Algol

CS4100

From *Principles of Programming Languages: Design, Evaluation, and Implementation (Third Edition),* by Bruce J. MacLennan, Chapters 3and 4, and based on slides by Istvan Jonyer.

After FORTRAN

- International language is needed
  - – 1964: New language is proposed to break away from platform dependence
  - – Preliminary spec: NPL (new programming language), then PL/I (programming language 1)
  - – PL/I is too big
    - Dijkstra: If Fortran is an infantile disorder, then PL/I is a fatal disease
    - Trying to be everything to everyone backfires

Chapter 3: Generality and Hierarchy: ALGOL-60

- An international language is needed
  - – A single, universal language would be valuable
  - – International (American and European) committee is set up to make recommendations
  - – Algol-58 is created in 8 days in Zurich, as a preliminary report
  - – Algol: Algorithmic Language

Implementations

- Because of the hype, many started implementation quickly
  - – This resulted in many dialects
  - – JOVIAL (Jules' Own Version of the International Algebraic Language)
- Committee meets again in 1960 to incorporate suggestions
  - – Algol-60 is born and is very different from the '58 report.
  - – Report is 17 pages long: remarkable achievement, mainly due to BNF notation (reports used to stretch to hundreds or thousands of pages)

Algol Report

- 1959 UNESCO Conference on Information Processing
  - – Backus presents a description of Algol '58
    - Uses formal syntax he developed
  - – Naur is editor of Algol Bulletin
    - Disagrees with some of Backus' interpretation
    - Need for more precise description
    - Develops a variant of Backus' formal syntax

Backus-Naur Form, aka BNF used for 1960 Algol Report

Algol's Objectives

- The language should be very close to mathematical notation
- Should be useful in publications to describe algorithms
- Mechanically translatable to machine code

Structural Organization

- Hierarchically structured language
  - Nesting is introduced (Fortran did not use nesting)
  - Control structures can also be nested
    - One can be made the body of the other

```
if N > 0 then
  for i := 1 step 1 until N do
        sum := sum + Data[i]
```

    - Advantage: decreases the number of GOTOs required
- Reserved words

Constructs

- Declarative or Imperative
  - (like in FORTRAN)
- Variable declarations
  - Types: integer, real, Boolean

```
integer i, j, k
```

  - Lower bounds of arrays need not be 1

```
real array Data[-50:50]
```

  - Switch, like FORTRAN's computed GOTO
- Subprogram declarations
  - Keyword: `procedure` and
  - Procedures can be *typed* (functions) and *untyped*

```
real procedure dist(x1, y1, x2, y2);
  real x1, y1, x2, y2;
  dist = sqrt((x1 – x2)^2 + (y1 – y2)^2)
```

Imperative Constructs

- Computational
  - Assignment: "variable := expression"
  - Operators:
    - Arithmetic: +, -, *, etc.
    - Relational: =, <, >, ≥, etc.
    - Logic: ∧, ∨, ¬, etc.
  - Why is assignment ':=' and not '='?
    - Assignment is different from definition and comparison
    - I = I + 1 ; I := I + 1

Imperative Constructs

- Control-flow
  - All imperative constructs alter flow of control (except assignment)
  - Has *if-then-else*
  - *for*-loop replaces *do*-loop
- No input/output constructs
  - I/O was left to be handled by platform-dependent library calls

Name Structures

- Algol-60 introduces the compound statement
  - Where 1 statement is allowed, more can be used, using begin-end

```
for i := 1 step 1 until N do
        sum := sum + Data[i]

for i := 1 step 1 until N do
  begin
        sum := sum + Data[i];
        Print Real (sum)
  end
```

  - Also, the body of a procedure is a single statement

Syntax - Program

- <program> ::= <block> | <compound statement>

- <block> ::= <unlabelled block> | <label>: <block>
- <compound statement> ::= <unlabelled compound> | <label>: <compound statement>

- <unlabelled compound> ::=
  begin <compound tail>
- <unlabelled block> ::=
  <block head> ; <compound tail>

Syntax - Block

- <block> ::= <unlabelled block> |
  <label>: <block>
- <unlabelled block> ::=
  <block head> ; <compound tail>
- <block head> ::= begin <declaration> |
  <block head> ; <declaration>
- <compound tail> ::= <statement> end |
  <statement> ; <compound tail>

Syntax - Statement

- <compound statement> ::= <unlabelled compound> |
  <label>: <compound statement>
- <unlabelled compound> ::= begin <compound tail>
- <compound tail> ::= <statement> end | <statement> ; <compound tail>

- <statement> ::= <unconditional statement> | <conditional statement> |
  <for statement>
- <unconditional statement> ::= <basic statement> |
  <compound statement> | <block>
- <basic statement> ::= <unlabelled basic statement> |
  <label>: <basic statement>
- <unlabelled basic statement> ::= <assignment statement> |
  <go to statement> | <dummy statement> |
  <procedure statement>

Name Binding

- Fortran binds a variable to a single memory location statically
- Algol-60 included the results of research done on name structures, which were problematic in Fortran
  - Sharing of data between subprograms
  - Parameter passing modes
  - Return values
  - Dynamic arrays
- Result of research: block structure

Blocks Define Nested Scopes

- Fortran
  - Had local and global declarations only
  - Had to redeclare using COMMON to share
- Algol-60
  - Introduces blocks

```
begin
  declarations;
  statements
end
```

  - Compound statements do not have ‘`declarations`’.
  - All declarations are visible to all statements in the block
  - Since statements can be blocks, scopes can be nested

Why do we need scopes?

- Reduce the context programmers have to keep in mind
- Make understanding and maintenance of program easier
- Scopes reduce visibility of names
  - Declare variable only where needed and used
- Nested blocks inherit names from outside
  - It would be very inconvenient if they did not

“COMMON” with Blocks

- The error-prone COMMON in Fortran can be implemented in a much better way using blocks

```
begin
  integer array Name, Loc, Type[1:100];
  procedure Lookup (n);
          . . . Lookup procedure . . .
  procedure Var (n, l, t);
          . . . Var procedure . . .
  procedure Array1 (n, l, t, dim1);
          . . . Array1 procedure . . .
end
```

Too Much Access

- Blocks provide "indiscriminate access"
  - Since functions must be accessible to users,
  - and data structures must be accessible to functions
  - → Data is also accessible to users
- Violates information hiding principle

Contour Diagrams

- Inner blocks implicitly inherit access to all variable in immediately surrounding block
- Names declared in a block are local to the block
- Names declared in surrounding blocks are nonlocal
- Names declared in outermost block are global

Contour Diagrams

- See Figure 3.3, page 102
- Do Exercise 3-1, page 104

Dynamic vs Static Scoping

- Static scoping
  - Procedure is called in the context of its declaration
    - Environment of Definition
  - Scope structure is determined at compile-time
  - Algol
- Dynamic scoping
  - Procedure is called in the context of its *caller*
    - Environment of Caller
  - Scope structure is determined at run-time
  - LISP

Example

- Draw static contour diagram
- Draw dynamic contour diagram for both calls to P

```
a:begin
    integer m           outer m
    procedure P
      m := 1;
  b:begin
      integer m;        inner m
      P                 inner call
    end
   P                    outer call
  end
```

Dynamic Scopes and Functions

- Dynamic scoping applies to all names (not just variables)
- Advantage:
  - We can write a general procedure that makes use of procedures in the caller's environment
    - No need to have all names defined in static context
- Disadvantage:
  - If caller's environment provides a different function than what is intended to be used (see example page 109)
    - Programmer has to think about envt when writing calls

Which one is better?

- General rule:
  - What is natural to humans will cause less problems in the long run
  - If humans can understand static scoping better, than it will result in higher quality programs in the long run
- Dynamic scoping is confusing
  - Generally rejected (not used in new languages)
  - Static scoping agrees more with the program's dynamic behavior

Blocks Permit Efficient Storage Management

- Fortran used EQUIVALENCE
  - Not safe, since there is no guarantee of exclusive use of memory
- Blocks permit reuse of memory

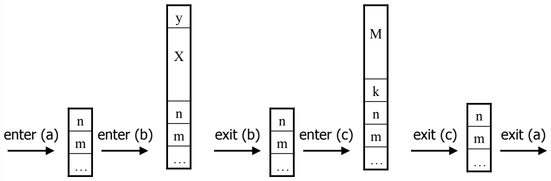
```
a:begin integer m, n;
  b:begin real array X[1:100], real y;
  ...
  end
...
  c:begin integer k; real array M[0:50];
  ...
  end
end
```

Run-Time Stacks

- Variables in blocks *b* and *c* are never used at the same time
- When exiting *b*, its variables may be discarded
- Notice: Block entered last will be exited first
  - LIFO (last-in first-out) order
  - Can use a stack to organize activation records
  - When block is entered, its AR is pushed onto stack
  - When block is exited, its AR is popped off stack
  - Assumption: No local variables are initialized

Example

- From previous program



Responsible Design

- Algol designers did not include EQUIVALENCE
  - Programmers might have asked for it…
  - Instead, they looked at the root of the problem
  - "Don't ask what they want, ask how the problem arises"
  - Language designers are responsible for the features in the language, not programmers

Principles of Programming

- The Responsible Design Principle
  - Do not ask programmers what they want, find out what they need.

Data Structures

- Primitives
  - Mathematical scalars, like in Fortran
  - integer, real, Boolean
  - complex and double not included
- Double: platform dependent
  - Not portable
  - Why? Because we need to know the size of a word to know how big double is.
  - Alternate approaches:
    - specify precision
    - Let compiler pick precision

Why no complex?

- Not primitive
  - Can be constructed using other types easily (2 reals)
- Is it easy to use *real*s for complex?
  - Yes, but inconvenient
  - Need supporting operations
    - ComplexAdd(x, y, z), etc.
- Designers' choice:
  - Is it worthwhile to add the complexity/overhead of another type? (conversions, coercion, operator overload, etc.)
  - Will they get enough use?

Strings

- Yet another data structure that needs full support (operation, etc.)
- Algol designers included strings as second-class citizens
  - `string` type is only allowed for formal parameters
  - String literals can only be actual parameters
  - No operations
  - Strings can only be passed around in procedures
  - Cannot actually *do* anything with them
- What's the point???
  - String will end up getting passed to output procedure written in a lower (machine) language that can handle it

Zero-One-Infinity

- Programmers should not be required to remember arbitrary constants
- Fortran examples
  - Identifiers have max. 6 characters
  - There are at most 19 continuation cards
  - Arrays can have at most 3 dimensions
- Regularity in Algol requires small number of exceptions
  - Gives rise to Zero-One-Infinity principle
  - E.g.: Identifier names should be either 0, 1 or unlimited length. (0 & 1 don't make much sense)

Principles of Programming

- The Zero-One-Infinity Principle
  - The only reasonable numbers in programming language design are zero, one and infinity.

Arrays are Generalized

- Arrays can have any number of dimensions
- Lower bound can be number other than 1
  - More intuitive, and less error prone than fixed lower bound
- Arrays are dynamic
  - Array bounds can be given as expressions, which allows recomputation every time the block is entered
  - Array size is set until block is exited
- (Fortran had fixed array sizes.)

Strong Typing

- Strong typed language
  - Prevents programmer to perform meaningless operations on data
  - Not to be confused with legitimate type conversions (integer + real (coercion))
- Fortran
  - Weakly typed
  - Permits adding to a Hollerith constant, etc.
  - Equivalence allows setting up the same memory for different types
    - Security and maintenance problem
    - Intentional type violation is not portable
- Exception: System programming (C)
  - Have to treat memory cells as raw storage without regard to type

Control Structures

- Primitive statements are similar to Fortran's
  - Assignment
  - Control flow
  - No input/output

Controls are Generalized: *if*

- Fortran had many restrictions
  - `if (exp) simple statement`
    - Statement restricted to GOTO, CALL, or assignment
- Algol removes restrictions
  - All statements are allowed (even 'if' in body of 'if')
  - 'else' added to address *false* condition

Controls are Generalized: *for*

- Algol's *for* is more general than Fortran's *do*

```
for i := 1 step 1 until N do
      sum := sum + Data[i]
```

  - Leading-decision loop:

```
for NewGuess := Improve(OldGuess)
  while abs(NewGuess – OldGuess) > 0.01
  do OldGuess := NewGuess
```

  - Same as while loop in newer languages:

```
NewGuess := Improve(OldGuess);
while abs(NewGuess – OldGuess) > 0.01 do
  begin
      OldGuess := NewGuess;
      NewGuess := Improve(OldGuess);
  end
```

Another for loop

```
for i := 3, 7,
      11 step 1 until 16,
      i ÷ 2 while i >= 1,
      2 step i until 32 do
   print( i );
```

3 7 11 12 13 14 15 16 8 4 2 1 2 4 8 16 32

Goal: Regularity

- Algol was designed around regularity
  - "Anything that you think you ought to be able to do, you will be able to do."
  - Elaboration on zero-one-infinity principle
    - Remove inexplicable exceptions from the language

begin … end

- Algol-58:
  - All control structures should be allowed to have any number of statements
  - All control statements were considered an opening bracket, with corresponding closing bracket
    - if … endif
- Algol-60
  - Largely due to the BNF notation, they realized that one bracketing mechanism is enough for all
  - Defined *begin-end* bracketing
    - Define compound statements
    - Makes one statement out of a group of statements
    - Allowed anywhere a single statement is expected

Example

```
for i := 1 step 1 until N do
     sum := sum + Data[i]

for i := 1 step 1 until N do
 begin
     sum := sum + Data[i];
     Print Real (sum)
 end
```

begin-end Issues

- Easy to omit begin-end
  - Especially when single statement is used first, then another is added
  - Especially the case with well-indented code

```
for i := 1 step 1 until N do
     sum := sum + Data[i];
     Print Real (sum)
```

  - This is a maintenance problem
  - Good convention: always use bracketing

begin-end Has Double Duty

- begin-end are used for
  - Compound statements
    - Collection of statements is handled as one statement
  - Blocks
    - Define nested scopes
    - Include definitions, in addition to statements
- Any difference?
  - Compound statements do not need an activation record
  - Compiler must determine whether begin-end has declarations, and generate block-entry code if so

Structured Programming

- Compound statements drastically reduce the number of GOTOs required
  - In Fortran, GOTO was the workhorse for control
  - Example: *if-then-else*
- GOTO-less programs were easier to read
  - This led people to experiment with abolishing GOTO
  - Dijkstra: "Go To Statement Considered Harmful"
    - Difficulty in reading programs came from conceptual gap between static and dynamic structure of program
    - i.e.: static layout on paper, versus runtime operation
    - Result: languages still have GOTOs, but we don't use them

Principles of Programming

- The Structure Principle
  - The static structure of the program should correspond in a simple way to the dynamic structure of corresponding computations.

Procedures are Recursive

- Recursive definitions are frequent in math and science
  - Define thing in terms of itself
  - Example:
    - Factorial: n! =
      n * (n – 1)! if n > 0
      1 if n = 0
- Algol permits recursive procedures

```
integer procedure fac(n);
  value n; integer n;
  fac := if n = 0 then 1 else n*fac(n-1);
```

  - 'n = 0' is called the stopping condition

Implementing Recursion

- What happens to local variable *n* on recursive call?
  - fac(3) is called, then fac(2), then fac(1), then fac(0)
  - Would location holding 3 be overwritten?
    - Yes, if same activation record was used
  - Solution:
    - Create new activation record for each invocation of *fac*()

Parameter Passing

- Modes in Algol
  - Pass by value
  - Pass by name
- Two modes attempt to distinguish between *input only* and *input/output* parameters

Pass by Value

```
integer procedure fac(n);
  value n; integer n;
```

- First part of pass by value-result (in Fortran)
  - Actual copied into variable corresponding to formal
  - Secure; local variable will not overwrite actual parameter
  - Does not allow output parameters (input only)
  - Inefficient for arrays (or other non-primitive data structures, in general)
    - Copy must be made of entire array in activation record
    - Copying takes time

Pass by Name

- Based on substitution
  - Consider

```
integer procedure Inc(n);
  integer n;
  n := n + 1;
```

  - And the call `Inc(i)`
- We need output parameter that will effect `i`, not just local `n`
  - Acts like `i` is substituted for `n`

```
i := i + 1
```

Copy Rule

- Procedure can be replaced by its body with actuals substituted for formals
- Revised Report 4.7.3
- Body of `Inc(n)`
  - `i := i + 1`
  - `A[k] := A[k] + 1`
- Not how it is implemented

Pass by Name is Powerful

- Evaluate the following using pass by value, reference, and name

```
procedure S(el,k);
  integer el, k;
  begin
         k := 2;
         el := 0;
  end
A[1] := A[2] := 1;
i := 1;
S(A[i], i)
```

Value	A[1] = 1, A[2] = 1, i = 1
Reference	A[1] = 0, A[2] = 1, i = 2
Name	A[1] = 1, A[2] = 0, i = 2

“Thunks”

- Implementing pass by name
  - Passing the text?
    - Would need to compile at runtime
      - not possible
  - Copying compiled code?
    - Would increase size of code…
  - Solution: “Thunks”
    - Pass address to compiled code
    - Address of memory location is returned to callee to use as variable

Pass by Name is Dangerous!

```
procedure Swap(x, y);
  integer x, y;
  begin integer t;
         t := x;
         x := y;
         y := t;
  end
```

- What is the effect of
  - `Swap(A[i], i)?`
  - `Swap(i, A[i])?`

- `Swap(r,s), where r=1,s=2`

```
procedure Swap(x, y);
  integer x, y;
  begin integer t;
         t := r;          t=1
         r := s;          r=2
         s := t;          s=1

  end
```

- `Swap(A[i], i) where A[i]=27, i=1`

```
procedure Swap(x, y);
  integer x, y;
  begin integer t;
         t := A[i];  t=27
         A[i] := i;  A[1]=1
         i := t;     i=27

  end
```

- `Swap(i, A[i]), where i=1, A[i]=27`

```
procedure Swap(x, y);
  integer x, y;
  begin integer t;

         t := i;      t=1
         i := A[i];   i=27
         A[i] := t;   A[27]=1

  end
```

Pass-by-name

- It can be shown that there is no way to define swap in Algol-60 that works for all parameters
- Design mistake when a simple (common) procedure has such surprising properties

Parameter Passing Modes

- Pass by value
  - Bind to value at time of call
  - Preserves actual (no output parameters)
  - Inefficient for arrays
- Pass by reference
  - Bind to address at time of call
  - Changes actual (can be used for output)
  - Efficient for all data types
- Pass by name
  - Bind to address of thunk at time of call
  - Changes actual (can be used for output)
  - Efficient, but expensive

Out-of-Block GOTOs

```
A: begin array x[1:100];
        ...
   B:  begin array y[1:100];
         ...
        goto exit;
         ...
        end;
exit:
  end
```

- What happens to activation records?
  - Program continues in different block
  - Pop activation record
  - Makes goto complicated

Even worse…

```
begin
  procedure P(n);
  value n; integer n;
         if n = 0 then goto out
         else P(n-1);
    P(25);
out:
end
```

- Recursive call 25 times then jump to `out`
  - Pop 25 activation records
- Data dependent, can't know at compile time
  - Implement goto as call of run-time routine

Feature Interaction

- Example:
  - GOTOs are simple
  - Recursion is simple
  - Combination is very messy
- In theory, each feature must be tested with every other one to avoid unintended consequences
- 100 features:
  - Every pair: 100x100 = 10,000 combinations
  - Every three: 100^3 = 1,000,000
  - …

The *for*-loop is Very General

```
for var := exp step exp2 until exp3 do stat
for var := exp while exp2 do stat
```

- Expressions can be any arithmetic expression, including
    - `for i := i/2 while i>1 do stat`
  - Lists
    - `for days := 31, 28, 31,30, 31, 30 do stat`
  - Conditional expressions (vs. conditional statements!)

    -
      ```
      for days := 31,
            if mod(year, 4) = 0 then 29 else 28,
           28, 31, 30, 31, 30 do stat
      ```
  - Combinations of above

    -
      ```
      for i := 3, 7,
                i/2 while i>1,
                11 step 1 until 16
            do stat
      ```

- <for statement> ::= <for clause> <statement> |
  <label>: <for statement>
- <for clause> ::= **for** <variable> := <for list> **do**
- <for list> ::= <for list element> | <for list> , <for list element>
- <for list element> ::= <arithmetic expression> |
  <arithmetic expression> **step** <arithmetic expression>
  until <arithmetic expression> |
  <arithmetic expression> **while** <Boolean expression>

- **for** q:=1 **step** s **until** n **do** A[q]:=B[q]
- **for** k:=1,V1x2 **while** V1<N **do**
  for j:=I+G,L,1 **step** 1 **until** N, C+D
  do A[k,j]:=B[k,j]

Baroque Features

- Fascination-oriented features of little use
  - They did it because they could
  - Getting away from assembly languages as far as possible
- Baroque takes on pejorative meaning

Baroque

- 1 : of, relating to, or having the characteristics of a style of artistic expression prevalent especially in the 17th century that is marked generally by use of complex forms, bold ornamentation, and the juxtaposition of contrasting elements often conveying a sense of drama, movement, and tension
- 2 : characterized by grotesqueness, extravagance, complexity, or flamboyance
- 3 : irregularly shaped —used of gems <a baroque pearl>
- baroque. (2009). In Merriam-Webster Online Dictionary. Retrieved April 28, 2009, from http://www.merriam-webster.com/dictionary/baroque

Handling Cases: `switch`

```
begin
   switch wageStatus = fulltime, parttime, hourly;
   ...
   goto wageStatus[i];
fulltime:        ...handle fulltime case...
                 goto done;
parttime:        ...handle parttime case...
                 goto done;
hourly:          ...handle hourly case...
                 goto done;
done:   ...
end;
```

- Elaboration on computed GOTO of Fortran (and IBM 650)
- Confusing, since switch, goto, and labels can be anywhere in the program
- Label list can contain conditionals (`if i>0 then M else N`)

Machine Independence

- Get away from formats tied to particular computers, punch cards -> free format
- How should a program be formatted?
  - Left justify, one statement per line
  - Like English sentence
  - Structured (hierarchical)
    - Obeys structure principle
- Most languages followed Algol in free format

Machine Independence

- Representation issues
  - Different hardware
    - Input devices
    - Character sets
  - Different conventions
    - Math vs cs
    - American vs European
    - Comma (European) vs point (American) almost defeats Algol

Machine Independence

- Theorem: The more trivial the point the more vehemently people will fight over it
- Which symbols
  - Only those available in all sets
    - Too limiting
  - Independent of particular sets
    - chosen

Compromise

- Three representations
  - Reference language used in language specifications
    - E.g. "up arrow"
  - Publication language used in publications
    - E.g. sub- and super-scripts
  - Hardware language to be used by implementers
    - Use appropriate character set
    - I/O for the computer system

Lexical Conventions

- Reserved words
  - Cannot be used as identifiers
  - Most languages
- Key words
  - Words used by language are marked
    - E.g. Different font or bold
    - Hard to type
  - Algol
- Keywords in context

Keywords in context

- FORTRAN
- Words used by language are only keywords in context where expected
  - Hard to catch errors
- Legal in PL/I

```
IF IF THEN
    THEN = 0;
ELSE
    ELSE = 0;
```

Some Design Considerations

- From David Billington, *The Tower and the Bridge* 1993
- Techological Activities
  - Values
    - Efficiency
    - Economy
    - Elegance
  - Dimensions
    - Scientific
    - Social
    - Symbolic

Efficiency

- Materials used
- Scientific Issue
- Memory
- Time
  - Programmer
  - Compiler
  - Run

Economy

- Cost-benefit
- Social Issue
- Benefit to programming community
- Cost: trade-offs
  - Computer vs programmer time
  - Increasing cost of residual bugs
  - Program maintenance vs development

Economy

- Social Influences
  - Manufacturer support
  - Prestigious universities teach
  - Approved by influential organizations
  - Standardized
  - Used by "real" programmers
- Monetary values are unstable as is social climate

Elegance

- Under-engineered
  - Risk of unanticipated interactions
- Over-engineered
  - Inefficient or uneconomical
- Can't always rely solely on mathematical analysis
  - Always incomplete
    - Simplifications
    - assumptions

Elegance

- General Principle: Designs that look good are good
- Function follows form
  - But needs to be deep (not superficial)
- Should be a joy to use
  - Comfortable and safe

Elegance

- Aesthetics comes from experience
- Design obsessively
  - Criticize
  - Revise
  - Discard

In Summary, Algol

- Never had widespread use
  - No I/O
  - Competing directly with FORTRAN
- Major milestones
  - Block-structured
  - Nested
  - Recursive
  - Free-form
  - BNF - mathematical theory of formal languages

Algol by reputation

- General
- Regular
- Elegant
- Orthogonal

Second Generation

- Elaborations and generalizations of first generation
  - Strong typing of built-in types
  - Name structures hierarchically nested
  - Structured control structures
    - Recursion
    - Parameter passing
  - Syntactic structures
    - Machine independent
    - Moving away from fixed formats