# Comparative Programming Languages

hussein suleman
uct csc304s 2003

## Course Structure

- 15 lectures
- 2 assignments
  - 1 x 2-week programming assignment
  - 1 x 1-week "written" tutorial
- open-book final (1/2 paper)

## Course Topics

- overview of paradigms
- evolution of languages
- assignment and expressions
- types, variables, binding and scope
- pointers and memory management
- control structures, subprograms
- runtime execution
- exceptions
- concurrency
- visual languages
- scripting languages

# Overview of Paradigms and PL Issues

## Why study programming languages?

- Understand differences and similarities.
- Understand concepts and structures independently of languages.
- Learn reasons for design decisions in particular languages.
- Develop the ability to choose appropriate languages and paradigms for particular tasks.
- Develop the ability to design languages.

## Common paradigms

**paradigm** *n.* pattern, model - Dorling Kindersley Dictionary

- Procedural
  - a.k.a. Imperative
    - imperative = peremptory, urgent, necessary
      - peremptory = imperious, dictatorial, precluding opposition
        - imperious = dominating, dictatorial
- Object-oriented
- Declarative
  - a.k.a. Logic
- Functional

## Examples

| Language | Paradigm |
|---|---|
| C | |
| C++ | |
| Java | |
| Clean | |
| Prolog | |
| Assembly Language | |
| Visual C++ | |
| HTML | |
| C# | |
| Javascript | |

## (Programming) Languages

- Languages are used to standardise communication.
- What makes it a programming language?
- Is HTML a programming language? Javascript? SQL?
- 3 traditional concepts:
  - sequence
  - selection
  - iteration

# Issues in comparing languages

- Simplicity
- Orthogonality
- Control structures
- Data types and type checking
- Syntax
- Abstractions
- Exceptions
- Aliasing

# Issue: Simplicity

- Languages that are complex to read and write are less likely to be understood by a large group of individuals equally well.
- Many different ways of expressing the same construct may be easier for writing but not for reading/translating.
  - Examples:

    ```
    if (a<b) { c=d } else { c=e };
    c = a<b?d:e
    c = e; c=d if (a<b);
    ```
- Is simplicity always good?

# Issue: Orthogonality

- Does the same set of rules apply to all constructs and data structures?
- If it seems reasonable that a feature should work, does it?
- Mathematical perspective:
  - Orthogonal vectors can form the basis for a vector space. (0,1) and (1,0) are orthogonal vectors so every 2-element vector (x,y) can be expressed as “a(0,1)+b(1,0)”.
  - Similarly orthogonal language features allow all possible combinations / programs.
- Is orthogonality always good?

# Issue: Control Structures

- Transfer control within a program to effect variable behaviour.
- Supports selection and iteration within languages.
- “goto considered harmful”
  - How does the inclusion of goto affect the ability of programmers to read, write and maintain code?
- Dijkstra and structured programming

## Issue: Data Types and Checking

- Supply appropriate types and type mechanisms that are meaningful and self-regulating.
- Example:
  - Enumerations restrict values to a small pre-defined set while implicit enumerations using an integer do not benefit from strong typing.
- More complex data structures make it easier to understand the logic.
- Run-time vs. compile-time type checking
  - Mathematica has to type-check at runtime, therefore less efficient!

---

## Issue: Syntax

- Complexity of syntax makes languages more difficult to learn/understand.
- "Arbitrary" restrictions, like identifier formats, have to be learnt.
- The same symbols/keywords with multiple meanings dependent on context decreases readability of code.
  - Examples:
    - "=" for equality testing and assignment in Pascal.
    - "=0" for pure virtual functions and assignment in C++.

## Issue: Expressivity

- Program writability is supported by concise syntax to express general and complex functions.
  - Example:
    - Perl regular expressions can perform widely varying functions with single lines of code:
      - s/(?<!wo)man/woman/go replaces all man with woman
      - s/([^ ]+) (.*)/$2 $1/go moves first word to last
      - /[a-zA-Z_][a-zA-Z0-9_]*/ checks for valid identifiers
- Is expressivity good for maintenance?

---

## Issue: Abstractions

- Data abstractions hide the details of complex data structures.
  - Example:
    - The C++ Standard Template Library
- Process abstraction hides the details of complex algorithms and processes.
  - Example:
    - Python modules
- Object-oriented programming supports both approaches!

# Issue: Exceptions

- Exceptions are special circumstances that must be handled in a non-standard manner in programs.
- Exceptions must be handled immediately or promoted to a higher level of abstraction.
  - Example:

    `try { … } catch (Exception e) { … }`
- Why do we use exceptions?

# Issue: Aliasing

- Multiple names for a single memory location.
- Aliases break the 1-to-1 correspondence between variables and storage locations, therefore affecting readability.
- Pointer-based data structures with aliases affect memory management strategies.
- Union data types circumvent strong type-checking.

## Machine architecture effects

- von Neumann computers are best suited for imperative languages.
- Object-oriented languages are an abstraction over imperative languages.
- Functional languages are frequently interpreted or compiled along with dynamic type-checking code to fit the imperative model.
- Declarative languages rely on an inference engine to execute the "programs".

## Execution models

- Compilation
  - Program converted to native machine code and executed directly on processor.
    - e.g., Visual C++
- Interpretation
  - Program understood and logic acted upon by interpreter.
    - e.g., Prolog
- Hybrid
  - Program converted to intermediate representation, which is "executed" in virtual machine.
    - e.g., C#

# Evolution of Languages

---

## Plankalkül Example

```
  | X + 1 => Y
V | 0        0
K | 2        1
S | 1.n      1.n
```

Equivalent Java: Y0[1] = X0[2] + 1

## Plankalkül

| | |
|---|---|
| Etymology | Plan = Plan<br>kalkül = Calculus<br>Plankalkül = Programming calculus |
| Who? | Konrad Zuse |
| When? | 1945 |
| Why? | To express computations in data processing |
| Interesting Features | Primitive support for matrices/indices, assertions |
| Notes | Way before its time! |

## Fortran Example

```
      i = 1

   10 if (i .le. 100) then
         i = 2*i
         write (*,*) i
         goto 10
      endif


      if (j-2) 20, 30, 40


  40  goto (50, 60) j-2
```

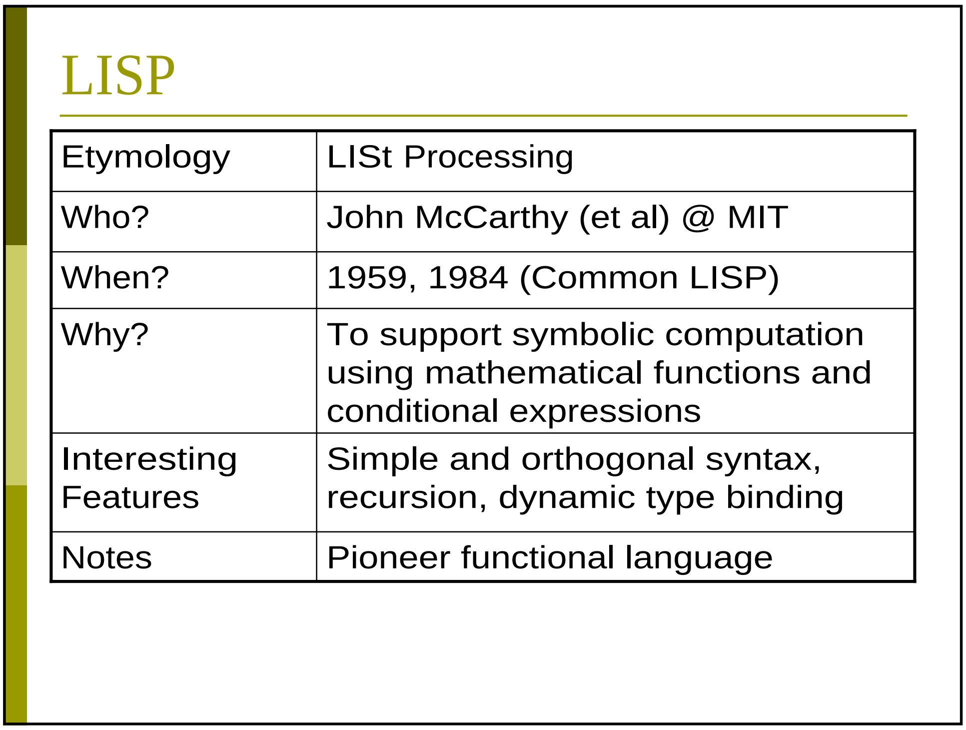
# Fortran

| | |
|---|---|
| Etymology | FORmula TRANslating system |
| Who? | John Backus (et al) @ IBM |
| When? | 1956 (I), 1958 (II), 1962 (VI), 1978 (77), 1992 (90) |
| Why? | Scientific computation – availability of IBM 704 machine |
| Interesting Features | Implicit typing based on name of variable, independent compilation (II), arithmetic IF, computed GOTO |
| Notes | First compiled high-level language! |

---

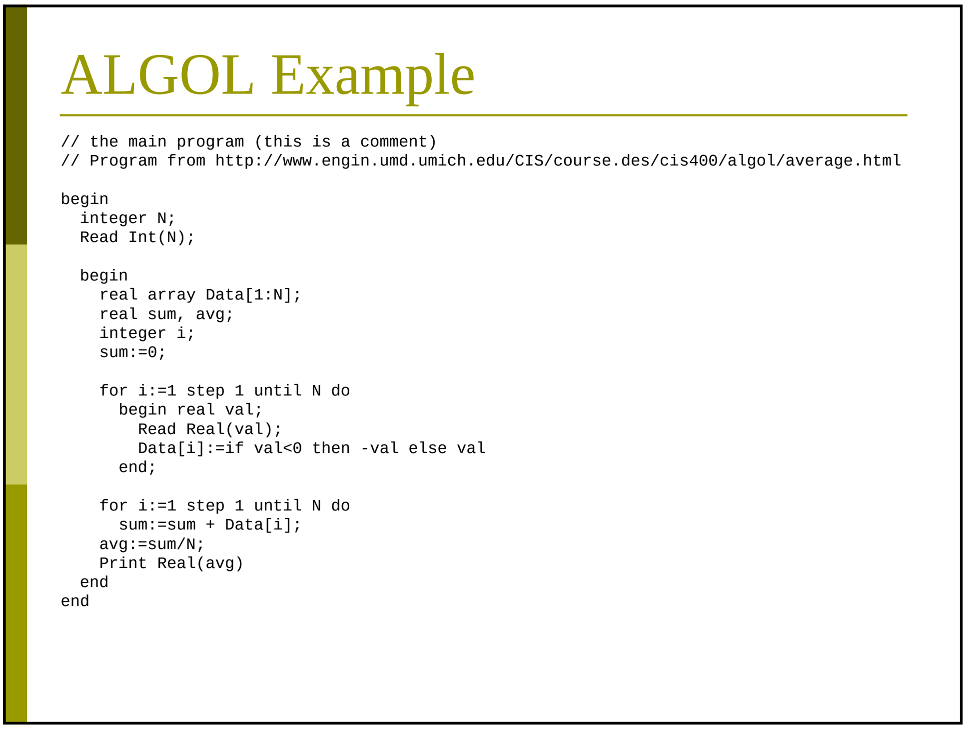
# LISP Example

```
;;; Simple towers of Hanoi program. Note that Start-Peg and Goal-Peg are
;;; integers from 1 to 3 indicating the peg number. Ie to move 4 discs,
;;; starting on the first peg and finishing on the last one, execute
;;; (Towers 4 1 3)
;;;
;;; 1992 Marty Hall. hall@aplcenmp.apl.jhu.edu

(defun Towers (Number-of-Discs Start-Peg Goal-Peg)
  (cond
    ((= 1 Number-of-Discs) (format t "~%Move Top Disc from peg ~D to peg ~D."
                                 Start-Peg Goal-Peg))
    (t                     (Towers (1- Number-of-Discs)
                                 Start-Peg
                                 (Remaining-Peg Start-Peg Goal-Peg))
                       (Towers 1 Start-Peg Goal-Peg)
                       (Towers (1- Number-of-Discs)
                                 (Remaining-Peg Start-Peg Goal-Peg)
                                 Goal-Peg))))

;;;==========================================================================
;;; Given two peg numbers, what is the peg number of the third peg?

(defun Remaining-Peg (Peg1 Peg2)
  (- 6 Peg1 Peg2))
```



# LISP

| | |
|---|---|
| Etymology | LISt Processing |
| Who? | John McCarthy (et al) @ MIT |
| When? | 1959, 1984 (Common LISP) |
| Why? | To support symbolic computation using mathematical functions and conditional expressions |
| Interesting Features | Simple and orthogonal syntax, recursion, dynamic type binding |
| Notes | Pioneer functional language |



# ALGOL Example

```
// the main program (this is a comment)
// Program from http://www.engin.umd.umich.edu/CIS/course.des/cis400/algol/average.html

begin
  integer N;
  Read Int(N);

  begin
    real array Data[1:N];
    real sum, avg;
    integer i;
    sum:=0;

    for i:=1 step 1 until N do
      begin real val;
        Read Real(val);
        Data[i]:=if val<0 then -val else val
      end;

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

# ALGOL

| | |
|---|---|
| Etymology | ALGOrithmic Language |
| Who? | GAMM / ACM – transatlantic group of representatives |
| When? | 1958, 1960, 1968 |
| Why? | To create a universal language |
| Ancestry | FORTRAN I |
| Interesting Features | Block structure / compound statements, BNF (60), Dynamic arrays, Call-by-name, Orthogonality in data types (68) |
| Notes | Machine-independent, formally specified |

# COBOL Example

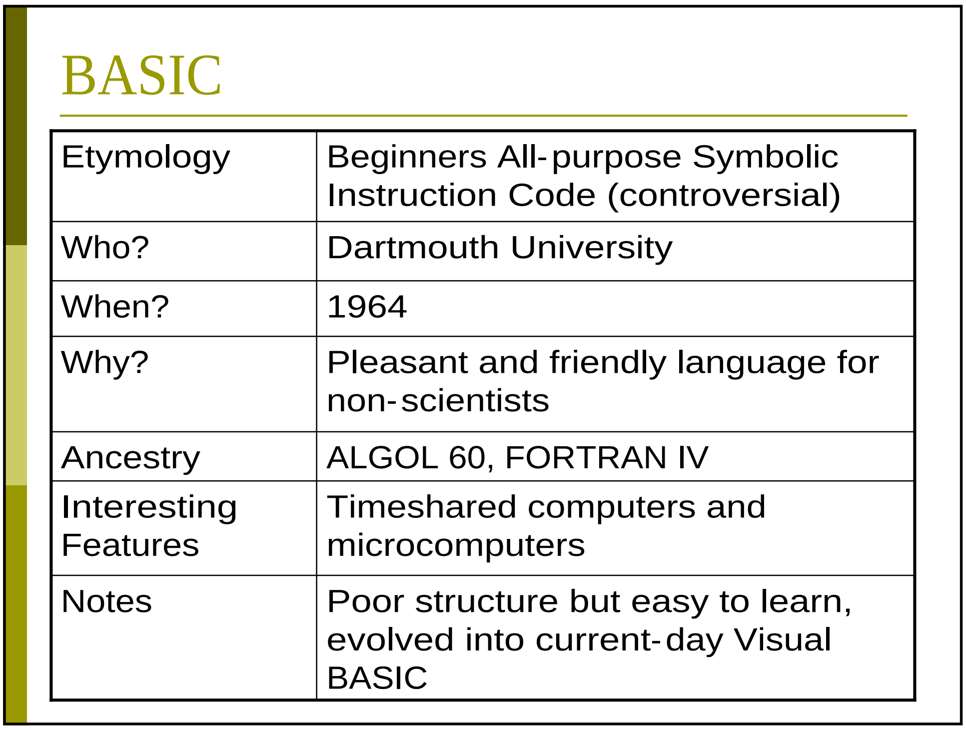
```
 $ SET SOURCEFORMAT"FREE"
IDENTIFICATION DIVISION.
PROGRAM-ID.  Multiplier.
AUTHOR.  Michael Coughlan.
* Example program using ACCEPT, DISPLAY and MULTIPLY to
* get two single digit numbers from the user and multiply them together

DATA DIVISION.

WORKING-STORAGE SECTION.
01  Num1                                PIC 9  VALUE ZEROS.
01  Num2                                PIC 9  VALUE ZEROS.
01  Result                              PIC 99 VALUE ZEROS.

PROCEDURE DIVISION.
    DISPLAY "Enter first number  (1 digit) : " WITH NO ADVANCING.
    ACCEPT Num1.
    DISPLAY "Enter second number (1 digit) : " WITH NO ADVANCING.
    ACCEPT Num2.
    MULTIPLY Num1 BY Num2 GIVING Result.
    DISPLAY "Result is = ", Result.
    STOP RUN.
```

# COBOL

| | |
|---|---|
| Etymology | COmmon Business Oriented Language |
| Who? | US Department of Defence |
| When? | 1960, 1968/74/85 (ANSI) |
| Why? | Easy to use, more English than scientific, broaden base of programmers |
| Interesting Features | Macros, hierarchical data structures, program divisions |
| Evaluation | Widespread use in business computing, especially electronic accounting |

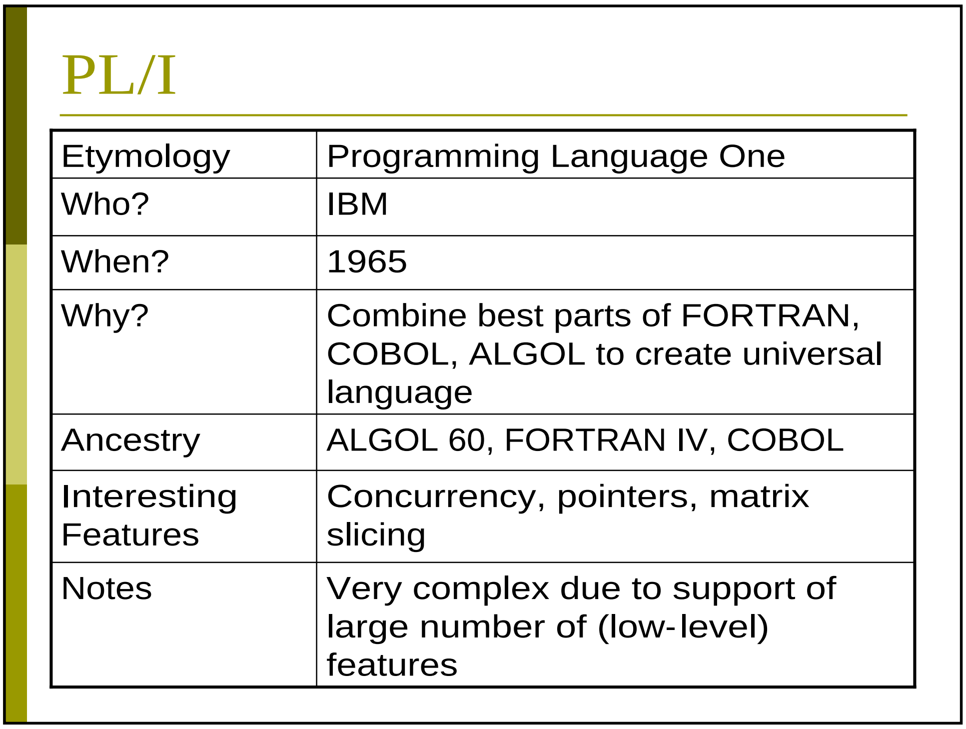
# BASIC Example

```
10 REM Very simple GW-BASIC program
20 REM
30 PRINT “How many students are in CSC304?“
40 INPUT N
50 FOR I=1 to N
60 GOSUB 90
70 NEXT I
80 GOTO 30
90 PRINT “Student number: “;I
100 PRINT “Enter a name”
110 INPUT NAME$
100 RETURN
```



# BASIC

| | |
|---|---|
| Etymology | Beginners All-purpose Symbolic Instruction Code (controversial) |
| Who? | Dartmouth University |
| When? | 1964 |
| Why? | Pleasant and friendly language for non-scientists |
| Ancestry | ALGOL 60, FORTRAN IV |
| Interesting Features | Timeshared computers and microcomputers |
| Notes | Poor structure but easy to learn, evolved into current-day Visual BASIC |



# PL/I

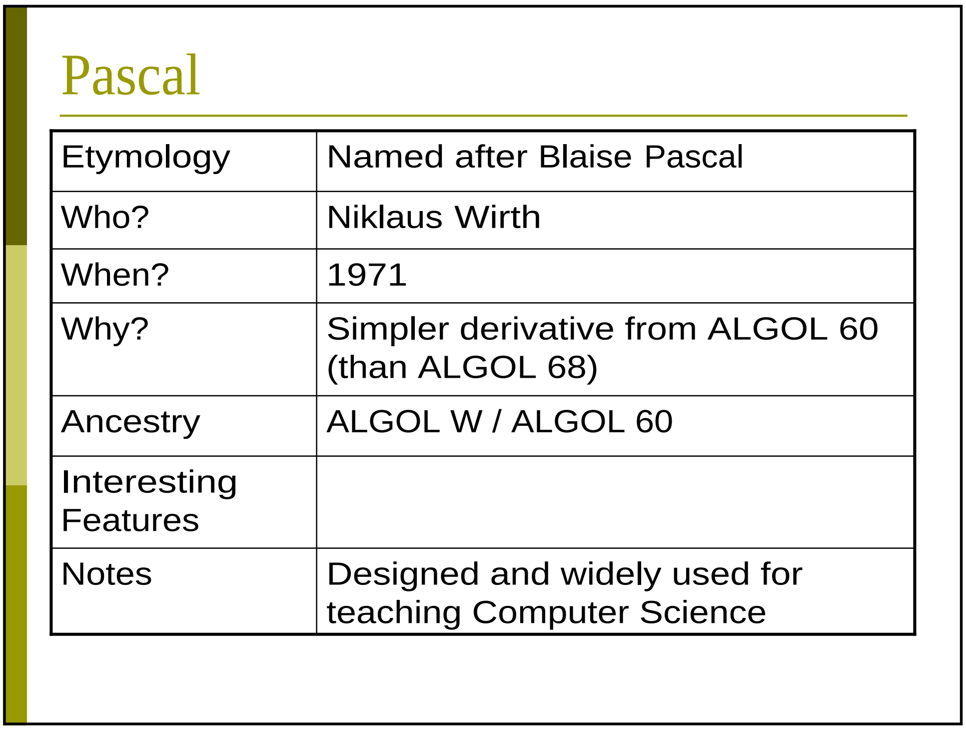
| | |
|---|---|
| Etymology | Programming Language One |
| Who? | IBM |
| When? | 1965 |
| Why? | Combine best parts of FORTRAN, COBOL, ALGOL to create universal language |
| Ancestry | ALGOL 60, FORTRAN IV, COBOL |
| Interesting Features | Concurrency, pointers, matrix slicing |
| Notes | Very complex due to support of large number of (low-level) features |

# SIMULA

| | |
|---|---|
| Etymology | Derived from “simulation” |
| Who? | Nygaard and Dahl |
| When? | 1964 (v1), 1967 (SIMULA 67) |
| Why? | System simulation where routines can restart at previously stopped positions |
| Ancestry | ALGOL 60 |
| Interesting Features | Classes |
| Notes | Forerunner of modern object-oriented languages |

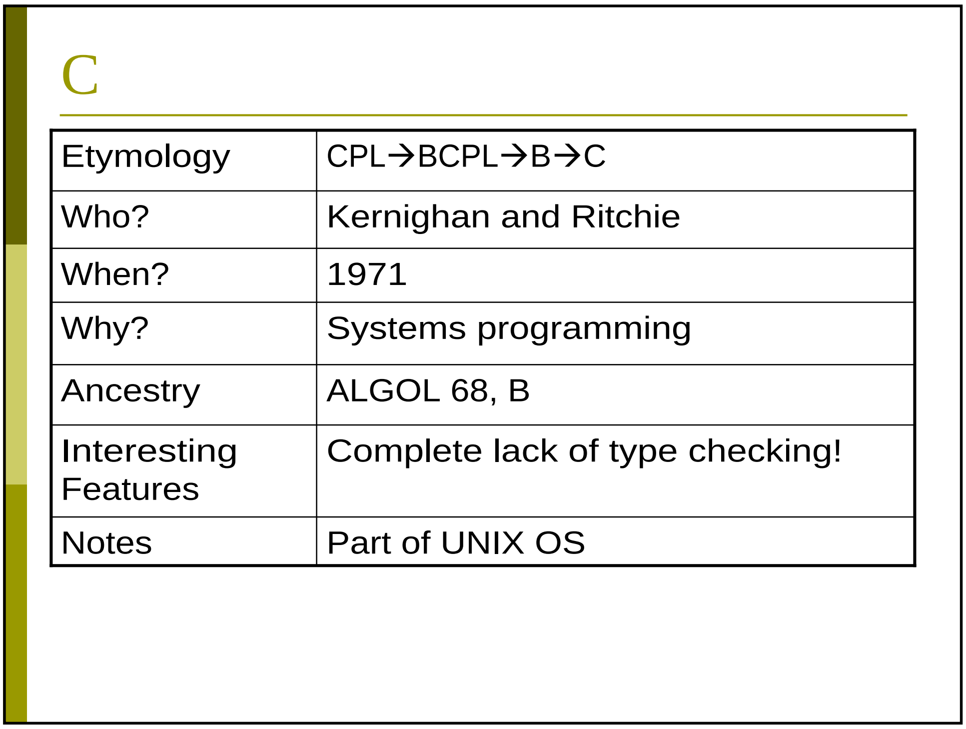
# PASCAL Example

```
program test;

var
   i, j : integer;

function square ( s : integer ) : integer;
var t : integer;
begin
   t := s*s;
   return t;
end;

begin
   Writeln (‘Test program for UCT-CPL’);
   Readln (i);
   j := square (i);
   Writeln (i, ‘ squared is ’, j);
end.
```



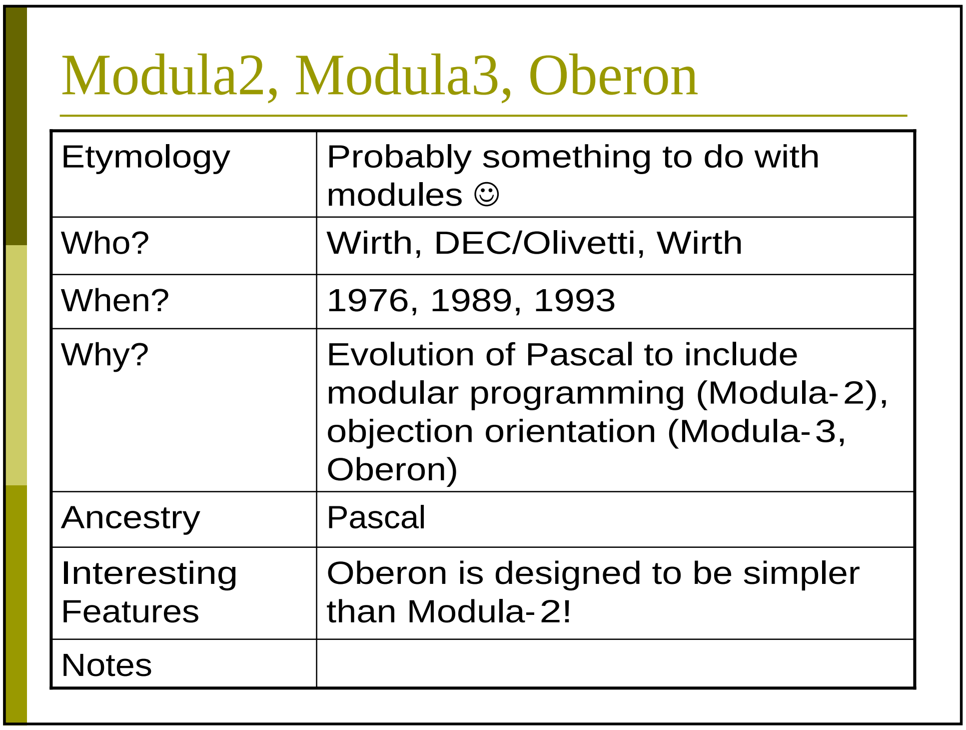
## Pascal

| | |
|---|---|
| Etymology | Named after Blaise Pascal |
| Who? | Niklaus Wirth |
| When? | 1971 |
| Why? | Simpler derivative from ALGOL 60 (than ALGOL 68) |
| Ancestry | ALGOL W / ALGOL 60 |
| Interesting Features | |
| Notes | Designed and widely used for teaching Computer Science |



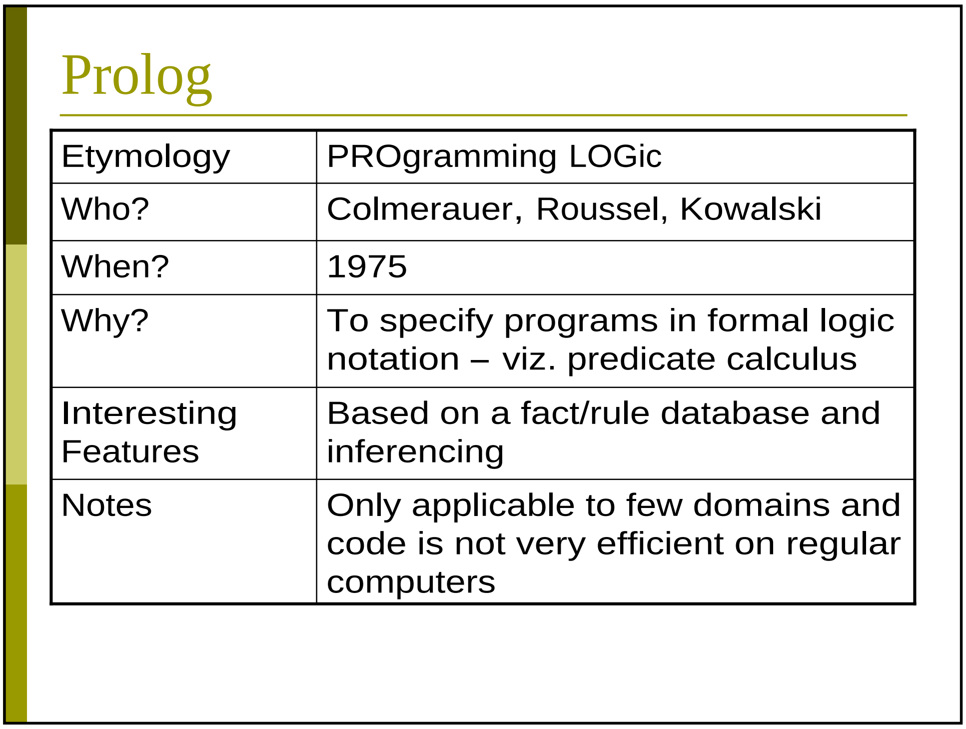
## C

| | |
|---|---|
| Etymology | CPL→BCPL→B→C |
| Who? | Kernighan and Ritchie |
| When? | 1971 |
| Why? | Systems programming |
| Ancestry | ALGOL 68, B |
| Interesting Features | Complete lack of type checking! |
| Notes | Part of UNIX OS |



## Modula2, Modula3, Oberon

| | |
|---|---|
| Etymology | Probably something to do with modules ☺ |
| Who? | Wirth, DEC/Olivetti, Wirth |
| When? | 1976, 1989, 1993 |
| Why? | Evolution of Pascal to include modular programming (Modula-2), objection orientation (Modula-3, Oberon) |
| Ancestry | Pascal |
| Interesting Features | Oberon is designed to be simpler than Modula-2! |
| Notes | |



## Prolog

| | |
|---|---|
| Etymology | PROgramming LOGic |
| Who? | Colmerauer, Roussel, Kowalski |
| When? | 1975 |
| Why? | To specify programs in formal logic notation – viz. predicate calculus |
| Interesting Features | Based on a fact/rule database and inferencing |
| Notes | Only applicable to few domains and code is not very efficient on regular computers |

# Ada Example

```
with Stack_Int;
use  Stack_Int;

procedure Demo_GS is
 -- Demonstrate the use of the Generic_Stack package by using a
 -- Stack of Integers.
 -- from ADA95 Lovelace Tutorial, David A. Wheeler

 Stack1, Stack2 : Stack;
 Dummy : Integer;
begin
 Push(Stack1, 1); -- Put 1 onto Stack1.
 Push(Stack1, 2); -- Put 2 onto the Stack1.
 Stack2 := Stack1; -- Copy stack1's contents into stack2.
 Pop(Stack2, Dummy); -- Dummy is now 2.
 Pop(Stack2, Dummy); -- Dummy is now 1.
 --  Now Stack2 is empty and Stack1 has two items.
end Demo_GS;
```

# Ada

| | |
|---|---|
| Etymology | Named after Augusta Ada Byron |
| Who? | US Department of Defence |
| When? | 1983, 1995 |
| Why? | To standardise a programming language for all the DoD’s operations and embedded systems |
| Ancestry | Pascal |
| Interesting Features | Generic program units, concurrency |
| Notes | |

## Smalltalk

| | |
|---|---|
| Etymology | |
| Who? | Alan Kay |
| When? | 1969, 1972, 1980 |
| Why? | To support highly interactive object-oriented desktop paradigm |
| Ancestry | SIMULA 67 |
| Interesting Features | Objects invoking methods by exchanging messages |
| Notes | Promoted WIMP methodology |

## C++

| | |
|---|---|
| Etymology | Probably the one after "C" in C++ syntax |
| Who? | Ellis and Stroustrup |
| When? | 1986, 1990→ |
| Why? | Make C object-oriented |
| Ancestry | C and SIMULA 67 |
| Interesting Features | Plain C + object-orientation |
| Notes | |

# PERL Example

```
# Get CGI parameters and convert to hash
# for a Web form processing application

$_=$ENV{'QUERY_STRING'};
@parray=split (/[=&]/);
foreach (@parray)
{
   s/\+/ /g;
   s/\n//g;
   s/%[0-9A-F][0-9A-F]/sprintf("%c", hex (substr ($&, 1)))/ge;
   while (substr ($_, 0, 1) eq ' ')
   { $_ = substr ($_, 1); }
   while (substr ($_, length ($_)-1, 1) eq ' ')
   { chop $_; }
   $_ = lc ($_);
}
%qarray=@parray;
```

# PERL

| | |
|---|---|
| Etymology | Practical Extraction and Report Language |
| Who? | Larry Wall (et al) |
| When? | 1987→ |
| Why? | Simple text processing and system-level scripting |
| Interesting Features | Regular expressions |
| Notes | Simple to write but difficult to read/modify |

# Mathematica Example

```
(* Bisection algorithm to find a given *)
(* value in an ordered list *)

Search[x_, values_] :=
  Module[{Mid, Start=1, Stop=Length[values]},
         While[Start+1 != Stop,
               Mid = Floor[(Start+Stop)/2];
               If[values[[Mid]] > x,
                  Stop=Mid,
                  Start=Mid
                 ]
              ];
         Start
        ]
```

# Mathematica

| | |
|---|---|
| Etymology | Related to mathematics |
| Who? | Wolfram Research Institute |
| When? | 1988→ |
| Why? | To support advanced mathematical calculations |
| Interesting Features | Functional and imperative programming |
| Notes | Tree structured programs and data (LISP-like) |

# Python Example

```
# example from www.python.org
# function to invert a table

def invert(table):
    index = {}                # empty
  dictionary
    for key in table.keys():
        value = table[key]
        if not index.has_key(value):
            index[value] = [] # empty list
        index[value].append(key)
    return index
```

# Python

| | |
|---|---|
| Etymology | From “Monty Python’s Flying Circus” |
| Who? | CWI, CNRI |
| When? | 1990→ |
| Why? | Extensible object-oriented scripting language |
| Interesting Features | Indentations for block structure |
| Notes | |

# Java

| Etymology | |
|---|---|
| Who? | SUN Microsystems |
| When? | 1995→ |
| Why? | For reliability in program consumer devices |
| Ancestry | C++ |
| Interesting Features | Embeddable in Web pages, no pointers – only references, single inheritance, garbage collection |
| Notes | No direct compilation – use of intermediate bytecode |

# PHP Example

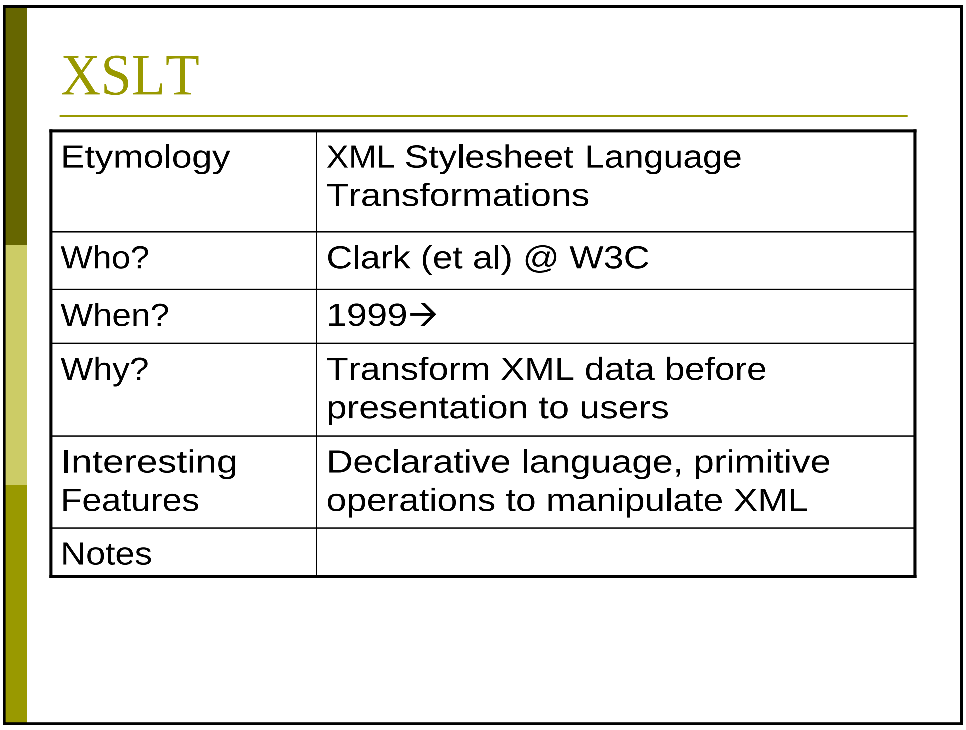
```
<!-- from J. Fulton, PHP Tutorial -->
<h2>Simple Form Example</h2>

<? function show_form($first="",$last="") {  ?>
<form action="simpleForm.php3" method="post">
First Name: <input type=text name=first value="<?echo $first?>"><br>
Last Name: <input type=text name=last value="<?echo $last?>"><br>

<input type=submit>
</form>
<? }

if(!isset($first)) {
        show_form();
}
else {
   if(empty($first) or empty($last)) {
        echo "You did not fill in all the
                fields, try again<p>";
        show_form($first,$last);
   }
   else {
        echo "Thank you, $first $last";
   }
}  ?>
```

# PHP

| | |
|---|---|
| Etymology | PHP Hypertext Processor |
| Who? | Rasmus Lerdorf (et al) |
| When? | 1994→ |
| Why? | Embedded scripting language for Web pages |
| Ancestry | C, Java, Perl |
| Interesting Features | Persistent and shared state between invocations |
| Notes | |

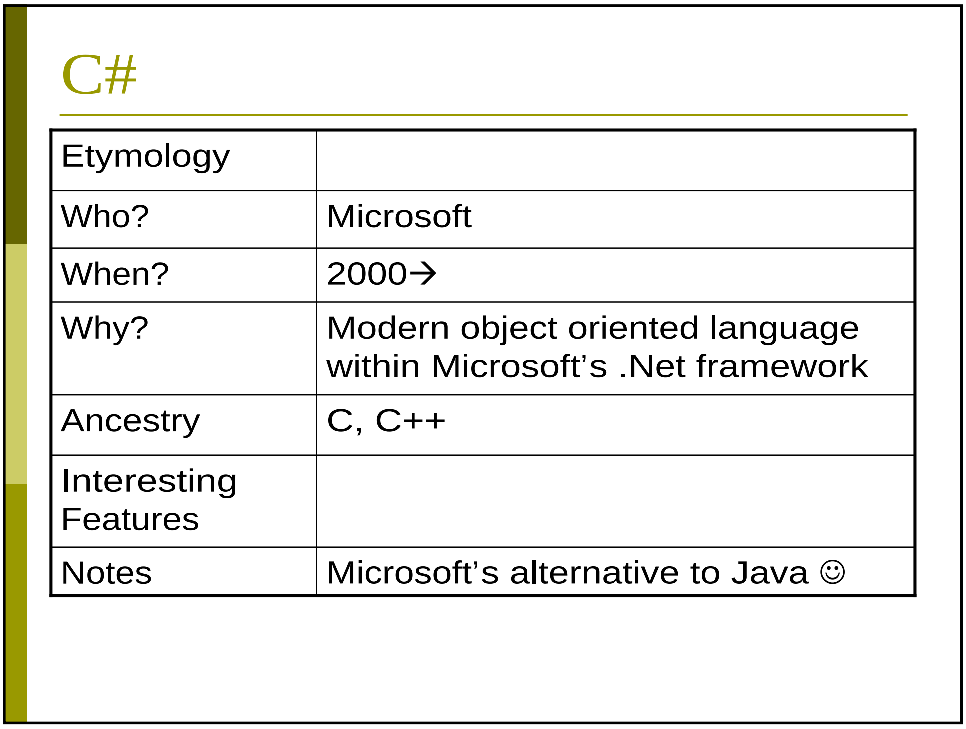
# XSLT Example

```
<stylesheet>
   <output method="xml"/>
   <variable name="institution"><text>UCT</text></variable>

   <template match="uct:uct">
      <oaidc:dc>
         <dc:title><value-of select="uct:title"/></dc:title>
         <apply-templates select="uct:author"/>
         <element name="dc:publisher">
            <value-of select="$institution"/>
         </element>
         <apply-templates select="uct:version"/>
      </oaidc:dc>
   </template>

   <template match="uct:author">
      <dc:creator>
         <value-of select="."/>
      </dc:creator>
   </template>

</stylesheet>
```



## XSLT

| | |
|---|---|
| Etymology | XML Stylesheet Language Transformations |
| Who? | Clark (et al) @ W3C |
| When? | 1999→ |
| Why? | Transform XML data before presentation to users |
| Interesting Features | Declarative language, primitive operations to manipulate XML |
| Notes | |



## C#

| | |
|---|---|
| Etymology | |
| Who? | Microsoft |
| When? | 2000→ |
| Why? | Modern object oriented language within Microsoft’s .Net framework |
| Ancestry | C, C++ |
| Interesting Features | |
| Notes | Microsoft’s alternative to Java ☺ |

# Describing Syntax and Semantics

## Context-free Grammars

- Recognise context-free languages, such as most programming languages.
- Describe structure using rules, terminals and non-terminals.
- Used to automatically generate a parser as part of a compiler for a given language.
- Example:

```
<start> → ‘<html>’ <head> <body> ‘</html>’
<head>  → ‘<head>’ <title> ‘</head>’
<title> → ‘<title>’ <text> ‘</title>’
<body>  → ‘<body>’ <bdata> ‘</body>’
<bdata> → <text> <bdata>
        | <tag> <bdata>
        |
```

# Regular Languages

- Context-free grammars which do not allow recursion are regular grammars.
- Can be recognised by a finite state machine.
- Used in programming languages to parse individual tokens/symbols of the language.
- Regular expressions are used for text processing.
- Example:
    - `[a-zA-Z_][0-9a-zA-Z_]*`
      matches identifiers in Java

# Recursive Descent Parsers

- Subprograms correspond to each non-terminal and match RHSs using the input stream, calling subprograms for other non-terminals encountered
- Example:

```
procedure start ()
begin
    get_next_token;
    head;
    body;
    get_next_token;
end;
```

- Recursive-descent fails on left-recursion!

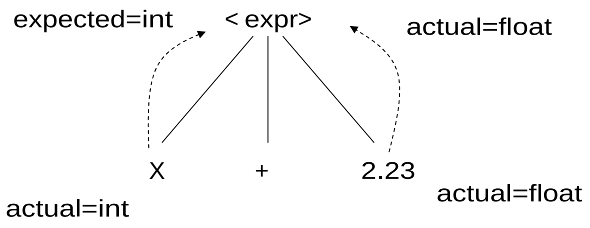
# Attribute Grammars

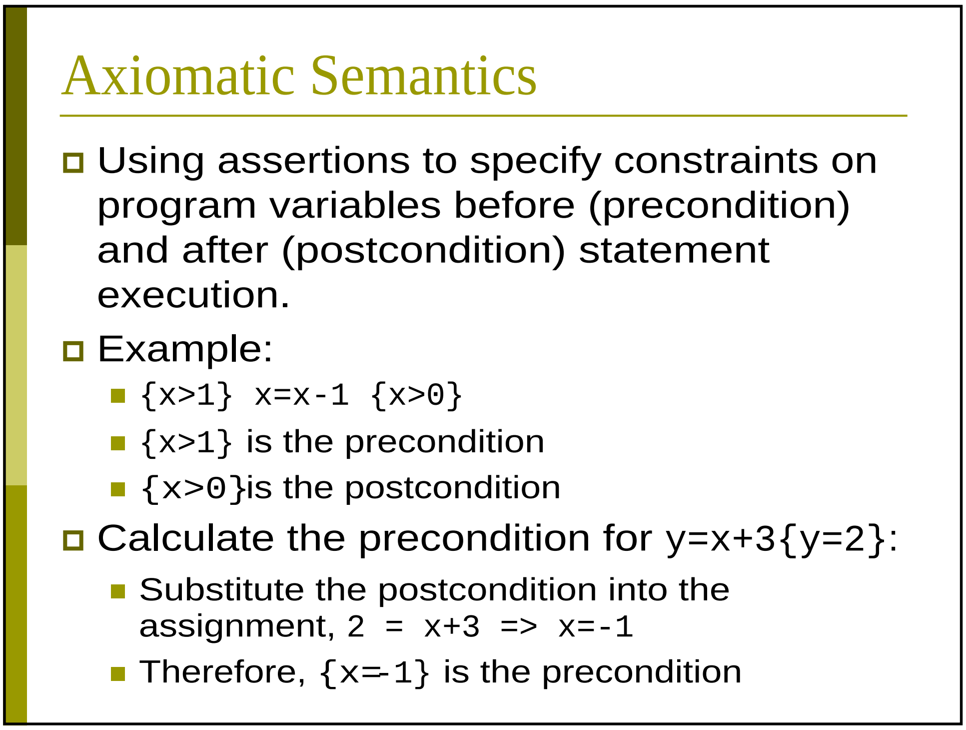
- Extension to context-free grammars encodes static semantics such as type conformance.
- Each grammar symbol has:
  - Synthesised attributes, determined only from the children of a node (e.g., actual types)
  - Inherited attributes, passed up the tree (e.g., expected types)
- Each production has a set of functions to compute attributes for its grammar symbols.
- Predicate functions on non-terminals enforce rules on attributes.

---

# Attribute Grammar Example

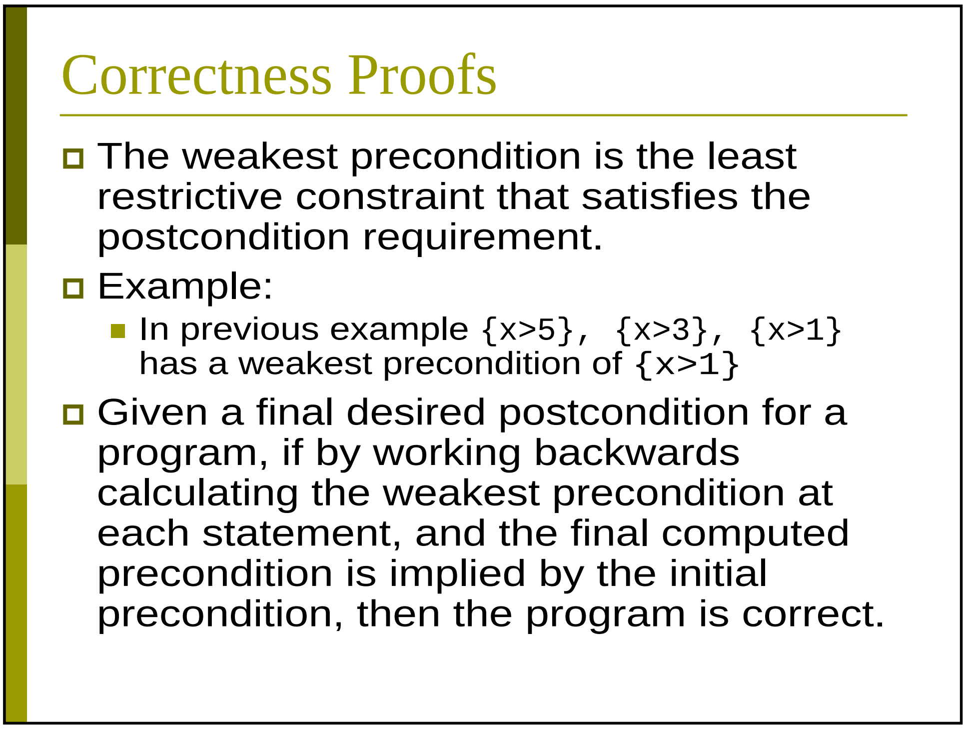
```
<expr> → <var> + <constant>

Syntax: <expr> → <var> + <constant>
Semantic rule: <expr>.expected_type = <var>.actual_type
Semantic rule: <expr>.actual_type =
                 if (<constant>.actual_type = float)
                   then float
                   else <var>.actual_type
Predicate: <expr>.actual_type == <expr>.expected_type
```





## Axiomatic Semantics

- Using assertions to specify constraints on program variables before (precondition) and after (postcondition) statement execution.
- Example:
  - `{x>1} x=x-1 {x>0}`
  - `{x>1}` is the precondition
  - `{x>0}`is the postcondition
- Calculate the precondition for `y=x+3{y=2}`:
  - Substitute the postcondition into the assignment, `2 = x+3 => x=-1`
  - Therefore, `{x=-1}` is the precondition



## Correctness Proofs

- The weakest precondition is the least restrictive constraint that satisfies the postcondition requirement.
- Example:
  - In previous example `{x>5}, {x>3}, {x>1}` has a weakest precondition of `{x>1}`
- Given a final desired postcondition for a program, if by working backwards calculating the weakest precondition at each statement, and the final computed precondition is implied by the initial precondition, then the program is correct.

# Inference Rules

- Inference rules are of the form:

  -
    ```
    S1, S2, ... Sn
    --------------
          S
    ```
- If S1, S2, … Sn are true, then S is true.
- Rule of consequence:

  -
    ```
    {P}S{Q},P’=>P,Q=>Q’
    --------------------
         {P’}S{Q’}
    ```
- 2-statement sequence:

  -
    ```
    {P1}S1{P2},{P2}S2{P3}
    ----------------------
        {P1}S1;S2{P3}
    ```

# Axiomatic Semantics Example

- Prove `{x>5}x=x-2{x>1}`
  - Substituting, we can prove `{x>3}x=x-2{x>1}.`
  - By the rule of consequence,
    ```
    {x>3}x=x-2{x>1}, {x>5}=>{x>3},{x>1}=>{x>1}
    -----------------------------------------
              {x>5}x=x-2{x>1}
    ```
  - Therefore, `{x>5}x=x-2{x>1}.`
- Prove `{x<0}x=-x;y=x+1{y>0}`
  - Substituting, we can compute `{x>-1}` as precondition for `y=x+1`.
  - Assume precondition `{x>-1}` for `y=x+1` is the postcondition for `x=-x`.
  - By the rule of consequence, we prove `{x<0}x=-x{x>-1}` is true.
  - By the 2-statement sequence rule, we prove `{x<0}x=-x;y=x+1{y>0}` is true.

# Denotational Semantics

- The meaning of a program can be specified by defining a set of mathematical objects corresponding to language elements and a set of functions to map the language elements to mathematical objects.
- Once mathematical equivalents are derived, rigorous mathematics can be applied to reason about programs.

---

# Denotational Semantics Example

- Assume the grammar:
  - `<prod> → <prod> ‘*’ <digit>`
  - `        | <digit>`
  - `<digit> → ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’`
  - `         |  ‘6’ | ‘7’ | ‘8’ | ‘9’`
- A denotational mapping to assign meaningful object to each language element could be the following set of functions:
  - M(‘0’)=0, M(‘1’)=1, M(‘2’)=2, …, M(‘9’)=9
  - M(<prod> ‘*’ <digit>)=M(<prod>) * M(<digit>)