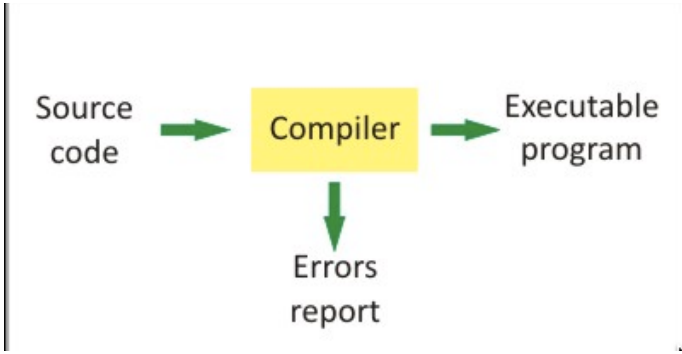
# Class 3

# Quiz

1. List two advantages of compiler over interpreter for implementing a language.
2. Explain what is compiled when in a JIT environment.

3. Give an example that shows you know the difference betweeen a token and a lexeme.



Source code

Compiler

Executable program

Errors report

**Recall we are translating…**

**From:**

```
float answer, initial, rate ;
answer := initial + rate * 60 ;
```

**To:**

```
load id3, R0
mult 60.0, R0
load id2, R1
add R1, R0
store R0, R1
```

# Lexical Analysis (Scanning)

**From:**

```
float answer, initial, rate ;
answer := initial + rate * 60 ;
```

**To:**

```
id <asgn> id <plus> id <mult> id
```

**Terminology:** lexeme, token, string table

# Syntax Analysis (Parsing)

**From:**

id <asgn> id <plus> id <mult> id

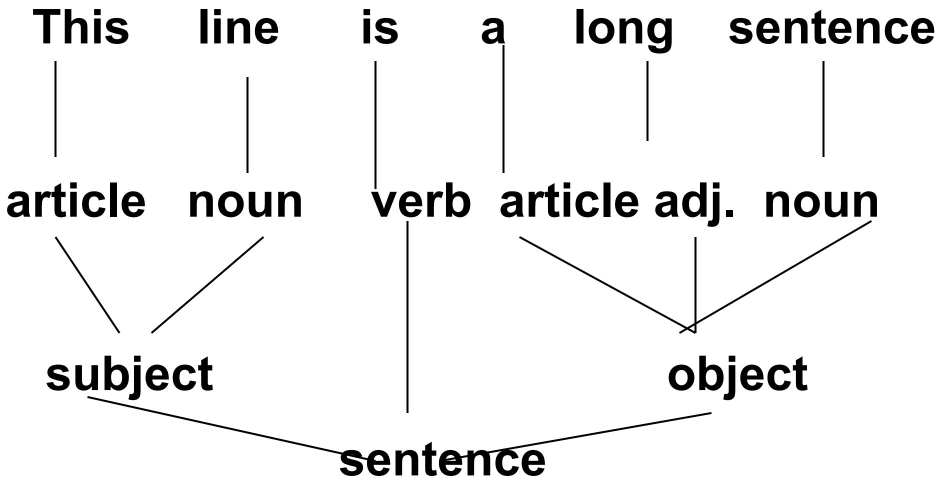
**To:**

```
          =
        /   \
     id        +
              /  \
            id     *
                  / \
                 id  60
```

**Terminology:** parse tree, ast

# Syntax Analyzer - Parser

Parsing similar to diagramming a natural language sentence



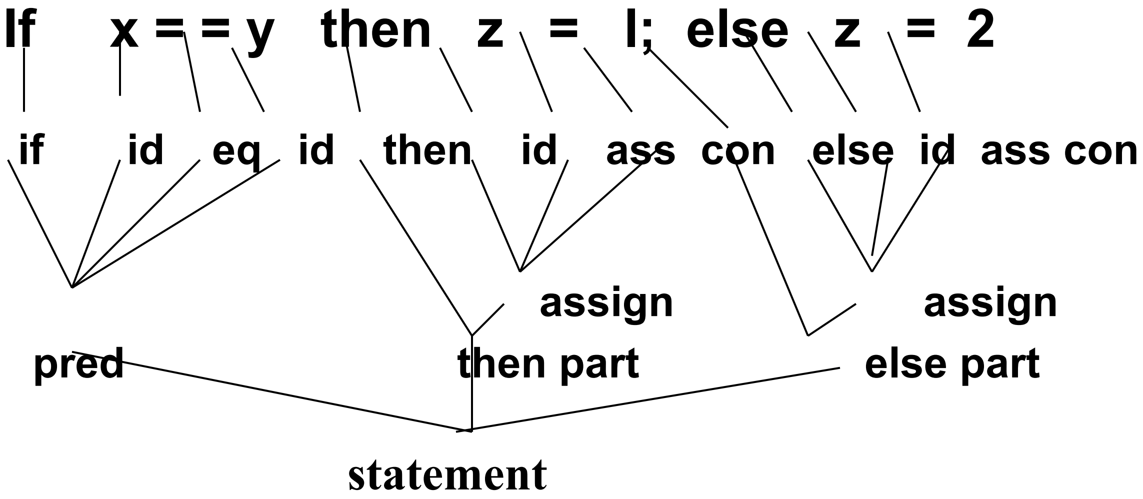
This line is a long sentence

article noun verb article adj. noun

subject object

sentence

## Parsing



If x = = y then z = l; else z = 2

if id eq id then id ass con else id ass con

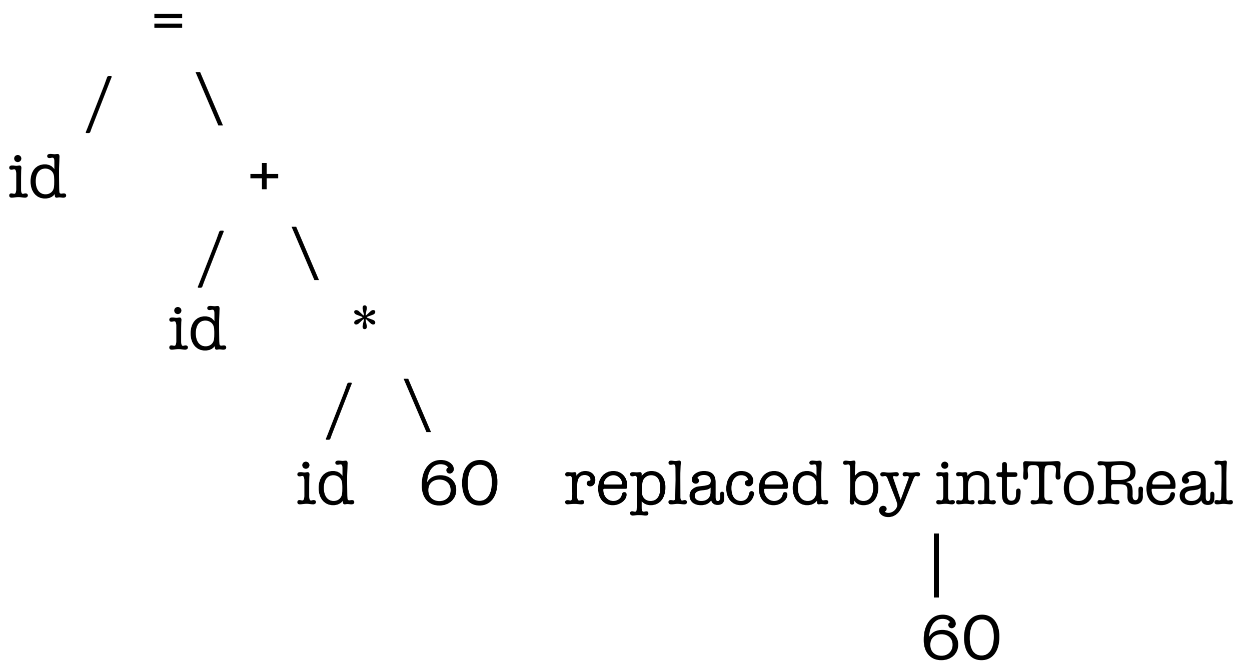
assign assign

pred then part else part

statement

# Semantic Analysis

**From/To:**

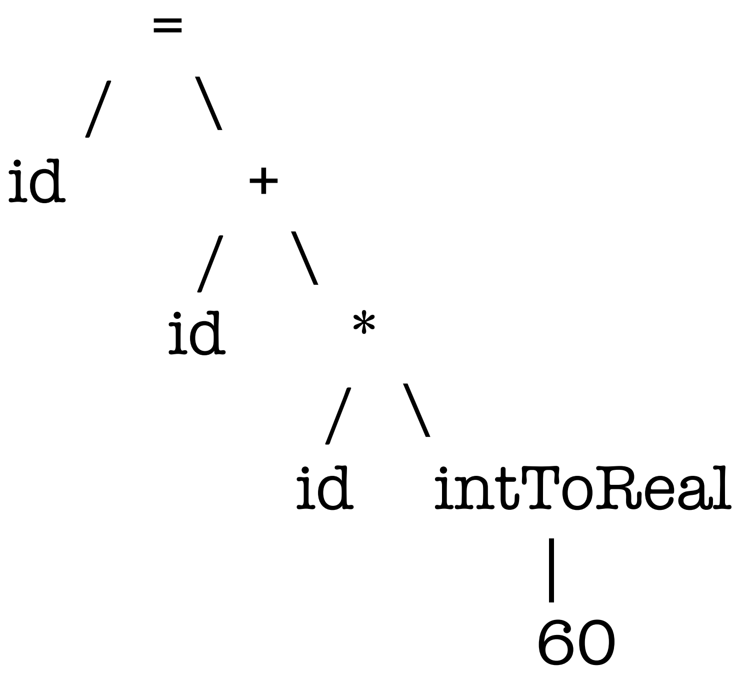


```
         =
       /   \
    id       +
           /   \
         id      *
               /   \
             id     60   replaced by intToReal
                                      |
                                      60
```

**Terminology:** type checking, scope, symbol table

# Intermediate Code Generation

**From:**



**To:**

```
temp1 = intToReal(60)
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3
```

**Terminology:** 3-address code, IR

# Code Optimization

**From:**

```
temp1 = intToReal(60)
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3
```

**To:**

```
temp1 = id3*60
id1 = id2 + temp1
```

**Terminology:** data flow analysis, code transformation

# Code Generation

**From:**

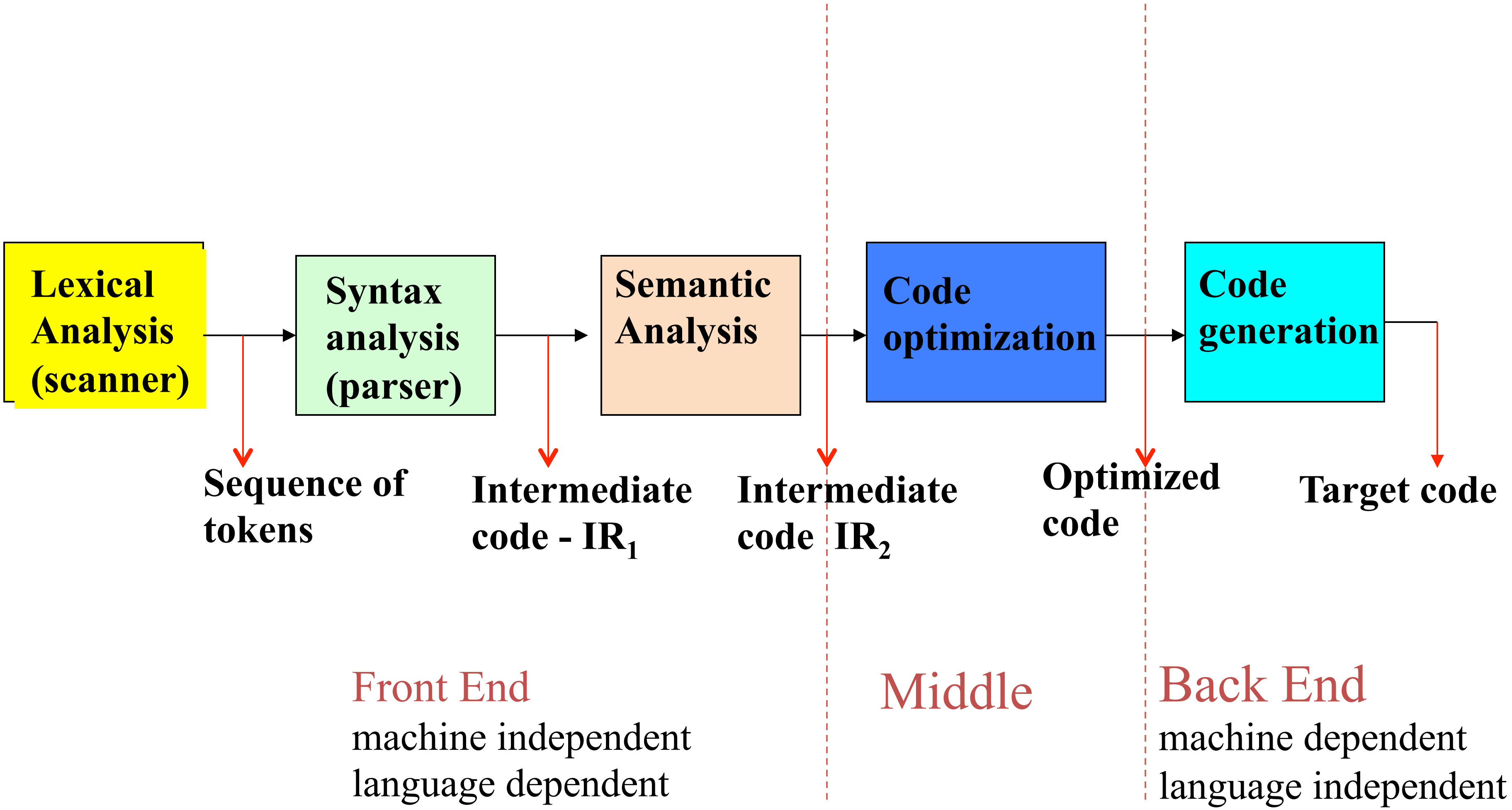
```
temp1 = id3*60
id1 = id2 + temp1
```

**To::**

```
load id3, R0
mult 60.0, R0
load id2, R1
add R1, R0
store R0, R1
```

**Terminology:** code selection, storage management, register allocation and assignment

# Summary:
# Conceptual phases of compiler



Front End
machine independent
language dependent

Middle

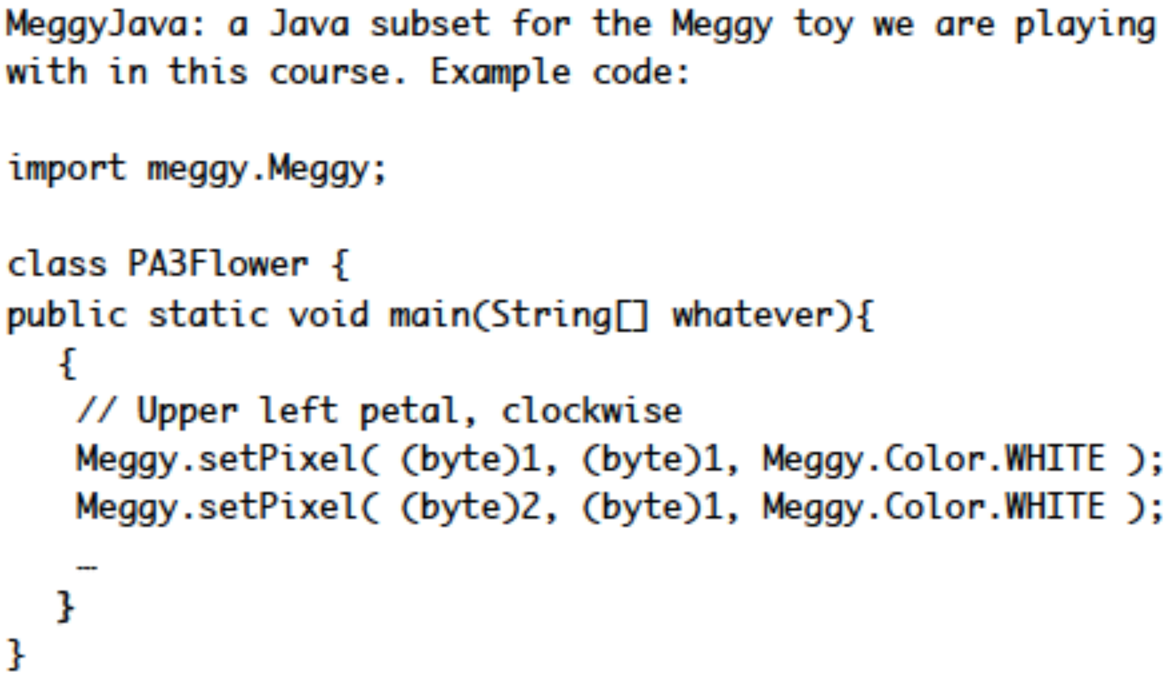
Back End
machine dependent
language independent

# Two More Components

**Error Checking**

**Symbol Table Management**

# The MeggyJava Project



MeggyJava: a Java subset for the Meggy toy we are playing with in this course. Example code:

```
import meggy.Meggy;

class PA3Flower {
public static void main(String[] whatever){
   {
    // Upper left petal, clockwise
    Meggy.setPixel( (byte)1, (byte)1, Meggy.Color.WHITE );
    Meggy.setPixel( (byte)2, (byte)1, Meggy.Color.WHITE );
    …
   }
}
```

# The MeggyJava Project

**PA1:** MeggyJava test suite, AVR programming

**PA2:** scanner, grammar, syntax-directed code generation

**PA3:** parser to construct AST and perform type checking
add control structures, expressions

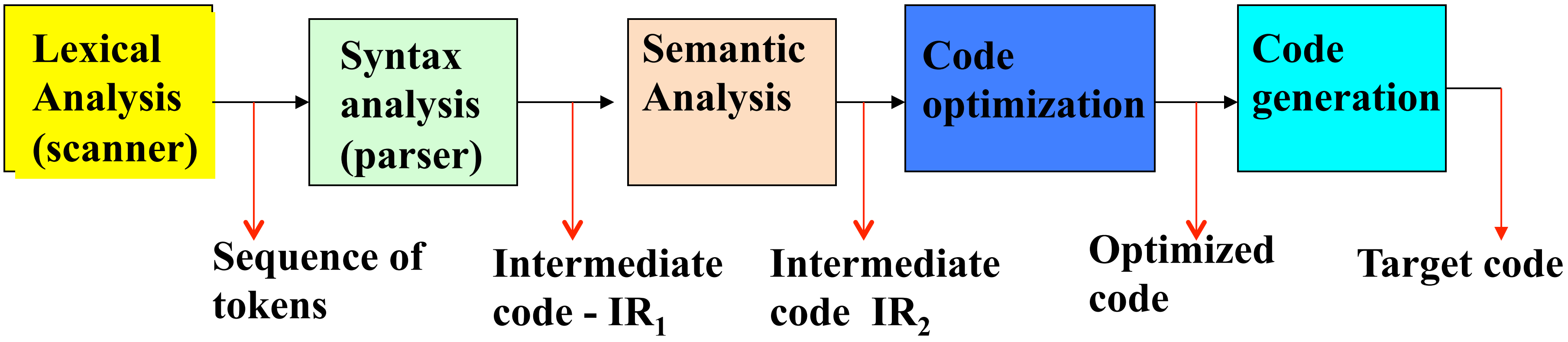
**PA4:** symbol table, add methods, tones, less than

**PA5:** storage management,
add classes, local vars, arrays, assignment

**PA6:** register allocation

Let's Draw some pictures of this!

# So, what information is needed to build a compiler?



# What CS Concepts are used to build compiler phases?

# What tools are available to help build the compiler?

Scanner generators:

Lex,  Flex, Jlex

Parser generators:

YACC, Bison, JavaCUP