Thực hành
CHƯƠNG TRÌNH DỊCH

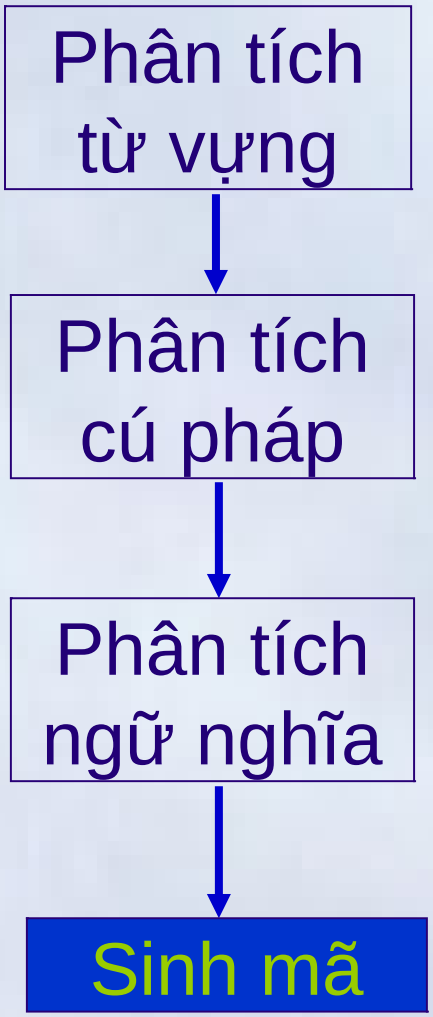
Bài 5: Sinh mã

Phạm Đăng Hải
haipd@soict.hut.edu.vn

Các bài thực hành

1. Xây dựng bảng ký hiệu
   - Giới thiệu máy ngăn xếp
   - Vấn đề xây dựng bảng ký hiệu
2. Sinh mã cho các câu lệnh
   - Giới thiệu bộ thông dịch KPLrun
   - Sinh mã cho các câu lệnh gán, rẽ nhánh, lặp..
3. Sinh mã lấy địa chỉ/giá trị
   - Lấy địa chỉ/giá trị của biến, của phần tử mảng của tham số hình thức
   - Sinh mã lấy địa chỉ của giá trị trả về của hàm
   - Sinh mã gọi thủ tục
     - Sinh mã tham số thực tế

Sinh mã đích



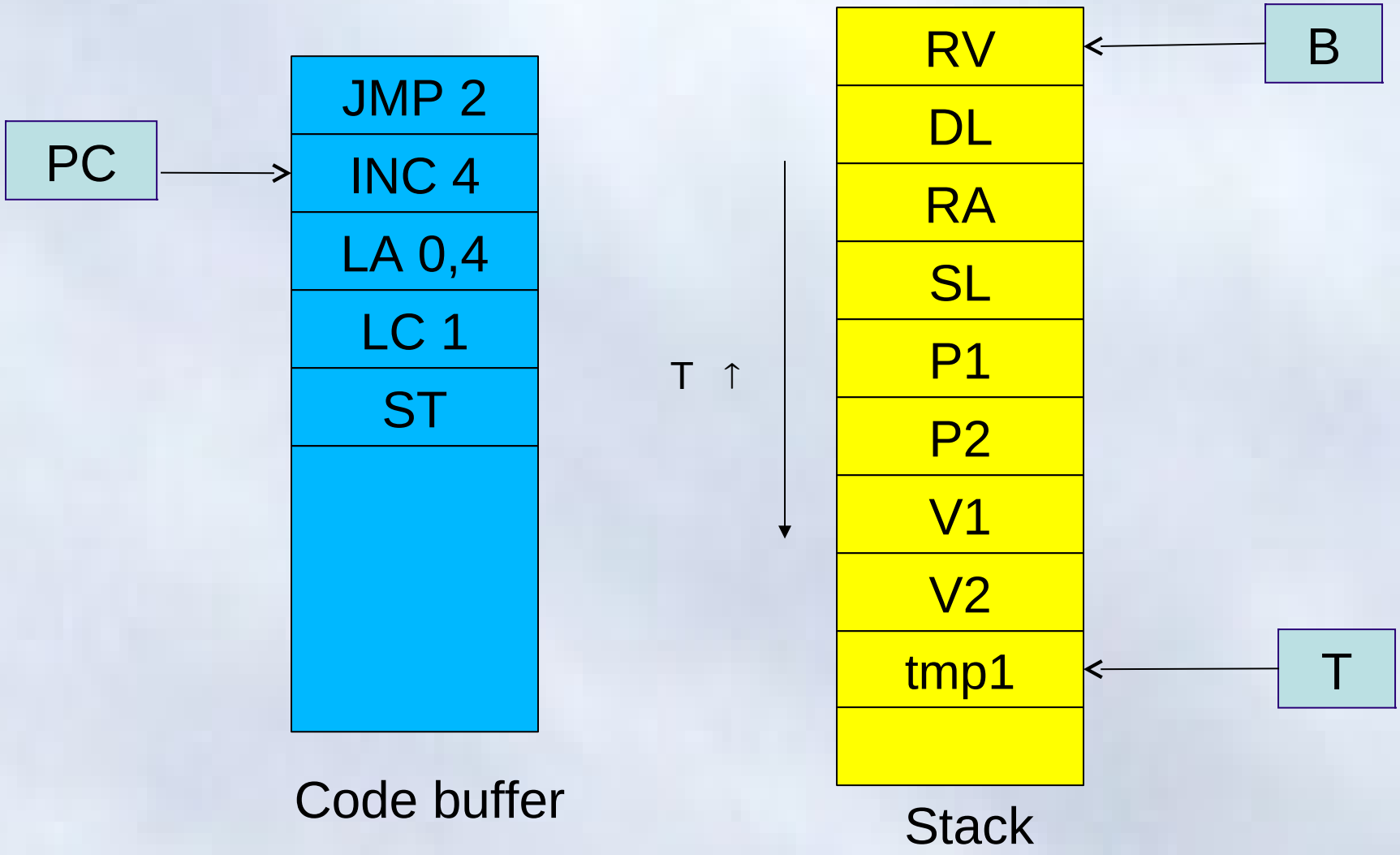
- Sinh mã là công đoạn biến đổi từ cấu trúc ngữ pháp của chương trình thành chuỗi các lệnh thực thi được của máy đích
- Cấu trúc ngữ pháp được quyết định bởi bộ phân tích cú pháp
- Các lệnh của máy đích được đặc tả bởi kiến trúc thực thi của máy đích
  - KPL sử dụng kiến trúc máy ngăn xếp

Máy ngăn xếp

- Máy ngăn xếp là một hệ thống tính toán
  - Sử dụng ngăn xếp để lưu trữ các kết quả trung gian của quá trình tính toán
  - Kiến trúc đơn giản
  - Bộ lệnh đơn giản
- Máy ngăn xếp có hai vùng bộ nhớ chính
  - **Khối lệnh:**
    - Chứa mã thực thi của chương trình
  - **Ngăn xếp:**
    - Lưu trữ các kết quả trung gian

Máy ngăn xếp

PC, B, T là các thanh ghi của máy



Máy ngăn xếp → Thanh ghi

- **PC (program counter):**
  - Con trỏ lệnh trỏ tới lệnh hiện tại đang thực thi trên bộ đệm chương trình
- **B (base):**
  - Con trỏ trỏ tới địa chỉ cơ sở của **vùng nhớ cục bộ**. Các biến cục bộ được truy xuất gián tiếp qua con trỏ này
- **T (top);**
  - Con trỏ, trỏ tới đỉnh của ngăn xếp

Máy ngăn xếp → Bản hoạt động (stack frame)

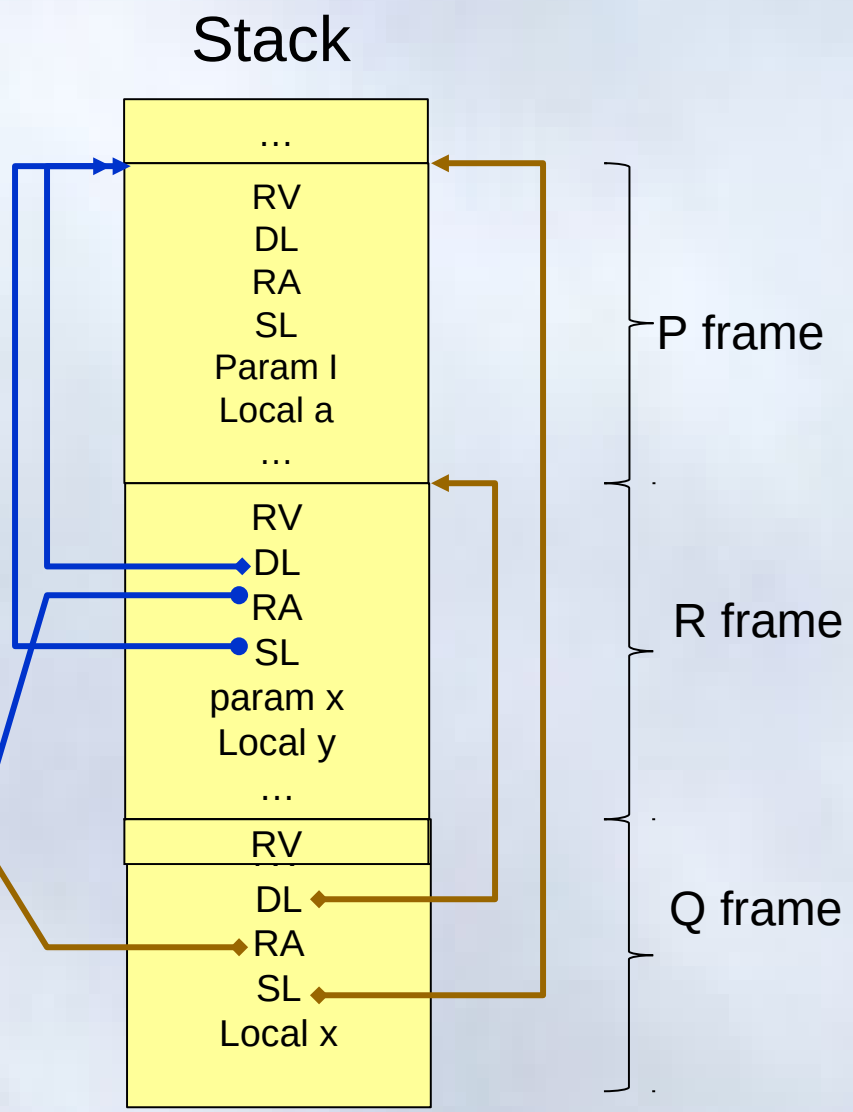
- Không gian nhớ cấp phát cho mỗi chương trình con (*hàm/thủ tục/chương trình chính*) khi chúng được kích hoạt
  - Lưu giá trị tham số
  - Lưu giá trị biến cục bộ
  - Lưu các thông tin quan trọng khác:
    - RV, DL, RA, SL
- Một chương trình con có thể có nhiều bản hoạt động

Máy ngăn xếp → Bản hoạt động (stack frame)

- RV (*return value*):
  - Lưu trữ giá trị trả về cho mỗi hàm
- DL (*dynamic link*):
  - Địa chỉ cơ sở của bản hoạt động của chương trình con gọi tới nó (caller).
  - Được sử dụng để hồi phục ngữ cảnh của chương trình gọi (caller) khi chương trình được gọi (called) kết thúc
- RA (*return address*):
  - Địa chỉ lệnh quay về khi kết thúc chương trình con
  - Sử dụng để tìm tới lệnh tiếp theo của caller khi called kết thúc
- SL (*static link*):
  - Địa chỉ cơ sở của bản hoạt động của chương trình con bao ngoài
  - Sử dụng để truy nhập các biến phi cục bộ

Máy ngăn xếp → Bản hoạt động → Ví dụ

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

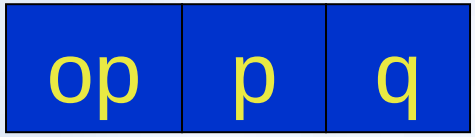


Máy ngăn xếp → Lệnh

- Lệnh máy có dạng : **Op p q**
  - Op : Mã lệnh
  - p, q : Các toán hạng.
    - Các toán hạng có thể tồn tại đầy đủ, có thể chỉ có 1 toán hạng, có thể không tồn tại
    - Ví dụ

```
J 1        % Nhảy đến địa chỉ 1
LA 0, 4    % Nạp địa chỉ từ số 0+4 lên đỉnh stack
HT         %Kết thúc chương trình
```

Máy ngăn xếp → Bộ lệnh (1/5)



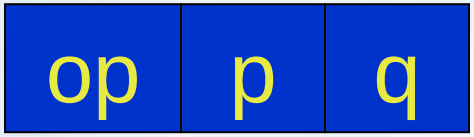
LA	Load Address	t:=t+1; s[t]:=base(p)+q;
LV	Load Value	t:=t+1; s[t]:=s[base(p)+q];
LC	Load Constant	t:=t+1; s[t]:=q;
LI	Load Indirect	s[t]:=s[s[t]];
INT	Increment T	t:=t+q;
DCT	Decrement T	t:=t-q;

Máy ngăn xếp → Bộ lệnh (2/5)

op	p	q

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];

Máy ngăn xếp → Bộ lệnh (3/5)



RC	Read Character	t=t+1; read one character into s[t];
RI	Read Integer	t=t+1; read integer to s[t];
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

Máy ngăn xếp → Bộ lệnh (4/5)

op	p	q

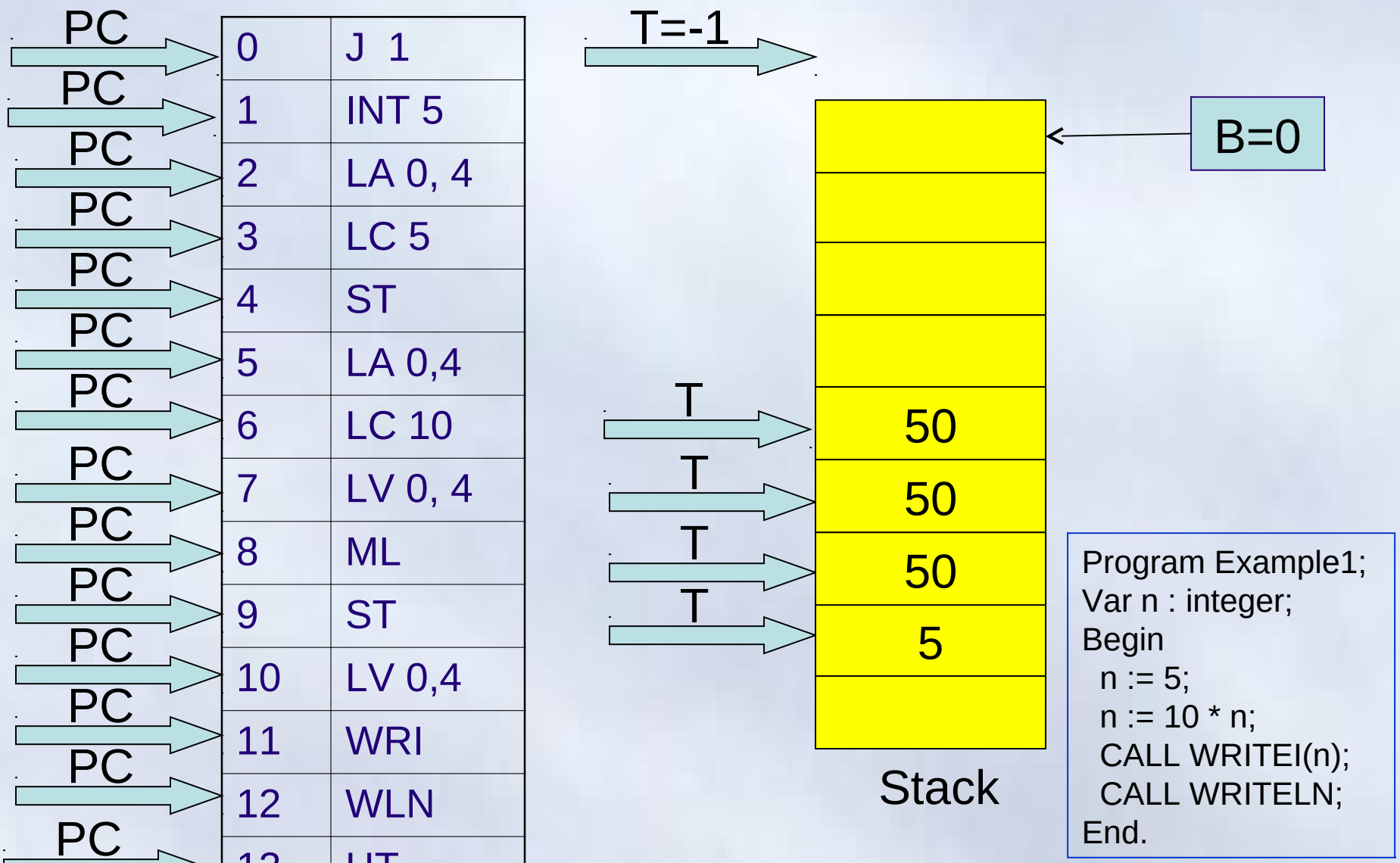
AD	Add	t:=t-1; s[t]:=s[t]+s[t+1];
SB	Subtract	t:=t-1; s[t]:=s[t]-s[t+1];
ML	Multiply	t:=t-1; s[t]:=s[t]*s[t+1];
DV	Divide	t:=t-1; s[t]:=s[t]/s[t+1];
NEG	Negative	s[t]:=-s[t];
CV	Copy Top of Stack	s[t+1]:=s[t]; t:=t+1;

Máy ngăn xếp → Bộ lệnh (5/5)

op	p	q

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;

Máy ngăn xếp → Ví dụ



Xây dựng máy ngăn xếp

Mã lệnh máy

```
typedef enum {
  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_ADD,   // Add
  OP_SUB,   // Substract
  OP_MUL,   // Multiple
  OP_DIV,   // 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.
} OpCode ;
```

Xây dựng máy ngăn xếp

Cấu trúc một lệnh

```
typedef struct{
        OpCode Op;
        int p;
        int q;
} Instruction;
```

Cấu trúc đoạn mã lệnh

```
Instruction Code [MAX_CODE];
```

Cấu trúc đoạn dữ liệu (stack)

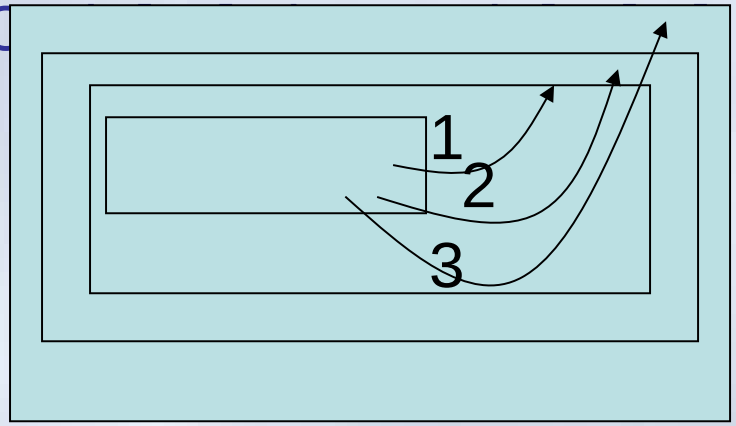
```
int Stack[MAX_DATA];
```

Xây dựng máy ngăn xếp

Hàm base(int L) dùng để tính địa chỉ cơ sở của chương trình con bao bên ngoài L mức

```
void intepreter(){
  int pc = 0, t = -1, b = 0;
  do {
     switch (Code[pc].Op) {
          case OP_LA:   t ++;
                    Stack[t] = base(Code[pc].p) + Code[pc].q;

                    break;
          case OP_LV:   t ++;
                    Stack[t] = Stack[base(C
                    break;
          case OP_INT:
                   t += Code[pc].q;
                   break;
          case OP_DCT: …
```



Xây dựng máy ngăn xếp

```
        ……..
        case OP_CALL:
                Stack[t+2] = b;                  // Dynamic Link
                Stack[t+3] = pc;                 // Return Address
                Stack[t+4] = base(Code[pc].p);  // Static Link
                b = t + 1;                       // Base & Result
                pc = Code[pc].q - 1;
                break;
        case OP_J:
                pc = Code[pc].q - 1;
                break;
        } //switch
        pc++;
    }while(Exit == 0);
}//intepreter
```

```
int base(int L) {
  int c = b;
  while (L > 0) {
    c = Stack[c + 3];
    L --;
  }
  return c;
}
```

Xây dựng bảng ký hiệu

- Bổ sung thông tin cho biến
  - Vị trí trên frame
- Bổ sung thông tin cho tham số
  - Vị trí trên frame
  - Phạm vi (bỏ thuộc tính *struct Object_ *function*)
- Bổ sung thông tin cho phạm vi
  - Kích thước của phạm vi
- Bổ sung thông tin cho hàm/thủ tục/chương trình
  - Địa chỉ bắt đầu
  - Kích thước của frame
  - Số lượng tham số của hàm/thủ tục

Bổ sung thông tin cho biến

- Vị trí trên frame của biến
  - Vị trí tính từ base của frame
- Phạm vi

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

Bổ sung thông tin cho tham số

- Vị trí trên frame của tham số
  - Vị trí tính từ base của frame
- Phạm vi

```
struct ParameterAttributes_ {
  enum ParamKind kind;
  Type* type;

  struct Scope_ *scope;
  int localOffset;
};
```

Bổ sung thông tin cho phạm vi

- Kích thước của frame

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

Bổ sung thông tin cho thủ tục

- Vị trí (địa chỉ của thủ tục)
- Số lượng tham số

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

   int paramCount;
   CodeAddress codeAddress;
};
```

Bổ sung thông tin cho hàm

- Vị trí (địa chỉ của hàm)
- Số lượng tham số

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

   int paramCount;
   CodeAddress codeAddress;
};
```

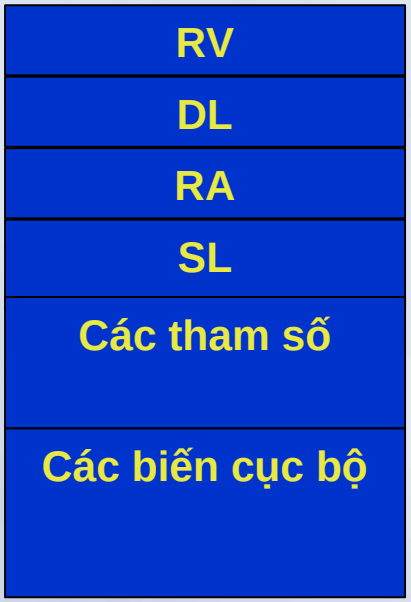
Bổ sung thông tin cho chương trình

- Vị trí (địa chỉ bắt đầu của chương trình)

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

Nhiệm vụ

- Viết các hàm sau trên symtab.c
  - int sizeOfType(Type* type);
  - void declareObject(Object* obj);
- Lưu ý:
  - Để đơn giản hóa, mỗi giá trị interger/char đều chiếm một từ (4 bytes) trên ngăn xếp.
  - Thứ tự các từ trên 1 frame như sau
    - 0: RV
    - 1: DL
    - 2: RA
    - 3: SL
    - 4 → (4+k-1): k tham số
    - (4+k) → (4+k+n-1): Nếu có n biến cục bộ



int sizeOfType(type* t)

- Trả về số ngăn nhớ trên stack mà một biến thuộc kiểu của tham số truyền vào sẽ chiếm.
  - Nếu kiểu của t là TP_INT/TP_CHAR
    - return INT_SIZE/CHAR_SIZE
      - *Theo quy ước, kiểu integer/char đều chiếm 1 từ trên stack*
  - Nếu kiểu của t là TP_ARRAY
    - return arraySize * sizeOfType(elementType)

void declareObject(Object* obj)

- Nếu là đối tượng toàn cục (currentScope=NULL)
  - Đưa vào symtab->GlobalObjectList
- Đối tượng khác:
  - Đưa vào symtab->currentScope->objList
  - Xét các trường hợp khác nhau của đối tượng được khai báo
    - Variable
    - Hàm
    - Thủ tục
    - Tham số

void declareObject(Object* obj)

- Đối tượng khác là **Variable**:
  - Cập nhật scope = currentScope
  - Cập nhật

    localOffset = currentScope -> frameSize
  - Tăng kích thước frameSize
    - frameSize += sizeOfType(Obj->varAttrs->type)

void declareObject(Object* obj)

- Đối tượng cục bộ là **Function**
  - Cập nhật outer = currentScope
- Đối tượng cục bộ là **Procedure**
  - Cập nhật outer = currentScope

void declareObject(Object* obj)

- Đối tượng cục bộ là **Parameter**
  - Cập nhật scope = currentScope
  - Cập nhật

    localOffset = currentScope->frameSize
  - Tăng kích thước frameSize
  - Cập nhật paramList của owner
  - Tăng paramCount của owner.

Program VD:

```
Program VD at address 0
    Type VEC = Arr(10,Int)
    Var N : Int at offset 4
    Var V : Arr(10,Int) at offset 5
    Var S : Int at offset 15
    Function FSUM : Int at address 0
        Param VAR A : Int at offset 4
        Param B : Int at offset 5
        Var S : Arr(10,Int) at offset 6
        Procedure PSUM1 at address 0
            Param N : Int at offset 4
            Var V : Arr(5,Arr(10,Int)) at offset 5
            Var S : Int at offset 55
```

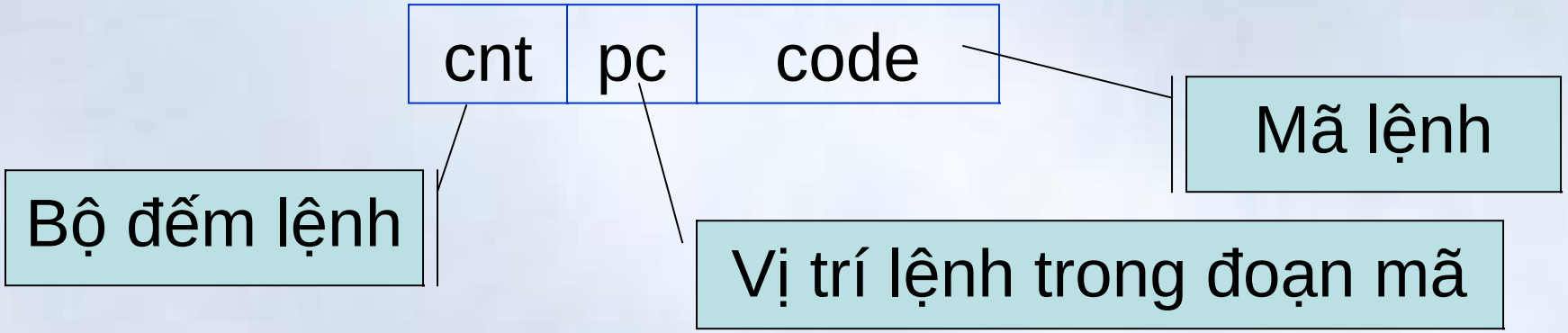
Các bài thực hành

1. Xây dựng bảng ký hiệu
   - Giới thiệu máy ngăn xếp
   - Vấn đề xây dựng bảng ký hiệu
2. Sinh mã cho các câu lệnh
   - Giới thiệu bộ thông dịch KPLrun
   - Sinh mã cho các câu lệnh gán, rẽ nhánh, lặp..
3. Sinh mã lấy địa chỉ/giá trị
   - Lấy địa chỉ/giá trị của biến, của phần tử mảng của tham số hình thức
   - Sinh mã lấy địa chỉ của giá trị trả về của hàm
   - Sinh mã gọi thủ tục
     - Sinh mã tham số thực tế

Bộ thông dịch KPLrun

- **kplrun**
  - $ ***kplrun*** <source> [-s = stack-size]
    [-c = code-size] [-debug] [-dump]
- Các tham số dòng lệnh
  - **s**: Định nghĩa kích thước stack
  - **c**: Định nghĩa kích thước tối đa của mã nguồn
  - **dump**: In mã ASM
  - **debug**: Chế độ gỡ rối

Bộ thông dịch KPLrun→Chế độ gỡ rối



Trong chế độ gỡ rối, có thể có các lệnh

- **a**: địa chỉ tuyệt đối của địa chỉ tương đối (level, offset)
- **m/M**: giá trị tại địa chỉ tương đối (level,offset)
- **t/T**: giá trị đầu ngăn xếp
- **c/C**: thoát khỏi chế độ gỡ rối
- **v/V:** Xem nội dung Stack
- « ***Enter*** » để thực hiện câu lệnh tiếp

Bộ thông dịch KPLrun →Chế độ gỡ rối →Ví dụ

```
D:\>kplrun example
50

Press any key to exit...
```

```
D:\>kplrun example -dump
0:   J 1
1:   INT 5
2:   LA 0,4
3:   LC 5
4:   ST
5:   LA 0,4
6:   LC 10
7:   LV 0,4
8:   ML
9:   ST
10:   LV 0,4
11:   WRI
12:   WLN
13:   HL
```

```
D:\>kplrun example -debug
      0-0    :   J 1

      1-1    :   INT 5

      2-2    :   LA 0,4
t
Top (5) = 4

      3-3    :   LC 5
t
Top (6) = 5

      4-4    :   ST
t
Top (4) = 5
c
50
```

KPLrun→ Instructions.h

Khai báo các mã lệnh của máy ngăn xếp

```
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.
};
```

KPLrun→ Virtual machine

```
//code: mảng các lệnh, mỗi lện gồm Op, p, q
//pc, t, b là các thanh ghi
while(code[pc].op != OP_HL) {
    switch (code[pc].op) {
      case OP_LA:  t := t + 1;  s[t] := base(p) + q; break;
      case OP_LV:  t := t + 1; s[t] := s[base(p) + q]; break;
      case OP_LC:  t := t + 1; s[t] :=q; break;
      case OP_ST:  s[s[t-1]] := s[t]; t := t -2; break;
      case OP_J:    pc := q - 1;
      case OP_FJ:  if s[t] = 0 then pc:=q-1; t := t - 1; break;
      case OP_WRI, write([t])  t := t-1; break;
      …
      } pc++
}
```

KPLrun→ Instructions.h

emitCode(CodeBlock* codeBlock, enum OpCode op, WORD p, WORD q)

//Hàm emitCode được gọi tới bởi các hàm sinh mã emitLA(), emitLT()..

```
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);
```

gencode.h

Các hàm sinh mã genLT(), gentLC() gọi tới các sinh mã emitLT(), emitLC().. Tương ứng ở trong instructionc

```
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);
```

Sinh mã cho câu lệnh gán

- **Lệnh gán** V := Exp

```
<code of Lvalue v>// đẩy địa chỉ của v lên stack
<code of exp>      // đẩy giá trị của exp lên stack
ST                 //S[S[t-1]] = S[t]
```

Sinh mã cho các câu lệnh rẽ nhánh

If <dk> Then statement

```
    <code of dk>     // đẩy giá trị điều kiện dk lên stack
    FJ  L
    <code of statement>
L:
```

If <dk> Then st1 Else st2

```
    <code of dk> // đẩy giá trị điều kiện dk lên stack
    FJ L1
    <code of st1>
    J L2
L1:
    <code of st2>
L2:
```

Sinh mã cho câu lệnh lặp While

While <dk> Do statement

```
L1:   <code of dk>
      FJ L2
      <code of statement>
      J L1
L2:
```

Sinh mã cho câu lệnh lặp For

For v := exp1 to exp2 do statement

```
    <code of l-value v>
    CV   // Sao chép địa chỉ của v lên đỉnh ngăn xếp
    <code of exp1>
    ST   // lưu giá trị đầu của v
L1:
    CV
    LI   // lấy giá trị của v
    <code of exp2>
    LE
    FJ L2
    <code of statement>
    CV;CV;LI;LC 1;AD;ST;  // Tăng v lên 1
    J      L1
L2:
    DCT 1                    //Giảm thanh ghi T 1 đơn vị
```

Nhiệm vụ

- **Điền vào codegen.c**

  //Tạm thời xem các biến đều nằm mức 0 (*trên frame hiện tại*)
- genVariableAddress(Object* var)

  // Đẩy địa chỉ một biến lên stack
  - Tính toán mức của biến (*var* → *varAttrs*→*scope*)
    - Xây dựng hàm computeNestedLevel(Scope* scope)
  - Tính toán vị trí offset của biến
  - Sinh ra lệnh **LA** (genLA(level, offset))
- genVariableValue(Object* var)

  // Đẩy giá trị một biến lên stack
  - Tính toán mức (level) của biến và offset của biến
  - Sinh ra lệnh **LV**(genLA(level, offset))

Nhiệm vụ

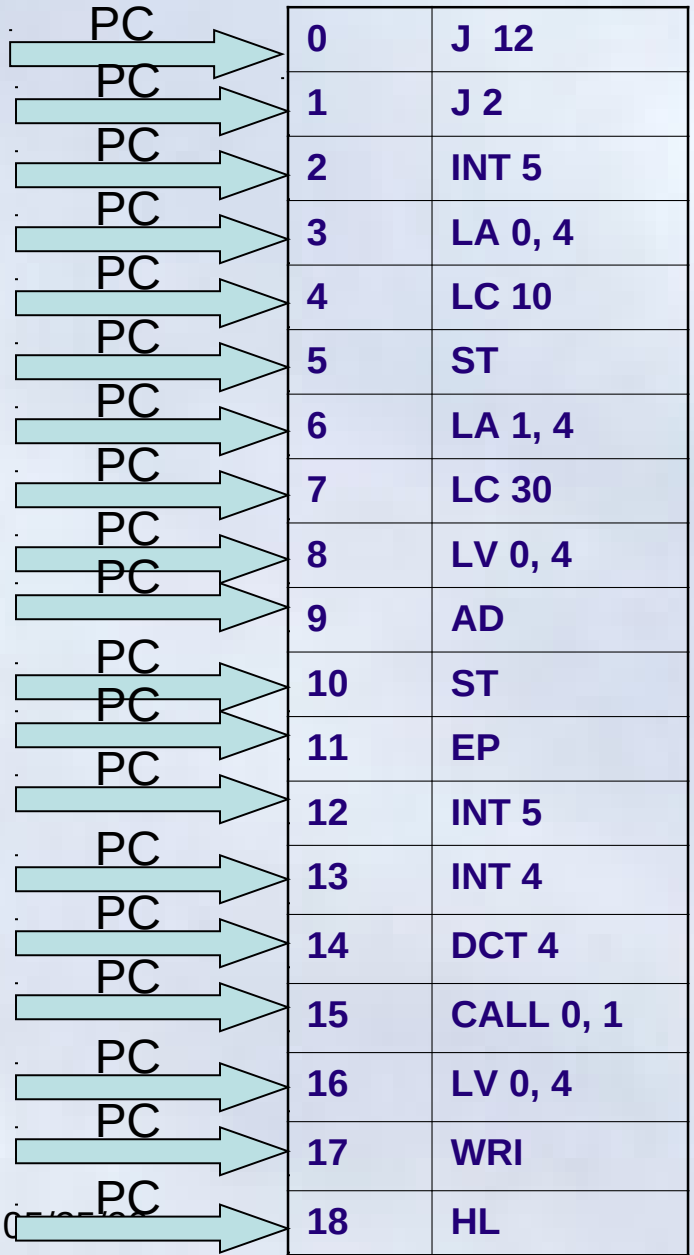
Điền vào parser.c

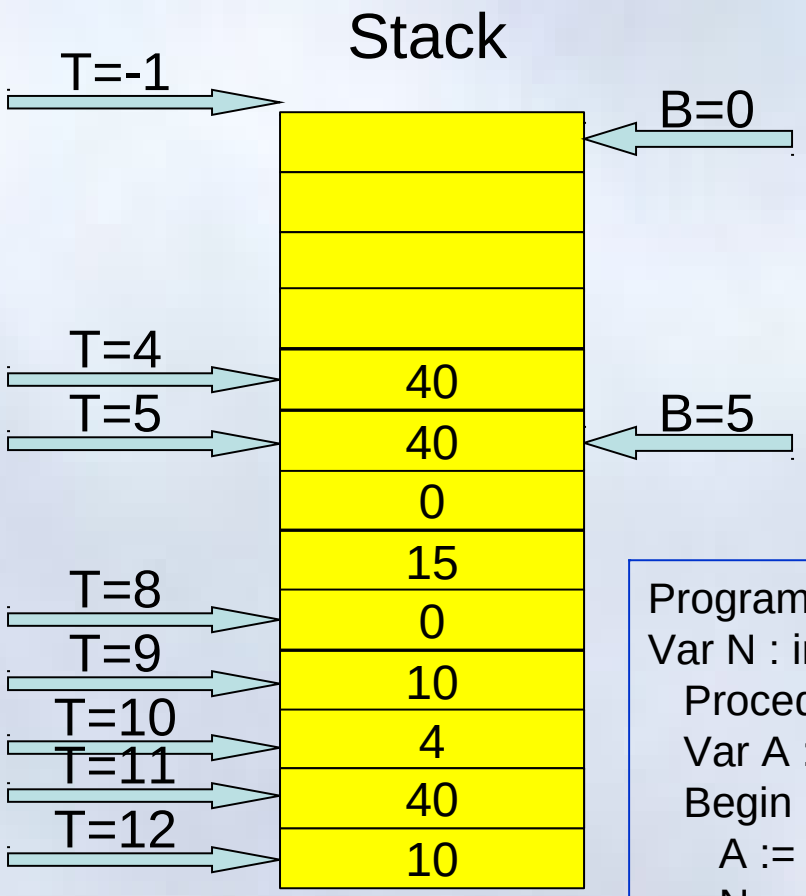
- Sinh mã L-value cho biến
  - Đặt địa chỉ của biến lên stack
- Sinh mã các câu lệnh: gán, if, while, for
  - Sinh mã cho LValue: Đặt địa chỉ lên đỉnh stack
  - Sinh mã cho biểu thức: Đặt giá trị lên dỉnh stack
  - ST: Lưu lại
- Sinh mã điều kiện
- Sinh mã biểu thức

Các bài thực hành

1. Xây dựng bảng ký hiệu
   - Giới thiệu máy ngăn xếp
   - Vấn đề xây dựng bảng ký hiệu
2. Sinh mã cho các câu lệnh
   - Giới thiệu bộ thông dịch KPLrun
   - Sinh mã cho các câu lệnh gán, rẽ nhánh, lặp..
3. Sinh mã lấy địa chỉ/giá trị
   - Lấy địa chỉ/giá trị của biến, của phần tử mảng của tham số hình thức
   - Sinh mã lấy địa chỉ của giá trị trả về của hàm
   - Sinh mã gọi thủ tục
     - Sinh mã tham số thực tế

Máy ngăn xếp → Ví dụ






```
Program Exp1;
Var N : integer;
   Procedure B;
   Var A : integer;
   Begin
      A := 10;
      N := 30 + A;
   End;
Begin
  Call B;
  CALL WRITEI(n);
End.
```

Sinh mã lấy địa chỉ/giá trị

- Lấy địa chỉ/giá trị biến
  - Có tính đến các biến bên ngoài
- Lấy địa chỉ/giá trị tham số hình thức
  - Có tính đến các biến phi cục bộ
- Lấy địa chỉ của giá trị trả về của hàm
- Sinh mã gọi hàm/thủ tục
  - Sinh mã tham số thực tế
- Lấy địa chỉ/giá trị của phần tử mảng

Lấy địa chỉ/giá trị biến

- Khi lấy địa chỉ/giá trị một biến cần tính đến phạm vi của biến
  - Biến cục bộ được lấy từ frame hiện tại
  - Biến phi cục bộ được lấy theo các StaticLink với cấp độ lấy theo "độ sâu" của phạm vi hiện tại so với phạm vi hiện thời
    - Hàm **computeNestedLevel(Scope* scope)** trả về độ sâu của biến

```
Level = 0;
Scope* tmp = symtab->currentScope;
While (tmp != scope) {tmp = tmp ->outer, level++ };
```

Lấy địa chỉ/giá trị biến

- Tìm vị trí biến trong stack
  - Level = *computeNestedLevel(Scope* scope*)
  - Offset = OFFFSET(Var)
- Lấy giá trị của biến Var
  - genLV(Level, Offset)
- Lấy địa của biến Var
  - genLA(Level, Offset)

Lấy địa chỉ của tham số hình thức

- Khi **`LValue`** là tham số, cũng cần tính độ sâu như biến
- Việc lấy giá trị, còn phụ thuộc vào đây là tham trị hay tham biến
  - Nếu là tham trị:
    - Địa chỉ /giá trị giống như với biến
  - Nếu là tham biến:
    - Địa chỉ của biến chính là địa chỉ truyền vào cho hàm/ thủ tục.

Lấy địa chỉ của tham số hình thức

- Tìm vị trí biến tham số trong stack
  - Level = *computeNestedLevel(Scope* scope*)
  - Offset = OFFFSET(Var)
- Nếu là tham trị, nạp giá trị lên stack
  - genLV(Level, Offset)
- Nếu là tham biến, nạp địa chỉ tham số:
  - genLA(Level, Offset)

Ví dụ→Truyền theo giá trị

```
Program Exp;
Var N : integer;
   Procedure Add(a:integer; b:integer);
   Begin
      N := a+b;
   End;
Begin
  N := 10;
  Call add(N,20*10);
End.
```

```
0:  J 9
1:  J 2
2:  INT 6
3:  LA 1,4
4:  LV 0,4
5:  LV 0,5
6:  AD
7:  ST
8:  EP
9:  INT 5
10:  LA 0,4
11:  LC 10
12:  ST
13:  INT 4
14:  LV 0,4
15:  LC 20
16:  LC 10
17:  ML
18:  DCT 6
19:  CALL 0,1
20:  HL
```

Ví dụ→Truyền theo biến

```
Program Example1;
Var N : integer;

   Procedure Add(Var a:integer);
   Begin
      a := 10*a;
   End;
Begin
  Call add(N);
  Call WRITEI(N);
End.
```

```
0:  J 10
1:  J 2
2:  INT 5
3:  LV 0,4
4:  LC 10
5: LV 0, 4
6: LI
7:ML
8:  ST
9:  EP
10:  INT 5
11:  INT 4
12:  LA 0,4
13:  DCT 5
14:  CALL 0,1
15:  LV 0,4
16:  WRI
17:  HL
```

Lấy địa chỉ của giá trị trả về của hàm

- Giá trị trả về luôn nằm ở offset 0 trên frame
- Chỉ cần tính độ sâu giống như với biến hay tham số hình thức

```
Program Exp;
Var N : integer;
   Function Sqrt(a:integer): integer;
   Begin
      Sqrt := A * A;
   End;
Begin
  N := Sqrt(10);
  Call WRITEI(N);
End.
```

```
0:  J 9
1:  J 2
2:  INT 5
3:  LA 0,0
4:  LV 0,4
5:  LV 0,4
6:  ML
7:  ST
8:  EF
9:  INT 5
10:  LA 0,4
11:  INT 4
12:  LC 10
13:  DCT 5
14:  CALL 0,1
15:  ST
16:  HL
```

Sinh lời gọi hàm/thủ tục

- Lời gọi
  - Hàm gặp trong sinh mã cho **factor**
  - Thủ tục gặp trong sinh mã lệnh **CallSt**
- Trước khi sinh lời gọi hàm/thủ tục cần phải nạp giá trị cho các tham số hình thức lên stack bằng cách
  - Tăng giá trị T lên 4 (bỏ qua RV,DL,RA,SL)
  - Sinh mã cho k tham số thực tế
    - Đặt các tham số(*giá trị/ địa chỉ*) lên đỉnh stack
  - Giảm giá trị T đi 4 + k → **genDCT(4+k)**
  - Sinh mã cho lệnh CALL→**genFun/ProcCall(obj)**

Lệnh CALL (p, q)

Lệnh CALL có hai tham số:

- p: Độ sâu của lệnh CALL, chứa static link.
  Base(p) = base của frame chương trình con chứa khai báo chương trình con A.
- q: Địa chỉ lệnh mới: địa chỉ đầu tiên của dãy lệnh cần thực hiện khi gọi A.

CALL (p, q)

```
s[t+2]:=b;          // Lưu lại dynamic link
s[t+3]:=pc;         // Lưu lại return address
s[t+4]:=base(p);    // Lưu lại static link
b:=t+1;       // Base mới và return value
pc:=q;              // địa chỉ lệnh mới
```

Lệnh CALL (p, q)

- Điều khiển **pc** chuyển đến địa chỉ bắt đầu của chương trình con /* pc = p */
- Lệnh đầu tiên thông thường là lệnh nhảy **J** để bỏ qua mã lệnh của các khai báo chương trình con cục bộ trên *đoạn mã*.
- Lệnh tiếp theo là lệnh **INT** tăng **T** đúng bằng kích thước *frame*
  - **Mục đích:** bỏ qua *frame* chứa vùng nhớ của các tham số và biến cục bộ.

Lệnh CALL (p, q)

- Thực hiện các lệnh và stack biến đổi tương ứng.
- Khi kết thúc
  - Thủ tục (lệnh **EP**): toàn bộ frame được giải phóng, con trỏ **T** đặt lên đỉnh frame cũ.
  - Hàm (lệnh **EF**): frame được giải phóng, chỉ chừa giá trị trả về tại offset 0, con trỏ **T** đặt lên đầu frame hiện thời (offset 0).

Sinh địa chỉ của phần tử mảng

- Biến mảng khai báo

  A : array(.n_1.) of .. of array(.n_k.) of integer/ char

  sẽ chiếm n1 * …* nk ô nhớ trên frame
- Phần tử A(.i_1.)..(.i_k.) được định vị tại địa chỉ

  = A + i_1 * n_2 *…* n_k + i_2* n_3 *…* n_k +…

  + i_{k-1} * n_k + i_k //Chỉ số bắt đầu từ 0
- Địa chỉ này được tính tích lũy theo tiến trình duyệt chỉ số

Sinh địa chỉ của phần tử mảng

```
Program Exm;
Var N : Array(.4.) of Array(.5.) of array(.6.) of integer;
Begin
    N(.3.)(.4.)(.5.) := 10;
End.
```

```
0:  J 1
1:  INT 124
2:  LA 0,4
3:  LC 3
4:  LC 30
5:  ML
6:  AD
7:  LC 4
8:  LC 6
```

```
9:  ML
10:  AD
11:  LC 5
12:  LC 1
13:  ML
14:  AD
15:  LC 10
16:  ST
17:  HL
```

Nhiệm vụ → Bổ sung vào **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)
  - Gán giá trị trả về cho một hàm
- void genReturnValueValue(Object* func)
- void genProcedureCall(Object* proc)
- void genFunctionCall(Object* func)

Nhiệm vụ→ Cập nhật **parser.c**

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