# CSc 520

# Principles of Programming Languages

## *1 : Introduction*

Christian Collberg

collberg+520@gmail.com

Department of Computer Science

University of Arizona

# What’s a Programming Language?

# What's a Language???

- A formal language is a notation for precisely communicating ideas.
- By formal we mean that we know exactly which "sentences" belong to the language and that every sentence has a well-defined meaning.
- A language is defined by specifying its syntax and semantics.
- The syntax describes how words can be formed into sentences. The semantics describes what those sentences mean.

# Example Languages

- English is a natural, not a formal language. The sentence

  Many missiles have many warheads.

  has multiple possible meanings.
- Programming languages: FORTRAN, LISP, Java, C++,. . .
- Text processing languages: LaTeX, troff,. . .

```
\begin{slide}{Example Languages}
\begin{itemize}
   \item English is a \highlightbox{natural}, not a formal
         language. The sentence
\end{itemize}
\end{slide}
```

- Specification languages: VDM, Z, OBJ,. . .

# Programming Language Design

# Programming Language Design

- Programming language design has a long history.
- The first modern language (The “Plankalkül”) was designed by Konrad Zuse in the 30s and 40s.
- The Language List (`http://wuarchive.wustl.edu/doc/misc/lang-list.txt` and `http://cui.unige.ch/langlist`) now contains some 2000 entries.

# Programming Language Design...

- Languages are used for a number of applications:
  - Programming (of course),
  - Robot control,
  - Specification (of compilers, safety-critical software systems, etc.),
  - Video game scripting,
  - Database access,
  - Typesetting, etc.

# Programming Language Design. . .

- Programming language design is a lot of fun. Lots of people have felt the urge to design their own language.
- Programming language design is hard. Most language designs are horrible because:
  - Most people don't know enough languages to know what is a good one and a bad one.

# Programming Language Design. . .

- Programming language design is a lot of fun. Lots of people have felt the urge to design their own language.
- Programming language design is hard. Most language designs are horrible because:
  - Most people don’t know enough languages to know what is a good one and a bad one.
  - Most people don’t know about the principles of language design.

# Programming Language Design. . .

- Programming language design is a lot of fun. Lots of people have felt the urge to design their own language.
- Programming language design is hard. Most language designs are horrible because:
  - Most people don't know enough languages to know what is a good one and a bad one.
  - Most people don't know about the principles of language design.
  - Most people don't know enough about compiler design.

# Programming Language Design. . .

- Programming language design is a lot of fun. Lots of people have felt the urge to design their own language.
- Programming language design is hard. Most language designs are horrible because:
  - Most people don't know enough languages to know what is a good one and a bad one.
  - Most people don't know about the principles of language design.
  - Most people don't know enough about compiler design.
  - Most people have no taste.

# Goals of Programming Language Design

These are some of the principles language designers have employed:

1. Simple
2. Expressive
3. Well-defined syntactic/semantic description
4. Reliable/safe
5. Easy to translate
6. Efficient object code
7. Orthogonal
8. All language objects should be first class

# Goals of Programming Language Design…

9. Transparent data types
10. Machine independence and portability
11. Verifiability
12. Consistency with familiar notations
13. Uniformity
14. Extensibility
15. Supports programming-in-the-large
16. Supports information hiding

# Goals of Programming Language Design. . .

- Not all principles can/should be applied everywhere in every language.
- Not all principles will apply to every type of language.
- Some principles may have made sense at some point in time, but don't anymore.

# Compilers and Languages

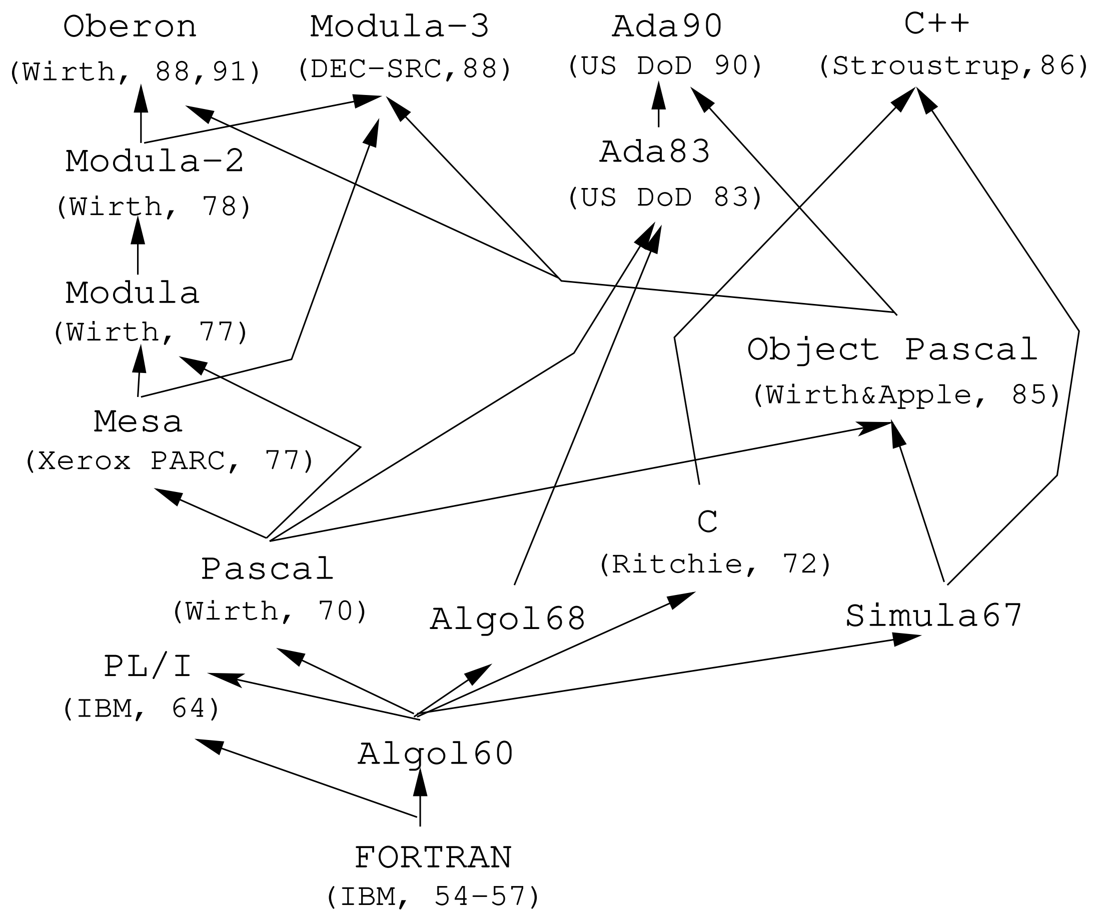
# Compilers and Languages

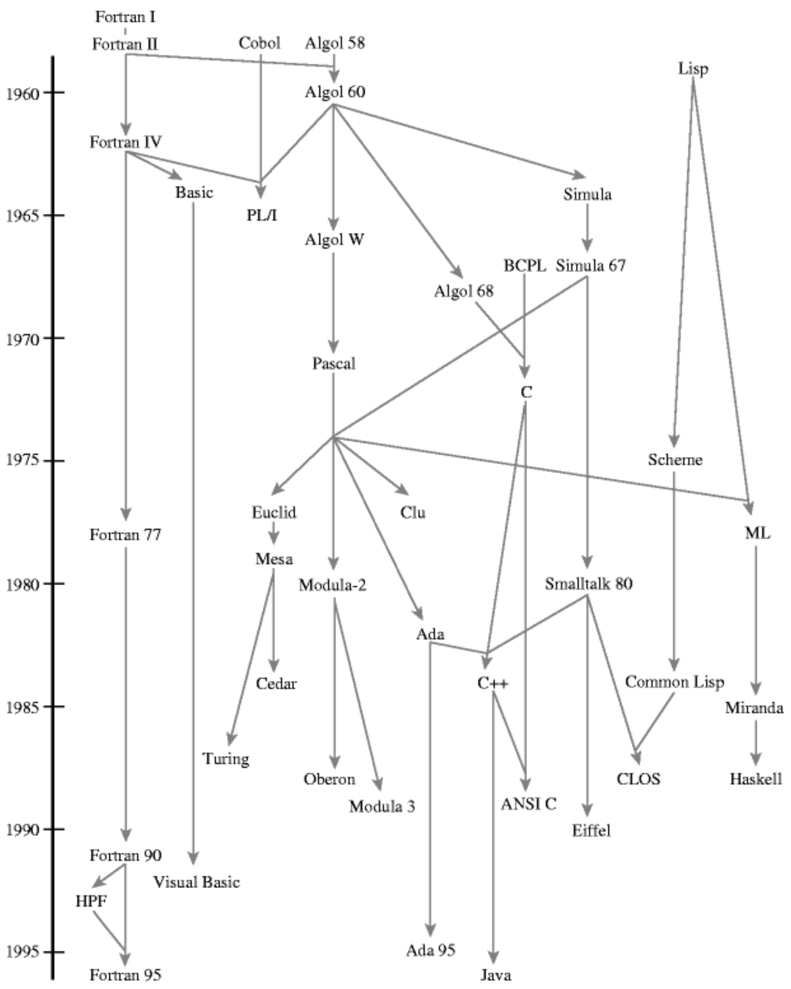
- The history of compiler design and language design go hand in hand:
  - The design of new language features have prompted new compiler technology,
  - New compiler technology has allowed new languages features.
- There is a constant struggle between the programming language user (“Please add this feature!”), the language designer (“How can I incorporate the new feature with the existing ones?”), and the compiler writer (“No more features!”).

# Compilers and Languages...

- Many successful languages have been designed concurrently with a compiler for the language.
- In contrast, many unsuccessful languages have been designed by a committee, without much input from compiler writers.
- It is important for the language designer to be aware of state-of-the-art compiler technology.
- It is important for the compiler designer (particularly, the compiler tool designer) to be aware of the requirements of modern languages.

# History of Procedural Languages






- Algol60 introduced structured programming.
- Simula67 introduced object-oriented programming.
- Mesa introduced modules.
- FORTRAN still rules!

# Goals of Language Design

# Simplicity

- It should be possible to learn the entire language.
- The language should have a small set of basic constructs.
- It should be easy for a user to figure out what it means to combine different language elements.
- A language-rich language is not necessarily a good one:
  1. Every feature has to be implemented by the compiler writer $\Rightarrow$ higher risk of compiler bugs.
  2. Every feature has to be specified in the language design document $\Rightarrow$ higher risk of design flaws and omissions.
  3. Features often interact $\Rightarrow$ it may be impossible to learn only a small part of the language.

# Expressiveness

- The language shouldn't be so simple that it becomes difficult or impossible to write real programs in it.
- Pascal has very simple procedures for IO: There is a read-statement and a write-statement:

```
var x:integer;
read x;
write x+1;
```

- But, there is no way to catch erroneous input! If the user entered hello! when the program expected to read an integer, the program will just fail.

# Well-defined description

- Lexical structure (what identifiers/numbers look like) is easy to define.
- Syntactic structure is easy to define.
- Semantics is hard to define for reasonable size languages. Often done informally or in "semi-formal" English.
- Type equivalence was left out of Pascal definition: some implementations used name equivalence some structural equivalence some declaration equivalence.

# Well-defined description. . .

```
TYPE T1 = RECORD a:CHAR; b:REAL END;
TYPE T2 = RECORD a:CHAR; b:REAL END;
VAR x1 :  T1;
VAR x2 :  T2;
VAR x3,x4:  RECORD a:CHAR; b:REAL END;
BEGIN
   x1 := x2;    (* OK, or not?  *)
   x3 := x4;    (* OK, or not?  *)
END
```

- name-equivalence: both assignments are illegal.
- declaration-equivalence: only 2nd assignment is legal.
- structural type equivalence: both assignments are legal.

# Well-defined description...

```
TYPE Shape = OBJECT
        METHOD draw (); ···
        METHOD move (X,Y:REAL); ···
     END;
TYPE Cowboy = OBJECT
        METHOD draw (); ···
        METHOD move (X,Y:REAL); ···
     END;
VAR s:S; c:C;
BEGIN s := c; (* OK? *) END
```

- In Modula-3 (which uses structural equivalence) `s` and `c` are compatible! In Object-Pascal (which uses name equivalence) they are not.

# Well-defined description. . .

- Some languages have a strict order of evaluation within an expression, others leave it up to the implementation:

```
x := f(a) + b;
```

- If `f` modifies b then the order matters.
- Java has a fixed order of evaluation.
- C leaves order of evaluation up to the implementation.

# Well-defined description...

- FORTRAN requires that parenthesis be honored: $(5.0 * x) * (6.0 * y)$ can't be evaluated as $(30.0 * x * y)$.
- Different orders of evaluation can yield different results.

$$(x * 0.00000001) * 10000000000.0$$

may evaluate differently then

$$(x * 1000.0)$$

# Reliability/Safety

- What happens when you leave out a new-line at the end of a Makefile:

```
x.o:  x.c
   cc -c x.c   # Last line of file; No end of line here!
```

- `make` ignores the rule! (At least some implementations.)
- `make` is probably the worst language design known to man.
- OK, I lie. I forgot about XML. And C++.

# Reliability/Safety…

- In 1990 AT&T's long distance service fails for nine hours due to a wrong `break` statement in a C program.

```
switch (e) {
    0 :
    1 :   S1;
          break;
    2 :   S2;          ⇐ Really meant to fall-through here?!?!
    3 :   S3;
          break;
}
```

- C's design allows several cases to share the same statement (as 0 and 1 do above).

# Reliability/Safety…

- Pascal achieves the same goal without C's safety problem:

**case** $(e)$ **of**
    `1,3 :` $S_1$`;`
    `4..9 :` $S_2$`;`
    `99 :` $S_3$`;`
**end**

# Fast Translation

- Important in the old days when
  - Computers were slow.
  - Languages had no module systems $\Rightarrow$ programs were huge and monolithic.
- Today programs are enormous (several million LOC) but modular. Most important is that modules can be compiled independently; speed of compilation of individual modules is not so important.

# Efficient Object Code

- Important in the old days when computers were slow.
- Sometimes matters today also, but programmer productivity is usually more important:
  - Many programs which were previously written in C for efficiency are now written in Perl for portability and because it requires less programming effort.
- Also depends on what the target application of the language is: FORTRAN is used for huge numerical programs (weather prediction, for example). The generated code must be fast.

# Orthogonality

*Orthogonality means that features can be used in any combination, that the combinations all make sense, and that the meaning of a given feature is consistent, regardless of the other features of the language.*

`[Scott, p. 256]`

- Pascal: functions can return integers, reals, etc. but cannot return arrays or records.
- Modula-2: integers, reals, etc. can be compared (with `<`, `<=`, etc.), but strings cannot.

# Orthogonality...

- Orthogonality can often be a red herring. Completely orthogonal languages (Algol 68, for example) can be so complex that no-one can implement them, or want to use them. Many combinations of features will be uninteresting to the average user.

# Orthogonality: Order of Declaration

- Pascal has a completely fixed order of declaration: Labels, Constants, Types, Variables, and then Procedures. (Pascal is known as a B&D (Bondage-&-Discipline) Language.)
- Other languages are more forgiving, but still require Declaration-Before-Use, i.e. a name must be declared before it is referenced.
- Other languages, still, allow a completely free order of declaration. This allows the programmer to write the declarations in the most natural order, but makes things more difficult for the compiler writer (Surprise!).

# Orthogonality: Order of Declaration. . .

Declaration-Before-Use

```
(* This is illegal:  *)
procedure bar (); begin foo() end;
procedure foo (); begin bar() end;
```

- `foo` is called before it is declared.
- In a strict declaration-before-use language it's impossible to declare mutually recursive procedures, like `foo` & `bar` above.

# Orthogonality: Order of Declaration. . .

Declaration-Before-Use

```
(* This is illegal:  *)
procedure bar (); begin foo() end;
procedure foo (); begin bar() end;
```

- `foo` is called before it is declared.
- In a strict declaration-before-use language it's impossible to declare mutually recursive procedures, like `foo` & `bar` above.
- In many dialects of Pascal we can forward declare `foo`:

```
procedure foo (); forward;
procedure bar (); begin foo() end;
procedure foo (); begin bar() end;
```

# Orthogonality: Order of Declaration. . .

Free Order of Declaration

- The compiler must be able to handle a reference to a name before it is declared:

```
PROCEDURE P (v:T); BEGIN x := 5 END P;
TYPE  T = ARRAY [1..C3] OF CHAR;
CONST C3 = 5;
VAR   x :  INTEGER;
```

# Orthogonality: Order of Declaration...

Free Order of Declaration

- The compiler must be able to handle a reference to a name before it is declared:

```
PROCEDURE P (v:T); BEGIN x := 5 END P;
TYPE  T = ARRAY [1..C3] OF CHAR;
CONST C3 = 5;
VAR   x :  INTEGER;
```

- The compiler must detect illegal recursive declarations:

```
CONST C1 = C2 + 1;
      C2 = C1 + 2;
TYPE  R1 = RECORD x :  R2 END;
      R2 = RECORD y :  R1 END;
```

# Orthogonality: Order of Declaration...

- Modula-2 and some other languages allow free order of declarations for some language elements (procedures) but require declaration-before-use for others (types and constants).
- Thus Modula-2 is completely non-orthogonal with respect to order of declaration!
- This compromise, however, makes life reasonably OK both for the programmer and the compiler-writer.

# First Class Citizenship

*Generally, a value in a programming language is said to have first-class status if it can be passed as a parameter, returned from a sub-routine, or assigned into a variable. [...] A second class value can be passed as a parameter, but not returned from a subroutine or assigned into a variable, and a third-class value can't even be passed as a parameter.* `[Scott, p. 143]`

- Labels are third-class in Pascal but second-class in Algol.
- Pascal functions can take other functions as arguments but cannot return a function as the result.

# First Class: Procedure Nesting...

- Since the early-60's programming language designers have wrestled with the problem of large name spaces. Any large program will contain many names (declared procedures, types, variables, etc). How do we prevent name-clashes?
- We may, for example, want a function `Append` that concatenates strings together, and another function `Append` the concatenates lists.
- In other words, we need to be able to control the visibility of names, i.e. make them visible in some part of the program and hidden in other parts.

# First Class: Procedure Nesting. . .

- Algol introduced nested procedures:

```
procedure P ();
var x :  char;
   procedure Q ();
   var x :  integer;
   begin x := 5; end;
begin x := "X"; Q(); end
```

- In Modula-2 you can pass a function as argument to another function. However, you can't pass a nested function. This makes life easier for the compiler, but hell for the programmer.

# First Class: Procedure Nesting...

```
TYPE F = PROCEDURE();
PROCEDURE R(func:F); BEGIN END;

PROCEDURE S(); BEGIN END;

PROCEDURE P();
   VAR X : INTEGER;
   PROCEDURE Q ();
   BEGIN X := 5; END;
BEGIN
   R(S); (* ⇐ Legal in Modula-2!   *)
   R(Q); (* ⇐ Illegal in Modula-2!   *)
END;
```

# Transparent Data Types

*A data type is transparent when all values of that type can be named and represented as literals within the language.*

- Pascal arrays and records have no literal representation. But, in Java you can say

```
int[] A = {2,3,5,7,9,11,13};
```

- Pascal, however, has literal sets:

```
X := [1, 5..9];
```

# Machine Independence/Portability

- The language specification can be “loose”, giving much leeway to the implementation:
  - How big is an INTEGER, REAL, CHAR?
  - What representation is used for characters (ASCII, Unicode, EBCDIC)?
  - How deeply can procedures be nested?
- Java is very strict, it specifies that
  - Characters are Unicode.
  - `int`, `floats` are 32-bit, double and longs 64-bit.
  - Mathematical functions (in `java.lang.StrictMath`) should be implemented as in `http://metalab.unc.edu`.

# Machine Independence/Portability. . .

- Java . . .
  - IEEE floating-point standards apply to `float` and `double` data types. (This sucks if you're writing a Java compiler for the Cray which has their own floating point format.)
- C++ is less strict, it specifies that
  - `short` and `int` could be the same.
  - `float`, `double`, `long` could be implemented the same.
- Ada and Modula-2 has a special `SYSTEM` module that contains any system-specific definitions.

```
DEFINITION MODULE SYSTEM;
CONST BITSPERLOC    = 8;
TYPE  LOC;         (* Smallest addressable unit of storage *)
      ADDRESS = POINTER TO LOC;

PROCEDURE ADDADR (addr: ADDRESS; offset: CARDINAL): ADDRESS;
  (* The address given by (offset + addr).  *)

PROCEDURE ADR (VAR v: <anytype>): ADDRESS;
  (* The address of variable v *)

PROCEDURE CAST(<targettype>; val: <anytype>): <targettype>;
  (* CAST is a type transfer function.  *)

PROCEDURE TSIZE (<type>; ...): CARDINAL;
  (* Number of LOCS used to store a value of type <type>. *)
END SYSTEM.
```

# Verifiability

- In the 70s there were several attempts at constructing languages where programs could be verified (proved) to be correct.
- This didn't go anywhere. Real programs are way too complex to be amenable to automatic analysis. There has been recent interest, however, in languages that allow you to find bugs automatically.
- In C++ you can say `assert (arg>=0 && arg<=100)`. A violation causes an exception to be thrown at runtime.
- Eiffel takes this one step further with its design by contract. Contracts can either be verified at runtime (expensive) or compile-time (hard).

# Verifiability: Design by Contract

```
class interface DICTIONARY [ELEMENT] feature
   put (x: ELEMENT; key: STRING) is
      --- Insert x so that it will be retrievable through key.
      require
         count <= capacity
         not key.empty
      ensure
         has (x)
         item (key) = x
         count = old count + 1
      invariant
         0 <= count
         count <= capacity
end
```

http://archive.eiffel.com/doc/manuals/technology/contract

# Consistency with Familiar Notations

*Respect common expectations regarding established notation.*

- COBOL: `ADD B TO C GIVING A`.
- APL: Expressions are evaluated right-to-left. There is no operator precedence. (I actually like this.)
- C has 16 levels of precedence in order to appear "natural."
- Notation that may appear natural to some people are not to others. (Ask your English-major friends about the distributive law of arithmentic.)

# Uniformity

*Similar things should have similar meanings.*
*Different things should have different meanings.*

- Ada: `F(x)` can either be a function call or an array reference. This kind-of makes sense (functions and arrays are somewhat similar), but not always:

```
y := F(x);    -- Array reference or function call
F(x) := 5;    -- Must be an array reference; functions
              -- can't be assigned to.
P(F);         -- F must be an array; functions can't
              -- be passed as arguments.
```

# Uniformity...

- Some languages support user-defined overloaded functions and operators to improve uniformity:

```
function Sin (Angles :  in Matrix) return Matrix;
function Sin (Angles :  in Vector) return Vector;
function Sin (Angle :  in Radians) return Real;
function "+" (X, Y : in Matrix) return Matrix;

begin
   X := Sin(Y);          -- Which Sin???
   S := A + B;           -- Which "+"???
```

- Java uses + both for addition (which is commutative) and string concatenation (which isn't). This is bad.

# Support for Programming-In-The-Large

- It was soon discovered that procedure nesting (as in Pascal) did not give enough visibility control.
- Instead, modules were introduced. A module is simply a language construct that collects a number of declarations together, and that controls their visibility. I.e. a module may make some of its names visible to other modules, and may hide others.
- We say that a module exports some of its names (makes them available to other modules), and imports names from other modules.

# Programming-In-The-Large. . .

- In many languages, the module is also the primary unit of separate compilation. We don't want to compile a large program all at once, and we want different programmers to be able to work on the same program simultaneously. We therefore make each module textually separate (each module is in its own file), and design our compiler so that it can compile one file at a time.
- The example in the next slide is from `Modula-2`. Each module has two parts, a definition and an implementation module. Each part is separately compiled.

# Programming-In-The-Large...

```
DEFINITION MODULE M;
    TYPE T = INTEGER;
    PROCEDURE P (x : T);
END M.

IMPLEMENTATION MODULE M;
    VAR X : T; (* Hidden from R. *)
    PROCEDURE P (x : T); BEGIN ··· END P;
END N.

IMPLEMENTATION MODULE R;
    FROM M IMPORT T, P;
    VAR X : T;
BEGIN P(); END R.
```

# Programming-In-The-Large...

- The definition part of the module defines the names that are exported from the module. The implementation part gives the actual definitions of the names, e.g. bodies of exported procedures.
- Obviously, modules and separate compilation complicates the compiler significantly:
  - We have to be able to compile one module at a time.
  - If a module imports the same name from more than one module (e.g. `Append` from the modules `List` and `String`) we have to be able to determine which symbol should actually be used.

# Support for Information Hiding

David Parnas' Principle of Information Hiding:

1. A module's specification must provide to the intended user *all* the information that he will need to use the program, *and nothing more*.
2. The specification must provide to the implementer *all* the information about the intended use that he needs to complete the program, and *no additional information*.

- Modula-2's opaque type is used to build modules that hide all information within a module's *implementation part*.
- Modula-2's opaque types must be pointers, however, so the construct isn't orthogonal.

# Support for Information Hiding...

```
DEFINITION MODULE Stack;
    TYPE T; (* An opaque type.  *)
    PROCEDURE Push (stack:T; element:INTEGER);
END M.

IMPLEMENTATION MODULE Stack;
    TYPE T = POINTER TO RECORD top:INTEGER; store:ARRAY...END;
    PROCEDURE Push (stack:T; element:INTEGER);
    BEGIN ··· END Push;
END N.

IMPLEMENTATION MODULE R;
    VAR S : Stack.T;
BEGIN
    Stack.Push(S,100);      ⇐ Can't access Stack's internals here!
END R.
```

# Extensibility

- We should be able to create new data types that behave much like the built-in ones.
- If you can declare an array of integers why can't you define your own hash table package and declare a hashtable of integer?
- Here we use an Ada generic module `Stack` to create two stacks, a stack of 100 integers, and a stack of 300 booleans:

```
package StackInt is new Stack(100, INTEGER);
package StackBool is new Stack(300, BOOLEAN);
begin
    StackInt.Push(123);
    StackBool.Push(TRUE);
end
```

# Extensibility...

Here's the implementation of the generic stack module:

```
generic
    Size :  POSITIVE;
    type ITEM is private;
package Stack is
    procedure Push (E: in ITEM);
end Stack;

package body Stack is
    type TABLE is array
        (POSITIVE range <>) of ITEM; ···
    procedure Push (E: in ITEM) is ···
end Stack;
```

# Software Engineering by Language Design

# Preventing interference

*How can I prevent other programmers from writing code that will interfere with mine?*

# Preventing interference

*How can I prevent other programmers from writing code that will interfere with mine?*

- Modules, restricted name spaces, information hiding

# Preventing illegal operations

*How can I prevent other programmers from applying the wrong operations to a variable (taking the length of an integer rather than a string)?*

# Preventing illegal operations

*How can I prevent other programmers from applying the wrong operations to a variable (taking the length of an integer rather than a string)?*

- Strong typing

# Preventing Redundancy

*How can I avoid having to write the same code over and over again, when it only differs in a minor way?*

# Preventing Redundancy

*How can I avoid having to write the same code over and over again, when it only differs in a minor way?*

- Generics, polymorphism, classes, higher-order functions, pattern languages

# Providing Flexibility

*How can I provide flexibility at runtime for cases where little information (about types of values, for example) are known?*

# Providing Flexibility

*How can I provide flexibility at runtime for cases where little information (about types of values, for example) are known?*

- Polymorphism, casting, coersion, dynamic loading of modules/classes

# Error Handling

*How can I deal with runtime errors in a efficient and effective way?*

# Error Handling

*How can I deal with runtime errors in a efficient and effective way?*

- Exceptions

# Control Flow

*How can I concisely express the flow of control in my programs?*

# Control Flow

*How can I concisely express the flow of control in my programs?*

- Control structures, iterators, coroutines, threads, exceptions, continuations

# User-defined Types and Operations

*How can I extend the language I'm programming in with my own types and type variants along with associated operations?*

# User-defined Types and Operations

*How can I extend the language I'm programming in with my own types and type variants along with associated operations?*

- Inheritance, classes, variant records, modules

# Memory Management

*How can I manage the storage I need to allocate for variables, so that it is reclaimed when no longer needed?*

# Memory Management

*How can I manage the storage I need to allocate for variables, so that it is reclaimed when no longer needed?*

- Stack allocation, arena allocation, garbage collection.

# Specification

*How can I ensure that other programmers use my code correctly?*

# Specification

*How can I ensure that other programmers use my code correctly?*

- Module specifications, pre-post conditions

# Summary

# References

- Check out the Language List: `http://cui.unige.ch/langlist`
- Or these resources
  - `http://home.nvg.org/~sk/lang/lang.html`
  - `http://extra.newsguy.com/~nedbush/proglang.htm`
  - `http://www.cs.mun.ca/~ulf/pld/pls.html`
  - `http://www-cs.canisius.edu/PL_TUTORIALS`
  - `http://dmoz.org/Computers/Programming/Languages`
- See what the language list writes about your favorite and least favorite language.
- Look up the ZUSE language in the Language List. Sounds good, doesn't it?

# Homework

- Read Chapter 1 Introduction in Scott.

# FORTRAN

# The First Major Language

- FORTRAN I was the first “high-level” programming language. It’s designers also wrote the first real compiler and invented many of the techniques that we use today.
- The FORTRAN manual can be found here:

  http://www.fh-jena.de/~kleine/history.
- The excerpt on the next few slides is taken from

  John Backus, *The history of FORTRAN I, II, and III*, History of Programming Languages, The first ACM SIGPLAN conference on History of programming languages, 1978.

# The First Compiler

```
Before 1954 almost all programming was done in machine
language or assembly language.  Programmers rightly regarded
their work as a complex, creative art that required human
inventiveness to produce an efficient program.  Much of their
effort was devoted to overcoming the difficulties created by
the computers of that era:  the lack of index registers, the
lack of builtin floating point operations, restricted
instruction sets (which might have AND but not OR, for
example), and primitive input- output arrangements.  Given
the nature of computers, the services which "automatic
programming" performed for the programmer were concerned with
overcoming the machine's shortcomings.  Thus the primary
concern of some "automatic programming" systems was to allow
the use of symbolic addresses and decimal numbers...
```

# The First Compiler. . .

Another factor which influenced the development of FORTRAN was the economics of programming in 1954. The cost of programmers associated with a computer center was usually at least as great as the cost of the computer itself. ... In addition, from one quarter to one half of the computer's time was spent in debugging. ...

This economic factor was one of the prime motivations which led me to propose the FORTRAN project ... in late 1953 (the exact date is not known but other facts suggest December 1953 as a likely date). I believe that the economic need ... provided for our constantly expanding needs over the next five years without ever asking us to project or justify those needs in a formal budget.

# The First Compiler...

It is difficult for a programmer of today to comprehend what "automatic program- ming" meant to programmers in 1954. To many it then meant simply providing mnemonic operation codes and symbolic addresses, to others it meant the simple process of obtaining subroutines from a library and inserting the addresses of operands into each subroutine. ...

We went on to raise the question "...can a machine translate a sufficiently rich mathematical language into a sufficiently economical program at a sufficiently low cost to make the whole affair feasible?" ...

# The First Compiler...

In view of the widespread skepticism about the possibility of producing efficient programs with an automatic programming system and the fact that inefficiencies could no longer be hidden, we were convinced that the kind of system we had in mind would be widely used only if we could demonstrate that it would produce programs almost as efficient as hand coded ones and do so on virtually every job.

As far as we were aware, we simply made up the language as we went along. We did not regard language design as a difficult problem, merely a simple prelude to the real problem: designing a compiler which could produce efficient programs. Of course one of our goals was to design a language which would make it possible for engineers and scientists to write programs themselves for the 704. ... Very early in our work we had in mind the notions of assignment statements, subscripted variables, and the DO statement....

# The First Compiler...

The language described in the "Preliminary Report" had variables of one or two characters in length, function names of three or more characters, recursively defined "expressions", subscripted variables with up to three subscripts, "arithmetic formulas" (which turn out to be assignment statements), and "DO-formulas".

One much-criticized design choice in FORTRAN concerns the use of spaces: blanks were ignored, even blanks in the middle of an identifier. There was a common problem with key-punchers not recognizing or properly counting blanks in handwritten data, and this caused many errors. We also regarded ignoring blanks as a device to enable programmers to arrange their programs in a more readable form without altering their meaning or introducing complex rules for formatting statements.

# The First Compiler. . .

```
Section I was to read the entire source program, compile what
instructions it could, and file all the rest of the
information from the source program in appropriate tables.
...
Using the information that was filed in section I, section 2
faced a completely new kind of problem; it was required to
analyze the entire structure of the program in order to
generate optimal code from DO statements and references to
subscripted variables.  ...
```

# The First Compiler...

section 4, ... analyze the flow of a program produced by sections I and 2, divide it into "basic blocks" (which contained no branching), do a Monte Carlo (statistical) analysis of the expected frequency of execution of basic blocks--by simulating the behavior of the program and keeping counts of the use of each block--using information from DO statements and FREQUENCY statements, and collect information about index register usage ... Section 5 would then do the actual transformation of the program from one having an unlimited number of index registers to one having only three.

The final section of the compiler, section 6, assembled the final program into a relocatable binary program...

# The First Compiler...

Unfortunately we were hopelessly optimistic in 1954 about the problems of debugging FORTRAN programs (thus we find on page 2 of the Report: "Since FORTRAN should virtually eliminate coding and debugging...")

Because of our 1954 view that success in producing efficient programs was more important than the design of the FORTRAN language, I consider the history of the compiler construction and the work of its inventors an integral part of the history of the FORTRAN language; ...