Expressions and Statements

Control

- Control:

  what gets executed, when, and in what order.

- Abstraction of control:
  - Expression
  - Statement
  - Exception Handling
  - Procedures and functions

Expression vs. Statement

- In pure (mathematical) form:
  - Expression:
    - no side effect
    - return a value
  - Statement:
    - side effect
    - no return value
- Functional languages aim at achieving this pure form
- No clear-cut in most languages

Expression

  - Basic expression (literal, identifiers)
  - More Complex Expressions are built recursively from Basic expressions by the applications of operators, functions, involving grouping symbols such as ()
- Number of operands:
  - unary, binary, ternary operators

Expression

- Operators can be written in **infix**, **postfix**, or **prefix** notation, corresponding to an inorder, postorder, or preorder traversal of the syntax tree of the expression.
- Operator, function: equivalent concepts
  - 3+4*5 infix notation written in postfix form as

    3 4 5 * + and in prefix form as + 3 * 4 5.
- The advantage of postfix and prefix forms is that they do not require parentheses to express the order in which operators are applied.
- Thus, operator precedence is not required to disambiguate an unparenthesized expression.
- For instance, (3 + 4) * 5 is written in postfix form as 3 4 + 5 * and in prefix form as * + 3 4 5.

Expression

- Associativity of operators is also expressed directly in postfix and prefix form without the need for a rule.
- For example, the postfix expression 3 4 5 + + right associates and 3 4 + 5 + left associates the infix expression 3 + 4 + 5.

- Many languages make a distinction between operators and functions.
- Operators are predefined and (if binary) written in infix form, with specified associativity and precedence rules, whereas
- functions, which can be either predefined or user-defined, are written in prefix form and the operands are viewed as arguments or actual parameters to calls of the functions.

Expression

- For example, in C, if we write the expression 3 + 4 * 5 with user-defined addition and multiplication operations, it would appear as

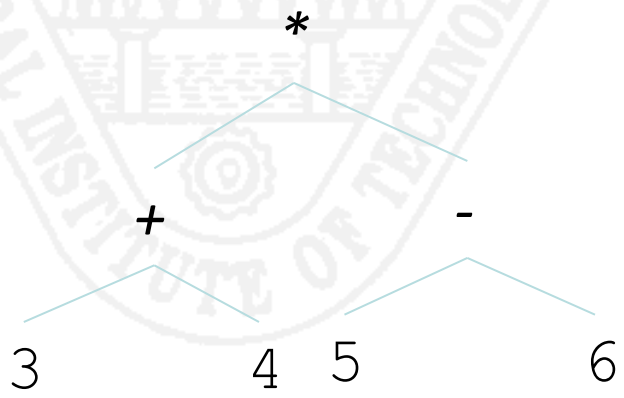
  add(3,mul(4,5)).

- In fact, the distinction between operators and functions is arbitrary, since they are equivalent concepts.
- For example, (3 + 4) * 5 can be written
  - mul(add(3,4), 5)
    - “*”(“+”(3,4),5) (Ada, prefix notation)
    - (* (+ 3 4) 5) (LISP, prefix notation)
- Stack-based languages such as PostScript and FORTH use postfix notation exclusively

Applicative Order Evaluation (Strict Evaluation)

- Evaluate the operands first, then apply operators (bottom-up evaluation)

  (subexpressions evaluated, no matter whether they are needed)



- Which is evaluated first? 3+4 or 5-6

Applicative Order Evaluation (Strict Evaluation)

- In applicative order evaluation, first the “+” and “−” nodes are evaluated to 7 and −1, respectively, and then the “*” is applied to get −7.
- Let us also consider how this would appear in terms of user-defined functions, written in prefix form in C syntax:
- mul(add(3,4),sub(5,6))
- Applicative order says evaluate the arguments first. Thus, calls to add (3,4) and sub(5,6) are made, which are also evaluated using applicative order. Then, the calls are replaced by their returned values, which are 7 and −1. Finally, the call mul(7,−1).
- Note that this process corresponds to a bottom-up “reduction” of the syntax tree to a value.
- First the + and - nodes are replaced by their values 7 and −1, and finally the root * node is replaced by the value −7, which is the result of the expression.

Expression and Side Effects

If Order of evaluation of expression causes no side effects, then the expression will yield the same result, regardless of the order of evaluations of its sub expressions.

In the presence of side effects, the Order of evaluation can make difference.

– Side effect: Order matters

Order Matters

C:

```
#include <stdio.h>
 int x = 1;
 int f(){
    x += 1;
    return x;
 }
 int p(int a, int b) {
    return a + b;
 }
 main(){
    printf("%d\n",p(x,f()));
    return 0;
 }
```

3 or 4 depending on the compiler

Java:

```
class example
{ static int x = 1;
  public static int f()
  {
    x = x+1;
    return x;
  }
  public static int p(int a, int b) {
    return a + b;
   }
  public static void main(String[] args)
  {
   System.out.println(p(x, f()));
  }
}
```

3

Expected Side Effect

- In C, assignment is an expression, not a statement: x = y not only assigns the value of y to x but also returns the copied value as its result.(When a type conversion occurs, the type of the copied value is that of x, not y)
- Thus, assignments can be combined in expressions, so that x = (y = z) assigns the value of z to both x and y.
- Note that in languages with this kind of construction, assignment can be considered to be a binary operator similar to arithmetic operators.
- In that case its precedence is usually lower than other binary operators, and it is made right associative.
- Thus, x = y = z + w in C assigns the sum of z and w to both x and y (and returns that value as its result).

Sequence Operator

- C and other expression languages also have a **sequence operator,** which allows several expressions to be combined into a single expression and evaluated sequentially.
- In C, the sequence operator is the comma operator, which has precedence lower than any other operator.
- (expr1, expr2, …, expr*n*)
  - Left to right (this is indeed specified in C)
  - The return value is expr*n*
- Example:

```
x=1;
y=2;
x = (y+=1,x+=y,x+1)
printf("%d\n",x);
```

- The expression returns the value 5 and leaves x with the value 5 and y with the value 3.

Non-strict evaluation

- Evaluating an expression without necessarily evaluating all the subexpressions.

- short-circuit evaluation of Boolean or logical expressions.
- if-expression, case-expression

Short-Circuit Evaluation

- `if (false and x) … if (true or x)…`
  - No need to evaluate x, no matter x is true or false
- What is it good for?
  - `if (i <=lastindex and a[i]>=x)…`

  A test for the validity of an array index can be written in the same expression as a test of its subscripted value, as long as the range test occurs first.
  - `if (p != NULL and p->next==q)…`

  Similarly, a test for a null pointer can be made in the same expression as a dereference of that pointer, using short-circuit evaluation:
- if (x != 0 and y % x == 0) …//ok
- but not if we write:

  if (y % x == 0 and x != 0) ... // not ok!

if-expression

- `if(test-exp, then-exp, else-exp)`

  ternary operator
  - test-exp is always evaluated first
  - Either then-exp or else-exp are evaluated, not both

  - `e1 ? e2 : e3` (C)

- `Different from if-statement?`

case-expression

```
case e1 of
    a => e2
    b => e3
    c => e4 ;
```

- which is more or less equivalent to a series of nested if-expressions: if e1 = a then e2 else if e1 = b then e3....

Normal order evaluation (lazy evaluation)

- When there is no side-effect:

  Normal order evaluation (Expressions evaluated in mathematical form)

  - Operation evaluated ***before*** the operands are evaluated;
  - Operands ***evaluated only when necessary***.

- ```
  int double (int x) { return x+x; }
  int square (int x) { return x*x; }
  ```

  Applicative order evaluation : square(double(2)) = …

  Normal order evaluation : square(double(2)) = …

- In the absence of side effects, normal order evaluation does not change the semantics of a program. (While it might seem inefficient, 2+2 gets evaluated twice instead of once)
  ```

Example: If C used normal order evaluation, the C if-expression *e1 ? e2 : e3 can be written (for each data type) as an ordinary C function, instead of needing special* syntax in the language:

```
int if_exp( int x, int y, int z ){
    if (x) return y; else return z;
}
```

Here y and z are only evaluated if they are needed in the code of if_exp, which is the behavior that we want.

What is it good for?

- On the other hand, the presence of side effects can mean that normal order evaluation substantially changes the semantics of a program.

```
int get_int(void) {
  int x;
  scanf("%d" &x);
  return x;
}
```

  - This function has a side effect (input).
- `square(get_int());` using normal order evaluation:
- It would be expanded into get_int()*get_int(), and *two integer values, rather than just* one, would be read from standard input—a very different outcome from what we would expect.

Statements

- If-statements, case-(switch-)statements, loops

Selection Statements

- A *selection statement* provides the means of choosing between two or more paths of execution
- Two general categories:
  - Two-way selectors
  - Multiple-way selectors

Two-Way Selection Statements

- General form:

```
if control_expression
   then clause
   else clause
```

- Design Issues:
  - What is the form and type of the control expression?
  - How are the then and else clauses specified?
  - How should the meaning of nested selectors be specified?

Guarded if

- if B1 -> S1

I B2 -> S2

I B3 -> S3

. . .

I Bn -> Sn

fi

Guarded if

- The semantics of this statement are as follows:
- each *Bi*'s is a Boolean expression, called a **guard**, and each *Si*'s is a statement sequence.
- If one *Bi*'s evaluates to true, then the corresponding statement sequence *Si* is executed.
- If more than one *Bi*'s is true, then the one and only corresponding *Si*'s is selected for execution.
- If no *Bi* is true, then an error occurs.

Guarded if

- Several Interesting features in this description:
- 1. it does not say that the first *Bi* that evaluates to true is the one chosen. Thus, the guarded if introduces nondeterminism into programming, a feature that becomes very useful in concurrent programming.
- 2. it leaves unspecified whether all the guards are evaluated. Thus, if the evaluation of a *Bi* has a side effect, the result of the execution of the guarded if may be unknown.
- 3. of course, the usual deterministic implementation of such a statement would sequentially evaluate the *Bi*'s until a true one is found, whence the corresponding *Si* is executed and control is transferred to the point following the guarded statement.
- The two major ways that programming languages implement conditional statements like the guarded if are as if-statements and case-statements.

Example of Ambiguity: Dangling-Else

Grammer rule:

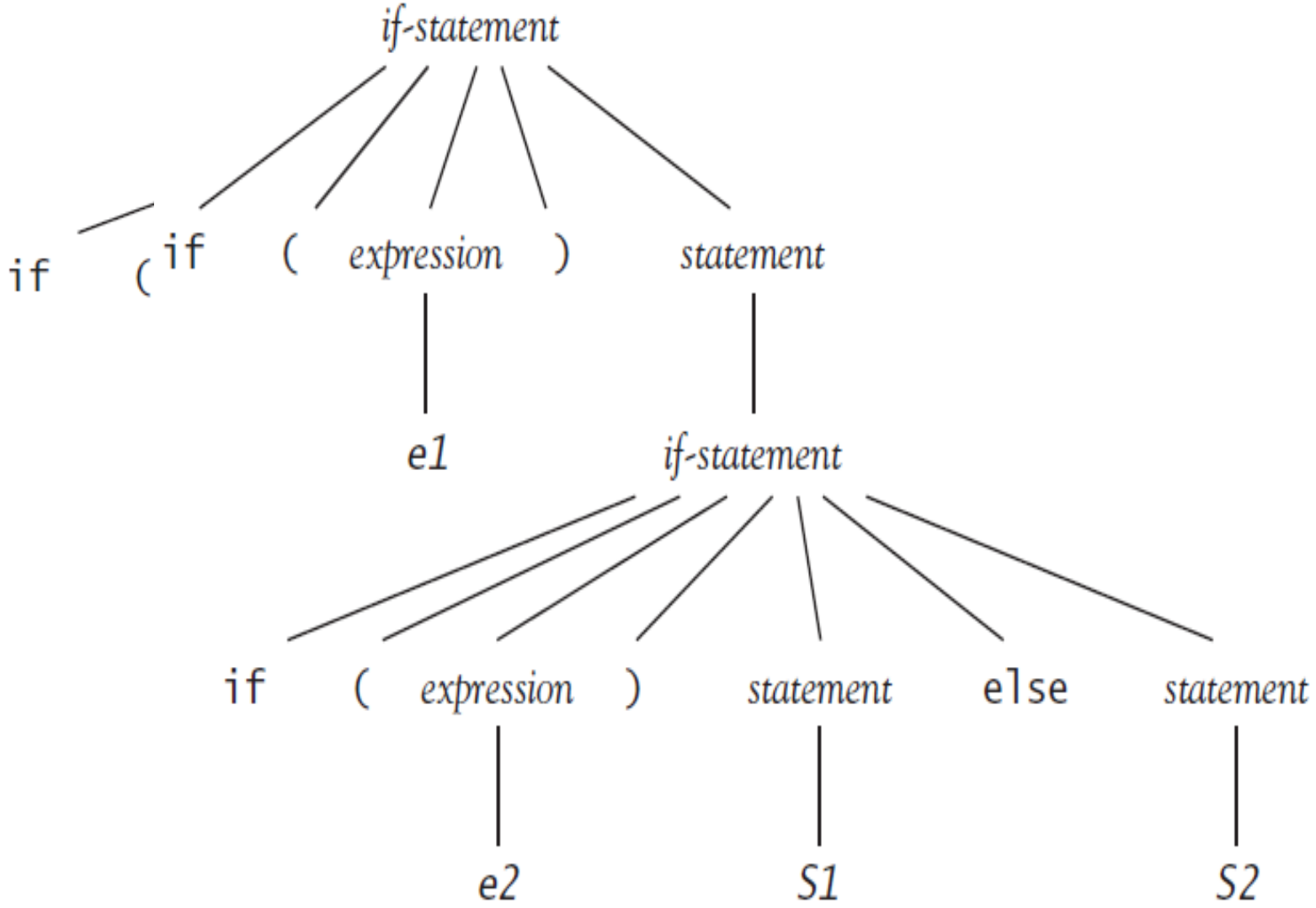
stmt → `if(` *expr* `)` *stmt* [`else` *stmt*]

Input String:

if (e1) if (e2) s1 else s2

No. of parse trees: 2

Then which parse tree is correct?



Example of Ambiguity: Dangling-Else

- The syntax does not tell us whether an else after two if-statements should be associated with the first or second if.
- C and Pascal solve this problem by enforcing a **disambiguating rule**, which works as follows:
- The else is associated with the closest preceding if that does not already have an else part. This rule is also referred to as the **most closely nested rule** for if-statements.
- As an illustration, if we want actually to associate the else with the first if in the preceding statement, we would have to write either:

  if (*e1*) { if (*e2*) *S1* } else *S2*

  or:

  if (*e1*) if (*e2*) *S1* else ; else *S2*

Bracketing Keyword

- *if-statement* → if *condition* then *sequence-of-statements*

  [else *sequence-of-statements*]

  end if ;

- The two bracketing keywords end if (together with the semicolon) close the if-statement and remove the ambiguity, since we must now write either:

  if *e1* then if *e2* then *S1* end if ; else *S2* end if;

  or:

  if *e1* then if *e2* then *S1* else *S2* end if ; end if ;

  to decide which if is associated with the else part. This also removes the necessity of using brackets (or a begin - end pair) to open a new sequence of statements.

- The if-statement can open its own sequence of statements and, thus, becomes fully structured:

- *stmt* → `if` cond then *stmt* [`else` *stmt*] fi;

- A similar approach was taken in the historical (but influential) language Algol68, which suggestively uses keywords spelled backward as bracketing keywords:
- if *e1* then *S1* else *S2* fi
- Thus, the bracketing keyword for if-statements is fi

If e1 then if e2 then s1 end if; else s2 fi;

Nesting Selectors

- Java example

```
if (sum == 0)
  if (count == 0)
      result = 0;
else result = 1;
```

- Which `if` gets the `else`?
- Java's static semantics rule: `else` matches with the nearest `if`

Nesting Selectors (continued)

- To force an alternative semantics, compound statements may be used:

```
if (sum == 0) {
 if (count == 0)
     result = 0;
}
else result = 1;
```

- The above solution is used in C, C++, Java and C#

Iterative Statements

- The repeated execution of a statement or compound statement is accomplished either by iteration or recursion
- General design issues for iteration control statements:
  1. How is iteration controlled?
  2. Where is the control mechanism in the loop?

Iterative Statements

Guarded do:

```
do B1 - > S1
| B2 - > S2
| B3 - > S3
. . .
| Bn -> Sn
od
```

Iterative Statements: Examples

- C's `for` statement

```
for ([expr_1] ; [expr_2] ; [expr_3]) statement
```

- The expressions can be whole statements, or even statement sequences, with the statements separated by commas
- There is no explicit loop variable
- Everything can be changed in the loop
- The first expression is evaluated once, but the other two are evaluated with each iteration

Iterative Statements: Examples

- C++ differs from C in two ways:
  1. The control expression can also be Boolean
  2. The initial expression can include variable definitions (scope is from the definition to the end of the loop body)
- Java and C#
  - Differs from C++ in that the control expression must be Boolean

Iterative Statements: Logically-Controlled Loops

- Repetition control is based on a Boolean
- General forms:

```
while (ctrl_expr)
   loop body
```

```
do
   loop body
while (ctrl_expr)
```

GOTO

Fortran77 code:

```
10 IF (A(I).EQ.0) GOTO 20
...
I = I + 1
GOTO 10
20 CONTINUE
```

C equivalent code is: while (a[i] != 0) i++;

GOTO

```
      IF (X.GT.0) GOTO 10
      IF (X.LT.0) GOTO 20
      X = 1
      GOTO 30
10 X = X + 1
      GOTO 30
20 X = -X
      GOTO 10
30 CONTINUE
```

An example of spaghetti code in Fortran77

Its equivalent code in C is

```
if(x>=0){x=x+1} else
{x=-x; x+=1}
```

Goto

- Proponents of structured exits argue that a loop should always have a single exit point, which is either at the beginning point of the loop (in the header of a while loop or a for loop) or at the end of the loop (in the footer of a do-while or repeat-until loop).

- In a language without a goto or similar construct, the syntax of high-level loop structures both supports and enforces this restriction.

- The opponents of this orthodoxy, typically programmers who use C, C++, Java, or Python, argue that this restriction forces the programmer to code solutions to certain problems that are more complicated than they would be if unstructured exits were allowed.

Loop exit

Table 9.1 Search methods using structured and unstructured loop exits

Search Using Structured Loop Exit

```
int search(int array[], int target){
   boolean found = false;
   int index = 0;
   while (index < array.length && ! found)
      if (array[index] == target)
         found = true;
      else
         index++;
   if (found)
      return index;
   else
      return -1;
}
```

Search Using Unstructured Loop Exit

```
int search(int array[], int target){
   for (int index = 0; index < array.length;
    index++)
      if (array[index] == target)
         return index;
   return -1;
}
```

Loop exit

- The first method uses a while loop with two conditions, one of which tests for the end of the array. The other tests for the detection of the target item.
- This version of the method requires a separate Boolean flag, as well as a trailing if-else statement to return the appropriate value.
- The method adheres to the structured loop exit principle, but at the cost of some complexity in the code.

- The second version of the method, by contrast, uses an embedded return statement to exit a for loop immediately upon finding the target value.
- If the target is not found, the loop runs to its structured termination, and a default of −1 is returned below the loop.
- The unstructured loop exit, thus, eliminates the need for a Boolean flag and a trailing if-else statement.

Loop exit

Table 9.2 Methods using structured and unstructured sentinel-based loop exits

Structured Loop Exit

```
void processInputs(Scanner s){
   int datum = s.nextInt();
   while (datum != -1){
      process(datum);
      datum = s.nextInt();
   }
}
```

Unstructured Loop Exit

```
void processInputs(Scanner s){
   while (true){
      int datum = s.nextInt();
      if (datum == -1)
         break;
      process(datum);
   }
}
```

Exception Handling

- Asynchronous Exceptions
- Synchronous Exceptions
- Synchronous exceptions can be caught
- Asynchronous exceptions cannot be caught
- Example: User defined exceptions (Synchronous)
- Communication failure, hardware failure (Asynchronous)

Exceptions

- What exceptions are predefined in the language?
- None in C++
- Can they be disabled?
- Since they are not defined in C++ NA.
- Can user defined exceptions be created?
- Yes
- What is their scope?
- Lexical

Exceptions

- Unusually, in C++ there is no special exception type, and thus no reserved word to declare them. Instead, any structured type (struct or class) may be used to represent an exception:
- struct Trouble {} trouble;
- struct Big_Trouble {} big_trouble;

Exceptions

```
struct Trouble{
string error_message;
int wrong_value;
} ;//declare exception object later i.e at the point where exception occurs.
```

Lexical scope rule for exception.

C++ standard exceptions

- **bad_alloc**: Failure of call to new.
- **bad_cast**: Failure of a dynamic_cast **out_of_range**: Caused by a checked subscript operation on library containers (the usual subscript operation is not checked).
- **overflow_error, underflow_error, range_error**: Caused by math library functions and a few others.

C++ exception

- C++ does not have predefined exception.

Exception Handlers

- How are they defined?
- Using try catch
- What is their scope?
- Their respective blocks
- What default handlers are provided for predefined exceptions?
- NA
- Can they be replaced?
- NA Standard exceptions can be replaced

C++

```
(1) try
(2) { // to perform some processing
(3) ...
(4) }
(5) catch (Trouble t)
(6) { // handle the trouble, if possible
(7) displayMessage(t.error_message);
(8) ...
(9) }
(10) catch (Big_Trouble b)
(11) { // handle big trouble, if possible
(12) ...
(13) }
(14) catch (...) // actually three dots here, not an ellipsis!
(15) { // handle any remaining uncaught exceptions
(16) }
```

C++

- Here the compound statement after the reserved word try is required and any number of catch blocks can follow the initial try block.
- Each catch block also requires a compound statement and is written as a function of a single parameter whose type is an exception type (which can be any class or structure type).
- The exception parameter may be consulted to retrieve appropriate exception information, as in line 7.
- Note how catchall catch block at the end that catches any remaining exceptions, as indicated by the three dots inside the parentheses on line 14.

Control

- How is control passed to a handler?
- Using throw with exception object
- Where does control pass after a handler is executed?
- Statement after catch block
- What runtime environment remains after an error handler executes?
- Whatever after catch block

Control

- C++ uses the reserved word throw and an exception object to raise an exception
- When an exception is raised, typically the current computation is abandoned, and the runtime system begins a search for a handler.
- This search begins with the handler section of the block in which the exception was raised. If no handler is found, then the handler section of the next enclosing block is consulted, and so on (this process is called **propagating the exception**).

Call Unwinding

- The process of exiting back through function calls to the caller during the search for a handler is called **call unwinding** or **stack unwinding.**

Resumption Model

- Once a handler has been found and executed, there remains the question of where to continue execution?
- One choice is to return to the point at which the exception was first raised and begin execution again with that statement or expression. This is called the **resumption model** of exception handling

Resumption Model

- It requires that, during the search for a handler and possible call unwinding, the original environment and call structure must be preserved and reestablished prior to the resumption of execution.

Termination model

- The alternative to the resumption model is to continue execution with the code immediately following the block or expression in which the handler that is executed was found.
- This is called the **termination model** of exception handling, since it essentially discards all the unwound calls and blocks until a handler is found.
- C++

Binary Search Tree

```
struct Tree{
int data;
Tree * left;
Tree * right;
};
```

Binary Search Tree

```
void fnd (Tree* p, int i) // helper procedure
{
        if (p != 0)
            if (i == p->data) throw p;
            else if (i <p->data) fnd(p->left,i);
            else fnd(p->right,i);
}
```

Binary Search Tree

```
Tree * find (Tree* p, int i)
{
        try {
                fnd(p,i);
        }
        catch(Tree* q){
                return q;
        }
return 0;
}
```

References

Text book

- Kenneth C. Louden and Kenneth Lambert “Programming Languages Principles and Practice” Third edition Cengage Learning Publication.

Reference Books:

- Terrence W. Pratt, Masvin V. Zelkowitz “Programming Languages design and Implementation” Fourth Edition Pearson Education.
- Allen Tucker, Robert Noonan “Programming Languages Principles and Paradigms second edition Tata MC Graw –Hill Publication.