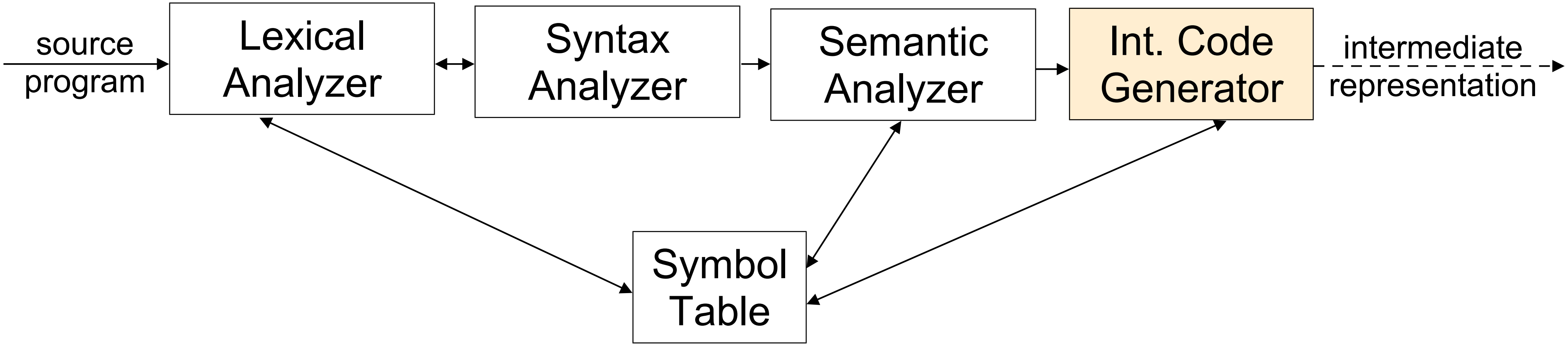
Intermediate Code Generation

- Front-end → intermediate code generation



- Advantages of intermediate representation (independent of target):

  1. Porting → change of back-end only
  2. First optimization independent of target

Intermediate Representation

- **Intermediate representation** ≡ data structure representing source program during translation (exmp: abstract tree + symbol table)

- Abstract tree
  - adequate representation of source program
  - inadequate for target program (exmp :loops → goto)

- **Intermediate code** → intermediate representation closest to target code

- Various forms of intermediate code: in general → linearization of abstract tree (≈ oper. semantics)

- Various abstraction levels
  - high → close to abstract tree
  - low → close to target machine

- Popular representations
  - **Three-address code**
  - **P-code**

  ≠ forms

Three-Address Code

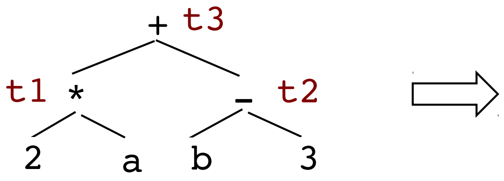
- Typical statement: designed to represent evaluation of simple arithmetic expressions

```
x = y op z
```

  - `x`, `y`, `z` = memory addresses (certainly `x`, while `y` and `z` may be constants)
  - *op* = either arithmetic operator (typically) or other

- Example:

```
2 * a + (b – 3)
```



```
t1 = 2 * a
t2 = b – 3
t3 = t1 + t2
```

- Notes:

  - Compiler → generation of names for temporaries (`t1`, `t2`, `t3`) → isomorphic to internal nodes
  - Three-address code = linearization (left to right) of abstract tree
  - Unless ∃ constraints on evaluation order, possible a different order: (≠ meaning of names)

```
t1 = b – 3
t2 = 2 * a
t3 = t2 + t1
```

  - In general: need for other forms of statements (∄ standard), exmp:

```
t2 = -t1
```

Three-Address Code (ii)

- Extended example: computation of factorial `x!`

```
read x;        /* input of integer */
if x > 0 then
    fact := 1;
    repeat
        fact := fact * x;
        x := x – 1
    until x = 0;
    write fact
endif
```

⇒

```
read x
t1 = x>0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x – 1
x = t3
t4 = x==0
if_false t4 goto L2
write fact
label L1
halt
```

- Notes:
  - **`read`** / **`write`**: directly translated into statements with one address
  - `if_false` (two addresses): used to translate **`if`** / **`repeat`**
  - `label` (one address): may be necessary in some implementations of three-address code
  - `halt` (zero addresses!): program termination
  - Assignments within source → mapped to copy statements

```
fact := fact * x
```

⇒

```
t2 = fact * x
fact = t2
```

(even if sufficient unique statement → optimization)

Three-Address Code (iii)

- Representation: typically
  - ∀ statement → record of fields
  - complete code
    - array
    - linked list

    of records

- Record (typically):

op	addr1	addr2	addr3

- Possible `null` value for some addresses

```
read x
t1 = x>0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x – 1
x = t3
t4 = x==0
if_false t4 goto L2
write fact
label L1
halt
```

⇨

```
(rd, x, -, -)
(gt, x, 0, t1)
(if_f, t1, L1, -)
(asn, 1, fact, -)
(lab, L2, -, -)
(mul, fact, x, t2)
(asn, t2, fact, -)
(sub, x, 1 , t3)
(asn, t3, x, -)
(eq, x, 0, t4)
(if_f, t4, L2, -)
(wri, fact, -, -)
(lab, L1, -, -)
(halt, -, -, -)
```

P-code

- Historically: designed ≈ 1980 to be the language of a **P-machine** of which an interpreter was implemented on different real platforms

- Requirement: portability of Pascal compilers → P-code designed to be directly executed

- We: simplified version of P-machine
  - code memory
  - data memory (for variables with names)
  - stack (for temporaries)
  - registers to
    - manage the stack (stack pointer)
    - support execution (program counter)

Example 1:

`2 * a + (b – 3)` ⇒

```
ldc 2        ;  load constant 2
lod a        ;  load value of var a
mpi          ;  integer multiplication
lod b        ;  load value of var b
ldc 3        ;  load constant 3
sbi          ;  integer substraction
adi          ;  integer addition
```

P-code (ii)

Example 2:

```
x := y + 1
```

⇒

```
lda x      ; load address of x
lod y      ; load value of y
ldc 1      ; load constant 1
adi        ; add
sto        ; store top to address below top and pop both
```

- Note: different semantics for
  - `lda` → load address → lvalue
  - `lod` → load value → rvalue

P-code (iii)

Example 3:

```
read x;      /* input of integer */
if x > 0 then
    fact := 1;
    repeat
        fact := fact * x;
        x := x – 1
    until x = 0;
    write fact
endif
```

⇨

```
lda x     ; load address of x
rdi       ; read integer, store to address on top and pop
lod x     ; load value of x
ldc 0     ; load constant 0
grt       ; compare top two values, pop them, push Boolean result
fjp L1    ; pop Boolean value, jump to L1 if false
lda fact  ; load address of fact
ldc 1     ; load constant 1
sto       ; pop two values, storing first to address given by second
lab L2    ; definition of label L2
lda fact  ; load address of fact
lod fact  ; load value of fact
lod x     ; load value of x
mpi       ; multiply
sto       ; store top to address of second and pop both
lda x     ; load address of x
lod x     ; load value of x
ldc 1     ; load constant 1
sbi       ; substract
sto       ; store (as before)
lod x     ; load value of x
ldc 0     ; load constant 0
equ       ; test for equality
fjp L2    ; pop Boolean value, jump to L2 if false
lod fact  ; load value of fact
wri       ; write top and, then, pop
lab L1    ; definition of label L1
stp       ; stop execution
```

Comparison between P-code and Three-Address Code

- Pros:
  - P-code closer to target code than Three-address code
  - P-code statements require less addresses (our examples: one address at most)
    (operands implicitly on the stack)

- Cons:
  - P-code less compact than Three-address code (number of statements)
  - P-code not self-contained: statements implicitly operate on a stack
    (implicit locations of stack = surrogate of addresses)

- Advantage in using a stack: contains all values necessary in any point of the code
  → unnecessary for compiler assigning names to values
  Also: automatic removal of temporaries

Intermediate Code as Synthesized Attribute

- Example: Subset of C expressions (assignment as expression)

expr → **id** = *expr* | *term*
term → *term* + *factor* | *factor*
factor → **(** *expr* **)** | **num** | **id**

Assumption: `id` and `num` : with lexical attribute `lexval` = string of characters

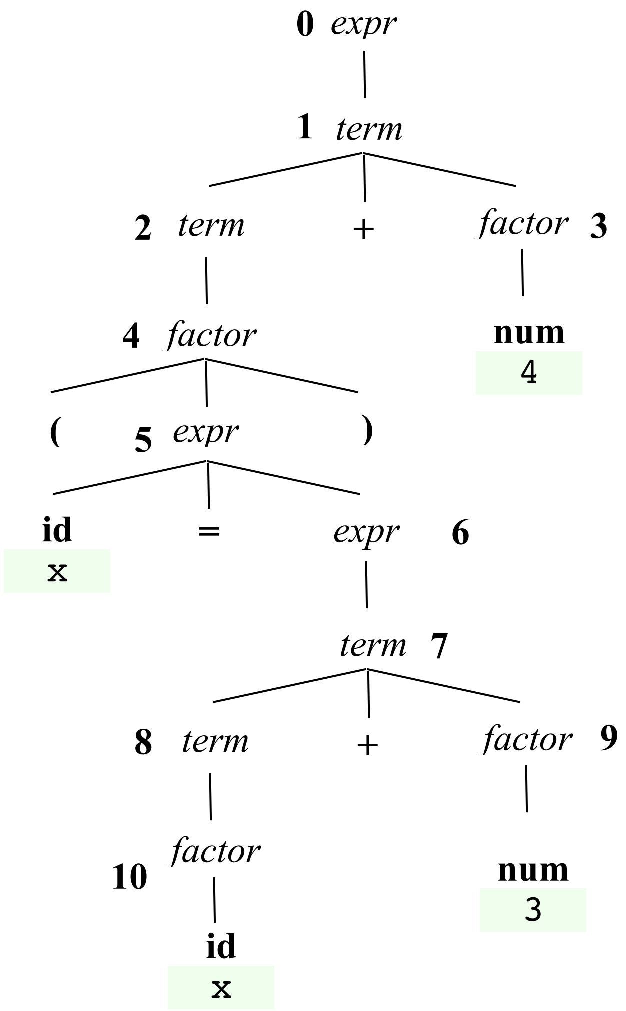
- P-code (A = { pcode }):
  - AG simpler (unnecessary generating names for temporaries) → A = { pcode }
  - Necessary a "*nondestructive store*" to support assignments as expressions → `stn`
  - Semantics of `stn`: $\begin{cases}\text{Stores value at address below it (as } \texttt{sto}\text{)} \\ \text{Leaves value on top of stack} \\ \text{Discards address}\end{cases}$

Production	*Semantic rules*
$expr_1$ → **id** = $expr_2$	$expr_1$.pcode = "`lda`" || **id**.lexval ++ $expr_2$.pcode ++ "`stn`"
expr → *term*	*expr*. pcode = *term*. pcode
$term_1$ → $term_2$ + *factor*	$term_1$.pcode = $term_2$.pcode ++ *factor*.pcode ++ "`adi`"
term → *factor*	*term*.pcode = *factor*.pcode
factor → **(** *expr* **)**	*factor*.pcode = *expr*.pcode
factor → **num**	*factor*.pcode = "`ldc`" || **num**. lexval
factor → **id**	*factor*.pcode = "`lod`" || **id**. lexval

|| = concatenation with space
++ = concatenation with newline

Intermediate Code as Synthesized Attribute (ii)

- Example (P-code): `(x = x + 3) + 4`



⇨

Nodes	*P-code*
10, 8	`lod x`
9	`ldc 3`

7, 6	`lod x`
	`ldc 3`
	`adi`

5, 4, 2	`lda x`
	`lod x`
	`ldc 3`
	`adi`
	`stn`

3	`ldc 4`

1, 0	**`lda x`**
	`lod x`
	`ldc 3`
	`adi`
	`stn`
	`ldc 4`
	`adi`

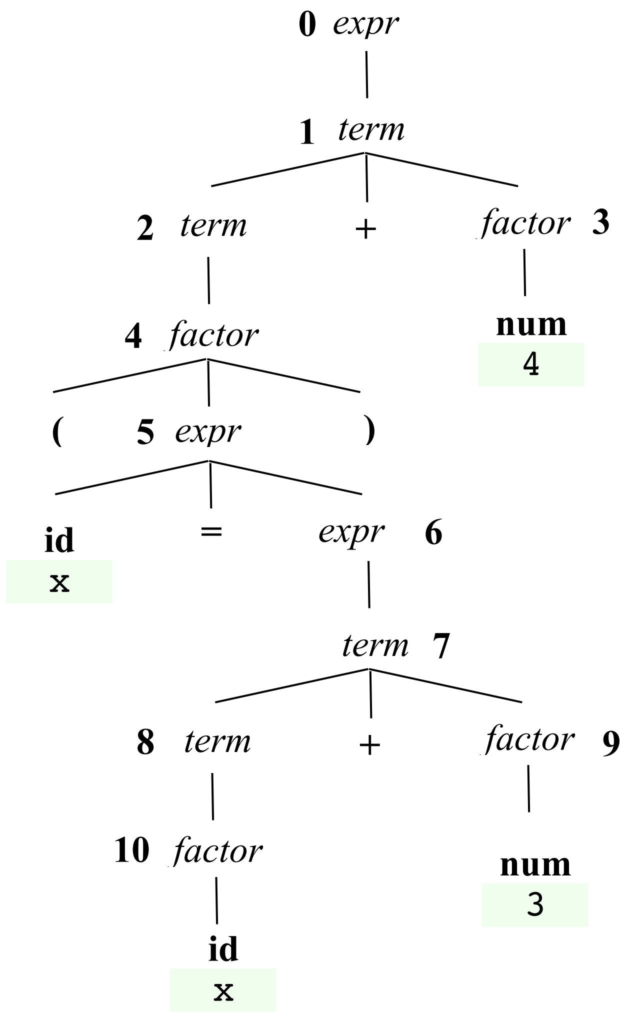
Intermediate Code as Synthesized Attribute (iii)

- <u>Example</u> (Three-address code): necessary assigning each expression with a name →
  A = { name, qcode }

Production	*Semantic rules*
$expr_1 \rightarrow$ **id** = $expr_2$	$expr_1$.name = $expr_2$.name $expr_1$.qcode = $expr_2$. qcode ++ **id**.lexval || "=" || $expr_2$.name
$expr \rightarrow term$	$expr$. name = $term$. name $expr$. qcode = $term$. qcode
$term_1 \rightarrow term_2 + factor$	$term_1$.name = `newtemp( )` $term_1$.qcode = $term_2$.qcode ++ $factor$.qcode ++ $term_1$.name || "=" || $term_2$.name || "+" || $factor$.name
$term \rightarrow factor$	$term$.name = $factor$.name $term$.qcode = $factor$.qcode
$factor \rightarrow$ **(** $expr$ **)**	$factor$.name = $expr$.name $factor$.qcode = $expr$.qcode
$factor \rightarrow$ **num**	$factor$.name = **num**. lexval $factor$.qcode = " "
$factor \rightarrow$ **id**	$factor$.name = **id**. lexval $factor$.qcode = " "

Intermediate Code as Synthesized Attribute (iv)

`(x = x + 3) + 4`



Nodes	*qcode*	*name*
10,8		`x`
9		`3`
3		`4`
7, 6	`t1 = x + 3`	`t1`

5, 4, 2	`t1 = x + 3`
	`x = t1`
		`t1`

1, 0	**`t1 = x + 3`**
	`x = t1`
	`t2 = t1 + 4`
		`t2`

Intermediate Code as Synthesized Attribute (v)

- <u>Notes</u> (on code generation as computation of a synthesized attribute)

  - Clearly shows relations between ⟨ code fragments / syntax sub-trees

  - Impractical, because:

    1. String concatenation ⟨ many copy operations / waste of memory

    2. Better generating small fragments of code and writing on file (or on data structures)

       ⇩

       semantic actions which <u>do not</u> adhere to attribute synthesis in post-order

    3. In general: code generation heavily depends on inherited attributes → complication in AG

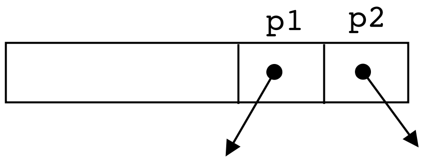
⇩

Need for more direct (practical) code-generation techniques <u>not</u> based on AG

Practical Code Generation

- Hp: Syntax tree with nodes having at most two children (easily generalizable)

```
procedure genCode(n: Node);
begin
   if n ≠ nil then
      generate code to prepare for code of left child of n;
      genCode(n.p1);
      generate code to prepare for code of right child of n;
      genCode(n.p2);
      generate code to implement the action of n
   endif
end.
```

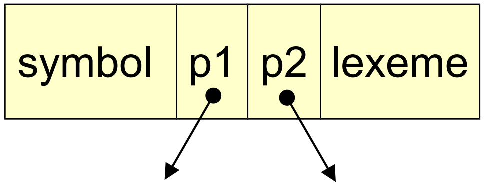


Practical Code Generation (ii)

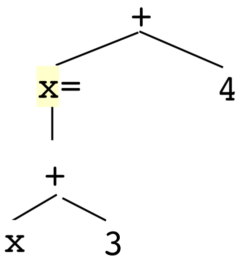
$expr \rightarrow \mathbf{id} = expr \mid term$
$term \rightarrow term\ +\ factor \mid factor$
$factor \rightarrow \mathbf{(}\ expr\ \mathbf{)} \mid \mathbf{num} \mid \mathbf{id}$

(for abstract tree)

```
typedef enum {PLUS, ASSIGN, NUM, ID} Symbol;

typedef struct t_node
{
    Symbol symbol;
    struct t_node *p1, *p2;
    char *lexeme;               /* for id, num */
} Node;

typedef Node *Pnode;
```



`(x = x + 3) + 4` ⇒

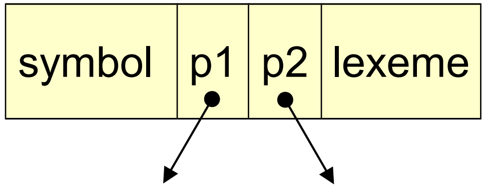


Note: Assignment identifier stored in the relevant node

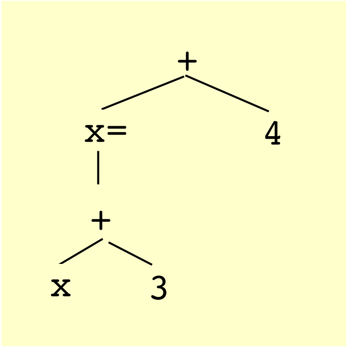
- If LHS complex → need for another (first) child

Practical Code Generation (iii)

```
void genCode(Pnode p)
{
  char code[MAXCODE]; /* max length of one line of P-code */

  switch(p->symbol)
  {
  case PLUS: genCode(p->p1);
             gencode(p->p2);
             emit("adi");
             break;
  case ASSIGN: sprintf(code, "lda %s", p->lexeme);
               emit(code);
               genCode(p->p1);
               emit("stn");
               break;
  case NUM: sprintf(code, "ldc %s", p->lexeme);
            emit(code);
            break;
  case ID: sprintf(code, "lod %s", p->lexeme);
           emit(code);
           break;
  }
}
```



symbol	p1	p2	lexeme



Code Generation for References to Data Structures

- Previously: code generation for expr/assignments, where values =
  - constants (`LDC`)
  - simple var (`LOD`, `LDA`)

- Generation of target code → var names replaced by addresses
  - registers
  - absolute addresses
  - relative addresses within AR

- In general: need to compute addresses by means of intermediate code
  - array indexing
  - record fields
  - pointers

⇩

<u>need to extend the intermediate notation to express the computation of addresses</u>

Extending P-code for Address Computation

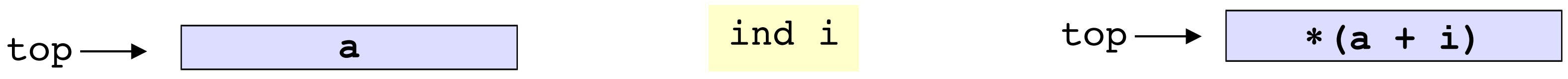
- Introduction of new statements to express ≠ address modes

  - **ind** *<offset>* (indirect load)

    top(stack) = *<address>*

    <offset> = integer

```
addr = <address> + <offset>
pop()
push(*addr)
```



top → a

ind i

top → *(a + i)

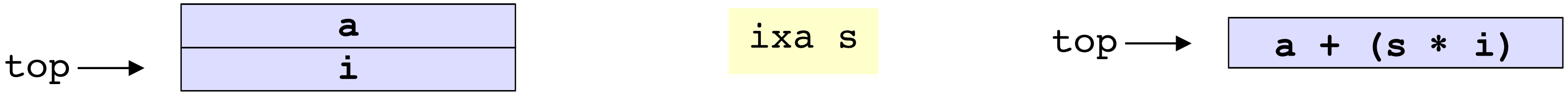
  - **ixa** *<scale>* (indexed address)

    top(stack) = *<offset>*

    subtop(stack) = *<base-addr>*

    <scale> = scale factor

```
addr = <base-addr> + (<scale> * <offset>)
pop(), pop()
push(addr)
```



a
top → i

ixa s

top → a + (s * i)

- Example: Write constant 2 at address of x plus 10 byte ⇨

```
lda x
ldc 10
ixa 1
ldc 2
sto
```

Code Generation for Array Manipulation

- Array elements stored sequentially
- Element address: computed by means of
  - base address of array
  - offset = linear function of index value

```
int a[SIZE], i, j;
…
a[i+1] = a[j*2] + 3;
```

address   integer   anyway → need to first compute address

- Address computation:

  1. Index "normalization" (when index does not start with `0`) → normalized index
  2. Multiplication of normalized index by scale factor = sizeof(elem) → scaled index
  3. Address = base + scaled index (at point 2.)

Code Generation for Array Manipulation (ii)

- Example (C): `a[i+1]` ⇨ $\underset{\text{base}}{\texttt{a}} + \overbrace{(\underset{\text{index}}{\texttt{(i+1)}} \; \texttt{*} \; \underset{\text{scale factor}}{\texttt{sizeof(}\textbf{int}\texttt{)}})}^{\text{scaled index}}$

- General formula: `a[t]` ⇨ `base(a) + ((t – lower_bound(a)) * elem_size(a))`

- Assumption for the independence from the target machine:

  1. Address of array variable ≡ base address `lda a` → push(base address of `a`)

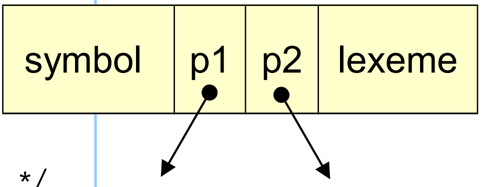
  2. `elem_size(a)` ≡ size of array element
     (statically known → replaced with a constant by compiler back-end)

Procedure of Code Gen. for References and Arrays

expr → *ixpr* = *expr* | *term*
term → *term* + *factor* | *factor*
factor → **(** *expr* **)** | **num** | *ixpr*
ixpr → **id** | **id [** *expr* **]**

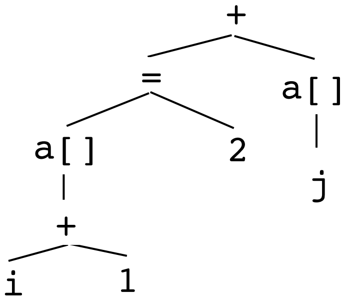
- Notes:
  - LHS of assignment = either identifier or indexing expression
  - Node structured as on pag. 16, but with new operation `IDX`:

```
typedef enum {PLUS, ASSIGN, IDX, NUM, ID} Symbol;

typedef struct t_node
{
    Symbol symbol;
    struct t_node *p1, *p2;
    char *lexeme;             /* for id, num */
} Node;

typedef Node *Pnode;
```



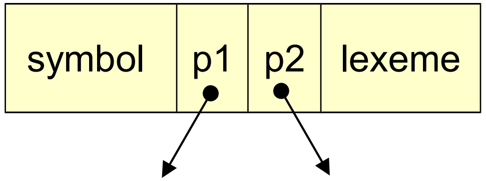
  - In general: no longer possible storing assignment var name in relevant node → assignment node: with two children (LHS, RHS), like `PLUS`
  - Indexing: only on identifiers→ possible storing them in `IDX` nodes (otherwise: two children for `IDX`):

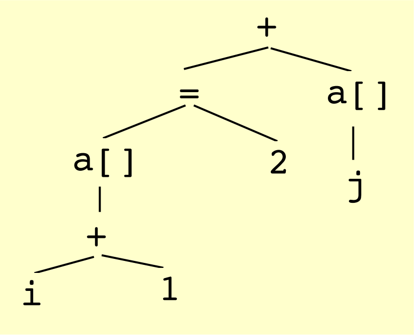
`(a[i+1] = 2) + a[j]`



Procedure of Code Gen. for References and Arrays (ii)

```
void genCode(Pnode p, int isAddr)
{
  char code[MAXCODE]; /* max length of a line of P-code */

  switch(p->symbol)
  {
  case PLUS: genCode(p->p1, FALSE);
             gencode(p->p2, FALSE);
             emit("adi");
             break;
  case ASSIGN: genCode(p->p1, TRUE);
               genCode(p->p2, FALSE);
               emit("stn");
               break;
  case IDX: sprintf(code, "lda %s", p->lexeme); emit(code);
            gencode(p->p1, FALSE);
            sprintf(code, "ixa elem_size(%s)", p->lexeme); emit(code);
            if(!isAddr) emit("ind 0");
            break;
  case NUM: sprintf(code, "ldc %s", p->lexeme); emit(code);
            break;
  case ID: if(isAddr) sprintf(code, "lda %s", p->lexeme);
           else sprintf(code, "lod %s", p->lexeme);
           emit(code);
           break;

  }
}
```





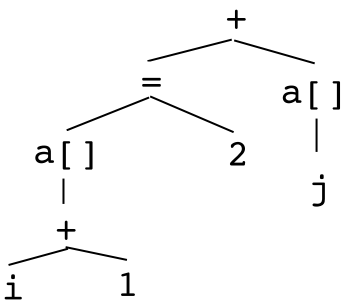
Procedure of Code Gen. for References and Arrays (iii)

- `isAddr` = inherited attribute distinguishing `IDX` and `ID` between positions ⟨ LHS / RHS

- `isAddr` = ⟨ `true` → returned address / `false` → returned value ⟩ of expressions

- Application of `genCode`:

```
(a[i+1] = 2) + a[j]
```

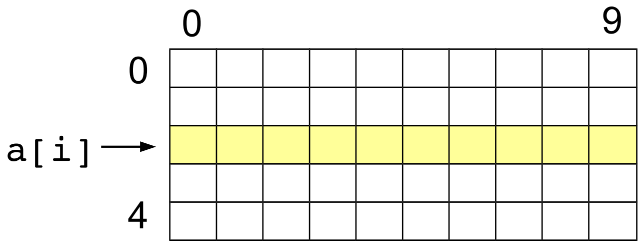


⟹

```
lda a
lod i
ldc 1
adi
ixa elem_size(a)
ldc 2
stn
lda a
lod j
ixa elem_size(a)
ind 0
adi
```

Multidimensional Arrays

```
int a[5][10];
```

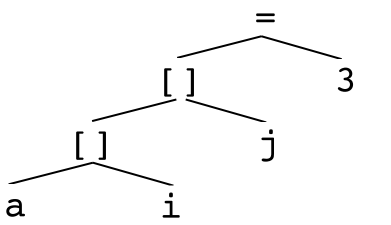


- Indexing
  - partial → array with restricted dimensions: `a[i]`= one-dimensional array
  - total → `a[i][j]` = integer

- Computation of address relevant to (partial/total) indexing expression

  → by iterated application of techniques introduced for one-dimensional arrays

```
a[i][j] = 3;
```



⇒

10*size(**int**)

```
lda a
lod i
ixa elem_size(a)
lod j
ixa size(int)
ldc 3
sto
```

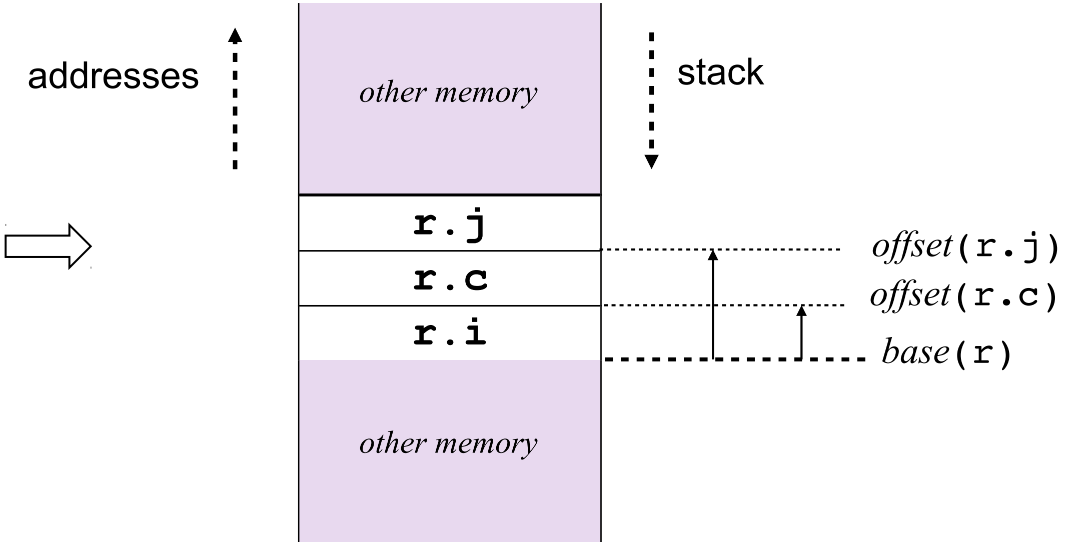
- In general: no longer possible storing array name in `IDX` node → `IDX` node with two children

Code Generation for Records and Pointers

- Computation of address of record field ≈ computation of address for indexing expression in array:

  1. Computation of base address of record
  2. Computation of offset relevant to field (typically: fixed size → static information)
  3. Address = base + offset

```
typedef struct rec
{
    int i;
    char c;
    int j;
} Rec;

  ...

Rec r;
```



addresses

stack

other memory

r.j

r.c

r.i

other memory

offset(r.j)

offset(r.c)

base(r)

Code Generation for Records and Pointers (ii)

- Notes:
  - Fields allocated linearly (typically: by increasing addresses)
  - Offset: constants
  - Offset of first field = 0
  - Offset(field) = *f* (size of data types in target machine):
    (no scale factor exists as for array because of inhomogeneity)
  - Code generation independent of target machine:

⇨ `field_offset(r, f)`

`r` = record variable

`f` = record field

typically: method of symbol table

P-code for Records and Pointers

```
typedef struct rec
{
    int i;
    char c;
    int j;
} Rec;
  …
Rec r;
```

```
r.j
```

```
lda r
ldc field_offset(r, j)
ixa 1
```

```
r.j = r.i
```

```
lda r
ldc field_offset(r, j)
ixa 1
lda r
ind field_offset(r, i)
sto
```

To get value of `r.i` without pre-computation of relevant address! (directly)

```
int *p, i;
```

```
*p = i;
```

```
lod p
lod i
sto
```

```
i = *p;
```

```
lda i
lod p
ind 0
sto
```

```
typedef struct tnode
{
    int val;
    struct tnode *p1, *p2;
} Node;
…
Node *p;
```

```
p->p1 = p;
p = p->p2;
```

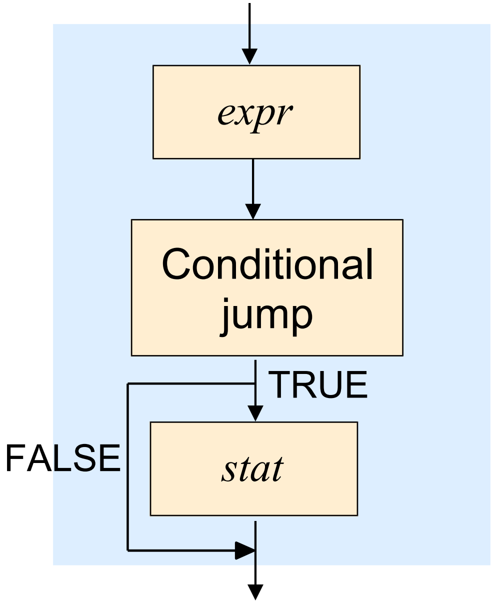
```
lod p
ldc field_offset(*p, p1)
ixa 1
lod p
sto
```

```
lda p
lod p
ind field_offset(*p, p2)
sto
```

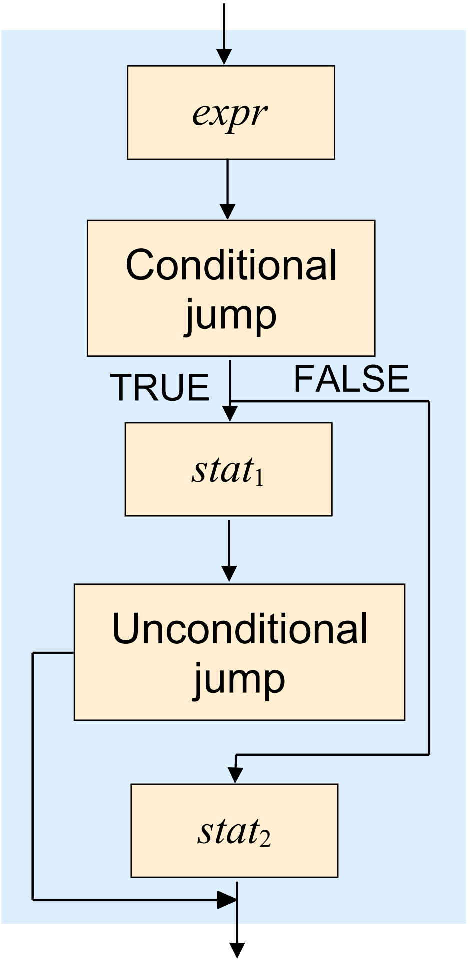
Code Generation for Control Structures

if-stat → **if (** *expr* **)** *stat* | **if (** *expr* **)** *stat* **else** *stat*
while-stat → **while (** *expr* **)** *stat*

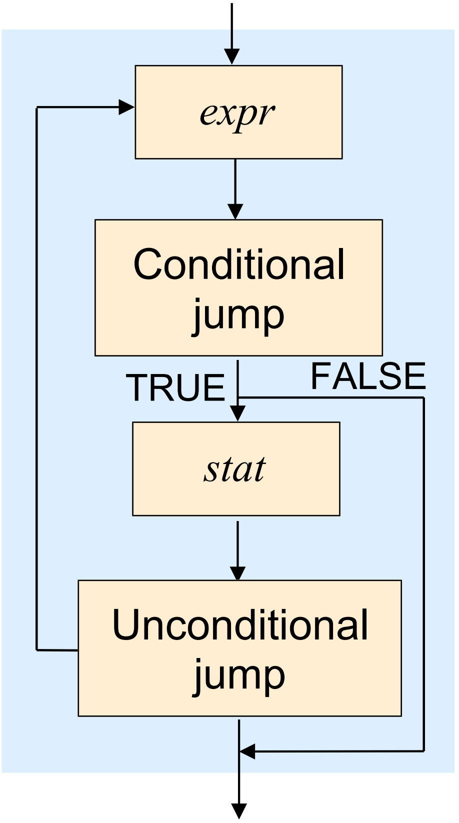
if (*expr* **)** *stat*



if (*expr* **)** *stat*$_1$ **else** *stat*$_2$



while (*expr* **)** *stat*



P-code for Control Structures

- Sufficient two kinds of jump: unconditional (**`ujp`**)
  conditional → to `FALSE` case (**`fjp`**)

if (*expr* **)** *stat*

```
⟨expr⟩
fjp L1
⟨stat⟩
lab L1
```

if (*expr* **)** $stat_1$ **else** $stat_2$

```
⟨expr⟩
fjp L1
⟨stat1⟩
ujp L2
lab L1
⟨stat2⟩
lab L2
```

while (*expr* **)** *stat*

```
lab L1
⟨expr⟩
fjp L2
⟨stat⟩
ujp L1
lab L2
```

- Notes:
  - All fragments of code end with a label statement → **exit label** of control
  - In many PLs: possible exiting loops from any point within the body

⇩

exit label = inherited attribute in code-generation functions called in loop

  - Translation mapping of **if (** *expr* **)** $stat_1$ **else** $stat_2$ still valid for conditional expressions

(*expr* **?** $expr_1$ **:** $expr_2$**)**

Code Generation for Logical Expressions

- If intermediate code has
  - `Boolean` type
  - logical operators (`and`, `or`, …)

  ⇨ Computation of boolean expression in natural way

- If ∄ `Boolean` → mapping of booleans to arithmetic values
  - `true` → 1
  - `false` → 0

- Short-circuit evaluation → need for explicit jumps to `load`
  - `(a` **`and`** `b)` → `b` evaluated only if `a = ` **`true`**
  - `(a` **`or`** `b)` → `b` evaluated only if `a = ` **`false`**

- Possible defining short circuit by conditional expression:
  - ($expr_1$ **and** $expr_2$) ≡ ($expr_1$? $expr_2$: **false**)
  - ($expr_1$ **or** $expr_2$) ≡ ($expr_1$? **true** : $expr_2$)

(x **?** y **:** z**)**		$expr_1$ **and** $expr_2$	$expr_1$ **or** $expr_2$

⟨x⟩
`fjp L1`
⟨y⟩
`ujp L2`
`lab L1`
⟨z⟩
`lab L2`
	⇨	⟨$expr_1$⟩
		`fjp L1`
		⟨$expr_2$⟩
		`ujp L2`
		`lab L1`
		`ldc FALSE`
		`lab L2`
			⟨$expr_1$⟩
			`fjp L1`
			`ldc TRUE`
			`ujp L2`
			`lab L1`
			⟨$expr_2$⟩
			`lab L2`

Logical Expressions Evaluated in Short Circuit

$expr_1$ `and` *$expr_2$*

$\langle expr_1 \rangle$
```
fjp L1
```
$\langle expr_2 \rangle$
```
ujp L2
lab L1
ldc FALSE
lab L2
```

`(x != 0) && (y == x)`

```
lod x
ldc 0
neq
fjp L1
lod y
lod x
equ
ujp L2
lab L1
ldc FALSE
lab L2
```

$expr_1$ `or` *$expr_2$*

$\langle expr_1 \rangle$
```
fjp L1
ldc TRUE
ujp L2
lab L1
```
$\langle expr_2 \rangle$
```
lab L2
```

`(x != 0) || (y == x)`

```
lod x
ldc 0
neq
fjp L1
ldc TRUE
ujp L2
lab L1
lod y
lod x
equ
lab L2
```

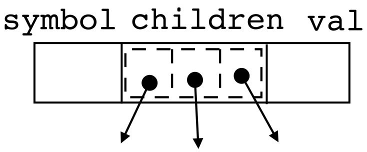
Procedures of Code Gen. for Control Structures

stat → *if-stat* | *while-stat* | **break** | **other**
if-stat → **if (** *expr* **)** *stat* | **if (** *expr* **)** *stat* **else** *stat*
while-stat → **while (** *expr* **)** *stat*
expr → **true** | **false**

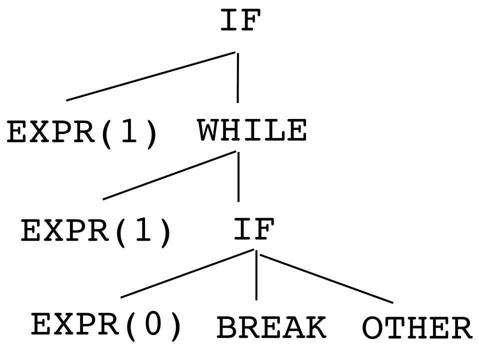
Note: Ambiguous G → ambiguity resolution by rule balancing the closest `then`

- Node of syntax tree:

```
typedef enum {EXPR, IF, WHILE, BREAK, OTHER} Symbol;
typedef struct snode
{
    Symbol symbol;
    struct snode *children[3];
    int val;
} Node;
typedef Node *Pnode;
```



```
if(true) while(true) if(false) break else other
```

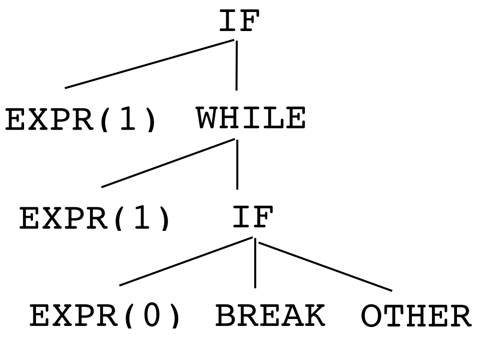


Procedures of Code Gen. for Control Structures (ii)

```
void genCode(Pnode p, char *label)
{
    char code[MAXCODE], *lab1, *lab2;

    switch(p->symbol)
    {
    case EXPR: sprintf(code, "ldc %s", (p->val == 0 ? "FALSE" : "TRUE")); emit(code); break;
    case IF: genCode(p->children[0], label); lab1 = newlab();
             sprintf(code, "fjp %s", lab1); emit(code);
             genCode(p->children[1], label);
             if(p->children[2] != NULL)
             {
                lab2 = newlab();
                sprintf(code, "ujp %s", lab2);
                emit(code);
             }
             sprintf(code, "lab %s", lab1); emit(code);
             if(p->children[2] != NULL)
             {
                genCode(p->children[2], label);
                sprintf(code, "lab %s", lab2);
                emit(code);
             }
             break;
    case WHILE: lab1 = newlab(); sprintf(code, "lab %s", lab1); emit(code);
                genCode(p->children[0], label); lab2 = newlab();
                sprintf(code, "fjp %s", lab2); emit(code);
                genCode(p->children[1], lab2); sprintf(code, "ujp %s", lab1); emit(code);
                sprintf(code, "lab %s", lab2); emit(code); break;
    case BREAK: sprintf(code, "ujp %s", label); emit(code); break;
    case OTHER: emit("other"); break;
    }
}
```

$\langle expr \rangle$
fjp L1
$\langle stat \rangle$
lab L1

$\langle expr \rangle$
fjp L1
$\langle stat_1 \rangle$
ujp L2
lab L1
$\langle stat_2 \rangle$
lab L2



lab L1
$\langle expr \rangle$
fjp L2
$\langle stat \rangle$
ujp L1
lab L2

Procedures of Code Gen. for Control Structures (iii)

- Notes:
  - `label` = additional parameter of `genCode` → to generate unconditional jump for `break`
  - `label`: changed only in recursive call to `genCode` relevant to body of `while` loop

    ⇩

    in order to exit the inner loop
  - Initial call to `genCode` → `label` = " " (empty string)

    ⇩

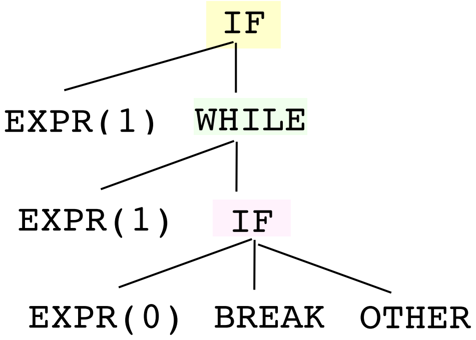
    `break` → generation of jump to empty label → error!
  - `lab1`, `lab2`: to save label names relevant to
    - jump
    - definitions

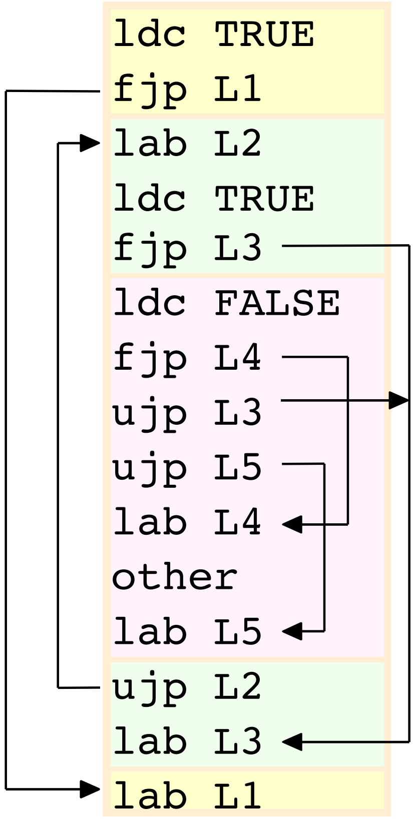
    } still dangling
  - "`other`" ≡ dummy P-code statement to complete translation mapping

Procedures of Code Gen. for Control Structures (iv)

```
if(true) while(true) if(false) break else other
```

⇒





```
ldc TRUE
fjp L1
lab L2
ldc TRUE
fjp L3
ldc FALSE
fjp L4
ujp L3
ujp L5
lab L4
other
lab L5
ujp L2
lab L3
lab L1
```

```
⟨expr⟩
fjp L1
⟨stat⟩
lab L1
```

```
⟨expr⟩
fjp L1
⟨stat₁⟩
ujp L2
lab L1
⟨stat₂⟩
lab L2
```

```
lab L1
⟨expr⟩
fjp L2
⟨stat⟩
ujp L1
lab L2
```

Code Generation for Subprograms

- Complications:
  - Diversification of call mechanisms in different target machines
  - Calls: heavily depend on organization of runtime environment (which depends on L)

⇩

difficult defining intermediate code enough general for ≠ ⟨ target architectures / runtime environments

- Intermediate code for subprograms:
  - Need for two constructs ⟨ definition / call } of subprogram ⟨ function / procedure
  - Runtime environment: built partially by ⟨ caller / called ⇨ **calling sequence**
  - Definition of subprogram:

    Entry instruction
    ⟨*body*⟩
    Return instruction

  - Call to subprogram:

    Begin-argument-computation instruction
    ⟨*arguments*⟩
    Call instruction

P-code for Subprograms

```
int f(int x, int y)
{
  return(x + y + 1);
}
```

⇨

```
ent f
lod x
lod y
adi
ldc 1
adi
ret
```

```
f(2+3, 4)
```

⇨

```
mst
ldc 2
ldc 3
adi
ldc 4
cal f
```

- <u>Notes</u>:
  - `ret`: <u>without</u> parameters (return value on top of stack)
  - `mst` = "mark stack": corresponding target code → allocates AR executing first statements of calling sequence
  - Operands computed from left to right (not necessarily)

P-code for Subprograms (ii)

program → *decl-list expr*

decl-list → *decl-list decl* | **ε**

decl → **function id (** *param-list* **)** = *expr*

param-list → *param-list* **, id** | **id**

expr → *expr* + *expr* | *call* | **num** | **id**

call → **id (** *arg-list* **)**

arg-list → *arg-list* **,** *expr* | *expr*

- Notes:
  - Program = (possibly empty) sequence of function definitions + program expression
  - ∄ variables or assignments, but only parameters, functions, and expressions
  - Unique type: integer
  - ∀ function → at least one parameter
  - Operations in expressions $\left\langle \begin{array}{l} \text{addition} \\ \text{function call} \end{array}\right.$

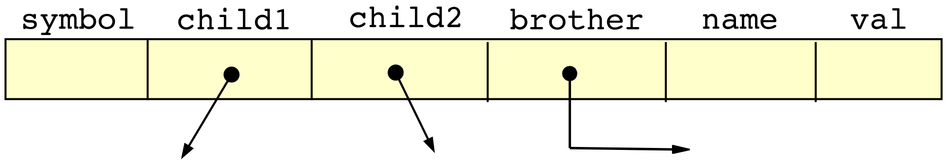
- Example of subprogram:

```
function f(x) = 2 + x
function g(x,y) = f(x) + y
g(3,4)
```

⇨ `g(3,4) = f(3) + 4 = 9`

P-code for Subprograms (iii)

- Node of syntax tree:



```
typedef enum {PROG, FUNC, PARAM, PLUS, CALL, NUM, ID} Symbol;
typedef struct snode
{
    Symbol symbol;
    struct snode *child1, *child2, *brother;
    char name;    /* with FUNC, PARAM, CALL, ID */
    int val;      /* with NUM */
} Node;
typedef Node *Pnode;
```

- Notes on typology of abstract tree:

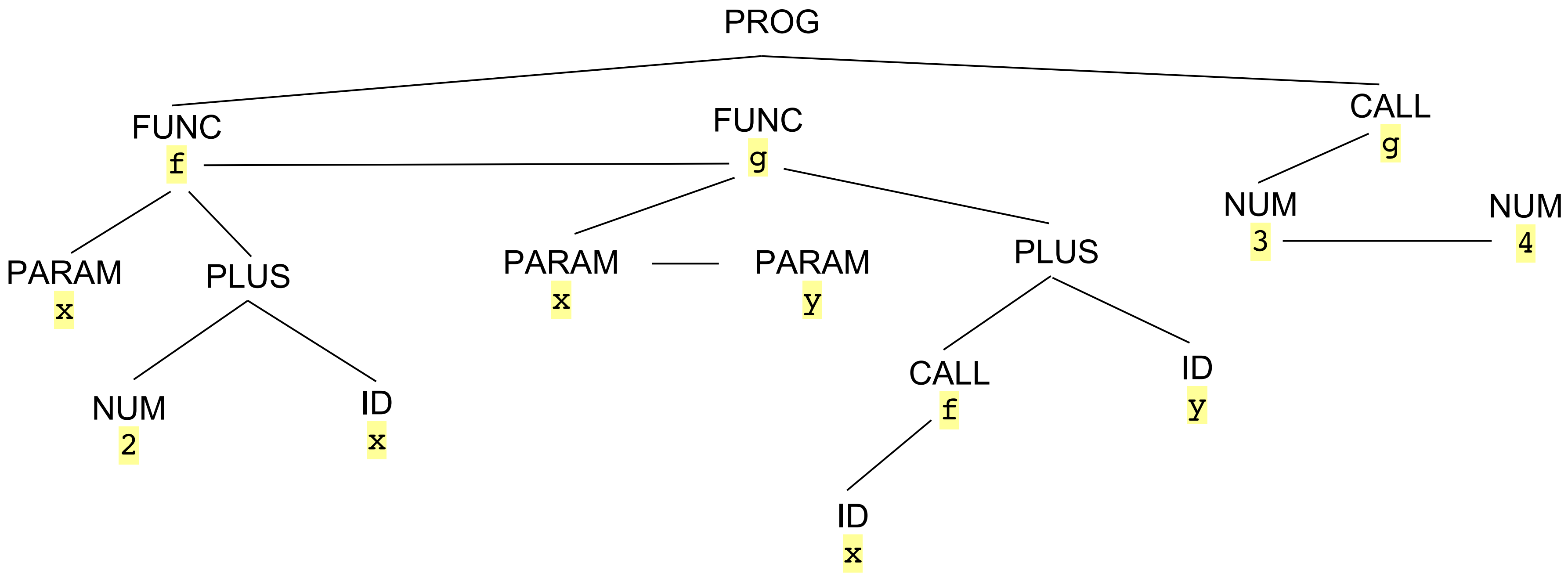
  - Root = `PROG`: associates
    - function declarations (list with head `child1`)
    - program expression (`child2`)

  - `FUNC`
    - `child1` = head of parameters
    - `child2` = function expression

  - `CALL`: `child1` = head of actual parameters

P-code for Subprograms (iv)

```
function f(x) = 2 + x
function g(x,y) = f(x) + y
g(3,4)
```

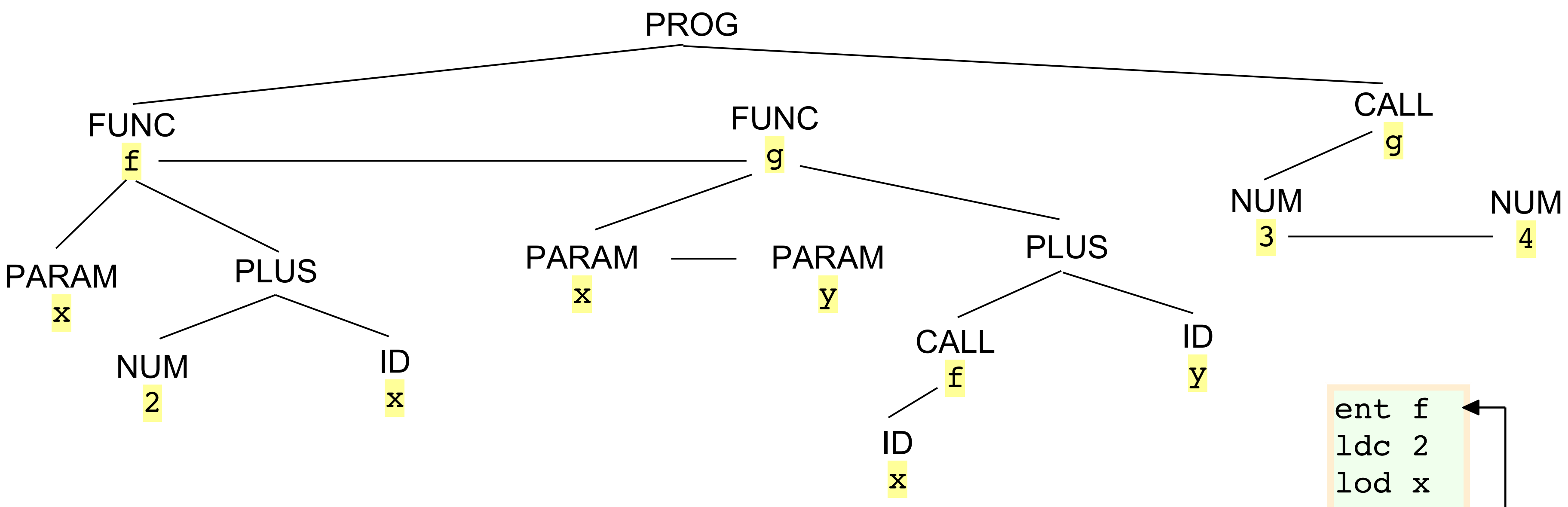


P-code for Subprograms (v)

```
void genCode(Pnode p)
{
    char code[MAXCODE]; Pnode t;

    switch(p->symbol)
    {
    case PROG: t = p->child1;
               while(t != NULL){genCode(t); t = t->brother;}
               genCode(p->child2);
               break;
    case FUNC: sprintf(code, "ent %s", p->name); emit(code);
               genCode(p->child2);
               emit("ret");
               break;
    case NUM: sprintf(code, "ldc %d", p->val); emit(code);
              break;
    case PLUS: genCode(p->child1); genCode(p->child2); emit("adi");
               break;
    case ID: sprintf(code, "lod %s", p->name); emit(code);
             break;
    case CALL: emit("mst"); t = p->child1;
               while(t != NULL){genCode(t); t = t->brother;}
               sprintf(code, "cal %s", p->name); emit(code);
               break;
    }
}
```

- Notes:
  - `FUNC`: "frames" function body with ⟨ `ent`
`ret`
  - `PARAM`: no code generation! (only previous semantic analysis)
  - `CALL`: `mst` + code generation for actual parameters + `cal`

P-code for Subprograms (vi)



```
function f(x) = 2 + x
function g(x,y) = f(x) + y
g(3,4)
```

⇨

```
ent f
ldc 2
lod x
adi
ret
ent g
mst
lod x
cal f
lod y
adi
ret
mst
ldc 3
ldc 4
cal g
```