UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS 164 **P. N. Hilfinger**
Spring 2005

CS 164: Programming Languages and Compilers
Class Notes #1: Introduction

1 Purposes of this Course

Not every programmer really has to know how to build a compiler. Although this course has the structure of compilers as its ostensible topic, my real agenda is broader.

- Acquire tools for building textual interfaces or other programs that process text.
- Understand the common structure of programming languages, so as better to learn them and better design programs with language-like capabilities.
- Acquire better intuition about program performance.
- Acquire practice in reading, designing, and writing complex programs.

2 An Extremely Abbreviated History of Programming Languages

- Initially, programs were either built into machines, or were entered by various electromechanical means, such as punched cards or tape (e.g., the Analytical Engine or the Jacquard loom), or wires and switches (e.g., the Eniac).
- Around 1944 came the idea of encoding programs as numbers (machine language) stored as data in the machine, whence came the Machester Mark I and the EDSAC.
- To make machine language easier to write, read, and maintain, assembly languages were introduced in the early 1950's: symbolic names for instructions and data locations, but still machine language—far from normal notation.
- FORTRAN: mid-1950's. Stands for FORmula TRANslator. Allowed use of usual algebraic notation in expressions; control structures (jumps, etc.) still close to machine language.

- LISP: late 1950's—dynamic, symbolic data structures.
- Algol 60: Europe's answer to FORTRAN. Many syntactic features of modern languages come from Algol 60. Also, it introduced the use of BNF (Backus-Naur Form), inspired by Chomsky's work, for describing syntax.
- COBOL (late 1950s): introduces business-oriented data structures, esp. records (structs for you C-folk).
- 1960s saw proliferation of increasingly elaborate languages: e.g., APL (array manipulation), SNOBOL (string manipulation), FORMAC (formula manipulation).
- 1967–68: Simula 67, an Algol 60 derivate intended for writing discrete-event simulations, introduces concept of inheritance, making it the first "object-oriented" language in the modern sense.
- 1968: Algol 68 attempted to synthesize both the numerical, FORTRANish line, the record-oriented line (such as COBOL), with dynamic (pointer-based) data structures. It also tried to extend BNF into describing the entire language. This last effort made it incomprehensible to many, and it faded away. Nevertheless, many C/C++ features may be found in Algol 68. PL/1 was IBM's clunkier but more commercially successful attempt to meet the same goals.
- 1968: announcement of the "Software Crisis." Trend toward simple languages: Pascal, Algol W, C (later).
- 1970s: emphasis on "methodology"—well-structured, modular programs. Several experimental languages (like CLU). Smalltalk introduced, inspired in part by Simula 67.
- Early 1970s: the Prolog language—declarative logic programming language. Originally intended for natural language processing. Later, it is pushed as a general-purpose high-level programming tool.
- Mid 1970s: ML (MetaLanguage) designed to implement LCF (Logic of Computable Functions). This is a functional language with some interesting ideas—type inference and pattern-directed function definition. Has evolved since then and spawned various progeny (e.g., Haskell).
- Mid-1970s: Dept. of Defense discovers it is supporting over 500 computer languages. Starts Ada project to consolidate (and then object-oriented Ada 95).
- Increasing complexity in ideas of object-orientation until ca. 1980 with advent of C++, which has continued to complexify.
- Reprising the move to simplicity in late 1970s, introduction of Java in early 1990s.

3 Problems to be Addressed

- How do we describe a programming language?
  - Users must be able to understand it unambiguously.
  - Implementors must be able understand it completely.
- Given a description, how to implement it?
  - How do we know we have it right?
    - Testing
    - Automating the conversion of description to implementation.
  - How do we save work?
    - Problem: multiple languages translated to multiple targets.
    - Automation (as above)
    - Designing so we can re-use pieces
    - Interpretation
- How do we make the end result usable?
  - Reasonable handling of errors.
  - Detection of questionable constructs.
  - Compilation speed.
    - A standard approach: design language and translator to allow compiling programs in sections.
    - Or, design translators to be really fast.
  - Execution speed. Problem: can make program run faster, if we are willing to have slower compilation. How do we make this trade-off?
  - Binding time. Problem: how to handle program whose parts change constantly, or even are unavailable at translation time?

4 Kinds of Translators

The purpose of translation is ultimately to *execute* a program. There is a spectrum of approaches.

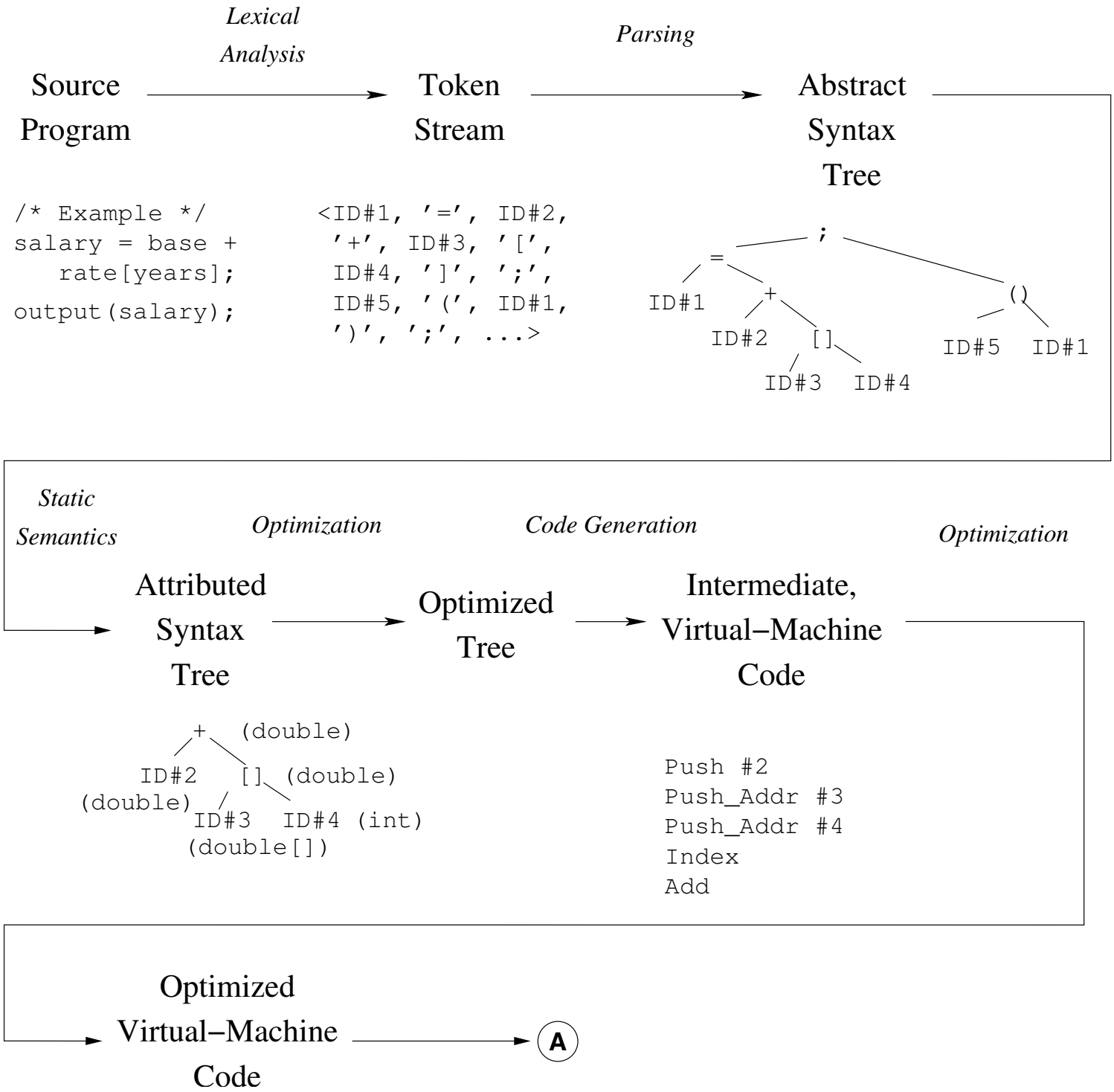
Compilation:	source $\overset{translate}{\longrightarrow}$ real machine language $\overset{execute}{\longrightarrow}$ actions/results
Interpretation:	source $\overset{translate}{\longrightarrow}$ virtual machine language $\overset{interpret}{\longrightarrow}$ actions/results
Direct Execution:	source $\overset{interpret}{\longrightarrow}$ actions/results

Most C/C++ systems are examples of compilation. Lisp interpretation is (in effect) an example of direct execution (the "translation" performed by the reader is trivial). Early Java

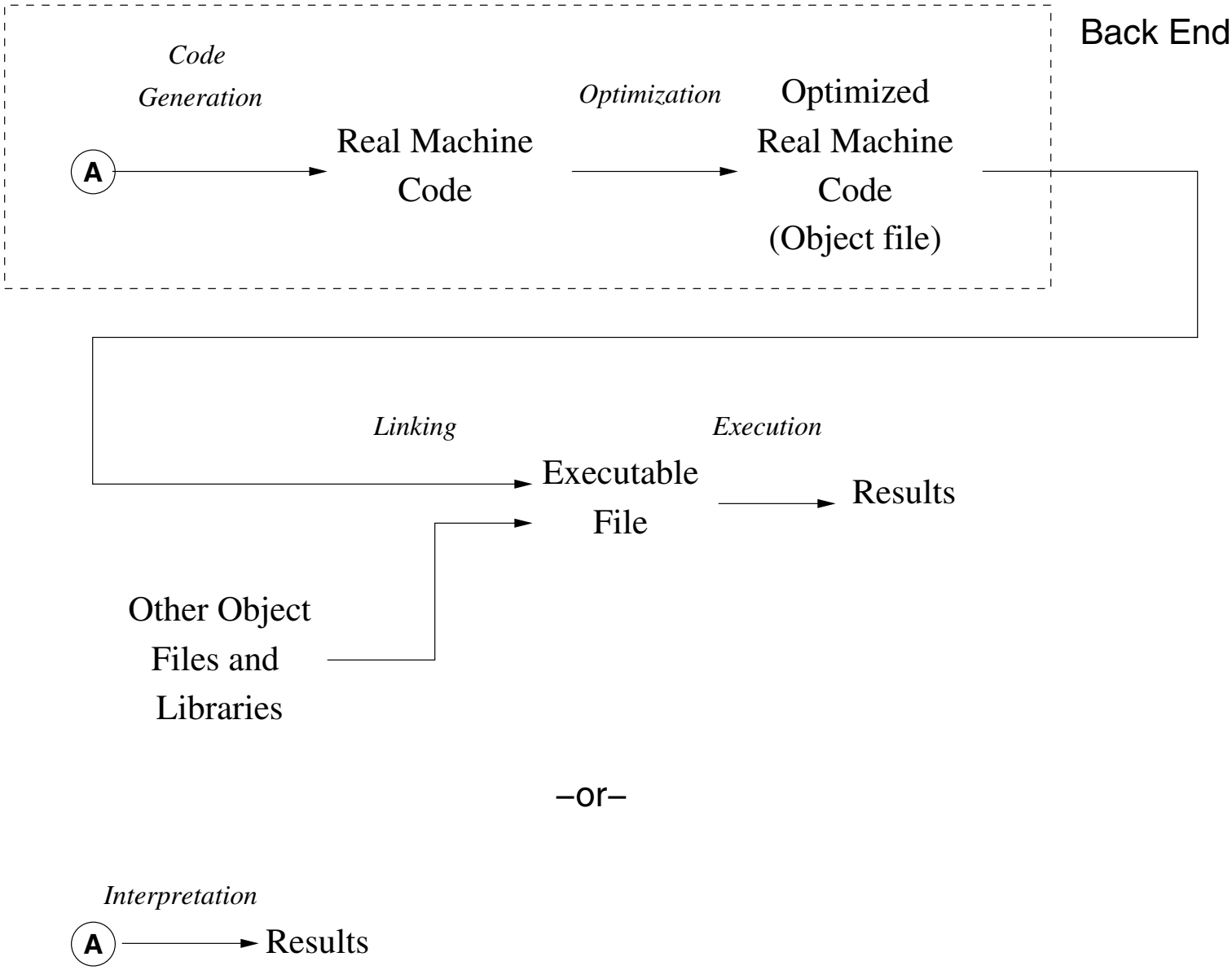
systems used interpretation; now they use just-in-time compilation. These boundaries are muddy, however. Some machines, for example, implement their instruction set by interpretation. Their "real" program is a *microprogram* that acts as an interpreter. For them, a C compiler is an example of an interpreter!

All three of these strategies are specific to an *implementation* of a language; they are *not* inherent in the language. *There are no such things as "interpreted languages" or "compiled languages."* For example, there are C interpreters, and there are Lisp compilers.

Structure of Idealized Compiler : Front End



Structure of Idealized Compiler



5 Programming Languages You May Not Know

```
C FORTRAN (OLD-STYLE) SORTING ROUTINE
C
      SUBROUTINE SORT (A, N)
      DIMENSION A(N)
      IF (N - 1) 40, 40, 10
10    DO 30 I = 2, N
         L = I-1
         X = A(I)
         DO 20 J = 1, L
            K = I - J
            IF (X - A(K)) 60, 50, 50
C
C FOUND INSERTION POINT: X >= A(K)
C
50             A(K+1) = X
               GO TO 30
C
C ELSE, MOVE ELEMENT UP
C
60             A(K+1) = A(K)
20       CONTINUE
         A(1) = X
30    CONTINUE
40    RETURN
      END

C ---------------------------------

C MAIN PROGRAM
      DIMENSION Q(500)
100   FORMAT(I5/(6F10.5))
200   FORMAT(6F12.5)

      READ(5, 100) N, (Q(J), J = 1, N)
      CALL SORT(Q, N)
      WRITE(6, 200) (Q(J), J = 1, N)
      STOP
      END
```

```
comment An Algol 60 sorting program;
procedure Sort (A, N)
   value N;
   integer N; real array A;
begin
   real X;
   integer i, j;
   for i := 2 until N do begin
      X := A[i];
      for j := i-1 step -1 until 1 do
         if X >= A[j] then begin
            A[j+1] := X; goto Found
         end else
             A[j+1] := A[j];
      A[1] := X;
   Found:
      end
   end
end Sort
```

```
⍝ An APL sorting program
∇ Z ← SORT A
  Z ← A[⍋A]
∇
```

```
/* A naive Prolog sort */

/* permutation(A,B) iff list B is a
   permutation of list A. */
permutation(L, [H | T]) :-
   append(V,[H|U],L),
   append(V,U,W),
   permutation(W,T).
permutation([], []).

/* ordered(A) iff A is in ascending order. */
ordered([]).
ordered([X]).
ordered([X,Y|R]) :- X <= Y, ordered([Y|R]).

/* sorted(A,B) iff B is a sort of A. */
sorted(A,B) :- permutation(A,B), ordered(B).
```