Compilers

Lecture 2

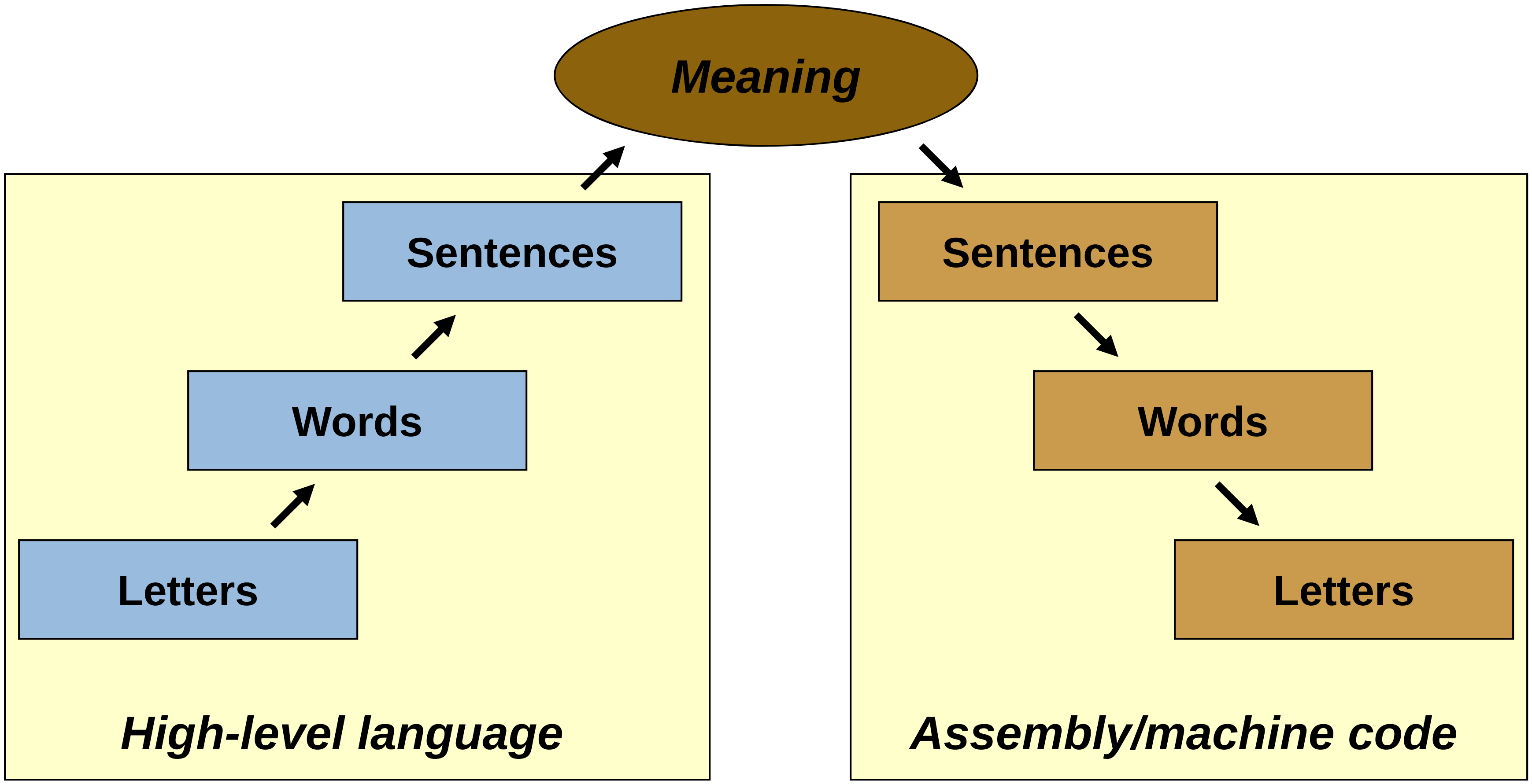
Overview

Yannis Smaragdakis, U. Athens
(original slides by Sam Guyer@Tufts)

Last time…

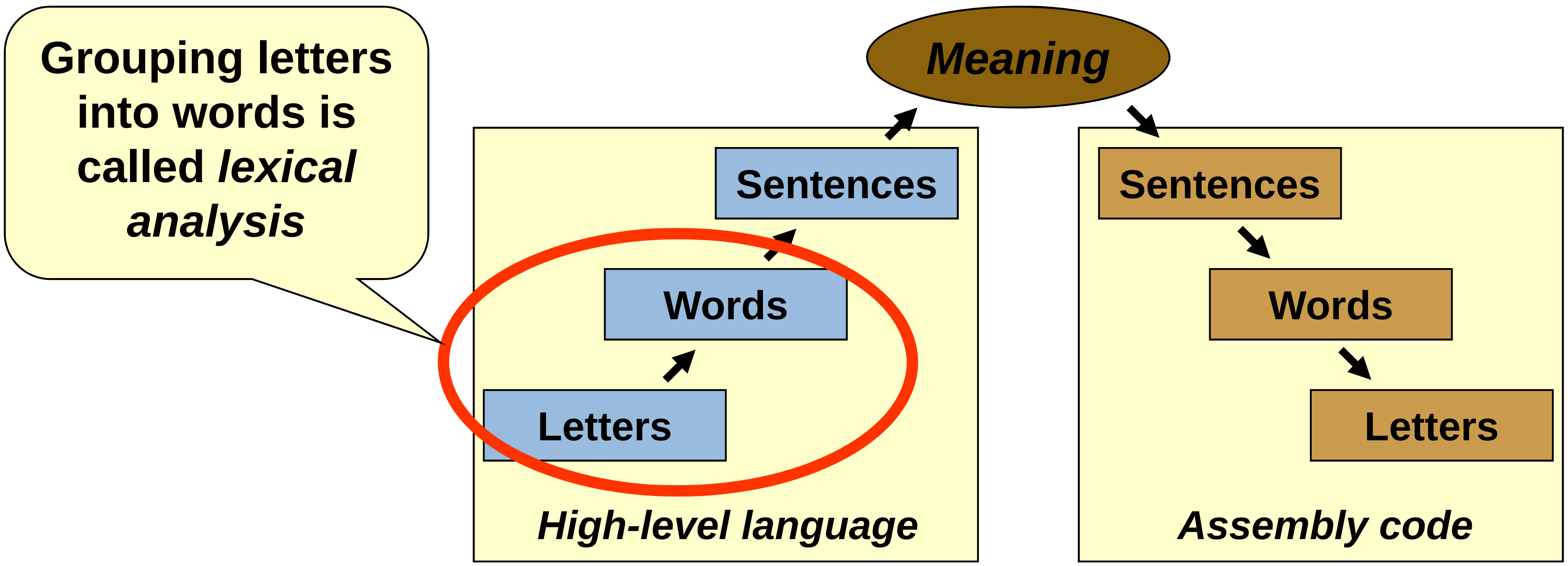
- The compilation problem
  - Source language
    - High-level abstractions
    - Easy to understand and maintain
  - Target language
    - Very low-level, close to machine
    - Few abstractions
- Concerns
  - Systematic, correct translation
  - High-quality translation

Translation strategy



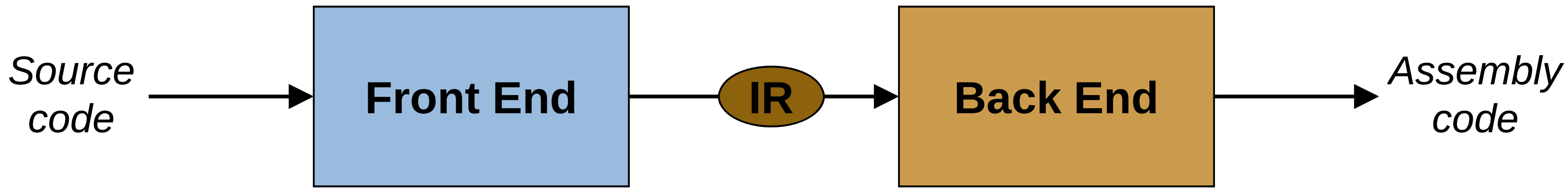
Compilation strategy

- Follows directly from translation strategy:



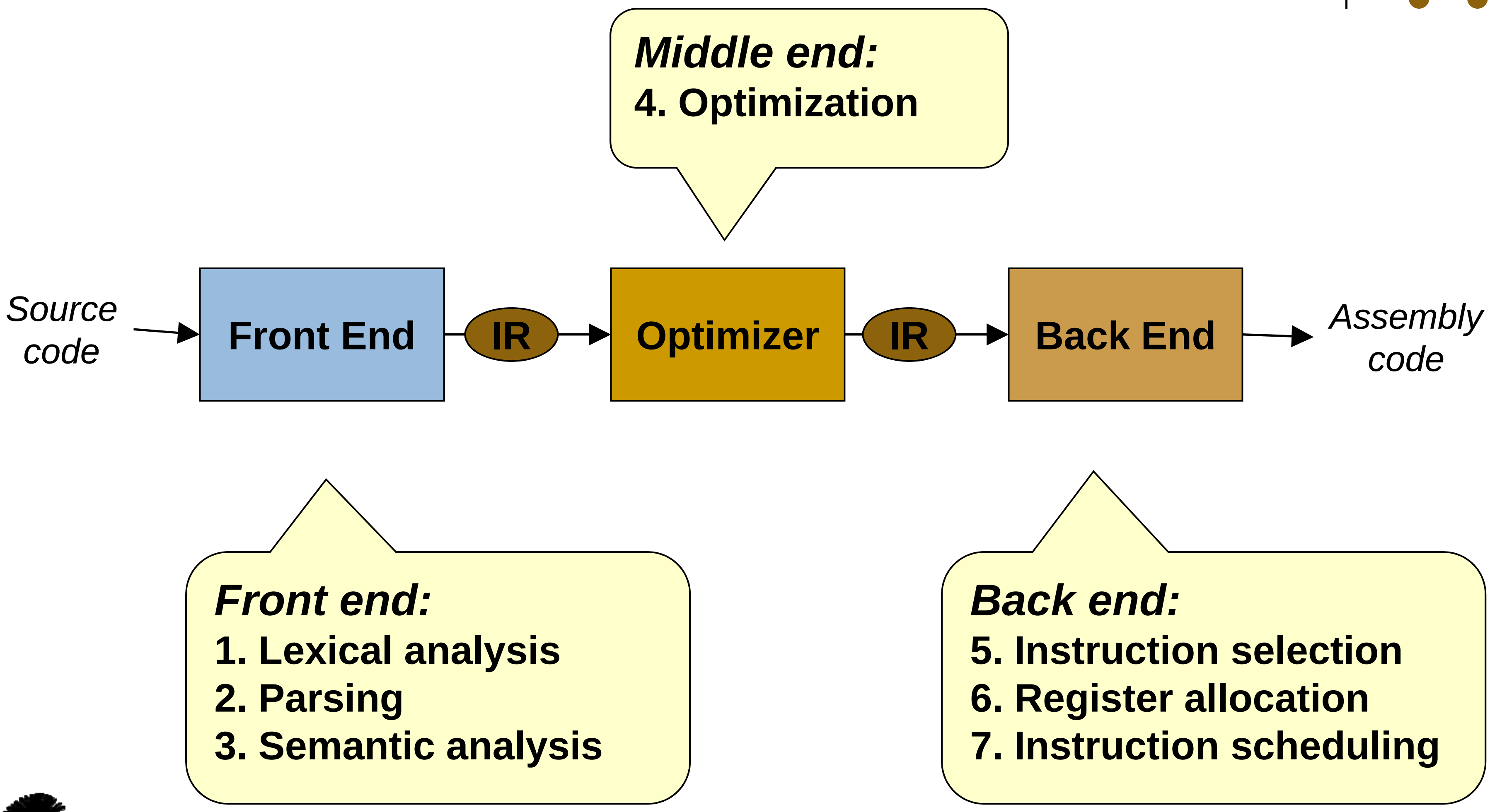
- A series of ***passes***
  - Each pass performs one step
  - Transforms the program representation

Basic compiler structure

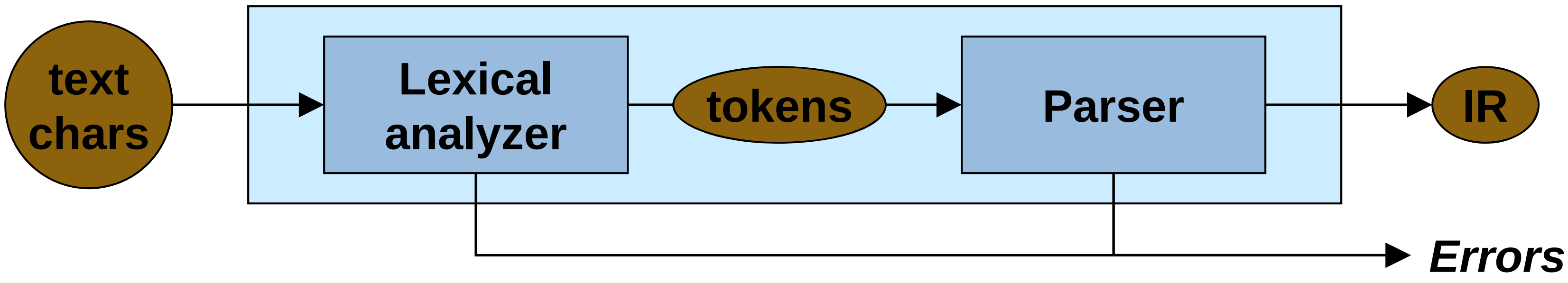


- Traditional two-pass compiler
  - ***Front-end*** reads in source code
  - ***Internal representation*** captures meaning
  - ***Back-end*** generates assembly
- Advantages?
  - Decouples input language from target machine

Modern optimizing compiler



The front end



- Responsibilities?
  - Recognize legal (and illegal programs)
  - Report errors in a useful way
  - Generate internal representation
- How it works
  - ***Good news***: linear time, mostly generated automatically
  - By analogy to natural languages…

Lexical Analysis

- First step: recognize words.
  - Smallest unit above letters

This is a sentence.

- Some lexical rules
  - Capital “T” (start of sentence symbol)
  - Blank “ “ (word separator)
  - Period “.” (end of sentence symbol)

More Lexical Analysis

- Lexical analysis is not trivial. Consider:

  ist his ase nte nce

- Often a key question:
  - What is the role of “white space” in the language?

- Plus, programming languages are typically more cryptic than English:

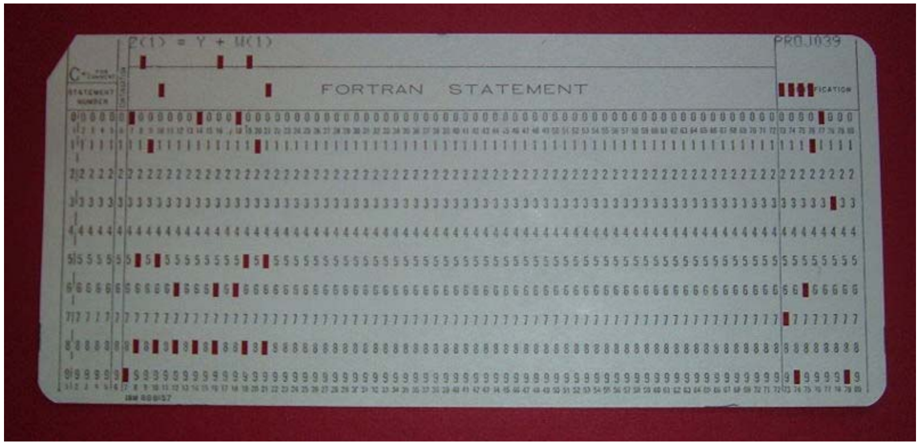
  *p->f ++ = -.12345e-5

Early compilers

- Strict formatting rules:

```
C AREA OF THE TRIANGLE
  799 S = (IA + IB + IC) / 2.0
      AREA = SQRT( S * (S - IA) * (S - IB) *
     +     (S - FLOATF(IC)))
      WRITE OUTPUT TAPE 6, 601, IA, IB, IC, AREA
```

- Why?
  - Punch cards!
  - And it's easier



Lexical analysis

- Another example:

```
void func(float * ptr, float val)
{
  float result;
  result = val/*ptr;
}
```

- Why is this case interesting?
  - “/*” is the comment delimiter

Lexical Analysis

- Lexical analyzer divides program text into “words” or ***tokens***

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

- Tokens have value and type:

  <if, *keyword*>, <x, *identifier*>, <==, *operator*>, etc….

Specification

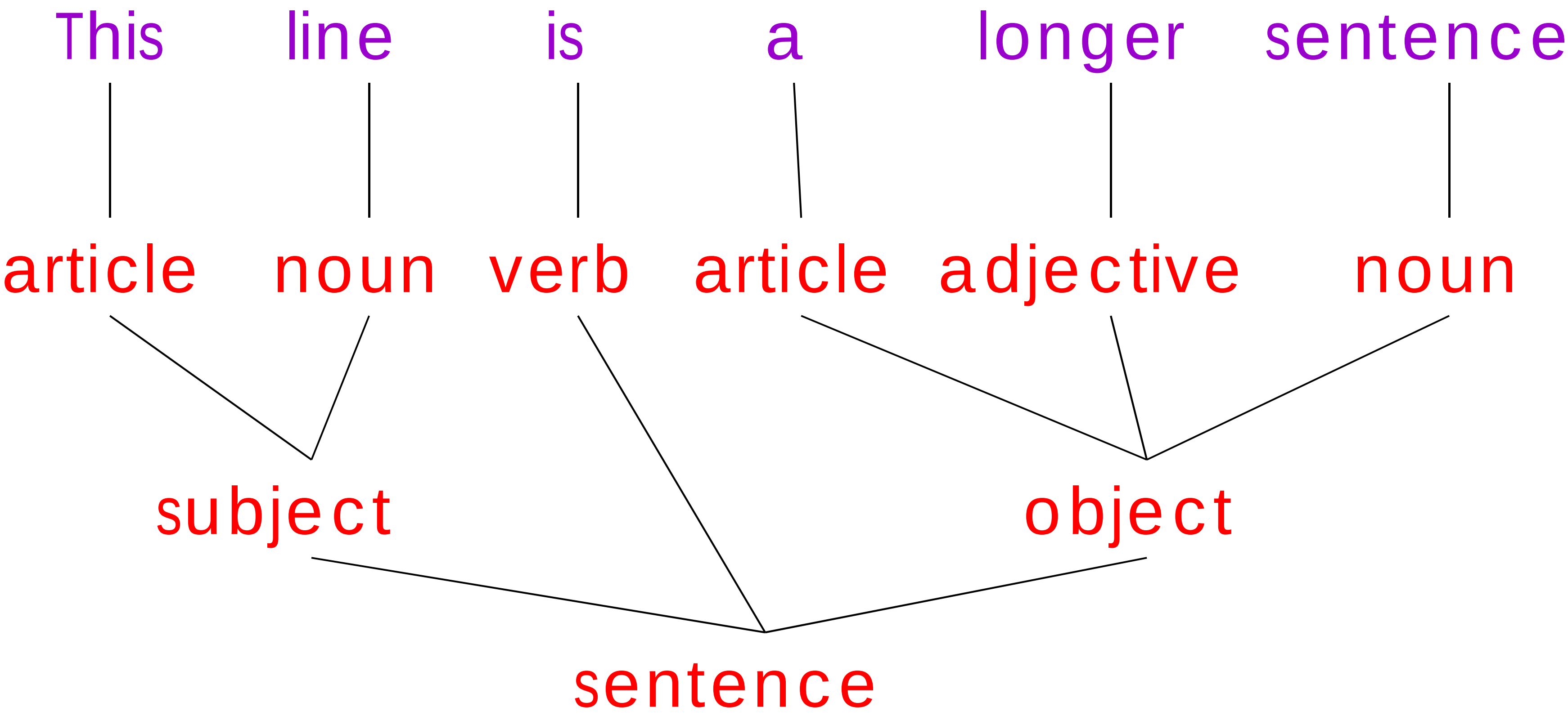
- How do we specify tokens?
  - Keyword – an exact string
  - What about identifier? floating point number?

- Regular expressions
  - Just like Unix tools grep, awk, sed, etc.
  - Identifier: `[a-zA-Z_][a-zA-Z_0-9]*`
  - Algorithms for matching regexps
    - Actually, generate code that does the matching
    - This code is often called a ***scanner***

Parsing

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

Diagramming a Sentence

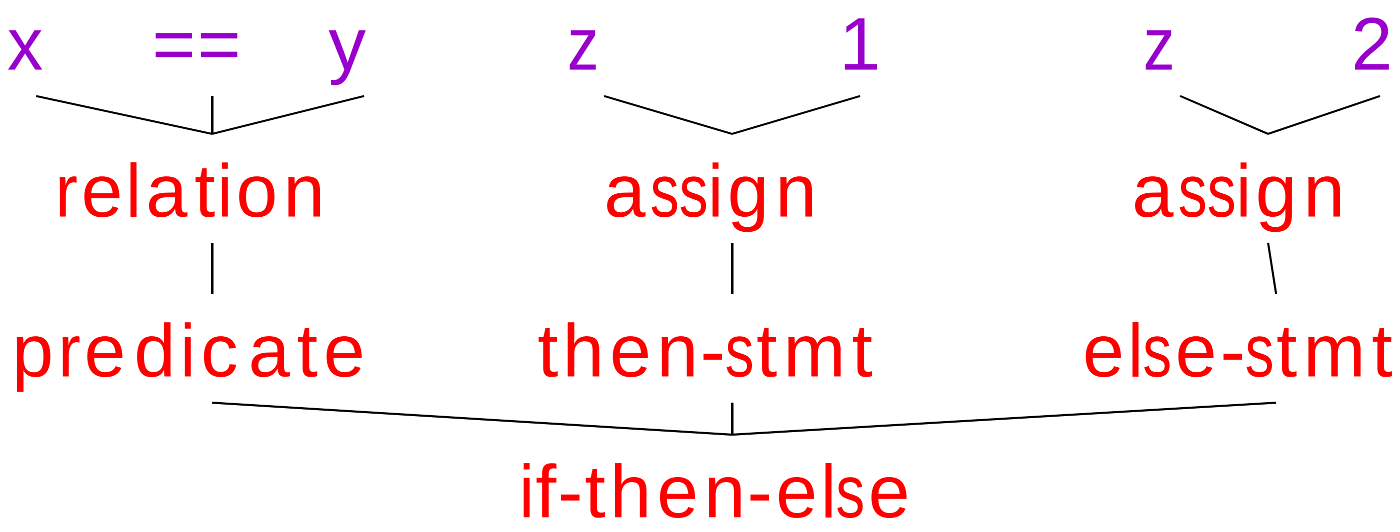


Diagramming programs

- Diagramming program expressions is the same
- Consider:

```
If x == y then z = 1; else z = 2;
```

- Diagrammed:



Specification

- How do we describe the language?

  Same as English: using grammar rules

```
1. sentence → subject  verb  object
2. subject → noun-phrase
3. noun-phrase → article  noun-phrase
4.                      |  adjective noun-phrase
5.                      |  noun
  ...etc...
```

```
1. goal → expr
2. expr → expr  op  term
3.         |  term
4. term → number
5.         |  id
6. op    → +
7.         |  -
```

Tokens from scanner

- Formal grammars
  - Chomsky hierarchy – ***context-free grammars***
  - Each rule is called a ***production***

Using grammars

- Given a grammar, we can ***derive*** sentences by repeated substitution
- ***Parsing*** is the reverse process – given a sentence, find a derivation (same as diagramming)

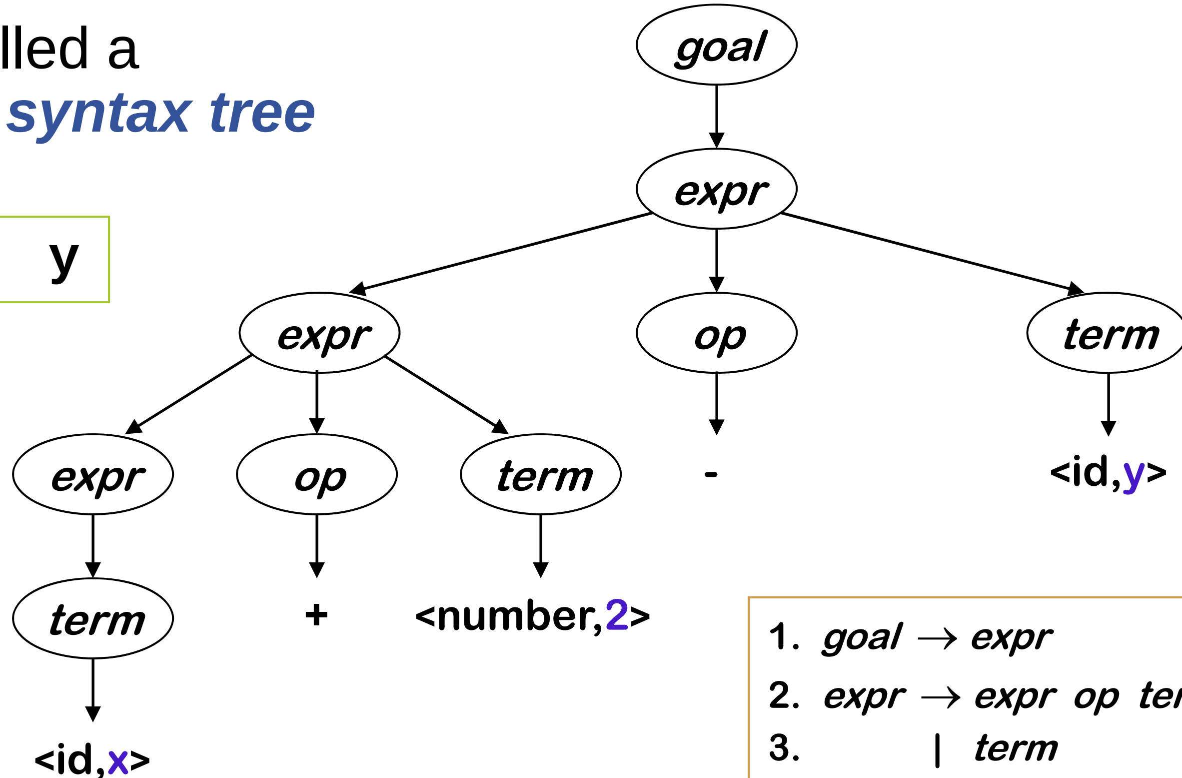
1. *goal* → *expr*
2. *expr* → *expr op term*
3. | *term*
4. *term* → <u>number</u>
5. | <u>id</u>
6. *op* → +
7. | -

Production	Result
	goal
1	*expr*
2	*expr op term*
5	*expr op* y
7	*expr* - y
2	*expr op term* - y
4	*expr op* 2 - y
6	*expr* + 2 - y
3	*term* + 2 - y
5	x + 2 - y

Representation

- Diagram is called a ***parse tree*** or ***syntax tree***

`x + 2 - y`

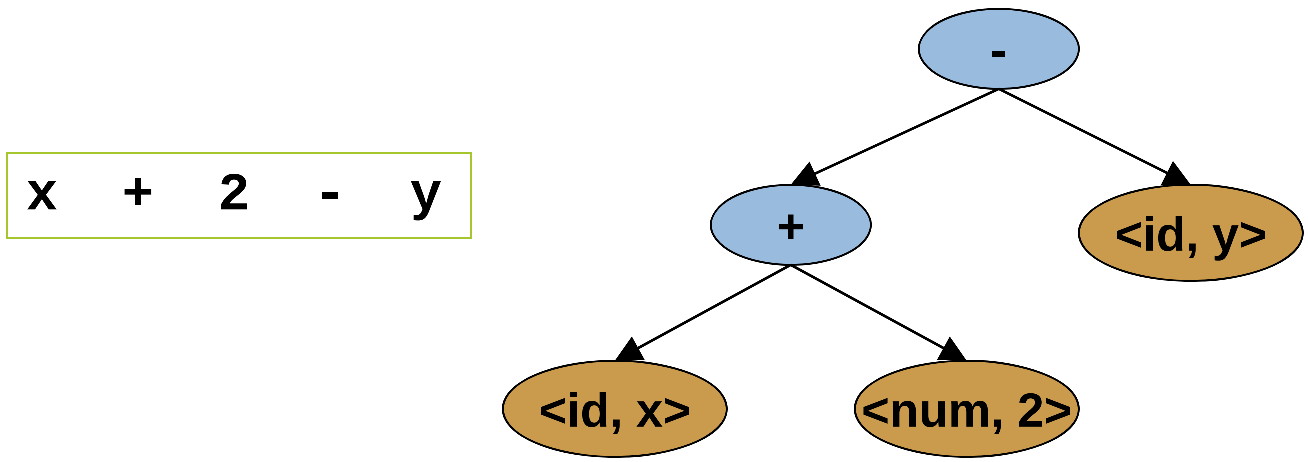


1. *goal* → *expr*
2. *expr* → *expr op term*
3. | *term*
4. *term* → <u>number</u>
5. | <u>id</u>
6. *op* → +
7. | -

- <u>Notice</u>: Contains a lot of unneeded information.

Representation

- Compilers often use an ***abstract syntax tree***



- More concise and convenient:
  - Summarizes grammatical structure without including all the details of the derivation
  - ASTs are one kind of ***intermediate representation*** *(IR)*

Semantic Analysis

- Once sentence structure is understood, we can try to understand "meaning"
  - What would the ideal situation be?
  - Formally check the program against a specification
  - This capability is coming
- Compilers perform limited analysis to catch inconsistencies
- Some do more analysis to improve the performance of the program

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 in programs

- Programming languages define strict rules to avoid such ambiguities

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

- What does this code print? Why?
  - This Java code prints "4"; the inner-most declaration is used.

More Semantic Analysis

- Compilers perform many semantic checks besides variable bindings

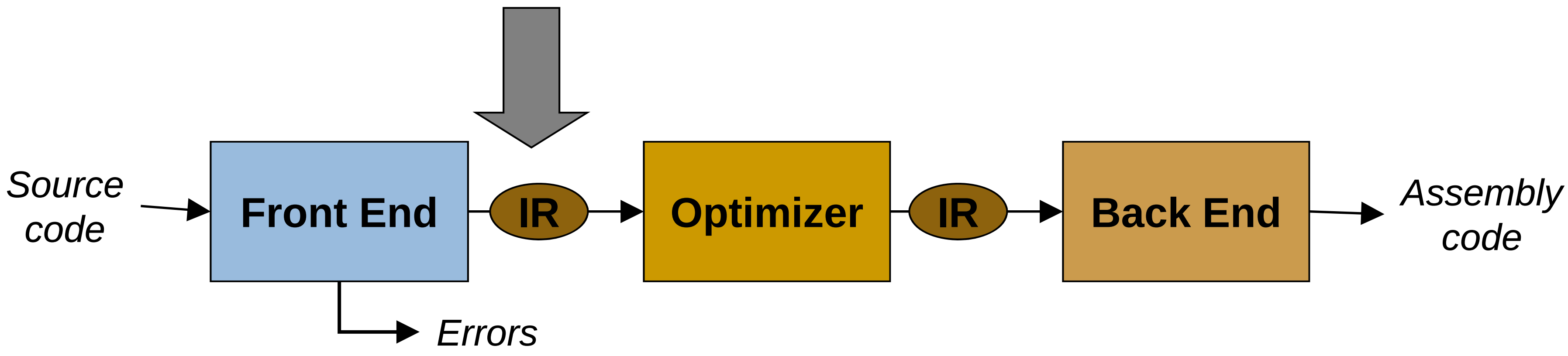
- Example:

  Jack left her homework at home.

- A “type mismatch” between her and Jack; we know they are different people

  (I’m assuming Jack is male)

Where are we?



- Front end
  - Produces fully-checked AST
  - Problem: AST still represents source-level semantics

Intermediate representations

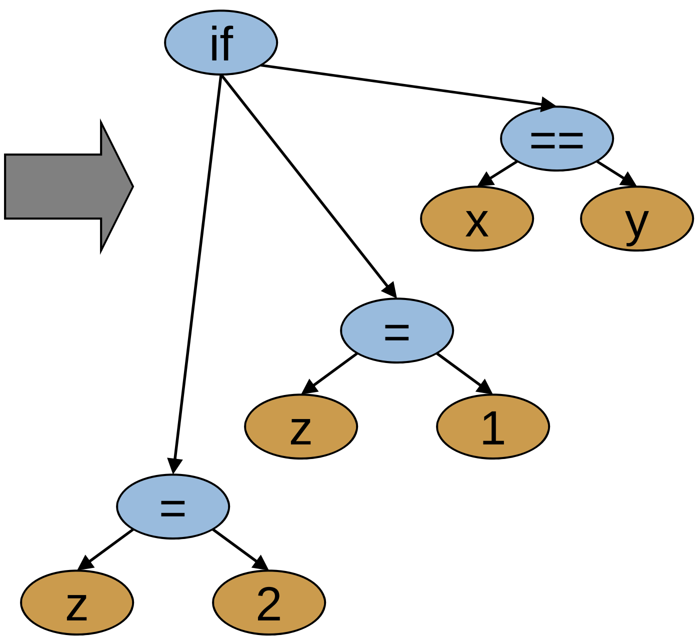
- Many different kinds of IRs
  - High-level IR (e.g. AST)
    - Closer to source code
    - Hides implementation details
  - Low-level IR
    - Closer to the machine
    - Exposes details (registers, instructions, etc)
  - Many tradeoffs in IR design

- Most compilers have 1 or maybe 2 IRs:
  - Typically closer to low-level IR
  - Better for optimization and code generation

IR lowering

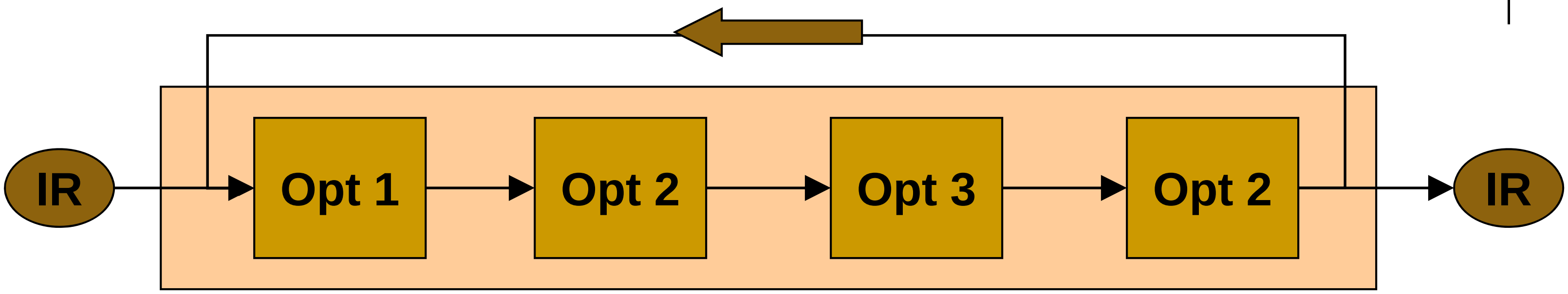
- Preparing for optimization and code gen
  - Dismantle complex structures into simple ones
  - Process is called ***lowering***
  - Result is an IR called ***three-address code***

```
if (x == y)
  z = 1;
else
  z = 2;
```




```
t0 = x == y
br t0 label1
goto label2
label1:
z = 1
goto label3
label2:
z = 2
label3:
```

Optimization



- Series of passes – often repeated
  - ***Goal***: reduce some cost
    - Run faster
    - Use less memory
    - Conserve some other resource, like power
  - Must preserve program semantics
- Dominant cost in most modern compilers

Optimization

- General scheme
  - Analysis phase:
    - Pass over code looking for opportunities
    - Often uses a formal analysis framework
  - Transformation phase
    - Modify the code to exploit opportunity
- Classic optimizations
  - Dead-code elimination, common sub-expression elimination, loop-invariant code motion, strength reduction
- This class: time permitting

Optimization example

- Array accesses

```
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
    A[i][j] = A[i][j] + C;
```

```
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++){
    t0 = &A + (i * M) + j
    (*t0) += C;
  }
```

```
for (i = 0; i < N; i++) {
  t1 = i * M
  for (j = 0; j < M; j++){
    t0 = &A + t1 + j
    (*t0) += C;
  }
}
```

```
t1 = 0;
for (i = 0; i < N; i++) {
  for (j = 0; j < M; j++){
    t0 = &A + t1 + j
    (*t0) += C;
  }
  t1 = t1 + M;
}
```

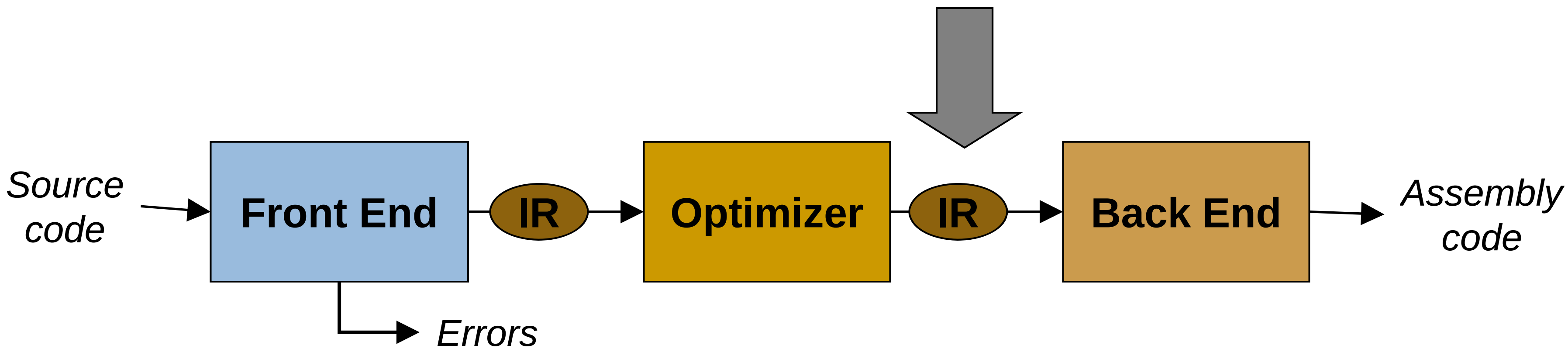
Optimization

- Often contain assumptions about performance tradeoffs of the underlying machine

  Like what?

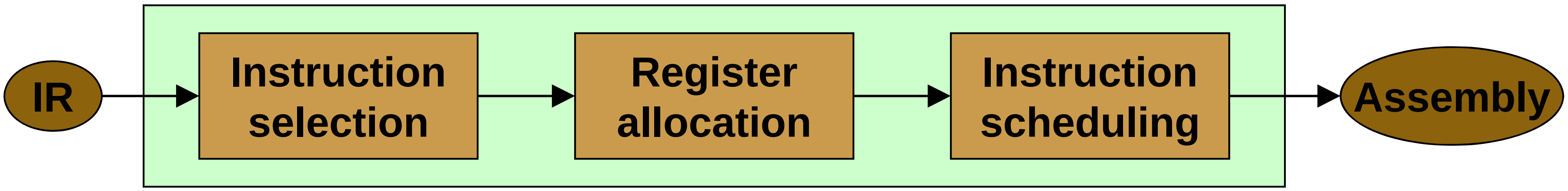
  - Relative speed of arithmetic operations – plus versus times
  - Possible parallelism in CPU
    - Example: multiple additions can go on concurrently
  - Cost of memory versus computation
    - Should I save values I've already computed or recompute?
  - Size of various caches
    - In particular, the instruction cache

Where are we?



- Optimization output
  - Transformed program
  - Typically, same level of abstraction

Back end



- Responsibilities
  - Map abstract instructions to real machine architecture
  - Allocate storage for variables in registers
  - Schedule instructions (often to exploit parallelism)

- How it works
  - ***Bad news***: very expensive, poorly understood, some automation

Instruction selection

- Example: RISC instructions

```
...
label1:
t1 = b * c
y = a + t1
z = d + t1
...
```

```
load @b         => r1
load @c         => r2
mult r1, r2     => r3
load @a         => r1
add r3, r1      => r1
store r1        => @y
load @d         => r1
add r3, r1      => r1
store r1        => @z
```

- Notice:
  - Explicit loads and stores
  - Lots of registers – “***virtual registers***”

Register allocation

- Goals:
  - Have each value in a register when it is used
  - Manage a limited set of resources
  - Often need to insert loads and stores

	Intel Nehalem	Intel Penryn
L1 Size / L1 Latency	64KB / 4 cycles	64KB / 3 cycles
L2 Size / L2 Latency	256KB / 11 cycles	6MB* / 15 cycles
L3 Size / L3 Latency	8MB / 39 cycles	N/A
Main Memory (DDR3)	107 cycles (33.4 ns)	160 cycles (50.3 ns)

- Algorithms
  - Optimal allocation is NP-complete
  - Many back-end algorithms compute approximate solutions to NP-complete problems

Instruction scheduling

- Change the order of instructions
  - Why would that matter?
  - Even single-core CPUs have parallelism
  - Multiple functional units – called superscalar
    - Group together different kinds of operations
    - E.g., integer vs floating point
  - Parallelism in memory subsystem
    - Initiate a load from memory
    - Do other work while waiting

Instruction scheduling

- Example:
  - Move loads early to avoid waiting
  - BUT: often creates extra register pressure

```
load @b        => r1
load @c        => r2
mult r1, r2    => r3
load @a        => r1
add r3, r1     => r1
store r1       => @y
load @d        => r1
add r3, r1     => r1
store r1       => @z
```

May stall on loads

```
load @b        => r1
load @c        => r2
load @a        => r4
load @d        => r5
mult r1, r2    => r3
add r3, r4     => r4
store r4       => @y
add r3, r5     => r5
store r5       => @z
```

Start loads early, hide latency, but need 5 registers

Finished program

- What else does the code need to run?

- Programs need support at run-time
  - Start-up code
  - Interface to OS
  - Libraries

- Varies significantly between languages
  - C – fairly minimal
  - Java – Java virtual machine

Run-time System

- Memory management services
  - Manage heap allocation
  - Garbage collection
- Run-time type checking
- Error processing (exception handling)
- Interface to the operating system
- Support of parallelism
  - Parallel thread initiation
  - Communication and synchronization