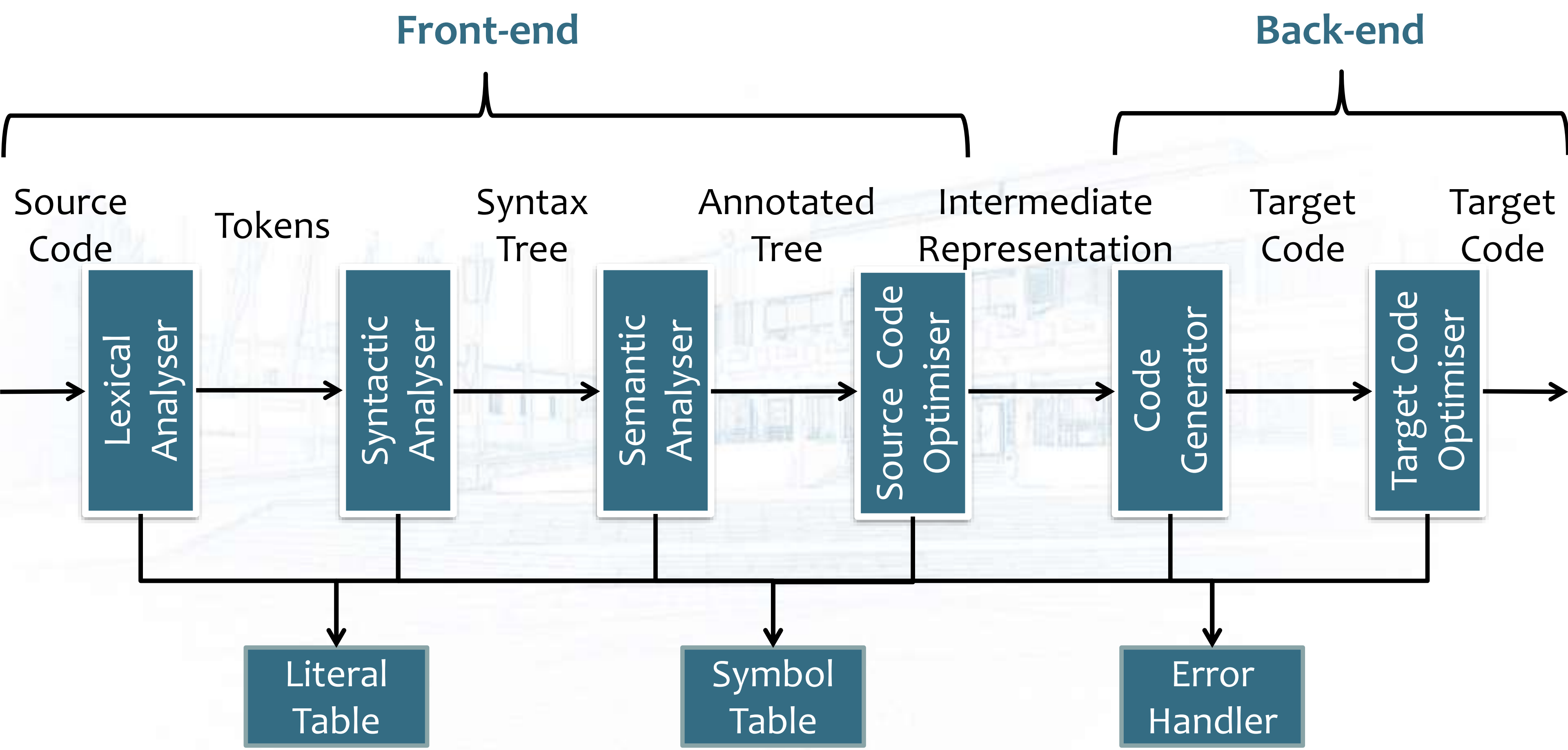
Prof. Radu Prodan

SEMANTIC ANALYSIS CODE GENERATION

Phases of a Compiler



Agenda

- **Introduction**
- Three-address code
- P-code
- Code generation techniques
- Data structure references
- Control statements and logical expressions
- Procedure and function calls

Intermediate Code

- Intermediate representation
  - Data structure representation of program during compilation
  - Abstract Syntax Tree (AST) represents source code, but not target code

- Intermediate code
  - Intermediate representation more similar to target code
  - AST linearization in sequential form

- Increases portability of compilers to new platforms
  - Only intermediate code needs translated to target code

- Intermediate code examples
  - Three-address code
  - P-code

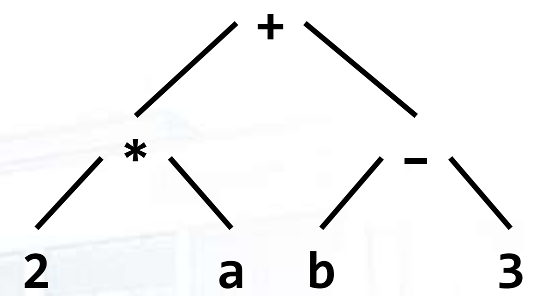
Agenda

- Introduction
- **Three-address code**
- P-code
- Code generation techniques
- Data structure references
- Control statements and logical expressions
- Procedure and function calls

Three-Address Code

- `x = y` *`op`* `z`
  - Address of **x** differs from **y** and **z**
  - **y** and **z** may represent constant or literal values with no runtime addresses
- Generate temporary variables for interior AST nodes
- Allocation of temporary variables unspecified
  - Registers, variables, …
- Unary operators such as negation
  - `t2 = - t1`
- Read, write operations
- No standard three-address code

`2 * a + (b - 3)`



- Left to right linearization

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

- Right to left linearization

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

Factorial in Pascal

```
read(x);        { input an integer }
if 0 < x then
begin           { don't compute if x <= 0 }
   fact := 1;
   repeat
      fact := fact * x;
      x := x – 1;
   until x == 0;
                { output factorial of x }
   write(fact)
end
```

▪ Copy instructions

```
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
```

Quadruple Data Structure

```
read x                      (rd, x, _, _)
t1 = x > 0                  (gt, x, 0, t1)
if_false t1 goto L1         (if_f, t1, L1, _)
fact = 1                    (asn, 1, fact, _)
label L2                    (lab, L2, _, _)
t2 = fact * x               (mul, fact, x, t2)
fact = t2                   (asn, t2, fact, _)
t3 = x - 1                  (sub, x, 1, t3)
x = t3                      (asn, t3, x, _)
t4 = x == 0                 (eq, x, 0, t4)
if_false t4 goto L2         (if_f, t4, L2, _)
write fact                  (wri, fact, _, _)
label L1                    (lab, L1, _, _)
halt                        (halt, _, _, _)
```

Triple Data Structure

```
(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, _, _, _)
```

```
(0) (rd, x, _)
(1) (gt, x, 0)
(2) (if_f, (1), (11))
(3) (asn, 1, fact)
(4) (mul, fact, x)
(5) (asn, (4), fact)
(6) (sub, x, 1)
(7) (asn, (6), x)
(8) (eq, x, 0)
(9) (if_f, (8), (4))
(10) (wri, fact, _)
(11) (halt, _, _)
```

Three-Address Code Generation

- Subset of C expressions with embedded assignments

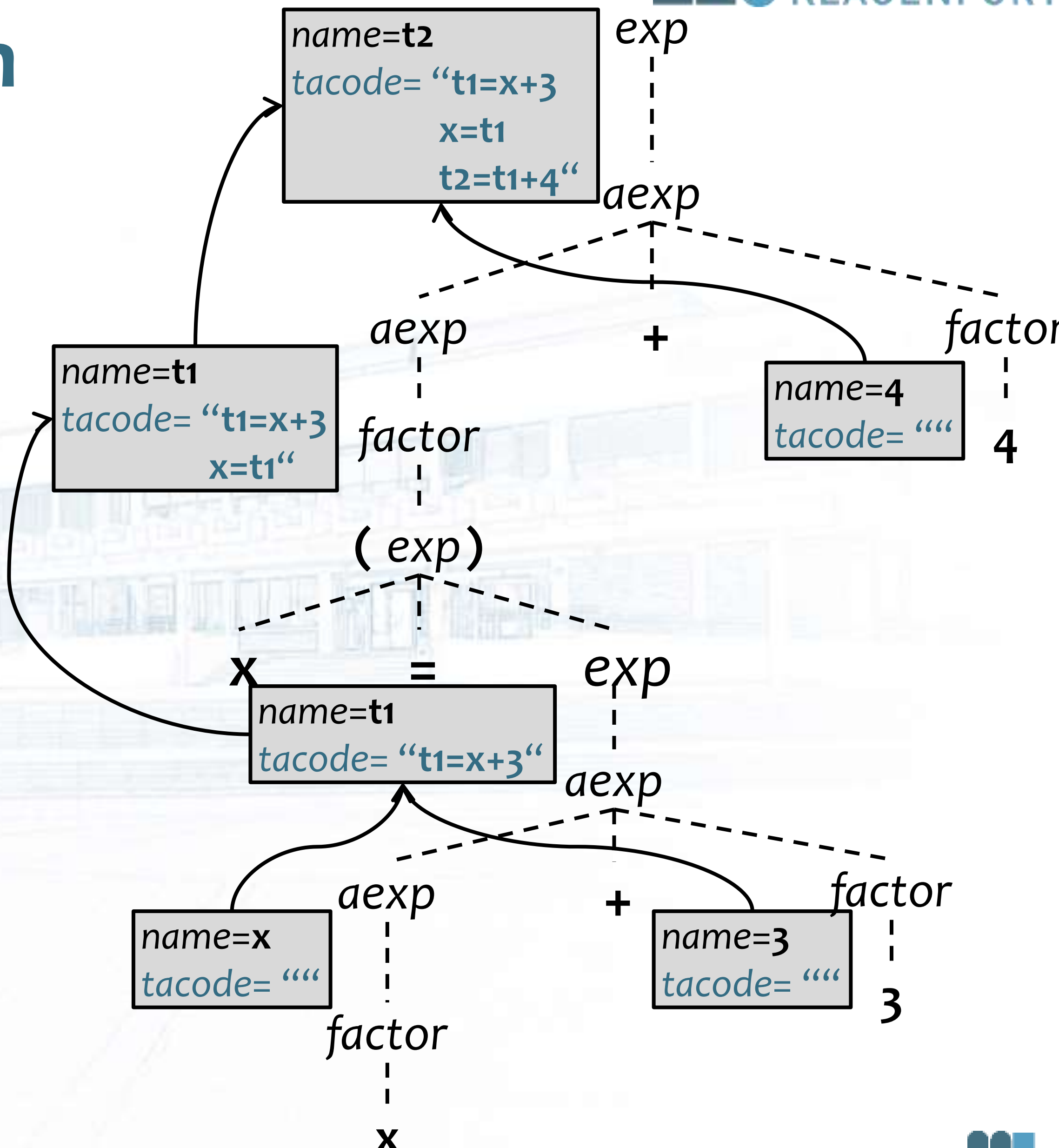
$$
\begin{aligned}
exp &\rightarrow \textbf{id} = exp \mid aexp \\
aexp &\rightarrow aexp + factor \mid factor \\
factor &\rightarrow (\ exp\ ) \mid \textbf{num} \mid \textbf{id}
\end{aligned}
$$

- ***tacode***: three-address code **synthesised attribute**
- ***name***: synthesised attribute for temporary expressions results

Three-Address Code Generation: Dependency Graph

- Input expression

```
(x=x+3)+4
```



- Three-address code

```
t1 = x + 3
x = t1
t2 = t1 + 4
```

Three-Address Code Generation: Attribute Grammar

Grammar Rule	Semantic Rules
$exp_1 \rightarrow$ **id** = exp_2	$exp_1.name = exp_2.name$ $exp_1.tacode = exp_2.tacode$ “\n” **id**.strval “=“ $exp_2.name$
$exp \rightarrow aexp$	$exp.name = aexp.name$ $exp.tacode = aexp.tacode$
$aexp_1 \rightarrow aexp_2$ + $factor$	$aexp_1.name$ = ***newtemp***( ) $aexp_1.tacode = aexp_2.tacode$ “\n” $factor.tacode$ “\n” $aexp_1.name$ “=“ $aexp_2.name$ “+” $factor.name$
$aexp \rightarrow factor$	$aexp.name = factor.name$ $aexp.tacode = factor.tacode$
$factor \rightarrow$ (exp)	$factor.name = exp.name$ $factor.tacode = exp.tacode$
$factor \rightarrow$ **num**	$factor.name$ = **num**.strval $factor.tacode$ = “”
$factor \rightarrow$ **id**	$factor.name$ = **id**.strval $factor.tacode$ = “”

Agenda

- Introduction
- Three-address code
- <u>**P-code**</u>
- Code generation techniques
- Data structure references
- Control statements and logical expressions
- Procedure and function calls

P-Code

- Produced by several Pascal compilers in 1970s and early 1980s
- Designed for hypothetical stack machine
- Closer to machine code than three address code
- All instructions have one or zero addresses
- Instructions operate on stack
- **`mpi`**, **`adi`**, and **`sbi`** pop two values from top of stack and push result
- **`sto`** stores value from top of stack to address below it and pops both

- `2 * a + (b – 3)`

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

- `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
```

Factorial P-Code

```
lda x     ; load address of x
rdi       ; read an integer, store
          ; to address on top of
          ; the stack and pop it
lod x     ; load the value of x
ldc 0     ; load constant 0
grt       ; pop, compare two values
          ; and 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       ; store top to address
          ; of 2nd and pop
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 2nd and pop
lda x     ; load address of x
lod x     ; load value of x
ldc 1     ; load constant 1
sbi       ; integer subtraction
sto       ; store top to address
          ; of 2nd and pop
lod x     ; load value of x
ldc 0     ; load constant 0
equ       ; pop two values,
          ; test equality and
          ; push boolean result
fjp L2    ; pop boolean value
          ; jump to L2 if false
lod fact  ; load value of fact
wri       ; write top of stack
          ; and pop
lab L1    ; definition of label L1
stp
```

```
read(x);
if 0 < x then
begin
 fact := 1;
  repeat
   fact := fact * x;
   x := x - 1;
  until x == 0;
 write(fact)
end
```

P-Code Code Generation

- Input expression

```
(x=x+3)+4
```

- P-Code

```
lda x
lod x
ldc 3
adi
stn
ldc 4
adi
```

- **`stn`**
  - **Non-destructive store**
  - Same as **`sto`**, but leaves value at top of stack
- ***pcode***
  - P-code **synthesised attribute**

P-Code Code Generation: Attribute Grammar

Grammar Rule	Semantic Rules
$exp_1 \rightarrow$ **id** = exp_2	$exp_1.pcode$ = "**lda** " **id**.strval "\n" $exp_2.pcode$ "\n" "**stn**"
$exp \rightarrow aexp$	$exp.pcode = aexp.pcode$
$aexp_1 \rightarrow aexp_2$ + $factor$	$aexp_1.pcode = aexp_2.pcode$ "\n" $factor.pcode$ "\n" "**adi**"
$aexp \rightarrow factor$	$aexp.pcode = factor.pcode$
$factor \rightarrow$ (exp)	$factor.pcode = exp.pcode$
$factor \rightarrow$ **num**	$factor.pcode$ = "**ldc** " **num**.strval
$factor \rightarrow$ **id**	$factor.pcode$ = "**lod** " **id**.strval

Agenda

- Introduction
- Three-address code
- P-code
- **Code generation techniques**
- Data structure references
- Control statements and logical expressions
- Procedure and function calls

Code Generation Techniques

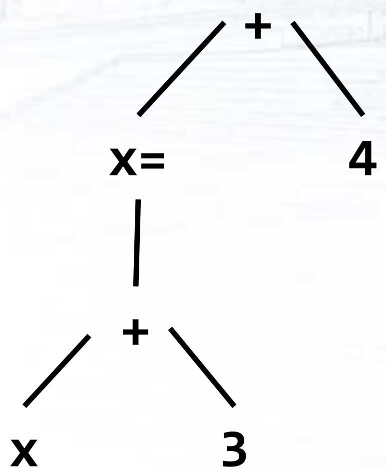
- Modified postorder traversal
  - Contains preorder and inorder components for generating preparation code

```
procedure genCode ( T : treenode ) ;
begin
  if T ≠ nil then
    generate code to prepare for code of left-child(T) ;
    genCode (left-child(T)) ;
    generate code to prepare for code of right-child(T) ;
    genCode (right-child(T)) ;
    generate code to implement the action of T ;
end ;
```

AST for Embedded Expressions

exp → **id** = *exp* | *aexp*
aexp → *aexp* + *factor*
        | *factor*
factor → (*exp*)
        | **num** | **id**

(x=x+3)+4




```
typedef enum { OpKind, ConstKind,
                           IdKind } NodeKind;

typedef enum { Plus, Assign } Optype;

typedef struct streenode {
   NodeKind kind;
   Optype op;     /* OpKind */
   int val;       /* ConstKind */
   char *strval; /* IdKind */
   struct streenode *lchild, *rchild;
} STreeNode;

typedef STreeNode *SyntaxTree;
```

Post Order P-Code Generation

```
void genCode ( SyntaxTree t ) {
  char codestr[CODESIZE];
  if (t != NULL) {
    switch (t->kind) {
    case OpKind:
      switch (t->op) {
      case Plus:     /*postorder */
        genCode(t->lchild);
        genCode(t->rchild);
        emitCode("adi");
        break;
      case Assign:
        /* pre- and postorder */
        sprintf(codestr, "%s %s",
                  "lda", t->strval);
        emitCode(codestr);
        genCode(t->lchild);
        emitCode("stn");
        break;
      default:
        emitCode("Error");
      }
      break;
    case ConstKind:
      sprintf(codestr, "%s %d",
                      "ldc", t->val);
      emitCode(codestr);
      break;
    case IdKind:
      sprintf(codestr,
            "%s %s", "lod", t->strval);
      emitCode(codestr);
      break;
    default:
      emitCode("Error");
    }
  }
}
```

Target Code Generation

- Generation of target code from intermediate code
- Final pass over intermediate code
- Intermediate code highly symbolic with little to no information about runtime environment or target machine
  - Supply all actual locations of variables and temporaries
  - Code necessary to maintain runtime environment
  - Register allocation
  - Maintenance of information on register use

Agenda

- Introduction
- Three-address code
- P-code
- Code generation techniques
- **Data structure references**
- Control statements and logical expressions
- Procedure and function calls

Three-Address Code Addresses

- Two new operators
  - **&: address of**
  - ***: indirect**

- Store value 2 at address of variable **x** plus 10 bytes

  t1 = &x + 10

  ***t1 = 2**

- Augment quadruple data structure on with one address mode field

```
typedef enum { None, Address, Indirect } AddrMode;

typedef struct {
   OpKind op;
   Address addr1, addr2, addr3;
   AddrMode mode1, mode2, mode3;
} Quint;
```

Three-Address Code Arrays

- Access **a[t]** using a base address and scale factor
  - *base_address*(**a**) + (**t** – *lower_bound*(**a**)) * *element_size*(**a**)
- Address of C array reference `a[i+1]`
  - `a + (i+1) * sizeof(int)`
- **Fetch operation: `t2 = a[t1]`**
- **Assign operation: `a[t2] = t1`**

C Code	Three-Address Code

`a[i+1] = a[j*2] + 3`	`t1 = j * 2`
	`t2 = a[t1]`
	`t3 = t2 + 3`
	`t4 = i + 1`
	`a[t4] = t3`

C Code	Three-Address Code

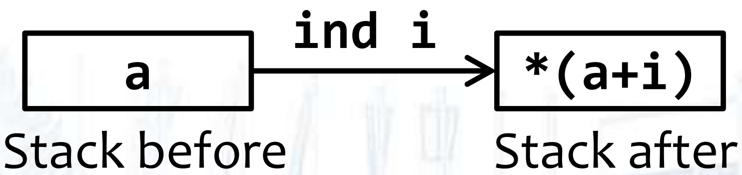
`t2 = a[t1]`	`t3 = t1 * elem_size(a)`
	`t4 = &a + t3`
	`t2 = *t4`

`a[t2] = t1`	`t3 = t2 * elem_size(a)`
	`t4 = &a + t3`
	`*t4 = t1`

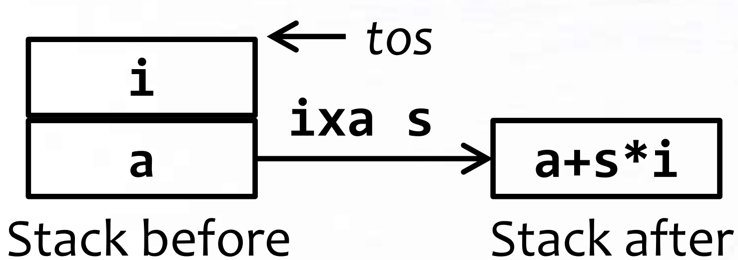
`a[i+1] = a[j*2] + 3`	`t1 = j * 2`
	`t2 = t1 * elem_size(a)`
	`t3 = &a + t2`
	`t4 = *t3`
	`t5 = t4 + 3`
	`t6 = i + 1`
	`t7 = t6 * elem_size(a)`
	`t8 = &a + t7`
	`*t8 = t5`

Address Calculation in P-Code

- **Indirect load: `ind`**
  - Pointer dereferencing



- **Indexed address: `ixa`**
  - **`a[i]`** with element size **`s`**



- Store value 2 at address of variable **`x`** plus 10 bytes

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

P-Code Array References

C Code	P-Code

`t2 = a[t1]`	`lda t2`
	`lda a`
	`lod t1`
	`ixa elem_size(a)`
	`ind 0`
	`sto`

`a[t2] = t1`	`lda a`
	`lod t2`
	`ixa elem_size(a)`
	`lod t1`
	`sto`

C Code	P-Code

`a[i+1] = a[j*2] + 3`	`lda a`
	`lod i`
	`ldc 1`
	`adi`
	`ixa elem_size(a)`
	`lda a`
	`lod j`
	`ldc 2`
	`mpi`
	`ixa elem_size(a)`
	`ind 0`
	`ldc 3`
	`adi`
	`sto`

P-Code Generation with Arrays

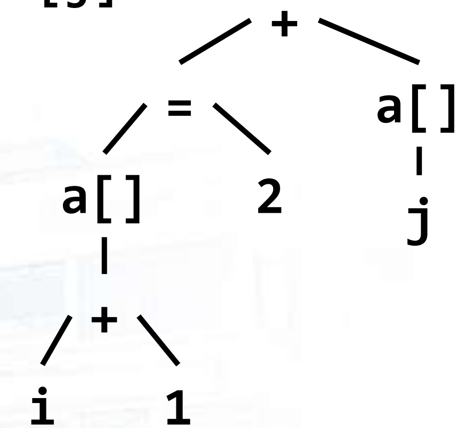
exp → *subs* = *exp* | *aexp*

aexp → *aexp* + *factor* | *factor*

factor → (*exp*) | **num** | *subs*

subs → **id** | **id** [*exp*]

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




```
typedef enum { Plus, Assign, Subs } Optype;
/* other AST declarations as on slide 20 */
```

```
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
```

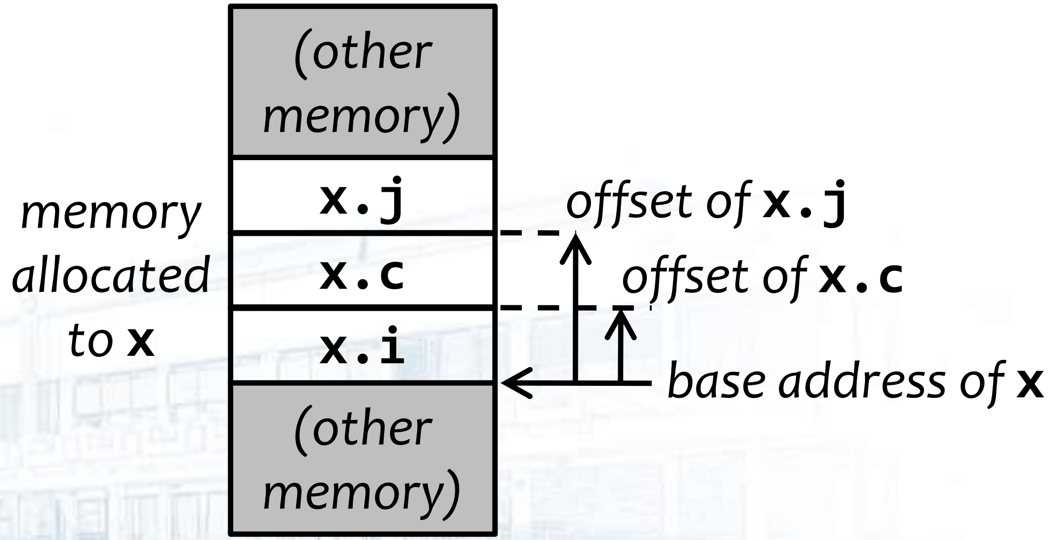
- Inherited attribute **isAddr** for *subs* code generation
  - **TRUE**: return **address** of left expression
  - **FALSE**: return **value** of right expression

P-Code Generation Procedure

```
void genCode ( SyntaxTree t, int isAddr ) {
  char codestr[CODESIZE]; /* max length of P-Code line */
  if (t != NULL) {
    switch (t->kind) {
    case OpKind:
      switch (t->op) {
      case Plus:
        if (isAddr) emitCode("Error");
        else {
          genCode(t->lchild, FALSE); /* aexp */
          genCode(t->rchild, FALSE); /* factor */
          emitCode("adi");
        }
        break;
      case Assign:
        genCode(t->lchild, TRUE);      /* subs */
        genCode(t->rchild, FALSE);     /* exp */
        emitCode("stn");
        break;
      case Subs:
        sprintf(codestr, "%s %s", "lda", t->strval);/* id */
        emitCode(codestr);
        genCode(t->lchild, FALSE);     /* exp */
        sprintf(codestr, "%s%s%s", "ixa elemsize(", t->strval, ")");
        emitCode(codestr);
        if (! isAddr) emitCode("ind 0"); /* dereference */
        break;
      default:
        emitCode("Error");
      }
      break;
    case ConstKind:
      if (isAddr) emitCode("Error");
      else { /* num */
        sprintf(codestr, "%s %d", "ldc", t->val);
        emitCode(codestr);
      }
      break;
    case IdKind:
      if (isAddr)         /* id */
        sprintf(codestr, "%s %s", "lda", t->strval)
      else sprintf(codestr, "%s %s", "lod", t->strval);
      emitCode(codestr);
      break;
    default:
      emitCode("Error");
    }                     /* switch */
  }                       /* if */
}
```

Record Structure and Pointers

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



- Base name plus usually fixed offset
- New function **field_offset**
  - Input: structure variable **x**, field name **j**
  - Output: field offset **x.j**

```
x.j = field_offset(x, j)
```

Three-Address Code for Record Structures and Pointers

<table>
<tr><th>C Code</th><th>Three-Address Code</th></tr>
<tr><td><pre>t1 = x.j</pre></td><td><pre>t1 = &x + field_offset(x, j)</pre></td></tr>
<tr><td><pre>x.j = x.i</pre></td><td><pre>t1 = &x + field_offset(x, j)
t2 = &x + field_offset(x, i)
*t1 = *t2</pre></td></tr>
<tr><td><pre>int *x; *x = i;</pre></td><td><pre>*x = i</pre></td></tr>
<tr><td><pre>i = *x;</pre></td><td><pre>i = *x</pre></td></tr>
<tr><td><pre>typedef struct treenode {
  int val;
  struct treenode *lchild, *rchild;
} TreeNode;
. . .

TreeNode *p;
p->lchild = p;
p = p->rchild;</pre></td><td><pre>t1 = p + field_offset(*p, lchild)
*t1 = p
t2 = p + field_offset(*p, rchild)
p = *t2</pre></td></tr>
</table>

P-Code for Structures and Pointers

C Code	P-Code

`x.j`	`lda x`
	`ldc field_offset(x, j)`
	`ixa 1`

`x.j = x.i`	`lda x`
	`ldc field_offset(x, j)`
	`ixa 1`
	`lda x`
	`ind field_offset(x, i)`
	`sto`

`*x = i`	`lod x`
	`lod i`
	`sto`

C Code	P-Code

`i = *x`	`lda i`
	`lod x`
	`ind 0`
	`sto`

`p->lchild = p`
`p = p->rchild`
	`lod p`
	`ldc field_offset(*p, lchild)`
	`ixa 1`
	`lod p`
	`sto`
	`lda p`
	`lod p`
	`ind field_offset(*p, rchild)`
	`sto`

Agenda

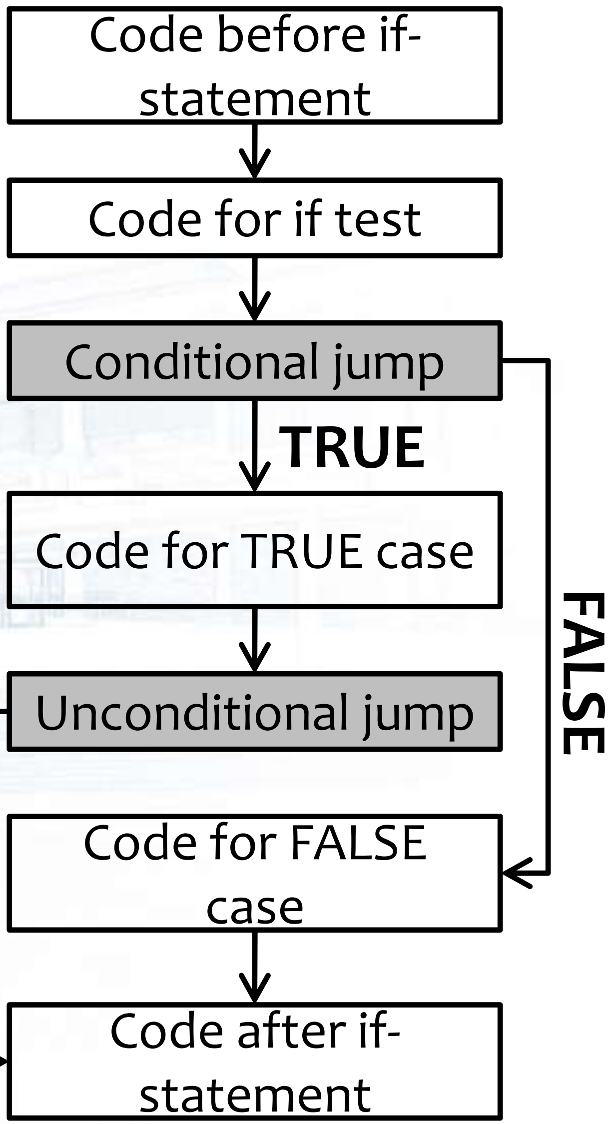
- Introduction
- Three-address code
- P-code
- Code generation techniques
- Data structure references
- **Control statements and logical expressions**
- Procedure and function calls

Code Generation for Control Statements

- `if`, `while`, `repeat`, `for`, `break`, …
- Involves generation of labels
  - Instruction addresses in target code
  - Similar to temporaries for variables
- Forward jumps to code locations not yet known
- Backpatch
  - Generate dummy jump instruction to fake location
  - When jump is known, location is used to fix (backpatch) missing code

Code Generation for If-Statements

if-stmt → **if** (*exp*) *stmt*
        | **if** (*exp*) *stmt* **else** *stmt*



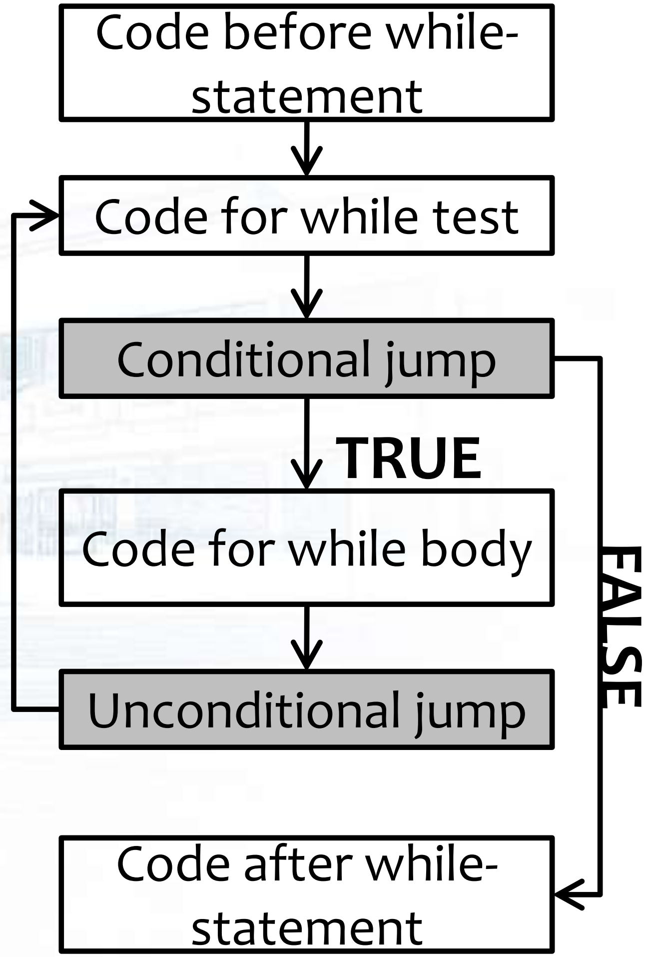
if (*E*) *S1* **else** *S2*

Three-address code	P-code
<code to evaluate *E* to t1>	<code to evaluate *E*>
if_false t1 goto L1	fjp L1
<code for *S1*>	<code for *S1*>
goto L2	ujp L2
label L1	lab L1
<code for *S2*>	<code for *S2*>
label L2	lab L2

Code Generation for While-Statements

while-stmt → **while** (*exp*) *stmt*

while (*E*) *S*



Three-address code	P-code
label L1	lab L1
<code to evaluate *E* to t1>	<code to evaluate *E*>
if_false t1 goto L2	fjp L2
<code for *S*>	<code for *S*>
goto L1	ujp L1
label L2	lab L2

Code Generation for Logical Expressions

- Most architectures do not have built-in boolean type
- Boolean values often represented arithmetically
  - `False`: 0
  - `True`: 1
  - Standard bitwise **and** and **or** operations
  - Comparison operations (**<**, **>**, **=**) normalised to 0 or 1
- Short circuit logical operations
  - `if ((p != NULL) && (p->val == 0)) ...`
  - *a* **and** *b* ≡ **if** *a* **then** *b* **else false**
  - *a* **or** *b* ≡ **if** *a* **then** *true* **else** *b*

- Short circuit P-code for C expression

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

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

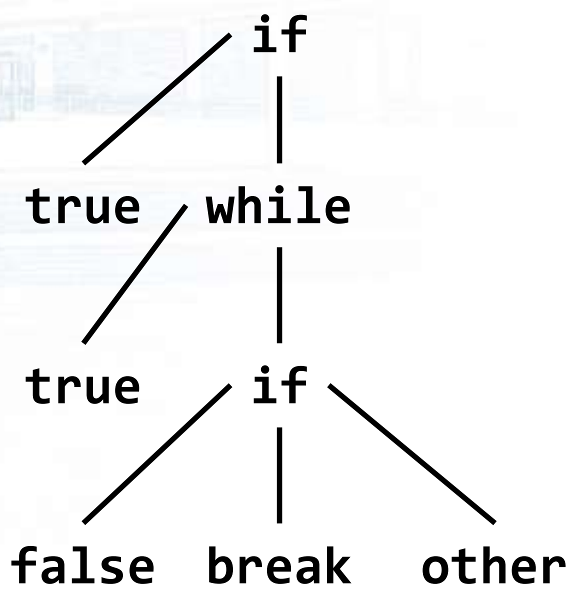
Example:
If and While Statement Grammar

stmt → *if-stmt* | *while-stmt* | **break** | **other**
if-stmt → **if** (*exp*) *stmt*
  | **if** (*exp*) *stmt* **else** *stmt*
while-stmt → **while** (*exp*) *stmt*
exp → **true** | **false**

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

```
typedef enum { ExpKind, IfKind, WhileKind,
               BreakKind, OtherKind } NodeKind;

typedef struct streenode
{  NodeKind kind;
   struct streenode *child[3];
   int val;          /* used with ExpKind */
}  STreeNode;
typedef STreeNode *SyntaxTree;
```



Code Generation for If and While Statements

```
void genCode ( SyntaxTree t, char *label  ) {
  char corestr[CODESIZE];
  char *lab1, *lab2;
  if (t == NULL) return;
  switch(t->kind) {
  case ExpKind:
    if (t->val == 0) emitCode(“ldc false“);
    else emitCode(“ldc true“);
    break;
  case IfKind:
    genCode(t->child[0], label); /* expression */
    lab1 = genLabel();
    sprintf(codestr, “%s %s“, “fjp“, lab1);
    emitCode(codestr);
    genCode(t->child[1], label); /* then */
    if (t->child[2] != NULL) {
      lab2 = genLabel();
      sprintf(codestr, “%s %s“, “ujp“, lab2);
      emitCode(codestr);
    }
    sprintf(codestr, “%s %s“, “lab“, lab1);
    emitCode(codestr);
    if (t->child[2] != NULL) {
      genCode(t->child[2], label);  /* else */
      sprintf(codestr, “%s %s“, “lab“, lab2);
      emitCode(codestr);
    }
    break;
  case WhileKind:
    lab1 = genLabel();
    sprintf(codestr, “%s %s“, “lab“, lab1);
    emitCode(codestr);
    genCode(t->child[0], label); /* exp */
    lab2 = genLabel();
    sprintf(codestr, “%s %s“, “fjp“, lab2);
    emitCode(codestr);
    genCode(t->child[1], lab2); /* body */
    sprintf(codestr, “%s %s“, “ujp“, lab1);
    emitCode(codestr);
    sprintf(codestr, “%s %s“, “lab“, lab2);
    emitCode(codestr);
    break;
  case BreakKind:
    sprintf(codestr, “%s %s“, “ujp“, label);
    emitCode(codestr);
    break;
  case OtherKind:
    emitCode(“Other“);
    break;
  default:
    emitCode(“Error“)
    break;
  }
}
```

Agenda

- Introduction
- Three-address code
- P-code
- Code generation techniques
- Data structure references
- Control statements and logical expressions
- **Procedure and function calls**

Code Generation for Procedure and Function Definition

- Name, parameters, and code
- Minimal intermediate code for a function/procedure definition

Entry instruction

<code for the function body>

Return instruction

- Signature and parameter information stored in symbol table

- Three-address code

C Function	Three-Address Code
`int f(int x, int y) { return x+y+1; }`	`entry f` `t1=x+y` `t2=t1+1` `return t2`

- P-Code

C Function	P-Code
`int f(int x, int y) { return x+y+1; }`	`ent f` `lod x` `lod y` `adi` `ldc 1` `adi` `ret`

Code Generation for Procedure and Function Calls

- Minimal intermediate code for function / procedure call
  - Actual parameters to the call
- Stack frame information stored in symbol table
  - Number, size, location of parameters
  - Size of stack frame
  - Size of local variables and temporary space
  - Indications of register usage
- Calling sequence managed by runtime environment

Begin-argument-computation instruction
<code to compute the arguments>
Call instruction

- Three-address code

C Function	Three-Address Code

`f(2+3, 4)`	`begin_args`
	`t1=2+3`
	`arg t1`
	`arg 4`
	`call f`

- P-Code
  - **`mst`**: mark stack (set up activation record)
  - **`cup/csp`**: call user/standard procedure

C Function	P-Code

`f(2+3, 4)`	`mst`
	`ldc 2`
	`ldc 3`
	`adi`
	`ldc 4`
	`cup f`

Example: Function Definition and Call

Grammar

program → *decl-list exp*

decl-list → *decl-list decl* | ε

decl → **fn id (** *param-list* **) =** *exp*

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

exp → *exp* **+** *exp* | *call* | **num** | **id**

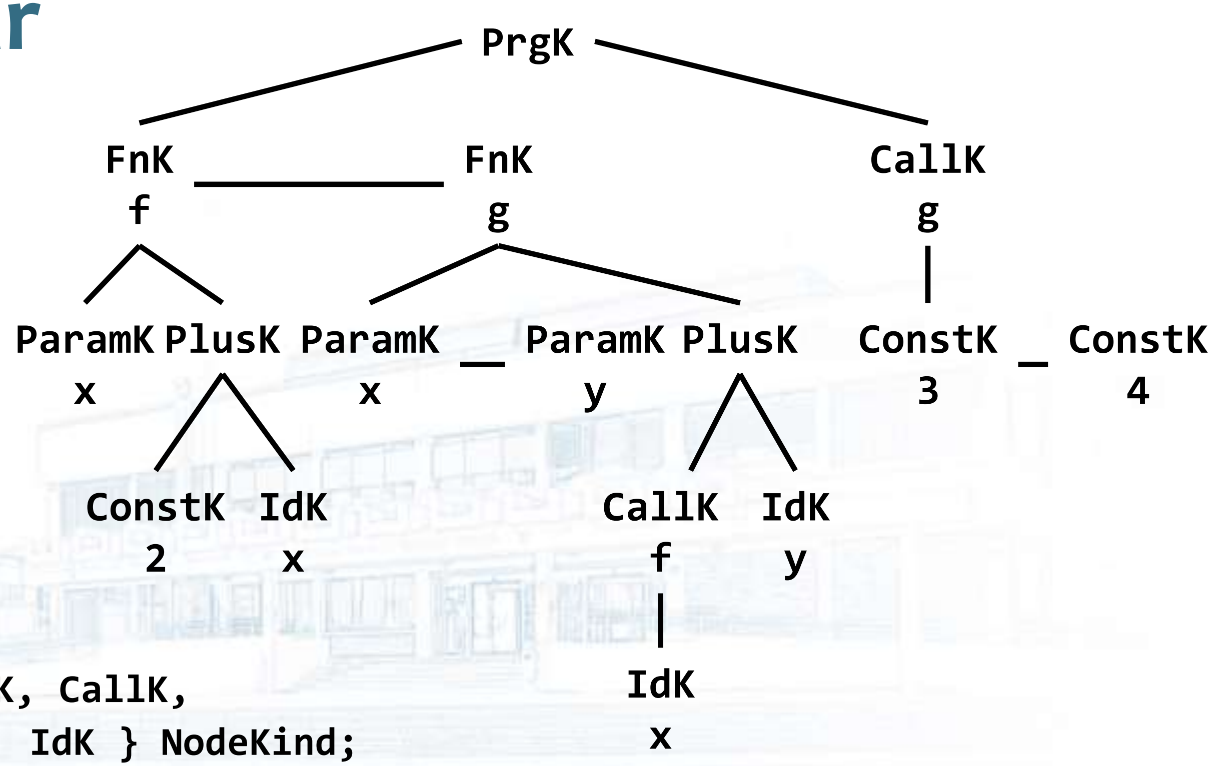
call → **id (** *arg-list* **)**

arg-list → *arg-list* **,** *exp* | *exp*

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

AST for Function Definition and Call Grammar

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




```
typedef enum { PrgK, FnK, ParamK, CallK,
                         ConstK, IdK } NodeKind;
typedef struct streenode {
  NodeKind kind;
  struct streenode *lchild, *rchild, *sibling;
  char *name;      /* FnK, ParamK, CallK, IdK */
  int val;         /* ConstK */
} STreeNode;
typedef STreeNode *SyntaxTree;
```

Code Generation for Function Definition and Call

```
void genCode (SyntaxTree t) {
  char codestr[CODESIZE];
  SyntaxTree p;
  if(t == NULL) return;
  switch(t->kind) {
  case PrgK:
    p = t->lchild;    /* decl-list */
    while (p != NULL) {
      genCode(p);     /* decl */
      p = p->sibling; }
    genCode(t->rchild); /* exp /*
    break;
  case FnK:
    sprintf(codestr, “%s %s“,
                        “ent“, t->name);
    emitCode(codestr);
    genCode(t->lchild); /* param-list */
    genCode(t->rchild); /* exp */
    emitCode(“ret“);
    break;
  case ParamK:        /* no actions */
    break;
  case ConstK:
    sprintf(codestr, “%s %d“,
                         “ldc“, t->val);
    emitCode(codestr);
    break;
  case PlusK:
    genCode(t->lchild);
    genCode(r->rchild);
    emitCode(“adi“);
    break;
  case IdK:
    sprintf(codestr, “%s %s“,
                        “lod“, t->name);
    emitCode(codestr);
    break;
  case CallK:
    emitCode(“mst“);
    p = t->rchild;
    while(p != NULL) {
      genCode(p); /* arg-list */
      p = p->sibling; }
    sprintf(codestr, “%s %s“,
                        “cup“, t->name);
    emitCode(codestr);
    break;
  default:
    emitCode(“Error“);
    break;
  }
}
```

Original Code	P-Code
`fn f(x) = 2 + x`	ent f ldc 2 lod x adi ret
`fn g(x, y) = f(x) + y`	ent g lod x cup f lod y adi ret
`g(3, 4)`	mst ldc 3 ldc 4 cup g

Conclusions

- Introduction
- Three-address code
- P-code
- Code generation techniques
- Data structure references
- Control statements and logical expressions
- Procedure and function calls