Languages and Compilers
(SProg og Oversættere)

Lecture 13
Programming Language Design
Expressions and Statements

Bent Thomsen

Department of Computer Science

Aalborg University

With acknowledgement to Simon Gay, John Mitchell and Elsa Gunter who’s slides this lecture is based on.

Learning goals

- Overview of common language constructs and design questions
- Understand
  - Explicit sequence control vs. Implicit sequence control
    - Evaluation of expressions
    - Statements
  - Structured sequence control vs. unstructured sequence control
    - Conditional Selection
    - Loop constructs
    - Jumps

Syntactic Elements

- Declarations and Definitions
  - Scopes and visibility
  - always before use or not, initialization or not,
- **Expressions**
- **Statements**
- Subprograms

- Separate subprogram definitions (Module system)
- Separate data definitions
- Nested subprogram definitions
- Separate interface definitions

Sequence control

- Implicit and explicit sequence control
  - Expressions
    - Precedence rules
    - Associativity
  - Statements
    - Sequence
    - Conditionals
    - Loop constructs
    - unstructured vs. structured sequence control

Expression Evaluation

- Determined by
  - operator evaluation order
  - operand evaluation order
- Operators:
  - Most operators are either infix or prefix (some languages have postfix)
  - Order of evaluation determined by operator precedence and associativity

Example

- What is the result of:

`3 + 4 * 5 + 6`

- Possible answers:
  - `41 = ((3 + 4) * 5) + 6`
  - `47 = 3 + (4 * (5 + 6))`
  - `29 = (3 + (4 * 5)) + 6 = 3 + ((4 * 5) + 6)`
  - `77 = (3 + 4) * (5 + 6)`

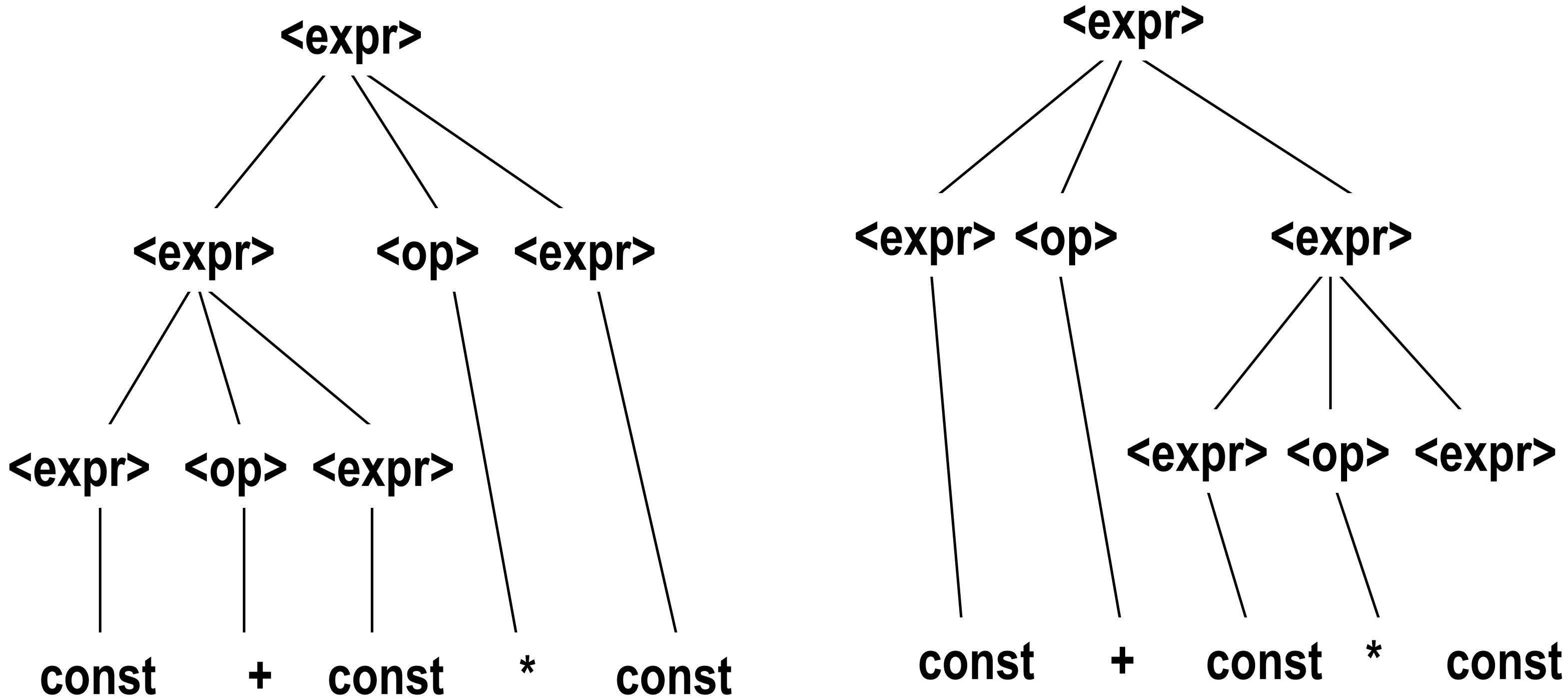
- In most languages, `3 + 4 * 5 + 6 = 29`
- … but it depends on the precedence of operators

An Ambiguous Expression Grammar

How to parse 3+4*5?

<expr> → <expr> <op> <expr> | const

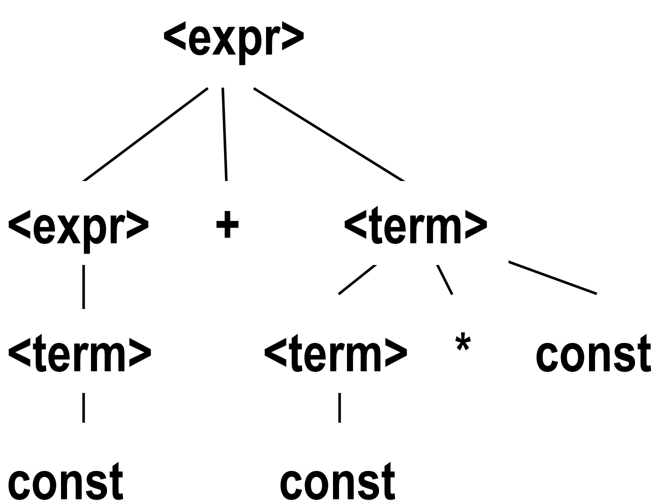
<op> → + | *



Expressing Precedence in grammar

- We can use the parse tree to indicate precedence levels of the operators

<expr> → <expr> + <term> | <term>
<term> → <term> * `const` | `const`



In LALR parsers we can specify
Precedence which translates into
Solving shift-reduce conflicts

Note in LL(1) parsers we have to use
Left recursion elimination

Expr → Term Expr1 .
Expr1 →+ Term Expr1
| .
Term → const Term1 .
Term1 →* const Term1
| .

Operator Precedence

- Operators of highest precedence evaluated first (bind more tightly).

- Precedence for operators usually given in a table, e.g.:

- In APL, all infix operators have same precedence

Level	Operator	Operation
Highest	** abs not	Exp, abs, negation
	* / mod rem	
	+ -	Unary
	+ - &	Binary
	= <= < > =>	Relations
Lowest	And or xor	Boolean

Precedence table for ADA

C precedence levels

- Precedence Operators Operator names
- 17 tokens, a[k], f() Literals, subscripting, function call
- ., -> Selection
- 16 ++, -- Postfix increment/decrement
- 15* ++, -- Prefix inc/dec
- ~, -, sizeof Unary operators, storage
- !, &, * Logical negation, indirection
- 14 typename Casts
- 13 *, /, % Multiplicative operators
- 12 +, - Additive operators
- 11 <<, >> Shift
- 10 <, >, <=, >= Relational
- 9 ==, != Equality
- 8 & Bitwise and
- 7 ^ Bitwise xor
- 6 | Bitwise or
- 5 && Logical and
- 4 || Logical or
- 3 ?: Conditional
- 2 =, +=, -=, *=, Assignment
- /=, %=, <<=, >>=,
- &=, ^=, |=
- 1 , Sequential evaluation

Associativity

- When we have sorted precedence we need to sort associativity!
- What is the value of:

```
7 - 5 - 2
```

- Possible answers:
  - In Pascal, C++, SML associate to the left

```
7 - 5 - 2 = (7 - 5) - 2 = 0
```

  - In APL, associate to the right

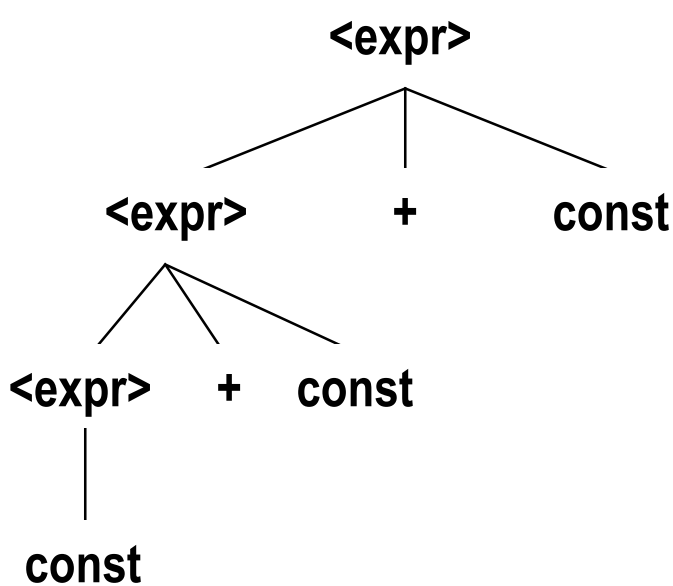
```
7 - 5 - 2 = 7 - (5 - 2) = 4
```

Again we can use syntax

- Operator associativity can also be indicated by a grammar

<expr> -> <expr> + <expr> | const (ambiguous)

<expr> -> <expr> + const | const (unambiguous)



In LALR parsers we can specify
Associativity which translates into
Solving shift-reduce conflicts

Operand Evaluation Order

- Example:

```
A := 5;
f(x) = {A := x+x; return x};
B := A + f(A);
```

- What is the value of B?
- 10 or 15?

Example

- If assignment returns the assigned value, what is the result of

```
x = 5;
y = (x = 3) + x;
```

- Possible answers: 6 or 8
- Depends on language, and sometimes compiler
  - C allows compiler to decide
  - SML forces left-to-right evaluation
    - Note assignment in SML returns a unit value
    - .. but we could define a derived assignment operator in SML as fn (x,v)=>(x:=v;v)

Solution to Operand Evaluation Order

- Disallow all side-effects
  - "Purely" functional languages try to do this – Miranda, Haskell
  - It works!
  - Consequence
    - No two-way parameters in functions
    - No non-local references in functions
  - Problem:
    - I/O, error conditions such as overflow are inherently side-effecting
    - Programmers want the flexibility of two-way parameters (what about C?) and non-local references

Solution to Operand Evaluation Order

- Disallow all side-effects in expressions but allow in statements
  - Problem: not applicable in languages with nesting of expressions and statements

Solution to Operand Evaluation Order

- Fix order of evaluation
  - SML does this – left to right
  - Problem: makes some compiler optimizations hard or impossible
- Leave it to the programmer to be sure the order doesn't matter
  - Problem: Usually requires lots of brackets
  - Problem: error prone

  - Fortress: Parallel evaluation unless specified to be sequential

Short-circuit Evaluation

- Boolean expressions:
- Example: **`x <> 0 andalso y/x > 1`**
- Problem: if **`andalso`** is ordinary operator and both arguments must be evaluated, then **`y/x`** will raise an error when **`x = 0`**

- Similar problem for conditional expressions
- Example **`(x == 0)?0:sum/x`**

Boolean Expressions

- Most languages allow (some version of) **`if`**...**`then`**...**`else`**, **`andalso`**, **`orelse`** not to evaluate all the arguments
- **`if true then A else B`**
  - doesn't evaluate B
- **`if false then A else B`**
  - doesn't evaluate A
- **`if b_exp then A else B`**
  - Evaluates b_exp, then applies previous rules

Boolen Expressions

- `Bexp1 andalso Bexp2`
  - If Bexp1 evaluates to false, doesn’t evaluate Bexp2
- `Bexp1 orelse Bexp2`
  - If Bexp1 evaluates to true, doesn’t evaluate Bexp2

Short-circuit Evaluation – Other Expressions

- Example: `0 * A = 0`
- Do we need to evaluate A?

- In general, in **f(x,y,…,z)** are the arguments to **f** evaluated before **f** is called and the values are passed? Or are the unevaluated expressions passed as arguments to **f** allowing **f** to decide which arguments to evaluate and in which order?

Eager Evaluation

- If a language requires all arguments to be evaluated before a function is called, the language does ***eager evaluation*** and the arguments are passed using pass by value (also called ***call by value***) or pass by reference

Lazy Evaluation

- If a language allows a function to determine which arguments to evaluate and in which order, the language does ***lazy evaluation*** and the arguments are passed using pass by name (also called ***call by name***)

Lazy Evaluation

- Lazy evaluation is mainly done in purely functional languages
- Some languages support a mix
- The effect of lazy evaluation can be implemented in functional languages with eager evaluation
  - Use thunking **`fn()=>exp`** and pass function instead of exp
- C# 2.0 has a Lazy evaluation construct:
  - **`yield return`** which can be used with Iterators

Call by name

- In call-by-name evaluation, the arguments to a function are not evaluated before the function is called — rather, they are substituted directly into the function body (using capture-avoiding substitution) and then left to be evaluated whenever they appear in the function.
- If an argument is not used in the function body, the argument is never evaluated
- If it is used several times, it is re-evaluated each time it appears
  - (in Pure lazy functional languages memorization can be used – why?)
- Algol 60 introduced call-by-name.
- Long consider too expensive and weird
  - but now in Scala
  - Can be simulated in C# using Expression<T> parameters
- The classical use case for call-by-name is Jensens device

Arithmetic Expressions

- Design issues for arithmetic expressions:
  1. What are the operator precedence rules?
  2. What are the operator associativity rules?
  3. What is the order of operand evaluation?
  4. Are there restrictions on operand evaluation side effects?
  5. Does the language allow user-defined operator overloading?
     - C++, Ada, C# allow user defined overloading
     - Can lead to readability problems
  6. What mode mixing is allowed in expressions?
     - Are operators of different types, e.g. int and float allowed
     - How is type conversion done

Pause

Syntactic Elements

- Definitions
- Declarations
- Expressions
- **Statements**
- Subprograms

- Separate subprogram definitions (Module system)
- Separate data definitions
- Nested subprogram definitions
- Separate interface definitions

Control of Statement Execution

- Sequential
- Conditional Selection
- Looping Construct
- Must have all three to provide full power of a Computing Machine

Basic sequential operations

- Skip (in C it is just a blanck statement with ;)
- Assignments
  - Most languages treat assignment as a basic operation
  - Some languages have derived assignment operators such as:
    - `+=` and `*=` in C
- I/O
  - Some languages treat I/O as basic operations
  - Others like, C, SML, Java treat I/O as functions/methods
- Sequencing
  - `C;C`
- Blocks
  - `begin …end`
  - `{…}`
  - `let .. in .. end`

Assignment Statements

- Simple assignments:
  - `A = 10` or `A := 10` or `A is 10` or `=(A,10)`
  - In SML assignment is just another (infix) function
    - `:= : ''a ref * ''a -> unit`
- More complicated assignments:
  1. Multiple targets (PL/I)
     `A, B = 10`
  2. Conditional targets (C, C++, and Java)
     `(first==true)? total : subtotal = 0`
  3. Compound assignment operators (C, C++, and Java)
     `sum += next;`

Assignment Statements

- More complicated assignments (continued):

  4. Unary assignment operators (C, C++, and Java)

     `a++;` `(increment a with one but return a)`

     `++a;` `(increment a with one but return a+1)`

     `What does ++a-- evaluate to?`

  C, C++, and Java treat = as an arithmetic binary operator

  e.g.

  `a = b * (c = d * 2 + 1) + 1`

  This is inherited from ALGOL 68

  – **`=`** Can be bad if it is overloaded for the relational operator for equality e.g. (PL/I) **`A = B = C;`**

  – Note difference from C

Assignment Statements

- Assignment as an Expression
  - In C, C++, and Java, the assignment statement produces a result
  - So, they can be used as operands in expressions e.g.

```
while ((ch = getchar())!=EOF){…}
```

  - Disadvantage
    - Another kind of expression side effect

Conditional Selection

- Design Considerations:
  - What controls the selection
  - What can be selected:
    - FORTRAN IF: **IF** (boolean_expr) statement

```
      IF (.NOT. condition) GOTO 20
               . . .
               . . .
   20 CONTINUE
```

    - Modern languages allow any kind of program block
  - What is the meaning of nested selectors

Conditional Selection

- Single-way
  - `IF … THEN …`
  - Controlled by boolean expression
- Two-way
  - `IF … THEN … ELSE`
  - Controlled by boolean expression
  - `IF … THEN …` usually treated as degenerate form of

    `IF … THEN … ELSE`
  - `IF…THEN` together with `IF..THEN…ELSE` require disambiguating associativity

Two-Way Selection Statements

- Nested Selectors
- e.g. (Java)

```
if ...
  if ...
    ...
else ...
```

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

Two-Way Selection Statements

- ALGOL 60's solution - disallow direct nesting

```
if ... then
   begin
   if ...
     then ...
     else ...
   end
```

```
if ... then
   begin
   if ... then ...
   end
 else ...
```

Two-Way Selection Statements

- FORTRAN 90 and Ada solution – closing special words
  - e.g. (Ada)

```
if ... then
   if ... then
      ...
   else
      ...
   end if
 end if
```

- Advantage: readability

```
if ... then
   if ... then
      ...
   end if
 else
   ...
 end if
```

- **`ELSEIF`**
  - Equivalent to nested **`if`**...**`then`**...**`else`**...

Multi-Way Conditional Selection

- **`SWITCH`**
  - Typically controlled by scalar type
  - Each selection has own block of statements it executes
  - What if no selection is given?
    - Language gives default behavior
    - Language forces total coverage, typically with programmer-defined default case
  - One block of code for whole switch
  - Selection specifies program point in block
  - **`break`** used for early exit from block

Switch on String in C#

```
Color ColorFromFruit(string s) {
   switch(s.ToLower()) {
      case "apple":
         return Color.Red;
      case "banana":
         return Color.Yellow;
      case "carrot":
         return Color.Orange;
      default:
         throw new InvalidArgumentException();

   }
}
```

Switch on Type in F#

```
type 'a Visitor =
  class
    abstract member visitPlusExp: 'a * 'a -> 'a
    abstract member visitMinusExp: 'a * 'a -> 'a
    abstract member visitTimesExp: 'a * 'a -> 'a
    abstract member visitDivideExp: 'a * 'a -> 'a
    abstract member visitIdentifier: string -> 'a
    abstract member visitIntegerLiteral: string -> 'a
    new() = {}
```

```
let rec TreeWalker (c:'a Visitor) (ee:Exp) =
  match ee with
  | :? PlusExp as e -> (c.visitPlusExp ((TreeWalker c e.e1),(TreeWalker c e.e2)))
  | :? MinusExp as e -> (c.visitMinusExp ((TreeWalker c e.e1),(TreeWalker c e.e2)))
  | :? TimesExp as e -> (c.visitTimesExp ((TreeWalker c e.e1),(TreeWalker c e.e2)))
  | :? DivideExp as e -> (c.visitDivideExp ((TreeWalker c e.e1),(TreeWalker c e.e2)))
  | :? Identifier as e -> (c.visitIdentifier e.f1)
  | :? IntegerLiteral as e -> (c.visitIntegerLiteral e.f1);;
```

```
type Interpreter =
  class
    inherit int Visitor
    override x.visitPlusExp (x,y) = x + y
    override x.visitMinusExp (x,y) = x - y
    override x.visitTimesExp (x,y) = x * y
    override x.visitDivideExp (x,y) = x / y
    override x.visitIdentifier s = Lookup s
    override x.visitIntegerLiteral s = System.Int32.Parse s
    new() = {}
end;;
```

Pattern matching in C# 7.0

The following is an example of pattern matching:

```
class Geometry();
class Triangle(int Width, int Height, int Base) : Geometry;
class Rectangle(int Width, int Height) : Geometry;
class Square(int width) : Geometry;

Geometry g = new Square(5);
switch (g)
{
    case Triangle(int Width, int Height, int Base):
        WriteLine($"{Width} {Height} {Base}");
        break;
    case Rectangle(int Width, int Height):
        WriteLine($"{Width} {Height}");
        break;
    case Square(int Width):
        WriteLine($"{Width}");
        break;
    default:
        WriteLine("<other>");
        break;
}
```

In the sample above you can see how we match on the data type and immediately destructure it into its components.

Loops

- Main types:
- Counter-controlled iterators (For-loops)
- Logical-test iterators
- Iterations based on data structures
- Recursion

For-loops

- Controlled by loop variable of scalar type with bounds and increment size
- Scope of loop variable?
  - Extends beyond loop?
  - Within loop?
- When are loop parameters calculated?
  - Once at start
  - At beginning of each pass

Iterative Statements

ALGOL 60 Design choices:

1. Control expression can be **`int`** or **`real`**; its scope is whatever it is declared to be
2. Control variable has its last assigned value after loop termination
3. The loop variable cannot be changed in the loop, but the parameters can, and when they are, it affects loop control
4. Parameters are evaluated with every iteration, making it very complex and difficult to read

Iterative Statements

Pascal:

- Syntax:

  `for` variable := initial (**`to`** | **`downto`**) final **`do`** statement

- Design Choices:

  1. Loop variable must be an ordinal type of usual scope
  2. After normal termination, loop variable is undefined
  3. The loop variable cannot be changed in the loop; the loop parameters can be changed, but they are evaluated just once, so it does not affect loop control
  4. Just once

Iterative Statements

Ada:

- Syntax:

  `for` var **`in`** [reverse] discrete_range **`loop`** ...

  `end loop`

- Design choices:

  1. Type of the loop variable is that of the discrete range; its scope is the loop body (it is implicitly declared)
  2. The loop variable does not exist outside the loop
  3. The loop variable cannot be changed in the loop, but the discrete range can; it does not affect loop control
  4. The discrete range is evaluated just once

Iterative Statements

C:

- Syntax:

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

  – The expressions can be whole statements, or even statement sequences, with the statements separated by commas

  – The value of a multiple-statement expression is the value of the last statement in the expression

  e.g.,

```
for (i = 0, j = 10; j == i;  i++) …
```

  – If the second expression is absent, it is an infinite loop

Iterative Statements

- C Design Choices:
  1. There is no explicit loop variable
  2. Loop variable, if there is one, has whatever was assigned last
  3. Everything can be changed in the loop
  4. The first expression is evaluated once, but the other two are evaluated with each iteration
- This loop statement is the most flexible
- But also rather difficult to analyze ..

Iterative Statements

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:

- Differs from C++ in that the control expression must be Boolean

Logic-Test Iterators

- While-loops
  - Test performed before entry to loop
- `repeat`...`until` and `do`...`while`
  - Test performed at end of loop
  - Loop always executed at least once
- Design Issues:
  1. Pretest or posttest?
  2. Should this be a special case of the counting loop statement (or a separate statement)?

Iterative Statements

C , C++, and Java – **`break:`**

- Unconditional; for any loop or **`switch`**; one level only (except Java's can have a label)
- There is also a **`continue`** statement for loops; it skips the remainder of this iteration, but does not exit the loop

Counter-Controlled Loops: Examples

- Python

  `for` loop_variable **`in`** object`:`

  – loop body

  [**`else`**`:`

  – else clause]

  - The object is often a range, which is either a list of values in brackets (`[2, 4, 6]`), or a call to the range function (**`range`**`(5)`, which returns `0, 1, 2, 3, 4`

  - The loop variable takes on the values specified in the given range, one for each iteration

  - The else clause, which is optional, is executed if the loop terminates normally

Iterative Statements

- Iteration Based on Data Structures
  - Concept: use order and number of elements of some data structure to control iteration
  - Control mechanism is a call to a function that returns the next element in some chosen order, if there is one; else exit loop
  - C's **`for`** can be used to build a user-defined iterator
  - e.g.

```
for (p=hdr; p; p=next(p))
{ ... }
```

  - Perl has a built-in iterator for arrays and hashes

    e.g.,

```
foreach $name (@names)
{ print $name }
```

C# Foreach Loops

```
foreach (T x in C) S
```

is implemented as

```
IEnumerable<T> c = C;
IEnumerator<T> e = c.GetEnumerator();
while (e.MoveNext())
{ T x = e.Current; S }
```

Recursion

- Recursion can simplify the solution of a problem, often resulting in shorter, more easily understood source code
  - i.e. Recursion is a technique that solves a problem by solving a smaller problem of the same type
  - How do I write recursive functions?
    - Determine the base case(s)
      - the one for which you know the answer
    - Determine the general case(s)
      - the one where the problem is expressed as a smaller version of itself
- Iteration can be used in place of recursion and visa versa
  - An iterative algorithm uses a looping construct
  - A recursive algorithm uses a branching structure

Recursion vs. iteration

- Recursive implementation

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

- Iterative implementation

```
int Factorial(int n)
{
 int fact = 1;

 for(int count = 2;
      count <= n;
      count++)
   fact = fact * count;

 return fact;
}
```

Counter-Controlled Loops: Examples

- F#
  - Because counters require variables, and functional languages do not have variables, counter-controlled loops must be simulated with recursive functions

```
let rec forLoop loopBody reps =
  if reps <= 0 then ()
  else
    loopBody()
    forLoop loopBody, (reps - 1)
```

  - This defines the recursive function `forLoop` with the parameters `loopBody` (a function that defines the loop's body) and the number of repetitions
  - `()` means do nothing and return nothing

Recursion vs. iteration

- Recursion can simplify the solution of a problem, often resulting in shorter, more easily understood source code
- Recursive solutions are often less efficient, in terms of both time and space, than iterative solutions
  - Well this is what the literature says …
  - This is usually true for languages such as C, Java and C# as method calls can be expensive and deep recursions can take up a lot of stack space
  - However, on modern hardware, functions calls call, especially tail recursive calls can be cheap. Thus modern functional languages like Haskell, SML, Scala and F# encourage recursion

Gotos

- Requires notion of program point
- Transfers execution to given program point
- Basic construct in machine language
- Implements loops
- Makes programs hard to read and reason about
- Hard to know how a program got to a given point
- Generally thought to be a bad idea in a high level language

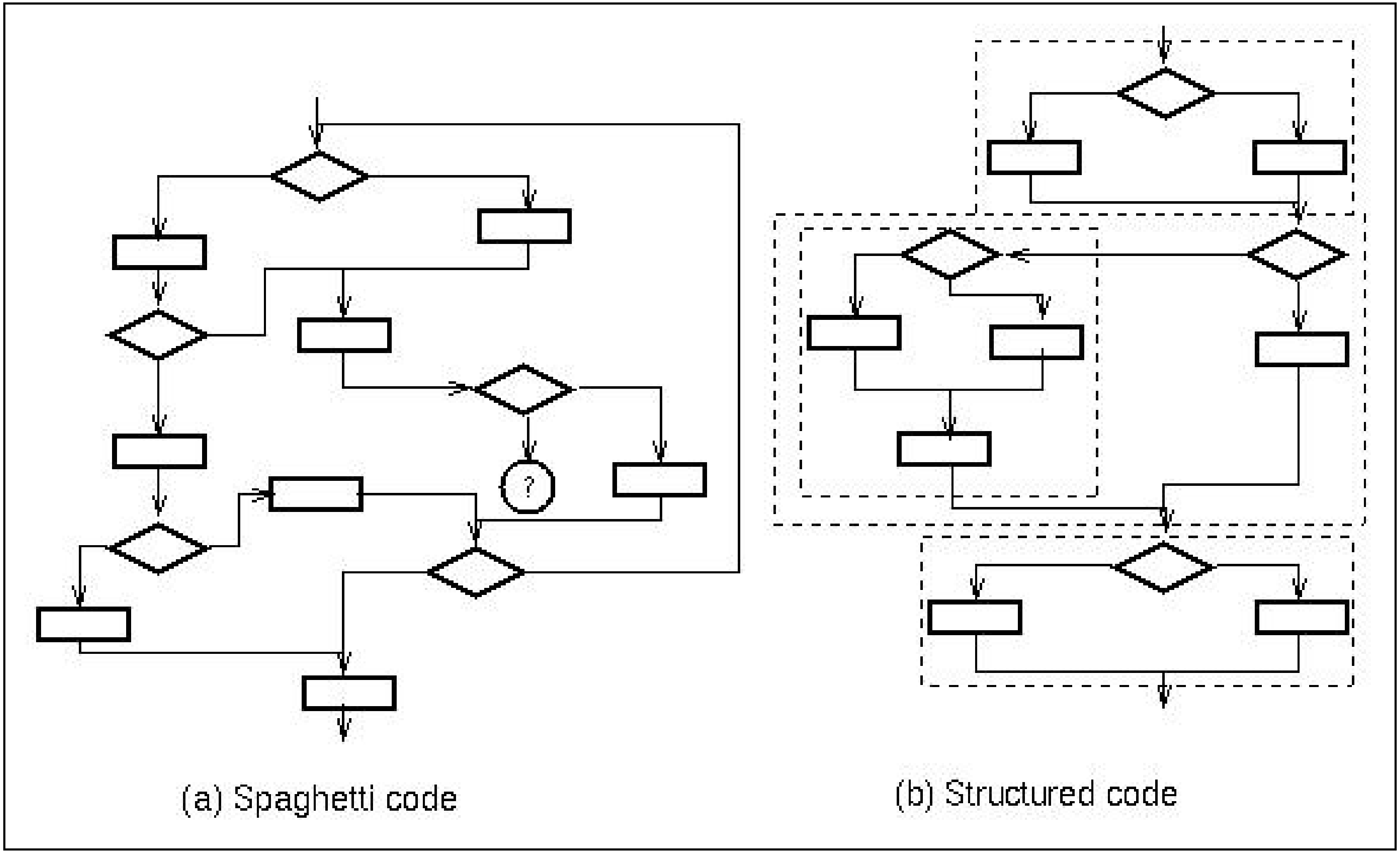
Fortran Control Structure

```
10 IF (X .GT. 0.000001) GO TO 20
11 X = -X
     IF (X .LT. 0.000001) GO TO 50
20 IF (X*Y .LT. 0.00001) GO TO 30
      X = X-Y-Y
30  X = X+Y
      ...
50 CONTINUE
      X = A
      Y = B-A
      GO TO 11
      …
```

Historical Debate

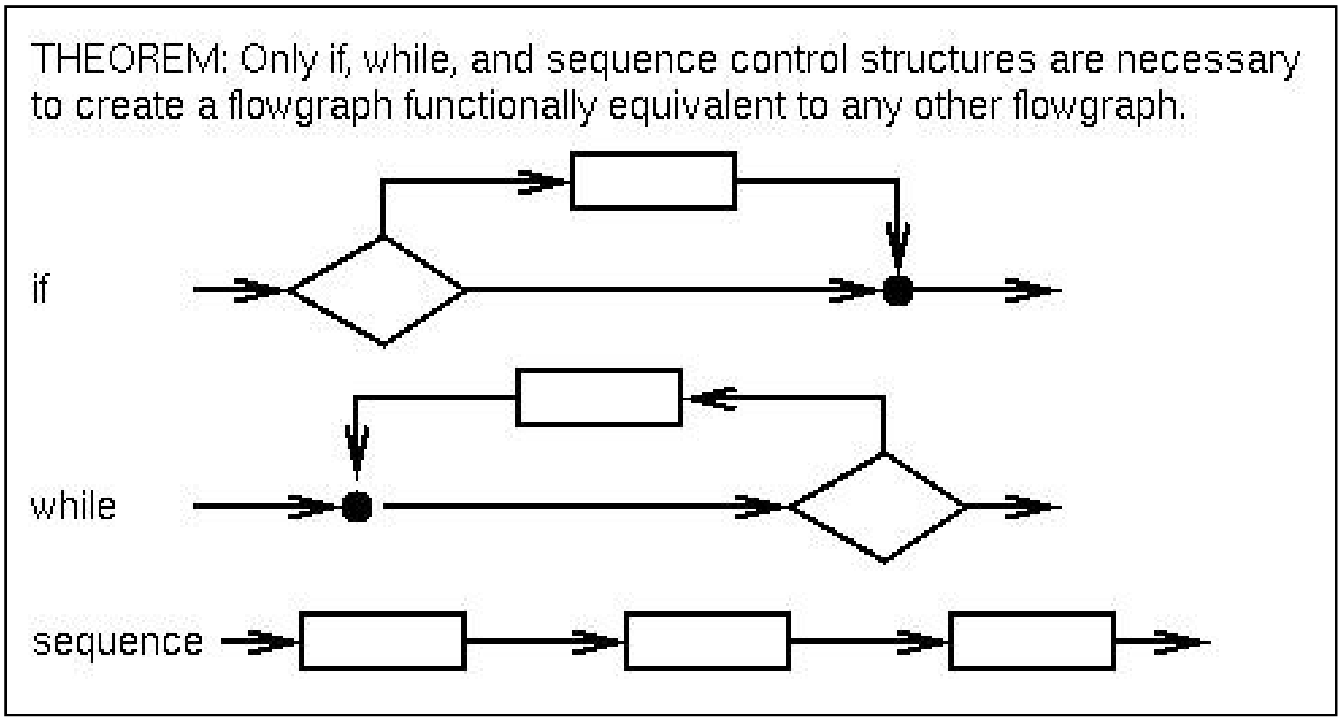
- Dijkstra, Go To Statement Considered Harmful
  - Letter to Editor, *C ACM*, March 1968
  - Now on web: http://www.acm.org/classics/oct95/
- Knuth, Structured Prog. with go to Statements
  - You can use goto, but do so in structured way …
- Continued discussion
  - Welch, GOTO (Considered Harmful)^n, n is Odd
- General questions
  - Do syntactic rules force good programming style?
  - Can they help?

Spaghetti code



Structured programming

- Issue in 1970s: Does this limit what programs can be written?
- Resolved by Structure Theorem of Böhm-Jacobini.
- Here is a graph version of theorem originally developed by Harlan Mills:



Advance in Computer Science

- Standard constructs that structure jumps

  `if … then … else … end`

  `while … do … end`

  `for … { … }`

  `case …`

- Modern style
  - Group code in logical blocks
  - Avoid explicit jumps except for function return
  - Cannot jump *into* middle of block or function body
- But there may be situations when “jumping” is the right thing to do!

Exceptions: Structured Exit

- Terminate part of computation
  - Jump out of construct
  - Pass data as part of jump
  - Return to most recent site set up to handle exception
  - Unnecessary activation records may be deallocated
    - May need to free heap space, other resources
- Two main language constructs
  - Declaration to establish exception *handler*
  - Statement or expression to *raise* or *throw* exception

Often used for unusual or exceptional condition, but not necessarily.

Summary of Control of Statement Execution

- Sequential
- Conditional Selection
- Looping Construct
- Must have all three to provide full power of a Computing Machine
- Sometimes jumps are needed!

What can you do in your projects now?

- Revisit your token grammer and CFG
- Test front end implementation techniques:
  - Recursive decent by hand
  - JavaCC, ANTLR, Jflex/CUP, SableCC
  - Use a toy language or a subset of your own language
- Generate AST
- Make a pretty printing tree walker
  - Composit, Visitor (GOF, static overloading, reflexsive)
  - Test that programs you input come out roughly the same!
- Make a scope and type checking tree walker