Lecture 1: Course Introduction

CS164: Programming Languages and Compilers
P. N. Hilfinger, 787 Soda
Spring 2015

Acknowledgement. Portions taken from CS164 notes by G. Necula.

Administrivia

- All course information, readings, and documentation is online from the course web page (constantly under construction).

- Get a class account electronically from the website with your CalId (under Account Administration), and register it there.
- If you don't have a CalId, send me mail.
- Projects *require* partnerships of 2–4. Start looking now.
- Please read Chapter 2 of the online Course Notes for the next few lectures.

Course Structure

- Lectures, discussions intended to discuss and illustrate material that you have *previously read.*
- Regular homework does theory, practical "finger exercises." Done individually.
- Projects are long programming assignments, done in teams.
- All submissions electronic.
- Target language for projects: Python (version 2.5, not latest). Implementation language: C++. You'll find on-line materials on web site.

Generic General Advice

DBC!

RTFM!

Plagiarism: Obligatory Warning

- We have software to detect plagiarism, and we Know How to Use It!
- If you must use others' work (in moderation), *cite it!*
- Remember that on projects, you necessarily involve your partner.
- Most cheating cases result from time pressure. Keep up, and *talk to us* as early as possible about problems.

Project

- Hidden agenda: programming design and experience.
- Substantial project in modules.
- Provides example of how complicated problem might be approached.
- Validation (testing) part of project.
- Chance to use version control for real.
- And this semester (shudder) C++.
- General rule: start early!

Implementing Programming Languages

- Strategy 1: Interpreter: program that runs programs.
- Strategy 2: Compiler: program that translates program into machine code (interpreted by machine).
- Modern trend is hybrid:
  - Compilers that produce virtual machine code for *bytecode interpreters*.
  - "Just-In-Time" (JIT) compilers interpret parts of program, compile other parts during execution.

Languages

- Initially, programs “hard-wired” or entered electro-mechanically
  - Analytical Engine, Jacquard Loom, ENIAC, punched-card-handling machines
- Next, stored-program machines: programs encoded as numbers (machine language) and stored as data:
  - Manchester Mark I, EDSAC.
- 1953: IBM develops the 701; all programming done in assembly
- Problem: Software costs $>$ hardware costs!
- John Backus: “Speedcoding” made a 701 appear to have floating point and index registers. Interpreter ran 10–20 times slower than native code.

FORTRAN

- Also due to John Backus (1954–7).
- Revolutionary idea at the time: convert high-level (algebraic formulae) to assembly.
- Called "automatic programming" at the time. Some thought it impossible.
- Wildly successful: language could cut development times from weeks to hours; produced machine code almost as good as hand-written.
- Start of extensive theoretical work (and Fortran is still with us!).

After FORTRAN

- Lisp, late 1950s: dynamic, symbolic data structures.
- Algol 60: Europe's answer to FORTRAN: modern syntax, block structure, explicit declaration.
  - Dijkstra: "A marked improvement on its successors."
  - Algol report Set standard for language description.
- COBOL: late 1950's (and still with us). Business-oriented. Introduces records (structs).

The Language Explosion

- APL (arrays), SNOBOL (strings), FORMAC (formulae), and many more.
- 1967-68: Simula 67, first "object-oriented" language.
- Algol 68: Combines FORTRANish numerical constructs, COBOLish records, pointers, all described in rigorous formalism. Remnants remain in C, but Algol68 deemed too complex.
- 1968: "Software Crisis" announced. Trend towards simpler languages: Algol W, Pascal, C

The 1970s

- Emphasis on "methodology": modular programming, CLU, Modula family.
- Mid 1970's: Prolog. Declarative logic programming.
- Mid 1970's: ML (Metalanguage) type inference, pattern-driven programming. (Led to Haskell, OCaml).
- Late 1970's: DoD starts to develop Ada to consolidate >500 languages.

Into the Present

- Complexity increases with C++.
- Then decreases with Java.
- Then increases again (C#, Java 1.5).
- Proliferation of little or specialized languages and scripting languages: HTML, PHP, Perl, Python, Ruby, ...

Example: FORTRAN

```
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 FOUND INSERTION POINT: X >= A(K)
50             A(K+1) = X
               GO TO 30
C ELSE, MOVE ELEMENT UP
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
```

Example: Algol 60

```
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
```

Example: APL

⍝ *An APL sorting program*

∇ $Z \leftarrow SORT\ A$

$Z \leftarrow A\,[⍋A]$

∇

Example: Python (2.5)

```
import sys, re

def format(x):
    return "%10.5f" % x

vals = map(float, re.split(r'\s+', sys.stdin.read().strip()))
vals.sort()
print '\n'.join([ ''.join(map(format, vals[i:i+6]))
                  for i in xrange(0,len(vals),6)])
```

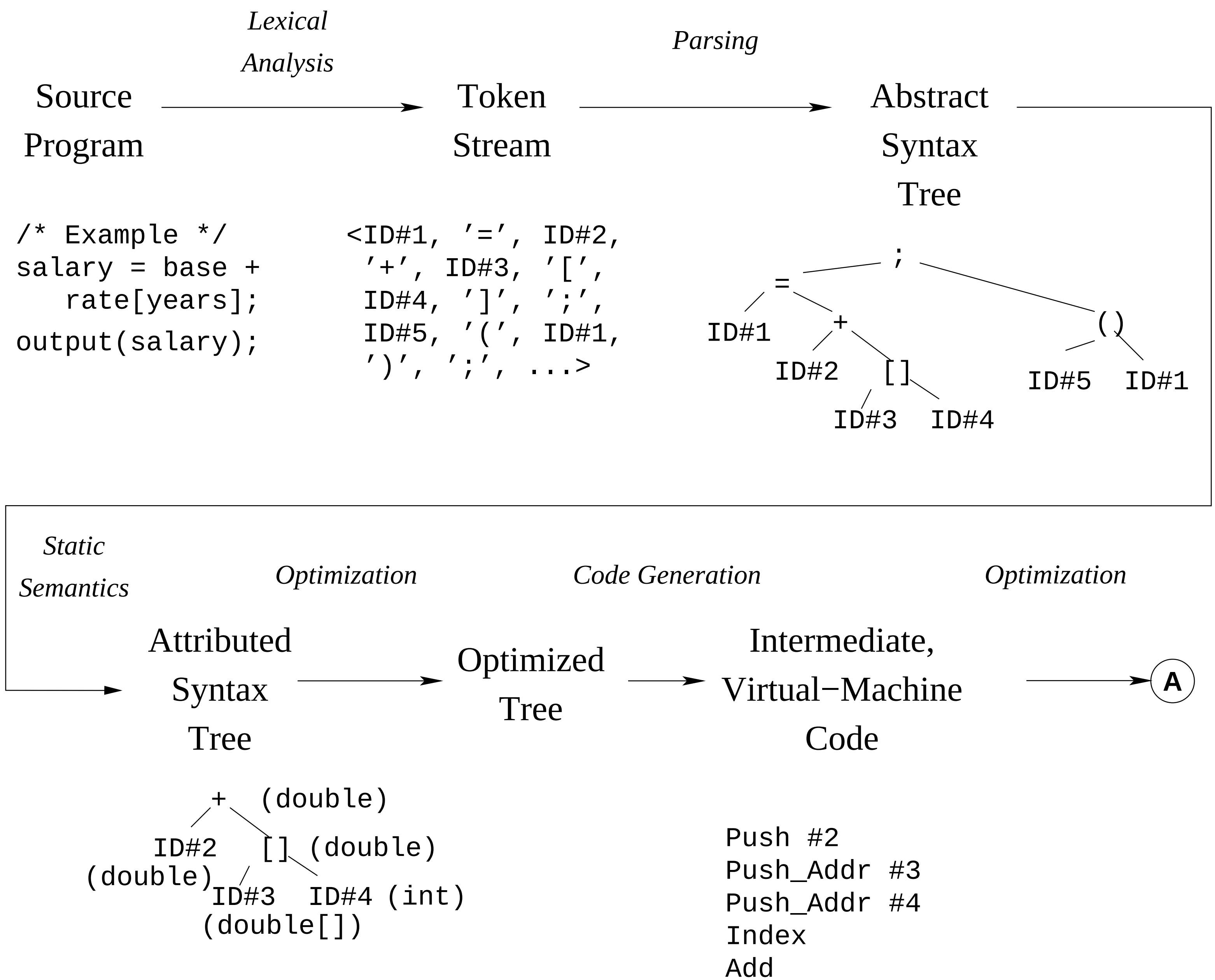
Example: Prolog

```
/* 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).
```

Problems to Address

- How to describe language clearly for programmers, precisely for implementors?
- How to implement description, and know you're right? Ans: Automatic conversion of description to implementation
- How to test?
- How to save implementation effort?
  - With multiple languages to multiple targets: can we re-use effort?
- How to make languages usable?
  - Handle errors reasonably
  - Detect questionable constructs
  - Compile quickly

Classical Compiler Structure (Front)



Classical Compiler Structure (Back)