Intro to compilers

Based on end of Ch. 1 and start of Ch. 2 of textbook, plus a few additional references

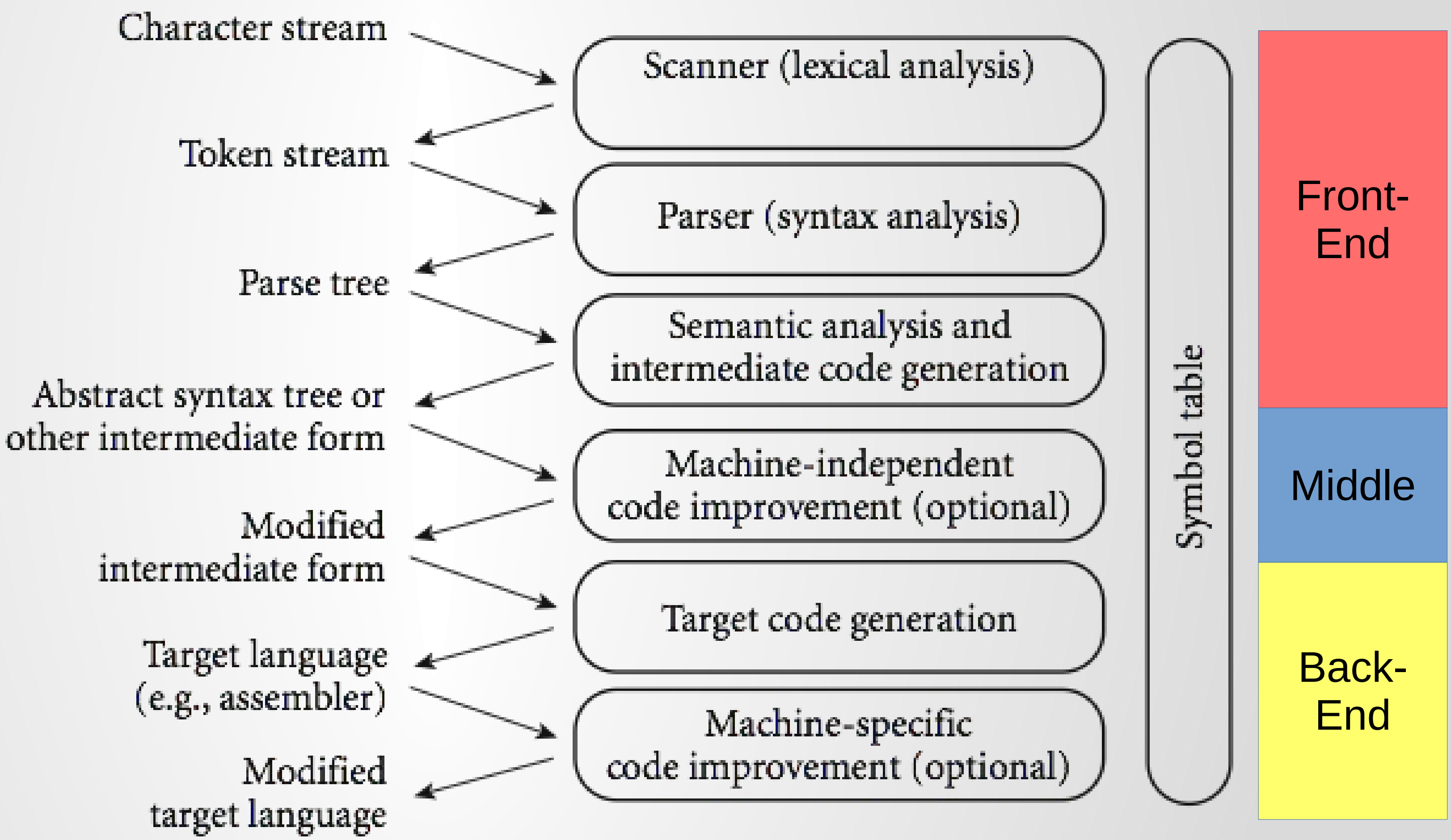
Compilation

- The process by which programming languages are turned into assembly or machine code
  - Take a moment to think about how cool this is

Compilers are translators, must understand language:

  - Language tokens (lexical analysis/scanning)
  - Grammar (syntactic analysis/parsing)
  - Meaning (semantic analysis)
- Output: assembly, machine code, or some intermediate language with same semantics
  - (i.e. Java ->Bytecode)

Compilation phases



Compiler phases

- The first 3 phases are known as the “front end”, where the goal is to figure out the meaning of the program
- Middle phase does machine-independent optimizations on an intermediate representation
- The last 2 are the “back end”, and are used to construct an equivalent target program in the output language

These are split to make things independent:

- The middle and back end can be shared between different source languages
- If you want to write a new language, you only need to specify how the language should be interpreted

Phase 1: lexing/scanning

- Divides the program into "tokens", which are the smallest meaningful units; this saves time, since character-by-character processing is slow
- We can tune the scanner better if its job is simple; it also saves complexity (lots of it) for later stages
- You can design a parser to take characters instead of tokens as input, but it isn't pretty
- Typically, scanning is recognition of a *regular language*, via a deterministic finite automata (DFA)

Lexing Example

Input: A sequence of characters,

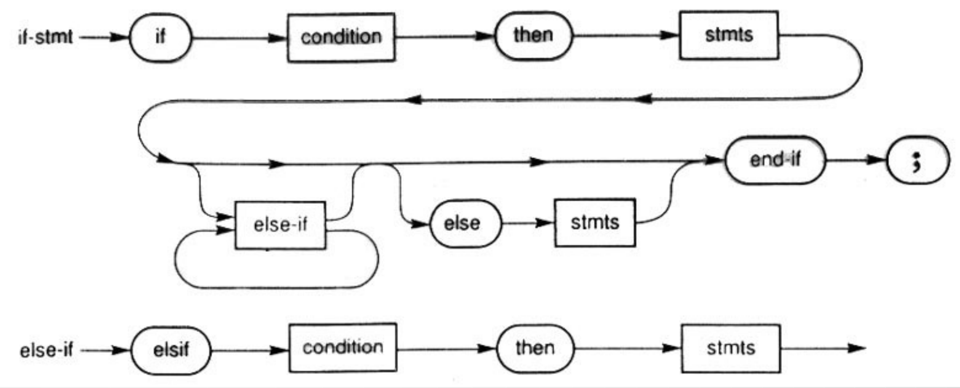
Output: A stream of tokens with types:

```
if ( x > 20 ) {
   z = x + 3.14;
}
```

```
IF LPAREN ID:x GREATER INT:20 RPAREN LBRACE
   ID:z ASSIGNMENT ID:x PLUS FLOAT:3.14
RBRACE
```

Phase 2: parsing

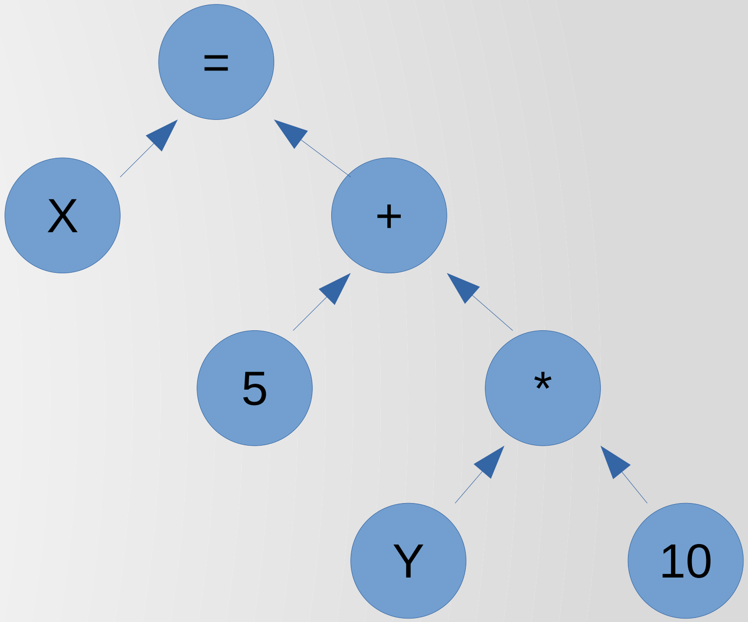
- ***Parsing*** is recognition of a *context-free language*, done via something called a push down automata (or PDA)
  - Parsing discovers the "context free" structure of the program
  - Informally, it finds the structure you can describe with syntax diagrams



Parsing Example

Input: Token stream

Output: Parse Tree



x = 5 + (y*10)

x = 5 + + 42 15 z

(syntax error)

Phase 3: semantic analysis

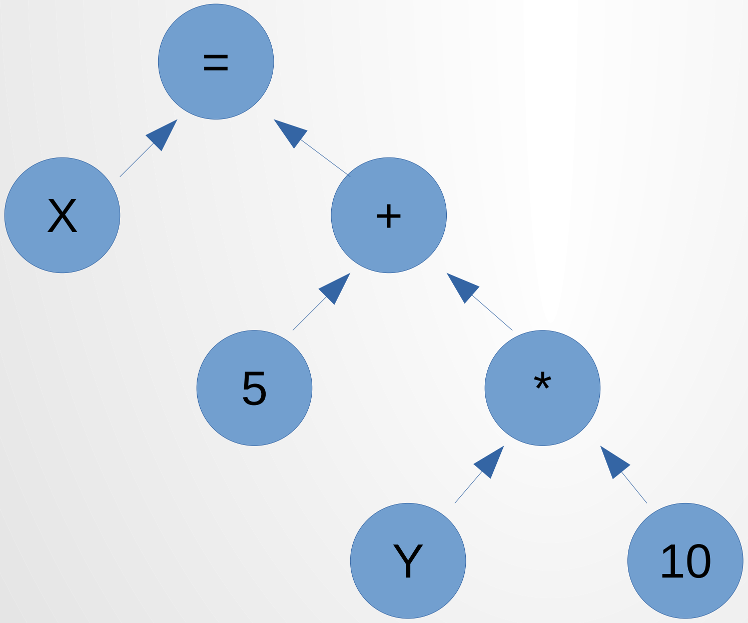
- ***Semantic analysis*** is the discovery of *meaning* in the program
  - The compiler actually does what is called STATIC semantic analysis. That's the meaning that can be figured out at compile time
  - Some things (e.g., array subscript out of bounds) can't be figured out until run time. Things like that are part of the program's DYNAMIC semantics

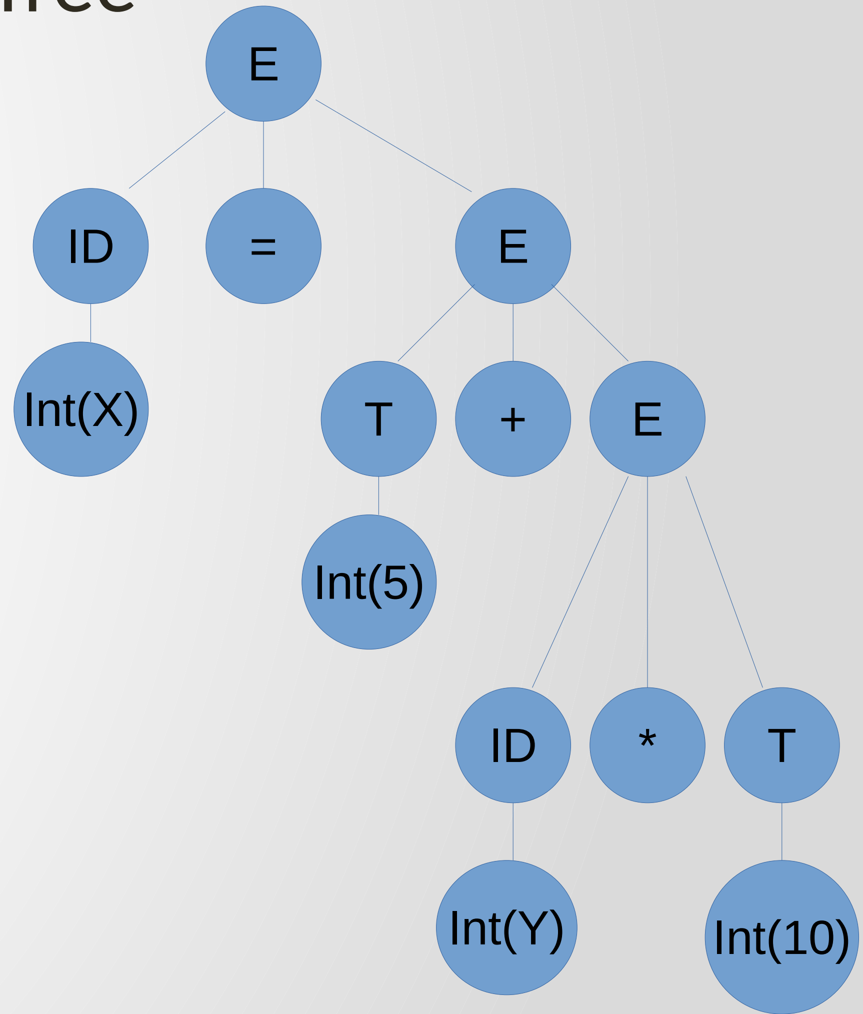
Semantic Analysis Example

Input: Parse Tree

Output: Abstract Syntax Tree

(E = expression,
 T = terminal)





Middle Phase: Machine-Independent Optimizations

- An ***Intermediate form*** (IF) is created after semantic analysis (*if* all checks pass)
  - IFs are often chosen for machine independence, ease of optimization, or compactness
    - Note: these are somewhat contradictory!
  - They often resemble machine code for some imaginary idealized machine; e.g. a stack machine, or a machine with arbitrarily many registers
  - Many compilers actually move the code through more than one IF

Machine-Independent Optimization Example

- Liveness analysis / dead code removal

```
never_used = big_computation()
if( never_happens ) {
    x = y + z;
}
```

- Common subexpression elimination

```
a = x + y;
b = x + y;
```

- Constant elimination

```
x = 42 + 404;
x = 446;
```

Machine-Independent Optimization Example

- Loop Optimizations – e.g. loop unrolling

```
for ( int i = 0; i < 5; i++ ){
   a[i] = i;
}
```

Versus:

```
i = 0;
a[i] = i;
i++;
a[i] = i;
i++;
a[i] = i;
i++;
a[i] = i;
i++;
a[i] = I;
```

Bottom phases

- ***Code generation phase*** produces assembly language or (sometime) machine language

Bottom phases (cont)

- Certain ***machine-specific optimizations*** (use of special instructions or addressing modes, etc.) may be performed during or after ***target code generation***
- ***Symbol table***: all phases rely on a symbol table that keeps track of all the identifiers in the program and what the compiler knows about them
  - This symbol table may be retained (in some form) for use by a debugger, even after compilation has completed

An Overview of Compilation

- Lexical and Syntax Analysis: back to our GCD Program (in C)

```
int main() {
  int i = getint(), j = getint();
  while (i != j) {
    if (i > j) i = i - j;
    else j = j - i;
  }
putint(i);
}
```

An Overview of Compilation

- Lexical and Syntax Analysis
  - GCD Program Tokens
    - Scanning (*lexical analysis*) groups characters into *tokens*, the smallest meaningful units of the program

```
int     main   (   )      {
int     i      =   getint (   )   ,   j   =   getint   (   )   ;
while   (      i   !=     j   )   {
if      (      i   >      j   )   i   =   i   -        j   ;
else    j      =   j      -   i   ;
}
putint  (      i   )      ;
}
```

An Overview of Compilation

- Context-Free Grammar and Parsing
  - Parsing organizes tokens into a *parse tree* that represents higher-level constructs in terms of their constituents
  - Potentially recursive rules known as a *context-free grammar* define the ways in which these tokens can combine

An Overview of Compilation

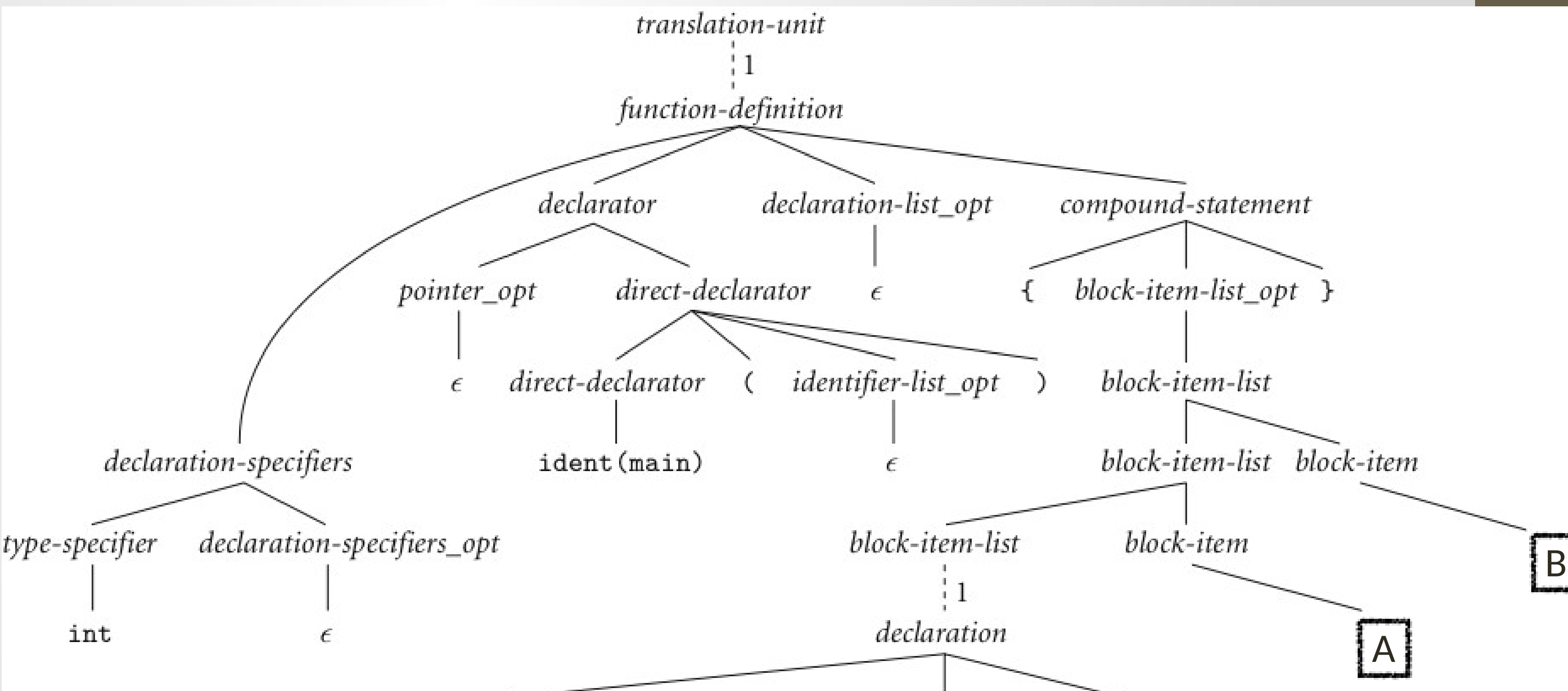
- Context-Free Grammar and Parsing
  - Example (`while` loop in C)

iteration-statement → *while (expression) statement*

statement, in turn, is often a list enclosed in braces:
statement → *compound-statement*
compound-statement → *{ block-item-list opt }*
where
block-item-list opt → *block-item-list*
or
block-item-list opt → ϵ
and
block-item-list → *block-item*
block-item-list → *block-item-list block-item*
block-item → *declaration*
block-item → *statement*

An Overview of Compilation

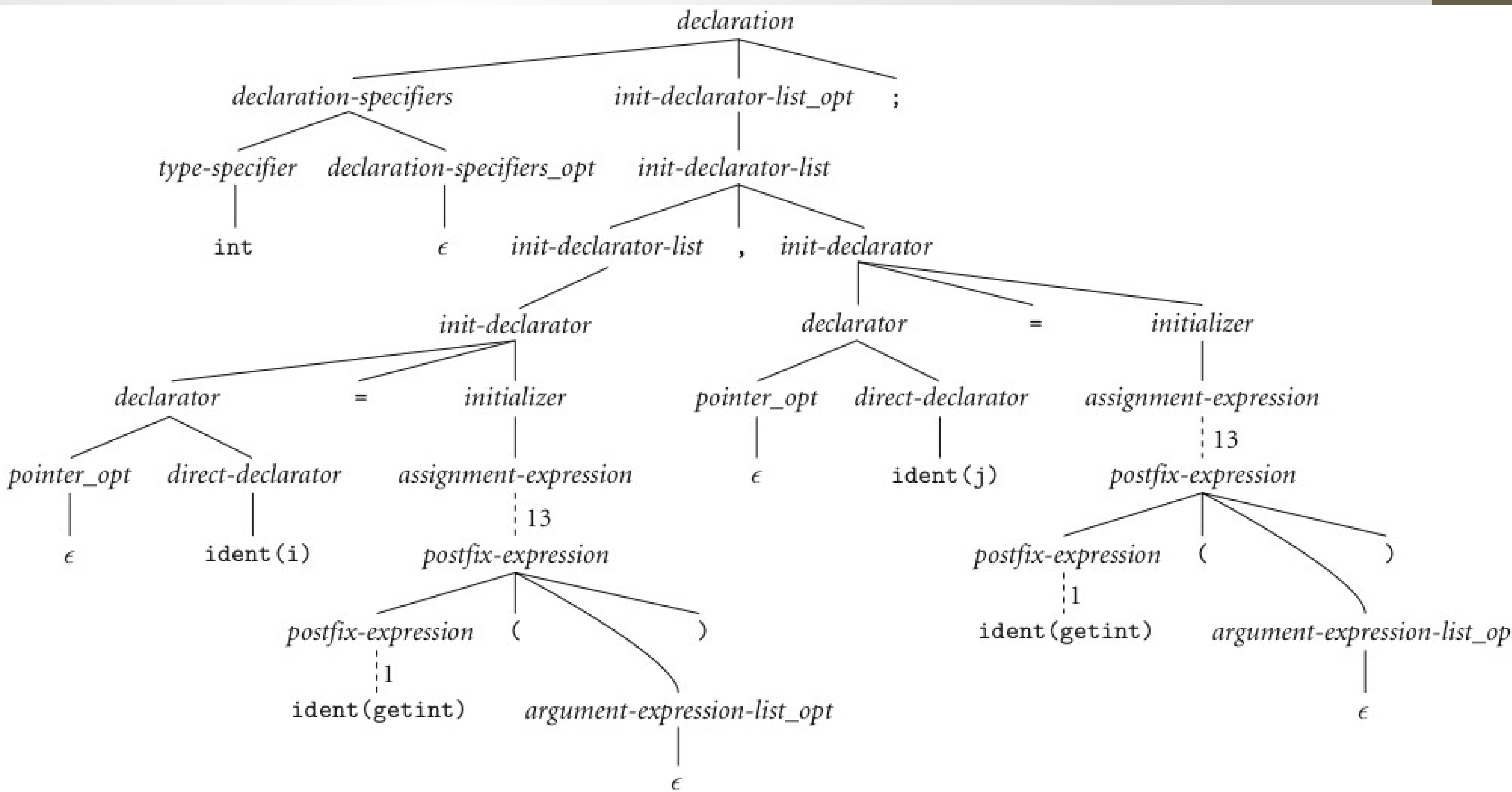
- Example: Our GCD Program Parse Tree



next slide

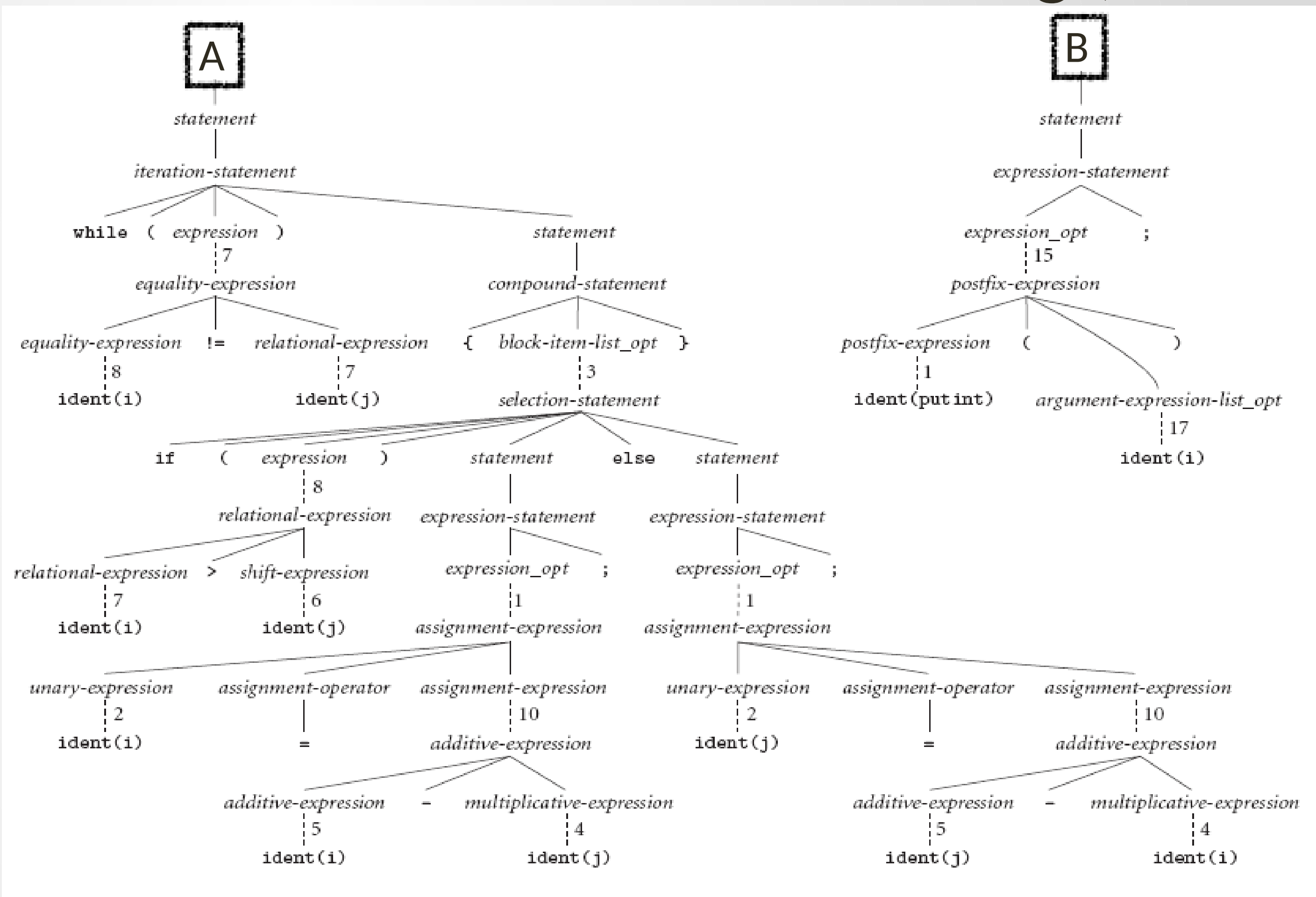
An Overview of Compilation

- Context-Free Grammar and Parsing (continued)



An Overview of Compilation

- Context-Free Grammar and Parsing (continued)



Ch. 2 – a deeper look

- We’ll take a deeper look at scanning and parsing, the two parts of the “front end” of this process
- Each has deeper ties to theoretical models of computation, and useful concepts like regular expressions
  - You may have seen these if you’ve done string manipulations.

Regular expressions

- A regular expression is defined (recursively) as:
  - A character
  - The empty string, ε
  - 2 regular expressions concatenated
  - 2 regular expressions connected by an “or”, usually written x | y
  - 0 or more copies of a regular expression – written *, and called the Kleene star

Regular languages

- Regular languages are then the class of languages which can be described by a regular expression
- Example: L1 = 0^*10^*
- Another: L2 = $(1|0)^*$

More regular languages

- Exercise: Give the regular expression for the language of binary strings that begin with a 0 and end with a 1
- Exercise (a bit harder): Give the regular expression for the language of binary strings that start with a 0 and have odd length

A more realistic example

- Unsigned integers in Pascal:
  - Examples: 4, or 82.3, or 5.23e-26
- Formally:

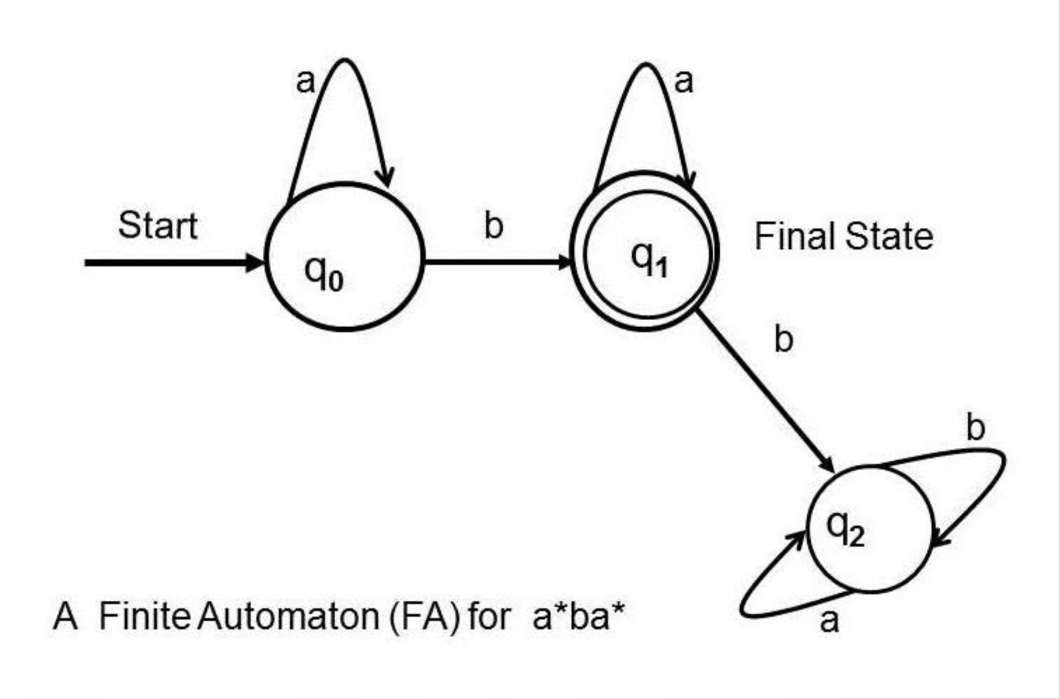
$$
\begin{aligned}
\textit{digit} &\longrightarrow \mathtt{0} \mid \mathtt{1} \mid \mathtt{2} \mid \mathtt{3} \mid \mathtt{4} \mid \mathtt{5} \mid \mathtt{6} \mid \mathtt{7} \mid \mathtt{8} \mid \mathtt{9} \\
\textit{unsigned\_integer} &\longrightarrow \textit{digit}\ \textit{digit}\,^{*} \\
\textit{unsigned\_number} &\longrightarrow \textit{unsigned\_integer}\ ((\ \mathtt{.}\ \ \textit{unsigned\_integer}) \mid \epsilon\,) \\
&\qquad (((\,\mathtt{e} \mid \mathtt{E}\,)\ (\,\mathtt{+} \mid \mathtt{-} \mid \epsilon\,)\ \textit{unsigned\_integer}) \mid \epsilon\,)
\end{aligned}
$$

Another view: DFAs

- Regular languages are also precisely the set of strings that can be accepted by a deterministic finite automata (DFA)
- Formally, a DFA is:
  - a set of states
  - an input alphabet
  - a start state
  - a set of accept states
  - a transition function: given a state and input, outputs another state

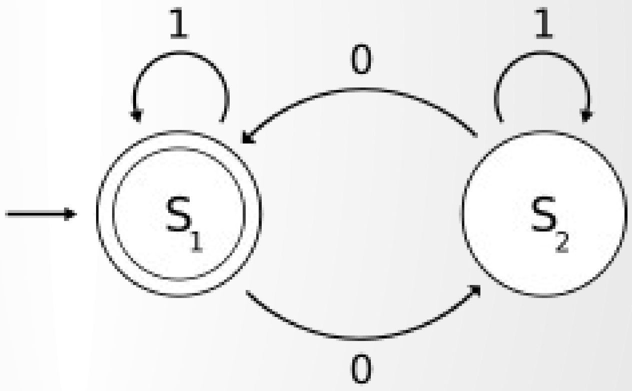
DFAs

- More often, we'll just draw a picture (like in graph theory)
- Example:



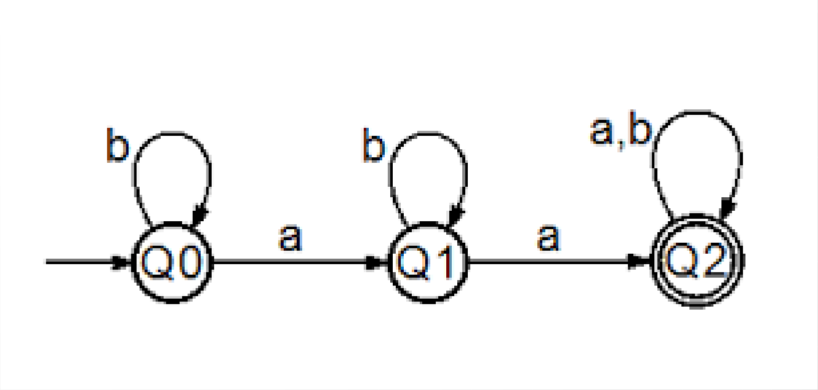
DFAs

- What regular language does the following DFA accept?



DFA examples

- What’s the DFA for the regular language: 1(0|1)*0
- What’s the regular language accepted by this DFA?



Scanning

- Recall scanner is responsible for
  - tokenizing source
  - removing comments
  - (often) dealing with *pragmas* (i.e., significant comments)
  - saving text of identifiers, numbers, strings
  - saving source locations (file, line, column) for error messages

Scanning

- Suppose we are building an ad-hoc (hand-written) scanner for Pascal:
  - We read the characters one at a time with look-ahead
- If it is one of the one-character tokens
  `{ ( ) [ ] < > , ; = + - etc }`
  we announce that token
- If it is a ., we look at the next character
  - If that is a dot, we announce .
  - Otherwise, we announce . and reuse the look-ahead

Scanning

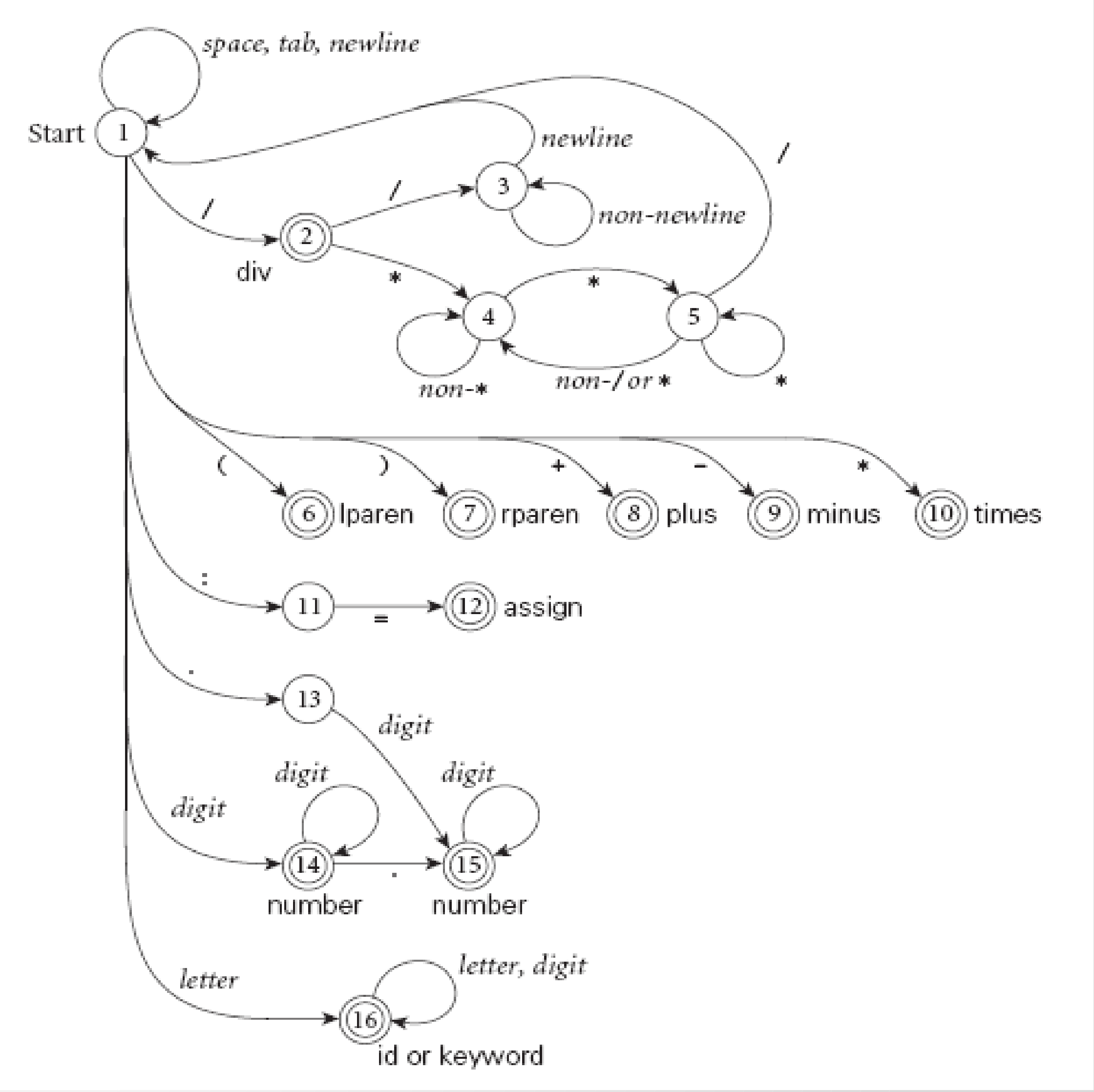
- If it is a <, we look at the next character
  - if that is a = we announce <=
  - otherwise, we announce < and reuse the look-ahead, etc
- If it is a letter, we keep reading letters and digits and maybe underscores until we can't anymore
  - then we check to see if it is a reserve word

Scanning

- If it is a digit, we keep reading until we find a non-digit
  - if that is not a . we announce an integer
  - otherwise, we keep looking for a real number
  - if the character after the . is not a digit we announce an integer and reuse the . and the look-ahead

Scanning

- Pictorial representation of a scanner for calculator tokens, in the form of a finite automaton



Coding DFAs (scanners)

- That’s all well and good – but how to we program this stuff?
  - A bunch of if/switch/case statements
  - A table and driver (flex or other tools)
- Both have merits, and are described further in the book.
- We’ll mainly use the second route in homework, simply because there are many good tools out there.

Scanners

- Writing a pure DFA as a set of nested case statements is a surprisingly useful programming technique
  - though it's often easier to use perl, awk, sed
  - for details see Figure 2.4 in text
- Table-driven DFA is what lex and scangen produce
  - lex (flex) in the form of C code – this will be an upcoming homework
  - scangen in the form of numeric tables and a separate driver (for details see Figure 2.12)

Next week

- We’ll see a bit more about DFAs, and introduce NFAs.
  - This is the rest of section 2.2, if you want to look ahead a bit.
- By the end of the week, we’ll move to discussing one table-driven DFA, flex, and have the first programming assignment over it the week after.