Experiment in
Compiler Construction

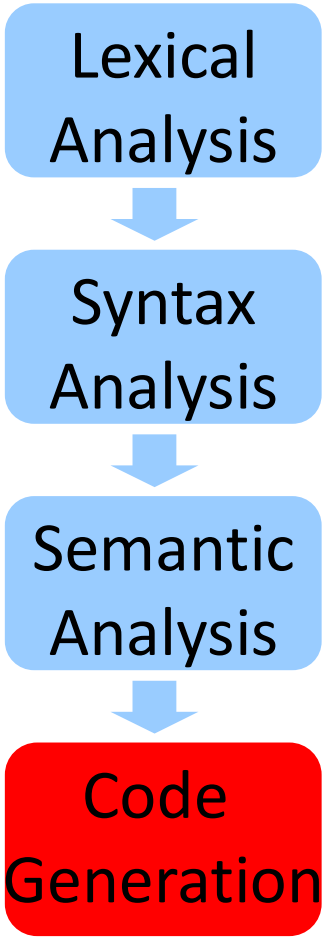
Code Generation

ONE LOVE. ONE FUTURE.

Overview

- Code generation overview
- Stack calculator
  - Stack calculator's memory
  - Instruction set
  - kplrun
- Code generation for a source code without array and subroutine
- Code generation for a source code with array and subroutine

What is code generation?

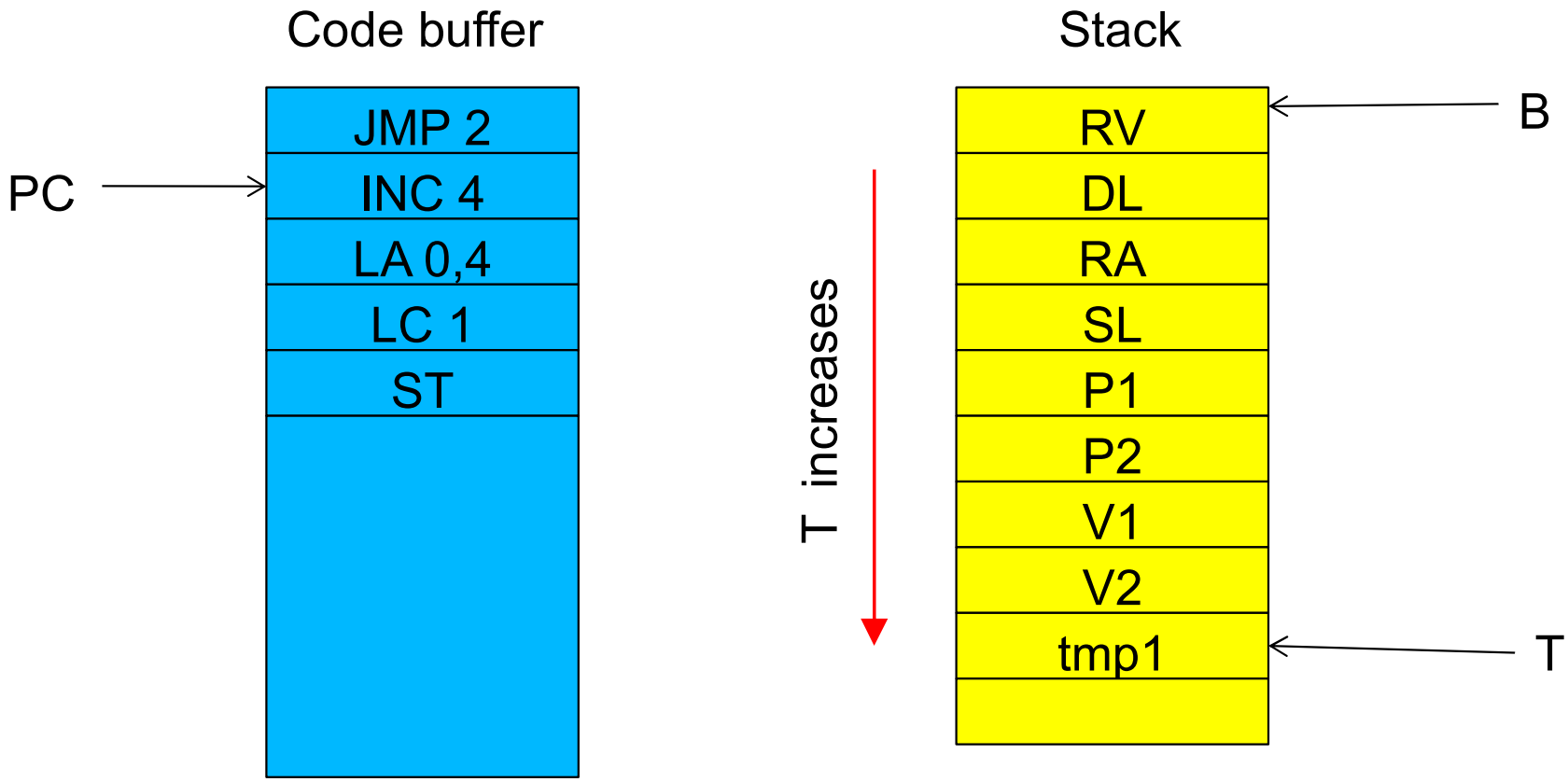


- Code generation is the phase that generates a sequence of target machine instructions corresponding to the source program's grammar.
- Program's grammar is checked and built by the syntax analyzer (parser)
- Target machine instructions are specified in execution model of target machine

Stack calculator

- Stack calculator is a computing system
  - Using stack to store intermediate results during computation process.
  - Simple organization
  - Simple instruction set
- Stack calculator consists of 2 memory areas
  - Code buffer: containing execution code corresponding to source program
  - Stack: storing intermediate results

Stack calculator



Stack calculator

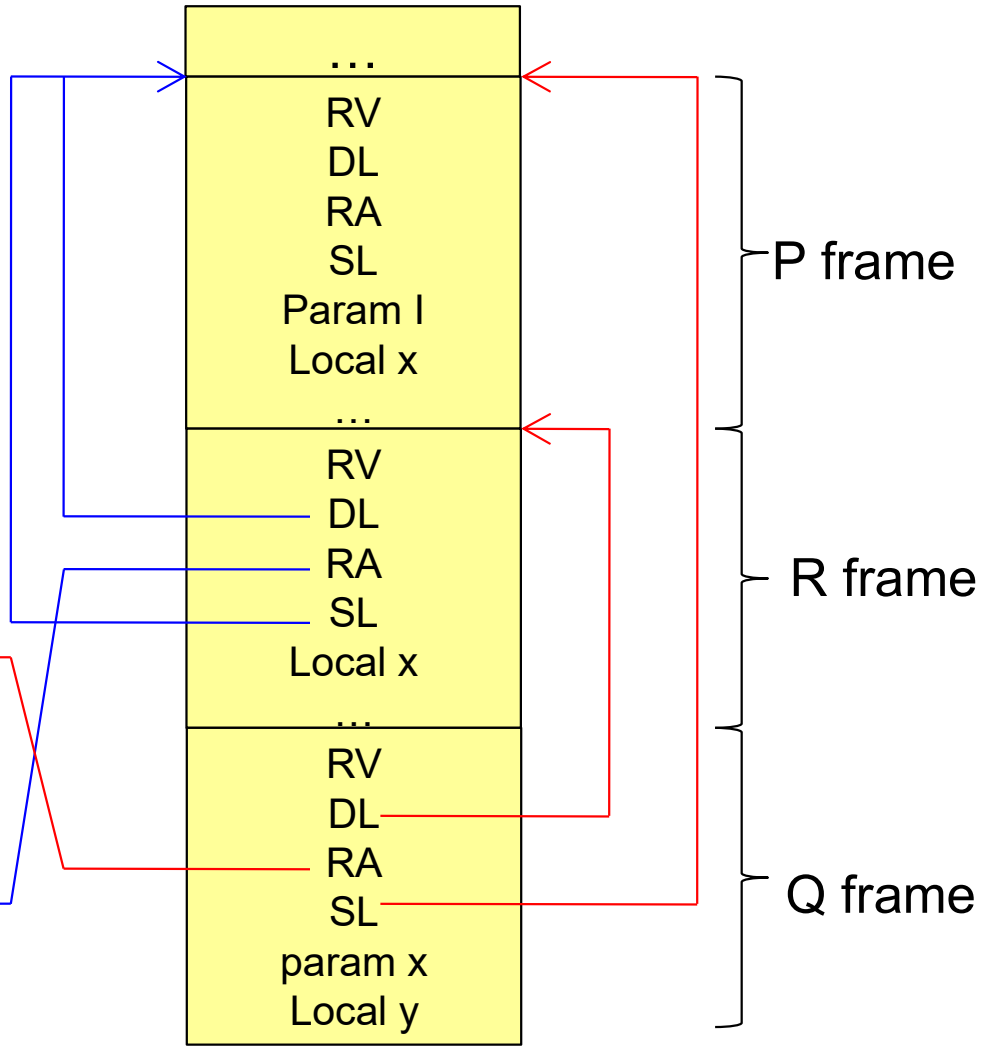
- Registers
  - PC (program counter): pointing to currently being executed instruction on Code buffer
  - B (base): pointing to the base address of data area of active block on Stack. Local varialbes are accessed via B
  - T (top): pointing to Stack's top element

Stack calculator

- Activation record / Stack frame
  - Is the memory area allocated to every function, procedure and the main program when it is activated (becoming active block)
    - Storing parameters' values
    - Storing local variables's values
    - Other information
      - Return value – RV
      - Dynamic link – DL
      - Return address – RA
      - Static link – SL
  - A function/procedure may have several Stack frames on Stack

Stack calculator

```
Procedure P(I : integer);
  Var a : integer;
  Function Q;
    Var x : char;
    Begin
          …
      Q:=…
    End;
  Procedure R(X: integer);
    Var y : char;
    Begin
      …
      y := Q+1;
      …
    End;
  Begin
    …
    Call R(1);
    …
  End;
…
```

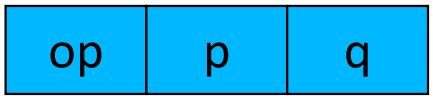


Stack calculator

- RV (return value): stores return value of a function
- DL (dynamic link): is the base address of caller's Stack frame. DL is used to recover caller's context when the callee ends.
- RA (return address): address of caller's instruction that would be executed when callee ends.
- SL (static link): base address of outer's Stack frame. SL is useful when we track non-local variables.

Instruction set of the stack calculator

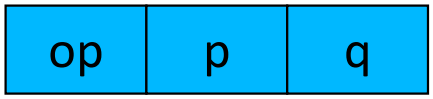
- Instruction set



LA	Load Address	t:=t+1; s[t]:=base(p)+q;
LV	Load Value	t:=t+1; s[t]:=s[base(p)+q];
LC	Load Constant	t:=t+1; s[t]:=q;
LI	Load Indirect	s[t]:=s[s[t]];
INT	Increment T	t:=t+q;
DCT	Decrement T	t:=t-q;

Instruction set of the stack calculator

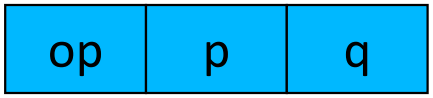
- Instruction set



J	Jump	pc:=q;
FJ	False Jump	if s[t]=0 then pc:=q; t:=t-1;
HL	Halt	Halt
ST	Store	s[s[t-1]]:=s[t]; t:=t-2;
CALL	Call	s[t+2]:=b; s[t+3]:=pc; s[t+4]:=base(p); b:=t+1; pc:=q;
EP	Exit Procedure	t:=b-1; pc:=s[b+2]; b:=s[b+1];
EF	Exit Function	t:=b; pc:=s[b+2]; b:=s[b+1];

Instruction set of the stack calculator

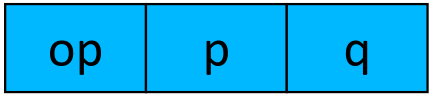
- Instruction set



RC	Read Character	read one character into s[s[t]]; t:=t-1;
RI	Read Integer	read integer to s[s[t]]; t:=t-1;
WRC	Write Character	write one character from s[t]; t:=t-1;
WRI	Write Integer	write integer from s[t]; t:=t-1;
WLN	New Line	CR & LF

Instruction set of the stack calculator

- Instruction set



AD	Add	t:=t-1; s[t]:=s[t]+s[t+1];
SB	Subtract	t:=t-1; s[t]:=s[t]-s[t+1];
ML	Multiply	t:=t-1; s[t]:=s[t]*s[t+1];
DV	Divide	t:=t-1; s[t]:=s[t]/s[t+1];
NEG	Negative	s[t]:=-s[t];
CV	Copy Top of Stack	s[t+1]:=s[t]; t:=t+1;

Instruction set of the stack calculator

- Instruction set

EQ	Equal	t:=t-1; if s[t] = s[t+1] then s[t]:=1 else s[t]:=0;
NE	Not Equal	t:=t-1; if s[t] != s[t+1] then s[t]:=1 else s[t]:=0;
GT	Greater Than	t:=t-1; if s[t] > s[t+1] then s[t]:=1 else s[t]:=0;
LT	Less Than	t:=t-1; if s[t] < s[t+1] then s[t]:=1 else s[t]:=0;
GE	Greater or Equal	t:=t-1; if s[t] >= s[t+1] then s[t]:=1 else s[t]:=0;
LE	Less or Equal	t:=t-1; if s[t] <= s[t+1] then s[t]:=1 else s[t]:=0;

Changes in symbol table

- Variable's new attributes
  - localOffset: variable's location on local frame.
  - scope
- Parameter's new attributes
  - localOffset: parameter's location on local frame.
  - scope
- Program/function/procedure's new attributes
  - codeAddress: address of first instruction on Code buffer
  - frameSize: size of corresponding Stack frame
  - paramCount: number of parameters

Variable's new attributes

- Scope
- localOffset: location in local frame (its distance to local frame's base)

```
struct VariableAttributes_ {
  Type *type;
  struct Scope_ *scope;
  int localOffset;
};
```

Parameter's new attributes

- Scope
- localOffset

```
struct ParameterAttributes_ {
  enum ParamKind kind;
  Type* type;
  struct Scope_ *scope;
  int localOffset;
};
```

Scope's new attribute

- frameSize

```
struct Scope_ {
  ObjectNode *objList;
  Object *owner;
  struct Scope_ *outer;
  int frameSize;
};
```

Function's new attributes

- codeAddress
- paramCount

```
struct FunctionAttributes_ {
   struct ObjectNode_ *paramList;
   Type* returnType;
   struct Scope_ *scope;

   int paramCount;
   CodeAddress codeAddress;
};
```

Procedure's new attributes

- codeAddress
- paramCount

```
struct ProcedureAttributes_ {
  struct ObjectNode_ *paramList;
  struct Scope_* scope;

  int paramCount;
  CodeAddress codeAddress;
};
```

Procedure's new attributes

- Program's new attribute
  - codeAddress

```
struct ProgramAttributes_ {
  struct Scope_ *scope;
  CodeAddress codeAddress;
};
```

Code generation for a program without array and subroutine

- kplrun utility
- Overview of instructions.*, codegen.*
- Generate code for (no subprogram/array)
  - ASSIGN (substitute) statement
  - IF statement
  - WHILE statement
  - FOR statement
  - CONDITION
  - EXPRESSION

kplrun

- Interpreter for Stack calculator. Syntax:

```
$ kplrun <source> [-s=stack-size] [-c=code-size] [-debug] [-dump]
```

- Options:
  - –s: define Stack size
  - –c: define maximum size of source program
  - –dump: output generatated instruction code to standard output
  - –debug: debugging mode

kplrun

- Options in debugging mode
  - a: corresponding absolute address of a Stack location (level, offset)
  - v: value stored in a Stack location (level, offset)
  - t: value stored in Stack's top
  - c: exit debugging mode

Components of incompleted code generator

#	Tên tệp	Nhiệm vụ
1	Makefile	Manage the project
2	scanner.c, scanner.h	Lexical analyzer
3	reader.h, reader.c	Read the source code by each character
4	charcode.h, charcode.c	Classify characters
5	token.h, token.c	Token declaration and recognition
6	error.h, error.c	Error handle
7	parser.c, parser.h	Syntax analyze
8	debug.c, debug.h	Print results
9	symtab.c symtab.h	Symbol Table constructing
10	semantics.c. semantics.h	Declarations and functions in semantic analysis
11	instruction.c, instruction.h	Code block management
12	codegen.c, codegen.h	Code generator
13	main.c	Main function

instructions.c

```
enum OpCode {
  OP_LA,    // Load Address:
  OP_LV,    // Load Value:
  OP_LC,    // load Constant
  OP_LI,    // Load Indirect
  OP_INT,   // Increment t
  OP_DCT,   // Decrement t
  OP_J,     // Jump
  OP_FJ,    // False Jump
  OP_HL,    // Halt
  OP_ST,    // Store
  OP_CALL,  // Call
  OP_EP,    // Exit Procedure
  OP_EF,    // Exit Function
```

```
  OP_RC,    // Read Char
  OP_RI,    // Read Integer
  OP_WRC,   // Write Char
  OP_WRI,   // Write Int
  OP_WLN,   // WriteLN
  OP_AD,    // Add
  OP_SB,    // Substract
  OP_ML,    // Multiple
  OP_DV,    // Divide
  OP_NEG,   // Negative
  OP_CV,    // Copy Top
  OP_EQ,    // Equal
  OP_NE,    // Not Equal
  OP_GT,    // Greater
  OP_LT,    // Less
  OP_GE,    // Greater or Equal
  OP_LE,    // Less or Equal

  OP_BP     // Break point.
};
```

Instructions.c

```
struct Instruction_ {
  enum OpCode op;
  WORD p;
  WORD q;
};


struct CodeBlock_ {
  Instruction* code;
  int codeSize;
  int maxSize;
};
```

```
CodeBlock* createCodeBlock(int maxSize);
void freeCodeBlock(CodeBlock* codeBlock);
void printInstruction(Instruction* instruction);
void printCodeBlock(CodeBlock* codeBlock);

void loadCode(CodeBlock* codeBlock, FILE* f);
void saveCode(CodeBlock* codeBlock, FILE* f);

int emitLA(CodeBlock* codeBlock, WORD p, WORD q);
int emitLV(CodeBlock* codeBlock, WORD p, WORD q);
int emitLC(CodeBlock* codeBlock, WORD q);
….
int emitLT(CodeBlock* codeBlock);
int emitGE(CodeBlock* codeBlock);
int emitLE(CodeBlock* codeBlock);

int emitBP(CodeBlock* codeBlock);
```

codegen.c

```
void initCodeBuffer(void);
void printCodeBuffer(void);
void cleanCodeBuffer(void);
int serialize(char* fileName);

int genLA(int level, int offset);
int genLV(int level, int offset);
int genLC(WORD constant);
….
int genLT(void);
int emitGE(void);
int emitLE(void);
```

Generate code for ASSIGN statement

v := exp

```
<code of l-value v>  // load address of v
<code of exp>        // load value of of exp
ST
```

Syntax of assignment (simplìied)

$S \rightarrow id := E$

$E \rightarrow - E_2 \mid +E_2 \mid E_2$

$E_2 \rightarrow TE_3$

$E_3 \rightarrow +TE_3 \mid -TE3 \mid \varepsilon$

$T \rightarrow FT_2$

$T_2 \rightarrow {}^*FT_2 \mid /FT_2 \mid \varepsilon$

$F \rightarrow id \mid num \mid (E)$

(Trường hợp F là biến có chỉ số hoặc lời gọi hàm xét sau)

Lvalue

```
case OBJ_VARIABLE:
    genVariableAddress(var);
    if (var->varAttrs->type->typeClass ==
TP_ARRAY)
    {varType = compileIndexes
    (var->varAttrs->type);}
    else
      varType = var->varAttrs->type;
    break;
```

Expression3

```
switch (lookAhead->tokenType)
{   case SB_PLUS:
      eat(SB_PLUS);
      checkIntType(argType1);
      argType2 =
      compileTerm();
      checkIntType(argType2);
      genAD();
      resultType =
  compileExpression3(argType1);
      break;
```

```
case SB_MINUS:
      eat(SB_MINUS);
      checkIntType(argType1);
      argType2 = compileTerm();
      checkIntType(argType2);
      genSB();
      resultType =

compileExpression3(argType1);
      break;
```

Term2

```
switch (lookAhead->tokenType)
{
  case SB_TIMES:
    eat(SB_TIMES);
    checkIntType(argType1);
    argType2 = compileFactor();
    checkIntType(argType2);
    genML();
    resultType =
    compileTerm2(argType1);
    break;
```

```
case SB_SLASH:
    eat(SB_SLASH);
    checkIntType(argType1);
    argType2 =
compileFactor();
    checkIntType(argType2);
    genDV();
    resultType =
    compileTerm2(argType1);
    break;
}
```

Generate code for IF statement

IF <cond> THEN <statement>

```
  <code of cond>      // load value of condition
  FJ L
  <code of statement>
L:
  …
```

IF <cond> THEN <statement1>ELSE <statement2>

```
  <code of cond>      // load value of condition
  FJ L1
  <code of st1>
  J L2
L1:
  <code of st2>
L2:
  …
```

Generate code for WHILE statement

WHILE <cond> DO <statement>

```
L1:
   <code of cond> // load value of condition
   FJ L2
   <code of statement>
   J L1
L2:
   …
```

Generate code for FOR statement

FOR v := <exp1> TO <exp2> DO <statement>

```
  <code of l-value v>
  CV   // copy top of stack – duplicate address of v
  <code of exp1>
  ST   // store original value of v
L1:
  CV
  LI   // get value of v
  <code of exp2>
  LE
  FJ L2
  <code of statement>
  CV;CV;LI;LC 1;AD;ST;  // increase v's value by 1
  J L1
L2:
  DCT 1
  …
```

Target code of a simple example

```
Program Example5;

Var j : Integer;
    i : Integer;

Begin
  for i := 1 to 2 do
    j := 1;
End.
```

```
0:  J 1
1:  INT 6
2:  LA 0,5
3:  CV
4:  LC 1
5:  ST
6:  CV
7:  LI
8:  LC 2
9:  LE
10:  FJ 23
11:  LA 0,4
12:  LC 1
13:  ST
14:  CV
15:  CV
16:  LI
17:  LC 1
18:  AD
19:  ST
20:  CV
21:  LI
22:  J 8
23:  DCT 1
24:  HL
```

Assignments

- Complete following function in *codegen.c*
  - `genVariableAddress(Object* var)`

    // push address of a variable to Stack's top
  - `genVariableValue(Object* var)`

    // push value of a variable to Stack's top

Note: non-local variable temporarily exclusive

Assignments (continued)

- Complete following functions in *parser.c*
  - Generate code for a variable l-value
  - Generate code for statements: Assign, If, While, For
  - Generate code for Condition
  - Generate code for Expression

Codegen for programs with arrays and subroutines

- Generate code for variable's address/value (non-local inclusive)
- Generate code for parameter's address/value (non-local inclusive)
- Generate code for address of function's return value
- Generate code for calling function/procedure
  - Generate code for arguments
- Treatment of array

Generate code for variable address

- When generate code for a variable's address/value, pay attention to its scope
  - Local variable: track in active Stack frame
  - Non-local variable: track static links and depth of tracking equals depth from current scope to variable's scope

```
computeNestedLevel(Scope* scope)
```

Generate code for PARAMETER's address

- Case: when `LValue` is a parameter
- Similar to variables, pay attention to its scope
- Call by value: push to top of Stack parameter's address.
- Call by reference: push to top of Stack parameter's value

Generate code for PARAMETER's value

- Case: when compute value of **`Factor`**
- As variable, pay attention to its scope
- Call by value: push to Stack parameter's value
- Call by reference: push to Stack the value located at the address which is parameter's value

Code generated for RV of a function

- Offset = 0 in a specified stack frame
- Level = depth from current scope to function's scope

Generate code for a function/procedure call

- Case:
  - Calling a function: when generate code for `factor`
  - Calling a procedure: when generate code for `CallSt` statement.
- Preparation: identify values of parameters
  - Increase value of T by 4 (omit RV, DL, RA, SL)
  - Generate code for k arguments
  - Decrease value of T by 4 + k
  - Generate code for CALL statement

Instruction CALL(p,q)

```
CALL (p, q)      s[t+2]:=b;          // store dynamic link
                 s[t+3]:=pc;         // store return address
                 s[t+4]:=base(p);    // store static link
                 b:=t+1;             // new base, new return value address
                 pc:=q;              // jump to new instruction
```

CALL (p, q) to a function/procedure A require 2 parameters

p: Depth of CALL statement

= depth of A's outer

= depth from current scope to scope of A's outer

p tells A's static link

q: Address of new instruction code

Operations when CALL(p,q) is executed.

1. pc changes to **`codeAddress`** (beginning address) of called sub-program`/* pc = p */`
2. Increase pc by 1 `/* pc ++ */`
3. First code instruction would be Jump instruction J to omit code instruction of local declaration in code buffer.
4. Next statement would be **`INT`** to increase **`T`** exactly by size of frame to omit Stack area corresponding to local parameters and variables.

Operations when CALL(p,q) is executed.

5. Execute next instructions and Stack would changes correspondingly.
6. Ending
    1. A procedure (instruction `EP`): release active frame and set `T` to previous frame's top.
    2. Function (lệnh `EF`): release active frame, except return value at `offset 0`, set `T` to `offset 0`.

Address of array elements

- An array that is declared like

  `A : array(.n1.) of … of array(.nk.) of integer/char`

  would occupies `n1 * …* nk` word in Stack frame
- Element `A(.i1.)…(.ik.)` is located at address

  = `A + (i1 – 1)* n2 *…* nk`

  `+ (i2 – 1)* n3 *…* nk`

  `…`

  `+ (ik-1 – 1)*nk`

  `+ (ik – 1)`
- This address is accumulated when compiling indexes

Assignments

- Complete functions in ***codegen.c***

```
int computeNestedLevel(Scope* scope);
void genVariableAddress(Object* var)
void genVariableValue(Object* var)
void genParameterAddress(Object* param)
void genParameterValue(Object* param)
void genReturnValueAddress(Object* func)
void genReturnValueValue(Object* func)
void genProcedureCall(Object* proc)
void genFunctionCall(Object* func)
```

Assignments

- Make changes to ***parser.c***

```
Type* compileLValue(void);
void compileCallSt(void);
Type* compileFactor(void);
Type* compileIndexes(Type* arrayType);
```

Assignments

- Implement following function in *symtab.c*

```
int sizeOfType(Type* type);
void declareObject(Object* obj);
```

- Note: for simplicity, each integer/char occupies one word (4 bytes) in Stack
- Order of words in a local frame is as following:
  - 0: RV
  - 1: DL
  - 2: RA
  - 3: SL
  - 4 $\rightarrow$ (4+k): for k parameters
  - (4+k+1) $\rightarrow$ (4+k+n): for local variables