



Lecture 9

Arrays: Design Issues, subscript binding, type signature, Compile time layout

Array data type

- Array data type allows the element indices to be computed at run time.
- Example: int A[30]; // in C language

```
for(int i=0;i<10;i++)
    A[(i+1)*2] = i*4;
```

is a valid C statement as the index is computed at run time.
- An error occurs if $(i+1)*2 = 30$, i.e. if the loop is executed for $i < 15$ (semantic error)
- Language design for an array type focusses on the non-spurious efficient access of data values

Subranges of an array : in C language

- Only size of the array is required to define the subrange
- The subrange is simply $[0, \dots, \text{size}-1]$
- i.e. the array elements are referenced with only these indices e.g. $A[0], A[1], \dots, A[\text{size}-1]$
- Address of $A[i] = \text{base} + (i \times w)$

where w is the width of the element, and base is the address of the array element $A[0]$
- w is known at compile time, while i (array subscript) is computed at run time.

Subranges of an array : in Pascal language

- Var A : array [low..high] of typeT;
- Subrange is (low,..,high)
- The array elements are referenced as A[low],
A[low+1], .., A[high]
- Address of A[i]= base + (i-low)x w
where w is the width of the element, and base is
the address of the array element A[low]
- The width w of the type T is known at compile time,
while i (array subscript) is computed at run time.

Array type signature (type expression)

- The type expression for an array variable can be defined by subrange or size and its basic element type.
- example type expression for Pascal array declared as

a: array [2..4] of integer;

is given as (array, <2,4>, integer)

Pascal array element type

```
Program arraydemo(output);

var
  a: array [2..4] of integer;
  b: array [2..4] of integer;
  c: array [6..8] of integer;
begin
  a[2]:=23;
  b[3]:= 12;
  b[4]:=11;
  c[7]:=20;
  a[3]:=a[2]+b[4];
  a[2]:= b[3] + c[7];
  writeln('a[ ', 3, ' ] = ', a[3] );
  writeln('a[ ', 2, ' ] = ', a[2] );
end.
```

For array elements it does not include subrange comparison for type checking.

```
$main
a[3] = 34
a[2] = 32
```

Bound checking

- Let the variable A be declared in a C-like language as below

```
int A[12];           //static source code
```

- A[10]- Whether index $10 < 12$ or not is computed at compile time
- A[k]- Whether index $k < 12$ or not is computed at run time
- A[i*k]-expression $i*k$ (say t1) is computed at run time and whether index $t1 < 12$ or not is computed at run time

Dynamic arrays

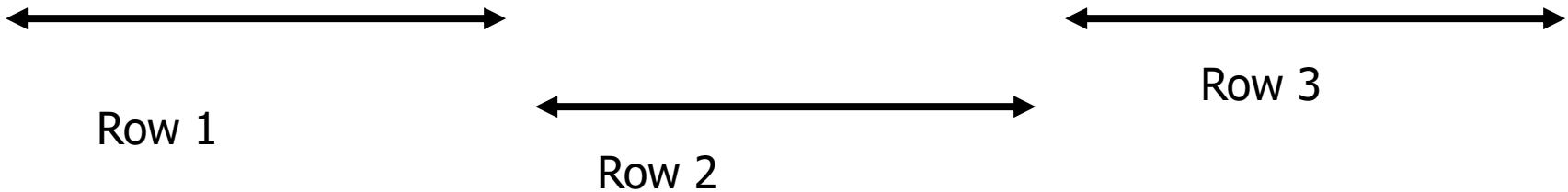
- Dynamic arrays are the array variables whose index ranges may be expanded at any time after creation, without changing the values of its current elements.
- Some languages allow dynamic arrays , e.g. perl
- C does not support dynamic arrays as the size is defined while instructing to resize (malloc and realloc in C)

Layout of Multidimensional arrays

- Address computation
 - row-major layout
 - rows appear side by side
 - column-major layout
 - columns appear side by side

Row major array layout (in C)

11	3	4	6	-1	1	4	9	-2	7	4	12
----	---	---	---	----	---	---	---	----	---	---	----



Address computation of an element $M[i][j]$

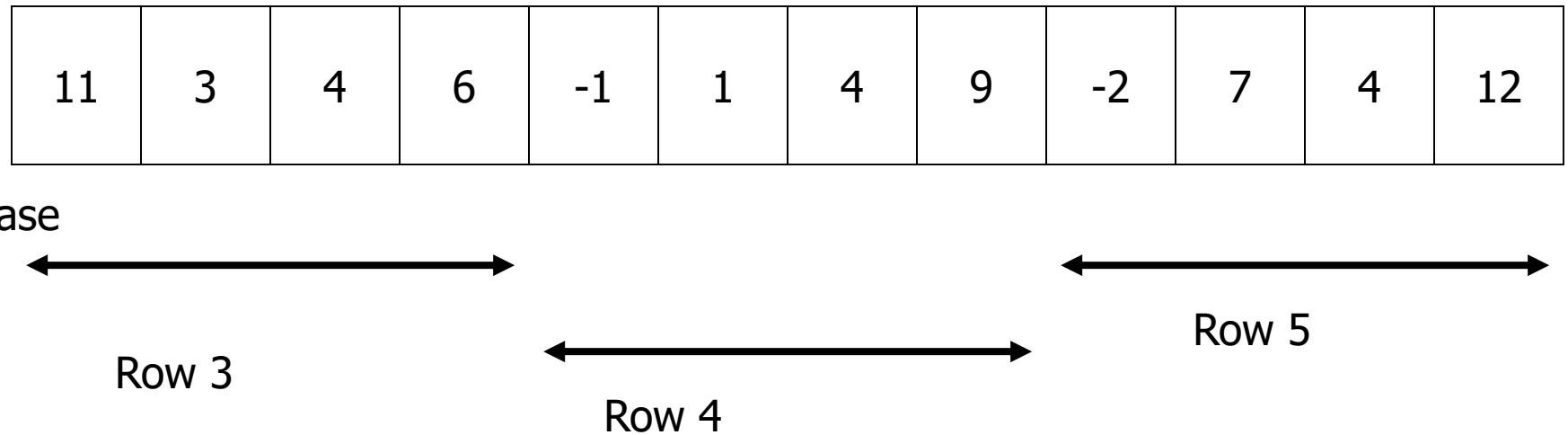
let the size of the matrix be $m \times n$

the type of all elements is same (Let w be the width of the type)

address of $M[i][j] = (i \times n + j) \times w$

Row major array layout (in Pascal)

var M: array[3..5] of [2..5] of integer



Address computation of an element $M[i][j]$

subranges [low1..high1] and [low2..high2]

the type of all elements is same (say w be the width)

address of $M[i][j] = ??$



Lecture 10

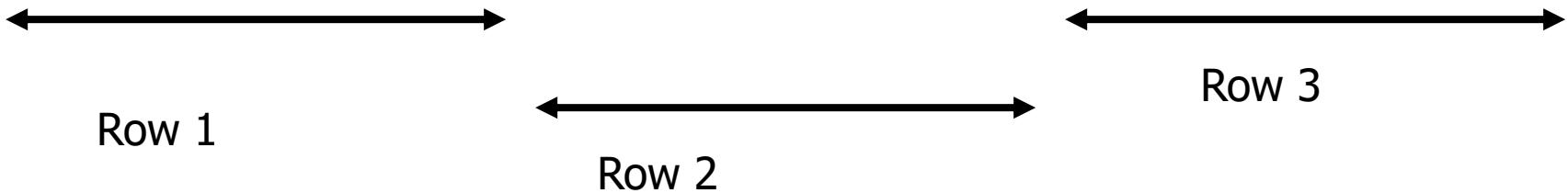
Arrays: Multidimensional array layout, types of arrays, element selector design, Design Issues

Layout of Multidimensional arrays

- Address computation
 - row-major layout
 - rows appear side by side
 - column-major layout
 - columns appear side by side

Row major array layout (in C)

11	3	4	6	-1	1	4	9	-2	7	4	12
----	---	---	---	----	---	---	---	----	---	---	----



Address computation of an element $M[i][j]$

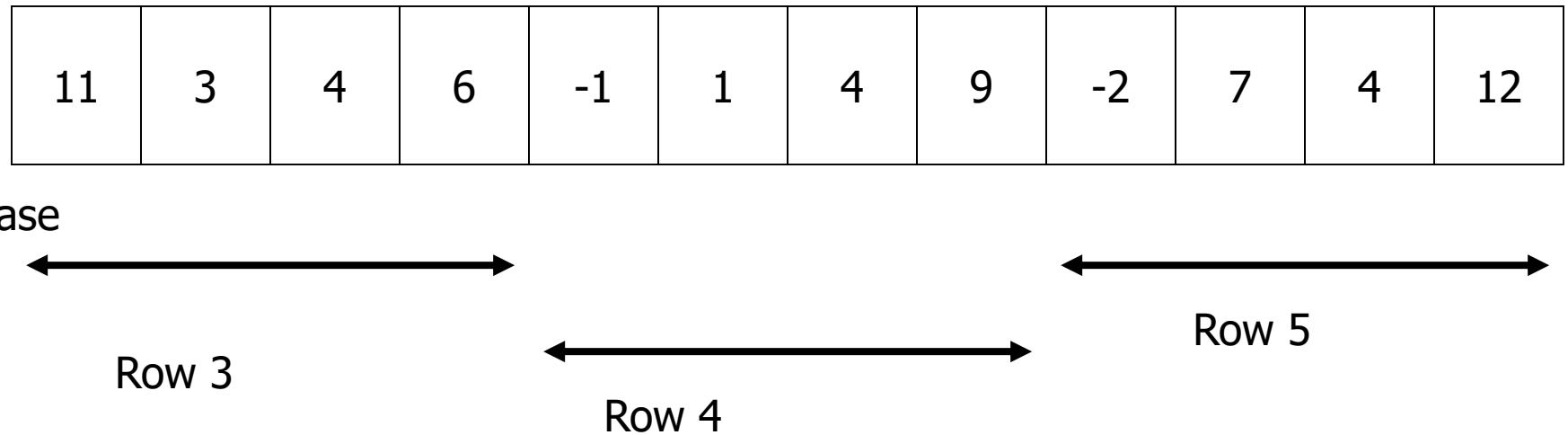
let the size of the matrix be $m \times n$

the type of all elements is same (Let w be the width of the type)

address of $M[i][j] = (i \times n + j) \times w$

Row major array layout (in Pascal)

var M: array[3..5][2..5] of integer



Address computation of an element $M[i][j]$

subranges [low1..high1] and [low2..high2]

the type of all elements is same (say w be the width)

address of $M[i][j] = ??$

Compute the address of A[9][4]

- Var A: array[6..10][3..6] of integer;
- Layout: row major form

Array values and initialization

- Array initialization is done while array declaration

```
int A[] = { 20, -1, 10, -19, 30 }
```

- Size is computed at compile time as is 5. Accordingly 5 contiguous memory locations are allocated for A.

Type signature of an array data type

- Is described by the size of the array and the type of each element
- Example 1
 - `typedef int ARRAY[20];`
 - Type of the data type ARRAY can be described as an expression `array(20, int)`
- Example 2
 - `typedef int matrix[15][20];`
 - Type of the data type matrix can be described as an expression `array(15, array(20,int))`

Design Issues

- What should be the type of the subscripts?
- Whether an expression can be an index of an array element?
- When should the index value be checked to be in permissible limits?
- When can the relative address be computed?
- What should be the layout form for multidimensional arrays?
- Can array be initialized when they have their storage allocated?

Array categories

Name the categories with respect to the subscript binding and storage allocation of arrays you have used.

- Static array
- Fixed stack-dynamic array
- Stack-dynamic array
- Fixed heap-dynamic array



Lecture 11

Operations on array variables, record and union data type, Design Issues, storage allocation

Array Operations

```
#include <stdio.h>
int main()
{
    int *p, *q, x, y;
    int A[7]={1,2, 3,4,5,6,7}, B[7];
    x = 4;
    y = 89;
    p = &x;
    q = &y;
    p = p + q;          //Line 1
    p = A ;             //Line 2
    p = p + 12;         //Line 3
    p = p + A;          //Line 4
    B = q;              //Line 5
    p = p + q;          //Line 6
    q = B;              //Line 7
    return 0;
}
```

Variables in the program: p, q, x, y, A, B

Line 1: Adding unsigned integers-address added to address

Line 2: Assigning start address of array to the location of p

Line 3: Adding a number to an address

Line 4: adding one address to another address

Line 5: Assigning an address to the location where address of B[0] is placed.

Line 6: adding addresses

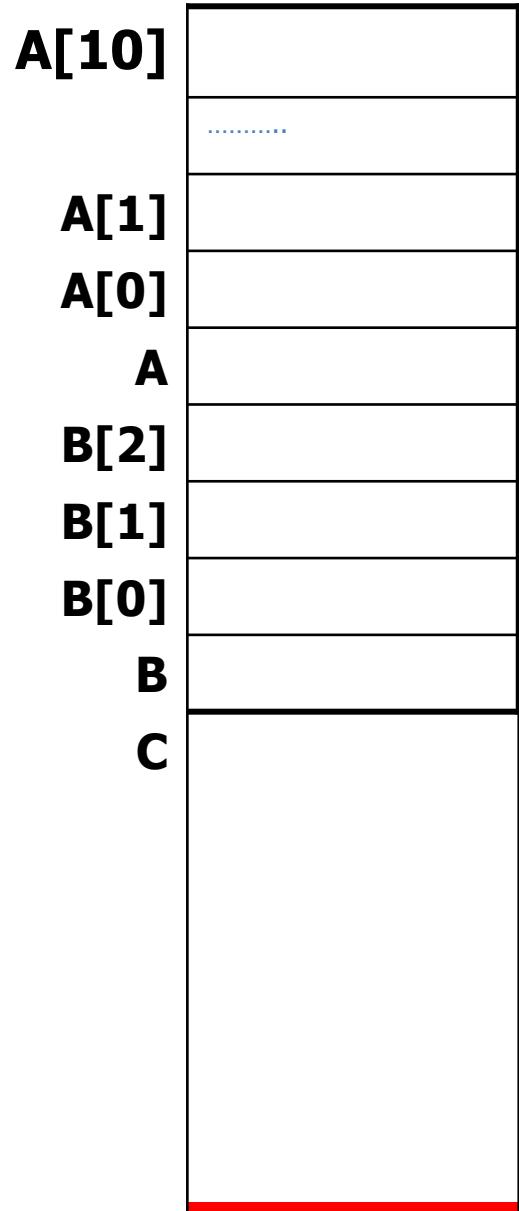
Line 7: assigning start address of the array to the variable q

Array operations

- The array operation refers to the operations on an array (and not its elements)
- Operations:
 - Concatenation (ADA, Python, Ruby)
 - Assignment (Perl, ADA, Python, FORTRAN 95)
 - Comparison (Python, Ruby- the operator ==)
 - Array slicing (Python, Matlab, Perl)
 - Addition (elemental in FORTRAN95)

Array concatenation

- arrays A[11], B[3], C[20] of integers
- $C = A + B$
- Possible Internal operations-
 - copying of all elements of A first and then B.
 - Copying of start address of A and mechanism of reaching B[0] after accessing A[11]
 - Copying of sum of both arrays



Array assignment

```
int x, *p, a[10], b[10];
```

```
x=34;
```

```
p=&x;
```

```
p=a; //valid or invalid in C language?
```

```
a=p; //valid or invalid?
```

- Python-array assignment is only reference change (`a=b`)

Array comparison

- Elemental comparison
- Fortran 95+ : assignment, arithmetic, relational and logical operators are overloaded for arrays of any size and shape.

Other types of arrays

- Jagged arrays- Rows can be of varying length
- Associative arrays- can be accessed using strings as hash keys A[“string”]

Array element selection

Static and dynamic: depending upon whether the index can be computed at compile time or at run time.

- $A[k]$ – run time
- $A[10]$ - compile time
- $A["abc"]$ - compile time (associative arrays)

Design the grammar for array element selection

- $\langle \text{element} \rangle \rightarrow \text{ID SQOP } \langle \text{index} \rangle \text{ SQCL}$
- $\langle \text{index} \rangle \rightarrow \text{ID}$ //run time computable
- $\langle \text{index} \rangle \rightarrow \text{NUM}$ //compile time computable
- Example: Draw parse trees for A[12], A[k]
- Update expression grammar to include array elements in it

$\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle \langle \text{op1} \rangle \langle \text{term} \rangle$

$\langle \text{expr} \rangle \rightarrow \langle \text{term} \rangle$

$\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle \langle \text{op2} \rangle \langle \text{factor} \rangle$

$\langle \text{term} \rangle \rightarrow \langle \text{factor} \rangle$

$\langle \text{factor} \rangle \rightarrow \text{ID} \mid \text{NUM} \mid \text{OP_PAR } \langle \text{expr} \rangle \text{ CL_PAR}$

$\langle \text{factor} \rangle \rightarrow \langle \text{element} \rangle$



Lecture 12

Records and its variants

Records

- Data needs to be described with various attributes of various types
- Grouping of relevant information through different variables forms a record
- All variables have all the properties common (fixed, hence fixed set of fields)
- Operation on record data type is “selection of a field by name”

struct (in C)

User defined type

```
typedef struct {  
    char name[15];  
    char ID[11]  
    int age;  
    char discipline[15];  
    float cgpa;  
} student_rec;  
student_rec s;
```

Access to individual field

```
s.name  
s.ID  
s.age  
s.discipline  
s.cgpa
```

Layout of the record data type

- Each field of a record has its own type, hence individual corresponding layout
- Whatever be the complex structure of a record, the layout can be known at compile time e.g.

```
typedef struct {  
    char name[10];  
    char ID[7];  
    int age;  
    float cgpa;  
} student_rec;  
student_rec s;
```

Example: Field sizes and size of the record variable

```
#include <stdio.h>

int main()
{
    typedef struct {
        char name[12];
        char ID[7];
        int age;
        float cgpa;
    } student_rec;
    student_rec s;
    printf("%d\n", sizeof(s.name));
    printf("%d\n", sizeof(s.ID));
    printf("%d\n", sizeof(s.age));
    printf("%d\n", sizeof(s.cgpa));
    printf("%d\n", sizeof(s));
    return 0;
}
```

Output

```
12
7
4
4
28
```

Understand starting address of the record variable and its fields

```
#include <stdio.h>

int main()
{
    typedef struct {
        char name[12];
        char ID[7];
        int age;
        float cgpa;
    } student_rec;
    student_rec s;
    printf("%d %u\n", sizeof(s.name), &s.name);
    printf("%d %u\n", sizeof(s.ID), &s.ID);
    printf("%d %u\n", sizeof(s.age), &s.age);
    printf("%d %u\n", sizeof(s.cgpa), &s.cgpa);
    printf("%d %u\n", sizeof(s), &s);
    return 0;
}
```

Output

```
12 2984457264
7 2984457276
4 2984457284
4 2984457288
28 2984457264
```

s.name

s.ID

s.age

s.cgpa

Example layout

- Each record gets storage equal to the sum of sizes of fields (aligned to words)
- The field name appears in the same order as they appear in the definition
- Data is aligned to 4 addressable bytes (some bytes remain unused)

Layout of record data type

- All the fields of a value of record type are laid out together
- If the record structure is constructed as
 - Record {
 type1 field1; type2 field 2 ; type3 field3;
}
- If w_1, w_2, w_3 are the sizes of type1,2 and 3 respectively then the data type is laid out as



Type signature (expression) of the record data type

- Type of a record data is defined as the Cartesian products of the fields
- Type of a variable of following record data type is defined as int x int

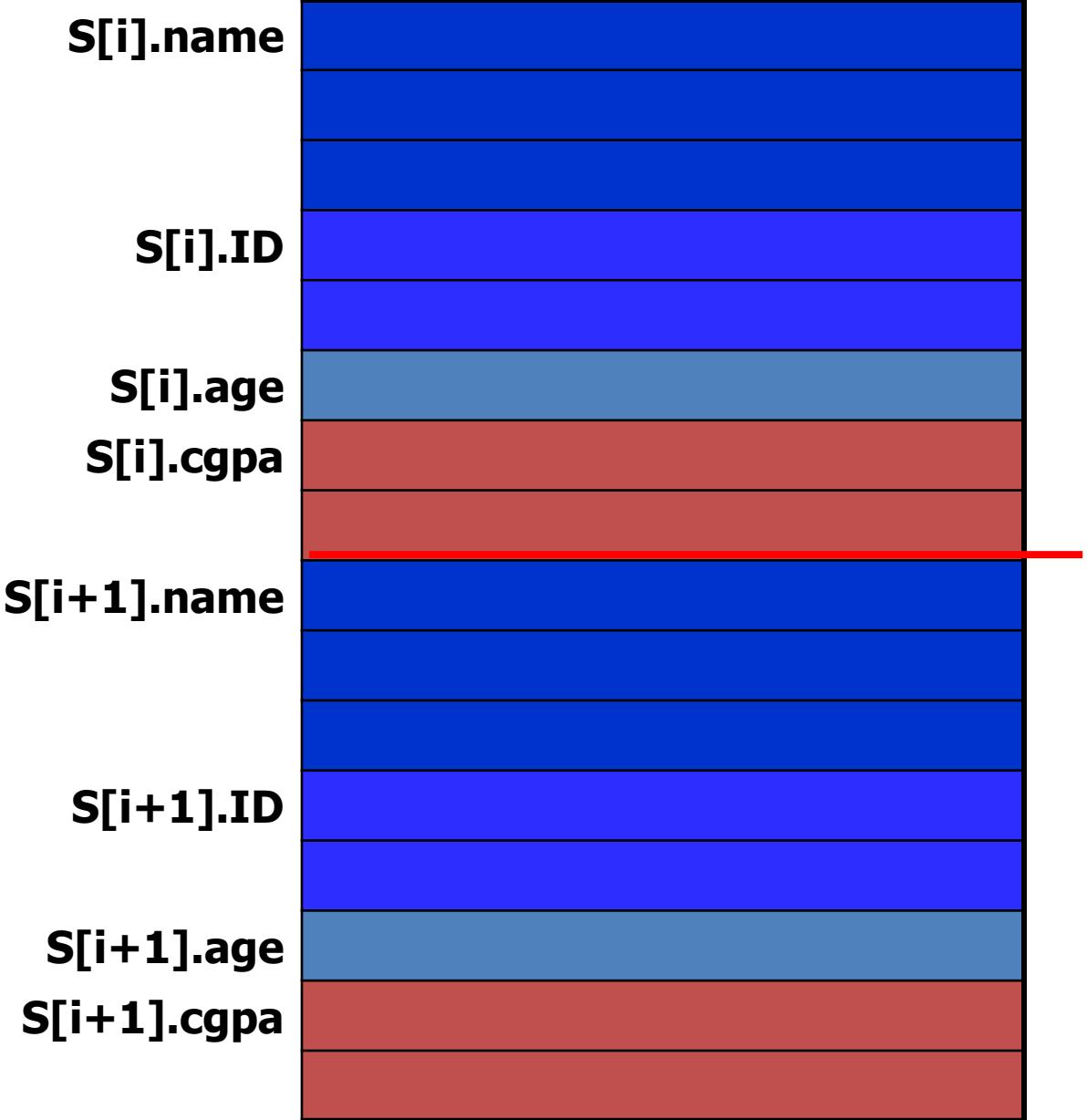
```
struct complex {  
    int real;  
    int imaginary;  
}
```

- Type of the **student_rec** record data type is defined as
array(12, char) x array(7, char) x int x float

Layout of array of records

- Each record is laid as a sequence of contiguous locations with the respective fields laid as discussed
- All records in the array are laid as a sequence of contiguous blocks of memory
- Example student_Rec S[10]

Offset can be
computed at
compile time



Home work: compute the relative offset of $S[i+1].age$

Comparison of Arrays and Records

- array – homogeneous collection of elements
- record- heterogeneous collection of elements

- array element – $A[i]$ can change at run time
- record- field (s.name) is fixed at compile time

- array element –selected by indices anytime
- record- selected by names that are known at compile time

Union type variable and sizes of its field data

```
#include <stdio.h>

int main()
{
    typedef union {
        char name[12];
        char ID[7];
        int age;
        float cgpa;
    } student_rec;
    student_rec s;
    printf("%d %u\n", sizeof(s.name), &s.name);
    printf("%d %u\n", sizeof(s.ID), &s.ID);
    printf("%d %u\n", sizeof(s.age), &s.age);
    printf("%d %u\n", sizeof(s.cgpa), &s.cgpa);
    printf("%d %u\n", sizeof(s), &s);
    return 0;
}
```

Output

```
-  
12 612403476  
7 612403476  
4 612403476  
4 612403476  
12 612403476
```

Issues with data access



Lecture 13

Union data type and variants of records

Union type variable and sizes of its field data

```
#include <stdio.h>

int main()
{
    typedef union {
        char name[12];
        char ID[7];
        int age;
        float cgpa;
    } student_rec;
    student_rec s;
    printf("%d %u\n", sizeof(s.name), &s.name);
    printf("%d %u\n", sizeof(s.ID), &s.ID);
    printf("%d %u\n", sizeof(s.age), &s.age);
    printf("%d %u\n", sizeof(s.cgpa), &s.cgpa);
    printf("%d %u\n", sizeof(s), &s);
    return 0;
}
```

Output

```
-  
12 612403476  
7 612403476  
4 612403476  
4 612403476  
12 612403476
```

Issues with data access

```
int main()
{
    union node{
        int x;
        float y;
        char z;
    } var1;
    printf(" storing and printing values\n");
    var1.x = 2;
    printf("%d \n", var1.x);
    var1.y = 2.3;
    printf("%f \n", var1.y);
    var1.z = 'y';
    printf("%c \n", var1.z);

    printf(" Printing the stored values\n");
    printf("%d \n", var1.x);
    printf("%f \n", var1.y);
    printf("%c \n", var1.z);

    printf(" Printing the stored values again\n");
    printf("%d \n", var1.x);
    printf("%f \n", var1.y);
    printf("%c \n", var1.z);

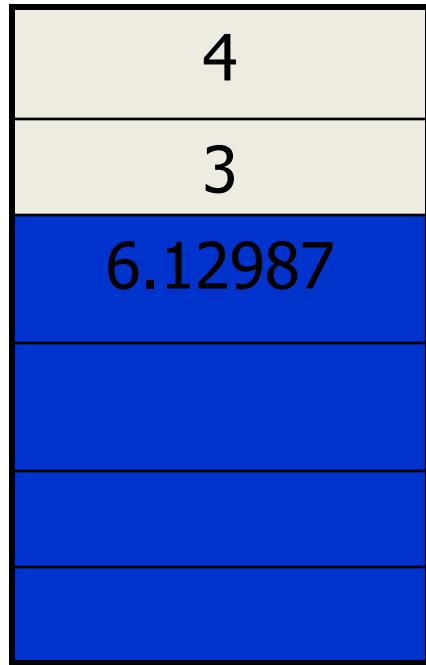
    printf("locations of the field data items\n");
    printf("%u \n", &var1.x);
    printf("%u \n", &var1.y);
    printf("%u \n", &var1.z);
    return 0;
}
```

```
$main
    storing and printing values
2
2.300000
y
    Printing the stored values
1075000185
2.300017
y
    Printing the stored values
1075000185
2.300017
y
    locations of the field data items
1385683868
1385683868
1385683868
```

Variant Records

- A maximum of 12 bytes of the memory is allocated for variable s.(total space layout for each variant record remains of fixed size)
- The variant field may leave some space unused depending upon the choice of the field in union.
- Variant records compromise type safety (tag and choice of the field is programmers job)

Example: Variant Record



```
struct weather{  
    int day;  
    int tag; //1:sunny,2:cloud, 3:rain  
    union data{  
        int brightness;  
        float humidity;  
        double rainfall;  
    }  
}
```

Variant records

- Variant Records have a part common to all records of that type, and a variant part, specific to some subset of the records.
- In C, union construct facilitates the variant records
- Example: registration status of a student

Memory layout of variant records

- The memory equal to the maximum of the sizes of the individual fields is allocated for a variable of union type.
- Example

```
struct var_Rec{  
{  
    type1    var1;    //fixed  
    type2    tag;     //fixed  
    union{  
        int x;  
        float y;  
        char name[12];  
    } var1; // variant  
} v;
```

...	W1(fixed)	W2(fixed)	W3(variant)	...
September 29, 2020	CS F301 [BITS PILANI]			7

Design a node for annotated parse tree

- Annotated parse tree nodes not only store the information about the address of children, they store information about semantics such as type and code.

Sets

- A set of n numbers, each less than n , is implemented by a bit vector of n bits
- Example: set $A = \{3, 6, 1, 8, 15, 0, 2\}$

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	1	0	0	1	1	1	1

- Represented- by a simple integer word.
- limitation - only small element values can find their place as a bit (else two or more words are used)



Lecture 14

Tagged union data types, sets and pointers data type

Union data type

- The variables of union data type may store values of different types at different times during execution of the program.
- C and C++ provide union constructs, but type checking is not in the design of the construct.
- This union construct is also known as **free union**.
- Type checking in union data type required an indicator for field usage.

Tagged or Discriminated union data type

- An indicator, also known as **tag** or **discriminator**, is used to specify the latest field usage for the union variables.
- ALGOL 68 was the first language to provide support for unions with tag.
- Tagged union is supported in ADA, ML, Haskell and F#.

Ada union type

```
with Ada.Text_IO; use Ada.Text_IO;
procedure shapedemo is
    type shape is (circle, triangle, rectangle);
    type colors is (red, green, blue);
    type figure (form : shape) is
        record
            filled : boolean;
            color: colors;
            case form is
                when circle =>
                    diameter : float;
                when triangle =>
                    left_side : integer;
                    right_side : integer;
                    angle : float;
                when rectangle =>
                    side_1:integer;
                    side_2:integer;
            end case;
        end record;
        f1: figure (form=>rectangle) ;
        f2: figure (form=>triangle);
begin
    f1:= (filled=>true, color=>blue, form=>rectangle, side_1=>12, side_2=>3);
end shapedemo;
```

Fields of the record data type

- filled, color and form of different types each.
- Type of field ‘filled’ is Boolean
- Type of field ‘color’ is colors
- Type of ‘form’ is union which can be represented as follows (indicating use of one)
 <circle | triangle | rectangle>

[circle x triangle x rectangle may represent the structure]

- Hence, the type expression for the **figure** record is

boolean x triangle x <circle | triangle | rectangle>

Type expression of discriminated variable **f1** of **figure** data type

- Variable declaration

f1: figure (form=>rectangle) ;

- Type expression of f1

type expression of figure x tag information

boolean x triangle x <circle | triangle | rectangle> x (tag=rectangle)

- Field access by f1

f1. diameter = f1.left_side+2;

- How to keep a check on correct access?

Use type expression of f1 as above (and not that of the figure alone)

What can go wrong with f1 or f2?

- If the following is seen at compile time,
 $f1.\text{diameter} = f.\text{left_side} + f1.\text{right_side}$

The compiler knows

```
f1:figure(form=>rectangle);
```

Then the type expression, includes information about the usage of the variable f1

Ada Union

- **Static type checking:** The variable f2 is declared constrained to be a triangle and cannot be changed to another variant.
- This is an example of discriminated union.
- This way the type checking is done at compile time.
- Therefore the possibility of any access to wrong data is prevented by reporting this as an error and making the data access type safe.

Ada union

- **Dynamic type checking:** The unconstrained variant record variable declared as

```
f1: figure;
```

The variable f1 has no initial value or a discriminator (tag). Therefore if the code has a initialization later in another time instance as shown below, then the type checking is done at run time.

```
f1:= (filled=>true, color=>blue, form=>rectangle,  
side_1=>12, side_2=>3);
```

Implementation of Union types

- The same address is used for all possible variants.
- Sufficient storage is allocated to the largest variant.
- The tag can be the part of the fixed part of the variant record. The tag can indicate use of the variant part of the record.

Tagged union example using C language

1. Type definition

```
#include <stdio.h>

int main()
{
    struct data1{
        int x;
        float y;
        char u;
    };
    struct data2{
        int A[10];
        int B[5];
    };
    union variant{
        int c; //tag=1
        float d; //tag=2
        struct data1 f1; //tag=3
        struct data2 f2; //tag=4
    };
    struct record{
        int value;
        int tag;
        union variant b;
    };
    struct record a;
    int i;
```

a

Tagged union example using C language

2. Use of tag with every new assignment to fields of variant record

```
printf("%d %d %d\n", sizeof(struct data1), sizeof(struct data2),
       sizeof(union variant), sizeof(struct record));
a.value = 30;

//every use of a field of variant record has preceding tag
   initialization
a.tag = 1;
a.b.c = 50;

a.tag = 2;
a.b.d = 4.67;

a.tag = 3;
a.b.f1.x = 34;
a.b.f1.y = 98.23;
a.b.f1.u = 'a';

a.tag = 4;
for(i=0; i<10; i++)
    a.b.f2.A[i] = i*2;
for(i=0; i<5; i++)
    a.b.f2.B[i] = i*3;
```

Tagged union example using C language

3. Use of tag with every field data access

```
//if the following two statements are uncommented and the code is
//executed, you get 50 as output
//a.tag = 1;
//a.b.c = 50;
if(a.tag == 1)
    printf("%d\n", a.b.c);

if(a.tag == 2)
    printf("%d\n", a.b.d);

if(a.tag == 3)
    printf("%d, %f, %c\n",a.b.f1.x, a.b.f1.y, a.b.f1.u);

if(a.tag == 4)
    printf("print arrays A and B appropriately\n");
return 0;
```

output

12 60 60 68
print arrays A and B appropriately

Tagged union example using C language

The latest value of tag is used for dynamic type checking.

```
//if the following two statements are uncommented and the code is
//executed, you get 50 as output
a.tag = 1;
a.b.c = 50;
if(a.tag == 1)
    printf("%d\n", a.b.c);

if(a.tag == 2)
    printf("%d\n", a.b.d);

if(a.tag == 3)
    printf("%d, %f, %c\n", a.b.f1.x, a.b.f1.y, a.b.f1.u);

if(a.tag == 4)
    printf("print arrays A and B appropriately\n");
return 0;
```

output

```
12 60 60 68
50
```

Sets

- A set of n numbers, each less than n , is implemented by a bit vector of n bits
- Example: set $A = \{3, 6, 1, 8, 15, 0, 2\}$

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	1	0	1	0	0	1	1	1	1

- Represented- by a simple integer word.
- limitation - only small element values can find their place as a bit (else two or more words are used)

Sets

- Operations on sets are implemented as bit operations
- Example (in C) (not supported by C directly)
 - AddElement(set A, element e):

```
mask=1;  
A=A | (mask<<e); // use left shift operator
```
 - set union: bitwise OR
 - set intersection: bitwise AND
 - set complement: bitwise NOT
 - isMember(set A, element e): if $((1<<e)!=0)$ then $e \in A$

Sets (in Pascal)

- Pascal supports the set type.
- Syntax:

```
var A: set of [1..3]
```

- The variable A can denote one of the following sets
 - [], [1], [2], [3], [1,2], [2,3], [3,1], [1,2,3]
- The subset [1,3] can be denoted by the bit string 101



Lecture 15

Pointer data type

Pointer Type

- A pointer type is a value that provides indirect access to elements of a known type.
- Declaration usage (in C)

```
int *p;
```

// p holds the address of an integer value

- Pointer data type
 - Range of values: Addresses and a NULL value
 - Operations: = (assignment), + (addition), - (subtraction), → (access), * (dereferencing)

Pointer as a variable

- Declared using some special ways
- In C language

```
int *p, *q;
```

```
struct node *r;
```

- In pascal

```
var p:^integer;
```

- Grammar for C like pointer declaration

$\langle \text{declarationStmt} \rangle \rightarrow \langle \text{Type} \rangle \langle \text{list} \rangle$

$\langle \text{list} \rangle \rightarrow \text{ID COMMA } \langle \text{List} \rangle \mid \text{ID} \mid \text{STAR ID COMMA } \langle \text{LIST} \rangle \mid \text{STAR ID}$

Example of Pointer usage

- In C language

```
int x; //declaration of x
```

```
int *p; // pointer declaration
```

...

```
x=15; //initialization of x
```

```
p=&x; //associating p to x
```

- Address of x: x301A

- Address of p:

x303E

- p=address of x

- *p is the value 15 (dereferencing)

Address	Data (shown in decimal for convenience)
x3002	..
x3006	..
x300A	20
x300E	12
x3012	34
x3016	18
x301A	15
x302E	78
x3032	
x3036	
x303A	
x303E	
x3042	

Example C code

```
#include <stdio.h>

int main()
{
    int x; //declaration of x
    int *p; // pointer declaration
    x=15; //initialization of x
    printf(" x= %d address of location bound to x = %u contents of
           memory location bound to p = %u\n", x, &x, p);
    //printf("contents of location addressed by p\n", *p); // Produces
               //garbage
    p=&x; //associating p to x
    printf(" x= %d address of location bound to x = %u contents of
           memory location bound to p = %u\n", x, &x, p);
    printf("contents of location addressed by p =%d\n", *p);
    return 0;
}
```

```
x= 15 address of location bound to x = 3930661620 contents of memory location bound to p = 0
x= 15 address of location bound to x = 3930661620 contents of memory location bound to p = 3930661620
contents of location addressed by p =15
```

Example Pascal code

```
program example1;
var
    x: integer;
    p: ^integer;

begin
    x := 20;
    writeln('x= ', x);

    p := @x;
    writeln('p points to a value: ', p^);

    p^ := 45;
    writeln('x= ', x);
    writeln('p points to a value: ', p^);
end.|
```

```
x= 20
p points to a value: 20
x= 45
p points to a value: 45
```

Pointers

- Pointers are treated as first class citizens (can be used with integers with simple arithmetic)
- The static checking does not prevent them from being computed as infeasible address (as a result of simple arithmetic)
- A pointer can point to a value of a predefined type.

Use of pointer data type

- Indirect addressing
- Way to manage dynamic storage

Heap dynamic variables

- Variables that are dynamically allocated from the heap are called heap dynamic variables.
- These variables do not have identifiers associated with them and are called as **anonymous variables**.
- These variables are accessed only by the pointer variables, which themselves are bound to the location in the call stack and store the starting address of the dynamically allocated memory.
- In C,

```
struct node *p;  
float *q;  
p = (struct node *) malloc (sizeof(struct node)*37);  
q = (float *) malloc (sizeof(float)*18);
```

Storage visualization for pointer to heap dynamic variable

```
struct node {  
    int x;  
    float u;}; // assume 2  
    locations used  
Struct node *p, *q;  
Struct node a;
```

```
p=(struct node *) malloc(sizeof(struct node)*3);
```

```
p->x = 1000;  
P→u = 12.45;
```

```
a=*p;
```

```
q = p;
```

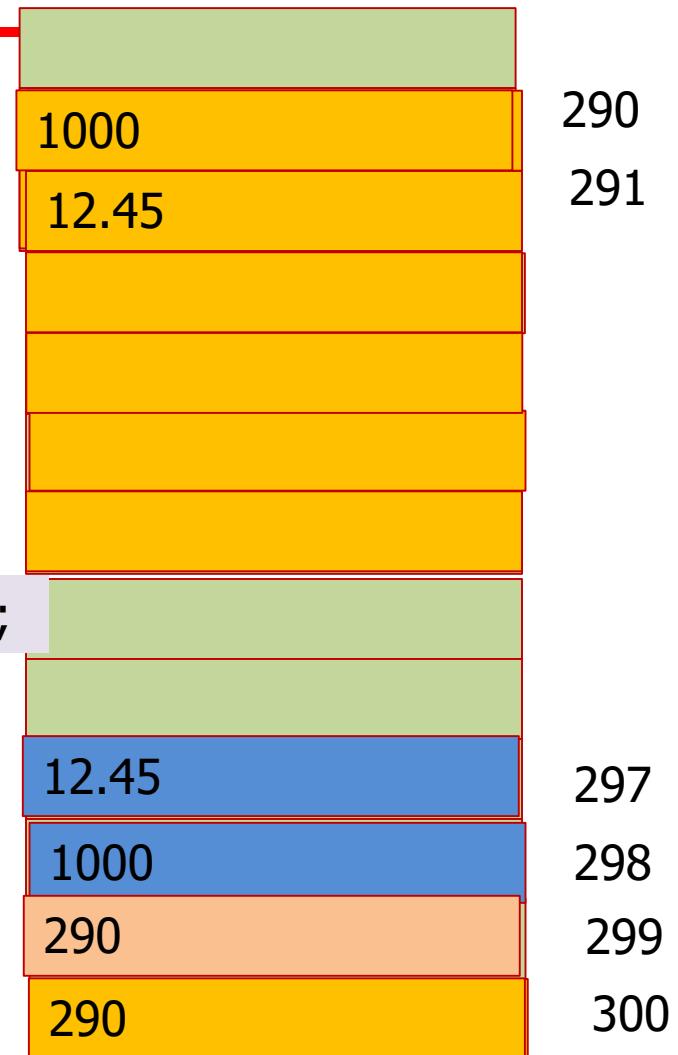
Heap

Call stack

a

q

p



Pointer operations

- Assignment
 - Contents of pointer variable are populated with an address
 - If the pointer variable is heap dynamic and is used to manage the heap dynamic storage, then in-built functions are used to assign address value to the pointer variable.

```
p = (struct node *) malloc (sizeof(struct node)*20);
```

- If the pointer variable is used for indirect addressing to variables that are not heap dynamic, then special operator is needed to associate the address of a variable

```
p = &x;
```

```
*p=20;
```

Pointer operations

- Dereferencing: When a memory location is indirectly accessed by a pointer as in $p=&x$, then in order to access the data stored in memory bound to x , an explicit operator $*$ is used.
- $*p$ dereferences the pointer p to get value of the location whose address is stored in the location of p .
- Some language implicitly dereference based on the type of variable p which is pointer to integer for example. In FORTRAN 95+, it is implemented implicitly.

Dereferencing a record fields

```
struct node {  
    int x, y;  
    float z;  
};
```

```
struct node *p;
```

Dereferencing the pointer

P→x

P→y

P→z

(*p).x

(*p).y

(*p).z

Problems in Pointers

- Dangling Pointer

The dangling pointer contains the address of the heap dynamic variable that has been deallocated.

```
Struct node *p, *q, *r;  
p = (struct node *) malloc (sizeof(structnode)*30);      //allocation  
q=p;  
free(p);          //deallocation  
printf("%d\n", q->x);        //using the dangling pointer
```

- Memory leak

```
p = (struct node *) malloc (sizeof(structnode)*30);      //M1  
r = (struct node *) malloc (sizeof(structnode)*20);      //M2  
p = r;  //causes memory M1 not reachable through p and r.
```

The anonymous heap dynamic variable is said to have been lost in this process.

Ada pointers

- Ada provides support to reduce possibilities of dangling pointers by not allowing user to explicitly deallocate.
- But it also has a feature that allows the user to deallocate explicitly by using a keyword 'Unchecked_Deallocation'
- Ada does not have inbuilt support to reduce the memory leaks.

Pointers

- Type checking prevents the pointer variables from pointing to wrong data type.
- Pointers have fixed size independent of what they point to.
- Pointer variable fits into a single machine location.

Design issues

- Which text segment is the pointer definition valid?
- Whether the pointer type definition survive across the functions or through out the program execution or not?
- What is the lifetime of the heap dynamic variable pointed to by the pointer variable?
- Should the pointer access be restricted for the type of variable it is pointing to?
- Should the pointer variable be used for indirect addressing or for storage management of the heap dynamic variable, or both?

Function types

- If $R_type \text{ function_name} (T1 \text{ arg1}, T2 \text{ arg2})$
 - the type of the function is defined as the function $(T1, T2) \rightarrow R_type$
 - Type mismatch occurs in
$$\text{value} = \text{function_name}(\text{val1}, \text{val2})$$
If either val1 is not of type T1
Or val2 is not of type T2
Or value is not of type R_type



Lecture 16

Procedure Activations

Elements of a procedure or subprogram

- Name for the procedure
- Body of code describing declarations and statements
- Formal parameters
- Result type

Function definition and call

- A **function definition** construct has a specific name and formal parameters to communicate with the calling procedure.
- A **function call** is a construct that uses name of existing procedures/ functions through the actual parameters.

Understanding flow of execution control: An example of procedure calls

```
int square(int x)
{
    int sq;
    sq=x*x;
    return sq;
}
```

```
function1()
{
```

```
    int a,b,c,d;
```

```
    a=5;b=6;
```

```
    c=square(a); //line1
```

```
....
```

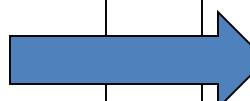
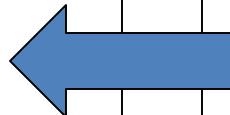
```
    d=square(b); //line2
```

```
main()
```

```
{
```

```
    function1();
```

```
    function1();
```



```
}
```

Flow of Execution control

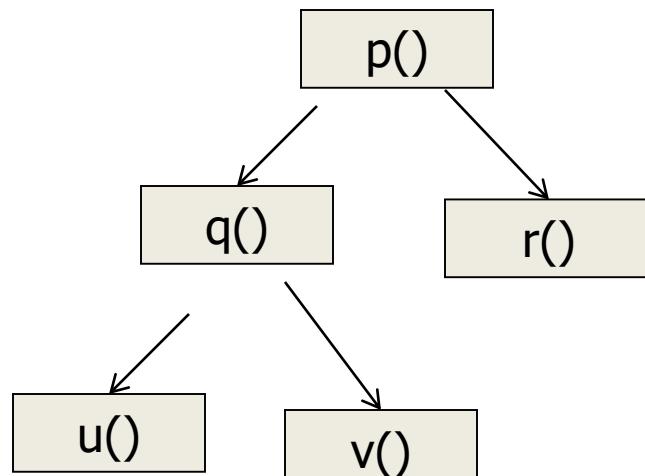
```
p()  
{  
    statements of p;  
    q();  
    r();  
    remaining statements of p  
}
```

- Procedure calls are nested in time
- If procedure **p** calls procedure **q** and procedure **r** (refer example)
 - The execution of **q** starts, the execution of **p** suspends
 - The execution of **r** starts only after execution of **q** is over
 - The execution of **p** resumes when execution of **r** is over

Activation tree

- The activations of procedures during execution of the entire program is represented by a tree, called an activation tree.

```
p()  
{  
    statements of p;  
    q();  
    r();  
    remaining statements of p  
}  
q()  
{  
    ...  
    u();  
    v();  
    ...  
}
```



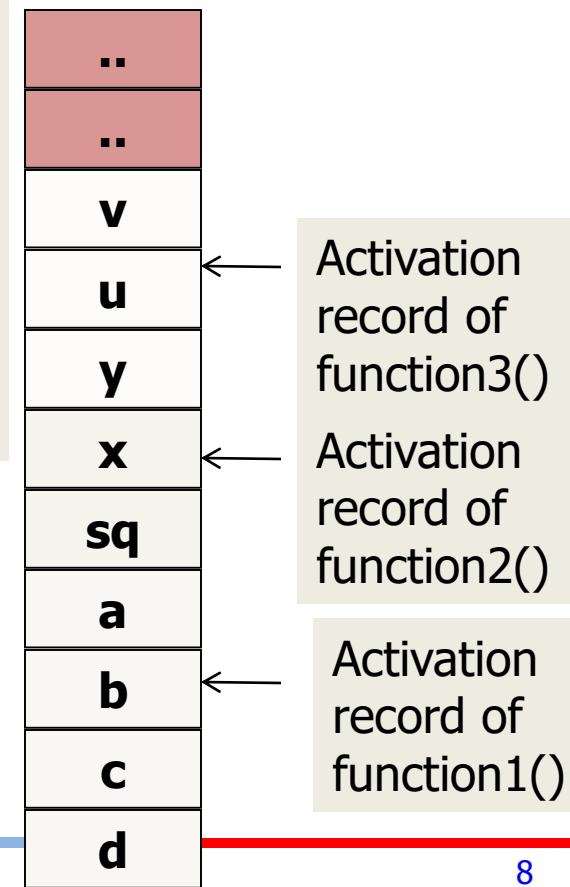
Sequence of procedure calls corresponds to the preorder traversal of the activation tree

Run time Stack

- Each live activation of a procedure call is maintained by a data structure called activation record
- When a procedure is called, its activation record is pushed onto the stack

Activation Records

1. Each time a procedure or function is called, space for its local variables is pushed onto a stack
2. When the procedure terminates, the space is popped off the stack.
3. Non overlapping functions may share the stack space.
4. function calls keep the stack growing
5. Execution of the functions keeps the stack shrinking



Activation Records

Why Stack area?

1. Space is allocated at **stack** area of the memory at each function call
2. Space includes not just the local variables, it is needed for keeping the return values, function parameters, control link etc.
3. Size of activation record is fixed corresponding to one function call

Actual parameters
Returned values
Control links
Access link
Saved machine status
Local data
temporaries

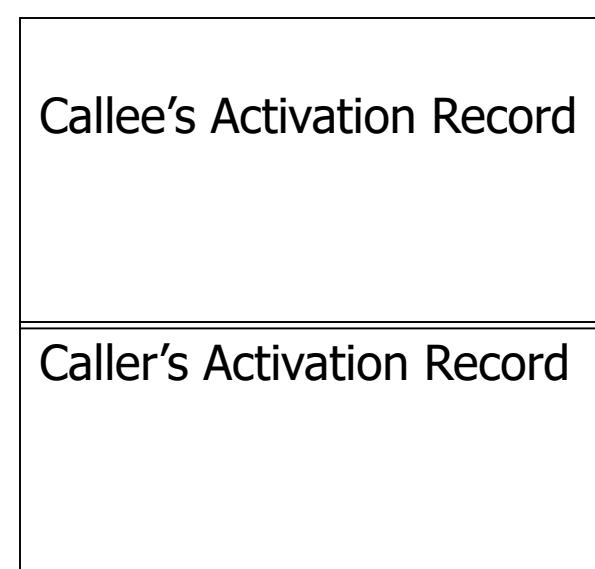
Calling Sequences

- Compiler code that allocates an activation record on the stack
- The calling sequences enter the fields of the activation records when a function is called

Communication of values between the caller and the callee



- **Formal Parameters** (part of callee's activation record)
 - **Actual parameters** (part of caller's activation record)
 - **Return value** (part of callee's activation record)
- **Above values are placed in the beginning of the callee's Activation Records**





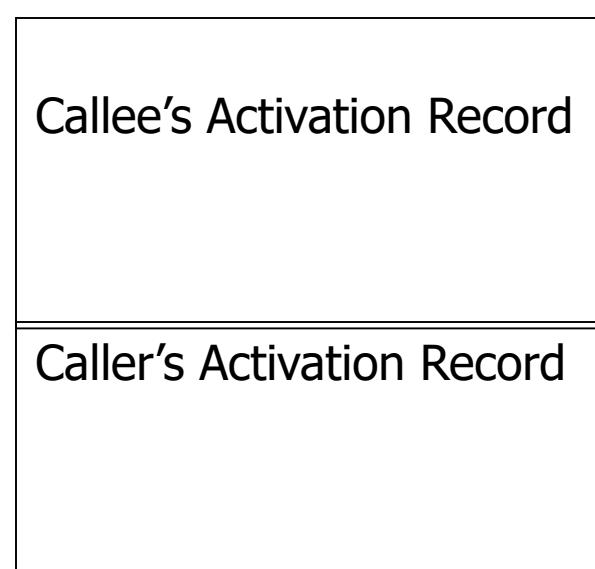
Lecture 17

Procedure Activations

Communication of values between the caller and the callee



- **Formal Parameters** (part of callee's activation record)
 - **Actual parameters** (part of caller's activation record)
 - **Return value** (part of callee's activation record)
- **Above values are placed in the beginning of the callee's Activation Records**



Activation record structure

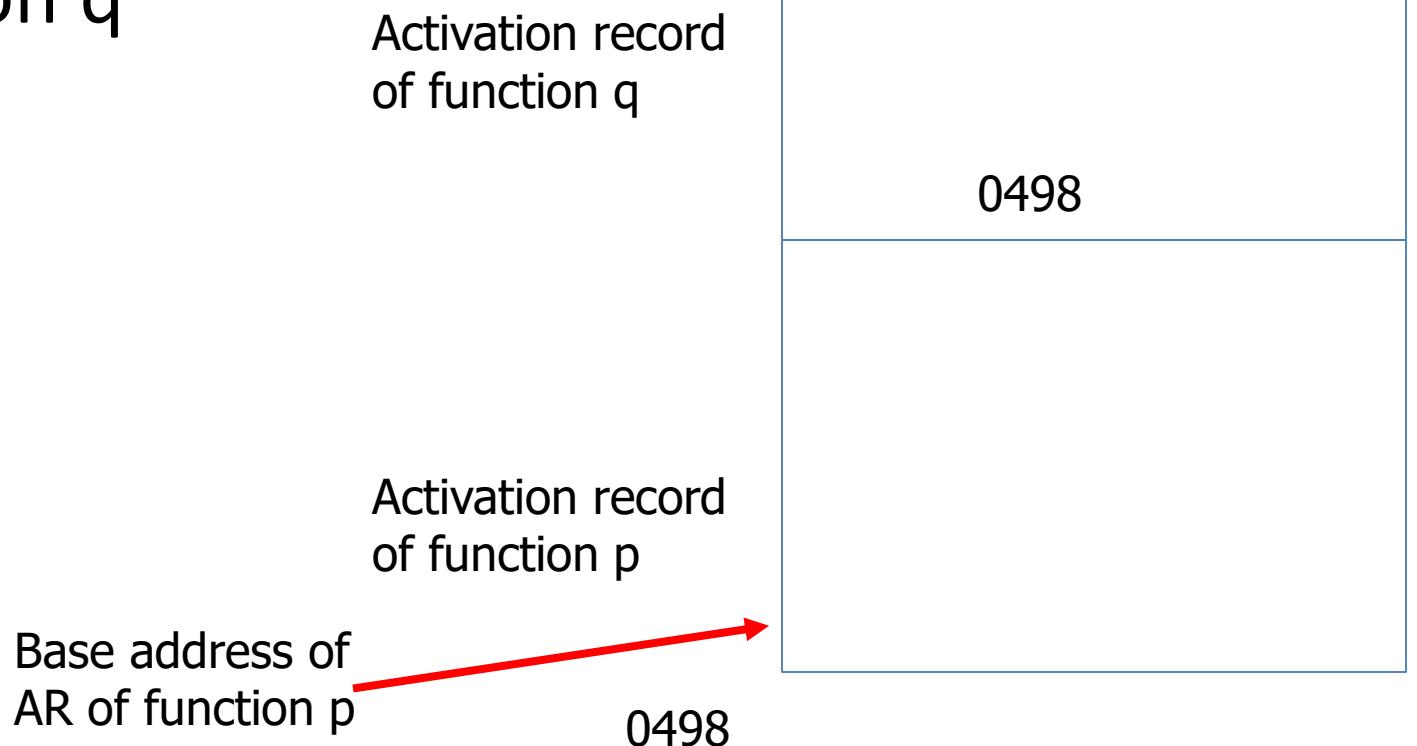
- Values communicated between caller and callee- beginning of the callee's activation record
- Fixed length items (control link, access link, m/c status fields)- middle of the activation record
- Local variables (fixed length, variable length)- placed at the end
- Temporaries (values are finally mapped in registers)- not known before code generation

Control and access links

- Control link: This is the link used at run time to maintain address of the calling function's activation record.
- Access link: This is used to implement the nested scoped languages following static scope rules.

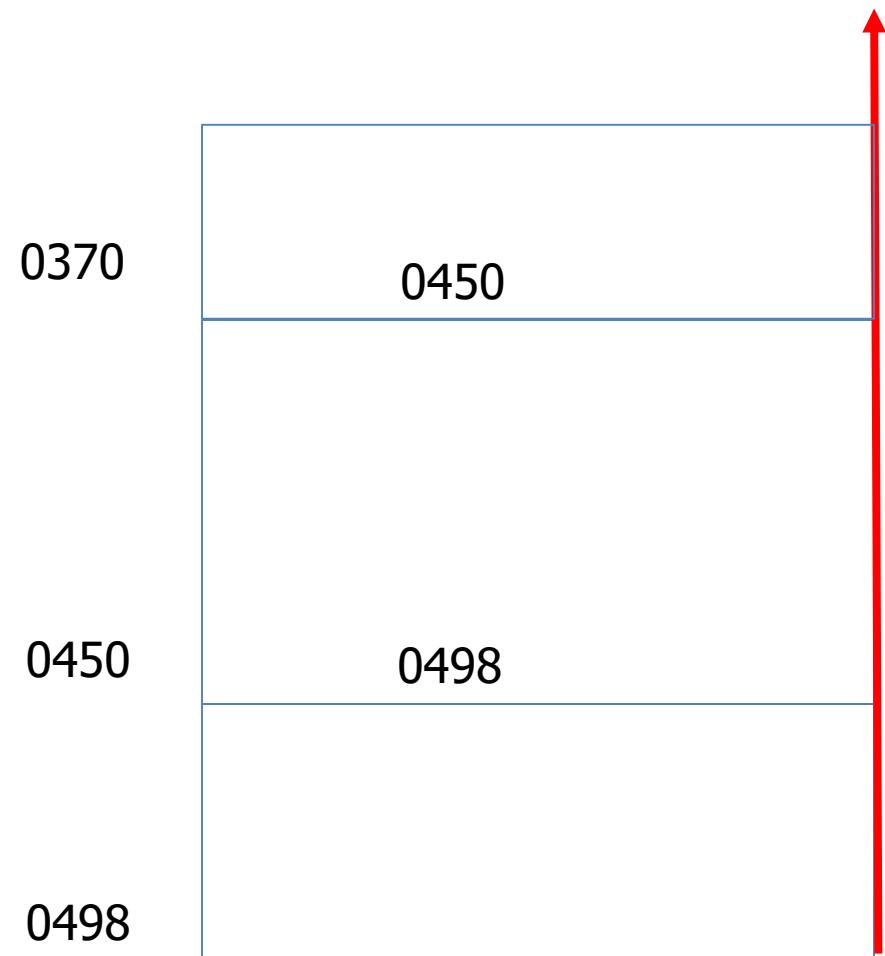
Control link

- Let function p call another function q



Growth of call stack

- `function_1()` calls `function_2()`
- `function_2()` calls `function_3()`
- Then there are three activation records active at run time on the call stack.
- The activation record of `function_3()` is at the top of the stack.
- The activation record of `function_1()` is at the bottom of the stack.



Scope of a variable

- Scope rules of a language determine which declaration of a name x applies to an occurrence of x in a program.
- Example:

```
#include <stdio.h>
#include<stdlib.h>
int main()
{
    int x, y;
    float z;
    x=20;
    printf("Nesting level:1 x = %d\n", x);
    {
        float x;
        x= 3.45;
        printf("Nesting level:2  x = %f\n", x);
    }
    printf("Nesting level:1  x = %d\n", x);
}
```

```
Nesting level:1 x = 20
Nesting level:2  x = 3.450000
Nesting level:1  x = 20
```

Nested procedure definitions

```
Function_definition <parameter_list>
{
    local variables x, y, z
    .....
    function_definition_f2 <parameter_list>
    {
        local variables u, v, w
        use of u, v, x in an expression
    }
}
```

Languages that support nested procedures

- Algol 60
 - Algol 68
 - Pascal
 - Ada
 - Javascript
 - Python
 - Ruby
 - Lua
- etc.

Scope rules

- Scope rules determine how a particular occurrence of a name is associated with a variable.
- This association provides base for **non local** accesses of the variables.
- The scope can be one block of code text, or a procedure/function definition or the code segment of another procedure/function (parent) having nested definition of another procedure/function (child)

Example: nested scope

```
#include<stdio.h>
main()
{
    int p = 7;
    int q = 4;
    int r = 15;
    {
        int p = 6;
        int r = 12;
        p = p + q + r;
        printf("L1: %d %d %d ", p,q,r);
        {
            int r =21;
            int q = -2;
            p = p + q + r;
            printf(" L2: %d %d %d ", p,q,r);
        }
        {
            int q = 11;
            int p = 19;
            p = p + q + r;
            printf(" L3: %d %d %d ", p,q,r);
        }
        p = p + q + r;
        printf(" L4: %d %d %d ", p, q, r);
    }
}
```

Local and non-local variables

- Local variable: A variable is local in a function definition or block, if it is declared there.
- Non-local variable: These variables are visible in the program block or function definition, but are not declared there.
 - A variable occurrence as an argument in an expression or as an actual parameter looks for the declaration of that variable that applies to it based on the scope rules
 - Example

```

int p = 6;
int r = 12;
p = p + q + r;
printf("L1: %d %d %d ", p,q,r);
{
    int r =21;
    int q = -2;
    p = p + q + r;
    printf(" L2: %d %d %d ", p,q,r);
}
  
```

The variable p is
non-local

Variables r and q
are local to the
block

The expression
uses p, q and r
on the RHS

Scope rules for binding names to non-local variables

- Static scope
- Dynamic scope

Static scope

- The scope of the variable is statically computed. It is **compile time** computable
- Which declaration applies to a name occurrence of a variable is known only by looking at the source code.
- The code segment holding the non-local variable's declaration is known as **Static Parent**.

Dynamic Scope

- This is based on the **calling sequence** of the functions.
- The association of the named occurrence (in a piece of code of the function say C1) of a variable to its declaration (type definition) does not depend on the position in the code text. It depends on the text segment of the function that called C1.

Example code

```
f1(){
    int x, y, z;          //D1
    x=10; y=15; z= 7;
    x=x*2+y;            //U1
    f2(){
        int w;  //D2
        w= z + x; //U2
    }
    f3(){
        int z, u; //D3
        z=17;
        u = z + y - x; //U3
        f2();
    }
    call f3();
}
```

Consider the variable occurrence of z at line U2

Which definition of z applies here? (to get the data from the associated location)

Think of offset of the location bound to z

Static scoping: D1 applies for z at U2

Dynamic scoping: D3 applies for z at U2

Example code

```

f1(){
    int x, y, z;          //D1
    x=10; y=15; z= 7;
    x=x*2+y;            //U1
    f2(){
        int w;  //D2
        w= z + x; //U2
    }
    f3(){
        int z, u; //D3
        z=17;
        u = z + y - x; //U3
        f2();
    }
    call f3();
}

```

Static parent of f2() is f1()

Static parent of f3() is f1()

Dynamic parent of f2() is f3()

Dynamic parent of f3() is f1()

Dynamic ancestor of f2() is f1()

Static scoping: D1 applies for x at U2

Dynamic scoping: D1 applies for x at U2



BITS Pilani
Pilani Campus

Principles of Programming Language

Amit Dua
Computer Science and Information Systems Department
BITS, Pilani





Lambda Calculus

Lecture slides courtesy Prof. Sriram Rajamani

Progress

- Lambda Calculus—Motivation
- Syntax
- Scope and binding
- α -Renaming
- β -Reduction
- Call by value Vs call by name

Today -

- Untyped lambda calculus
 - Syntax & Operational semantics
 - "Programming" with lambda calculus
- References:
1. Ben Pierce's book: "Types and programming languages"
 2. George Neukirch's lecture notes

Syntax:

$e ::=$

- $x \leftarrow \text{variable}$
- $\lambda x. e \leftarrow \text{function abstraction}$
- $e_1 e_2 \leftarrow \text{function application}$
- $(e) \leftarrow \text{bracketed expression}$

terms

Conventions:

$e_1 e_2 e_3 \equiv (e_1 e_2) e_3 \leftarrow \text{application}$
 $\text{is left associative}$

$\lambda x. x \lambda y. xy \equiv \lambda x. (x \lambda y. (xy))$
 $\text{scope of } \lambda \text{ expands as far right as possible}$

Examples:

$\lambda x. x$

$\lambda x. \lambda y. x$

$\lambda f. \lambda x. f(f x)$ function that takes
function f , value x
and applies f on x
twice

Identity

function that takes
2 arguments x & y
and returns first
argument x

Lambda calculus

From Wikipedia, the free encyclopedia

http://en.wikipedia.org/wiki/Lambda_calculus

We will
use lambda
calculus to
study foundations
of type systems

In [mathematical logic](#) and [computer science](#), lambda calculus, also λ -calculus, is a [formal system](#) designed to investigate [function](#) definition, function application, and [recursion](#). It was introduced by [Alonzo Church](#) and [Stephen Cole Kleene](#) in the [1930s](#); Church used lambda calculus in 1936 to give a negative answer to the [Entscheidungsproblem](#). Lambda calculus can be used to define what a [computable function](#) is. The question of whether two lambda calculus expressions are equivalent cannot be solved by a general algorithm. This was the first question, even before the [halting problem](#), for which [undecidability](#) could be proved. Lambda calculus has greatly influenced [functional programming languages](#), such as [Lisp](#), [ML](#) and [Haskell](#).

Lambda calculus can be called the smallest universal programming language. It ~~consists~~ of a single transformation rule (variable substitution) and a single function definition scheme. Lambda calculus is universal in the sense that any computable function can be expressed and evaluated using this formalism. It is thus equivalent to the [Turing machine](#) formalism. However, lambda calculus emphasizes the use of transformation rules, and does not care about the actual machine implementing them. It is an approach more related to software than to hardware.

Entscheidungsproblem

From Wikipedia, the free encyclopedia

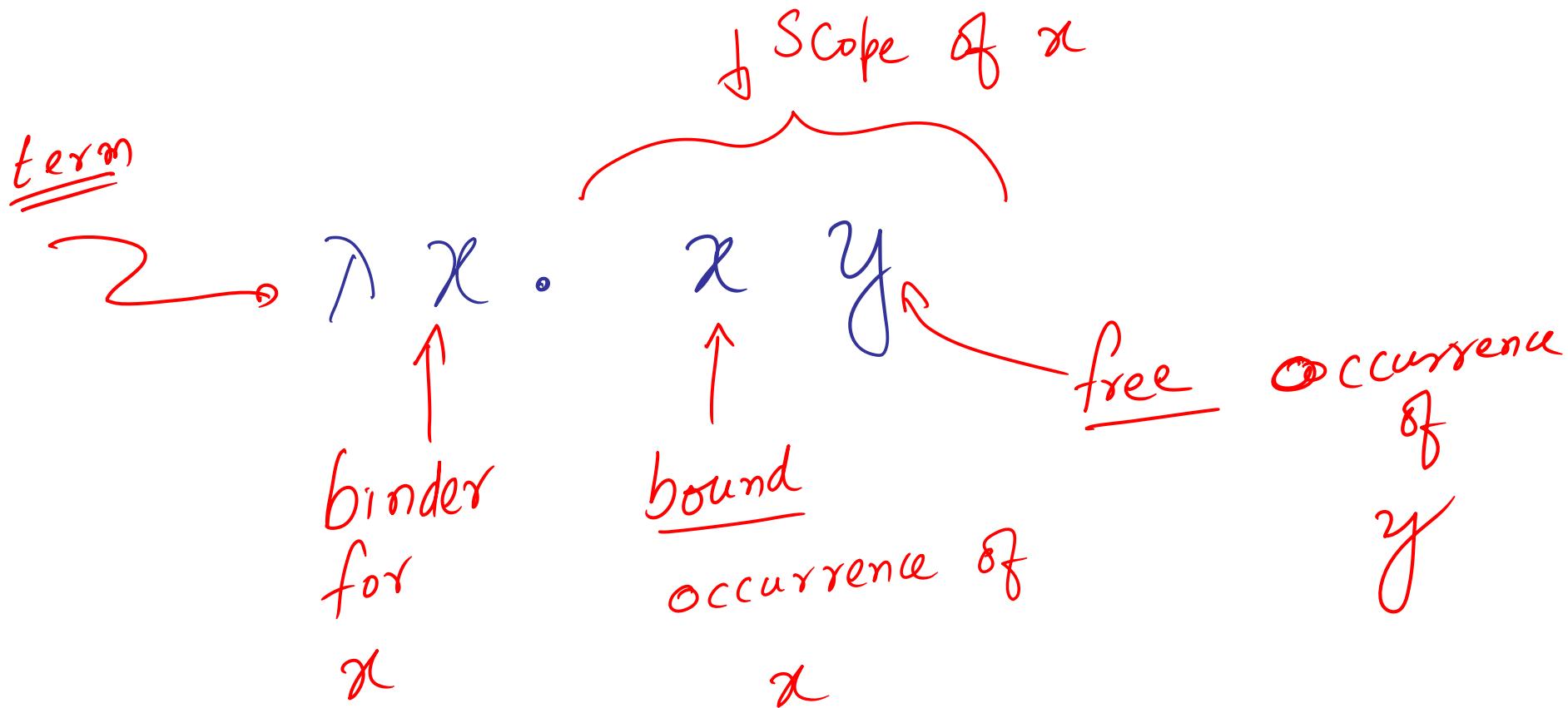
<http://en.wikipedia.org/wiki/Entscheidungsproblem>

In [mathematics](#), the *Entscheidungsproblem* ([German](#) for '[decision problem](#)') is a challenge posed by [David Hilbert](#) in 1928.

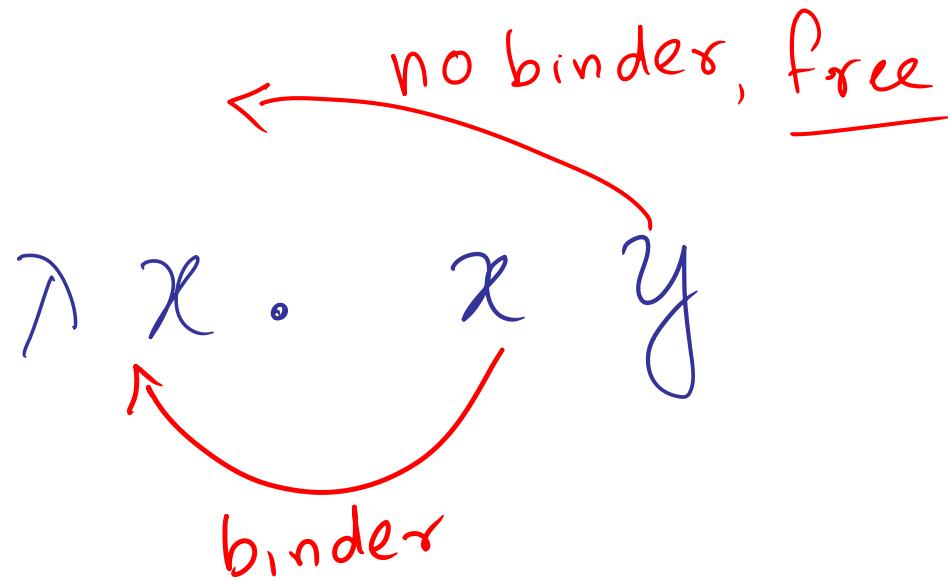
The Entscheidungsproblem asks for a computer program that will take as input a description of a formal language and a mathematical statement in the language and return as output either "True" or "False" according to whether the statement is true or false. The program need not justify its answer, or provide a proof, so long as it is always correct. Such a computer program would be able to decide, for example, whether statements such as the [continuum hypothesis](#) or the [Riemann hypothesis](#) are true, even though no proof or disproof of these statements is known.

In 1936, [Alonzo Church](#) and [Alan Turing](#) published independent papers showing that it is impossible to decide algorithmically whether statements in [arithmetic](#) are true or false, and thus a general solution to the Entscheidungsproblem is impossible. This result is now known as the Church-Turing Theorem

Scope, binding, bound & free occurrences



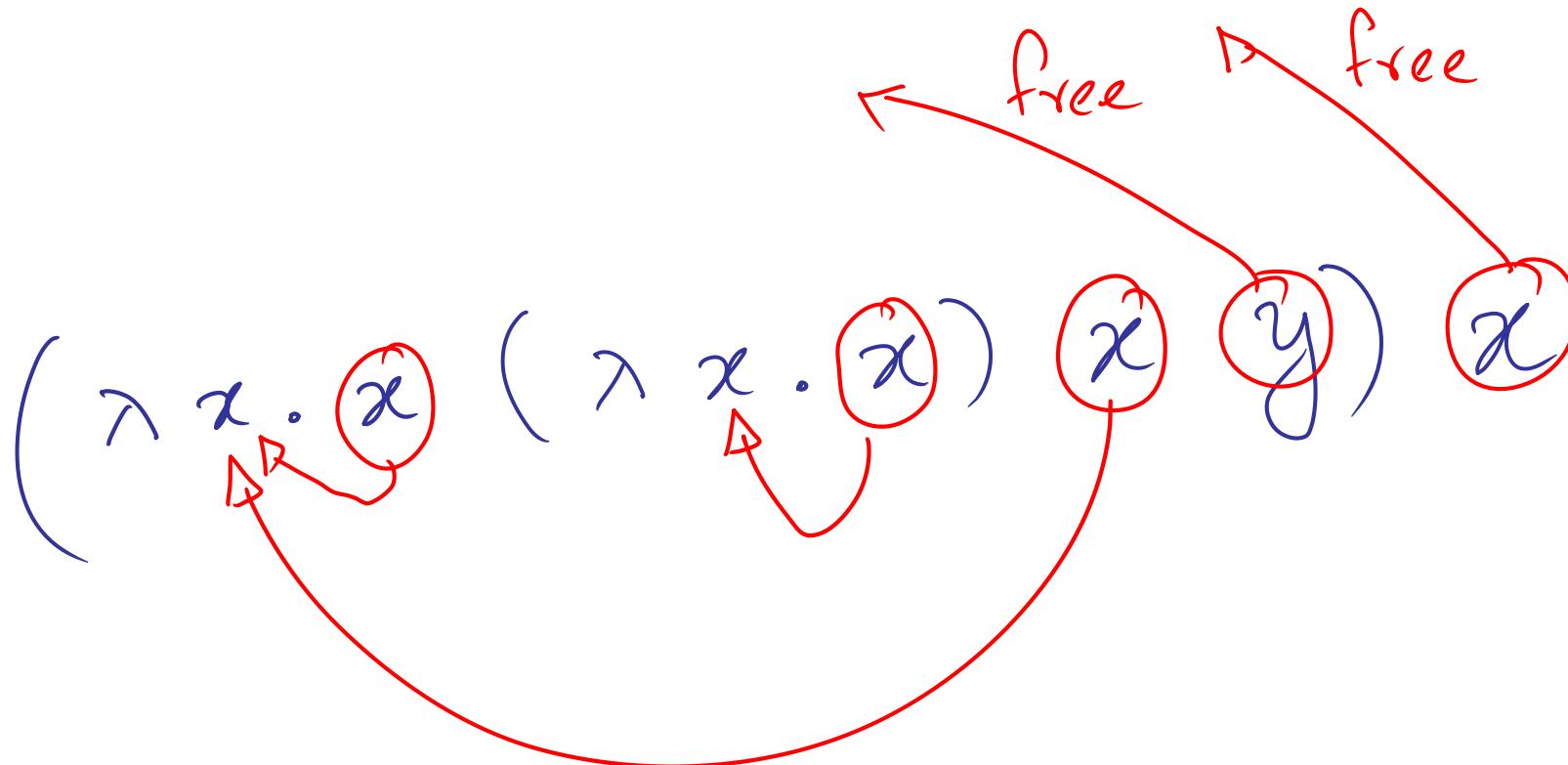
Scope, binding, bound & free occurrences



Find the bound & free occurrences &
binders for bound occurrences

$$(\lambda x. x (\lambda x. x) x y) x$$

Find the bound & free occurrences &
binders for bound occurrences



λ - renaming
Renaming bound variables

$$\lambda x. x \equiv \lambda y. y \equiv \lambda \bar{y}. \bar{y}$$

$$\lambda x. x (\lambda x. x) x \equiv \lambda x. x (\lambda y. y) x$$

↑
easier to understand,
only one binding per variable

de Bruijin notation for λ -terms

de Bruijin index of a variable
 \triangleq number of λ s that separate the
occurrences from the binder

de Bruijin notation : replace variable occurrences
by de Bruijin indexes

$$\lambda x. \lambda y. x y \equiv \lambda. \lambda. 1 \circ$$

$$\lambda x. \lambda x. x \equiv \lambda. \lambda. 0$$

$$\lambda z. \lambda y. z y \equiv \lambda. \lambda. 0$$

Combinators

A λ -term without any free variables
is a combinator

e.g.:

$$I = \lambda x. x$$

$$K = \lambda x. \lambda y. x$$

$$S = \lambda f. \lambda g. \lambda x. f x (g x)$$

$$D = \lambda x. x x$$

$$Y = \lambda f. (\lambda x. f(x x))(\lambda x. f(x x))$$

Theorem:

Any combinator is "equivalent" to
one written with S, K & I

eg: $D =_P S \text{ II}$

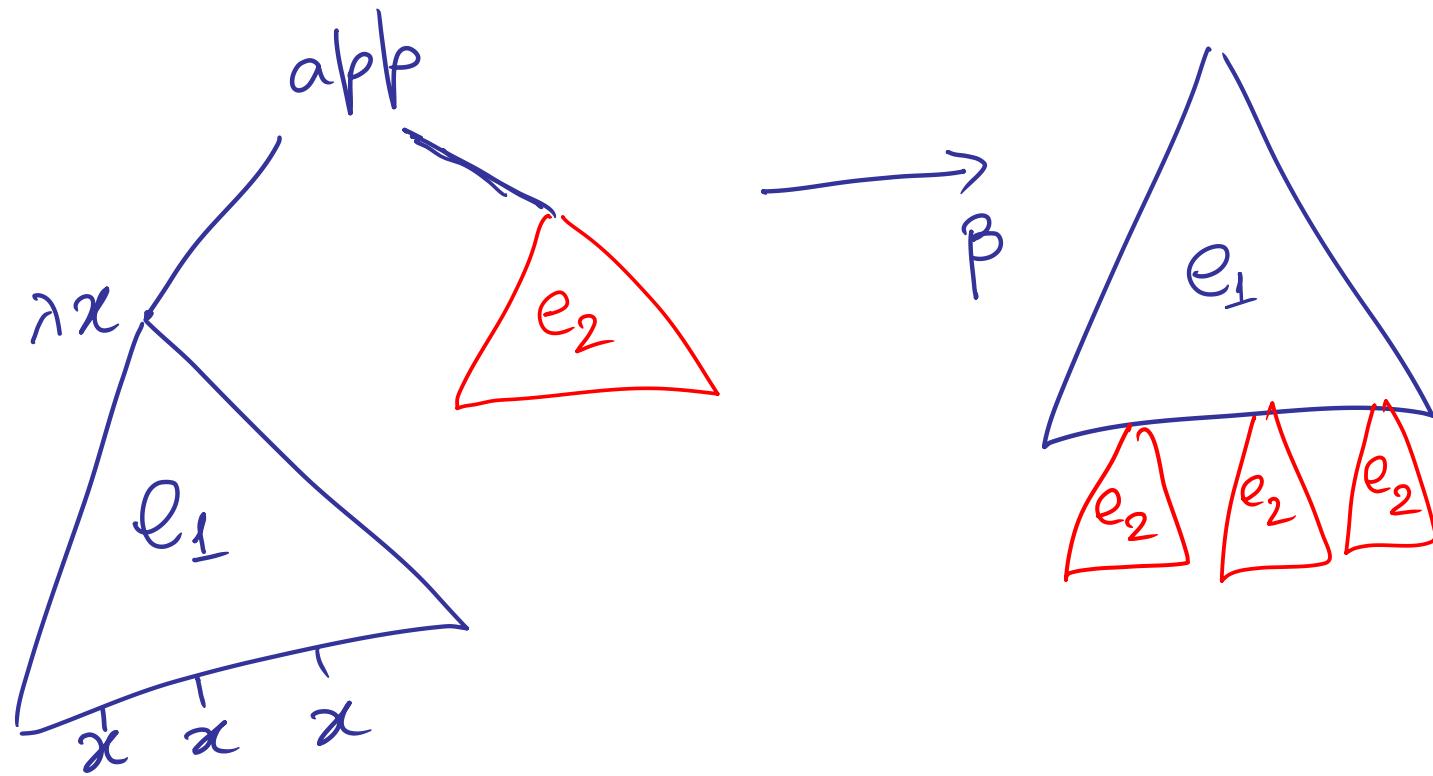
\uparrow
*define
later*

Operational semantics

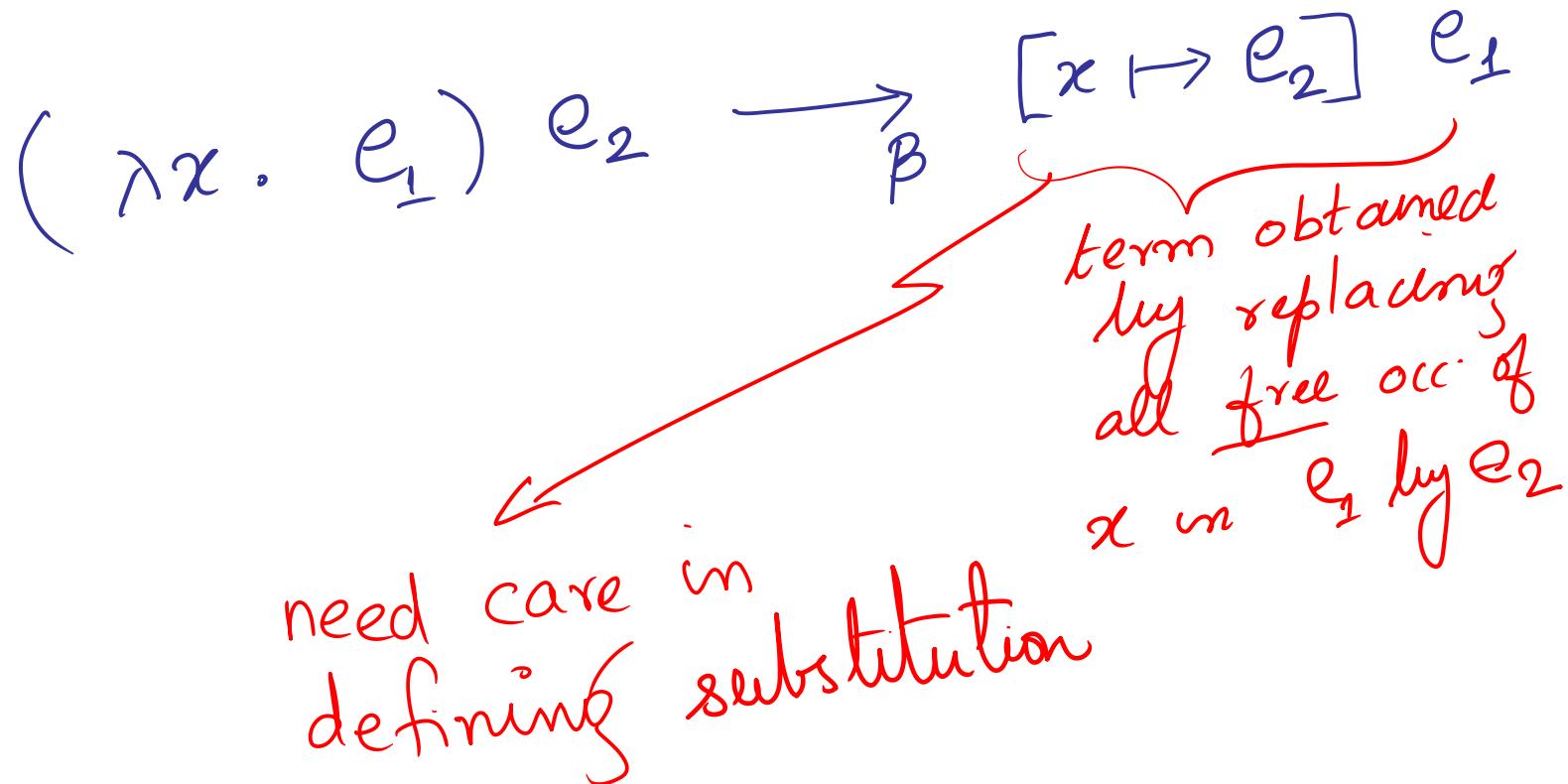
$$(\lambda x. e_1) e_2 \xrightarrow{\beta} [x \mapsto e_2] e_1$$

term obtained
by replacing
all free occ. of
 x in e_1 by e_2

Pictorially



Operational semantics



Defining substitution first attempt

$$[x \mapsto s] x = s$$

$$[x \mapsto s] y = y, \text{ if } x \neq y$$

$$[x \mapsto s] (\lambda y \cdot e) = \lambda y \cdot [x \mapsto s] e$$

$$[x \mapsto s] (e_1 e_2) = ([x \mapsto s] e_1) ([x \mapsto s] e_2)$$

$$\text{eg!}: [x \mapsto \lambda z \cdot z^w] (\lambda y \cdot x) =$$

$$[x \mapsto y] (\lambda x \cdot x) =$$

Defining substitution first attempt

$$[x \mapsto s] x = s$$

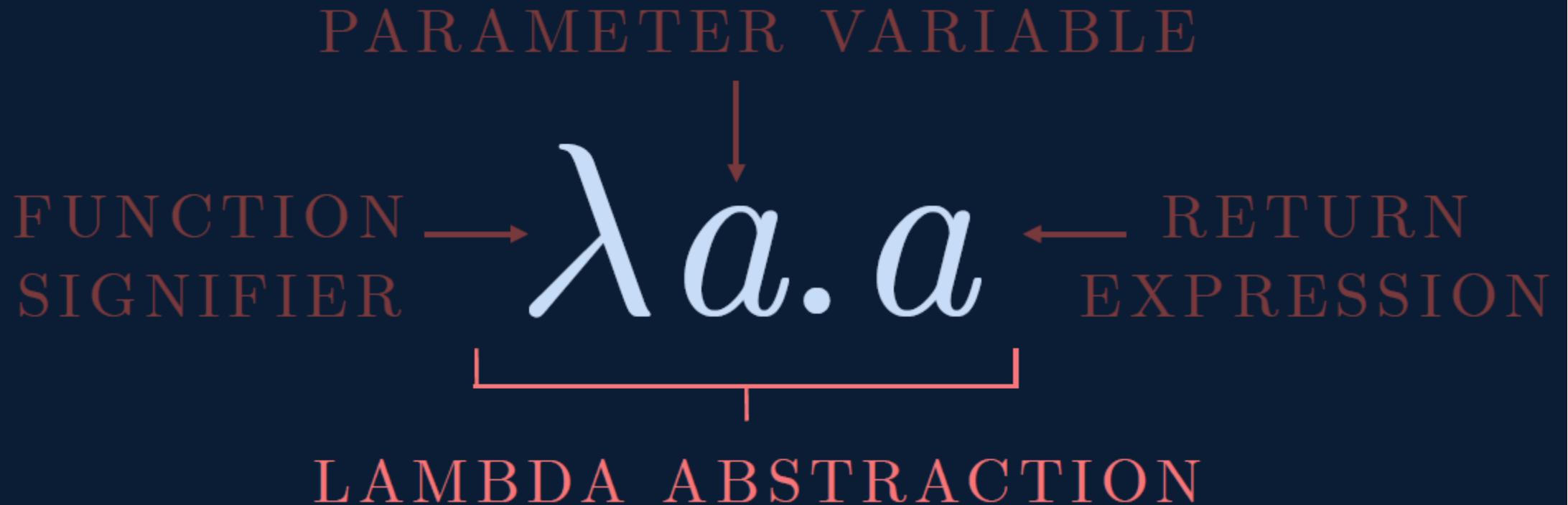
$$[x \mapsto s] y = y, \text{ if } x \neq y$$

$$[x \mapsto s] (\lambda y. e) = \lambda y. [x \mapsto s] e$$

$$[x \mapsto s] (e_1 e_2) = ([x \mapsto s] e_1) ([x \mapsto s] e_2)$$

eg¹: $[x \mapsto \lambda z. z^\omega] (\lambda y. x) = \lambda y. \lambda z. z^\omega$ ✓

$$[x \mapsto y] (\lambda x. x) = \lambda x. y \times$$



λ -CALCULUS SYNTAX

expression ::= variable	<i>identifier</i>
expression expression	<i>application</i>
λ variable . expression	<i>abstraction</i>
(expression)	<i>grouping</i>

APPLICATIONS

$f\ a$

$\mathbf{f}(\mathbf{a})$

$f\ a\ b$

$\mathbf{f}(\mathbf{a})(\mathbf{b})$

$(f\ a)\ b$

$(\mathbf{f}(\mathbf{a}))(\mathbf{b})$

$f(a\ b)$

$\mathbf{f}(\mathbf{a}(\mathbf{b}))$

β -REDUCTION

$$(((\lambda a.a)\lambda b.\lambda c.b)(x)\lambda e.f$$

β -REDUCTION*

$$\begin{aligned} & ((\lambda a.a)\lambda b.\lambda c.b)(x)\lambda e.f \\ = & (\lambda b.\lambda c.b) \quad (x)\lambda e.f \\ = & (\lambda c.x) \quad \lambda e.f \\ = & x \end{aligned}$$

β -NORMAL FORM

COMBINATORS

functions with no free variables

$\lambda b.b$ combinator

$\lambda b.a$ not a combinator

$\lambda ab.a$ combinator

$\lambda a.ab$ not a combinator

$\lambda abc.c(\lambda e.b)$ combinator

Haskell Example

```
fType :: Int -> Int -> Int  
fType x y = x*x + y*y  
main = print (fType 2 4)
```

Haskell Example

```
add :: Integer -> Integer -> Integer      --function declaration  
add x y = x + y                            --function definition  
main = do  
    putStrLn "The addition of the two numbers is:"  
    print(add 2 5)                          --calling a function
```

Defining substitution Second attempt

$$[x \mapsto s] x = s$$

$$[x \mapsto s] y = y \quad , \text{ if } x \neq y$$

$$[x \mapsto s] (\lambda y. e) = \begin{cases} \lambda y. [x \mapsto s] e & \text{if } y \neq x \\ \lambda y. e & \text{if } y = x \end{cases}$$

$$[x \mapsto s] (e_1 e_2) = ([x \mapsto s] e_1) ([x \mapsto s] e_2)$$

eg 2: $[x \mapsto z] (\lambda z. x) =$

Defining substitution Second attempt

$$[x \mapsto s] x = s$$

$$[x \mapsto s] y = y \quad , \text{ if } x \neq y$$

$$[x \mapsto s] (\lambda y. e) = \begin{cases} \lambda y. [x \mapsto s] e & \text{if } y \neq x \\ \lambda y. e & \text{if } y = x \end{cases}$$

$$[x \mapsto s] (e_1 e_2) = ([x \mapsto s] e_1) ([x \mapsto s] e_2)$$

eg 2: $[x \mapsto z] (\lambda z. x) = \lambda \vec{z}. \vec{z} X$
 capture!

Defining substitution Second attempt

$$[x \mapsto s] x = s$$

$$[x \mapsto s] y = y, \text{ if } x \neq y$$

$$[x \mapsto s] (\lambda y. e) = \begin{cases} \lambda y. [x \mapsto s] e & \text{if } y \neq x \wedge y \notin FV(s) \\ \lambda y. e & \text{if } y = x \end{cases}$$

$$[x \mapsto s] (e_1 e_2) = ([x \mapsto s] e_1) ([x \mapsto s] e_2)$$

eg 2: $[x \mapsto z] (\lambda z. x) =$

Defining substitution

final attempt
correct!

$$[x \mapsto s] x = s$$

$$[x \mapsto s] y = y \quad , \text{ if } x \neq y$$

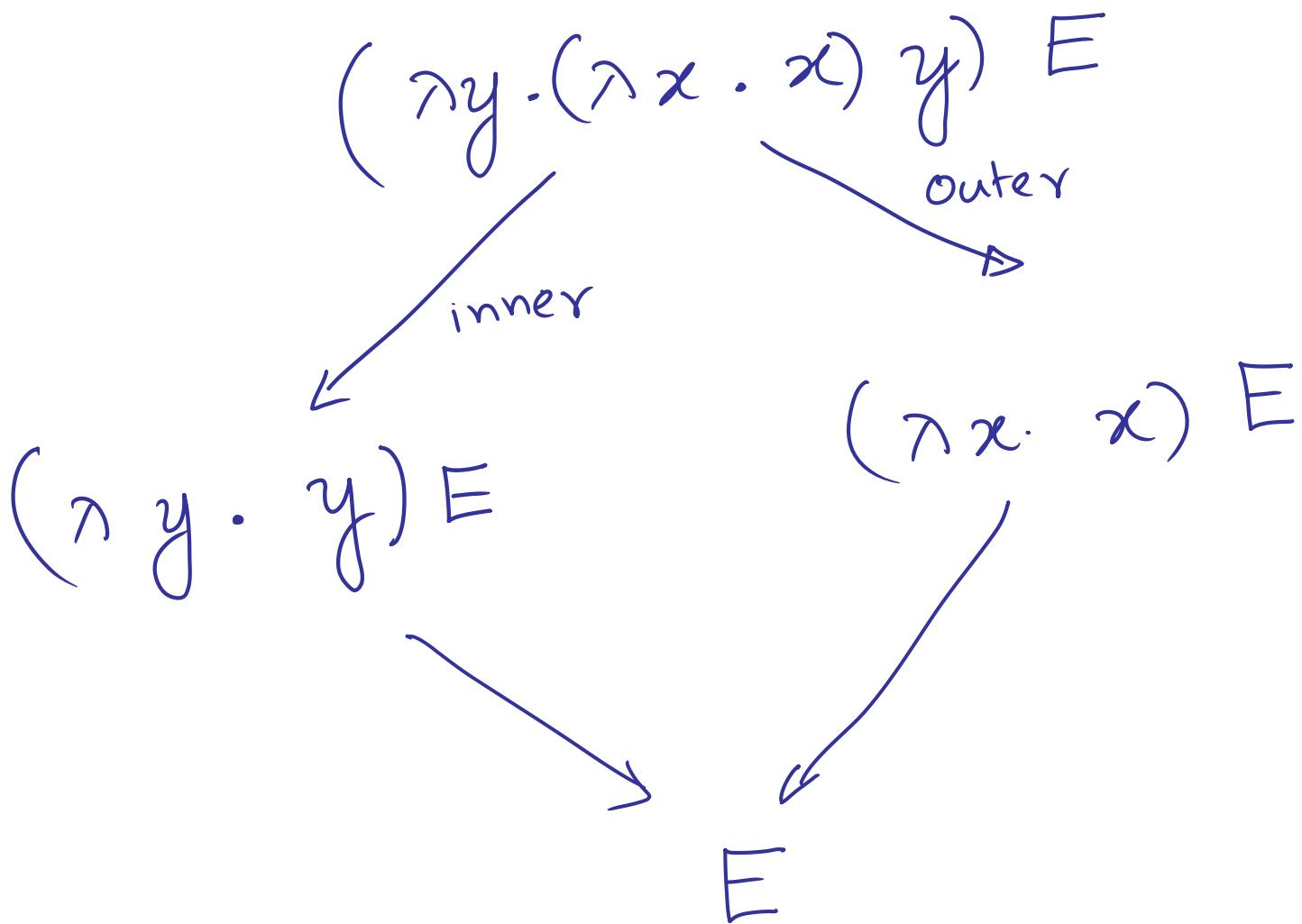
$$[x \mapsto s] (\lambda y \cdot e) = \begin{cases} \lambda y \cdot [x \mapsto s] e & \text{if } y \neq x \wedge y \notin FV(s) \\ \lambda y \cdot e & \text{if } y = x \end{cases}$$

$$[x \mapsto s] (e_1 e_2) = ([x \mapsto s] e_1) ([x \mapsto s] e_2)$$

eg 2:

$$[x \mapsto z] (\lambda z \cdot x) =$$
$$[x \mapsto z] (\lambda w \cdot x) = (\lambda w \cdot z) \checkmark$$

More than one β -reduction sequence possible



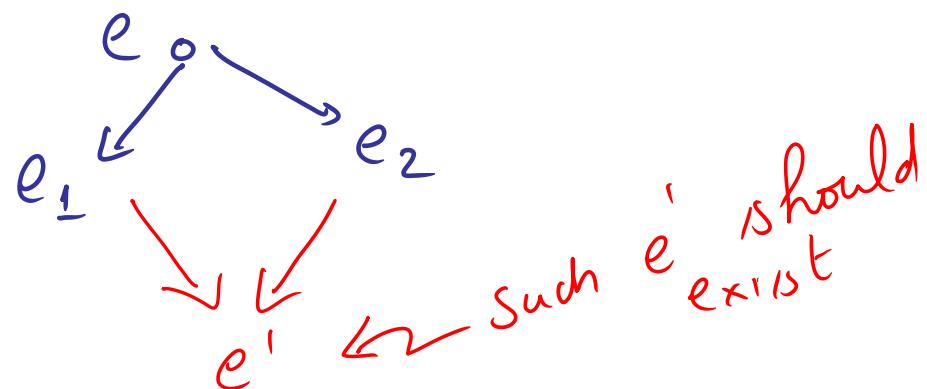
A relation \rightarrow has diamond property if

$\forall e_1, e_2, e \ \text{s.t.}$

$$e \rightarrow e_1$$

$$e \rightarrow e_2$$

$\exists e' \text{ s.t. } e_1 \rightarrow e' \text{ and } e_2 \rightarrow e'$



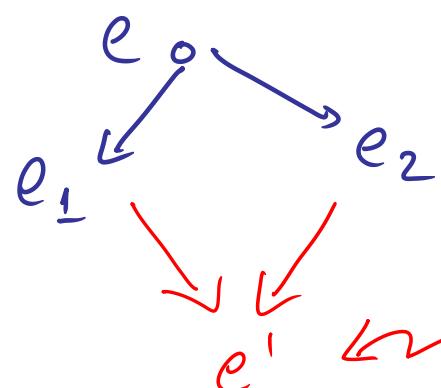
A relation \rightarrow has diamond property if

$\forall e_1, e_2, e \text{ s.t.}$

$$e \rightarrow e_1$$

$$e \rightarrow e_2$$

$\exists e' \text{ s.t. } e_1 \rightarrow e'$ and $e_2 \rightarrow e'$



such e' exist

\rightarrow_B does not satisfy
diamond property

\rightarrow_B^* satisfies diamond property

should

Also called
CHURCH-ROSSER
Thm

Normal form

- A term without β -redexes is in normal form
- β -reduction stops at normal form
- Church-Rosser thm says that independent of reduction strategy we will not find more than one normal form.
 - ... but.. some reduction strategies might fail to find a normal form

$$(\lambda x.y)((\lambda y.y\ y)(\lambda y.y\bar{y}))$$

$$\xrightarrow{\beta} (\lambda x.y) \underline{((\lambda y.y\bar{y})(\lambda y.y\bar{y}))}$$

$\xrightarrow{\beta} \dots$

But ...

$$(\lambda x.y)((\lambda y.y\ y)(\lambda y.y\ y))$$

$$\xrightarrow{\beta} (\lambda x.y) \underline{((\lambda y.y\ y)(\lambda y.y\ y))}$$

$$\xrightarrow{\beta} \dots$$

$$(\lambda x.y) \underline{((\lambda y.y\ y)(\lambda y.y\ y))}$$

$$\xrightarrow{\beta} y$$

reduction strategies

- Normal order - no reduction under \rightarrow
leftmost outermost redex is reduced.

Thm:

If e has normal form e' , then
normal order reduction will reduce e to e'

Call by name

Two rules:

- No reduction inside a λ
- Don't evaluate the argument of a function

$$\frac{e_1 \xrightarrow{*_n} \lambda x. e' \quad [x \mapsto e_2] e' \xrightarrow{*_n} e}{e_1, e_2 \xrightarrow{*_n} e}$$

$$\lambda x. e \xrightarrow{*_n} \lambda x. e$$

Demand driven, expression not evaluated
unless it is needed

$$(\lambda y. (\lambda x. x) y) ((\lambda u. u) (\lambda v. v)) \xrightarrow{\beta_n}$$

$$\underline{(\lambda y. (\lambda x. x) y)} ((\lambda u. u) (\lambda v. v)) \xrightarrow{\beta_n}$$

$$\underline{(\lambda y. y)} ((\lambda u. u) (\lambda v. v)) \xrightarrow{\beta_n}$$

$$(\lambda u. u) (\lambda v. v) \xrightarrow{\beta_n}$$

$$(\lambda v. v)$$

Call by value

Two rules:

- No reduction inside a λ
- DO evaluate the argument if \in function

$$\begin{array}{c} e_1 \xrightarrow{*_v} \lambda x. e'_1 \quad e_2 \xrightarrow{*_v} e'_2 \\ [e'_2 \mapsto x] e'_1 \xrightarrow{*_v} e \end{array}$$

$$\lambda x. e \xrightarrow{*_v} \lambda x. e$$

$$e_1 e_2 \xrightarrow{*_v} e$$

Most languages are call by value -

$$(\lambda y. (\lambda x. x) y) ((\lambda u. u) (\lambda v. v)) \xrightarrow{\beta_n}$$

$$(\lambda y. (\lambda x. x) y) \underline{((\lambda u. u) (\lambda v. v))} \xrightarrow{\beta_v}$$

$$\underline{(\lambda y. (\lambda x. x) y)} (\lambda v. v) \xrightarrow{\beta_v}$$

$$\underline{(\lambda x. x) (\lambda v. v)} \xrightarrow{\beta_v}$$

$\lambda v. v$

Programming in the λ -calculus

- λ -calculus is expressive enough to encode turing machines

- Let $=_\beta^2$ be defined as reflexive, symmetric, transitive closure of \rightarrow_β

$e =_\beta^2 e'$ is undecidable

Take away from today's class

- New programming language – Lambda calculus
- Syntax
- Scope
- Renaming and Reduction
- Call by value and name

Next class

How to express

- Boolean expression
- Numbers
- addition
- multiplication
- recursion



BITS Pilani
Pilani Campus

Principles of Programming Language

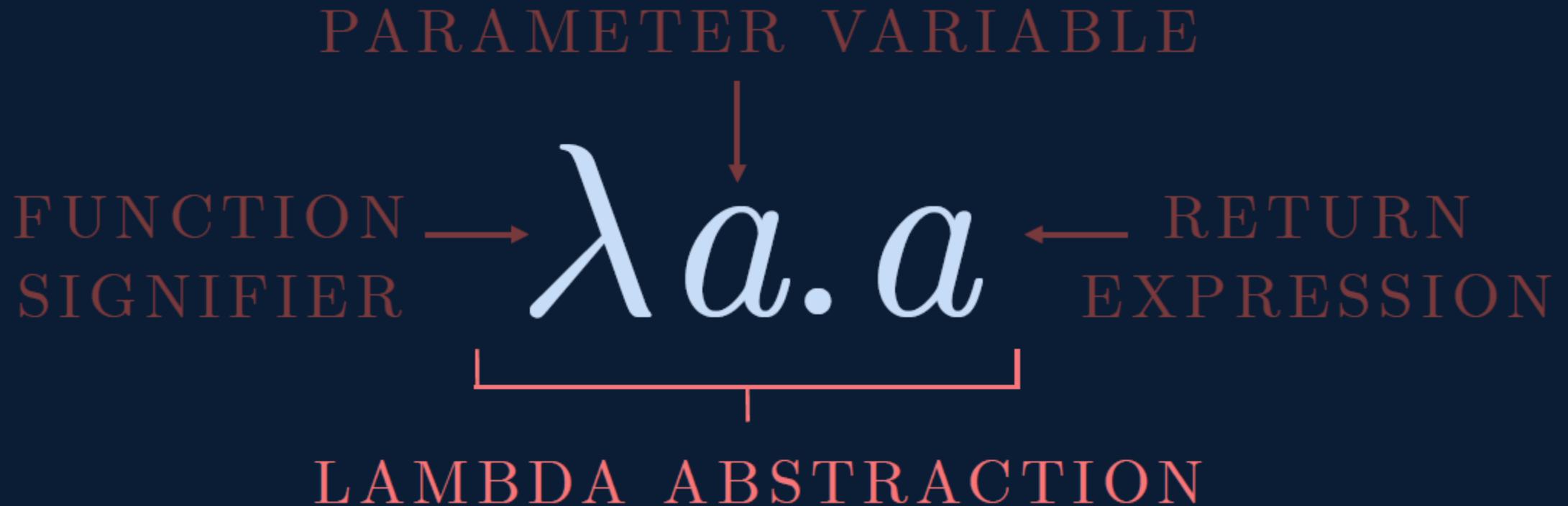
Amit Dua
Computer Science and Information Systems Department
BITS, Pilani





Lambda Calculus

Lecture slides courtesy Prof. Sriram Rajamani, Prof. Adam Doupe, and Prof. Gabriel Lebec
Lecture material from Reference book titled “Types and programming languages” by Benjamin C. Pierce



λ -CALCULUS SYNTAX

expression ::= variable	<i>identifier</i>
expression expression	<i>application</i>
λ variable . expression	<i>abstraction</i>
(expression)	<i>grouping</i>

Disambiguation Rules

- $E \rightarrow E E$ is left associative
 - $x y z$ is
 - $(x y) z$
 - $w x y z$ is
 - $((w x) y) z$
- $\lambda ID . E$ extends as far to the right as possible, starting with the λ ID
 - .
 - $\lambda x . x y$ is
 - $\lambda x . (x y)$
 - $\lambda x . \lambda x . x$ is
 - $\lambda x . (\lambda x . x)$

APPLICATIONS

$f\ a$

$\mathbf{f}(\mathbf{a})$

$f\ a\ b$

$\mathbf{f}(\mathbf{a})(\mathbf{b})$

$(f\ a)\ b$

$(\mathbf{f}(\mathbf{a}))(\mathbf{b})$

$f(a\ b)$

$\mathbf{f}(\mathbf{a}(\mathbf{b}))$

β -REDUCTION

$$(((\lambda a.a)\lambda b.\lambda c.b)(x)\lambda e.f$$

β -REDUCTION*

$$\begin{aligned} & ((\lambda a.a)\lambda b.\lambda c.b)(x)\lambda e.f \\ = & (\lambda b.\lambda c.b) \quad (x)\lambda e.f \\ = & (\lambda c.x) \quad \lambda e.f \\ = & x \end{aligned}$$

β -NORMAL FORM

COMBINATORS

functions with no free variables

$\lambda b.b$ combinator

$\lambda b.a$ not a combinator

$\lambda ab.a$ combinator

$\lambda a.ab$ not a combinator

$\lambda abc.c(\lambda e.b)$ combinator

Haskell Example

```
fType :: Int -> Int -> Int  
fType x y = x*x + y*y  
main = print (fType 2 4)
```

Haskell Example

```
add :: Integer -> Integer -> Integer      --function declaration  
add x y = x + y                            --function definition  
main = do  
    putStrLn "The addition of the two numbers is:"  
    print(add 2 5)                          --calling a function
```

Take away from Last couple of classes

- New programming language – Lambda calculus
- Syntax
- Scope
- Renaming and Reduction
- Call by value and name

Today's class

How to express

- Combinators
- Boolean expression
- Numbers
- addition

Identity Combinator

$I := \lambda a.a$

$I\ x = ?$

$I\ x = x$

$I\ I = ?$

$I\ I = I$

$\lambda a.a$
IDENTITY

$M := \lambda f. ff$

$M I = ?$

$M I = I I = ?$

$M I = I I = I$

$M M = ?$

$M M = M M = ?$

// stack overflow



$\lambda ab.a$

KESTREL

$$\begin{aligned} K &:= \lambda ab.a \\ &= \lambda a.\lambda b.a \end{aligned}$$

$$K M I = ?$$

$$K M I = M$$

$$K I M = ?$$

$$K I M = I$$

$$K I x = I$$

$$K I x y = I y = y$$

KITE Combinator

$$KI := \lambda ab.b \\ = K I$$

$$KI M K = ?$$

$$KI M K = K$$

$$KI K M = ?$$

$$KI K M = M$$



COMBINATORS

Sym.	Bird	λ -Calculus	Use	Haskell
I	Idiot	$\lambda a.a$	identity	<code>id</code>
M	Mockingbird	$\lambda f.f f$	self-application	(cannot define)
K	Kestrel	$\lambda ab.a$	first, const	<code>const</code>
KI	Kite	$\lambda ab.b = \text{KI}$	second	<code>const id</code>

CARDINAL Combinator

$C := \lambda fab.fba$

$C K I M = ?$

$C K I M = M$



COMBINATORS

Sym.	Bird	λ -Calculus	Use	Haskell
I	Idiot	$\lambda a.a$	identity	<code>id</code>
M	Mockingbird	$\lambda f.f\!f$	self-application	(cannot define)
K	Kestrel	$\lambda ab.a$	first, const	<code>const</code>
KI	Kite	$\lambda ab.b = \text{KI} = \text{CK}$	second	<code>const id</code>
C	Cardinal	$\lambda fab.fba$	reverse arguments	<code>flip</code>

Boolean Logic

- $T = (\lambda x . \lambda y . x)$
- $F = (\lambda x . \lambda y . y)$
- $\text{and} = (\lambda a . \lambda b . a b F)$
- $\text{and } T T$
 - $(\lambda a . \lambda b . a b (\lambda x . \lambda y . y))$

and T T

- $(\lambda a . \lambda b . a b (\lambda x . \lambda y . y)) (\lambda x . \lambda y . x) (\lambda x . \lambda y . x)$
- $(\lambda b . a b (\lambda x . \lambda y . y))[a \rightarrow (\lambda x . \lambda y . x)] (\lambda x . \lambda y . x)$
- $(\lambda b . (\lambda x . \lambda y . x) b (\lambda x . \lambda y . y)) (\lambda x . \lambda y . x)$
- $((\lambda x . \lambda y . x) b (\lambda x . \lambda y . y))[b \rightarrow (\lambda x . \lambda y . x)]$
- $(\lambda x . \lambda y . x) (\lambda x . \lambda y . x) (\lambda x . \lambda y . y)$
- $(\lambda y . x)[x \rightarrow (\lambda x . \lambda y . x)] (\lambda x . \lambda y . y)$
- $(\lambda y . (\lambda x . \lambda y . x)) (\lambda x . \lambda y . y)$
- $(\lambda x . \lambda y . x)[y \rightarrow (\lambda x . \lambda y . y)]$
- $(\lambda x . \lambda y . x)$
- T

and T F

- $(\lambda a . \lambda b . a b F) T F$
- $(\lambda b . a b F)[a \rightarrow T] F$
- $(\lambda b . T b F) F$
- $(T b F)[b \rightarrow F]$
- $(T F F)$
- $(\lambda x . \lambda y . x) F F$
- $(\lambda y . x)[x \rightarrow F] F$
- $(\lambda y . F) F$
- $F[y \rightarrow F]$
- F

and F T

- $(\lambda a . \lambda b . a b F) F T$
- $F T F$
- F

and F F

- $(\lambda a . \lambda b . a b F) F F$
- $F F F$
- F

not

- $\text{not } T = F$
- $\text{not } F = T$
- $\text{not} = (\lambda a . a F T)$
- $\text{not } T$
 - $(\lambda a . a F T) T$
 - $T F T$
 - F
- $\text{not } F$
 - $(\lambda a . a F T) F$
 - $F F T$
 - T

If Branches

```
if c then  
    a  
else  
    b
```

- if c a b
- if T a b = a
- if F a b = b
- if = $(\lambda a . a)$

Examples

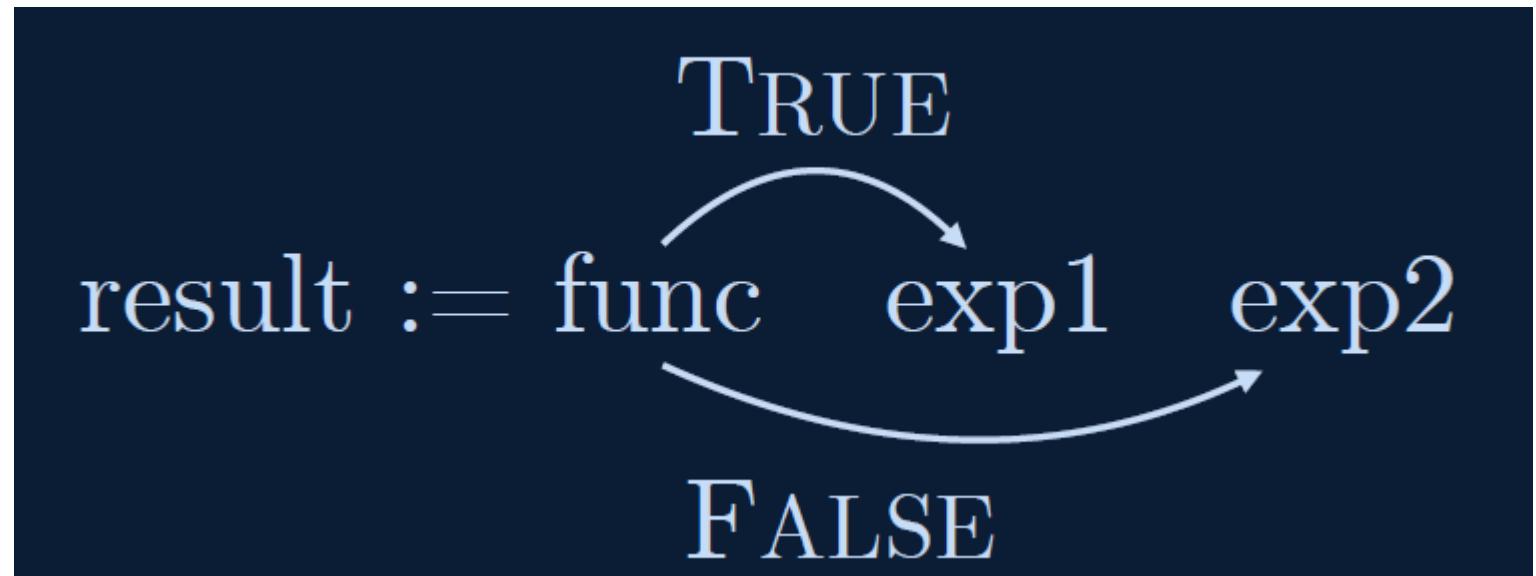
- if T a b
 - $(\lambda a . a) T a b$
 - T a b
 - a
- if F a b
 - $(\lambda a . a) F a b$
 - F a b
 - b

```
const result = bool ? exp1 : exp2
```

```
result := bool ? exp1 : exp2
```

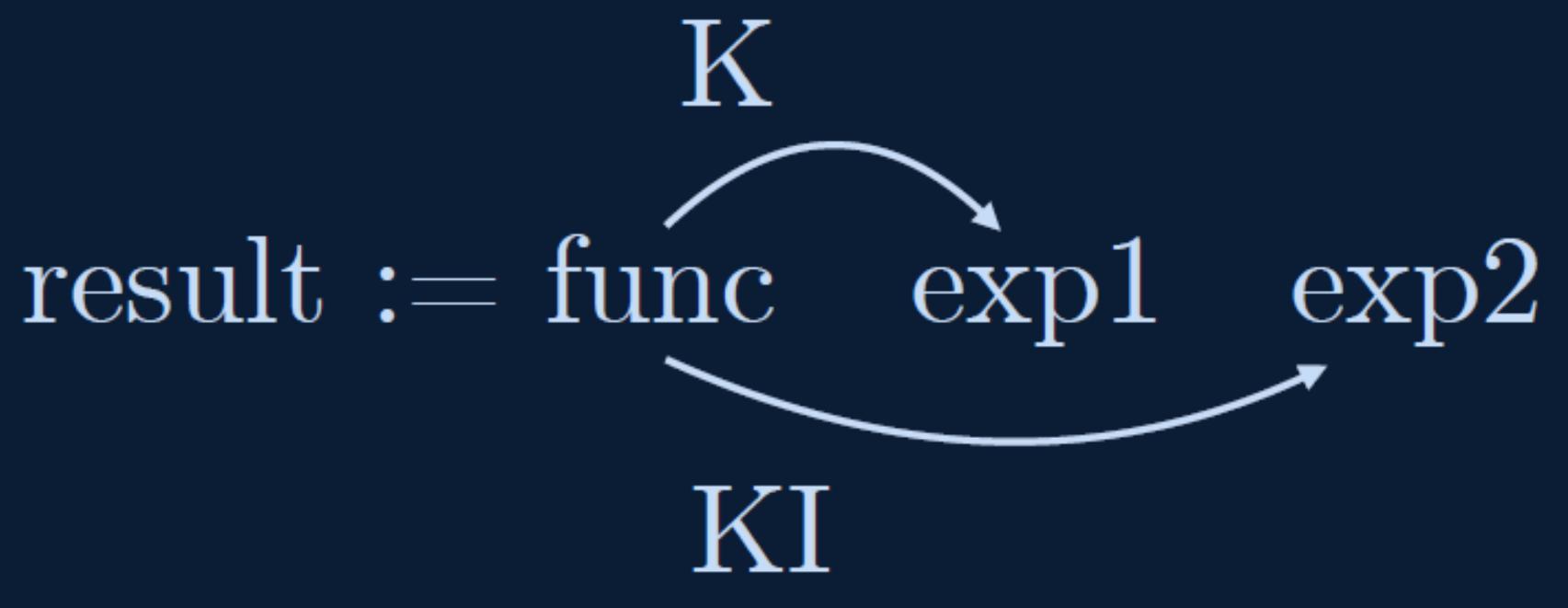
```
result := bool exp1 exp2
```

```
result := func exp1 exp2
```



TRUE := K

FALSE := KI = C K



NOT

NOT := $\lambda p.pFT$



CHURCH ENCODINGS: BOOLEANS

Sym.	Name	λ -Calculus	Use
T	TRUE	$\lambda ab.a = K$	encoding for true
F	FALSE	$\lambda ab.b = KI = CK$	encoding for false
	NOT	$\lambda p.pFT$	negation

$$\lambda ab.a \quad \lambda ba.a$$

$$\text{NOT } K = KI$$

$$\text{NOT } (KI) = K$$

$$\lambda ba.a \quad \lambda ab.a$$

$$C\ K = KI$$

$$C\ (KI) = K$$

$$C\ T = F$$

$$C\ F = T$$

CHURCH ENCODINGS: BOOLEANS

Sym.	Name	λ -Calculus	Use
T	TRUE	$\lambda ab.a = K$	encoding for true
F	FALSE	$\lambda ab.b = KI = CK$	encoding for false
	NOT	$\lambda p.pFT$ or C	negation

Church's Numerals

- $0 = \lambda f. \lambda x. x$
- $1 = \lambda f. \lambda x. fx$
- $2 = \lambda f. \lambda x. f(fx)$
- $3 = \lambda f. \lambda x. f(f(fx))$
- $4 = \lambda f. \lambda x. f(f(f(fx)))$
 - $\lambda f. \lambda x. (f(f(f(fx))))$
- $4 a b$
 - $a(a(a(a b)))$

Successor Function

- $\text{succ} = \lambda n . \lambda f . \lambda x . f(n f x)$
- $0 = \lambda f . \lambda x . x$
- $\text{succ } 0$
 - $(\lambda n . \lambda f . \lambda x . f(n f x)) 0$
 - $\lambda f . \lambda x . f(0 f x)$
 - $\lambda f . \lambda x . f((\lambda f . \lambda x . x) f x)$
 - $\lambda f . \lambda x . fx$
- $1 = \lambda f . \lambda x . fx$
- $\text{succ } 0 = 1$

Successor Function

- $\text{succ} = \lambda n . \lambda f . \lambda x . f(n f x)$
- $1 = \lambda f . \lambda x . f x$
- $\text{succ } 1$
 - $(\lambda n . \lambda f . \lambda x . f(n f x)) 1$
 - $\lambda f . \lambda x . f(1 f x)$
 - $\lambda f . \lambda x . f((\lambda f . \lambda x . f x) f x)$
 - $\lambda f . \lambda x . f(f x)$
- $2 = \lambda f . \lambda x . f(f x)$
- $\text{succ } 1 = 2$
- $\text{succ } n = n + 1$

Addition

- add 0 1 = 1
- add 1 2 = 3
- add = $\lambda n . \lambda m . \lambda f . \lambda x . n f (m f x)$
- add 0 1
 - $(\lambda n . \lambda m . \lambda f . \lambda x . n f (m f x)) 0 1$
 - $(\lambda m . \lambda f . \lambda x . 0 f (m f x)) 1$
 - $\lambda f . \lambda x . 0 f (1 f x)$
 - $\lambda f . \lambda x . 0 f (f x)$
 - $\lambda f . \lambda x . f x$

Addition

- $\text{add} = \lambda n . \lambda m . \lambda f . \lambda x . n f (m f x)$
- $\text{add } 1\ 2$
 - $(\lambda n . \lambda m . \lambda f . \lambda x . n f (m f x))\ 1\ 2$
 - $(\lambda m . \lambda f . \lambda x . 1 f (m f x))\ 2$
 - $\lambda f . \lambda x . 1 f (2 f x)$
 - $\lambda f . \lambda x . 1 f (f (f x))$
 - $\lambda f . \lambda x . (f (f (f x)))$
 - 3

Next classes

How to express

- Combinators ✓
- Boolean expression ✓
- Numbers ✓
- addition ✓
- multiplication
- recursion



BITS Pilani
Pilani Campus

Principles of Programming Language

Amit Dua
Computer Science and Information Systems Department
BITS, Pilani





Lambda Calculus

Lecture slides courtesy Prof. Sriram Rajamani, Prof. Adam Doupe, and Prof. Gabriel Lebec
Lecture material from Reference book titled “Types and programming languages” by Benjamin C. Pierce

λ -CALCULUS SYNTAX

expression ::= variable	<i>identifier</i>
expression expression	<i>application</i>
λ variable . expression	<i>abstraction</i>
(expression)	<i>grouping</i>

Disambiguation Rules

- $E \rightarrow E E$ is left associative
 - $x y z$ is
 - $(x y) z$
 - $w x y z$ is
 - $((w x) y) z$
- $\lambda ID . E$ extends as far to the right as possible, starting with the λ ID
 - .
 - $\lambda x . x y$ is
 - $\lambda x . (x y)$
 - $\lambda x . \lambda x . x$ is
 - $\lambda x . (\lambda x . x)$

Take away from Last couple of classes

- New programming language – Lambda calculus
- Syntax
- Scope
- Renaming and Reduction
- Call by value and name

Combinators

$\lambda a. a$
IDENTITY

$\lambda f. ff$
MOCKINGBIRD

$\lambda ab. a$
KESTREL

$\lambda ab. b$
KITE

$\lambda fab. fba$
CARDINAL

COMBINATORS

Sym.	Bird	λ -Calculus	Use	Haskell
I	Idiot	$\lambda a.a$	identity	<code>id</code>
M	Mockingbird	$\lambda f.f\!f$	self-application	(cannot define)
K	Kestrel	$\lambda ab.a$	first, const	<code>const</code>
KI	Kite	$\lambda ab.b = \text{KI} = \text{CK}$	second	<code>const id</code>
C	Cardinal	$\lambda fab.fba$	reverse arguments	<code>flip</code>

Boolean Logic

- $T = (\lambda x . \lambda y . x)$
- $F = (\lambda x . \lambda y . y)$
- $\text{and} = (\lambda a . \lambda b . a b F)$
- $\text{and } T T$
 - $(\lambda a . \lambda b . a b (\lambda x . \lambda y . y))$

not

- $\text{not } T = F$
- $\text{not } F = T$
- $\text{not} = (\lambda a . a F T)$
- $\text{not } T$
 - $(\lambda a . a F T) T$
 - $T F T$
 - F
- $\text{not } F$
 - $(\lambda a . a F T) F$
 - $F F T$
 - T

If Branches

```
if c then  
    a  
else  
    b
```

- if c a b
- if T a b = a
- if F a b = b
- if = $(\lambda a . a)$

Examples

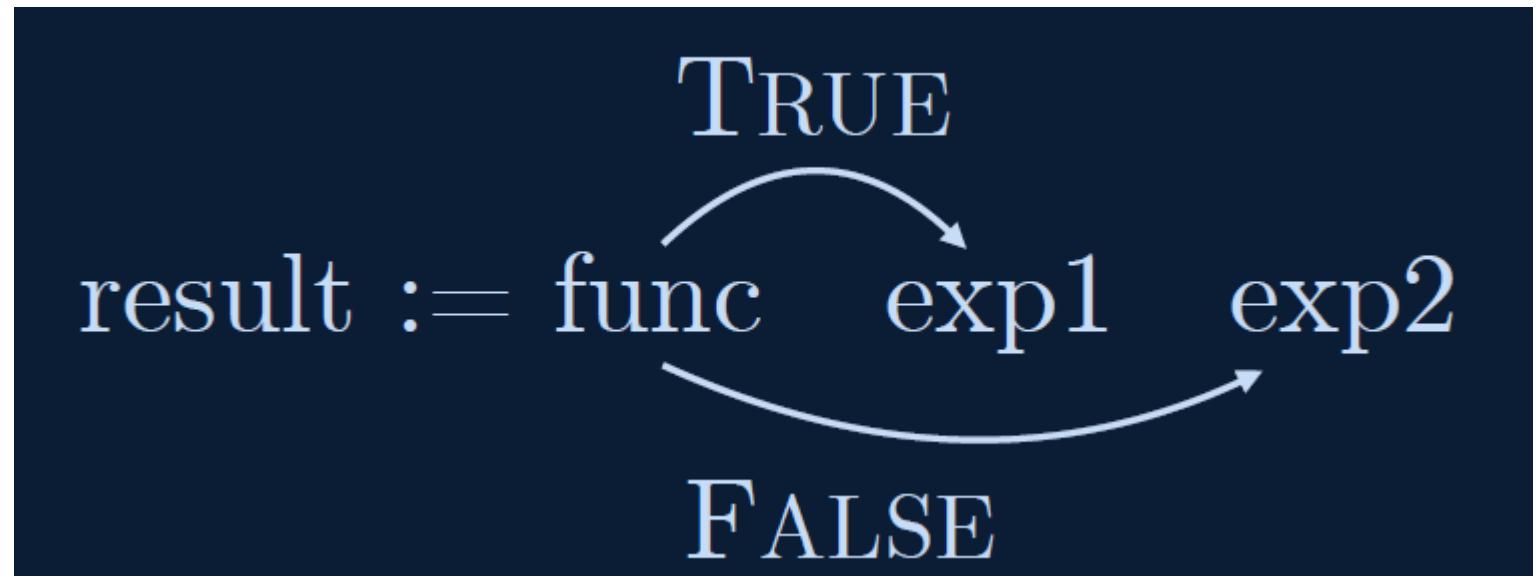
- if T a b
 - $(\lambda a . a) T a b$
 - T a b
 - a
- if F a b
 - $(\lambda a . a) F a b$
 - F a b
 - b

```
const result = bool ? exp1 : exp2
```

```
result := bool ? exp1 : exp2
```

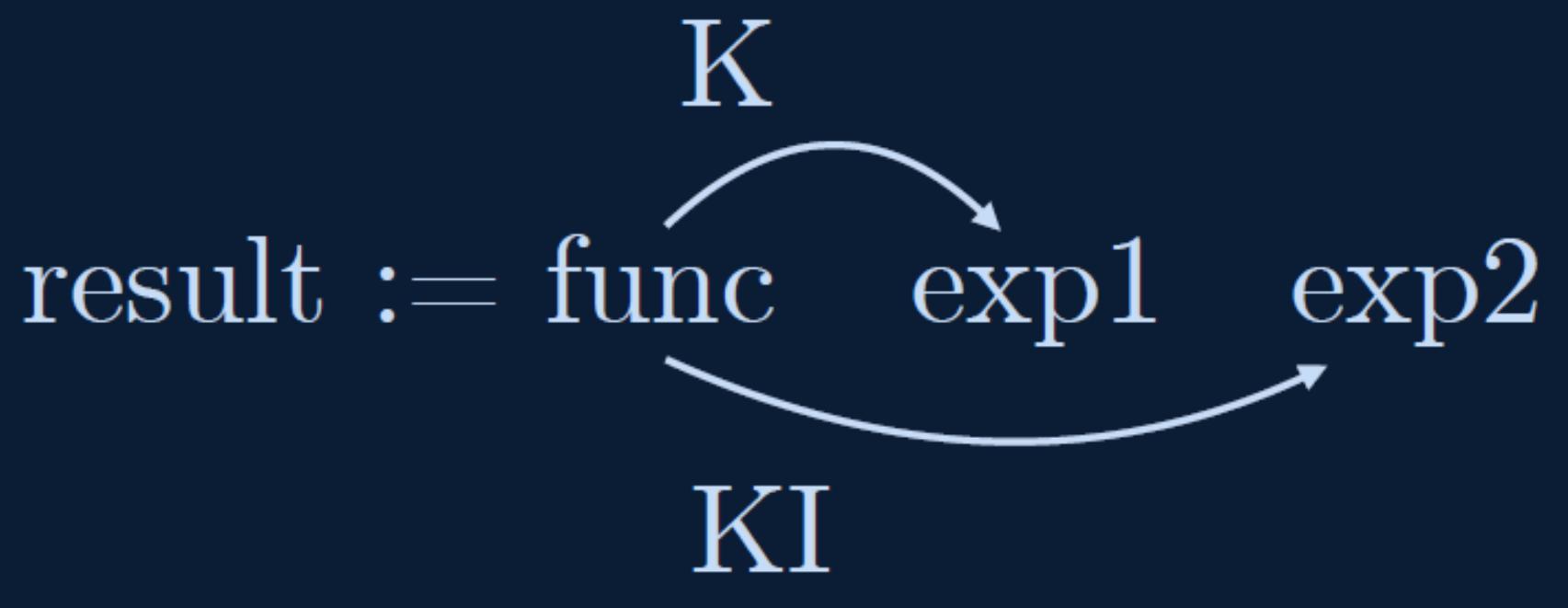
```
result := bool exp1 exp2
```

```
result := func exp1 exp2
```



TRUE := K

FALSE := KI = C K



NOT

NOT := $\lambda p.pFT$



CHURCH ENCODINGS: BOOLEANS

Sym.	Name	λ -Calculus	Use
T	TRUE	$\lambda ab.a = K$	encoding for true
F	FALSE	$\lambda ab.b = KI = CK$	encoding for false
	NOT	$\lambda p.pFT$	negation

$$\lambda ab.a \quad \lambda ba.a$$

$$\text{NOT } K = KI$$

$$\text{NOT } (KI) = K$$

$$\lambda ba.a \quad \lambda ab.a$$

$$C\ K = KI$$

$$C\ (KI) = K$$

$$C\ T = F$$

$$C\ F = T$$

CHURCH ENCODINGS: BOOLEANS

Sym.	Name	λ -Calculus	Use
T	TRUE	$\lambda ab.a = K$	encoding for true
F	FALSE	$\lambda ab.b = KI = CK$	encoding for false
	NOT	$\lambda p.pFT$ or C	negation

Church's Numerals

- $0 = \lambda f. \lambda x. x$
- $1 = \lambda f. \lambda x. fx$
- $2 = \lambda f. \lambda x. f(fx)$
- $3 = \lambda f. \lambda x. f(f(fx))$
- $4 = \lambda f. \lambda x. f(f(f(fx)))$
 - $\lambda f. \lambda x. (f(f(f(fx))))$
- $4 a b$
 - $a(a(a(ab)))$

Successor Function

- $\text{succ} = \lambda n . \lambda f . \lambda x . f(n f x)$
- $0 = \lambda f . \lambda x . x$
- $\text{succ } 0$
 - $(\lambda n . \lambda f . \lambda x . f(n f x)) 0$
 - $\lambda f . \lambda x . f(0 f x)$
 - $\lambda f . \lambda x . f((\lambda f . \lambda x . x) f x)$
 - $\lambda f . \lambda x . fx$
- $1 = \lambda f . \lambda x . fx$
- $\text{succ } 0 = 1$

Successor Function

- $\text{succ} = \lambda n . \lambda f . \lambda x . f(n f x)$
- $1 = \lambda f . \lambda x . f x$
- $\text{succ } 1$
 - $(\lambda n . \lambda f . \lambda x . f(n f x)) 1$
 - $\lambda f . \lambda x . f(1 f x)$
 - $\lambda f . \lambda x . f((\lambda f . \lambda x . f x) f x)$
 - $\lambda f . \lambda x . f(f x)$
- $2 = \lambda f . \lambda x . f(f x)$
- $\text{succ } 1 = 2$
- $\text{succ } n = n + 1$

Addition

- add 0 1 = 1
- add 1 2 = 3
- add = $\lambda n . \lambda m . \lambda f . \lambda x . n f (m f x)$
- add 0 1
 - $(\lambda n . \lambda m . \lambda f . \lambda x . n f (m f x)) 0 1$
 - $(\lambda m . \lambda f . \lambda x . 0 f (m f x)) 1$
 - $\lambda f . \lambda x . 0 f (1 f x)$
 - $\lambda f . \lambda x . 0 f (f x)$
 - $\lambda f . \lambda x . f x$

Addition

- $\text{add} = \lambda n . \lambda m . \lambda f . \lambda x . n f (m f x)$
- $\text{add } 1 \ 2$
 - $(\lambda n . \lambda m . \lambda f . \lambda x . n f (m f x)) \ 1 \ 2$
 - $(\lambda m . \lambda f . \lambda x . 1 f (m f x)) \ 2$
 - $\lambda f . \lambda x . 1 f (2 f x)$
 - $\lambda f . \lambda x . 1 f (f (f x))$
 - $\lambda f . \lambda x . (f (f (f x)))$
 - 3

So Far

How to express

- Combinators ✓
- Boolean expression ✓
- Numbers ✓
- addition ✓
- multiplication
- recursion

Multiplication

- $\text{mult } 0 \ 1 = 0$
- $\text{mult } 1 \ 2 = 2$
- $\text{mult } 2 \ 5 = 10$
- $\text{mult} = \lambda n . \lambda m . m (\text{add } n) 0$
- $\text{mult } 0 \ 1$
 - $(\lambda n . \lambda m . m (\text{add } n) 0) \ 0 \ 1$
 - $(\lambda m . m (\text{add } 0) 0) \ 1$
 - $1 (\text{add } 0) 0$
 - $\text{add } 0 \ 0$
 - 0

Multiplication

- mult 1 2
 - $(\lambda n . \lambda m . m (\text{add } n) 0) 1 2$
 - $(\lambda m . m (\text{add } 1) 0) 2$
 - $2 (\text{add } 1) 0$
 - $(\text{add } 1) ((\text{add } 1) 0)$
 - $(\text{add } 1) (\text{add } 1 0)$
 - $(\text{add } 1) (1)$
 - $(\text{add } 1 1)$
 - 2

Factorial

- $n!$
 - $\text{fact}(0) = 1$
 - $\text{fact}(n) = n * \text{fact}(n-1)$

```
int fact(int n)
{
    if (n == 0)
    {
        return 1;
    }
    return n * fact(n-1);
}
```

Factorial

- (assuming that we have definitions of the iszero and pred functions)
- $\text{fact} = (\lambda n . \text{if } (\text{iszero } n) (1) (\text{mult } n (\text{fact } (\text{pred } n))))$
- However, we cannot write this function!

Y Combinator

- $Y = (\lambda x . \lambda y . y (x x y)) (\lambda x . \lambda y . y (x x y))$
- $Y \text{ foo}$
 - $(\lambda x . \lambda y . y (x x y)) (\lambda x . \lambda y . y (x x y)) \text{ foo}$
 - $(\lambda y . y ((\lambda x . \lambda y . y (x x y)) (\lambda x . \lambda y . y (x x y)) y)) \text{ foo}$
 - $\text{foo } ((\lambda x . \lambda y . y (x x y)) (\lambda x . \lambda y . y (x x y)) \text{ foo})$
 - $\text{foo } (Y \text{ foo})$
 - $\text{foo } (\text{foo } (Y \text{ foo}))$
 - $\text{foo } (\text{foo } (\text{foo } (Y \text{ foo})))$
 - ...

Recursion

- $\text{fact} = (\lambda n . \text{if } (\text{iszero } n) (1) (\text{mult } n (\text{fact } (\text{pred } n)))$
- $\text{fact} = Y (\lambda f . \lambda n . \text{if } (\text{iszero } n) (1) (\text{mult } n (f (\text{pred } n)))$
- **fact 1**
 - $Y (\lambda f . \lambda n . \text{if } (\text{iszero } n) (1) (\text{mult } n (f (\text{pred } n))) 1$
 - $(\lambda f . \lambda n . \text{if } (\text{iszero } n) (1) (\text{mult } n (f (\text{pred } n))) (Y (\lambda f . \lambda n . \text{if } (\text{iszero } n) (1) (\text{mult } n (f (\text{pred } n)))) 1$
 - $(\lambda n . \text{if } (\text{iszero } n) (1) (\text{mult } n ((Y (\lambda f . \lambda n . \text{if } (\text{iszero } n) (1) (\text{mult } n (f (\text{pred } n)))) (\text{pred } n))) 1$
 - $\text{if } (\text{iszero } 1) (1) (\text{mult } 1 ((Y (\lambda f . \lambda n . \text{if } (\text{iszero } n) (1) (\text{mult } n (f (\text{pred } n)))) (\text{pred } 1)))$
 - $\text{if } F (1) (\text{mult } 1 ((Y (\lambda f . \lambda n . \text{if } (\text{iszero } n) (1) (\text{mult } n (f (\text{pred } n)))) (\text{pred } 1)))$
 - $\text{mult } 1 ((Y (\lambda f . \lambda n . \text{if } (\text{iszero } n) (1) (\text{mult } n (f (\text{pred } n)))) (\text{pred } 1))$
 - $\text{mult } 1 (\lambda f . \lambda n . \text{if } (\text{iszero } n) (1) (\text{mult } n (f (\text{pred } n))) (Y (\lambda f . \lambda n . \text{if } (\text{iszero } n) (1) (\text{mult } n (f (\text{pred } n)))) (\text{pred } 1))$
 - $\text{mult } 1 (\lambda n . \text{if } (\text{iszero } n) (1) (\text{mult } n ((Y (\lambda f . \lambda n . \text{if } (\text{iszero } n) (1) (\text{mult } n (f (\text{pred } n)))) (\text{pred } n))) (\text{pred } 1))$
 - $\text{mult } 1 (\lambda n . \text{if } (\text{iszero } n) (1) (\text{mult } n ((Y (\lambda f . \lambda n . \text{if } (\text{iszero } n) (1) (\text{mult } n (f (\text{pred } n)))) (\text{pred } n)))) 0$
 - $\text{mult } 1 (\text{if } (\text{iszero } 0) (1) (\text{mult } 0 ((Y (\lambda f . \lambda n . \text{if } (\text{iszero } n) (1) (\text{mult } n (f (\text{pred } n)))) (\text{pred } 0))))$
 - $\text{mult } 1 (\text{if } T (1) (\text{mult } 0 ((Y (\lambda f . \lambda n . \text{if } (\text{iszero } n) (1) (\text{mult } n (f (\text{pred } n)))) (\text{pred } 0))))$
 - $\text{mult } 1 1$
 - 1

So

How to express

- Combinators ✓
- Boolean expression ✓
- Numbers ✓
- addition ✓
- multiplication ✓
- recursion ✓



BITS Pilani
Pilani Campus

Principles of Programming Language

Amit Dua
Computer Science and Information Systems Department
BITS, Pilani





Functional programming in Haskell

Agenda

- Currying
- Types in Haskell
- Lists and operations
- Defining functions in haskell
- Polymorphism
- Higher order functions
- Tail recursive functions
- Predicate functions

programs in functional languages

- Program: is a set of rules to transforms inputs to outputs
- Computation : process of application of rules
- Program applies these rules to the input that is given to us in order to produce the output
- Use built in functions to form more complex functions

Currying

- Haskell uses currying
- Every function takes one argument
- A multi argument function is a sequence of single argument function

e.g. $\text{sum } m \ n = m+n$

$((\text{sum } m) \ n)$

We have seen it as application in lambda calculus.

Types in haskell

sum m n = m+n

Currying: ((sum m) n)

Type: Int->(Int-> Int)

Sum m n p = m + n + p

Application: (((sum m) n) p)

Type: Int-> (Int -> (Int ->Int))

Lists

Lists are sequence of values of a uniform type

List of values of type T has type [T]

[1,2,3,4] is a list of Int with type [Int]

[True, False, False, True] is a list of Bool with type [Bool]

[1,2,3,4] :: [Int]

[True, False, False, True] :: [Bool]

Head and tail functions

List is build up from [] (empty list) by append operator :

[1,2,3] is actually 1 : (2 : (3 : []))

1: [2,3] == [1,2,3]

==1:2:3:[]

== 1:2:[3]

any list [x1,x2,x3,...xn] == x0 : (x1 : (x2 : (x3 : (... :(xn: [])...)))

head(x:xs) = x

tail(x:xs) = xs

head returns value and tail returns a list

Access time in lists

Accessing the j th item is proportional to j . Why?

We need to peel off $j-1$ items from the list before accessing the item

Contrasting to arrays where we access in constant time

Lists don't support random access.

Built in functions on list

head, tail, length, sum, reverse,...

init | returns all but the last element

init [1,2,3,4,5] => [1,2,3,4]

init [7] => []

last | returns the last element in the list

last [1,2,3,4,5] => 5

last[7] => 7

Defining functions in haskell

fname :: Type --declaration

fname definition

attach :: [Int] -> [Int] -> [Int]

attach [] l = l

attach (x:xs) l = x : (attach xs l)

attach [3,5] [4,5,6] => [3,5,4,5,6]

built in operator in haskell ++ for list concatenation

[3,5] ++ [4,5,6] => [3,5,4,5,6]

take and drop in haskell

take n l => returns the first n elements of the list l

drop n l => leaves the first n elements of the list l

(take n l) ++ (drop n l) == ?

Polymorphism

- Function that works across multiple types
 - Use type variable to denote flexibility
 - a , b , c are place holders for types
 - [a] is a list of type a
-
- mylength :: [a] -> Int
 - myreverse :: [a] -> [a]
 - head :: [a] -> a
 - myinit :: [a] -> [a]

mylength with polymorphism

mylength :: [Int] -> Int

mylength [] = 0

mylength (x:xs) = 1+ mylength xs

mylength :: [a] -> Int

mylength [] = 0

mylength (x:xs) = 1+ mylength xs

Higher order functions

- We can pass function as an argument
 - $\text{apply } f \ x \Rightarrow f \ x$
 - applies first argument to the second argument
-
- Type of apply ??
 - f is a generic function that is of the form $a \rightarrow b$
-
- $\text{apply} :: (a \rightarrow b) \rightarrow a \rightarrow b$

Higher order functions

e.g. apply plus3 5 => 8

Sorting a list: sort function

- in addition to list provide
- the comparison function either ascending or descending

Factorial function

```
fact :: Int-> Int
```

```
fact n
```

```
  | n <=0 = 1
```

```
  | n >0 = n * (fact (n-1))
```

```
ghci
```

```
:load fact.hs
```

```
fact 100
```

Tail recursive function

fact :: Int-> Int

fact2 :: Int-> Int -> Int

fact n = fact2 n 1

fact2 n acc

| n == 0 = acc

| otherwise = fact2 (n-1) (acc*n)

predicate function

Takes an Input and returns either true or false

```
predicate :: Int -> Bool
```

```
predicate n
| mod n 2 ==0 = True
| otherwise = False
```

```
ghci
:load predicate.hs
predicate 5
predicate 12
```

Defining own Data types in haskell

```
data Day = Sunday|Monday|Tuesday|Wednesday|Thursday|Friday|Saturday  
data Bool = True|False
```

```
data Shape = Circle Float  
           | Square Float  
           | Rectangle Float Float
```

```
area :: Shape -> Float  
area (Circle r)      = 3.14*r*r  
area (Square x)     = x*x  
area (Rectangle l w)= l*w
```



BITS Pilani
Pilani Campus

Principles of Programming Language

Amit Dua
Computer Science and Information Systems Department
BITS, Pilani





Support for Object-Oriented Programming

Courtesy: Dr. Lavika Goel

Refer: Concepts of programming languages by Robert W. Sebesta

Agenda



- Introduction to OO paradigm
- Design Issues in OO programming
- How C++ and Java handle these issues
- Implementing Object-Oriented Constructs

Introduction

Many object-oriented programming (OOP) languages

- Some support procedural and data-oriented programming (e.g., Ada 95+ and C++)
- Some support functional program (e.g., CLOS)
- Newer languages do not support other paradigms but use their imperative structures (e.g., Java and C#)
- Some are pure OOP language (e.g., Smalltalk & Ruby)

Characteristics of OOP Languages

- Abstract Data Types
- Inheritance
- Dynamic Binding

Issues in programming with ADTs

- Reuse of data types [future requirements change and require modification in the code]
 - Student and Faculty ADTs require address details
- Independent type definitions where categories of types are related [Program problem space is difficult to manage and difficult to organize program code]
 - Vehicle and Car ADTs

Inheritance : An OOP Approach

- Offers solutions to both modifications – ADT reuse and Program Organization
- ADTs supported with inheritance
 - Inherit the data
 - Inherit functionality of some existing data type
 - Are allowed to modify some entities
 - Allowed to add new entities [without changes to the reused datatype]

Object Oriented Programming: Terminology

- Classes – **ADTs**
- Instance of classes – **Objects**
- Class defined through inheritance – **Derived class or sub-class**
- Class from which the new class is derived –
parent class or superclass
- Subprograms that define the operations on
objects of a class - **Methods**
- Calls to Methods – **Messages**

Derived (sub) Class

- Parent class can keep some of its variables or methods invisible to the subclass (**private access**)
- Subclass can add new variables and methods.
- Subclass can modify the behavior of one or more of its inherited methods. (**Override** the inherited method, then called as Overridden method)

Object-Oriented Concepts (continued)

There are two kinds of variables in a class:

- **Class variables** - one/class
- **Instance variables** - one/object

Single vs. Multiple Inheritance:

one parent class v/s multiple parent classes.

Disadvantage of Inheritance

- Creates dependencies in an inheritance hierarchy
- Limitation: impossible to increase reusability without creating dependencies

Dynamic Binding

- Dynamic binding of messages to method definitions (dynamic dispatch).
- Class that is able to reference objects of the class and objects of any of its descendants.
- From Overridden methods

```
public class A { draw () }  
public class B extends A { draw () }
```

```
A myA = new A();  
myA.draw();  
myA = new B();  
myA.draw();
```

Dynamic Binding Concepts

- An *abstract method* is one that does not include a definition (it only defines a protocol).
[called as pure virtual method in C++].
- An *abstract class* is one that includes at least one abstract method.
[called as abstract base class in C++].
- An abstract class **cannot be instantiated**.
- Any subclass of an abstract class that is to be instantiated must provide definitions of all the inherited abstract methods.

Design Issues for OOP Languages

The Exclusivity of Objects

Are Subclasses Subtypes?

Single and Multiple Inheritance

Object Allocation and Deallocation

Dynamic and Static Binding

Nested Classes

Initialization of Objects

The Exclusivity of Objects

- Everything is an object
 - Advantage - elegance and purity
 - Disadvantage - slow operations on simple objects
- Everything imperative + Object oriented support
- Primitive scalar types+ all structured types as objects

Are Subclasses Subtypes?

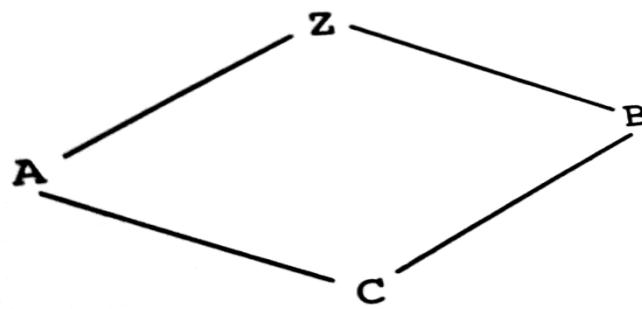
- Does an “**is-a**” relationship hold between a parent class object and an object of the subclass?
- Def: If a derived class is-a parent class, then objects of the derived class must behave the same as the parent class object

```
Subtype Small_int is Integer range -100..100; //Ada
```

- A derived class is a subtype if it has an is-a relationship with its parent class
 - Subclass can only add variables and methods and override inherited methods in “**compatible**” ways
- Call to overriding method can replace call to overridden method

Multiple Inheritance

- Multiple inheritance allows a new class to inherit from two or more classes
 - A and B both define display, C inherits both A and B
 - C needs to reference both versions of display
 - One of them must be overridden in sub class.
- Disadvantages of multiple inheritance:
 - Language and implementation **complexity** (in part due to name collisions)
 - **Potential inefficiency** - dynamic binding is more complex with multiple inheritance

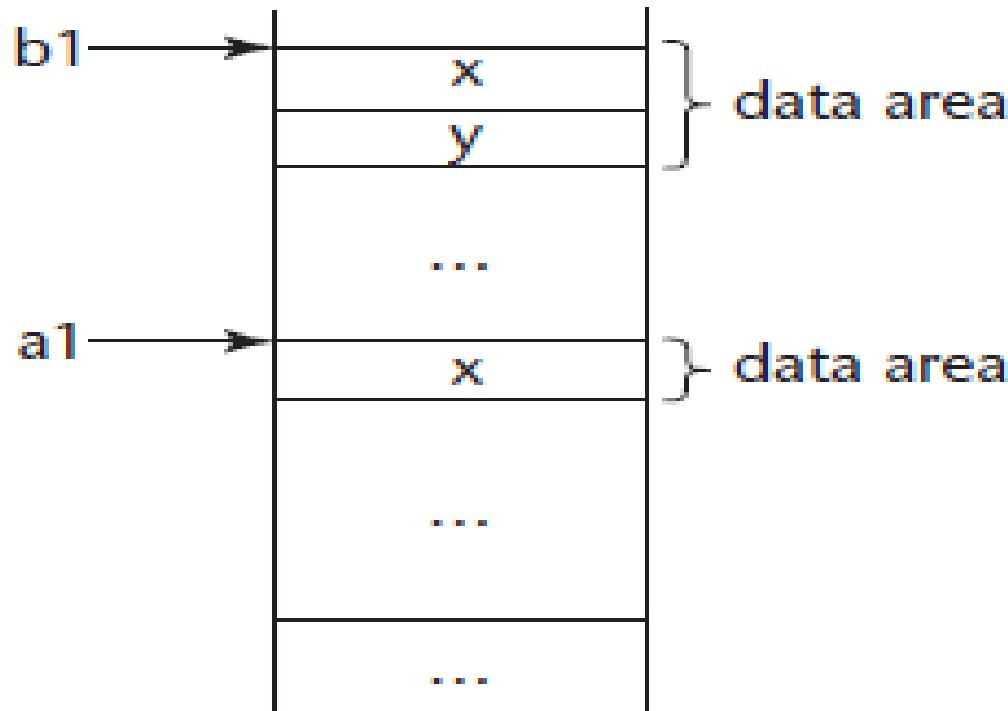


Allocation and Deallocation of Objects: Some Issues

- Each object class gets allocated space to its data members
- Space can be **Run time stack** or the **heap** (using *new* keyword)
- If the space is allocated on **Run time stack**, then if the inherited class has added data variables then the space allocated to the parent class will not be sufficient.
- If the **heap** is used then issue is relating the deallocation-implicit or explicit, and if it is implicit how to reclaim the space not in use and not directly referred.

Object Slicing: Truncated values if $a1 = b1$ where $a1$ and $b1$ are of types A and B

```
class A {  
    int x;  
    ...  
};  
class B : A {  
    int y;  
    ...  
}
```



object slicing occurs when an **object** of a subclass type is copied to an **object** of superclass type:
the superclass copy will not have any of the member variables defined in the subclass.

Dynamic and Static Binding

- Should all binding of messages to methods be dynamic?
 - If none are, you lose the advantages of dynamic binding
 - If all are, it is inefficient
- Allow the user to specify

Nested Classes

- If a new class is needed by only one class, there is no reason to define so it can be seen by other classes
 - Can the new class be nested [inside the class](#) that uses it?
 - In some cases, the new class is [nested inside a subprogram](#) rather than directly in another class

Other issues:

- [Which facilities of the nesting class should be visible to the nested class and vice versa](#)

Initialization of Objects

- Are objects initialized to values when they are created?
 - Implicit or explicit initialization
- How are parent class members initialized when a subclass object is created?



BITS Pilani

Pilani Campus



Support for Object-Oriented Programming in C++

Support for OOP in C++

General Characteristics:

- Evolved from C and SIMULA 67
- Among the most widely used OOP languages
- **Mixed typing system:** Imperative and OOP style
- **Constructors and destructors:** Destructors are used for explicit de-allocation of memory in heap dynamic objects

Support for OOP in C++ (continued)

Inheritance

- A class need not be the subclass of any class
- Access controls for members are
 - Private (visible only in the class and friends)
 - Public (visible in subclasses and clients)
 - Protected (visible in the class and in subclasses, but not clients)

Support for OOP in C++ (continued)

In addition, the subclassing process can be declared with access controls (private or public), which define potential changes in access by subclasses

- **Private derivation** - inherited public and protected members are private in the subclasses
- **Public derivation** - public and protected members are also public and protected in subclasses

Inheritance Example in C++

```
class base_class {  
private:  
    int a;  
    float x;  
protected:  
    int b;  
    float y;  
public:  
    int c;  
    float z;  
};  
  
class subclass_1 : public base_class { ... };  
//      In this one, b and y are protected and  
//      c and z are public  
  
class subclass_2 : private base_class { ... };  
//      In this one, b, y, c, and z are private,  
//      and no derived class of subclass_2 has access to any  
//      member of base_class  
// Data members a and x of base class are not accessible in either subclass_1 or  
// subclass_2.
```

Reexportation in C++

- Private derived classes cannot be subtypes. Why?
- A member that is not accessible in a subclass (because of private derivation) can be declared to be visible there using the scope resolution operator (::), e.g.,

```
class subclass_3 : private base_class {  
    base_class :: c;  
    ...  
}
```

Support for OOP in C++ (continued)

Multiple inheritance is supported

- If there are two inherited members with the same name, they can both be referenced using the scope resolution operator (::)

```
class Thread { ... }
```

```
class Drawing { ... }
```

```
class DrawThread : public Thread, public Drawing { ... }
```

Support for OOP in C++ (continued)

Dynamic Binding

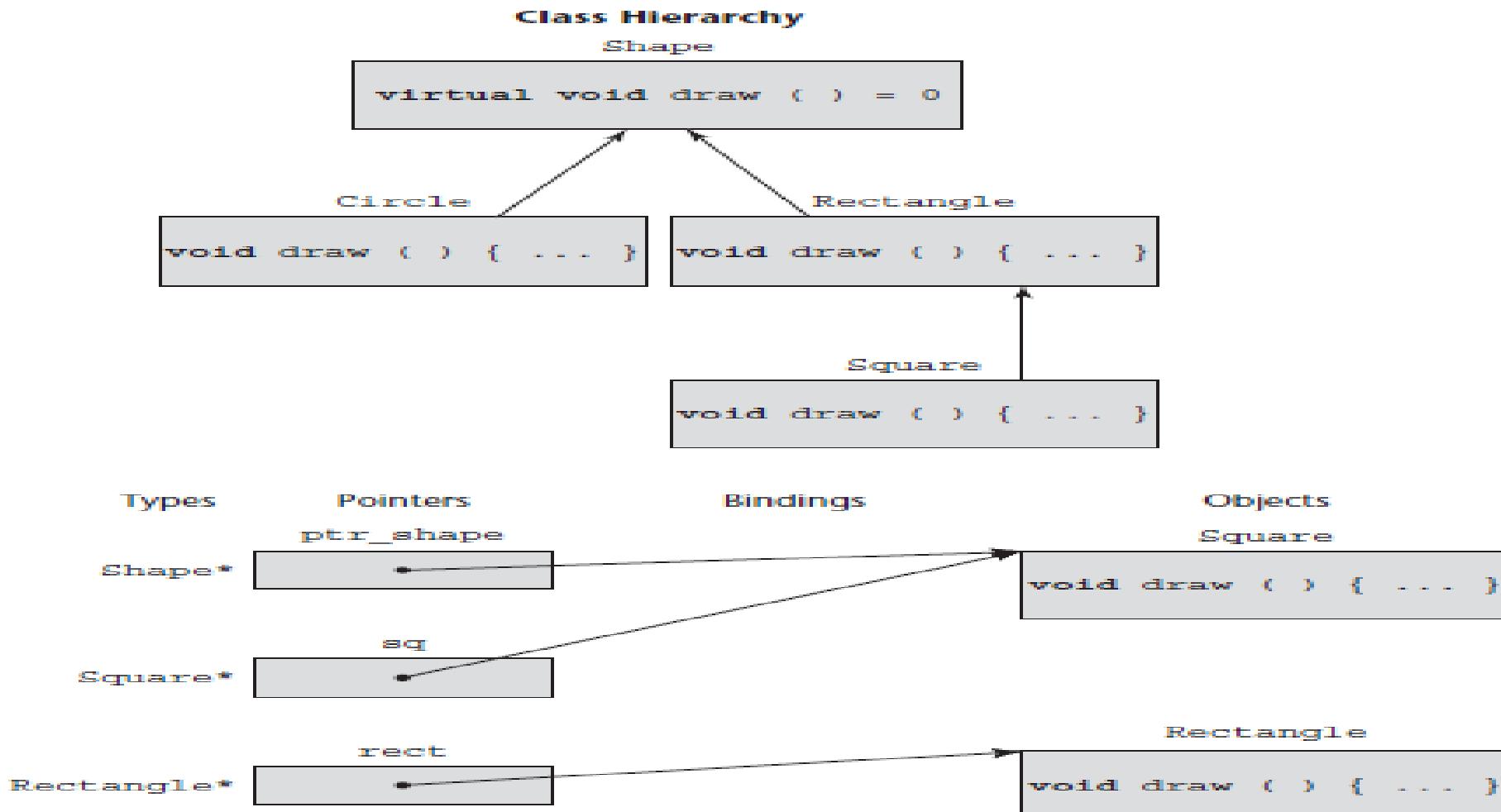
- A method can be defined to be **virtual**, which means that they can be called through polymorphic variables and dynamically bound to messages
- A pure virtual function has no definition at all
- A class that has at least one pure virtual function is **an abstract class**

Support for OOP in C++ (continued)

```
class Shape {  
public:  
    virtual void draw() = 0;  
    ...  
};  
class Circle : public Shape {  
public:  
    void draw() { ... }  
    ...  
};  
class Rectangle : public Shape {  
public:  
    void draw() { ... }  
    ...  
};  
class Square : public Rectangle {  
public:  
    void draw() { ... }  
    ...  
};
```

```
Square* sq = new Square;  
Rectangle* rect = new Rectangle;  
Shape* ptr_shape;  
ptr_shape = sq; // points to a Square object  
ptr_shape ->draw(); // Dynamically  
                    // bound to draw in Square  
rect->draw(); // Statically bound to  
                // draw in Rectangle
```

Dynamic Binding



Support for OOP in C++ (continued)

If objects are allocated from the stack, it is quite different

```
Square sq;      // Allocates a Square object from the stack
Rectangle rect; // Allocates a Rectangle object from the stack
rect = sq;      // Copies the data member values from sq object
rect.draw();    // Calls the draw from Rectangle, rect still references the Rectangle
                // object
```

Support for OOP in C++ (continued)

Evaluation

- C++ provides **extensive access controls** (unlike Smalltalk)
- C++ provides **multiple inheritance**
- In C++, **the programmer must decide** at design time which methods will be statically bound and which must be dynamically bound
 - Static binding is faster!
 - Dynamic binding of C++ is faster than Smalltalk because in C++, binding a virtual member function call in C++ to a function definition has a fixed cost regardless of the distance in the inheritance hierarchy the method definition appears.
- C++ **type checking is static** while in Smalltalk type checking is dynamic
- Because of interpretation and dynamic binding, Smalltalk is ~10 times slower than C++
- Reference assignment for stack dynamic objects and pointer assignment for heap dynamic objects

BITS Pilani

Pilani Campus



Support for Object-Oriented Programming in Java

Support for OOP in Java

General Characteristics

- All data are **objects** except the primitive types (Boolean, character, etc.)
- All primitive types are converted to an object of the **wrapper class** of the primitive value's type that store one data value. This process is called as boxing.
- All Java classes must be subclass of the root class **Object** or some descendant of Object.
- All objects are **heap-dynamic**, allocated with **new** operator but there is no explicit de-allocation operator. Garbage collection is used for storage reclamation.

If an object has access to other resources than heap memory like file or lock on a shared resource, these cannot be reclaimed by garbage collector.

- A **finalize** method (similar to C++ destructor function) is implicitly called when the garbage collector is about to reclaim the storage occupied by the object.

Support for OOP in Java (continued)

Inheritance

- Use of **final class** to indicate that the class cannot be derived (final method means that the method cannot be overridden in a subclass).
- Java does not support private and protected **derivations** of C++.
- **Single inheritance** supported only, but there is an abstract class that provides some of the benefits of **multiple inheritance (interface)**
- An **interface** can include only method declarations and named constants.
- A class **implements** the interface and it can implement any no. of interfaces.
- **Interface is not a replacement for multiple inheritance as it lacks code reuse.**

Support for OOP in Java (continued)

Dynamic Binding

- In Java, all messages are dynamically bound to methods, unless the method is **final** (i.e., it cannot be overridden, therefore dynamic binding serves no purpose).
- Static binding is also used if the methods is **static** or **private** both of which disallow overriding.

Support for OOP in Java (continued)

Evaluation

- Design decisions to support OOP are similar to C++
- No parentless classes
- **Dynamic binding** is used as “normal” way to bind method calls to method definitions
- **Uses interfaces** to provide a simple form of support for multiple inheritance



Implementing Object-Oriented Constructs

Implementing OO Constructs

Two interesting and challenging parts

- Storage structures for instance variables
- Dynamic binding of messages to methods

Instance Data Storage

- Class instance records (CIRs) store the storage structure of the instance variables of a class i.e. state of an object
 - Static (built at compile time)
- If a class has a parent, the subclass instance variables are added to the parent CIR
- Because CIR is static, access to all instance variables is done using constant offsets from the beginning of the CIR instance as it is in records
 - Efficient

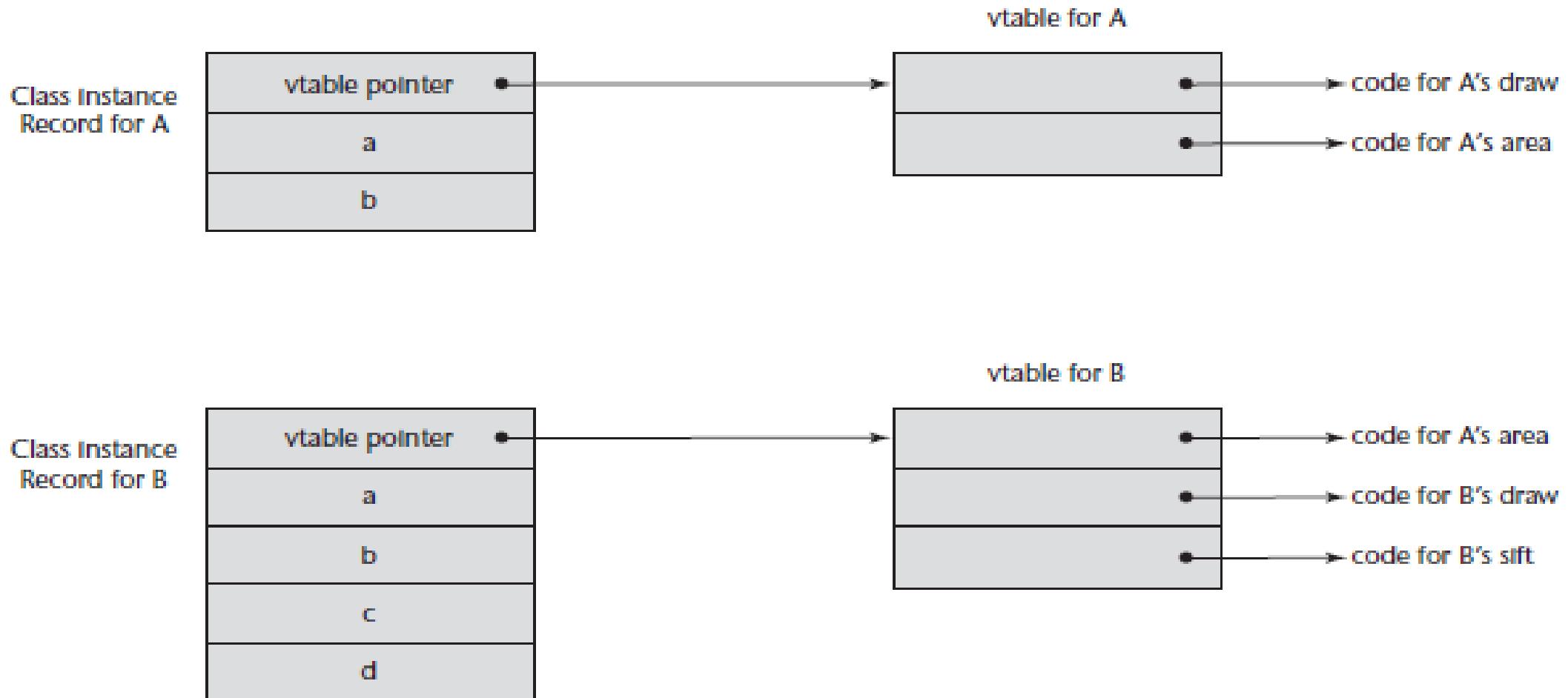
Dynamic Binding of Methods Calls

- Methods in a class that are statically bound need not be involved in the CIR
- Methods that will be dynamically bound must have entries in the CIR
 - Calls to dynamically bound methods can be connected to the corresponding code through a pointer in the CIR
 - The storage structure is sometimes called *virtual method tables* (vtable)
 - Method calls can be represented as **offsets** from the beginning of the vtable
 - Polymorphic variables of a base class always reference the CIR of the correct type object thereby binding to the correct version of the dynamically bound method.

Dynamic Binding of Methods Calls

```
public class A {  
    public int a, b;  
    public void draw() { ... }  
    public int area() { ... }  
}  
  
public class B extends A {  
    public int c, d;  
    public void draw() { ... }  
    public void sift() { ... }  
}
```

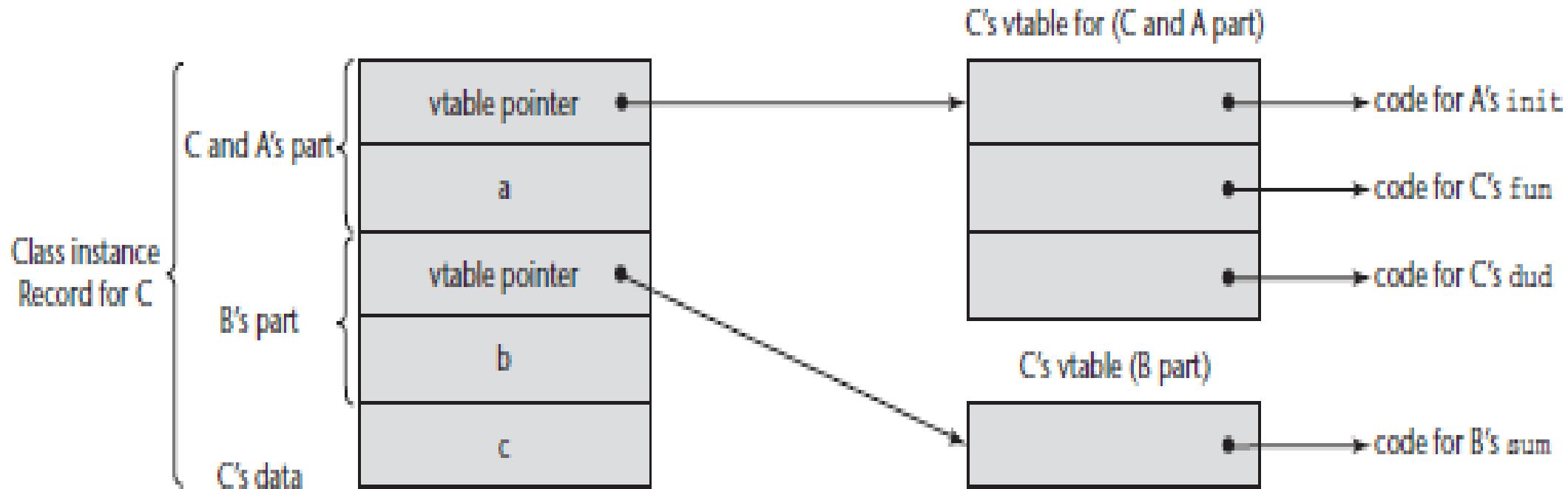
CIR with single inheritance



Multiple inheritance

```
class A {  
public:  
    int a;  
    virtual void fun() { . . . }  
    virtual void init() { . . . }  
};  
class B {  
  
public:  
    int b;  
    virtual void sum() { . . . }  
};  
class C : public A, public B {  
public:  
    int c;  
    virtual void fun() { . . . }  
    virtual void dud() { . . . }  
};
```

CIR with multiple inheritance



Summary

- OO programming involves three fundamental concepts:
 - ADTs,
 - Inheritance,
 - dynamic binding
- Major design issues:
 - exclusivity of objects,
 - subclasses and subtypes, t
 - ype checking and polymorphism,
 - single and multiple inheritance,
 - dynamic binding,
 - explicit and implicit de-allocation of objects, and
 - nested classes



Lecture 26: Logic Programming Languages

Design objectives

- Real world knowledge representation of objects and relationship among objects
- Have constructs which can facilitate reasoning over knowledge
- Constructs for theorem proving and question answering
- Provide extensive support in Artificial Intelligence research

Logic programming

- The program is declarative in nature.
- It consists of two important pieces of knowledge – facts and rules
- A program is used to prove if a statement is true and to answer the queries.
- There are no constructs such as loops, conditional statements or functions.
- The language supports addition of new facts and deletion of old facts.
- The language has inbuilt implementation of the inference engine to process the facts and rules.

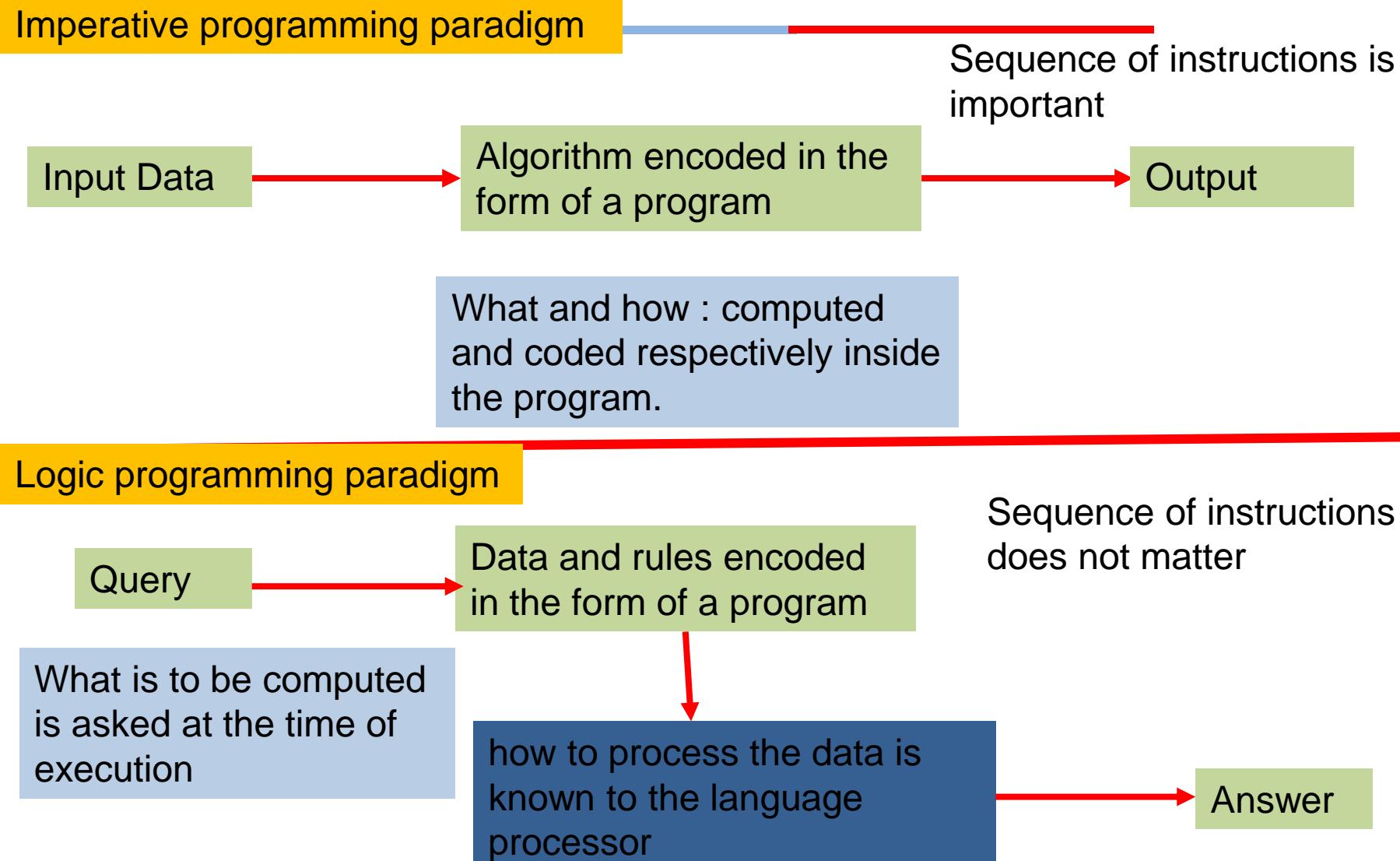
Logic Programming

- The language is declarative as the programs consist of declarations rather than the assignment and control flow statements.
- These declarations are the propositions in symbolic logic.
- The logic programming language uses declarative semantics.
- The declarative semantics is simpler than semantics of imperative languages.
- The meaning of a proposition does not depend on the textual context or execution sequence.

Logic programming

- The programming in a logic programming language is **nonprocedural** unlike the imperative and functional languages.
- The programmer of imperative and functional languages know exactly **what** is to be accomplished and instructs the computer on exactly **how** to achieve that.
- The programs in logic languages do not need to specify exactly **how** a result is to be computed. The language implementation is such that it knows how the result will be calculated.

Imperative Vs. logic program execution



Knowledge processing in logic

- The programmer codes the knowledge of the real world in logic programming language.
- The language implements the algorithm such as **forward chaining**, **backward chaining**, **resolution** etc. internally to process the supplied knowledge.
- **Resolution** is an inference rule that allows inferred propositions to be computed from given propositions.

Comparative ways of thinking in different language paradigms

Insertion sort in Haskell programming language – A functional approach

```

insert :: Int -> [Int] -> [Int]
insert x [] = [x]
insert x (y:ys)
  | x < y = x:y:ys
  | otherwise = y : insert x ys
insertionSort :: [Int] -> [Int]
insertionSort [x] = [x]
insertionSort (x:xs) = insert x (insertionSort xs)

main = do
    print(insertionSort [3, 19, 1, 872, 56, 39, 23, 10, 345, 180, 200, 2, -10])

```

Recursive function call returning a list

Insertion sort in Prolog programming language – A predicate calculus based logical approach

```

insert_sort(List, Sorted):-isort(List, [],Sorted).
isort([], Acc, Acc).
isort([H|T], Acc, Sorted):- insert(H, Acc, New), isort(T, New, Sorted).
insert(X, [Y|T], [Y|New]) :- X>Y, insert(X,T,New).
insert(X, [Y|T], [X,Y|T]) :- X=<Y.
insert(X, [], [X]).
```

Predicate evaluated to true or false

Prolog

- It is a programming language that was developed in 1972.
- It was initially used for natural language processing.
- It is widely used for specifying algorithms, searching databases, writing compilers, pattern matching etc.
- The language deals with representation of the facts and rules, and processes them using implicit inference engine.
- Logic programming deals with relations rather than functions.

Logic

- A proposition is a logical statement that is made if it is true. E.g. Today is Tuesday.
- Symbolic logic uses propositions to express the objects of the real world and relationship between them. E.g. reads(X, Y)
- The propositions can be atomic or compound.
- For symbolic logic, the first order predicate calculus is used.

Propositions

- The objects are represented by simple terms.
- The terms are either constants or variables.
- The propositions consist of compound terms.
- A compound term is composed of two parts: a functor and an ordered list of parameters.
- The functor is the function symbol that names the relation. E.g. likes(jerry, tom), friends(X, jerry) etc. These are also known as predicates in First Order Logic.
- Propositions can either represent the facts and rules, or they can represent the query.

Basic Elements of Prolog

- **Term:** A Prolog term is a constant, a variable or a structure. A constant is either an atom or an integer.
- **Atom:** These are the symbolic values of Prolog. E.g., delhi, alice, bob, abc_d etc.
- **Variables:** Variable is any string of characters or digits that begin with _ (in some versions of prolog) or with a letter in upper case.
- **Structure:** They represent the atomic propositions of predicate calculus.

Data and program in Prolog

- Distinction between data and program is blurred in Prolog. For example, mother(eric) is itself a data (fact) and is also an atomic proposition.
- Facts and rules are used as data.

A simple prolog program

```

mother(anna, john).
mother(anna, eric).
father(tom, john).
father(tom, eric).
father(mike, tom).
sibling(X,Y) :- parent(Z,X), parent(Z,Y).
grandfather(X,Y) :- father(X,Z), father(Z,Y).
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
  
```

Propositions:
facts

?- [p1].

true.

```

?- mother(anna, X).
X = john ;
X = eric.
  
```

```

?- mother(X, eric).
X = anna.
  
```

```

?- sibling(john, eric).
true ;
true.
  
```

```

?- grandfather(X, eric).
X = mike.
  
```

```

?- grandfather(X, john).
X = mike.
  
```

```

?- father(mike, X).
X = tom.
  
```

constant

Compiling or
loading the program
p1.pl

query

variable

Consequent

Antecedent

Propositions:
rules

Program execution

```
mother(anna, john).  
mother(anna, eric).  
father(tom, john).  
father(tom, eric).  
father(mike, tom).  
sibling(X,Y) :- parent(Z,X), parent(Z,Y).  
grandfather(X,Y) :- father(X,Z), father(Z,Y).  
parent(X,Y) :- father(X,Y).  
parent(X,Y) :- mother(X,Y).
```

Query:

```
?- mother( X, john).
```

Combines query clause and facts and rules
 $\text{mother(anna, john)} \cap \text{mother(} X, \text{john)}$
 $\text{mother(anna, eric)} \cap \text{mother(} X, \text{john)}$

Processing

Available clauses

1. mother(anna, john).
2. mother(anna, eric)

It then matches the literals in the query and uses substitution for X as $\sigma = \{X/\text{anna}\}$

Program execution

```

mother(anna, john).
mother(anna, eric).
father(tom, john).
father(tom, eric).
father(mike, tom).
sibling(X,Y) :- parent(Z,X), p
grandfather(X,Y) :- father(X,Z),
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).

```

Query:

```
?- grandfather( X, eric).
```

Processing

Available clauses

1. Grandfather(X,Y) :-
father(X,Z), father(Z,

First it instantiates Y using $\sigma = \{Y/eric\}$

grandfather(X, eric) :- father(X,Z), father(Z,eric)

Looks for predicate father's instances

father(tom, john)
father(tom, eric)

Uses $\sigma = \{Z/tom\}$ to get

grandfather(X, eric) :- father(X,tom), father(tom,eric)

Next uses father(mike, tom) and instantiates

grandfather(mike, eric) :- father(mike,tom), father(tom,eric)

As the antecedent is true, the consequent becomes true with $\sigma = \{X/mike\}$

Output is X = mike