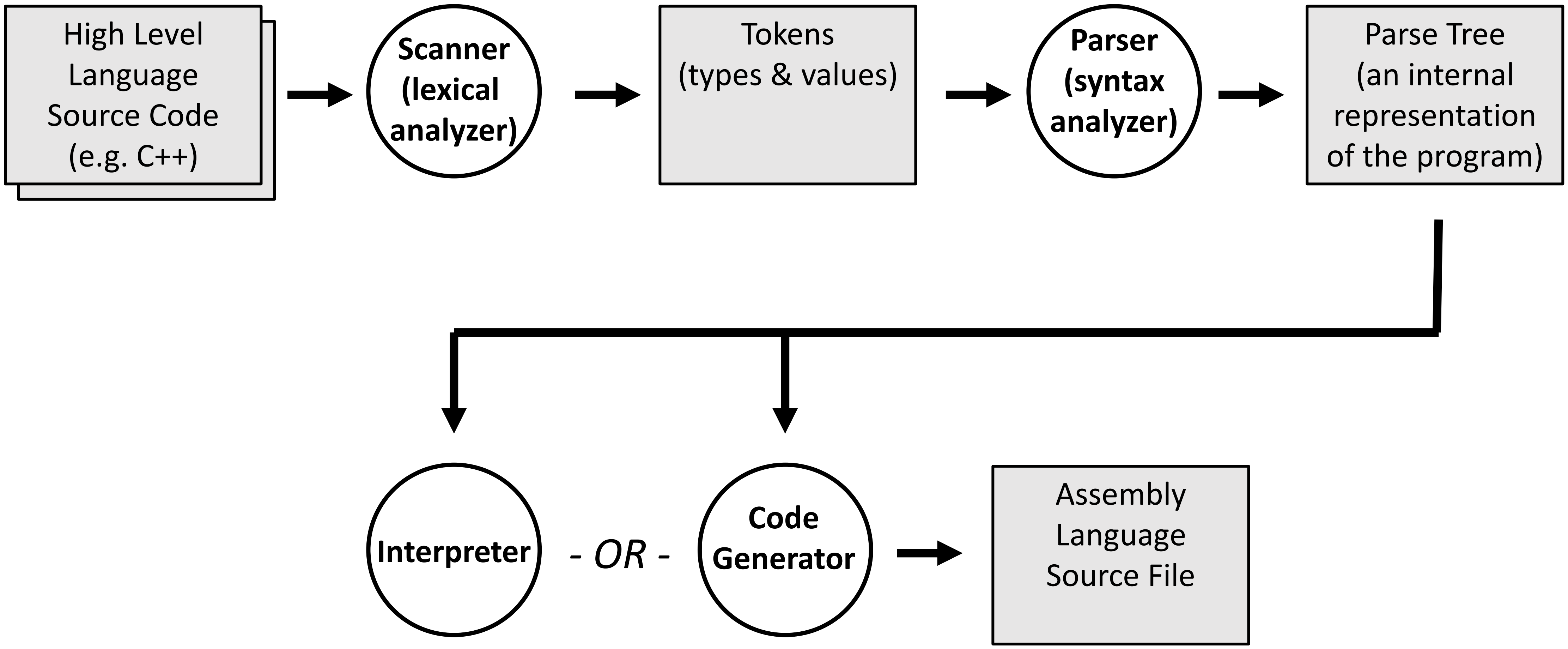
CMPE 220

Week 9 – Compilers (part 2)

Compiler Components

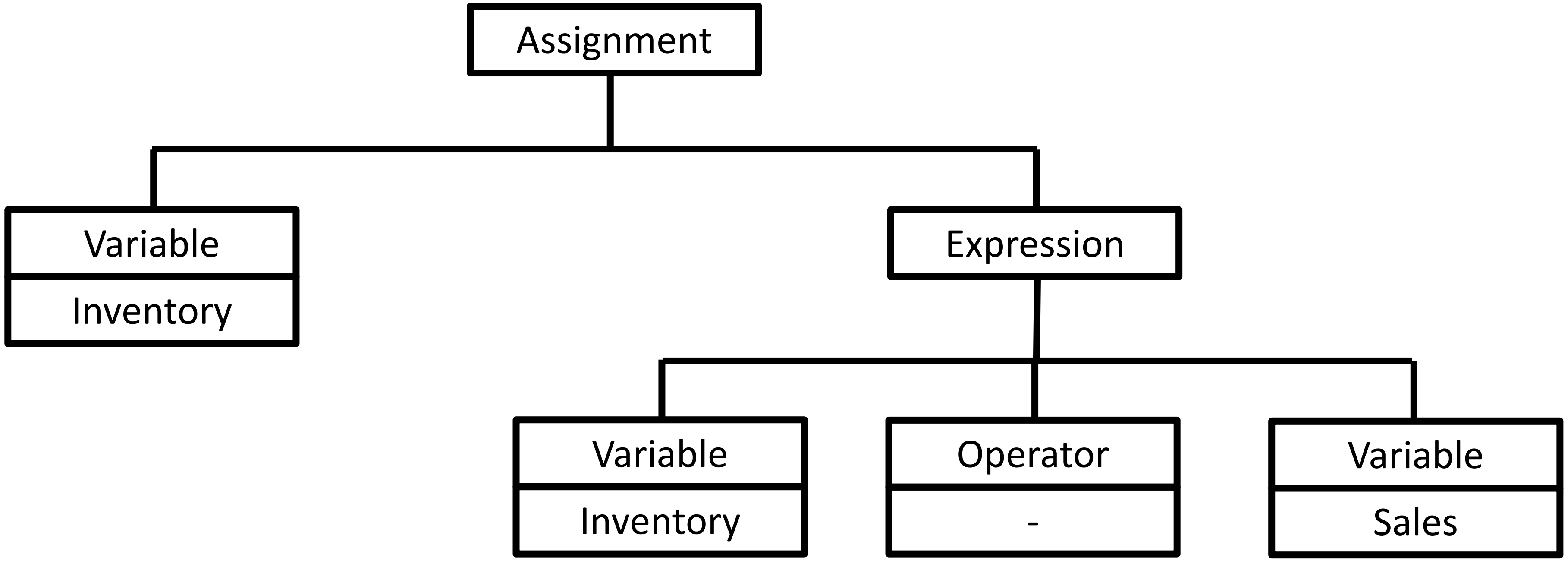


Elements of a Compiler

Scanner	**Parser**	**Code Generator**
Deals with the *lexicon*, or vocabulary of the source language	Deals with the *syntax*, or grammar of the source language	Deals with the *semantics*, or meaning of the language
Algorithm is Based On State transition table	Algorithm is Based On Grammar rules, typically represented as a *Grammar Tree*	Algorithm May be ad hoc, or may be an algorithm tied to the specific grammar class
Input character stream from input file	Input tokens	Input *Parse Tree* (program structure)
Output tokens	output *Parse Tree* (program structure)	output assembly language source code

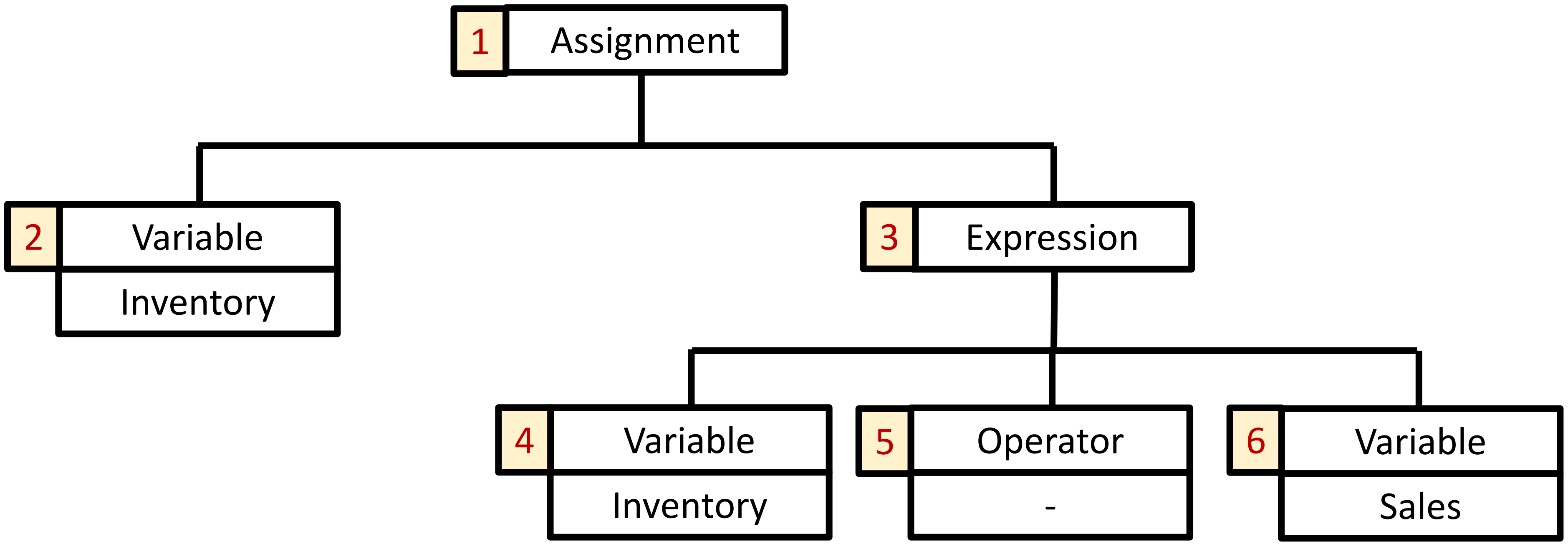
The Parse Tree

- A data structure generated by the *parser*, and used as input by an *interpreter* or *code generator*
- Inventory = Inventory - Sales



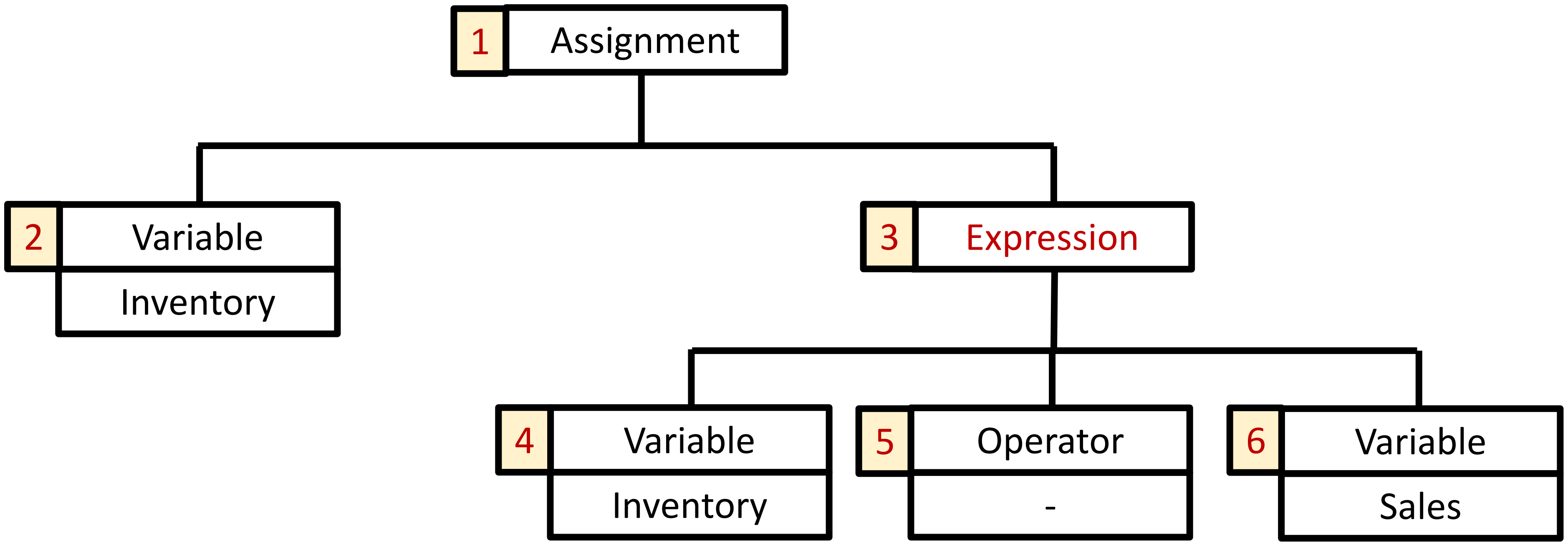
Notation for Examples

- Because we won't be discussing how a tree data structure is implemented, I will simply assign a number to each node for purposes of discussion. *This is just for convenience purposes during the lecture!*



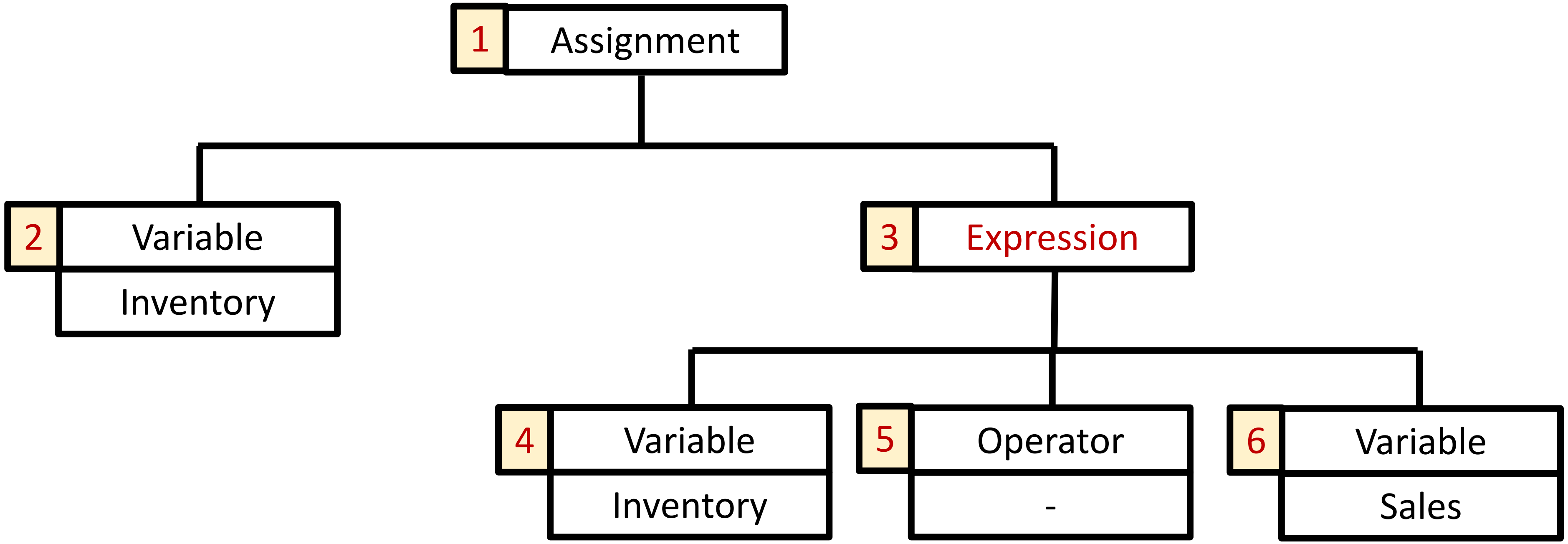
Code Generation

- The code generator “walks” the tree, visiting each node.
- It has a function corresponding to each node type, e.g.
  function processAssignment(1)



Code Generation: expression

- The function *processExpression(3)* will generate code that will subtract Sales from Inventory, and return the value.
- Should it generate an **integer** subtract, or a **floating point** subtract? (We'll answer that question in a few minutes)



Code Generation: Symbol Table

- Just like an assembler, a compiler builds and uses a *symbol table* – but the information is different
- Since the symbol table contains *semantic* information, it is built and used by the code generator
- The symbol table contains variable names, data types, and *scope*
- The specifics of the symbol table depend on the language being compiled

Symbol Use: Compiler Versus Assembler

High Level Program		Assembly Program		
AdjustInv	Inventory = Inventory – Sales;	AdjustInv	LDA	Inventory
			SUB	Sales
			STA	Inventory

- The *assembler* needs to determine addresses so that they can be assembled into object code
- The *compiler* does not need to know about addresses, because it simply passes the labels as part of the assembly language
  - Labels that are encountered in the source code are assigned to assembly language instructions by the code generator
  - The assembler resolves addresses, as we've already seen

Symbol Table: Language Dependencies

- Languages may require all type declarations, variables, and function names *before* they are used
- Because declarations precede their use, the code generator can build the symbol table and generate the code in a single pass
- If the language does not require declarations to precede use, then the code generator will require two passes:
  - Pass 1 builds symbol table
  - Pass 2 generates code

Symbol Table: Language Dependencies

- The symbol table may need to store definitions for complex data types: single- and multi-dimensional arrays, structures, and so on
- These type definitions are needed to generate the correct code both to create the data structures, and to access them
- The details on accomplishing this are usually covered in an intermediate compiler class

Generating a Symbol Table

- Two Pass code generation
  - First pass walks the parse tree and builds the symbol table
  - Second pass emits code
- Single Pass code generation
  - Symbol table is built on the fly as symbols are encountered in the parse tree
  - Much easier with languages that require symbols to be defined before use

Symbol Table: Generated Symbols

- The code generator may also need to *create new symbols* that don’t appear in the source program
  - Symbols pointing into complex data structures and allowing them to be referenced
  - Symbols as jump targets
  - Address labels for literals

Generated Symbols – Data Structures

Symbols pointing into complex data structures and allowing them to be referenced in assembly language

- struct product { int weight; double price; } apple;
  - apple_weight
  - apple_price

Generated Symbols – Jump Targets

- if (var1 == var2)
  {
  some code;
  }

```
                LDA           var1
                COMP          var2
                JLT           generated_label_1
                JGT           generated_label_1
                       some code
generated_label_1  next instruction
```

Generated Symbols - Literals

- myString = “the cow jumped over the moon”;

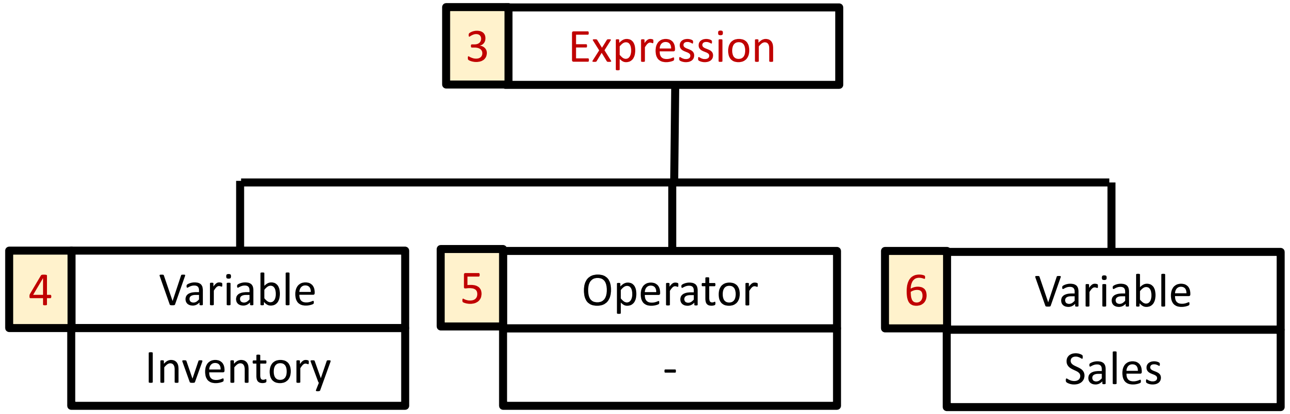
```
literal_01   BYTE   C‘the cow jumped over the moon’

                •
                •
                •

                LDA     literal_01
                STA     myString
```

Code Generation: Pass 2

- We've built the symbol table in Pass 1, so we know the type of the variables (integer). Let's look at some code

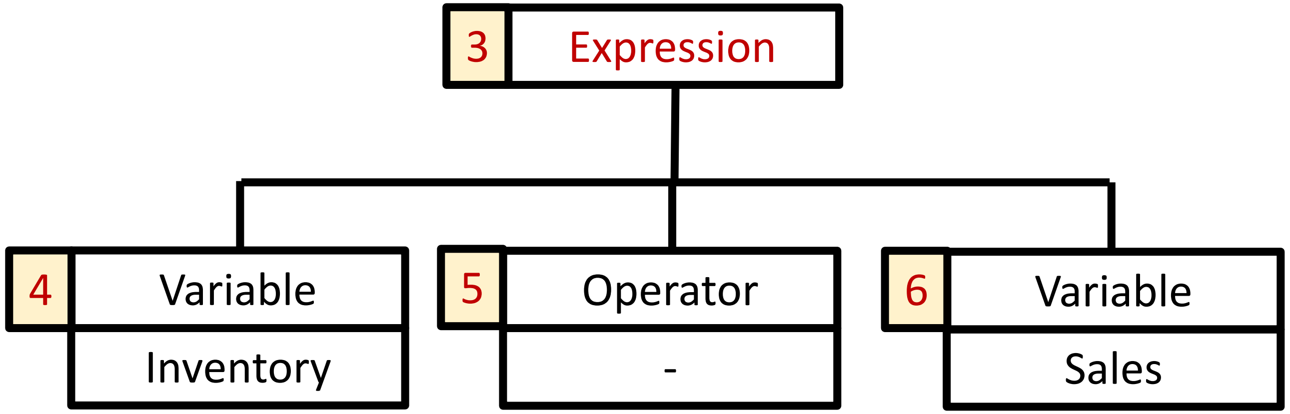


Code Generation: Pass 2

(ANIMATE)

```
function processExpression( node = 3 ) {
        if ( (type = getType(4) ) is integer) {
                processIntVariable(4);
        }
```

```
function processIntVariable( node ) {
        emit( ”\tLDA” );
        emit( “\t” + node.value  );
        emit ( “\r” );
        return 0;
}
```

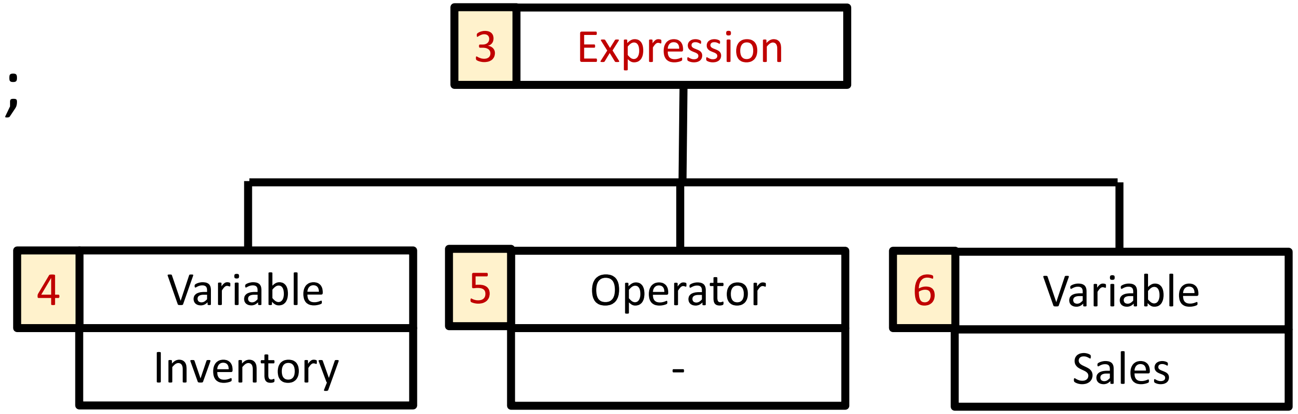


Code Generation: Pass 2

(ANIMATE)

```
function processExpression( node = 3 ) {
        if ( (type = getType(4) ) is integer) {
                processIntVariable(4);
        }
        switch ( getValue(5) ) {
        case '-':
                subIntVariable(6);
                break;
```

```
function subIntVariable( node ) {
        emit( "\tSUB" );
        emit( "\t" + node.value  );
        emit ( "\r" );
        return 0;
}
```



Code Generation: 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;
}
```

Generated Code

- 08-annotated-assembly-code.pdf

Generated Code

```
;
; Object file for SICTRAN source program 'fibonacci.sic'.
;
;
; Name table
;
fib         WORD  0
i           WORD  0
prev        WORD  0
prev_prev   WORD  0
;
; String table
;
L000001     BYTE  C'Fibonacci Sequence'
L000003     BYTE  C' Divisible by 5!'
L000002     BYTE  C': '
;
; Start program
;
            START 1000
;
; i=0
;
            LDA   0
            STA   i
```

Notice string literals have been declared and initialized, with labels generated by the compiler.

Generated Code

```
;
; prev_prev=-1
;
        LDA   -1
        STA   prev_prev
;
; prev=1
;
        LDA   1
        STA   prev
;
; PRINT("Fibonacci Sequence")
;
        LDA   1             ; set print mode to 1 (print string pointed to by B)
        LDB   #L000001      ; get address of literal string to print
        JSUB  PRINT
;
; PRINT()
;
        LDA   0             ; set print mode to 0 (print line break)
        JSUB  PRINT
;
; PRINT()
;
        LDA   0             ; set print mode to 0 (print line break)
        JSUB  PRINT
```

Generated Code

```
;
; WHILE(i<=50)
;
L000004     LDA   i
            COMP  50
            JGT   L000005
;
; fib=prev_prev+prev
;
            LDA   prev_prev
            ADD   prev
            STA   fib
;
; PRINT(i)
;
            LDA   2             ; set print mode to 2 (print value  in location pointed to
by B)
            LDB   #i            ; get address of value to print
            JSUB  PRINT
;
; PRINT(": ")
;
            LDA   1             ; set print mode to 1 (print string pointed to by B)
            LDB   #L000002      ; get address of literal string to print
            JSUB  PRINT
;
; PRINT(fib)
;
            LDA   2             ; set print mode to 2 (print value in location in B
            LDB   #fib          ; get address of value to print
            JSUB  PRINT
```

Notice that labels (jump targets) have been generated by the compiler.

Generated Code

```
;
; IF(fib%5==0)
;
            LDA    fib
            LDB    5
            JSUB   MOD
            COMP   0
            JLT    L000006
            JGT    L000006
;
; PRINT(" Divisible by 5!")
;
            LDA    1
            LDB    #L000003
            JSUB   PRINT
;
; PRINT()
;
L000006     LDA    0
            JSUB   PRINT
;
; i=i+1
;
            LDA    i
            ADD    1
            STA    i
```

Unsupported operations such as **%** may be implemented as subroutine calls.

This is a common way to support BCD arithmetic on machines where it is not supported in hardware.

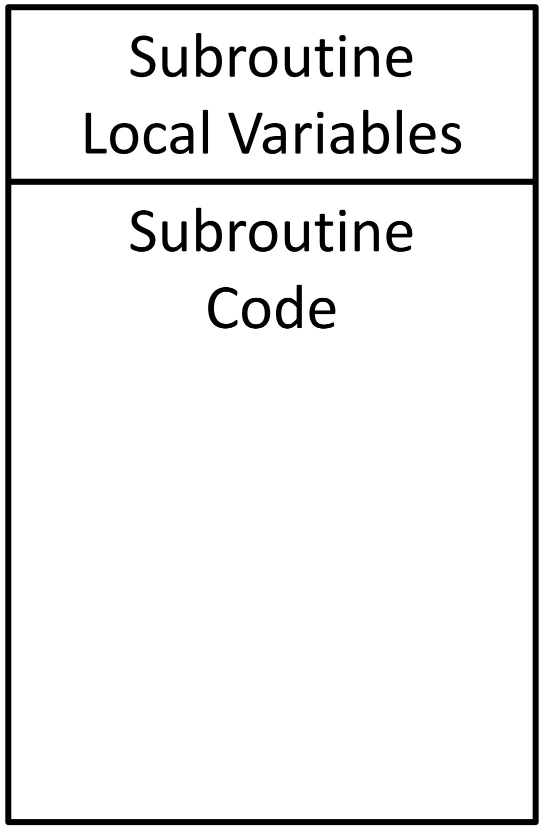
Generated Code

```
;
; prev_prev=prev
;
            LDA    prev
            STA    prev_prev
;
; prev=fib
;
            LDA    fib
            STA    prev
            JMP    L000004
;
; End program
;
L000005     JMP    ENDPROGRAM
```

Code Generation: Local Data

This is a *machine dependent* compiler feature

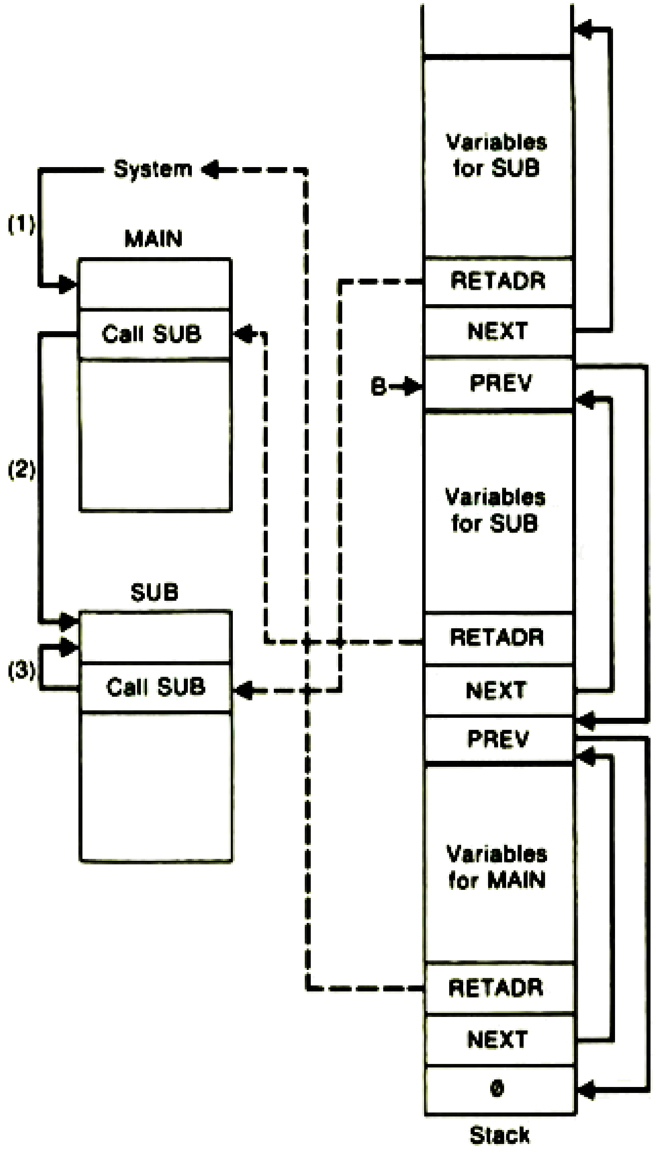
- Data may be stored with the subroutine code – but it will be overwritten if the subroutine calls itself (recursion)



Code Generation: Local Data & Recursion

<u>This is a *machine dependent* compiler feature</u>

- Many languages save subroutine arguments, local variable instances, and the return address each time a subroutine is called
  - Necessary for *recursion*
- The best mechanism for doing this is a *stack*
- Each time a subroutine is called, arguments are placed on the stack, and space is allocated for local variables
  - Code needs to be *emitted* at the start and the end of a subroutine to handle this



Code Generation: The Stack

- When contained in a subroutine, variable references are not fixed – *we cannot simply pass the labels through to the assembler*
- Local variables need to be “flagged” in the symbol table
- Address references for local variables are relative to the *stack pointer*
- *Accomplishing this depends on the addressing modes available on the underlying machine*

Code Optimizations: Register Allocation

Register allocation is a *machine dependent* code optimization

- Registers are faster than memory – so we want to make effective use of them!
- Optimizing registers requires knowledge of the machine architecture
- The code generator must have a *register allocation function* to keep track of which registers are in use as the parse tree is walked and code is generated

Code Optimizations: Register Allocation

Reduce Memory Accesses by Eliminating Redundant Loads/Stores

OldInventory = Inventory;

Inventory = Inventory – Sales;

Unoptimized		Optimized	
LDA	Inventory	LDA	Inventory
STA	OldInventory	STA	OldInventory
LDA	Inventory		
SUB	Sales	SUB	Sales
STA	Inventory	STA	Inventory

Code Optimizations: Register Allocation

- One Register versus Multiple Registers
  if (expression1 == expression2)

One Register	Multiple Registers
Evaluate expression1 and return result in A Store A in a temporary memory location Evaluate expression2 and return result in A Compare A to the temporary memory location	Evaluate expression1 and return result in A Evaluate expression2 and return result in B Compare A to B

The Code Generator makes use of a register allocation function to keep track of what's in each register, and which are available for use

Code Optimizations: Register Allocation

Use Registers for Most Frequently Accessed Data

- Within each code block (single-entry, single exit)
- Load data into registers when entering block, store when exiting block
- Determine most-referenced data locations
- Keep the values of those locations in registers

Code Optimizations: Register Allocation

Optimize Register Usage in Inner Loops

- Programs spend most of their time in “inner loops”
- The register allocation function should place a higher emphasis on register usage as nesting depth increases
- Optimization doesn’t always make the right decisions!
  - A deeply-nested inner loop that is executed three times is less critical than an outer loop that is executed 1,000 times

Code Optimizations: Invariant Code

This is a *machine* *independent* code optimization

- ***Invariant* Code Optimization:** computations that do not change should be removed from loops:

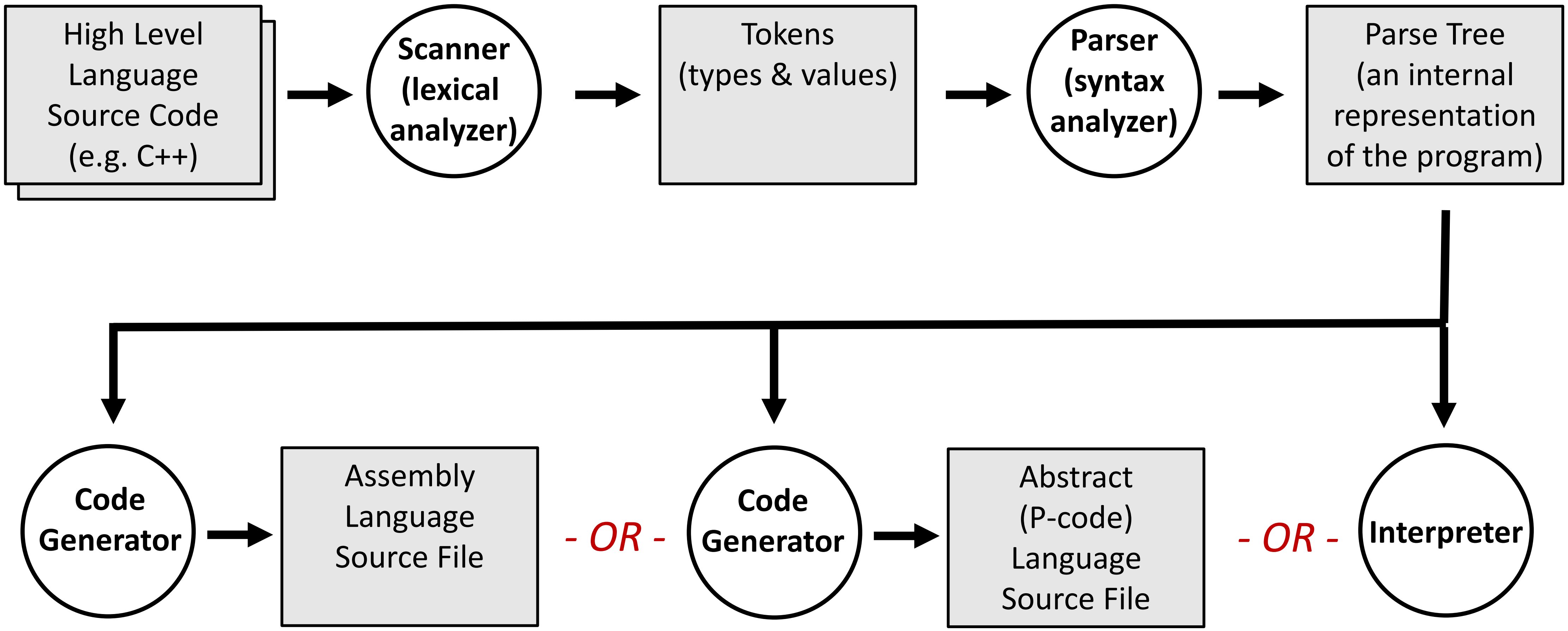
```
for (index = 0; index < max-1; index++ ) {
        if (array[index] > array[index+1] {
                temp = array[index];
                 array[index]  = array[index+1];
                 array[index+1] = temp;
        }
}
```

Code Optimizations: Invariant Code

- ***Invariant* Code Optimization:** the *optimized* assembly language would be equivalent to:

```
stop = max-1;
for (index = 0; index < stop; index++ ) {
        if (array[index] > array[index+1] {
                temp = array[index];
                 array[index]  = array[index+1];
                 array[index+1] = temp;
        }
}
```

Compiler Output Options



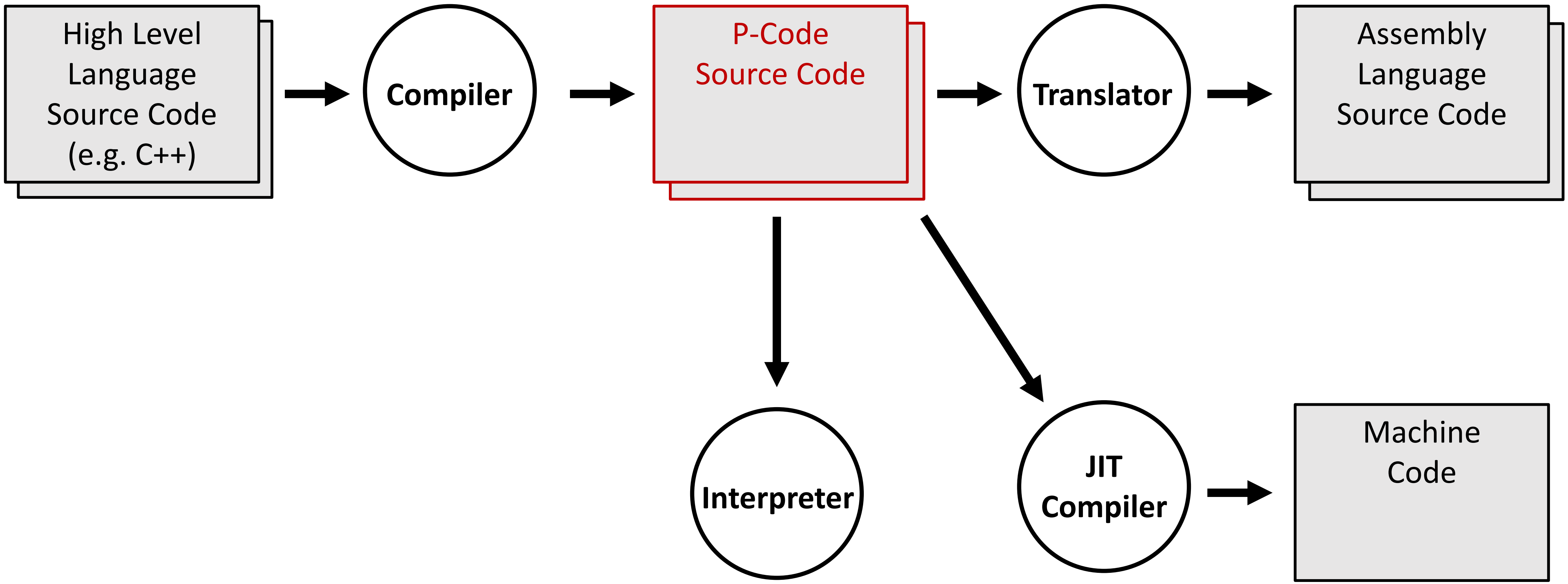
Compiler Output Options: Assembly

- The original – and still the primary – purpose of a high-level language compiler is to convert the high-level code to assembly language for a particular machine
- There are other options that can be considered for different purposes

Compiler Output Options: P-code

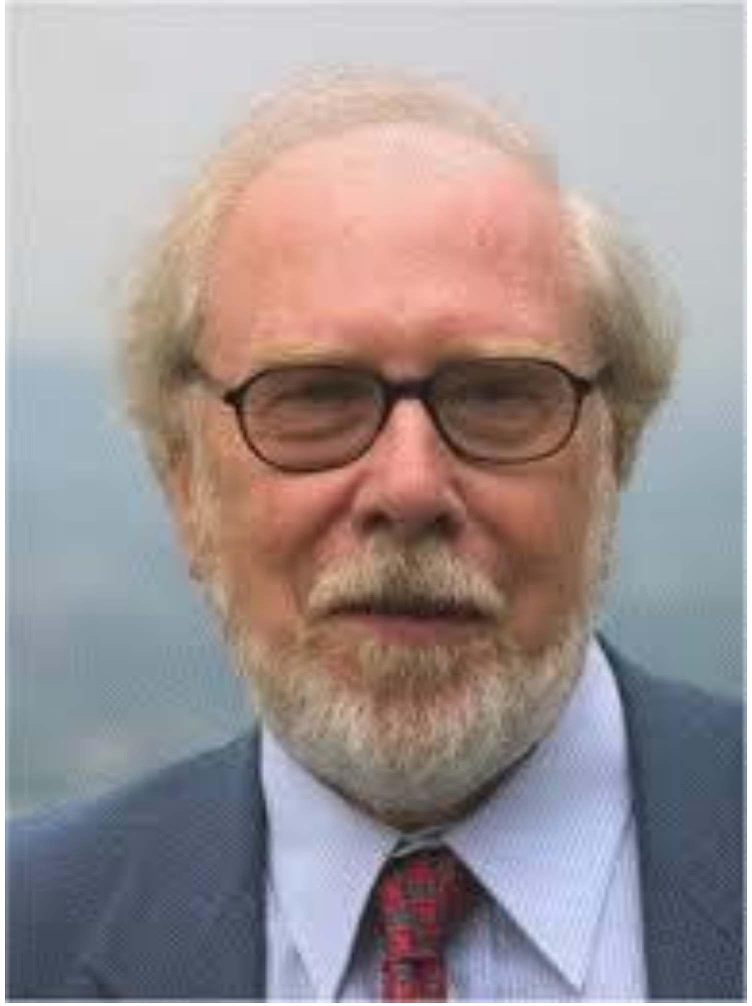
- **P-code:** precompiled code, or portable code, or Pascal code
- Assembly language code for an abstract (hypothetical) machine
- P-code may be *interpreted*
- P-code may be translated into an assembly language instruction set for the current machine
- P-code may be “compiled” on the fly to machine code
- Since P-code interpreters and translators exist on any machine, writing a compiler that emits p-code is a quick way to achieve language portability

Building Software



Compiler Output Options: P-code Examples

- **P-code:** may also refer to *Pascal code* – an early portable layer
- PASCAL – 1970 – created by Swiss computer scientist Nicklaus Wirth as a highly structured instructional language; quickly replaced ALGOL as the teaching language of choice
- First P-code emitter - 1973
- UCSD P-Machine – 1977 – widely used in academia, as well as commercially



Compiler Output Options: P-code Examples

- **JVM (Java Virtual Machine):** a virtual machine built by James Gosling and Brendan Eich – 1995 – specifically for Java - but now widely used
  - Virtual machine instruction set
  - Manages memory and system resources

Interpreters

- **Early interpreters (e.g. BASIC – 1964):** Line-by-line scanning, parsing and interpreting

Modern Approaches

- Scan and Parse entire program – generate parse tree
  - Interpret parse tree rather than source code
- Compile on-the-fly – as each line or statement is encountered, scan it, parse it, and generate code.
  - Great for optimizing loop performance

Pure Interpreters Are *Applications*

- A traditional interpreter is not “system software”
  - They don’t produce executable code
  - They don’t have system dependencies
- They simply perform operations that are described to them in a high-level programming language

Really? Just an Application?

- Write a calculator application that accepts keyboard input - any string of numbers separated by operators – and prints the result:
  *15.3 / 12.77 + 0.8 * 87*
- You could write that application. You might write a scanner to break the input down into tokens
- Now let's add the ability to group operations:
  *17.4 * (18.2 – 3.0) + (7.77 / 8.9)*
- Maybe at this point you'll write a parser to break down the elements and build a tree structure that you can walk to actually perform the operations

Yes. Just an Application.

- Now let's add the idea of variables, so we can store and reuse values:
  *savethis = 17.4 * (18.2 – 3.0) + (7.77 / 8.9)*
  *result = 0.15 * savethis*
- You can parse this into a parse tree, and walk the tree to perform the operations.
- You'll need to create a data structure to save the contents of the variables
- But it's still just an application!
  - It doesn't generate code
  - It isn't system dependent

Variable Name	Contents
savethis	265.353
result	39.802

Interpreted Languages Can Be Very Rich

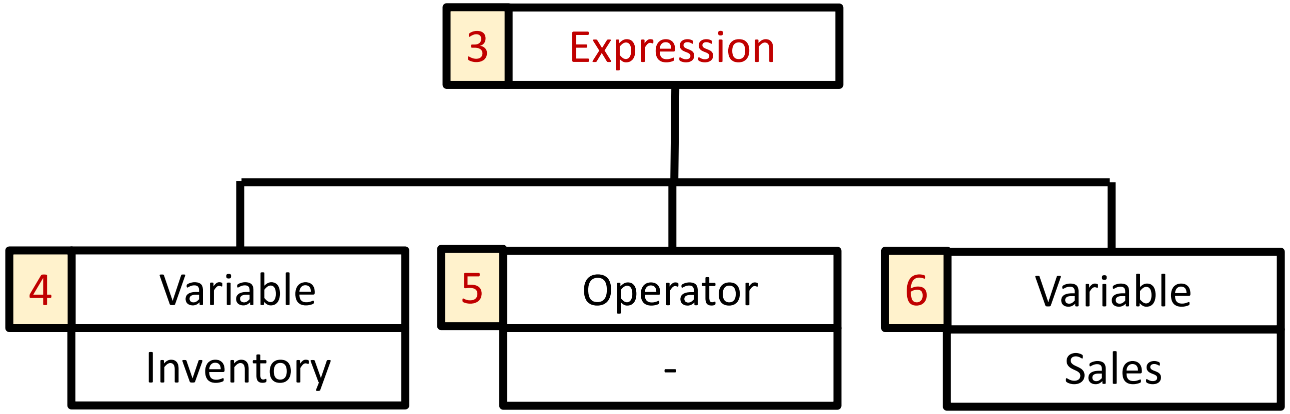
- We started with a simple calculator, added grouped expressions, and then assignments and variables.
- We can continue to add features:
  - Type declarations
  - While loops
  - If/then/else statements
  - Functions and function calls
- The process of writing an interpreter is the same:
  - scanner
  - parser
  - Executer (walks the tree and executes instructions)

Interpreting a Parse Tree

(ANIMATE)

```
function processExpression( node = 3 ) {
        leftValue = processVariable(4);
        rightValue = processVariable(6);
```

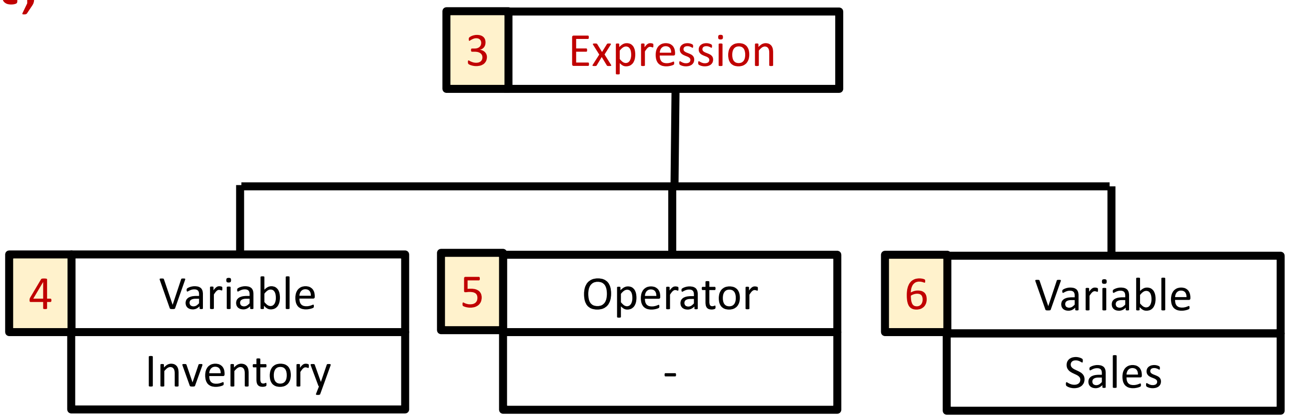
```
function processVariable( node ) {
    variableName = getValue( node );
    return getContents( variableName );
}
```



Variable Name	**Contents**
Inventory	345
Sales	22

Interpreting a Parse Tree

```
function processExpression( node = 3 ) {
        leftValue = processVariable(4);
        rightValue = processVariable(6);
        switch ( getValue(5) ) {
        case '-':
                result = left - right;
                return result;
```



Why Write an Interpreter?

Easy to Use / Great for Beginners

- BASIC (Beginner's All-Purpose Symbolic Instruction Code) – 1964 – created as an easy-to-use teaching language, which became popular due to its simplicity
- Immediate syntax checking
- Immediate execution – doesn't require a complicated build process

Why Write an Interpreter?

Portability!

- Allows languages to be run on any operating system and any architecture without writing a compiler
- PHP – 1995 – now the most widely used language for web applications.
- JavaScript – 1995 – embedded in web browsers on a wide range of platforms. Portability is integral to the very concept of Javascript.

PHP and Facebook: Case Study

- Facebook was originally written in PHP
- To improve performance, Facebook adopted the HipHop *translator* in 2010, which translated the php code to C++ (which was then compiled)
- The compiled C++ code improved performance by a factor of two
  - Issues: some PHP language constructs did not translate well with HipHop
- In 2013, Facebook switched to the HipHop Virtual Machine (HHVM), a p-code abstract machine.
  - PHP code is compiled into the HHVM instruction set
  - HHVM instructions are compiled on demand into machine code
- New code being written in Hack, a *strongly-typed* PHP-like language

PHP and Facebook: the Takeaway

- There are a *range of technologies* today that can be used for software development and production operations:
  - Macro Processors
  - Compilers
  - Cross-compilers
  - Language Translators
  - Assemblers
  - Interpreters
  - IDEs & Debuggers
  - Just-In-Time Compilers
- Key software concepts provide a basis for all of these technologies:
  - Lexical analysis (scanning)
  - Syntax analysis (parsing)
    - Formal grammars
  - Sematic processing
    - Code generation
    - Interpreting

For Next Week

- Log in to Canvas and complete Assignment 5