Introduction to Programming Languages and Compilers: Parsing and More

CS164

Recall, Lexical Analysis

- A 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, ;
```

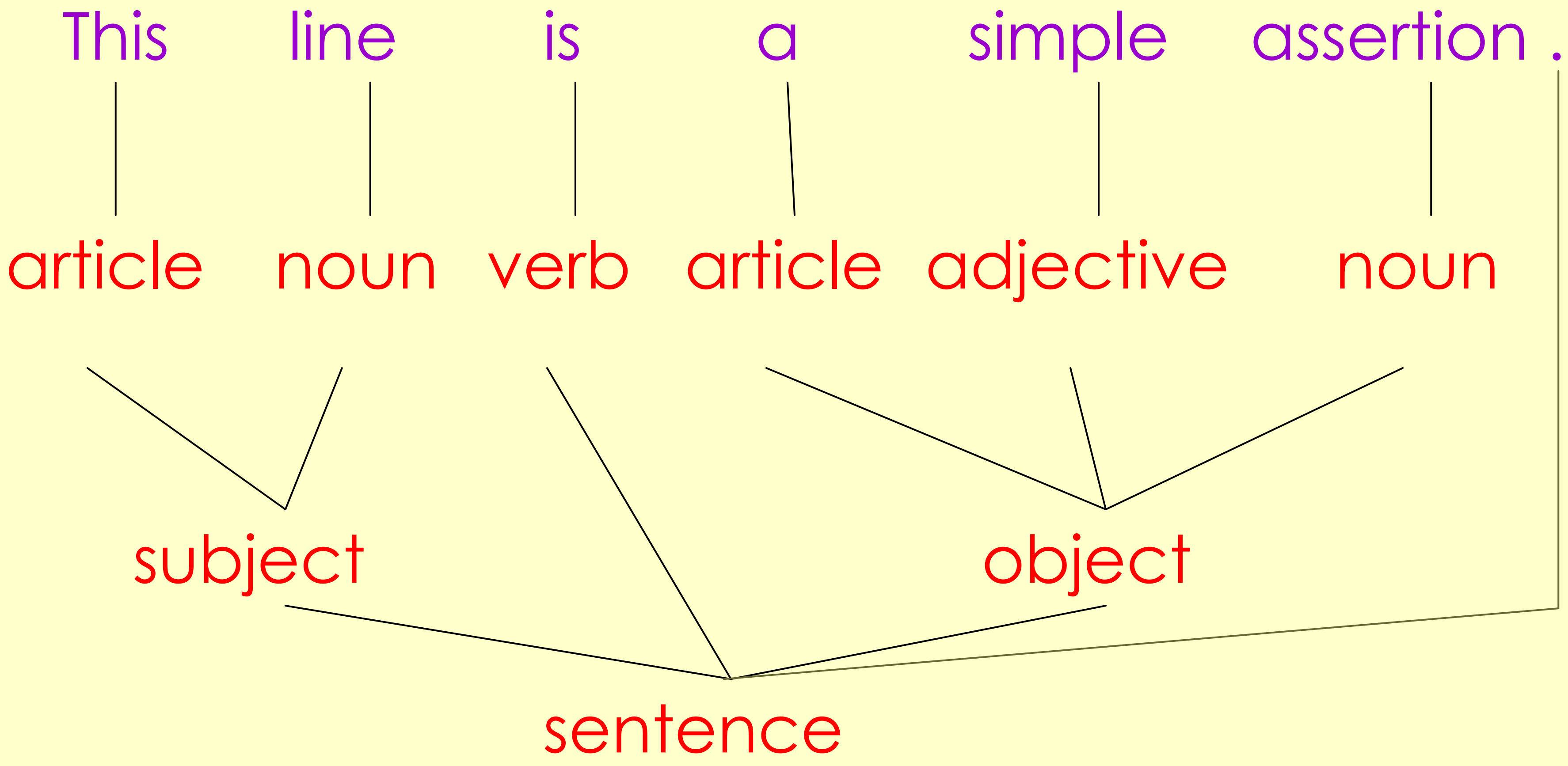
- Some tokens variously categorized as operators, numbers, identifiers, keywords.

Parsing

- Once words are understood, the next step is to understand sentence structure as we saw in CS61a

- Parsing = Diagramming Sentences
  - The diagram is a tree

Diagramming a Sentence

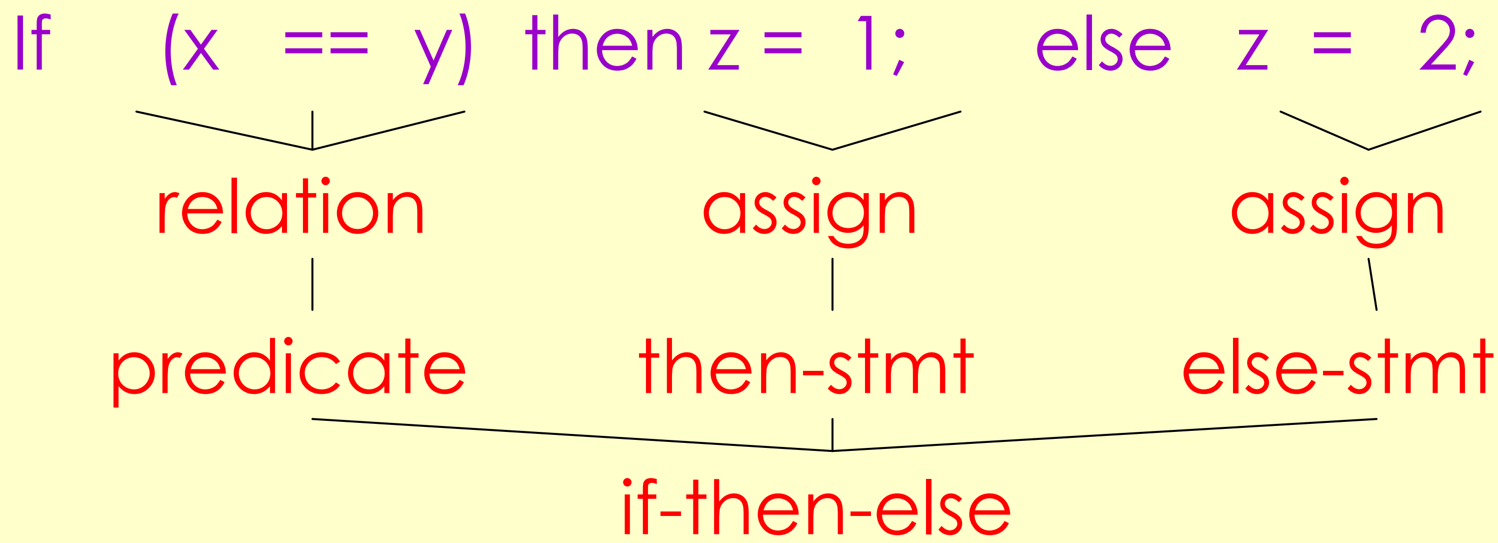


Parsing Programs

- Parsing program expressions is similar
- Consider (in some typical language):

  If (x == y) then z = 1; else z = 2;

- Diagrammed:



Semantic Analysis

- Once we have a diagram of the sentence structure, it is easier to write a program that attributes to it some "meaning."
- Some texts are not sentences (syntax errors).
- Some texts are sentences, but still have bugs (e.g. type errors, uninitialized variables).
- Some texts have no "language" errors but just compute the wrong thing. Or nothing. (SEMANTICS)
- Compilers perform limited analysis to catch inconsistencies "soon".

Semantic Analysis in English

- Example:

  Jack said Jerry left his hat at home.

  What does “his” refer to? Jack or Jerry?

- Even worse:

  Jack said Jack left his hat at home.

  How many Jacks are there?

  Which one left the hat?

  Jack said Jack left his xyzzy at lalalala

Semantic Analysis in Programming

- Programming languages define strict rules to avoid such ambiguities

- This C++ code prints "4"; the inner definition is used

```
{
  int Jack = 3;
  {
     int Jack = 4;
     cout << Jack;
  }
}
```

More Kinds of Semantic Analysis

- Compilers perform many semantic checks besides variable bindings
- Example in English:

  Jack left her hat at home.

- Is there a "type mismatch" between her and Jack? Yes, if we know Jack is male and her refers to Jack.

Corresponding Semantic Analysis in a compiler..

- Limited analysis can catch inconsistencies “soon”. 34+"abc" is, in most languages, unacceptable.

- Some processors do more analysis to improve the performance of the program. if true then 2+3 else 6 is probably the same as 5
- But this is kind of change is generally called <u>optimization...</u>

Optimization

- English optimization: "In order to be understood most clearly, you should make your statements in the shortest and most straightforward way." →"Be brief"
- For computer language implementation: modify programs so that they compute the same answer but
  - Run faster and/or
  - Use less memory
  - In general, conserve some resource
  - May remove error checks.
- The class project will not include much optimization, though much current CS compiler research involves this. Squeezing performance out of new weird architectures (parallel, pipelined, distributed...)

Optimization Example

$X = Y * 0$ is the same as $X = 0$

Maybe NO!

For some languages, this may be OK for integers, but not for floating point numbers

Almost the last task: Code Generation

- Produces assembly code (usually). Though it might generate binary instructions (even lower level) or bytes code for a "virtual" machine. (Java, CLISP)
- There may be another optimization pass after code generation.
  - Just-in-time compilation of Java byte codes

A trick that is used repeatedly: Intermediate Languages (IL)

- Many compilers perform translations between successive intermediate forms
  - All but first and last are *intermediate languages* internal to the compiler
  - We will use Lisp data structures for several. All these languages are encoded as trees (lists).
- ILs generally ordered in descending level of abstraction
  - Highest is source
  - Lowest is assembly

Intermediate Languages (Cont.)

- IL's are useful because lower levels expose features hidden by higher levels. At low level we may see
  - registers : load 1, a; load 2,b; add 1,2; store 1,c
  - memory layout: data on stack, global data, ptrs
  - Prefetch instructions,
  - Rearrangement to avert pipeline stalls
- But lower levels obscure high-level meaning: c:=a+b

Some pervasive issues for implementation

- Compiling can be almost as simple as we've indicated, but there are many pitfalls.
  - How are erroneous programs handled?
  - How to know if the compiler is correct?
  - How to know if a program is "safe" to run?

- Language design has a big impact on compiler.
  - Determines what is easy and hard to compile
  - Course theme: many trade-offs in language design and implementation

Compilers Today

- The overall structure of almost every compiler adheres to our outline.
- The relative efforts devoted to different phases have changed since FORTRAN
  - Early: lexing, parsing most complex, expensive, neat techniques were just being figured out (1955-60)
  - Emphasis on "phases" because programs did not fit in memory! (Some compilers had 20 or more)
  - Today: optimization dominates all other phases, lexing and parsing are cheap.

Trends in Compilation

- Compilation for absolutely highest speed is not the sole criterion except in a few cases:
  - large-scale scientific programs
  - Speed-critical embedded systems (Digital Signal Processors, advanced speculative architectures)
  - Marketing of hardware (benchmarks).
- Ideas from compilation used for improving code reliability:
  - memory safety
  - detecting data races
  - ...

Of all the languages .. why use Lisp?

- http://www.paulgraham.com/avg.html

yahoo!stores is written in Lisp. Our text author explains how this is not accidental. (the travel site, orbitz.com is also lisp based)

See also the other interesting articles on the paulgraham web site.)

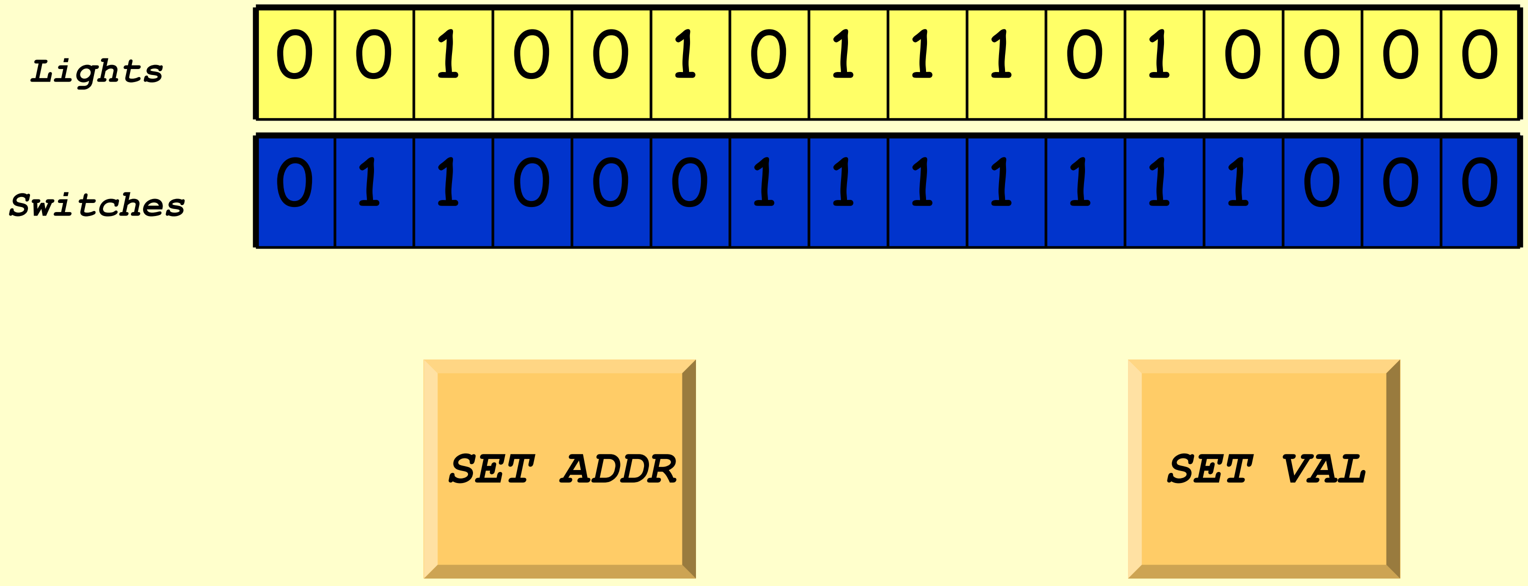
The Lisp environment supports PROGRAMMERS first and foremost.

Still to Do REAL SOON

- Learn to run emacs/lisp on your favorite machine
- Read ANSI CL (see homework sheet for specifics) and get started on ASSIGNMENT 1.
- Make sure you can get to at least one discussion section
- Assignment zero (picture, please)

A brief survey of programming languages: motivation and design

PDP-11/10 Programming Language Interface



0	= switch set off

Why use programming languages at all?

The natural language for binary digital computers is streams of 0 and 1. Imagine how difficult it might be to program a computer on a “naked” PDP-11/10.

set 16 on-off switches to a 16 bit binary address. Press
"set address”
set 16 switches to the contents you wish loaded there.
Press
"set value”

the circuitry conveniently provided that repeated presses of "set value” set values in address +2 , +4 , etc.

What is the first program YOU would write?

The first (and only) program loaded in this way was---

a program to load additional programs from some other medium like cards, tape, disk, network interface, ...

This is a painful interface.

Programming a computer to bend toward human readability is a natural goal.

For humans.

If there were no humans..

If computers were conveying programs to other computers, a stream of 0,1 .. would be pretty good.

Other objectives of a computer-to-computer language would be compactness and perhaps error-checking.

Could a Pentium send a program to a Macintosh PowerPC this way?

Humans are more important

Can we program a computer so that a vague description of a procedure in a "natural" language like English will get it going?

(Startrek fans .... perhaps you recall "I, Mudd" TOS, where Spock tries to confound the central android control by asking it (Norman) to "compute the last digit of pi"... It goes haywire and starts talking in a squeeky voice, ...)

Humans are more important

So I tried . I asked it "What is the last digit of pi" and it didn't talk in a squeeky voice.
It offered various web pages… For example,

David Bailey, Peter Borwein and Simon Plouffe have recently computed the ten billionth digit in the hexadecimal expansion of pi. They utilized an astonishing formula:

$$\pi = \sum_{n=0}^{\infty} \left(\frac{4}{8 \cdot n+1} - \frac{2}{8 \cdot n+4} - \frac{1}{8 \cdot n+5} - \frac{1}{8 \cdot n+6} \right) \cdot \left(\frac{1}{16} \right)^n$$

In reality, plain language is not really the goal of most computer language designers: English is too ambiguous and context dependent. [Define those words…]

History/ Prog. Lang. Timeline

http://www.computerhistory.org/timeline

Some of the first programming languages were assembly language systems, or very simple interpreters which provided floating-point "instructions" on top of machine language. Some fundamental ideas in using stored-program computers go back to the mid 1940's (ENIAC was built in 1942).. like subroutines, repetition, and conditional execution based on logic.

The usual histories place the burden of being the first programming language on Fortran (1957), with nods to Algol (1959), Lisp (1959), COBOL (1961) and shortly thereafter an enormous blossoming of languages, most of which have since died out. [[Why so many?: different application domains with sometimes conflicting needs: business, science, logic/AI, OS programming]]

There is now recognition that Konrad Zuse anticipated many language ideas in his Plankalkul language (in 1945, in Germany).

Continued next time… Languages galore

Why were there so many?
Not sufficiently pretty?
Not sufficiently productive?
Support new “methodologies” [an embarrassing term at best]