Introduction to Programming Languages and Compilers

CS164
3:30-5:00 TT
10 Evans

Announcement

- all tomorrow's discussion sections will be held in 330 Soda (Windows PC lab)
- discussion section agenda:
  - Eclipse tutorial
  - how to install the PA1 starter kit
  - using CVS
  - remote testing

Overview

- trends in programming languages
- and why they matter to you
- the structure of the compiler
- the project
- course logistics
- why you want to take this course

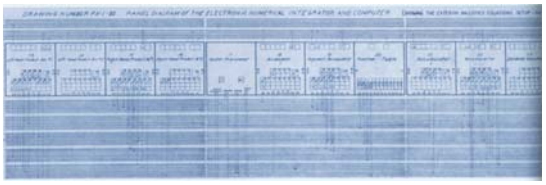
trends in programming languages

Trends in programming languages

- programming language and its compiler:
  - programmer's key tools
- languages undergo constant change
  - from C to C++ to Java in just 12 years
  - be prepared to program in new ones
- design simple languages yourselves
  - an example in this lecture
- to see the trend
  - let's examine the history…

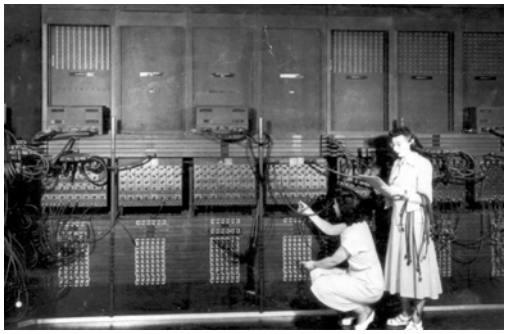
ENIAC (1946, University of Philadelphia)

ENIAC program for external ballistic equations:



(how different is ENIAC in spirit from gaming video cards?)

Programming ENIAC



ENIAC (1946, University of Philadelphia)

- programming done by
  - rewiring the interconnections
  - to set up desired formulas, etc
- Problem:
  - slow, error-prone,
  - this is how program was loaded
- Lesson:
  - store the program in memory! (the von Neuman paradigm)

UDSAC (1947, Cambridge University)

- the first real computer
  - large-scale, fully functional, **stored-program** electronic digital computer (by Maurice Wilkes)
- problem: Wilkes realized:
  - "a good part of the remainder of my life was going to be spent in finding errors in ... programs"
- solution: so he invented procedures (1951)
  - reusable software was born
  - procedure: the first (implemented) language construct

Assembly – the language (UNIVAC 1, 1950)

- Idea translate mnemonic code (assembly) by hand
  - write programs with mnemonic codes (add, sub), with symbolic labels,
  - then assign addresses by hand
- Example:

  clear-and-add a
  add b
  store c
- translate it to something like

  B100 A200
  C300

Assembler – the compiler (Manchester, 1952)

- it was assembler nearly as we know it, called AutoCode
- a loop example, in MIPS, a modern-day assembly code:

```
loop: addi $t3, $t0, -8
      addi $t4, $t0, -4
      lw $t1, theArray($t3)       # Gets the last
      lw $t2, theArray($t4)       #  two elements
      add $t5, $t1, $t2           # Adds them together...
      sw $t5, theArray($t0)       # ...and stores the result
      addi $t0, $t0, 4            # Moves to next "element"
                                  # of theArray
      blt $t0, 160, loop          # If not past the end of
                                  # theArray, repeat
      jr $ra
```


Assembly programming caught on, but

- Problem: Software costs exceeded hardware costs!
- John Backus: "Speedcoding"
  - An interpreter for a high-level language
  - Ran 10-20 times slower than hand-written assembly
    - way too slow

FORTRAN I (1954-57)

- The first compiler
  - Produced code almost as good as hand-written
  - Huge impact on computer science (laid foundations for cs164)
  - Modern compilers preserve its outlines
  - FORTRAN (the language) still in use today
- By 1958, >50% of all software is in FORTRAN
- Cut development time dramatically
  - 2 wks → 2 hrs
  - that's more than 100-fold

FORTRAN I (IBM, John Backus, 1954)

- Example: nested loops in FORTRAN
  - a big improvement over assembler,
  - but annoying artifacts of assembly remain:
    - labels and rather explicit jumps (CONTINUE)
    - lexical columns: the statement must start in column 7
  - The MIPS loop in FORTRAN:

```
      DO 10 I = 2, 40
      A[I] = A[I-1] + A[I-2]
10    CONTINUE
```


Designing a good language is hard

- A good language protects against bugs, but lessons take a while.
- An example that cause a failure of a NASA planetary probe:

  buggy line:
  DO 15 I = 1.100
  what was intended (a dot had replaced the comma):
  DO 15 I = 1,100
  because Fortran ignores spaces, compiler read this as:
  DO15I = 1.100
  which is an assignment into a variable DO15I, not a loop.

- This mistake is harder (if at all possible) to make with the modern lexical rules (white space not ignored) and loop syntax

```
for (i=1; i < 100; i++) { … }
```


Object-oriented programming (1970s)

- inheritance faked using procedural programming:

```
draw(2DElement p) {
  switch (p.type) {
    SQUARE: … // draw a square
            break;
    CIRCLE:  … // draw a circle
            break;
  }
}
```

- **Problem:**
  - unrelated code (drawing of SQUARE and CIRCLE) mixed in the same procedure;
  - hard to reuse code common to both

Object-oriented programming

- In Java, the same code has the desired separation:

```
class Circle extends 2DElement {
  void draw() { <draw circle> }
}
// similar for Square
```

- the dispatch is now much simpler:
  - p.draw()

Review of historic development

- wired interconnects
- von Neuman machines & machine code
- procedures
- assembly (compile by hand)
- assembler
- FORTRAN I
- "cleaner" loops
- object-oriented programming in C
- virtual calls

Do you see a trend?

… and why the trends matter to you

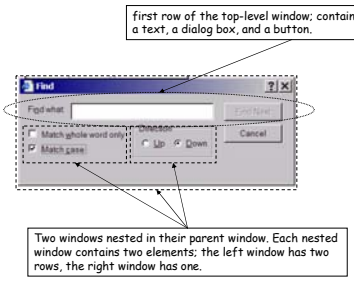
Where will languages go from here?

- As you just saw, the trend is towards higher-level abstractions
  - express the algorithm concisely!
  - which means hiding often repeated code fragments
  - new language constructs hide more of these low-level details.
- Or at least try to detect more bugs when the program is compiled
  - stricter type checking
  - we'll leave this for later

A simple GUI-building language

- You'll be able to design a simple language yourself
  - to simplify a repetitive programming task
  - next example is from Fall'03 final exam
  - problem: writing GUI clients is pain; simplify it!

A window contains nested windows



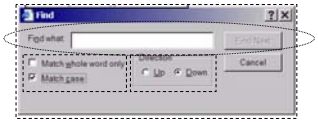
Current programming model

```
Window top = new Window(null); top.setTitle(``Find'');
Text t = new Text(top); t.setPosition(0,0); t.setLabel("Find what:");
Dialog d = new Dialog(top); d.setPosition(20,0); d.setWidth(18*someConstant);
Button f = new Button(top);  setType(REGULAR_BUTTON); f.setPosition(80,0);  f.setLabel("Find Next");
Window w1 = new Window(top); w1.setPosition(0,50);
Selection s1 = new Selection(w1);  s1.setPosition(0,0); s1.setLabel("Match whole word only");
Selection s2 = new Selection(w1);  s2.setPosition(0,50); s2.setLabel("Match case"); s2.setSelected(true);
Window w2 = new Window(top); w2.setPosition(45,50); w2.setTitle("Direction");   w2.setFramed(true);
Button r1 = new Button(w2);  r1.setType(RADIO); r1.setPosition(0,0); r1.setLabel("Up");
Button r2 = new Button(w2);  r2.setType(RADIO); r2.setPosition(50,0); r2.setLabel("Down");
     r2.setSelected(true);
Button c = new Button(top);  c.setType(REGULAR_BUTTON); c.setPosition(80,50); c.setLabel("Cancel");
top.draw();
```

(See Fall'03 final exam for comments and details)

Design yourself a better programming model

```
window "Find" {
   "Find what:"  <------------------>  ["Find next"],
   window "" {
       x "Match whole word only",
       X "Match case"
   }
   window framed "Direction" {
       o "Up"
       O "Down"
   }
   ["Cancel"]
}
```



the structure of a compiler

Three execution environments

- Interpreters
  - Scheme, lisp, perl, python
  - popular interpreted languages later got compilers
- Compilers
  - C
  - Java (compiled to bytecode)
- Virtual machines
  - Java bytecode runs on an interpreter
  - interpreter often aided by a JIT compiler

The Structure of a Compiler

1. Scanning (Lexical Analysis)
2. Parsing (Syntactic Analysis)
3. Type checking (Semantic Analysis)
4. Optimization
5. Code Generation

The first 3, at least, can be understood by analogy to how humans comprehend English.

Lexical Analysis

- Lexical analyzer divides program text into "words" or "tokens"

  if x == y then z = 1; else z = 2;

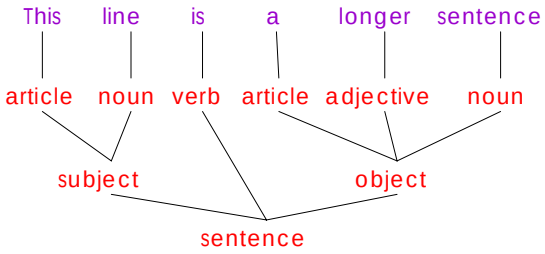
- Units:

  if, x, ==, y, then, z, =, 1, ;, else, z, =, 2, ;

Parsing

- Once words are understood, the next step is to understand sentence structure
- Parsing = Diagramming Sentences
  - The diagram is a tree

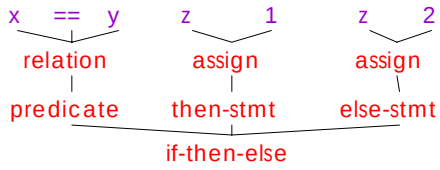
Diagramming a Sentence



Parsing Programs

- Parsing program expressions is the same
- Consider:

  If x == y then z = 1; else z = 2;
- Diagrammed:



Semantic Analysis in English

- Example:

  Jack said Jerry left his assignment at home.

  What does "his" refer to? Jack or Jerry?
- Even worse:

  Jack said Jack left his assignment at home?

  How many Jacks are there?

  Which one left the assignment?

Semantic Analysis I

- Programming languages define strict rules to avoid such ambiguities
- This Java code prints "4"; the inner definition is used

```
{
  int Jack = 3;
  {
      int Jack = 4;
      System.out.
            print(Jack);
  }
}
```


Semantic Analysis II

- Compilers also perform checks to find bugs
- Example:

  Jack left her homework at home.
- A "type mismatch" between her and Jack
  - we know they are different people (presumably Jack is male)

Code Generation

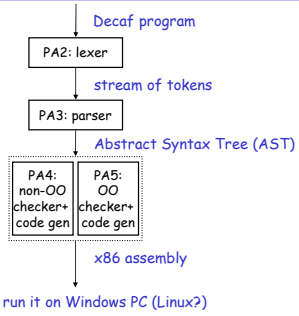
- A translation into another language
  - Analogous to human translation
- Compilers for Java, C, C++
  - produce assembly code (typically)
- Code generators
  - produce C or Java

the project

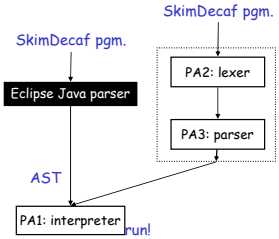
The project

- A compiler for Decaf (a small Java)
  - PA1: interpreter of a subset of Decaf, a warm-up
  - PA2-5: the compiler of Decaf, in four easy pieces
    - PA2: scanner (plus scanner generator)
    - PA3: parser (plus parser generator)
    - PA4: code generator for non-OO features
    - PA5: code generator for OO features

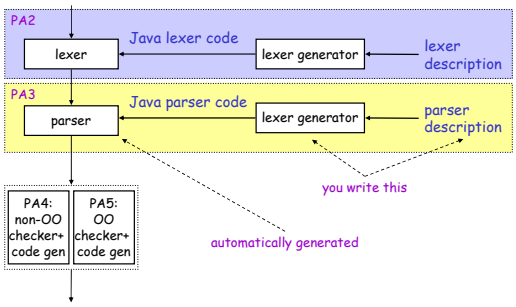
The Decaf compiler



PA1: SkimDecaf Interpreter



How you will implement the scanner, parser



course logistics

(see course info on the web for more)

Academic (Dis)honesty

- Read the policy at:
  - http://www.eecs.berkeley.edu/Policies/acad.dis.shtml
- We'll be using a state-of-the art plagiarism detector.
  - Before you ask: yes, it works very well.
- You are allowed to discuss the assignment
  - but you must acknowledge (and describe) help in your submission.

Grading

- This is going to be a fun course, but graded on a on a curve, as customary
  - so, yes, you're competing against one another.
- grades will follow department guidelines
  - course average GPA will be around 2.9 (before extra credit for the optimization contest)
  - more at http://www.eecs.berkeley.edu/Policies/ugrad.grading.shtml
  - this has proven to be fair and just
- A lot of grade comes from a project
  - form a strong team
  - use the course newsgroup to find a partner

Remote testing

- A new testing infrastructure
  - to help you debug your compiler
  - being introduced this semester (may have a few rough edges, so bear with us)
- The rationale:
  - give you (indirect) access to our reference solution
  - you will be able to compile and run your Decaf programs on our compiler
    - and check if your compiler behaves like our compiler does

Remote testing

- The process:
  - you write test programs to test your compiler
  - store them with your compiler in a CVS repository
  - our scripts will pick them up and run your tests on your compiler and also our compiler
    - mismatch in outputs indicates a bug (guess in whose code)
  - our scripts will also measure "test coverage"
    - what fraction of our compiler did your tests execute
    - low coverage indicates you didn't write enough tests, and hence a bug in your code may be undetected
  - you pick up results of remote testing via CVS
    - and display them using a special Eclipse plugin (on cs164 web site)

conclusion

Why are you taking cs164?

- To learn how languages are executed
  - compiler is programmer's most frequently used tool
  - be prepared for new languages
- To go through a cool project
  - where major parts are automatically generated
  - with your own generators!
- To develop your own small languages
  - and a compiler (or code generator) for it
  - become a more productive programmer

Take cs164. Become unoffshorable.

"We design them here, but the labor is cheaper in Hell."