

# CS 153: Concepts of Compiler Design

## November 9 Class Meeting

---

Department of Computer Science  
San Jose State University



Fall 2017  
Instructor: Ron Mak  
[www.cs.sjsu.edu/~mak](http://www.cs.sjsu.edu/~mak)



# What Your Current Status Should Be

---

- ❑ Created a grammar for your source language.
- ❑ Generated a parser, a scanner, and parse trees using ANTLR 4.
- ❑ Incorporated the symbol table code.

# What's Left

---

- ❑ Decide what Jasmin assembly code to generate for your language's statements.
- ❑ Allow ANTLR to walk the parse tree using the visitor design pattern.
- ❑ Override the relevant default visit methods to generate Jasmin assembly code.
- ❑ Write sample programs in your source language that you can successfully compile and execute on the JVM.

# Example: Newton's Square Root Function

<sup>#0</sup>  
FUNCTION sqrt(x : real) : real;

VAR  
  <sup>#1</sup>i : integer;  
  <sup>#2</sup>root : real;

BEGIN  
  i := 0;  
  root := x;

REPEAT  
  root := (x/root + root)/2;  
  i := i + 1;  
UNTIL i > 10;

sqrt := root;

END;

```
iconst_0  
istore_1    ; i := 0  
fload_0  
fstore_2    ; root := x
```

L000:

```
fload_0    ; x  
fload_2    ; root  
fdiv       ; /  
fload_2    ; root  
fadd       ; +  
fconst_2   ; 2.0  
fdiv       ; /  
fstore_2   ; ==> root  
iinc 1 1   ; i := i + 1;
```

```
iload_1    ; i  
bipush 10  ; 10  
if_icmpgt L001 ; if i > 10 goto L001  
iconst_0   ; false  
goto L002
```

L001:

```
iconst_1   ; true
```

L002:

```
ifne L003  ; if true goto L003  
goto L000
```

L003:

```
fload_2  
freturn    ; return root
```

# Pcl2

- Pcl2, a tiny subset of Pascal, version 2.
- The **Pcl2.g4** grammar:

```
grammar Pcl2; // A tiny subset of Pascal

@header {
    import wci.intermediate.*;
    import wci.intermediate.syntabimpl.*;
}

program      : header mainBlock '.' ;
header       : PROGRAM IDENTIFIER ';' ;
mainBlock    : block;
block        : declarations compoundStmt ;

declarations : VAR declList ';' ;
declList     : decl ( ';' decl )* ;
decl         : varList ':' typeId ;
varList      : varId ( ',' varId )* ;
varId        : IDENTIFIER ;
typeId       : IDENTIFIER ;
```

# Pcl2, cont'd

```
compoundStmt : BEGIN stmtList END ;
```

Pcl2.g4

```
stmt : compoundStmt  
    | assignmentStmt  
    ;
```

```
stmtList      : stmt ( ';' stmt )* ;  
assignmentStmt : variable ':' '=' expr ;
```

```
variable : IDENTIFIER ;
```

```
expr locals [ TypeSpec type = null ]  
  : expr mulDivOp expr # mulDivExpr  
  | expr addSubOp expr # addSubExpr  
  | number              # unsignedNumberExpr  
  | signedNumber        # signedNumberExpr  
  | variable            # variableExpr  
  | '(' expr ')'        # parenExpr  
  ;
```

# Pcl2, cont'd

```
mulDivOp : MUL_OP | DIV_OP ;  
addSubOp : ADD_OP | SUB_OP ;
```

Pcl2.g4

```
signedNumber locals [ TypeSpec type = null ]  
    : sign number  
    ;  
sign : ADD_OP | SUB_OP ;
```

```
number locals [ TypeSpec type = null ]  
    : INTEGER      # integerConst  
    | FLOAT        # floatConst  
    ;
```

```
PROGRAM : 'PROGRAM' ;  
VAR      : 'VAR' ;  
BEGIN   : 'BEGIN' ;  
END      : 'END' ;
```

# Pcl2, cont'd

```
IDENTIFIER : [a-zA-Z][a-zA-Z0-9]* ;
INTEGER    : [0-9]+ ;
FLOAT      : [0-9]+ '.' [0-9]+ ;

MUL_OP : '*' ;
DIV_OP : '/' ;
ADD_OP : '+' ;
SUB_OP : '-' ;

NEWLINE : '\r'? '\n' -> skip ;
WS       : [ \t]+ -> skip ;
```

Pcl2.g4



# Sample Pcl2 Program

- ❑ Main program
- ❑ Declarations
- ❑ Compound statement
- ❑ Assignment statements only
- ❑ Arithmetic expressions

```
PROGRAM sample;  
  
VAR  
    i, j : integer;  
    alpha, beta5x : real;  
  
BEGIN  
    i := 32;  
    j := i;  
    i := -2 + 3*j;  
  
    alpha := 9.3;  
    beta5x := alpha;  
    beta5x := alpha/3.7 - alpha*2.88;  
    beta5x := 8.45*(alpha + 9.12)  
END.
```

# Code Generation in Two Tree Visits

---

- ❑ After the parser and lexer have generated the parse tree and the default visit methods, we will visit the tree twice.
- ❑ The first visit will process declarations.
  - Enter identifiers into the symbol tables (Pcl1).
  - Emit **.field** assembly directives for the program variables.
  - Emit boilerplate code for the constructor method.
  - Decorate the parse tree nodes with data type specifications.

# Code Generation in Two Tree Visits, *cont'd*

---

- The second visit will emit code for statements.
  - Code for the main program header and epilogue.
  - Code for assignment statements.
  - Code for arithmetic expressions.
- The two visits override different visit methods.
- All generated Jasmin code is written into a **.j** file named after the program name.
  - Example: **PROGRAM sample** → **sample.j**

# Code Generation in Two Tree Visits, *cont'd*

```
public class Pcl2
{
    public static void main(String[] args) throws Exception
    {
        String inputFile = null;

        if (args.length > 0) inputFile = args[0];
        InputStream is = (inputFile != null)
            ? new FileInputStream(inputFile)
            : System.in;

        ANTLRInputStream input = new ANTLRInputStream(is);
        Pcl2Lexer lexer = new Pcl2Lexer(input);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        Pcl2Parser parser = new Pcl2Parser(tokens);
        ParseTree tree = parser.program();

        Pass1Visitor pass1 = new Pass1Visitor();
        pass1.visit(tree);

        PrintWriter jFile = pass1.getAssemblyFile();

        Pass2Visitor pass2 = new Pass2Visitor(jFile);
        pass2.visit(tree);
    }
}
```

# Assignment #6

---

- ❑ Begin code generation.
- ❑ Generate Jasmin assembly object code for declarations, assignment statements, control statements, and expressions.
- ❑ You should be able to assemble and run your generated code.
  - Produce runtime printed output, including the elapsed time at the end.
- ❑ Follow the Pcl1 and Pcl2 examples.
- ❑ Formal assignment write-up soon in Canvas.

# Looping Statement Code Template

*loop-label:*

Code for statements before the test

Code to evaluate the boolean test expression

**ifne** *next-label*

Code for statements after the test

**goto** *loop-label*

*next-label:*

- The code that evaluates the boolean expression leaves either **0 (false)** or **1 (true)** on top of the operand stack.
  - **ifne** branches if [TOS] is **not 0** (the expression value is **true**)
- There might not be any code before or after the test.

# Example: FOR Statement

```
PROGRAM ForTest;
VAR
    j, k, n : integer;

BEGIN {FOR statements}
    ...

    FOR k := j TO 5 DO BEGIN
        n := k;
    END;

    ...
END.
```

Remember that **program variables** are translated into **Jasmin static fields**, and so they have names, not slot numbers.

```

                                getstatic    fortest/j I
                                putstatic    fortest/k I

L001:
                                getstatic    fortest/k I
                                iconst_5
                                if_icmpgt    L003
                                iconst_0
                                goto         L004

L003:
                                iconst_1

L004:
                                ifne        L002

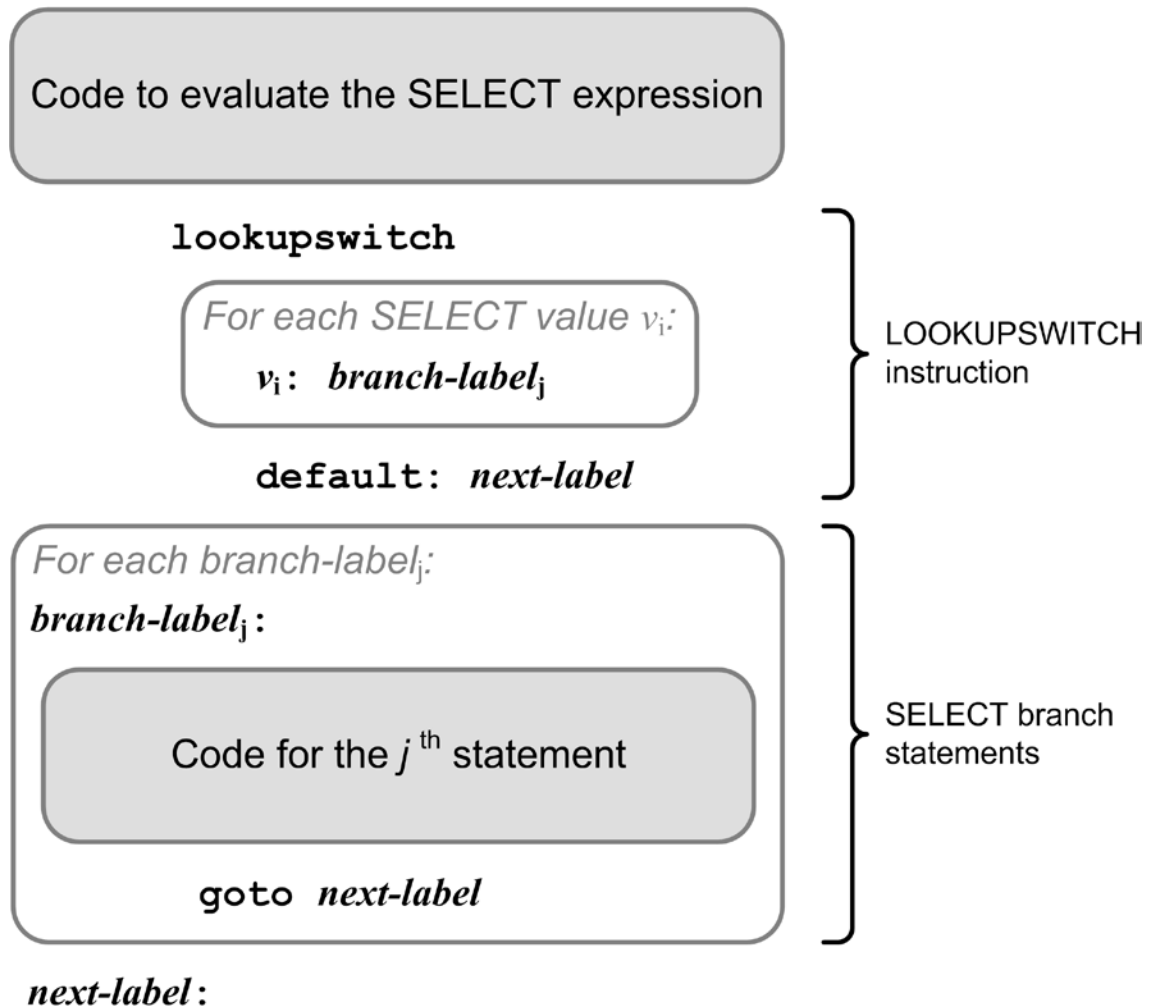
                                getstatic    fortest/k I
                                putstatic    fortest/n I

                                getstatic    fortest/k I
                                iconst_1
                                iadd
                                putstatic    fortest/k I
                                goto         L001

L002:
```

This is code emitted for a general > test. It can be much improved!

# SELECT Statement





# Example: CASE Statement

```

    VAR #0 i, #1 j : integer;
    ...
    CASE i OF
        100,105:      j := 1000;
        200,256,282: j := 2000;
    END

```

```

    iload_0 ; i

```

```

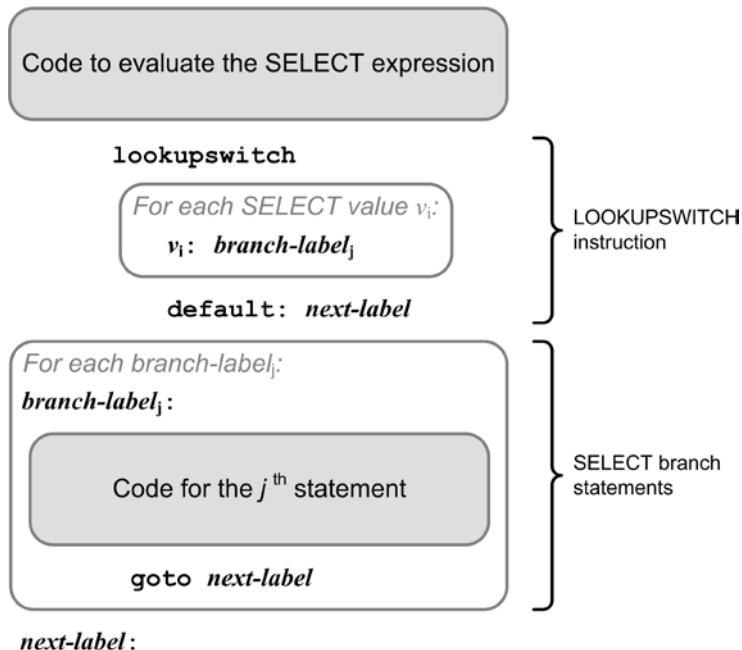
    lookupswitch
        100: L010
        105: L010
        200: L020
        256: L020
        282: L020
        default: L099

```

```

L010:
    sipush 1000
    istore_1 ; j := 1000
    goto L099
L020:
    sipush 2000
    istore_1 ; j := 2000
    goto L099
L099:

```



# Pascal Procedures and Functions

---

- ❑ Analogous to Java methods.
- ❑ Two major simplifications for our Pascal compiler:
  - Standard Pascal is not object-oriented.
  - Therefore, Pascal procedures and functions are more like the private static methods of a Java class.
- ❑ Java does not have nested methods.
  - The JVM does not easily implement nested methods.
  - Therefore, we will compile only “top level” (level 1) Pascal procedures and functions.

# Procedures and Functions, *cont'd*

## □ A Pascal program:

```
PROGRAM ADDER;

VAR
  i, j, sum : integer;

FUNCTION add(n1, n2 : integer) : integer;

  VAR
    s : integer;

  BEGIN
    s := i + j + n1 + n2;
    add := s;
  END;

BEGIN
  i := 10;
  j := 20;

  sum := add(100, 200);
  writeln('Sum = ', sum)
END.
```

## □ The roughly equivalent Java class:

- Fields and methods are **private static**.

```
public class Adder
{
    private static int i, j, sum;

    private static int add(int n1, int n2)
    {
        int s = i + j + n1 + n2;
        return s;
    }

    public static void main(String args[])
    {
        i = 10;
        j = 20;

        sum = add(100, 200);
        System.out.println("Sum = " + sum);
    }
}
```

# Code for a Pascal Main Program

```
PROGRAM Adder;
```

```
VAR
```

```
  i, j, sum : integer;
```

```
FUNCTION add(n1, n2 : integer) : integer;
```

```
  VAR
```

```
    s : integer;
```

```
  BEGIN
```

```
    s := i + j + n1 + n2;
```

```
    add := s;
```

```
  END;
```

```
BEGIN
```

```
  i := 10;
```

```
  j := 20;
```

```
  sum := add(100, 200);
```

```
  writeln('Sum = ', sum)
```

```
END.
```

```
.class public super Adder
```

```
.super java/lang/Object
```

```
.private field static i I
```

```
.private field static j I
```

```
.private field static sum I
```

Private static  
class fields.

```
.method public <init>()V
```

Void method.  
No parameters.

```
  aload_0
```

```
  invokenonvirtual java/lang/Object/<init>()V
```

```
  return
```

```
.limit stack 1
```

```
.limit locals 1
```

```
.end method
```

```
...
```



Each Jasmin class must have a constructor named **<init>**.

- The local variable in **slot #0** contains the value of “**this**”.
- Each constructor must call the superclass constructor.

# Code for a Pascal Function (Static Method)

```
PROGRAM ADDER;
```

```
VAR  
  i, j, sum : integer;
```

```
FUNCTION add(n1, n2 : integer) : integer;
```

```
  VAR  
    s : integer;
```

```
  BEGIN  
    s := i + j + n1 + n2;  
    add := s;  
  END;
```

```
BEGIN  
  i := 10;  
  j := 20;  
  
  sum := add(100, 200);  
  writeln('Sum = ', sum)  
END.
```

```
.method static add(II)I
```

```
  getstatic Adder/i I  
  getstatic Adder/j I  
  iadd  
  iload_0      ; n1 (slot #0)  
  iadd  
  iload_1      ; n2 (slot #1)  
  iadd  
  istore_2     ; s  (slot #2)  
  
  iload_2  
  ireturn     ; s
```

```
.limit stack 2  
.limit locals 3  
.end method
```

- Use **getstatic** with a fully qualified name and type to push the value of a static field onto the operand stack.

# Code to Call a Function (Static Method)

```
PROGRAM ADDER;
```

```
VAR
```

```
  i, j, sum : integer;
```

```
FUNCTION add(n1, n2 : integer) : integer;
```

```
  VAR
```

```
    s : integer;
```

```
  BEGIN
```

```
    s := i + j + n1 + n2;
```

```
    add := s;
```

```
  END;
```

```
BEGIN
```

```
  i := 10;
```

```
  j := 20;
```

```
  sum := add(100, 200);
```

```
  writeln('Sum = ', sum)
```

```
END.
```

```
.method public static main([Ljava/lang/String;)V
```

```
.limit stack 4
```

```
.limit locals 1
```

```
  bipush 10
```

```
  putstatic Adder/i I
```

```
  bipush 20
```

```
  putstatic Adder/j I
```

```
  bipush 100
```

```
  sipush 200
```

```
  invokestatic Adder/add(II)I
```

```
  putstatic Adder/sum I
```

```
  ...
```

A function call leaves its return value on top of the operand stack of the caller.

- ❑ Use **putstatic** with a fully qualified field name and type signature to pop a value off the operand stack and store it into a static field.
- ❑ Use **invokestatic** with a fully-qualified method name and a type signature to call a static method.

# Code to Call `System.out.println()`

## □ What does the method call

```
System.out.println("Hello, world!")
```

require on the operand stack?

- A reference to the `java.io.PrintStream` object `System.out`.
- A reference to the `java.lang.String` object `"Hello, world!"`

object

type descriptor of object

```
getstatic    java/lang/System/out Ljava/io/PrintStream;  
ldc          "Hello, world!"  
invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
```

Note: invokevirtual

method

parm type descriptor

no return type (void)

# System.out.println(), cont'd

- Compile the Pascal call

```
writeln('Sum = ', sum)
```

as if it were the Java

Remember to  
use javap!

```
System.out.println(  
    new StringBuilder(" Sum = ")  
        .append(sum)  
        .toString()  
);
```

Each call to `invokevirtual` requires an object reference and then any required actual parameter values on the operand stack.

```
getstatic java/lang/System/out Ljava/io/PrintStream;  
new      java/lang/StringBuilder  
dup ←  
ldc "Sum = "  
invokenonvirtual java/lang/StringBuilder/<init>(Ljava/lang/String;)V  
getstatic      Adder/sum I  
invokevirtual  java/lang/StringBuilder/append(I)Ljava/lang/StringBuilder;  
invokevirtual  java/lang/StringBuilder/toString()Ljava/lang/String;  
invokevirtual  java/io/PrintStream/println(Ljava/lang/String;)V
```

Why do we need this `dup` instruction?



# String.format()

- A more elegant way to compile a call to Pascal's standard `writeln()` procedure is to use Java's `String.format()` method.
- Compile Pascal

```
writeln('The square root of ', n:4,  
        ' is ', root:8:4);
```

as if it were the Java

```
System.out.print(  
    String.format(  
        "The square root of %4d is %8.4f\n",  
        n, root)  
);
```

## `String.format()`, *cont'd*

- ❑ The Java `String.format()` method has a **variable-length** parameter list.
- ❑ The first parameter is the **format string**.
- ❑ Similar to C's format strings for `printf()`.
- ❑ The code generator must construct the format string.

- Pascal:

```
('The square root of ', n:4, ' is ', root:8:4)
```

- Equivalent Java:

```
("The square root of %4d is %8.4f\n", n, root)
```

## `String.format()`, *cont'd*

---

- ❑ The remaining parameters are the values to be formatted, one for each format specification in the format string.
- ❑ Jasmin passes these remaining parameters as a one-dimensional array of objects.
- ❑ Therefore, we must emit code to create and initialize the array and leave its reference on the operand stack.

# String.format(), cont'd

```
s = String.format(
    "The square root of %4d is %8.4f\n",
    n, root);
```

- Instruction **aastore** operands on the stack:
- Array reference
  - Index value
  - Element value (object reference)

ldc	"The square root of %4d is %8.4f\n"
iconst_2	
anewarray	java/lang/Object
dup	
iconst_0	
getstatic	FormatTest/n I
invokestatic	java/lang/Integer.valueOf(I)Ljava/lang/Integer;
aastore	
dup	
iconst_1	
getstatic	FormatTest/root F
invokestatic	java/lang/Float.valueOf(F)Ljava/lang/Float;
aastore	
invokestatic	java/lang/String.format(Ljava/lang/String;[Ljava/lang/Object;)Ljava/lang/String;
putstatic	FormatTest/s Ljava/lang/String;

Create an **array of size 2** and leave the array reference on the operand stack.

Store **element 0**:  
The value of **n**.

Why the **dup**  
instructions?

Store **element 1**:  
The value of **root**.

# String.format(), cont'd

```
System.out.print(  
    String.format("The square root of %4d is 8.4f\n",  
        n, root);  
);
```

```
getstatic    java/lang/System/out Ljava/io/PrintStream;  
ldc          "The square root of %4d is %8.4f\n"  
iconst_2  
anewarray    java/lang/Object  
dup  
iconst_0  
getstatic    FormatTest/n I  
invokestatic  java/lang/Integer.valueOf(I)Ljava/lang/Integer;  
aastore  
dup  
iconst_1  
getstatic    FormatTest/root F  
invokestatic  java/lang/Float.valueOf(F)Ljava/lang/Float;  
aastore  
invokestatic  java/lang/String.format(Ljava/lang/String;  
                                         [Ljava/lang/Object;)Ljava/lang/String;  
invokevirtual java/io/PrintStream.print(Ljava/lang/String;)V
```

Easier: Use the newer  
`System.out.printf()`.

# Code Generation for Arrays and Subscripts

---

- ❑ Code to allocate memory for an array variable.
- ❑ Code to allocate memory for each non-scalar array element.
- ❑ Code for a subscripted variable in an expression.
- ❑ Code for a subscripted variable that is an assignment target.

# Arrays and Subscripts, *cont'd*

---

- Allocate memory for single-dimensional arrays:
  - Instruction `newarray` for scalar elements.
  - Instruction `anewarray` for non-scalar elements.
- Allocate memory for multidimensional arrays:
  - Instruction `multianewarray`.

# Allocating Memory for Arrays

- Recall the code template for a Jasmin method.

Code to allocate **arrays** here!

- Pascal automatically allocates memory for arrays declared in the main program or locally in a procedure or function.
  - The memory allocation occurs whenever the routine is called.
  - This is separate from dynamically allocated data using pointers and **new**.

*Routine header*

```
.method private static signature return-type-descriptor
```

Code for local variables

Code for structured data allocations

Code for compound statement

Code for return

*Routine epilogue*

```
.limit locals n  
.limit stack m  
.end method
```

Therefore, our generated Jasmin code must implement this automatic runtime behavior.



# Example: Allocate Memory for Scalar Arrays

```
PROGRAM ArrayTest;
```

```
TYPE
```

```
vector = ARRAY[0..9] OF integer;  
matrix = ARRAY[0..4, 0..4] OF integer;  
cube   = ARRAY[0..1, 0..2, 0..3] OF integer;
```

```
VAR
```

```
i, j, k, n : integer;  
a1          : vector;  
a2          : matrix;  
a3          : cube;
```

```
BEGIN
```

```
...
```

```
END.
```

```
bipush  10  
newarray int  
putstatic      arraytest/a1 [I  
  
iconst_5  
iconst_5  
multianewarray  [[I 2  
putstatic      arraytest/a2 [[I  
  
iconst_2  
iconst_3  
iconst_4  
multianewarray  [[[I 3  
putstatic      arraytest/a3 [[[I
```

# Access an Array Element of a 2-D Array

```
PROGRAM ArrayTest;
```

```
TYPE
```

```
    matrix = ARRAY[0..2, 0..3]  
              OF integer;
```

```
VAR
```

```
    i, j, k : integer;  
    a2      : matrix;
```

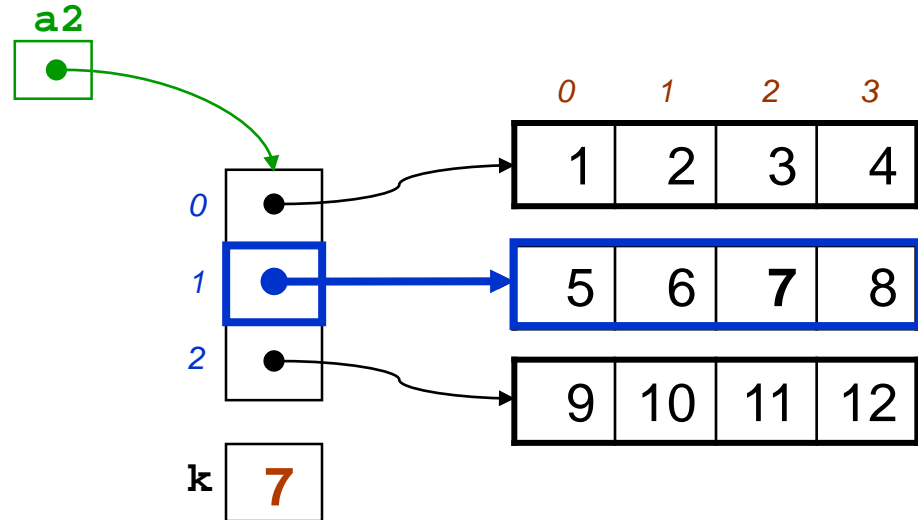
```
BEGIN
```

```
    ...  
    i := 1;  
    j := 2;  
    k := a2[i, j];
```

```
    ...
```

```
END.
```

1	2	3	4
5	6	7	8
9	10	11	12



```
getstatic arraytest/a2 [[I  
getstatic arraytest/i I  
aload  
getstatic arraytest/j I  
iaload  
putstatic arraytest/k I
```