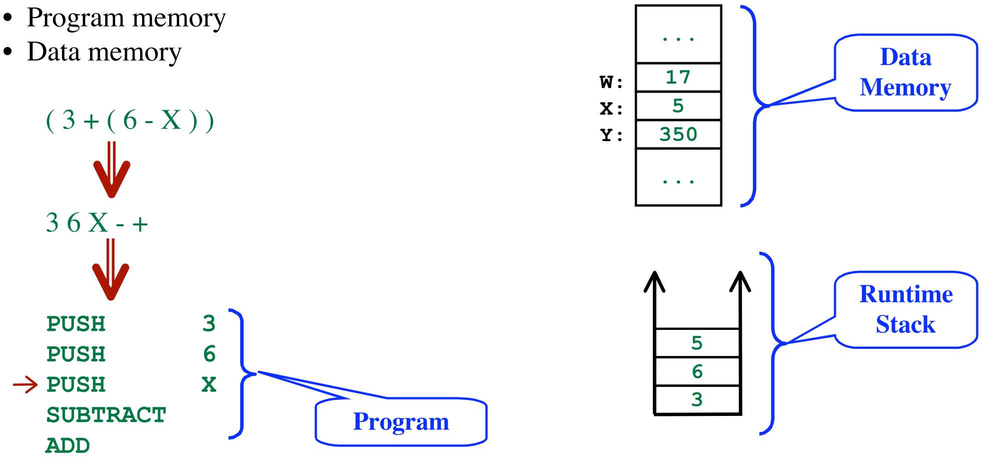
(3 + (6 - X))

3 6 X - +

```
  PUSH       3
→ PUSH       6
  PUSH       X
  SUBTRACT
  ADD
```

Abstract Stack Machine

- Limited / simple instruction set
- No registers
- Stack of data values
- Program memory
- Data memory



(3 + (6 - X))

⇓

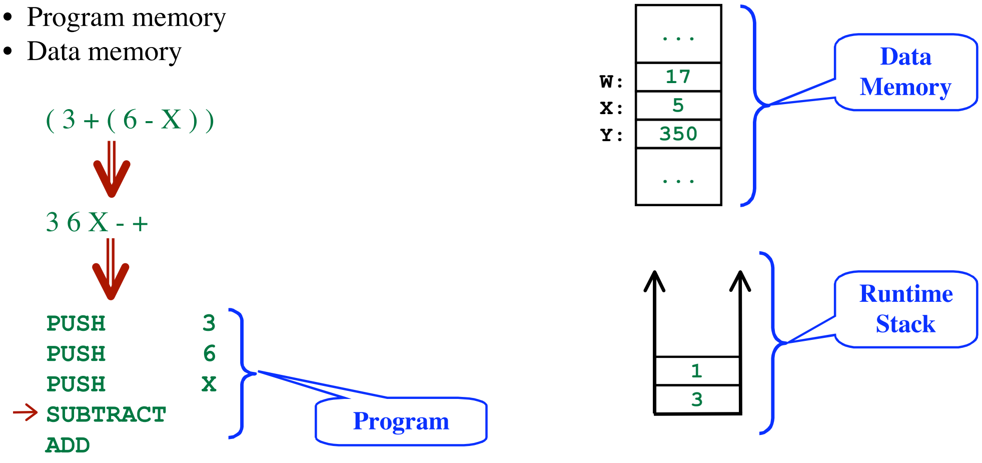
3 6 X - +

⇓

```
  PUSH      3
  PUSH      6
→ PUSH      X
  SUBTRACT
  ADD
```

Abstract Stack Machine

- Limited / simple instruction set
- No registers
- Stack of data values
- Program memory
- Data memory



(3 + (6 - X))

⇓

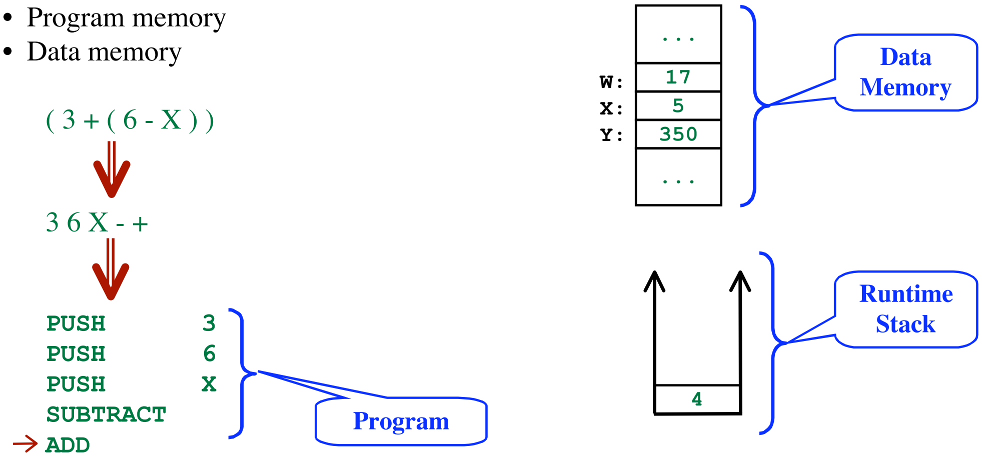
3 6 X - +

⇓

```
  PUSH      3
  PUSH      6
  PUSH      X
→ SUBTRACT
  ADD
```

Abstract Stack Machine

- Limited / simple instruction set
- No registers
- Stack of data values
- Program memory
- Data memory



(3 + (6 - X))

3 6 X - +

```
  PUSH      3
  PUSH      6
  PUSH      X
  SUBTRACT
→ ADD
```

L-Values versus R-Values

L-Value:

- Need the variable’s *location*.

R-Value:

- Need the variable’s *value*.

Example:

```
x = y * (z + 5);
```

L-Values versus R-Values

L-Value:

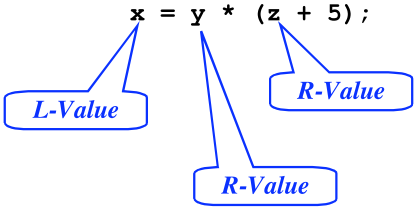
- Need the variable’s *location*.

R-Value:

- Need the variable’s *value*.

Example:

```
x = y * (z + 5);
```



L-Values versus R-Values

L-Value:

- Need the variable’s *location*.

R-Value:

- Need the variable’s *value*.

Example:

```
x = y * (z + 5);
```

Example:

```
p.computeTaxes (x);
```

L-Values versus R-Values

L-Value:

- Need the variable’s *location*.

R-Value:

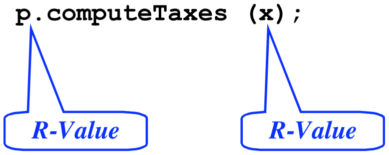
- Need the variable’s *value*.

Example:

```
x = y * (z + 5);
```

Example:

```
p.computeTaxes (x);
```



R-Value

R-Value

L-Values versus R-Values

L-Value:

- Need the variable’s *location*.

R-Value:

- Need the variable’s *value*.

Example:

```
x = y * (z + 5);
```

Example:

```
p.computeTaxes (x);
```

Example:

```
for i = 1 to 100 do ...
```

L-Values versus R-Values

L-Value:

- Need the variable's *location*.

R-Value:

- Need the variable's *value*.

Example:

```
x = y * (z + 5);
```

Example:

```
p.computeTaxes (x);
```

Example:

```
for i = 1 to 100 do ...
```

L-Value

L-Values versus R-Values

L-Value:

- Need the variable's *location*.

R-Value:

- Need the variable's *value*.

Example:

```
x = y * (z + 5);
```

Example:

```
p.computeTaxes (x);
```

Example:

```
for i = 1 to 100 do ...
```

Example:

```
read(x);
```

L-Values versus R-Values

L-Value:

- Need the variable’s *location*.

R-Value:

- Need the variable’s *value*.

Example:

```
x = y * (z + 5);
```

Example:

```
p.computeTaxes (x);
```

Example:

```
for i = 1 to 100 do ...
```

Example:

```
read(x);
```

L-Value

Stack Machine Instructions

Arithmetic Instructions

```
ADD
SUB
MULT
DIV
```

...etc...

Stack/Data Manipulation Instructions

```
PUSH     N
RVALUE   N
LVALUE   N
ASSIGN
POP
COPY
```

Flow of Control Instructions

```
GOTO     L
LABEL    L
GOFALSE  L
GOTRUE   L
HALT
```

Stack Machine Instructions

Arithmetic Instructions

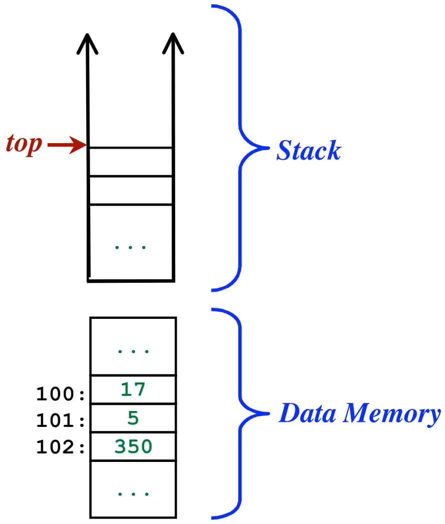
```
ADD
SUB
MULT
DIV
...etc...
```

Stack/Data Manipulation Instructions

```
PUSH     N
RVALUE   N
LVALUE   N
ASSIGN
POP
COPY
```

Flow of Control Instructions

```
GOTO     L
LABEL    L
GOFALSE  L
GOTRUE   L
HALT
```



Stack Machine Instructions

Arithmetic Instructions

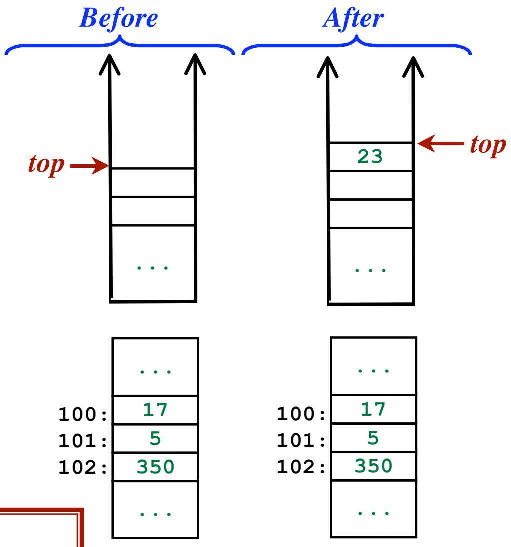
```
ADD
SUB
MULT
DIV
...etc...
```

Stack/Data Manipulation Instructions

```
PUSH     N
RVALUE   N
LVALUE   N
ASSIGN
POP
COPY
```

Flow of Control Instructions

```
GOTO     L
LABEL    L
GOFALSE  L
GOTRUE   L
HALT
```



```
...
PUSH    23
...
```

Stack Machine Instructions

Arithmetic Instructions

```
ADD
SUB
MULT
DIV
```

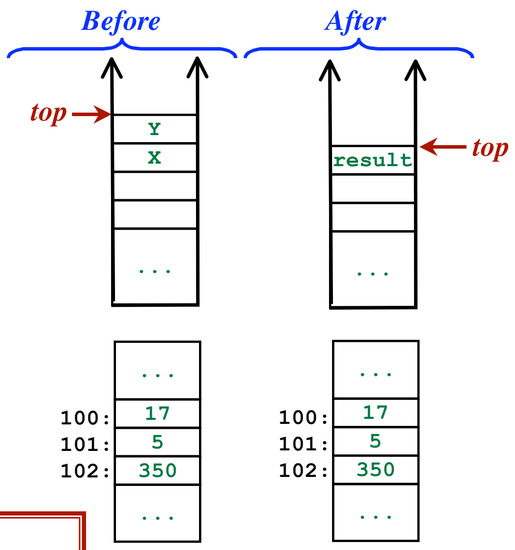
...etc...

Stack/Data Manipulation Instructions

```
PUSH     N
RVALUE   N
LVALUE   N
ASSIGN
POP
COPY
```

Flow of Control Instructions

```
GOTO     L
LABEL    L
GOFALSE  L
GOTRUE   L
HALT
```



```
...
PUSH  ...X...
PUSH  ...Y...
SUB
...
```

Stack Machine Instructions

Arithmetic Instructions

```
ADD
SUB
MULT
DIV
```

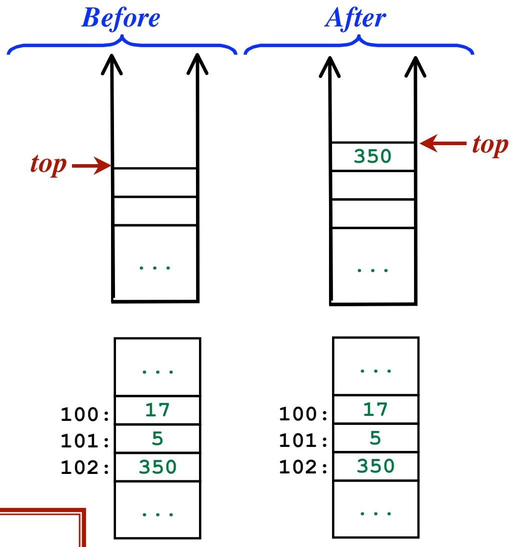
...etc...

Stack/Data Manipulation Instructions

```
PUSH     N
RVALUE   N
LVALUE   N
ASSIGN
POP
COPY
```

Flow of Control Instructions

```
GOTO     L
LABEL    L
GOFALSE  L
GOTRUE   L
HALT
```



```
...
RVALUE   102
...
```

Stack Machine Instructions

Arithmetic Instructions

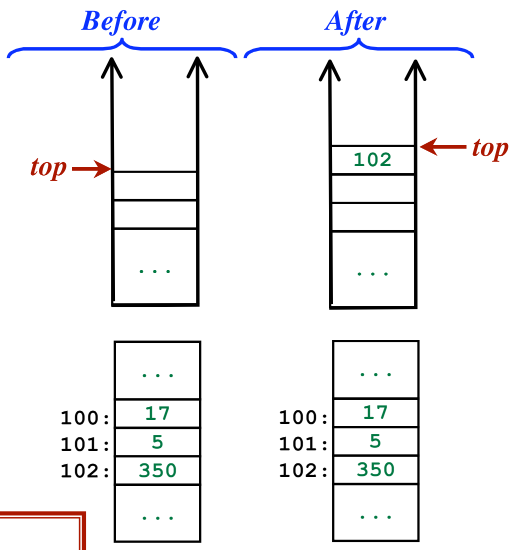
```
ADD
SUB
MULT
DIV
...etc...
```

Stack/Data Manipulation Instructions

```
PUSH     N
RVALUE   N
LVALUE   N
ASSIGN
POP
COPY
```

Flow of Control Instructions

```
GOTO     L
LABEL    L
GOFALSE  L
GOTRUE   L
HALT
```



```
...
LVALUE   102
...
```

Stack Machine Instructions

Arithmetic Instructions

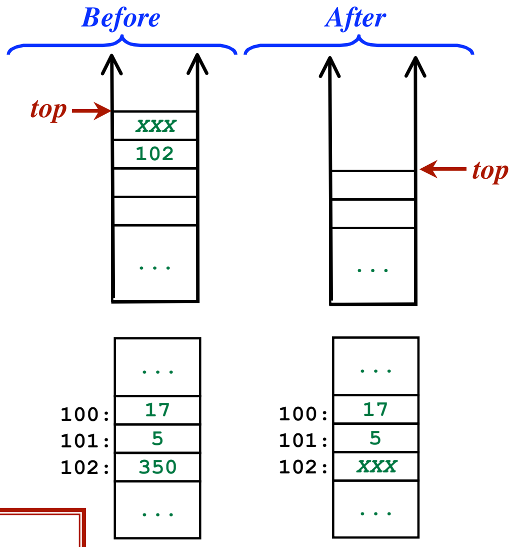
```
ADD
SUB
MULT
DIV
...etc...
```

Stack/Data Manipulation Instructions

```
PUSH     N
RVALUE   N
LVALUE   N
ASSIGN
POP
COPY
```

Flow of Control Instructions

```
GOTO     L
LABEL    L
GOFALSE  L
GOTRUE   L
HALT
```



```
...
LVALUE 102
PUSH   XXX
ASSIGN
...
```

Stack Machine Instructions

Arithmetic Instructions

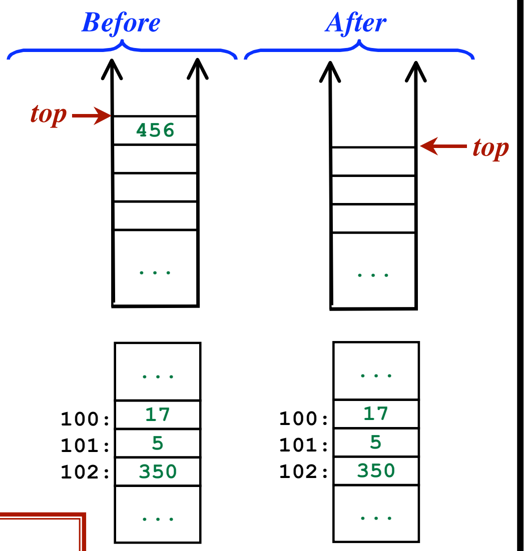
```
ADD
SUB
MULT
DIV
...etc...
```

Stack/Data Manipulation Instructions

```
PUSH     N
RVALUE   N
LVALUE   N
ASSIGN
POP
COPY
```

Flow of Control Instructions

```
GOTO     L
LABEL    L
GOFALSE  L
GOTRUE   L
HALT
```



```
...
POP
...
```

Stack Machine Instructions

Arithmetic Instructions

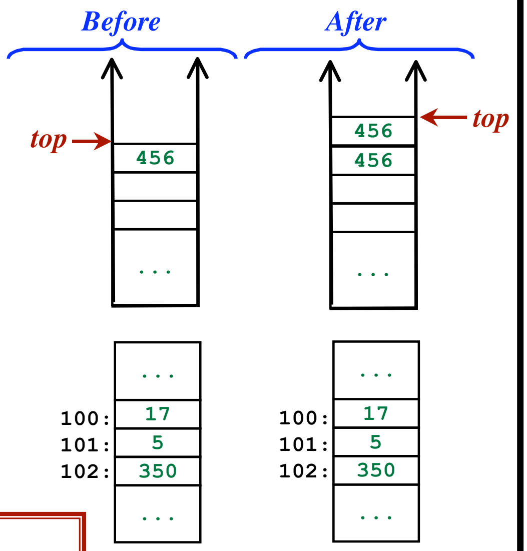
```
ADD
SUB
MULT
DIV
...etc...
```

Stack/Data Manipulation Instructions

```
PUSH     N
RVALUE   N
LVALUE   N
ASSIGN
POP
COPY
```

Flow of Control Instructions

```
GOTO     L
LABEL    L
GOFALSE  L
GOTRUE   L
HALT
```



```
...
COPY
...
```

Flow of Control

Option 1: **Absolute Addresses**

```
   ...
1004:   PUSH  123
1005:   SUB
1006:   GOTO  1004
    ...
```

Flow of Control

Option 1: **Absolute Addresses**

```
   ...
1004:   PUSH  123
1005:   SUB
1006:   GOTO  1004
    ...
```

Assembly Code:

```
              ...
MyLab:
              PUSH  123
              SUB
              GOTO  MyLab
              ...
```

Flow of Control

Option 1: **Absolute Addresses**

```
   ...
1004:  PUSH  123
1005:  SUB
1006:  GOTO  1004
   ...
```

Option 2: **Relative Addresses**

```
   ...
1004:  PUSH  123
1005:  SUB
1006:  GOTO  -2
   ...
```

Flow of Control

Option 1: **Absolute Addresses**

```
   ...
1004:  PUSH  123
1005:  SUB
1006:  GOTO  1004
   ...
```

Option 2: **Relative Addresses**

This code can be relocated!!!

```
   ...
1004:  PUSH  123
1005:  SUB
1006:  GOTO  -2
   ...
```

Flow of Control

Option 1: **Absolute Addresses**

```
   ...
1004:  PUSH  123
1005:  SUB
1006:  GOTO  1004
   ...
```

Option 2: **Relative Addresses**

```
   ...
1004:  PUSH  123
1005:  SUB
1006:  GOTO  -2
   ...
```

Option 3: **Symbolic Labels**

```
   ...
1003:  LABEL MyLab
1004:  PUSH  123
1005:  SUB
1006:  GOTO  MyLab
   ...
```

Flow of Control

Option 1: **Absolute Addresses**

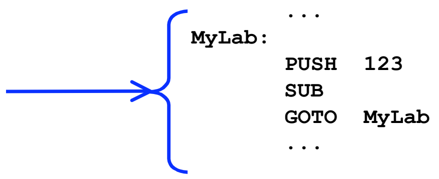
```
   ...
1004:  PUSH  123
1005:  SUB
1006:  GOTO  1004
   ...
```

Option 2: **Relative Addresses**

```
   ...
1004:  PUSH  123
1005:  SUB
1006:  GOTO  -2
   ...
```

Option 3: **Symbolic Labels**

```
   ...
1003:  LABEL MyLab
1004:  PUSH  123
1005:  SUB
1006:  GOTO  MyLab
   ...
```



```
              ...
MyLab:
              PUSH  123
              SUB
              GOTO  MyLab
              ...
```

Stack Machine Instructions

Arithmetic Instructions

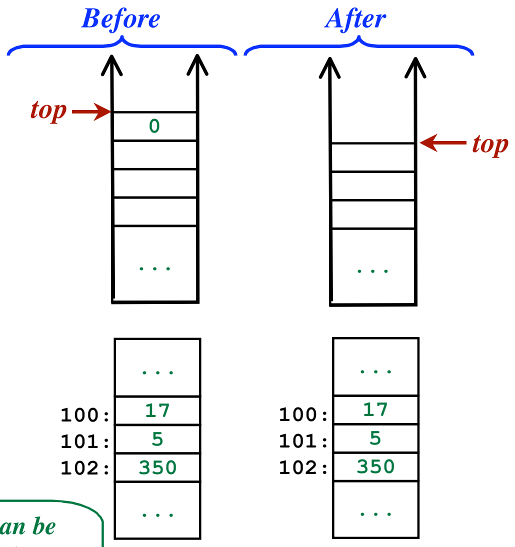
```
ADD
SUB
MULT
DIV
...etc...
```

Stack/Data Manipulation Instructions

```
PUSH     N
RVALUE   N
LVALUE   N
ASSIGN
POP
COPY
```

Flow of Control Instructions

```
GOTO     L
LABEL    L
GOFALSE  L
GOTRUE   L
HALT
```



Boolean values can be pushed onto stack:
0 = FALSE
Any other = TRUE

Example

Source Code:

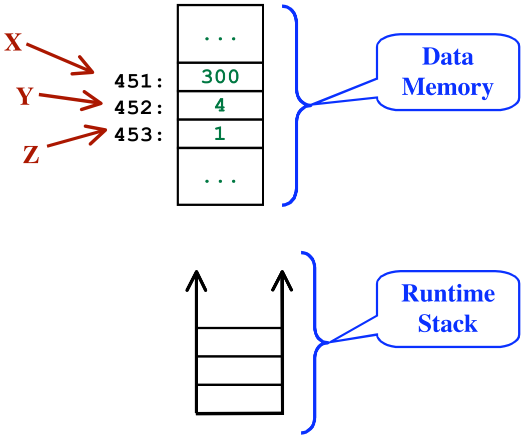
x = (y + 2) * (3 - z) ;

Postfix:

x y 2 + 3 z - * =

Instruction Memory:

```
   . . .
→ LVALUE    451
  RVALUE    452
  PUSH      2
  ADD
  PUSH      3
  RVALUE    453
  SUB
  MULT
  ASSIGN
  . . .
```



Example

Source Code:

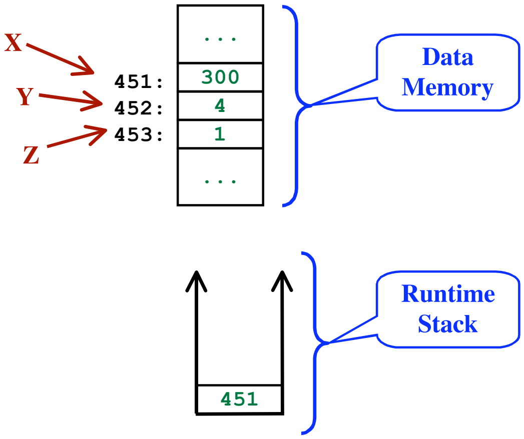
x = (y + 2) * (3 - z) ;

Postfix:

x y 2 + 3 z - * =

Instruction Memory:

```
   ...
→ LVALUE     451
  RVALUE     452
  PUSH       2
  ADD
  PUSH       3
  RVALUE     453
  SUB
  MULT
  ASSIGN
   ...
```



Example

Source Code:

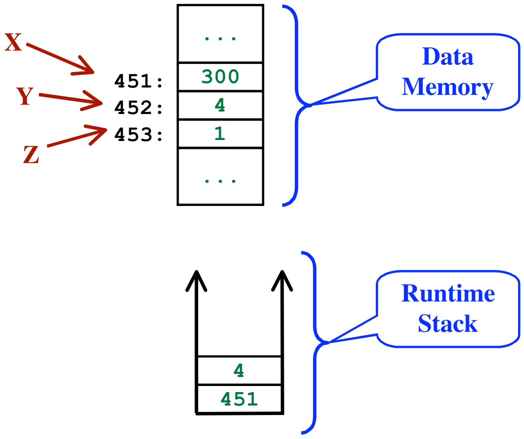
x = (y + 2) * (3 - z) ;

Postfix:

x y 2 + 3 z - * =

Instruction Memory:

```
   ...
  LVALUE     451
→ RVALUE     452
  PUSH       2
  ADD
  PUSH       3
  RVALUE     453
  SUB
  MULT
  ASSIGN
   ...
```



Example

Source Code:

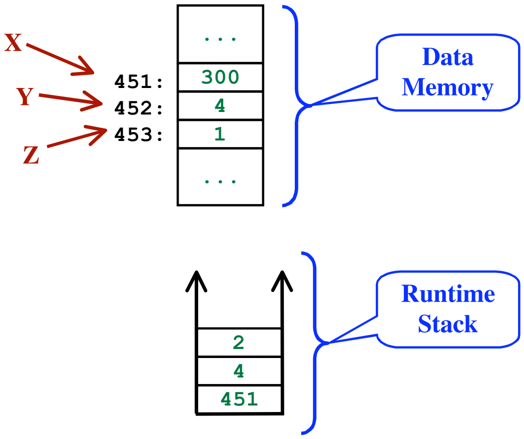
x = (y + 2) * (3 - z) ;

Postfix:

x y 2 + 3 z - * =

Instruction Memory:

```
   ...
   LVALUE     451
   RVALUE     452
→  PUSH       2
   ADD
   PUSH       3
   RVALUE     453
   SUB
   MULT
   ASSIGN
   ...
```



Example

Source Code:

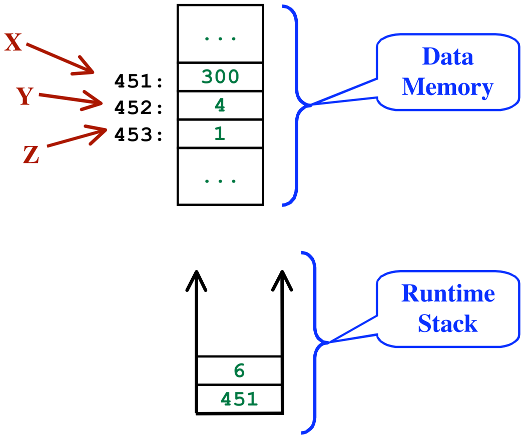
x = (y + 2) * (3 - z) ;

Postfix:

x y 2 + 3 z - * =

Instruction Memory:

```
   ...
   LVALUE     451
   RVALUE     452
   PUSH       2
   ADD
→  PUSH       3
   RVALUE     453
   SUB
   MULT
   ASSIGN
   ...
```



Example

Source Code:

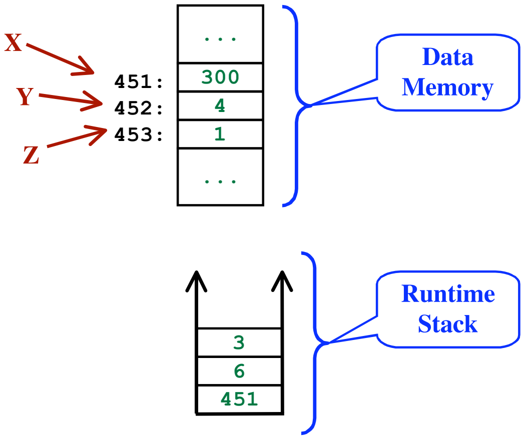
x = (y + 2) * (3 - z) ;

Postfix:

x y 2 + 3 z - * =

Instruction Memory:

```
     . . .
     LVALUE     451
     RVALUE     452
     PUSH       2
     ADD
  →  PUSH       3
     RVALUE     453
     SUB
     MULT
     ASSIGN
     . . .
```



Example

Source Code:

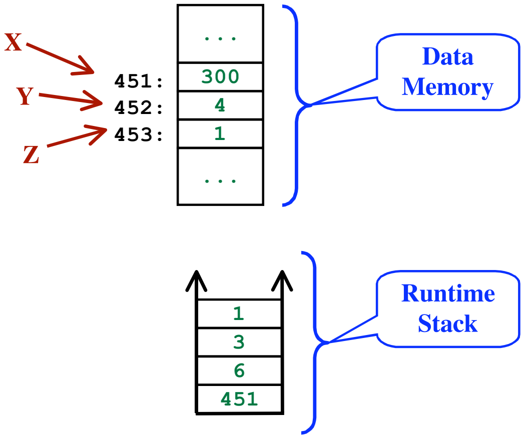
x = (y + 2) * (3 - z) ;

Postfix:

x y 2 + 3 z - * =

Instruction Memory:

```
     . . .
     LVALUE     451
     RVALUE     452
     PUSH       2
     ADD
     PUSH       3
     RVALUE     453
  →  SUB
     MULT
     ASSIGN
     . . .
```



Example

Source Code:

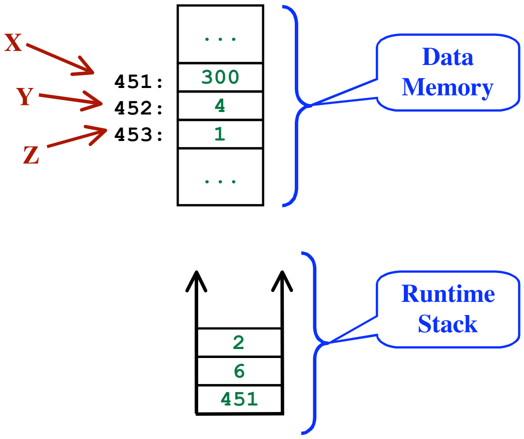
x = (y + 2) * (3 - z) ;

Postfix:

x y 2 + 3 z - * =

Instruction Memory:

```
    . . .
    LVALUE    451
    RVALUE    452
    PUSH      2
    ADD
    PUSH      3
    RVALUE    453
 →  SUB
    MULT
    ASSIGN
    . . .
```



Example

Source Code:

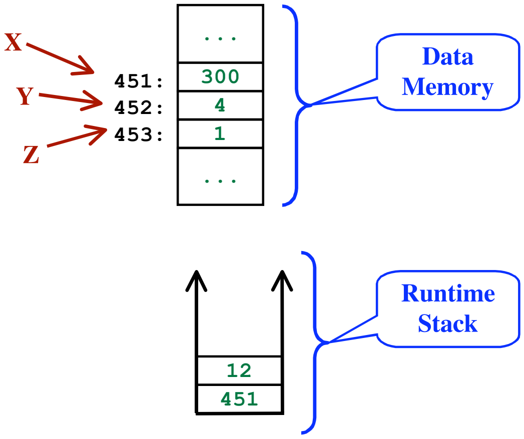
x = (y + 2) * (3 - z) ;

Postfix:

x y 2 + 3 z - * =

Instruction Memory:

```
    . . .
    LVALUE    451
    RVALUE    452
    PUSH      2
    ADD
    PUSH      3
    RVALUE    453
    SUB
    MULT
 →  ASSIGN
    . . .
```



Example

Source Code:

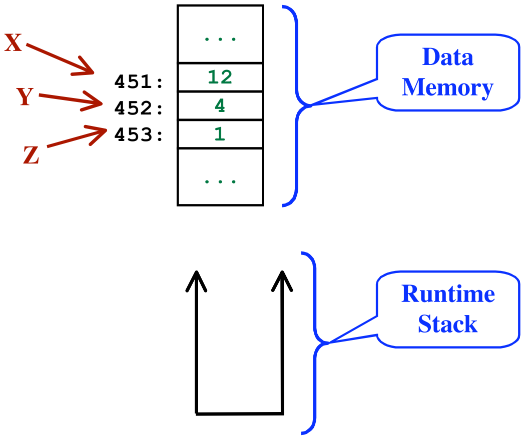
x = (y + 2) * (3 - z) ;

Postfix:

x y 2 + 3 z - * =

Instruction Memory:

```
   ...
   LVALUE    451
   RVALUE    452
   PUSH      2
   ADD
   PUSH      3
   RVALUE    453
   SUB
   MULT
→  ASSIGN
   ...
```



Producing Translations

Target: Code for Abstract Stack Machine

ParseExpr ()

Parses an expression

... and produces the code for it.

ParseStmt ()

Parses a statement

... and produces the code for it

... using ParseExpr and ParseStmt recursively.

Assignment Stmt:

ID = Expr ;

Translation:

```
LVALUE   ID
... Code for Expr...
ASSIGN
```

For example: **(X-3)*Y**

```
RVALUE   X
PUSH     3
SUB
RVALUE   Y
MULT
```

Translating a WHILE statement

Source:

```
...
while Expr do
  Stmts
 endWhile
...
```

Translation:

```
...
LABEL    Lab_43
   ... Code for Expr...
GOFALSE Lab_44
   ... Code for Stmts...
GOTO     Lab_43
LABEL    Lab_44
...
```

Generating Unique Labels

Function called: NewLabel

Returns a hitherto unused label.

Example:

```
Lab_17
Lab_18
Lab_19
...
```

Helper Function: EMIT()

Function: **Emit()**

Passed:

- An op-code
- Additional argument, if any

Writes one instruction to the output.

- To “stdout”
- To internal storage area

→ internal representation → target code → output file

Example of compiler code:

```
...
lab = NewLabel ();
Emit ("label", lab);
...
Emit ("goto", lab);
...
```

Translating Statements

```
Stmt   →  ID “=” Expr “;”
       →  while Expr do Stmts endWhile
       →  if Expr then Stmts else Stmts endIf
       →  ...
Expr   →  ...
```

Translating Statements

```
Stmt   →  ID “=” Expr “;”
       →  while Expr do Stmts endWhile
       →  if Expr then Stmts else Stmts endIf
       →  ...
Expr   →  ...
```

Source:

$W = \underbrace{X + Y + Z}_{Expr};$

Code:

```
LVALUE   W
RVALUE   X
RVALUE   Y
RVALUE   Z
ADD
ADD
ASSIGN
```

Emitted by ParseExpr

Translating Statements

```
Stmt   → ID "=" Expr ";"
       → while Expr do Stmts endWhile
       → if Expr then Stmts else Stmts endIf
       → ...
Expr   → ...
```

Translation Scheme for ASSIGN-STMT:

```
Stmt   → ID
          { Emit ("LVALUE", ID.svalue) }
          "="
          Expr
          { Emit ("ASSIGN") }
          ";"
```

Source:

$W = \underbrace{X + Y + Z}_{Expr};$

Code:

```
LVALUE   W
RVALUE   X  ⎫
RVALUE   Y  ⎪ Emitted
RVALUE   Z  ⎬   by
ADD         ⎪     ParseExpr
ADD         ⎭
ASSIGN
```

Translating Statements

```
Stmt   → ID "=" Expr ";"
       → while Expr do Stmts endWhile
       → if Expr then Stmts else Stmts endIf
       → ...
Expr   → ...
```

Source:

```
while A-B do
   X=Y;
endWhile
```

Code:

```
LABEL    Lab_4
RVALUE   A  ⎫ Emitted
RVALUE   B  ⎬   by
SUB         ⎭     ParseExpr
GOFALSE  Lab_5
LVALUE   X  ⎫ Emitted
RVALUE   Y  ⎬   by
ASSIGN      ⎭     ParseStmts
GOTO     Lab_4
LABEL    Lab_5
```

Translating Statements

```
Stmt   → ID "=" Expr ";"
       → while Expr do Stmts endWhile
       → if Expr then Stmts else Stmts endIf
       → ...
Expr   → ...
```

Translation Scheme for WHILE-STMT:

```
Stmt → while
       { topLabel = NewLabel()
         bottomLabel = NewLabel()
         Emit ("LABEL", topLabel) }
       Expr
       { Emit ("GOFALSE", bottomLabel) }
       do Stmts endWhile
       { Emit ("GOTO", topLabel)
         Emit ("LABEL", bottomLabel) }
```

Source:

```
while A-B do
   X=Y;
endWhile
```

Code:

```
LABEL    Lab_4
RVALUE   A  } Emitted
RVALUE   B  }   by
SUB         }     ParseExpr
GOFALSE  Lab_5
LVALUE   X  } Emitted
RVALUE   Y  }   by
ASSIGN      }     ParseStmts
GOTO     Lab_4
LABEL    Lab_5
```

```
function ParseStmt ()
  var topLabel, bottomLabel: String
  if nextToken == ID then
    Emit ("LVALUE", token.svalue)
    MustHave (ID)
    MustHave ("=")
    ParseExpr ()
    Emit ("ASSIGN")
    MustHave (";")
  elseIf nextToken == WHILE then
    MustHave (WHILE)
    topLabel = NewLabel ()
    bottomLabel = NewLabel ()
    Emit ("LABEL", topLabel)
    ParseExpr ()
    Emit ("GOFALSE", bottomLabel)
    MustHave (DO)
    ParseStmts ()
    MustHave (ENDWHILE)
    Emit ("GOTO", topLabel)
    Emit ("LABEL", bottomLabel)
  elseIf
    ...
  endIf
endFunction
```

```
Stmt → ID
       { Emit("LVALUE",ID.svalue) }
       "="
       Expr
       { Emit("ASSIGN") }
       ";"
Stmt → while
       { topLabel = NewLabel()
         bottomLabel = NewLabel()
         Emit("LABEL",topLabel) }
       Expr
       { Emit("GOFALSE",bottomLabel)}
       do Stmts endWhile
       { Emit("GOTO",topLabel)
         Emit("LABEL",bottomLabel) }
```

Short-Circuit Operators

$\texttt{if}\ \underbrace{\underbrace{\texttt{(i <= max)}}\ \texttt{and}\ \underbrace{\texttt{(a[i] == -1)}}}\ \texttt{then}\ \ldots$

Do we need to evaluate the second expression?

b = Expr_1 **and** Expr_2

⇓

b = **if** Expr_1 **then** Expr_2 **else** FALSE **endIf**

Short-Circuit Operators

$\texttt{if}\ \underbrace{\underbrace{\texttt{(i <= max)}}\ \texttt{and}\ \underbrace{\texttt{(a[i] == -1)}}}\ \texttt{then}\ \ldots$

Do we need to evaluate the second expression?

b = Expr_1 and Expr_2

⇓

b = if Expr_1 then Expr_2 else FALSE endIf

Translation:

```
. . .
   ... Code for Expr1...
COPY
GOFALSE Lab_43
POP
   ... Code for Expr2...
LABEL   Lab_43
. . .
```

Short-Circuit Operators

```
if (i <= max) and (a[i] == -1) then ...
```

Do we need to evaluate the second expression?

b = Expr$_1$ **and** Expr$_2$

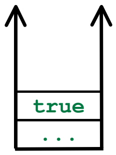
⇓

b = **if** Expr$_1$ **then** Expr$_2$ **else** FALSE **endIf**

Translation:

```
    . . .
      ... Code for Expr1...
→   COPY
    GOFALSE  Lab_43
    POP
      ... Code for Expr2...
    LABEL    Lab_43
    . . .
```

Case 1: Expr-1 is true:



Short-Circuit Operators

```
if (i <= max) and (a[i] == -1) then ...
```

Do we need to evaluate the second expression?

b = Expr$_1$ **and** Expr$_2$

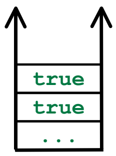
⇓

b = **if** Expr$_1$ **then** Expr$_2$ **else** FALSE **endIf**

Translation:

```
    . . .
      ... Code for Expr1...
    COPY
→   GOFALSE  Lab_43
    POP
      ... Code for Expr2...
    LABEL    Lab_43
    . . .
```

Case 1: Expr-1 is true:



Short-Circuit Operators

```
if (i <= max) and (a[i] == -1) then ...
```

Do we need to evaluate the second expression?

b = Expr$_1$ **and** Expr$_2$

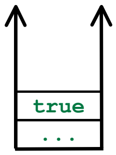
⇓

b = **if** Expr$_1$ **then** Expr$_2$ **else** FALSE **endIf**

Translation:

```
    ...
      ... Code for Expr1...
    COPY
→   GOFALSE Lab_43
    POP
      ... Code for Expr2...
    LABEL   Lab_43
    ...
```

Case 1: Expr-1 is true:



Short-Circuit Operators

```
if (i <= max) and (a[i] == -1) then ...
```

Do we need to evaluate the second expression?

b = Expr$_1$ **and** Expr$_2$

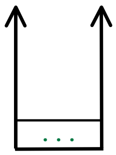
⇓

b = **if** Expr$_1$ **then** Expr$_2$ **else** FALSE **endIf**

Translation:

```
    ...
      ... Code for Expr1...
    COPY
    GOFALSE Lab_43
→   POP
      ... Code for Expr2...
    LABEL   Lab_43
    ...
```

Case 1: Expr-1 is true:



Short-Circuit Operators

$\underline{\texttt{if}}\ \underbrace{\texttt{(}\underbrace{\texttt{i <= max}}\texttt{)}\ \underline{\texttt{and}}\ \texttt{(}\underbrace{\texttt{a[i] == -1}}\texttt{)}}\ \underline{\texttt{then}}\ \texttt{...}$

Do we need to evaluate the second expression?

b = Expr_1 **and** Expr_2

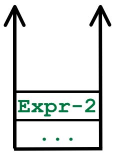
$\Downarrow$

b = **if** Expr_1 **then** Expr_2 **else** FALSE **endIf**

Translation:

```
   . . .
     ... Code for Expr1...
   COPY
   GOFALSE Lab_43
   POP
     ... Code for Expr2...
→LABEL    Lab_43
   . . .
```

Case 1: Expr-1 is true:



Short-Circuit Operators

$\underline{\texttt{if}}\ \underbrace{\texttt{(}\underbrace{\texttt{i <= max}}\texttt{)}\ \underline{\texttt{and}}\ \texttt{(}\underbrace{\texttt{a[i] == -1}}\texttt{)}}\ \underline{\texttt{then}}\ \texttt{...}$

Do we need to evaluate the second expression?

b = Expr_1 **and** Expr_2

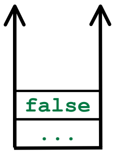
$\Downarrow$

b = **if** Expr_1 **then** Expr_2 **else** FALSE **endIf**

Translation:

```
   . . .
     ... Code for Expr1...
→COPY
   GOFALSE Lab_43
   POP
     ... Code for Expr2...
   LABEL    Lab_43
   . . .
```

Case 2: Expr-1 is false:



Short-Circuit Operators

```
if (i <= max) and (a[i] == -1) then ...
```

Do we need to evaluate the second expression?

b = Expr$_1$ **and** Expr$_2$

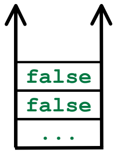
⇓

b = **if** Expr$_1$ **then** Expr$_2$ **else** FALSE **endIf**

Translation:

```
    ...
      ... Code for Expr1...
    COPY
→   GOFALSE Lab_43
    POP
      ... Code for Expr2...
    LABEL   Lab_43
    ...
```

Case 2: Expr-1 is false:



Short-Circuit Operators

```
if (i <= max) and (a[i] == -1) then ...
```

Do we need to evaluate the second expression?

b = Expr$_1$ **and** Expr$_2$

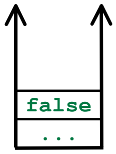
⇓

b = **if** Expr$_1$ **then** Expr$_2$ **else** FALSE **endIf**

Translation:

```
    ...
      ... Code for Expr1...
    COPY
    GOFALSE Lab_43
    POP
      ... Code for Expr2...
→   LABEL   Lab_43
    ...
```

Case 2: Expr-1 is false:



Short-Circuit Operators

And

b = Expr_1 **and** Expr_2

Or

b = Expr_1 **or** Expr_2

Conditional (ternary) operator

b = Expr_1 **?** Expr_2 **:** Expr_3

Means: b = (if Expr_1 then Expr_2 else Expr_3 endIf)

Same as:

```
if $\text{Expr}_1$ then
    b = $\text{Expr}_2$
else
    b = $\text{Expr}_3$
endIf
```