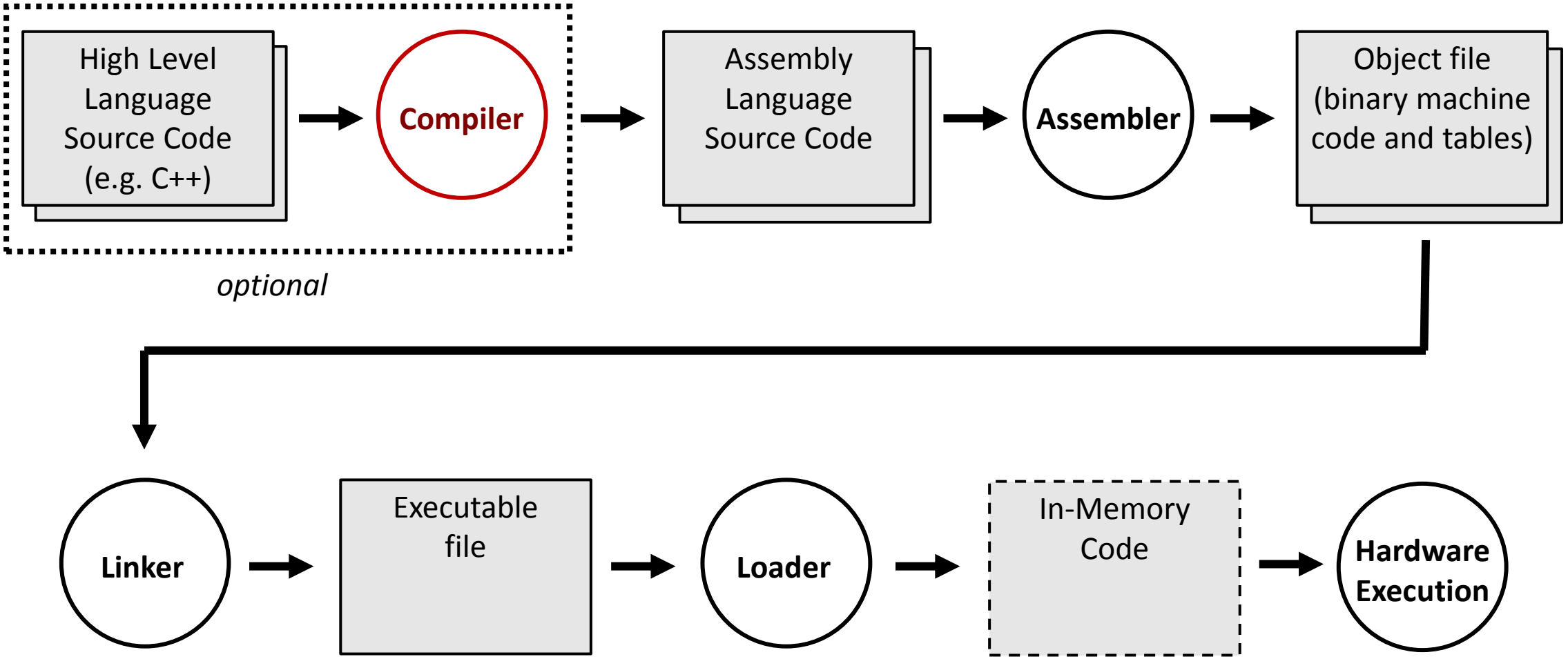
CMPE 220

Class 8 – Compilers (part 1)

Building Software



High Level Languages

- Compilers convert “high level languages” to assembly language
- The original goals of high level languages were two-fold:
  - Make it easier to write and to read programs
  - Write portable programs that could run on any hardware
- This latter goal meant that the language had to be independent of the machine architecture… it had to provide a level of *abstraction* that hid the details of registers, opcodes, and even memory architecture.
- *Computers had to have memory and processing speed to spare!*
- Early languages had very rigid formats and few “features” – but languages evolved, experimenting with new ways of programming (and making the task of writing compilers progressively harder).

21 Important Programming Languages

- My list – others may differ
- The purpose of reviewing these languages is to:
  - get a sense of how languages evolved and continue to evolve
  - understand some of the key features of programming languages today, and how they affect compilers and code generation
- As we go through the list, keep track of:
  - How many of these languages you have heard of
  - How many of these languages you have used

A Few Important Languages – the Early Years

- A (or A-0) – 1952 – created by Grace Hopper, and often considered the first high-level language
- FORTRAN (FORmula TRANslation) - 1957 – developed at IBM - focus on math and science programming
- LISP (LISt Processor) – 1958 – built in support for "list" data structures – emphasis on mathematical programming
- ALGOL (ALGOrithmic Language) – 1958 – in many ways, ahead of its time – a very "structured" language, used in teaching for decades
- RPG (Report Program Generator) – 1959 – a *non-procedural language* that allowed users to describe/specify the desired output, and how it was produced.

More Important Languages

- COBOL (COmmon Business Oriented Language) – 1960 – focus on business programming – user friendly syntax and words – the first portable language
- BASIC (Beginner's All-Purpose Symbolic Instruction Code) – 1964 – created as an easy-to-use teaching language, which became popular due to its simplicity. Often implemented as an *interpreter*.
- B – 1969 – created at Bell Labs; an ALGOL-like language best known as the precursor to C
- PASCAL – 1970 – created by Nicklaus Wirth as a highly structured instructional language; quickly replaced ALGOL as the teaching language of choice
- C – 1972 – developed by Dennis Ritchie and Brian Kernigan at Bell Labs. A powerful (and dangerous!) language frequently used for OS programming.

More Important Languages

- Smalltalk – 1972 – a dynamically typed, object-oriented programming language, developed at Xerox Palo Alto Research Center (PARC). Slow to catch on, Smalltalk represented a paradigm shift in programming languages.
- SQL (Structured Query Language) – 1975 – developed at IBM, based on Edgar Codd's relational database model
- C++ - 1980 – added object-oriented programming constructs – and a lot of other things – to C. Very powerful, but often derided by computer language experts as overly complex and error prone.
- MATLAB (MATrix LABoratory) – 1984 – a multi-paradigm numerical computing language and toolkit – an early specialized language
- Perl – 1987 – designed for text processing; still frequently used for scripting on POSIX systems
- Python – 1991 – dynamically-typed, garbage-collected language with an emphasis on readability

More Important Languages – the Web Years

- HTML (Hyper-Text Markup Language) – 1989 – a non-procedural text formatting language that formed the basis of the World Wide Web.
- Ruby – 1995 - dynamically typed and uses garbage collection. Supports multiple programming paradigms, including procedural, object-oriented, and functional programming. Originally interpreted.
- Java – 1995 – a garbage-collected, object-oriented language, created by James Gosling at Sun. Widely used in teaching, and in commercial software development.
- PHP (Personal Home Page) – 1995 – a dynamically typed language with built-in support for structures. The most widely used language for web programming. Usually interpreted, though *PHP compilers* may be used for high-demand applications.
- Javascript – 1995 – a lightweight, dynamically typed, interpreted language embedded in web browsers (and other devices)

A Timeline of Language Advances

Language	Date
A (or A-0)	1952
FORTRAN (FORMula TRANslator)	1957
LISP (LISt Processor)	1958
ALGOL (ALGOrithmic Language)	1958
RPG (Report Program Generator)	1959
COBOL (COmmon Business Oriented Language)	1960
BASIC (Beginner's All-purpose Symbolic Instruction Code)	1964
B	1969
SQL (SEQUEL)	1975

Language	Date
PASCAL	1970
C	1972
Smalltalk	1972
C++	1980
MATLAB (MATrix LABoratory)	1984
HTML	1989
PYTHON	1991
Ruby	1995
Java	1995
PHP	1995
Javascript	1995

That’s 21 Influential Languages

- I believe these languages are important because:
  - Many are still widely used today
  - They introduced or popularized new ways of programming
  - They influenced later languages
- There are a host of new languages since 2000, but none (in my opinion) have broken out into the mainstream as truly influential
- **How many have you heard of or used?**

How Many Languages Have You Used?

- My totals:
  - Heard of 21 (obviously, since it's my list!)
  - Used 13: FORTRAN, ALGOL, COBOL, BASIC, SQL, PASCAL, C, C++, HTML, Python, Java, PHP, JavaScript (plus dozens of others)

Why Are There No Famous Assemblers?

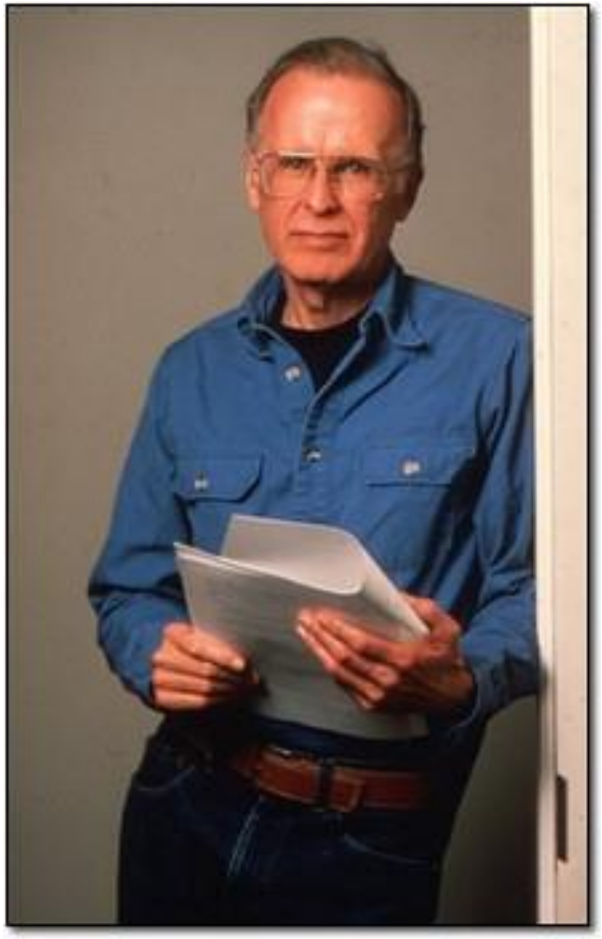
- Assembly languages are, by their nature, tied to a specific machine architecture
- No assembly language could become ubiquitous…. it could not spread beyond the users of a particular computer
- Assembly languages change as processors evolve… most don’t “persist” more than a few years

History

- Kathleen Booth is credited with creating the first *assembler* in 1947, while working on the ARC2 (Automatic Relay Calculator) computer at the University of London.
- The first high level language *compiler* – for a primitive language called A-0 – was developed by Grace Hopper in 1952. The A-0 system combined features of what we would call a compiler, assembler, linker and loader.
  - Several versions followed: A-1, A-2, ARITH-MATIC, FLOW-MATIC

History - FORTRAN

- The first FORTRAN (FORmula TRANslation) compiler was developed at IBM in 1957, by a team led by John Backus
  - Backus pioneered *parsing* techniques based on a formal set of *syntax rules* (a *grammar*)
- FORTRAN was the first *commercial* compiler – it was an extra cost add-on when you bought an IBM computer
- It is sometimes called the first modern compiler, because it employed formal grammar rules



History - COBOL

- The first COBOL (Common Business Oriented Language) compiler was released in 1960.
- COBOL was designed by an industry-wide committee – proposed by Mary Hawes, a Burroughs Corporation programmer
- It was the first “standardized” programming language
- It was heavily influenced by Grace Hopper, who advised the committee
- It was the first language to run on multiple computers (the UNIVAC II and the RCA 501) – thus achieving the goal of *portability*

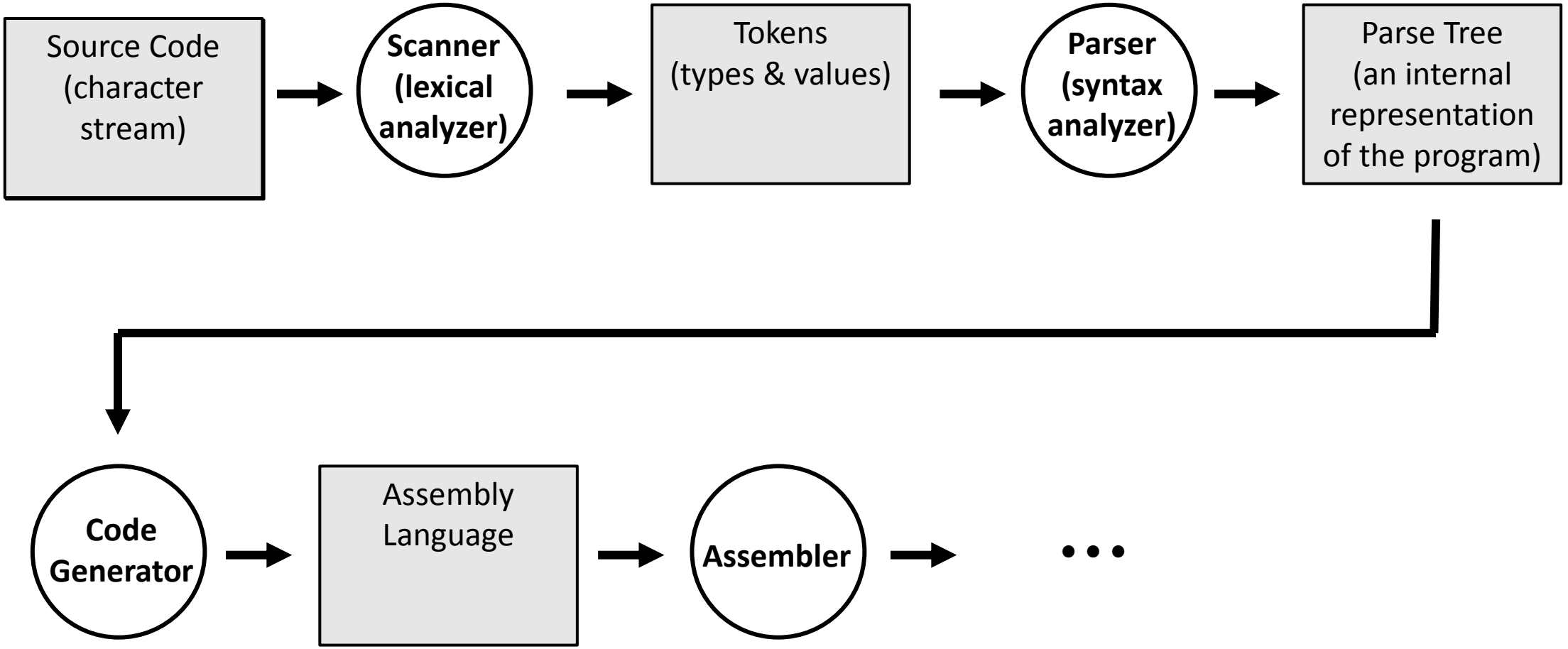
New Challenges

- Assemblers are easy to write.
  - No special techniques are needed
  - Anyone with a few years of programming experience should be able to write an assembler
  - The earliest assemblers were written using “brute force” programming
- Compilers raised the bar
  - It is *almost impossible* to write a compiler using “brute force” methods
  - New programming concepts and techniques were needed – and that’s what we’ll be talking about this week and next

Assemblers versus Compilers

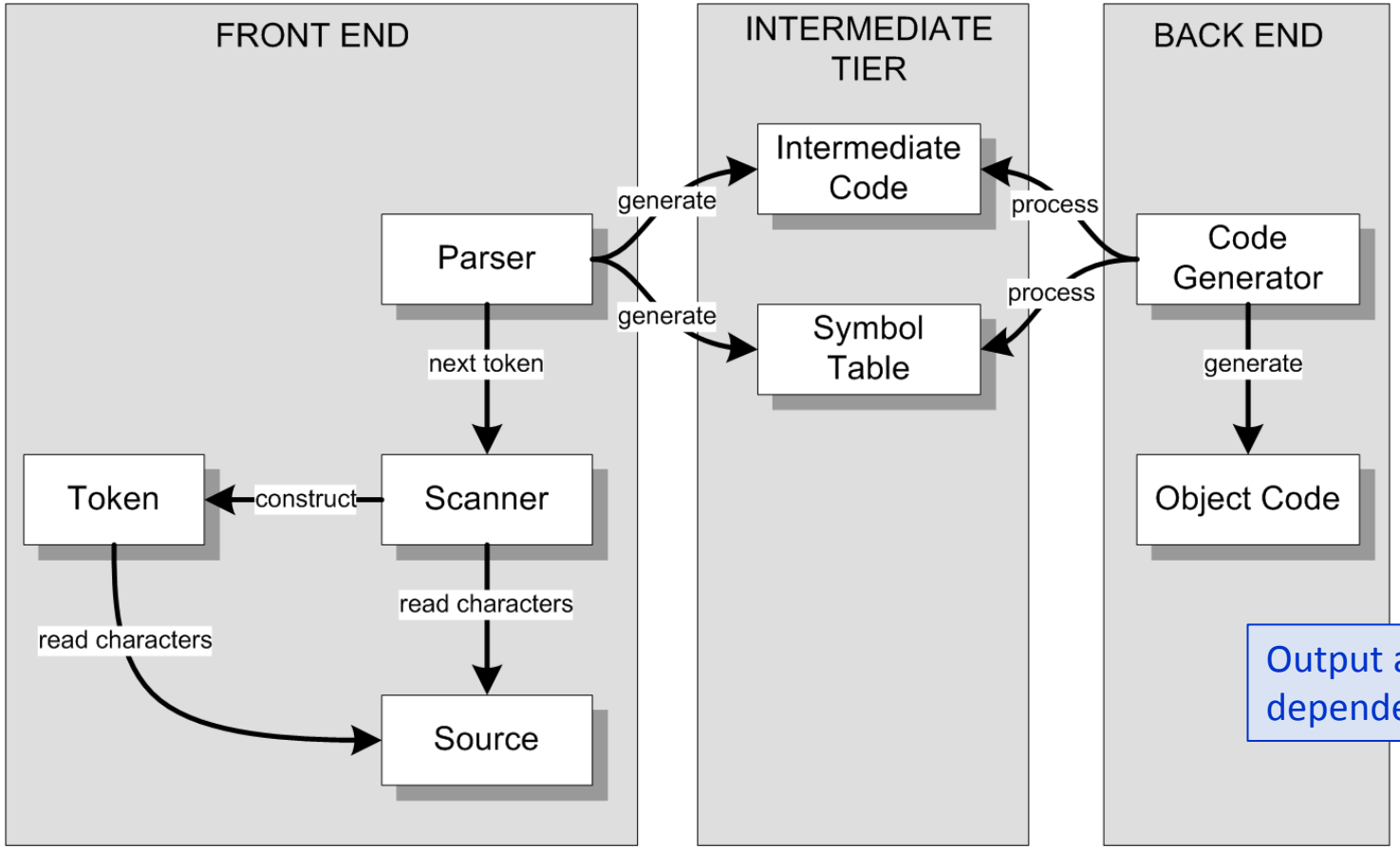
Assembler	Compiler
• **Scanline():** look for label, opcode, operands, comments	• **Scanner:** also called a lexical analyzer or tokenizer. Breaks the input into a series of *tokens*
• **Processline():** indentify the opcode, process operands, etc.	• **Parser:** a semantic analyzer, that recognizes the structure of the program based on the token stream, and builds an internal representation of the program
• **Second pass:** Assemble the instructions and generate machine code into an object file	• **Code generator:** process the internal representation and generate assembly-language code for a specific machine
• **Symbol Table:** stores symbols and information about their type and definition	• **Symbol Table:** stores symbols and information about their type and definition

Compiler Components



Conceptual Design

- We can architect a compiler with three major parts:



Output a low-level, system dependent assembly language

Scanning

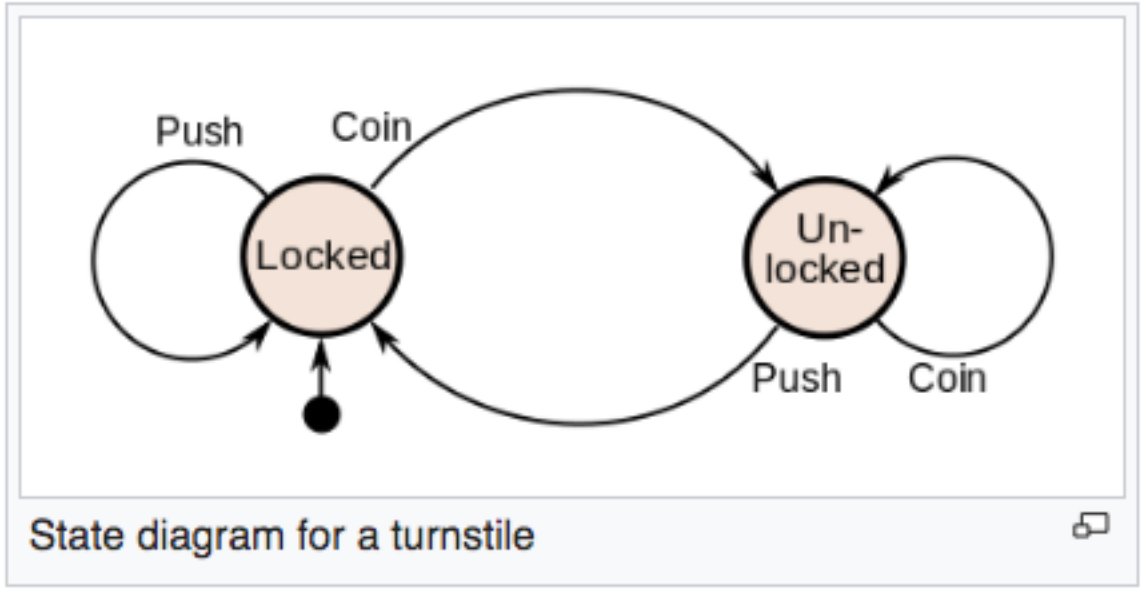
- A scanner converts the input stream into a series of *tokens*, or lexical components that are part of the language:
  label keyword ( ) + / ; { }
- Scanning typical uses a *finite state machine* or *finite state automaton* driven by an underlying state table.
- The finite state machine is an efficient way to recognize a pre-defined lexicon of tokens in an input stream – and to report an error when a token cannot be recognized

Finite State Machines

- **Finite State Machine (FSM):** an abstract machine that can be in exactly one of a finite number of states at any given time
- The FSM can change from one state to another in response to *inputs*
- The change from one state to another is called a *transition*
- The FSM may define *end states* which represent the completion of a sequence of inputs
- We often use diagrams to represent a finite state machine, because it's easy to visualize and understand

Turnstyle: a Real-Word FSM

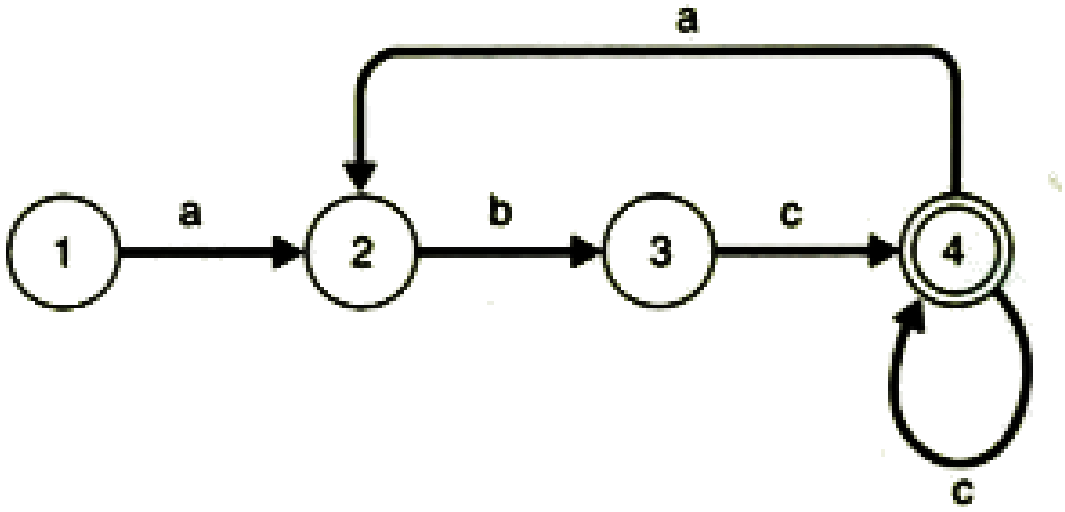
- Two States: *locked* and *un-locked*



State diagram for a turnstile

A Simple FSM

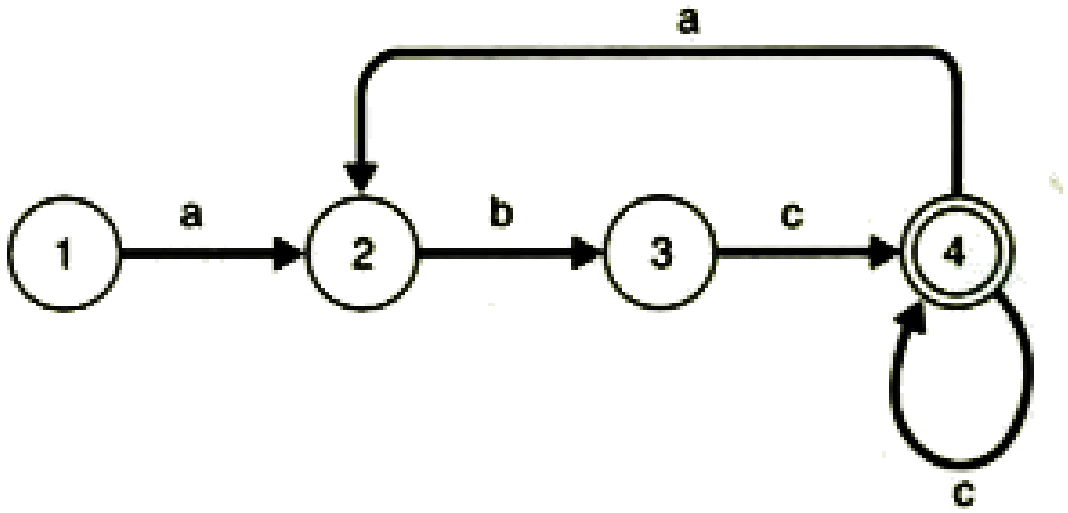
Recognizes a valid sequence of inputs



Input Sequence	Result
abc	VALID: 4 is an end state
abcccabc	VALID: 4 is an end state
acc	INVALID: error in state 2
ab	INVALID: 3 is not an end state

- 1: start state (aka initial state)
- 4: end state (aka final state)

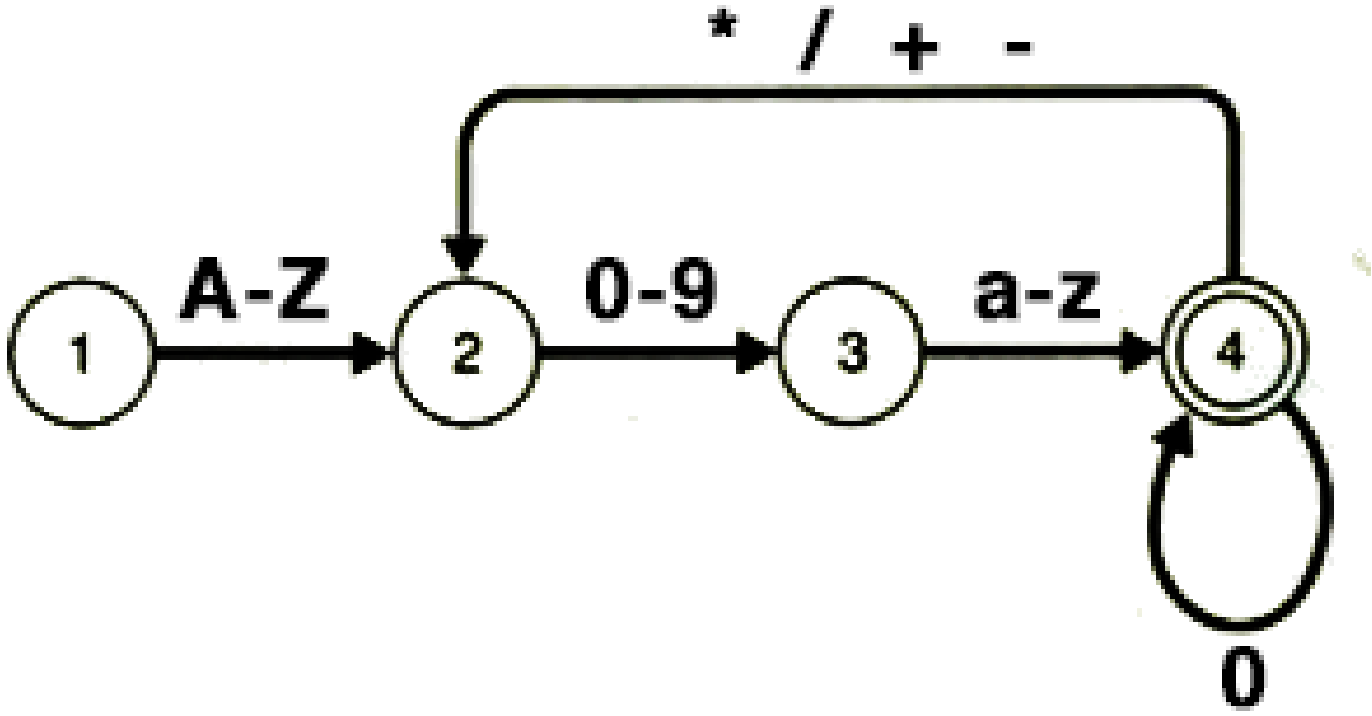
A Simple State Transition Table



STATE	a	b	c
1	2	-	-
2	-	3	-
3	-	-	4
(4)	2	-	4

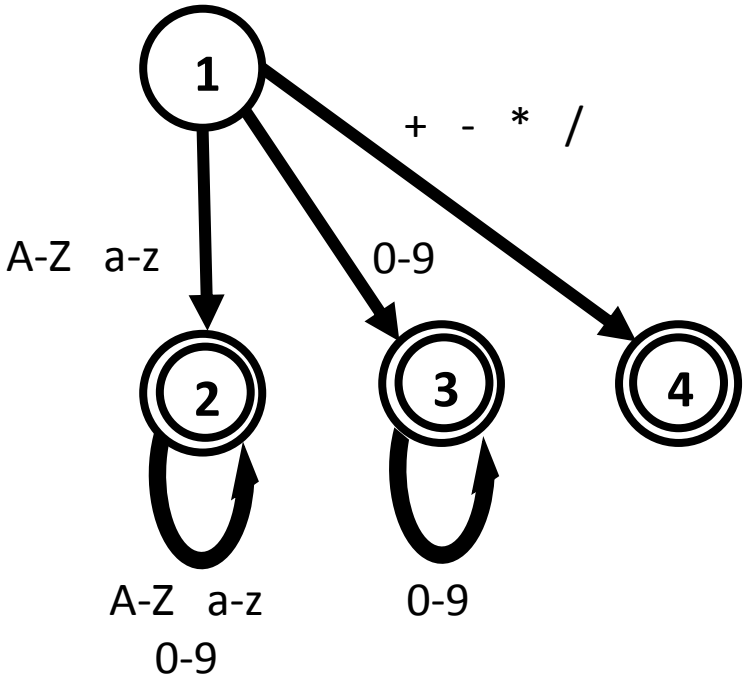
- We can represent a state machine as a *state transition table*
- Note the use of *( )* to indicate an end state
  - At the code level, we might use a negative number, or add a column to the table to flag end states

Additional “Transition” Notation



- A-Z: any uppercase character
- a-z: any lowercase character
- 0-9: any digit
- A *set* of possible characters:
  ** / + -*

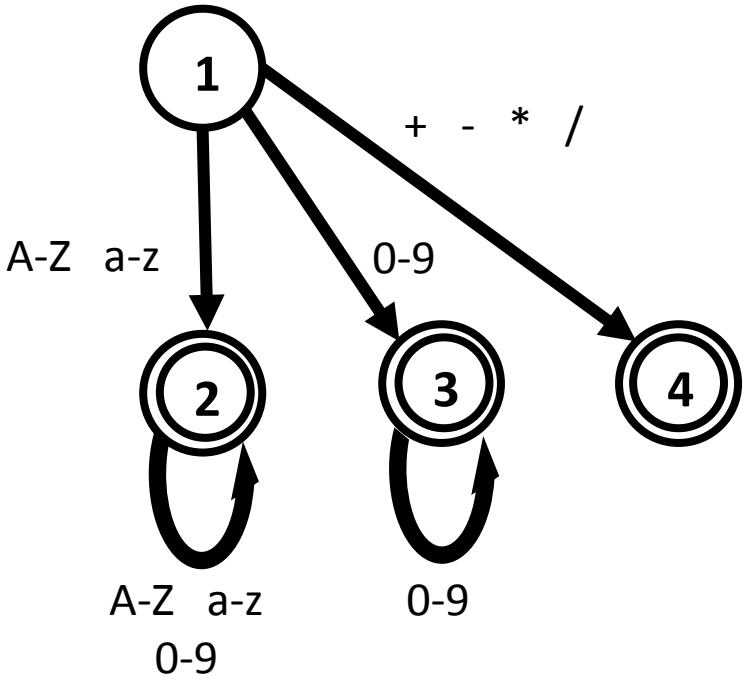
A More Complex (and useful) FSM



STATE	TYPE	A-Z	a-z	0-9	+ - * /
1		2	2	3	4
(2)	Symbol	2	2	2	-
(3)	Integer	-	-	3	-
(4)	Operator	-	-	-	-

- Note that we've added a *token type* to the State Transition Table

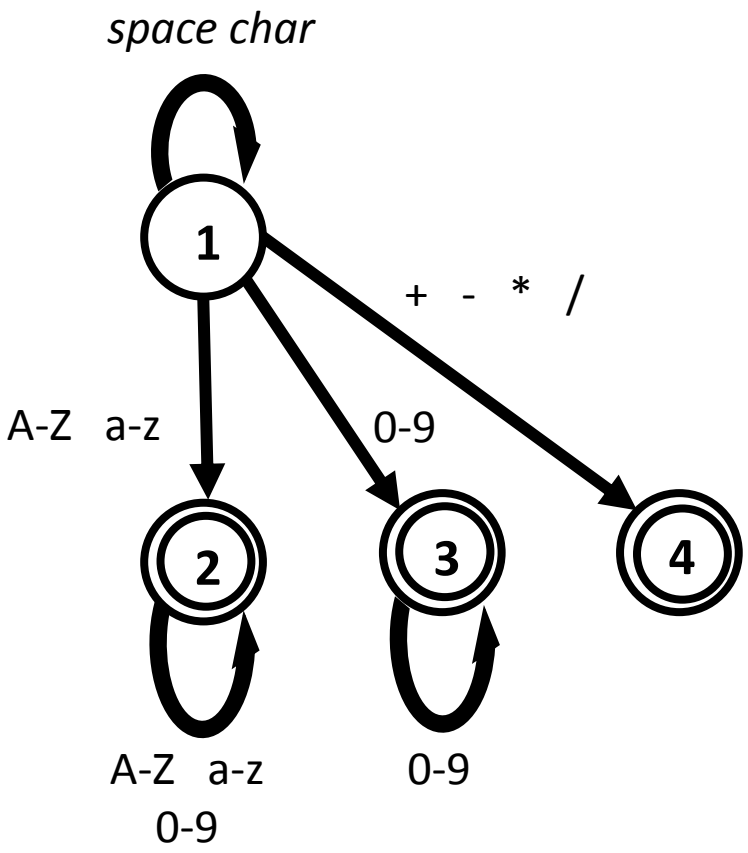
A Scanner / Lexical Analyzer / Tokenizer



- Input = character stream
  Output = token stream
- EXAMPLE: Inventory * 25
  Symbol Operator Integer
- EXAMPLE: Inventory - Sales
  Symbol Operator Symbol
- EXAMPLE: - / Sales Inventory
  Operator Operator Symbol Symbol

- Our FSM always considers ‘space’ characters to be a terminator, but NOT trigger a transition. This is a default rule for many languages.

The Scanner Is Called Repeatedly



- It is used to break the input stream into a sequence of *tokens*
- In <u>most</u> languages, 'space' characters (including tab and end-of-line) can terminate a token, but are otherwise not part of the language lexicon

STATE	TYPE	*space*	A-Z	a-z	0-9	+ - * /
1		1	2	2	3	4
(2)	Symbol	-	2	2	2	-
(3)	Integer	-	-	-	3	-
(4)	Operator	-	-	-	-	-

FSM Function

STATE	TYPE	*space*	A-Z	a-z	0-9	+ - * /
1		1	2	2	3	4
(2)	Symbol	-	2	2	2	-
(3)	Integer	-	-	-	3	-
(4)	Operator	-	-	-	-	-

State Transition Table

Reminder: (n) indicates a valid *end state*

```
state = 1;
value = “”;
error = none;
while ((inchar = getnextchar()) != EOF) {
        nextstate = STT_lookup(state, inchar);
        if (nextstate == ‘-’) {     // no valid transition
                if (state is an endstate) {
                        unget(inchar);    // for next token
                }
                else {
                        error = ”invalid token";
                }
                break;
        }
        state = nextstate;
        value .= inchar;          // appended inchar to value
}
if (error == none)
        return type[state], value;
else
        return error;
```

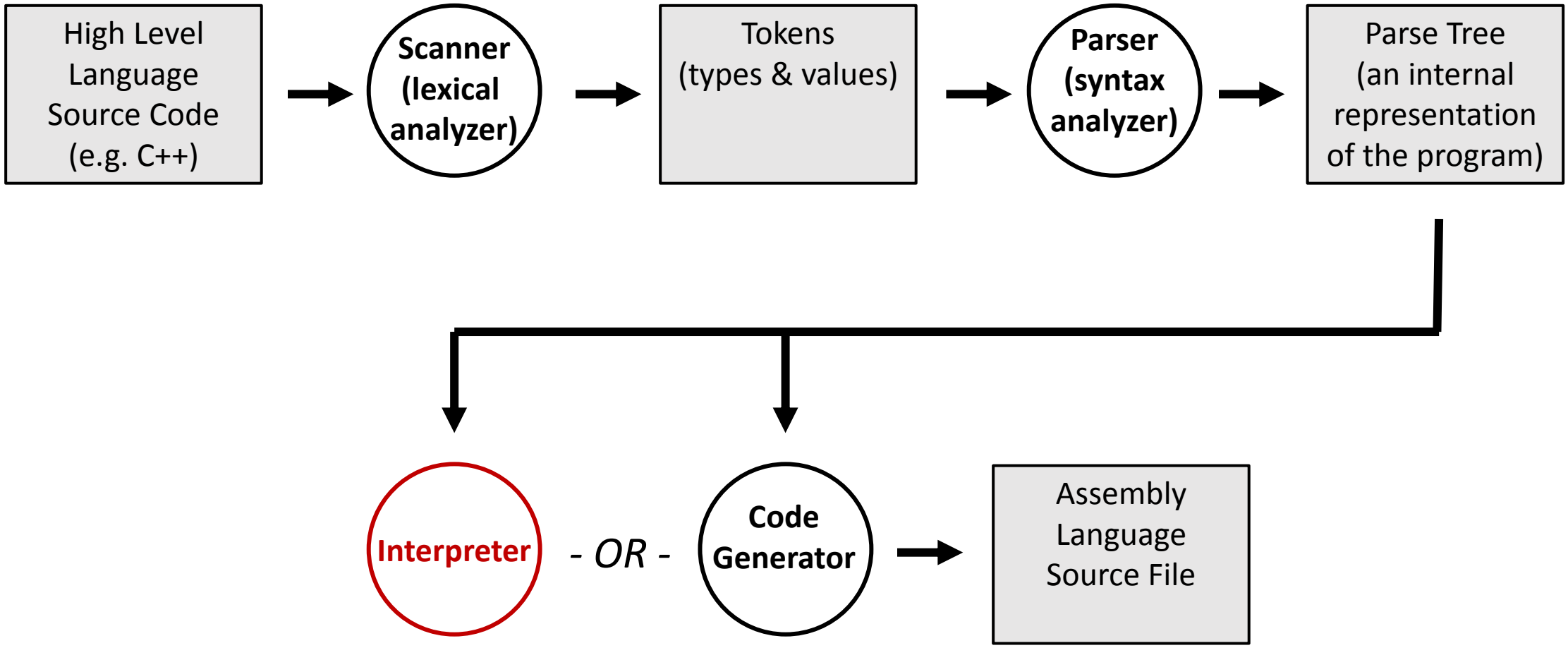
A Scanner is NOT a *Syntax* Analyzer

- The scanner recognizes the language *lexicon*, or vocabulary
- It will (happily) return sequences of tokens that are not valid syntax:
  - + / Sales Inventory
    Operator Operator Symbol Symbol

Token Use: Parsing Versus Code Generation

- In addition to the token type, the scanner returns the actual token values
  - Inventory - Sales
    Symbol (Inventory) Operator (-) Symbol (Sales)
- In order to determine if a string of tokens represents a valid sequence, the *parser* needs to know the token types:
  Symbol Operator Symbol - VALID
- In order to build code for execution, the *code generator* must have the specific values of the tokens:
  Inventory – Sales
- In general, the syntax analyzer (*parser*) is concerned with *token type*, while the code generator is concerned with both *token type* and *token value*

Compiler Components



Recognizing Syntax: Parsing

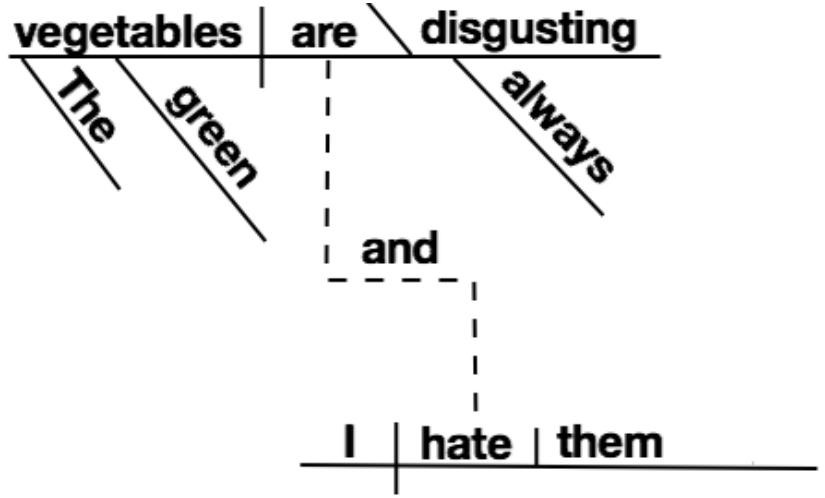
- Languages are made up of a *lexicon* or vocabulary of tokens, which are combined using grammatical rules (syntax) to form meaningful statements
- The *parser* recognizes the *syntax*, or *grammatical structure* of the token stream
- The parser returns an internal representation of the program being compiled. The most common representation is a *parse tree*.

Scanner (lexical analyzer)	Parser (syntax analyzer)
Recognizes lexical elements in the input stream	Recognizes the grammatical structure of the program
Returns token types, and actual token values *integer (356)*	Returns an internal representation of the program *typically a "parse tree"*

History: Describing a Grammar

- The first formal grammar definition dates back to the 4^{th} – 6^{th} century BC
- An Indian scholar named Pāṇini developed grammatical rules for Sanskrit
- Even today, grammar is taught by "parsing" a sentence and breaking it down into a formal structure:

The green vegetables are always disgusting, and I hate them



History: Describing a Grammar

- In 1959, John Backus (who led the FORTRAN team) had an insight: grammar rules were useful for things besides torturing students.
- John Backus and Peter Naur created a notation for describing the *grammar*, or *syntax*, of a computer language
- This notation is called *BNF*, which stands for Backus-Naur Form or Backus Normal Form
  - Many extensions and variants exist today, including Extended Backus–Naur form (EBNF) and Augmented Backus–Naur form (ABNF).

Backus Naur Form

- Every rule in Backus-Naur form has the following structure:
  <name> ::= expansion
- Every non-terminal symbol is enclosed by brackets
- Symbols may be concatenated in the expansion, indicating a sequence:
  <expr> ::= <term> <operator> <expr>
- Alternative expansions are separated by a vertical bar: |
  <expr> ::= <term> <operator> <expr> | <term>

BNF Example

<loop statement> ::= <while loop> | <for loop>

<while loop> ::= while “(“ <condition> “)“ <statement>

<for loop> ::= for “(“ <expression> “;” <expression> “;” <expression> “)“ <statement>

<assignment statement> ::= <variable> = <expression>

<digit> ::= “0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9”

<integer> ::= <digit> | <integer> <digit>

<letter> ::= <lowercase letter> | <uppercase letter>

Formal Grammars Eliminate Ambiguity

- A = 2 + 3 * 5
  Does A equal 25 or 17?
- 25:
  <expression> ::= <term> | <term> <operator> <expression>
  <term> ::= <variable> | <number>
  <operator> ::= “+” | “-” | “*” | “/”
- 17:
  <expression> ::= <term> | <term> <addoperator> <expression> |
  <term> <multoperator> <expression>
  <term> ::= <variable> | <number>
  <addoperator> ::= “+” | “-”
  <multoperator> ::= “+” | “/”

Using a Grammar

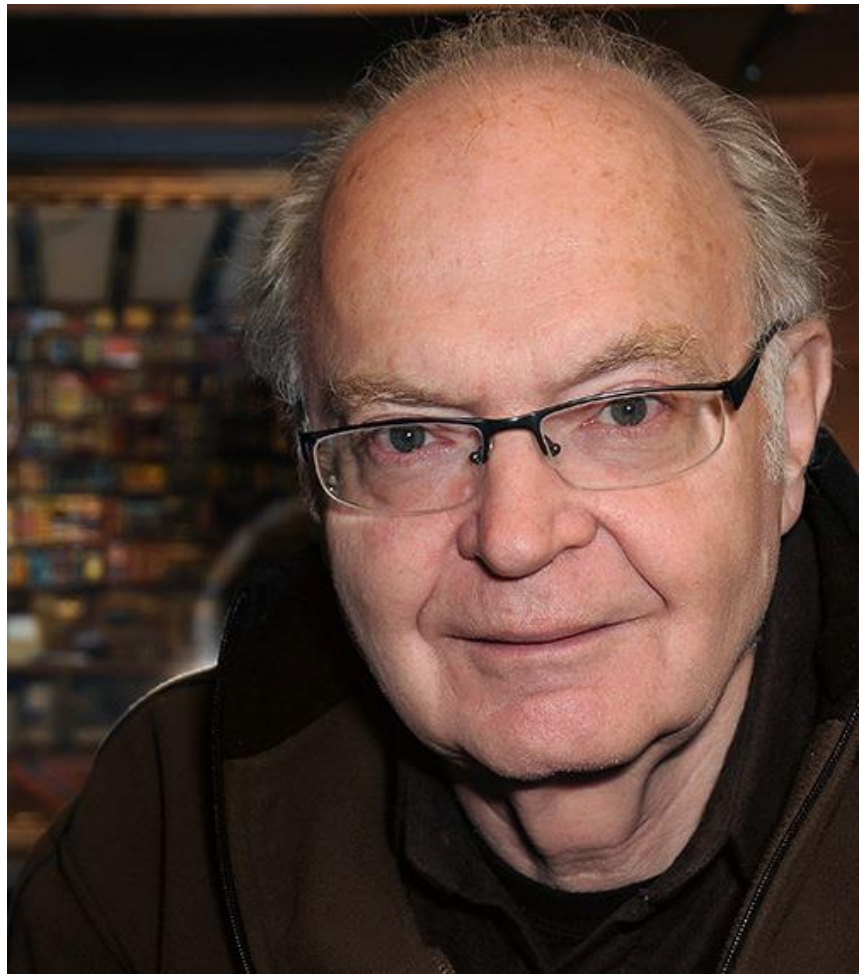
- Once a *grammar* is defined:
  - it can be used as a guide for hand-coding a parser, or
  - It can be used by a variety of modern tools – such as *YACC/Bison* or *ANTLR* - that will automatically build the code for a parser
- There are many “parser builders” – and each uses its own specification language for defining the grammar

Grammar “Formalism”

- It is possible to create grammars that are not parseable
- Grammars are categized based on certain characteristics, and specific algorithms have been created to parse each class of grammar
- Parser builders are based on specific classes of grammars and their associated algorithms.
  - They will reject grammars that they cannot parse

LR Grammars

- In 1965, Donald Knuth – author of *The Art of Computer Programming* - invented the *LR* parser (**L**eft to Right, **R**ightmost derivation). LR grammars and parsers are extremely memory intensive.
- In 1969, Frank DeRemer proposed a simplified version of the LR parser, called the *Look-Ahead LR* (LALR) – the most widely used type of grammar today
- Look-Ahead grammars rely on the ability to see the *next* token, without fetching it



LR versus LL

- Both *process tokens* left to right
- LL (left-to-right, leftmost derivation) grammars expand or derive the leftmost non-terminal first
  - Given a *grammar tree*, they attempt to expand the leftmost non-terminal first
- LR (left-to-right, rightmost derivation) grammars expand or derive the rightmost non-terminal first
  - Given a *grammar tree*, they attempt to expand the rightmost non-terminal first
- The grammar type affects the specific languages that can be parsed (and hence the design of programming languages), and the amount of processing power and memory required to parse them

Parser Generators

- In the early 70s, Stephen Johnson at Bell Labs created the *YACC* (Yet Another Compiler Compiler) parser generator, and the LEX lexical analyzer, drawing heavily on Knuth's algorithms.
- *Bison*, a YACC replacement, is included in most POSIX distributions.
- Currently popular parser generators include JavaCC (Java Compiler Compiler) and ANTLR (Another Tool for Language Recognition)
- ANTLR is able to process LL(0) and LL(1) (Left-to-right, Leftmost derivation) grammars, where the number represents the degree of look-ahead required
- **The classes of grammars are beyond the scope of this course, and *won't appear on assignments or exams*.**

Example: A Simple Program in “SICTRAN”

```
PRINT("Fibonacci Sequence \n\n\n");


i = 0;
prev_prev = -1;
prev      = 1;

WHILE (i <= 30)
{
    fib = prev_prev + prev;
    PRINT(i); PRINT(": "); PRINT(fib);
    IF (fib%5 == 0)
        PRINT(" Divisible by 5!");
    PRINT(“\n”);


    i = i + 1;
    prev_prev = prev;
    prev = fib;
}
```

Output of Simple Program

Fibonacci Sequence

0: 0 Divisible by 5!
1: 1
2: 1
3: 2
4: 3
5: 5 Divisible by 5!
6: 8
7: 13
8: 21
9: 34
10: 55 Divisible by 5!
11: 89
12: 144
13: 233
14: 377
15: 610 Divisible by 5!
16: 987
17: 1597
18: 2584
19: 4181
20: 6765 Divisible by 5!
21: 10946
22: 17711
23: 28657
24: 46368
25: 75025 Divisible by 5!
26: 121393
27: 196418
28: 317811
29: 514229
30: 832040 Divisible by 5!

Grammar for SICTRAN – Using *ANTLR*

```
program  : stmtList ;
stmtList : stmt ( ';' stmt )* ;

stmt : assignmentStmt
     | ifStmt
     | whileStmt
     | printStmt
     | compoundStmt
     ;
```

Char (s)	Meaning
:	Separates rule name from expansion
;	End of a rule
|	Or
(…)*	Repeat contents 0 or more times
(…)+	Repeat contents 1 or more times
'c'	Input character *c* – returned by scanner
NAME	Uppercase NAME indicates a token type that is returned by the scanner
# string	Comment

- The complete grammar is in the download files in Canvas

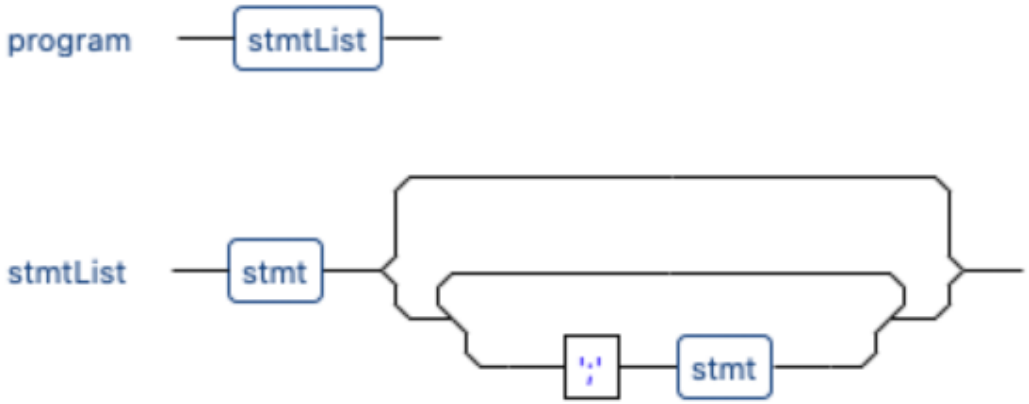
Grammar of SICTRAN - continued

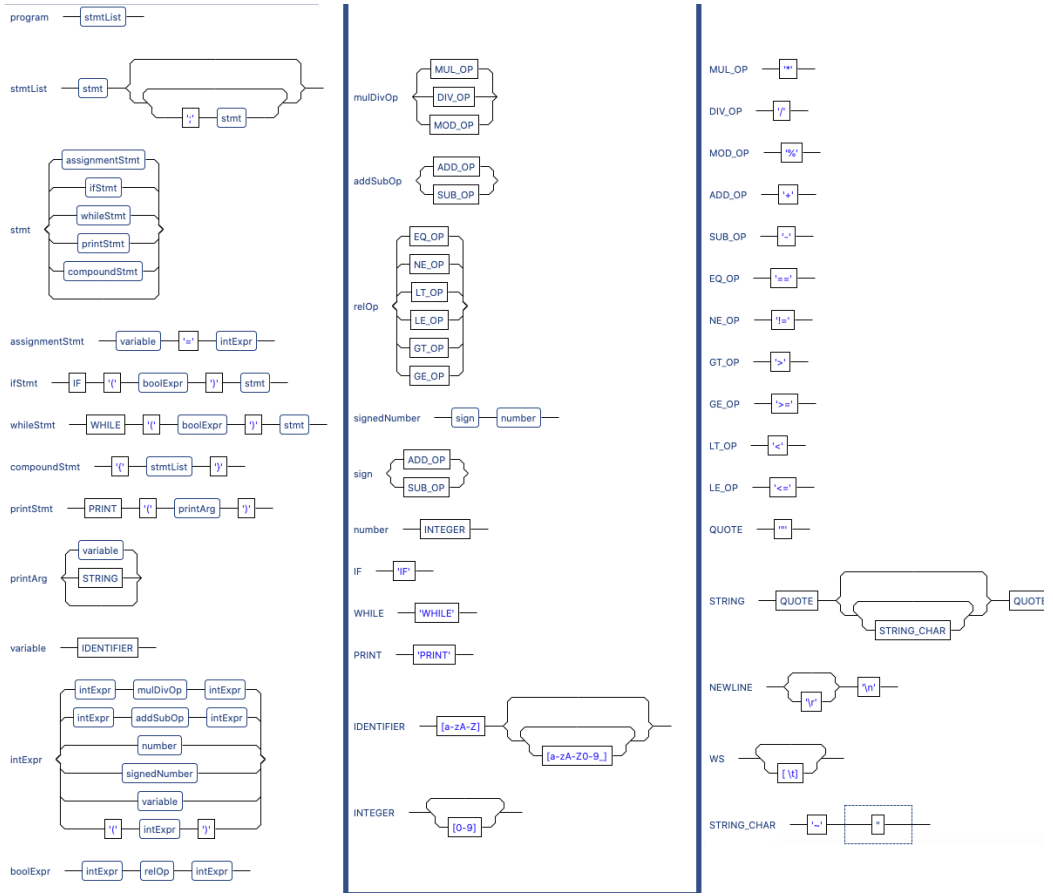
```
assignmentStmt : variable '=' intExpr ;
ifStmt  : IF '(' boolExpr ')' stmt ;
whileStmt  : WHILE '(' boolExpr ')'
stmt ;
compoundStmt   : '{' stmtList '}' ;

printStmt : PRINT '(' printArg ')' ;

printArg  : variable  # printVar
        | STRING    # printStr ;
```

```
variable : IDENTIFIER ;

intExpr : intExpr mulDivOp intExpr
        | intExpr addSubOp intExpr
        | number
        | signedNumber
        | variable
        | '(' intExpr ')'
        ;
```

- Full Antler grammar for SICTRAN is 39 rules

Grammar Diagram for SICTRAN





- Useful visual aide, generated by parser builder
- Full Antler grammar diagram is in the download files in Canvas

Using the Grammar: Writing a Parser

- **Grammar Tree:** Consider all of the rules in a grammar arranged into a tree structure
- Parsers may walk the tree *bottom up* or *top down*, with variants of each, based on the grammar class and the parser algorithm
- **Bottom Up:** start with the token in the bottom (or leaf) rule, and attempt to walk upwards in the tree by matching the input token stream against the tree.

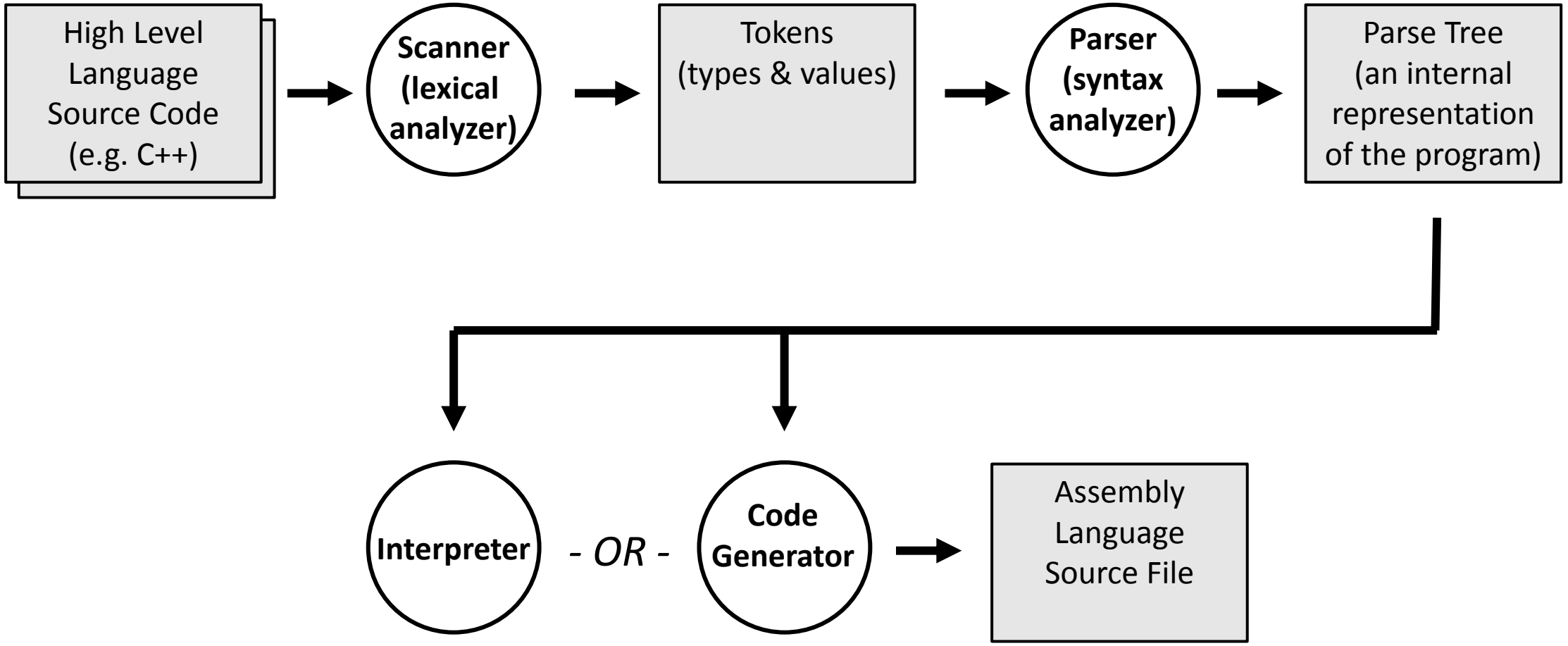
Using the Grammar: Writing a Parser

- **Top down:** start with the top-most rule (*program: stmtList ;*) and recursively walk down the tree looking for matches in the token stream.
- **Reminder:** The process in which a function calls itself directly or indirectly is called *recursion* and the corresponding function is called a *recursive function*
- We will learn how to write a *top-down*, *recursive-descent* parser to build a *parse tree*, which can then be passed to the *code generator*

What Does the Parser Do?

- The parser is the heart of the compiler
- Given a set of grammar rules, it analyzes the token stream and converts it to a data structure that can be used to generate code… typically a *parse tree*
- It calls on a *scanner*, or *lexical analyzer*, which accepts characters from the input stream and returns *tokens* to the parser.

Compiler Components - reminder



Aside: Data Structures and Algorithms

- As we get deeper into compilers, we will talk about a number of data structures, such as stacks, tables, arrays, trees, linked lists, and so on.
- There are well established coding techniques to create, manipulate, and traverse these structures. There are formal courses and books that explore them:
  - CMPE 126: Algorithms and Data Structure Design
  - CMPE 180A: Data Structures and Algorithms in C++
  - Don Knuth's *Art of Computer Programming*, volumes 1-3
- The pioneers in computing had to invent everything from the ground up, but modern software developers can draw on decades of knowledge

Building a Parser by Hand

- A top-down, recursive-descent parser can be coded entirely by hand, simply by writing a function for each rule.
- The “rule” functions may call functions for other rules
- Rule functions accept *parse trees* returned by the lower level functions they call, and combine those trees… returning the resulting tree to the caller.
- Let’s look at some concrete examples…

Top-Down, Recursive Descent Parser

program : stmtList ;

```
function parse_program {
   if ((tree = parse_stmtList()) == error)
      return error;
   return tree;
}
```

Top-Down, Recursive Descent Parser

stmtList : stmt (';' stmt)* ;

```
function parse_stmtList {
   if ((tree = parse_stmt()) == error)
      return error;
   while (parse_SEMICOLON() != error) {
      if ((subtree = parse_stmt()) != error)
         attach subtree to tree;
      else {
         ungetToken();    // the ability to “unget” a token is called a look-ahead rule
         break;
      }
   }
   return tree;
}
```

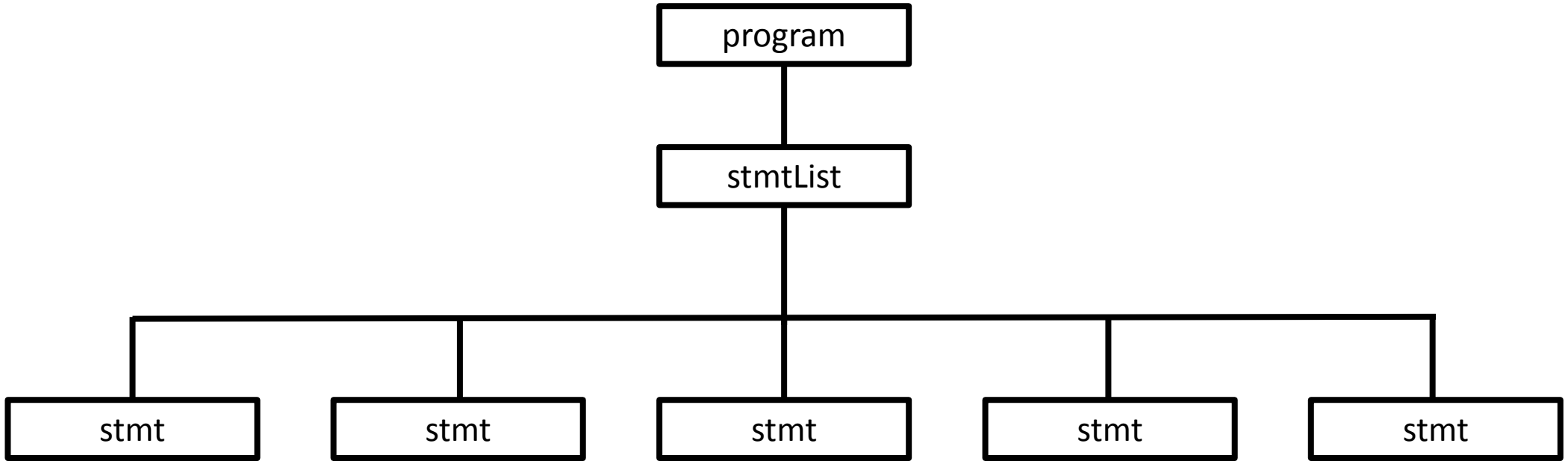
Top-Down, Recursive Descent Parser

SEMICOLON : ‘;’ ;

```
function parse_SEMICOLON {
    nextToken = getNextToken();     // call to scanner
    if (nextToken.type == SEMICOLON)
        return nextToken;
    else {
      ungetToken();    // the ability to “unget” a token is called a look-ahead rule
      return error;
     }
}
```

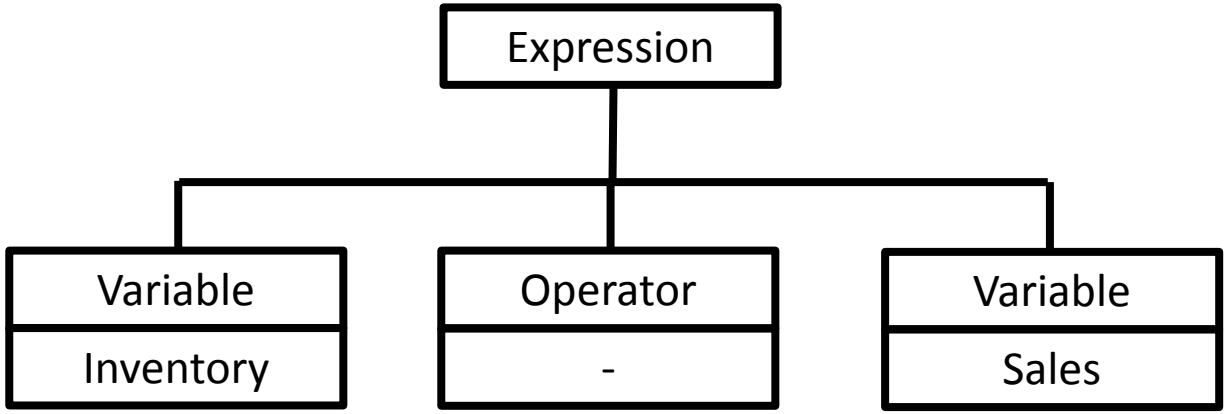
The Parse Tree – the Output of the Parser

- Sample Parse Tree (based on the example on the previous few slides)



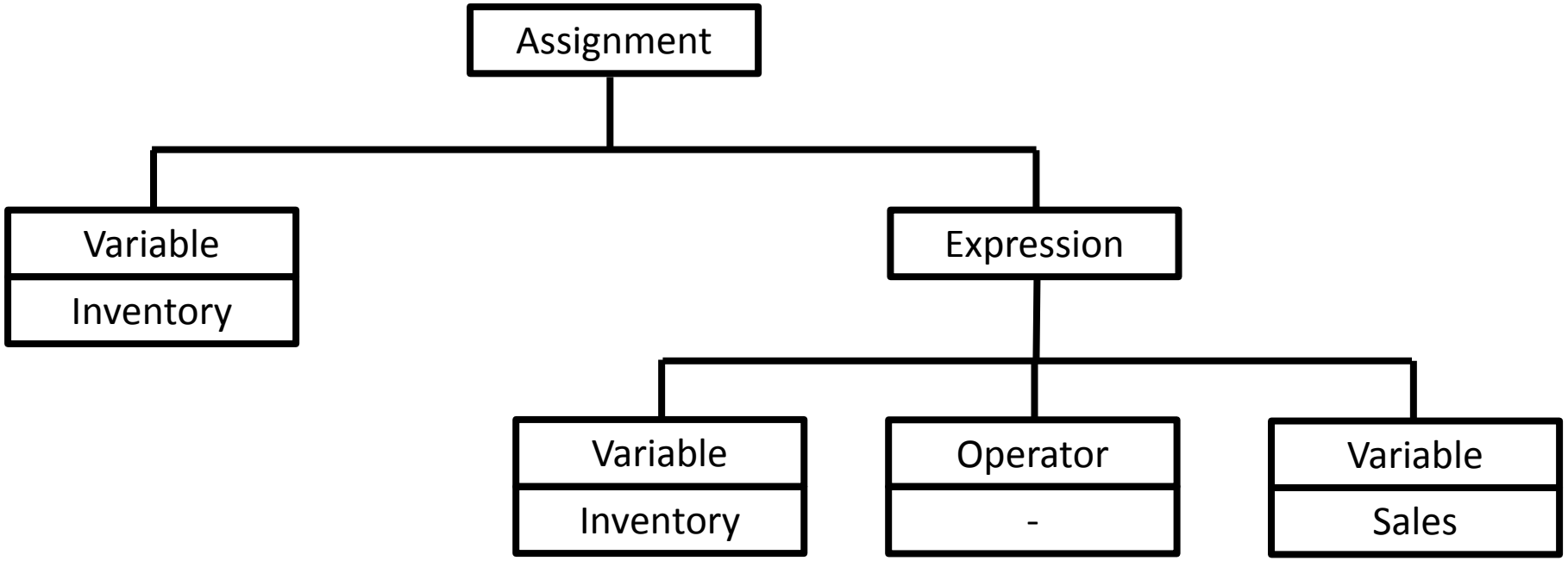
The Parse Tree – Additional Examples

- Inventory - Sales



The Parse Tree – Additional Examples

- Inventory = Inventory - Sales

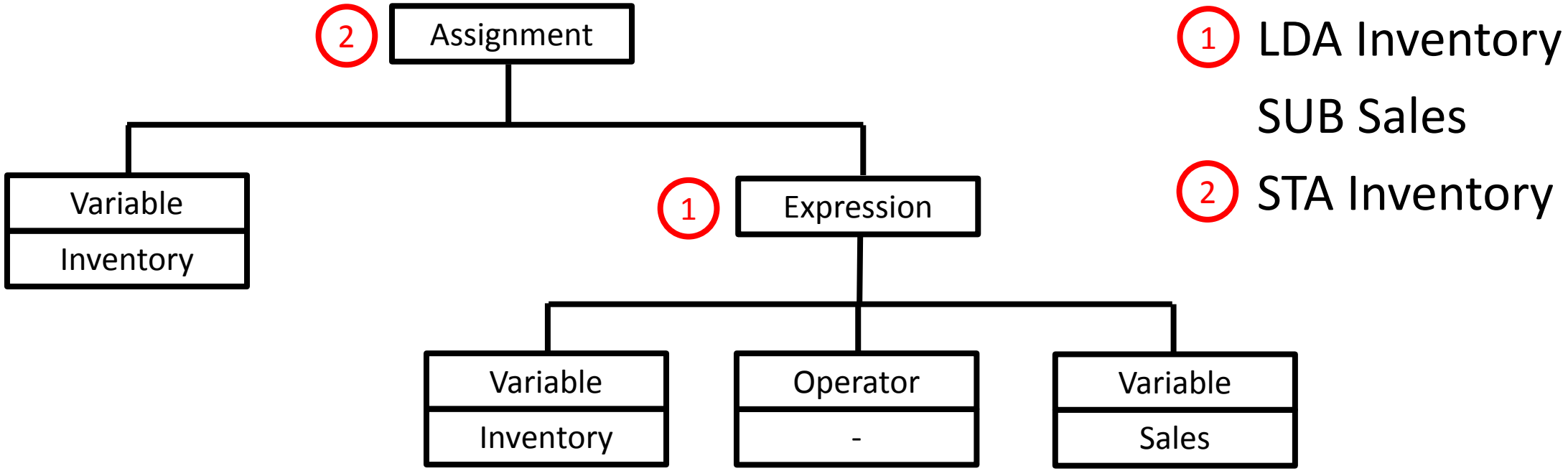


Hand Coding Versus Parser Builders

- It is certainly *possible* to write a parser entirely by hand, by writing a function for each rule in the grammar
  - A typical term project in compiler classes
- SICTRAN – a *very* simple, PASCAL-like language, has 39 rules
- Grammars for modern real-world programming languages may have many thousands of rules
- Parser builders make it feasible to create parsers for large, complex languages
- They are also useful for simple languages, to avoid the minutia of parser programming

Code Generation (hand waving)

- To generate code, we “walk” the tree and take appropriate action for each statement type



Walking the Parse Tree

- In the previous example, why did we visit and process the tree nodes in a rather strange order?
- That's the responsibility of the functions written for the code generator
- The code generator, like the parser, will have a separate function for each node type.
- The code in that function will determine how – and in what order - it processes child nodes

Fibonacci - A Simple Program in “SICTRAN”

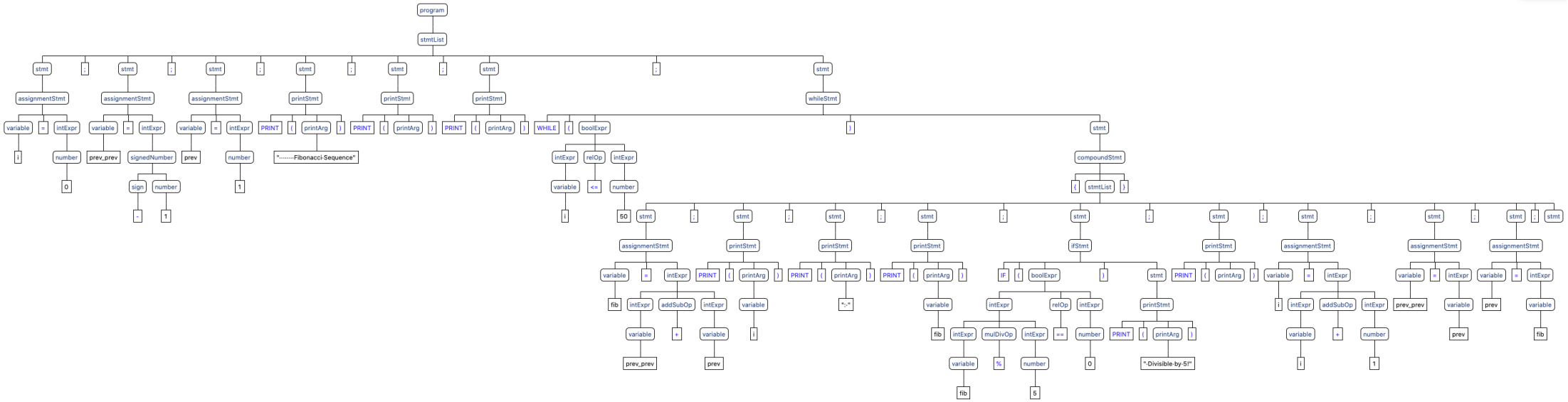
```
PRINT("Fibonacci Sequence \n\n\n");


i = 0;
prev_prev = -1;
prev      = 1;

WHILE (i <= 30)
{
    fib = prev_prev + prev;
    PRINT(i); PRINT(": "); PRINT(fib);
    IF (fib%5 == 0)
        PRINT(" Divisible by 5!");
    PRINT(“\n”);


    i = i + 1;
    prev_prev = prev;
    prev = fib;
}
```

Fibonacci Output

Fibonacci Sequence

0: 0 Divisible by 5!
1: 1
2: 1
3: 2
4: 3
5: 5 Divisible by 5!
6: 8
7: 13
8: 21
9: 34
. . .

Fibonacci Parse Tree – a 14 line program!



The complete parse tree is in the download files in Canvas