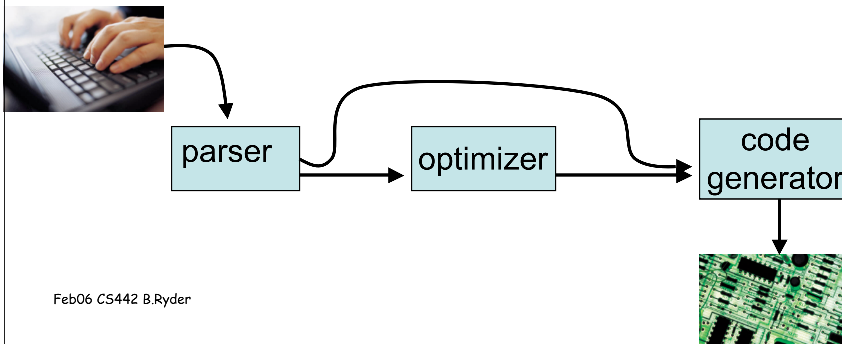
# What are compilers?

Dr. Barbara G. Ryder
Dept of Computer Science
ryder@cs.rutgers.edu
http://www.cs.rutgers.edu/~ryder



# A Compiler

- A program that translates computer programs that people write, into a language that a machine can execute



# Parser

- Programs are written in a high-level language such as Java or C++
  - A grammar description of the programming language describes a well-formed program
  - Example of an English grammar excerpt:
    - sentence = noun verb  John swims
    - sentence = noun verb adverb John swims well
    - sentence = article adjective noun verb the tall boy swims
    - sentence = article noun verb the boy swims
- Parsers check that a program adheres to the rules of the programming language's grammar
  - If so, parser translates the program into an internal representation used by the compiler



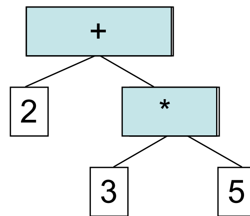
# Code Generator

- Translates the internal representation of a program into machine language
- Has all the info it needs in the internal representation and knows the program is correct according to the rules of the grammar
- Is targeted to output a specific machine language for a specific kind of computer
  - Can change to a different computer chip with a different instruction set by changing code generators, without other changes to the compiler

# Arithmetic Expressions

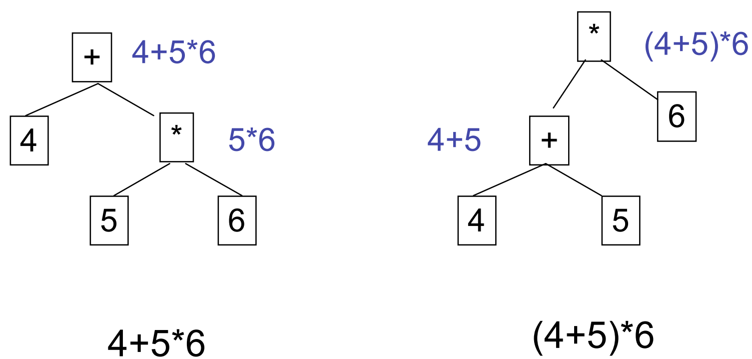
- Arithmetic expressions using +* operations
  - Assume the acc can perform acc=acc <op> mem[const] where <op> can be any of +*
- Assume we only use integer constants in our expressions
- How can we represent an expression?

  2 + 3 * 5?






# Examples



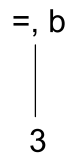
# Internal Representation

- As we parse an expression we can build a (tree) representation of it
- Let's consider expressions involving integer variables and integer constants



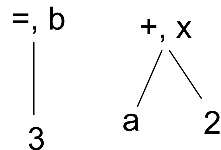
# Example

b = 3

x = a + 2 


y = b + 1

z = y * x

w = a + 2 


u = 4 * x

# Example

b = 3
x = a + 2
y = b + 1
z = y * x
w = a + 2
u = 4 * x

+, y
=, b
+, x
3
a
2
1

*, z
+, y
=, b
+, x
3
a
2
1



# Example

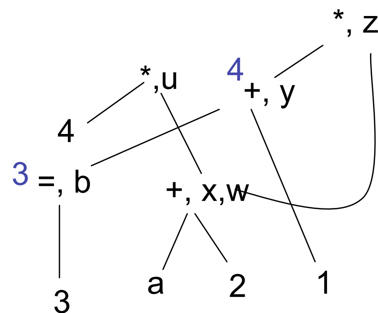
b = 3
x = a + 2
y = b + 1
z = y * x
w = a + 2
u = 4 * x

*, z
+, y
=, b
+, x,w
3
a
2
1

*, z
*,u
+, y
4
=, b
+, x,w
3
a
2
1

# Example

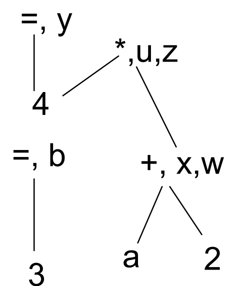
- **Optimizations**
  - Two labels on a+2 node saves computation; is encoded as x=a+2; w=x;
  - Can figure out constant operands




*After find constants, Then z and u are same expression!*

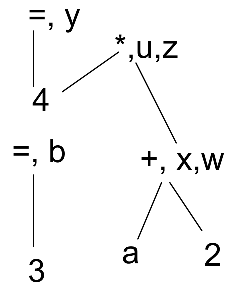


# Example




**Now how to generate machine language for this expression? Walk the graph and at each internal node, generate appropriate code.**

# Transformed Code




```
b = 3
x = a + 2
w = x
y = 4
z = 4 * x
u = z
```



# Comparison

*Original code*

```
b = 3
x = a + 2
y = b + 1
z = y * x
w = a + 2
u = 4 * x
```

*Optimized code*

```
b = 3
x = a + 2
w = x
y = 4
z = 4 * x
u = z
```

Note: fewer arithmetic operations and many inexpensive copies.

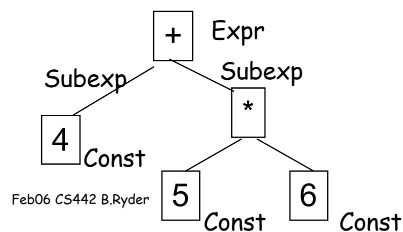
# Code Generation

```
b = 3            mem[42] =3
x = a + 2        acc = 2
                 acc = acc + mem[43]
                 mem[44] = acc
w = x            mem[45] = acc
y = 4            mem[46] = 4
z = 4 * x        acc = 4
                 acc = acc * mem[44]
                 mem[46] = acc
u = z            mem[47] = acc
```



# Digging Deeper - Grammars

- How do we define well-formed expressions?
  Expr = Const <op> Const, where <op> is +*
- How do we show the rules of arithmetic for unparenthesized expressions?
  Expr = Subexp + Subexp
  Subexp = Const * Const
  Subexp = Const

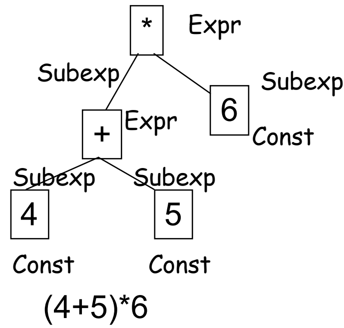



Grammar rules correspond to shape of the tree.

# Examples

Expr = Subexp + Subexp
Subexp = Const* Const
Subexp = Const

Adding parenthesized expressions requires new rule:
Subexp = ( Expr )



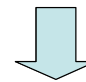

(4+5)*6



# Example

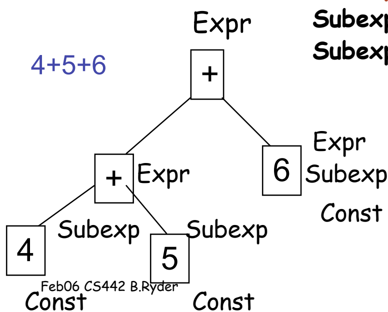
Adding arbitrary length, nested subexpressions requires changing the grammar.

Expr = Subexp + Subexp
Subexp = Const * Const
Subexp = Const
Subexp = ( Expr )

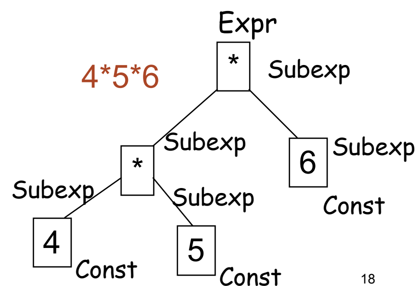


Expr = Expr + Expr
Expr = Subexp
Subexp = Subexp * Subexp
Subexp = Const
Subexp = ( Expr )

4+5+6



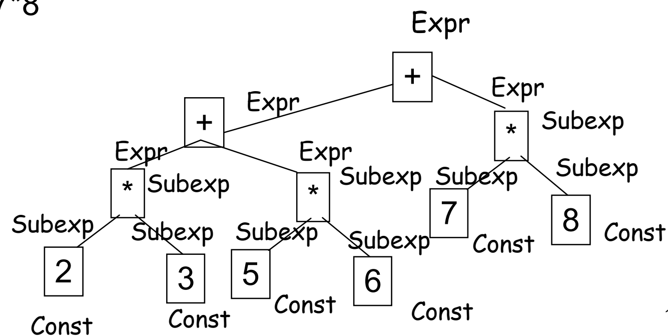

4*5*6



# Complicated Example

Expr = Expr + Expr
Expr = Subexp
Subexp = Subexp * Subexp
Subexp = Const
Subexp = ( Expr )

2*3+5*6+7*8






# Summing Up

- **Parser uses grammar rules to check expressions for correct structure -- *syntax***
- **If correct, then builds the expression graphs**
- **Optimizes the graphs to find repeated subexpressions and constants that can be evaluated at compile-time**
- **Then generates code from the graph**

# Interpreters

- *A compiler translates a program into machine language*
- *An interpreter translates the statements in a program by executing equivalent commands*
  - No real translation step
- **Interpretation requires that a programming language have a defined meaning for its statements -- *semantics***
  - Sometimes defined mathematically, sometimes in English.



# Expression Interpreter

- Requires
  - input expression
  - rules for operator evaluation
  - a stack -- storage for partial results
    - Think of how you store plates in your cupboard;
      - » Take next plate to use off the top of the pile
      - » Stack newly cleaned plates on the top of the pile
      - » LIFO: last-in, first-out
- Example
  - Interpreter for un-parenthesized arithmetic expressions

# Example

Initially,
Input: 2 * 3 + 5

| | Operator stack: | Operand stack: |
|---|---|---|
| Initially, Input: 2 * 3 + 5 | *empty* | *empty* |
| Input: 2 * 3 + 5 | *empty* | 2 |
| Input: * 3 + 5 | * | 2 |
| Input: 3 + 5 | * | 3 (top of stack), 2 |



# Example

2 * 3 + 5

| | Operator stack: | Operand stack: |
|---|---|---|
| | * | 3 (top of stack), 2 |
| Input: + 5 | + | 6 |
| Input: 5 | + | 5 (top of stack), 6 |
| Input: *empty* | *empty* | **11** |

Answer on top of operand stack

# Example

Initially,
Input: 2 + 3 * 5

Operator stack: *empty*
Operand stack: *empty*

Input: 2 + 3 * 5
Operator stack: *empty*
Operand stack: 2

Input: + 3 * 5
Operator stack: +
Operand stack: 2

Input: 3 * 5
Operator stack: +
Operand stack: 3, 2 (top of stack: 3)

top of stack





# Example

2 + 3 * 5

Operator stack: +
Operand stack: 3, 2

Input: * 5
Operator stack: *, +
Operand stack: 3, 2

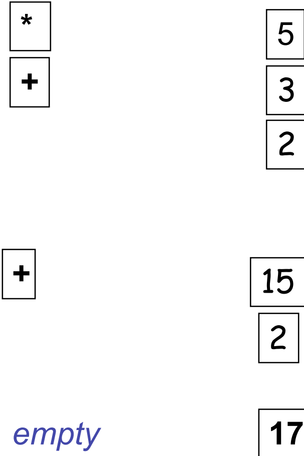
Input: 5
Operator stack: *, +
Operand stack: 5, 3, 2

top of stack

# Example

2 + 3 * 5

Input: empty






# Algorithm

- **When see operator input, compare to top of operator stack.**
  - **If + on stack and + in input, pop 2 operands, evaluate their sum, push result on top of operand stack**
  - **If + on stack and * in input, push operator**
  - **If * on stack and + in input, pop 2 operands, evaluate the product, push result on top of operand stack**
  - **If * on stack and * in input, pop 2 operands, evaluate their product, push result on top of operand stack**
- **Always push operands onto operand stack**
- **When input is empty, evaluate all operators left on stack**
- **Answer is on top of operand stack**

## What's going on?

- Algorithm is enforcing rules of arithmetic, assuming we accumulate sums and products from left to right.
  - If + on stack and + in input, pop 2 operands, evaluate, push result on top of operand stack
    - » 2+3+4 ~ (2+3) + 4
  - If + on stack and * in input, push operator
    - » Matches ? + ? * ?
  - If * on stack and + in input, pop 2 operands, evaluate, push result on top of operand stack
    - » Matches ? * ? + ?
  - If * on stack and * in input, pop 2 operands, evaluate, push result on top of operand stack
    - » 2*3*4 ~ (2*3) * 4



## How are interpreters useful?

- Allow prototyping of new programming languages (PL's)
  - Get to test out PL design quickly
  - E.g., Scheme, Prolog, Java
- A way to achieve portability and universality for a PL
  - Generate code to be interpreted by a Virtual Machine (VM)
  - Can install the PL on a different machine (i.e., chip) merely by rewriting the VM
  - As long as PL definition is carefully written (syntax and semantics), programs should work equivalently!
  - Model for Java (e.g., JVM - Java Virtual Machine)

# Java

- Language definition ~mid-1990's
- Used to write applications built out of pieces (e.g., libraries, components, middleware)
  - Built by different people, in different places, on different machines
  - Works because of VM mechanism
- Interpretation frees user from worries about machine-dependent translation details



# PLs & Compilers: An Incomplete History

- 1950's
  - Machine language programming
  - Scientific computation in Fortran with first compilers
  - LISP for non-numerical computation
- 1960's
  - First optimizing Fortran compiler (IBM)
- 1970's
  - First program analyses designed to enable complex optimizations
  - C language and UNIX (Linux is a form of UNIX)
  - Optimizing for space and time savings

# PLs & Compilers: An Informal History

- 1980's
  - First widely-used object-oriented PLs - Smalltalk, C++
  - Compilers translate for parallel machines (e.g., Thinking Machines, Cray)
  - PLs allowing explicit parallelism (i.e., use of multiple processors; Ada)
- 1990's
  - Birth of the Internet
  - PLs for explicitly distributed computation (e.g., across machines in an network)
  - Object-oriented PLs - Java (VMs)
- 2000's
  - Compiling for low power
  - Special purpose (domain specific) PLs
  - Scalability, distributed computation, ubiquity