

# Compiler Optimizations

# Compiler Optimizations

- ❖ to minimize or maximize some attributes of an executable program
  - Execution time
  - Code size (e.g. IoT devices)
  - Power (e.g. embedded sensors)
  - Compile time may also be an optimization target

# Why Different Optimizations

- Q: *Why small code does not necessary yield better runtime?*
- Q: *Why faster execution does not always consume less power?*
- Q: *Why compile time may be more important than code execution time?*
- Q: *Why not leave performance job to Moore's law?*
- Q: *Why not let AI deep learning take over compiler optimizations?*

# Compiler Optimizations (cont.)

## ❖ Optimization Basics

- Never compromise original program valid behavior
- Optimization is a misnomer
  - Optimal runtime solutions are often un-decidable
  - Many optimization problems are NP-hard
- Compile time and space are limited
  - Especially for Runtime compilation such as JIT
- Some transformations may impede performance for some other cases
- ➔ Heuristic methods,
- ➔ Sub-optimal solutions,
- ➔ Often rely on compiler options/flags controlled by users

# Compiler Optimizations (cont.)

## ❖ Optimization Roadblocks

- Static translation
- Procedure calls
- Separate compilation
- Aliasing
- Debugging needs
- Parallel execution requirement
- Volatile memory

# Optimizations Related Issues

- ❖ Machine independent vs. Machine dependent optimizations
- ❖ Compile time (Static) vs. Runtime (Dynamic) tradeoffs (as in JIT or adaptive optimization systems)
- ❖ Domain specific vs. General purpose optimizations
- ❖ Static vs. Profile driven vs. Runtime optimizations
- ❖ Predictive Heuristics vs. Iterative Optimization
- ❖ Locality vs. Parallelism

# Popular PL and Impact on Opt.

Dec 2017	Dec 2016	Change	Programming Language	Ratings	Change
1	1		Java	13.268%	-4.59%
2	2		C	10.158%	+1.43%
3	3		C++	4.717%	-0.62%
4	4		Python	3.777%	-0.46%
5	6	▲	C#	2.822%	-0.35%
6	8	▲	JavaScript	2.474%	-0.39%
7	5	▼	Visual Basic .NET	2.471%	-0.83%
8	17	▲	R	1.906%	+0.08%
9	7	▼	PHP	1.590%	-1.33%
10	18	▲	MATLAB	1.569%	-0.25%
11	13	▲	Swift	1.566%	-0.57%
12	11	▼	Objective-C	1.497%	-0.83%
13	9	▼	Assembly language	1.471%	-1.07%
14	10	▼	Perl	1.437%	-0.90%
15	12	▼	Ruby	1.424%	-0.72%

# Most Popular PL. (2016-2017)

Indeed		IEEE Spectrum
1	Java	Java
2	JavaScript	C++
3	.NET	Python
4	HTML	C++
5	Python	JavaScript
6	SQL	C#
7	C or C++	PHP
8	Node.js	HTML
9	Ruby	Ruby
10	PHP	Swift

A job search site      Dice and CareerBuilder

# PL Trend Impact

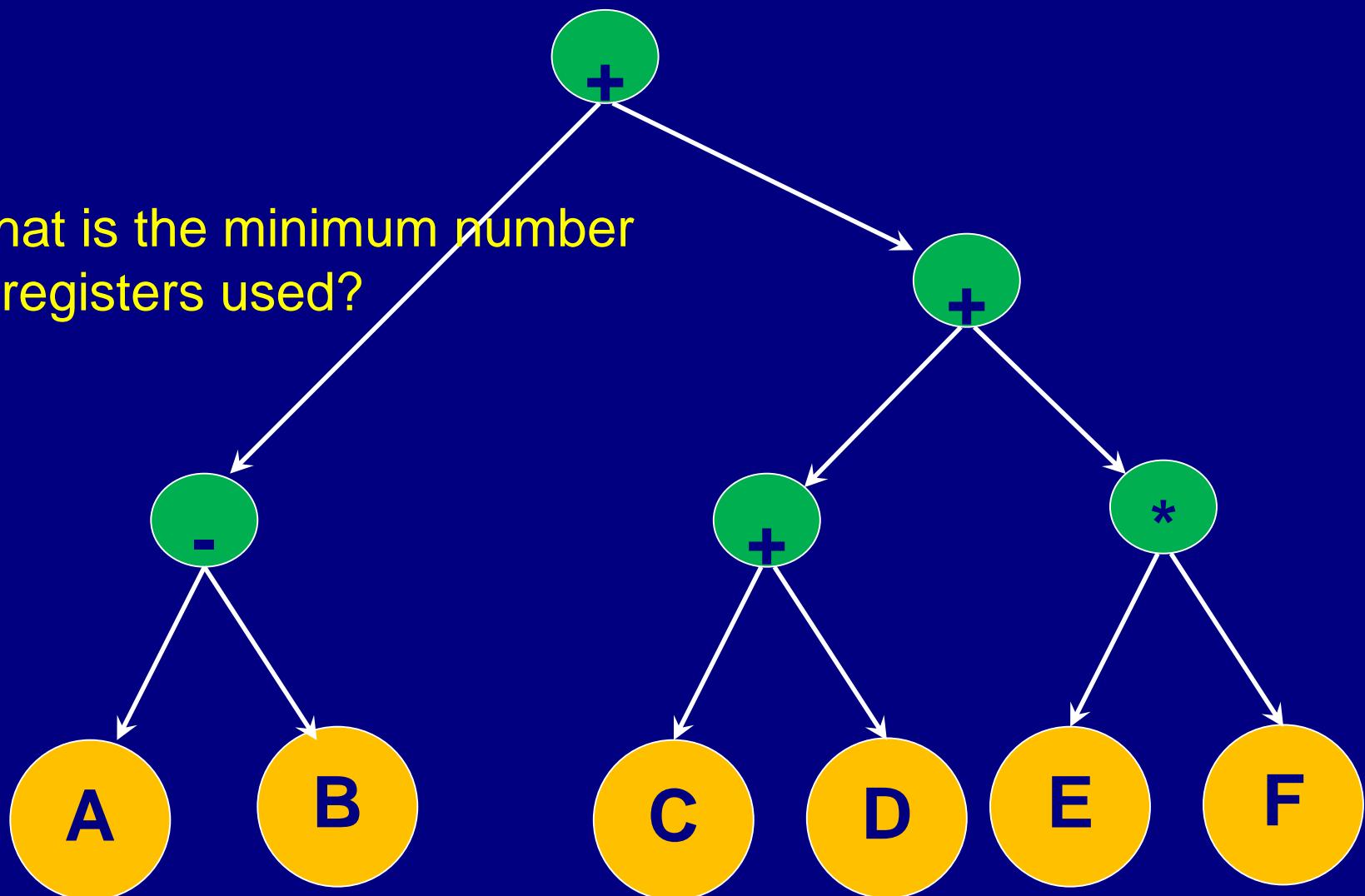
- ❖ More languages require runtime/adaptive translations:  
Java, C#, PHP, Python, JavaScript, Perl,  
Ruby, AJAX
- ❖ More scripting languages:  
Programmers favor languages with simple  
syntax and semantics
- ❖ Optimizations
  - Some traditional: constant folding, loop  
transformations, pointer analysis, ...
  - Some new: type inference, function caching,..

# Some Basic Code Generation Techniques

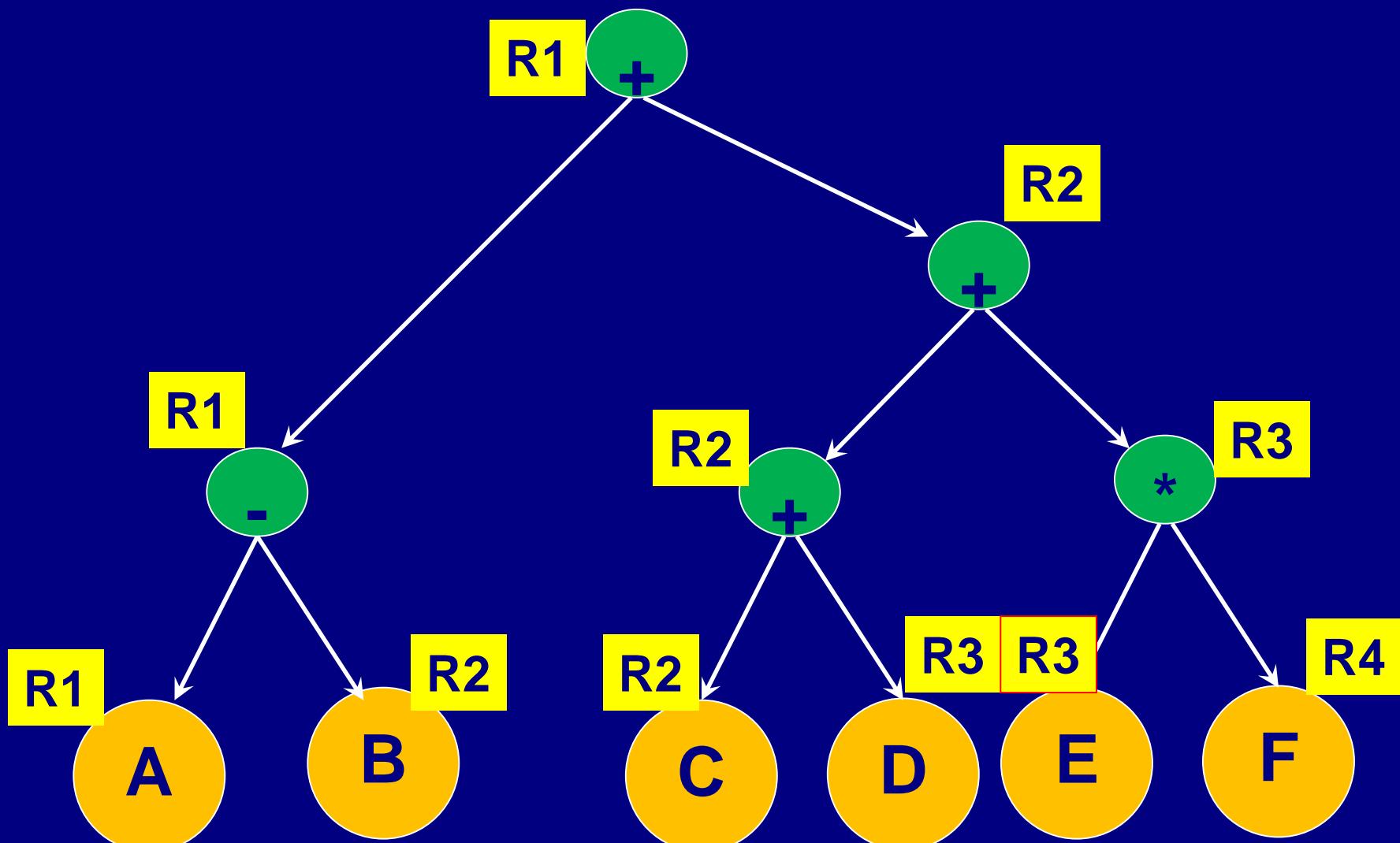
- Sethi-Ullman (or Ershov) Numbering
- Value Numbering (CSE)
- DAG Building
- Automatic Instruction Selection
- Register Allocation: Graph Coloring
- Local Code Scheduling
- Global Code Scheduling

# Translating Expression Trees

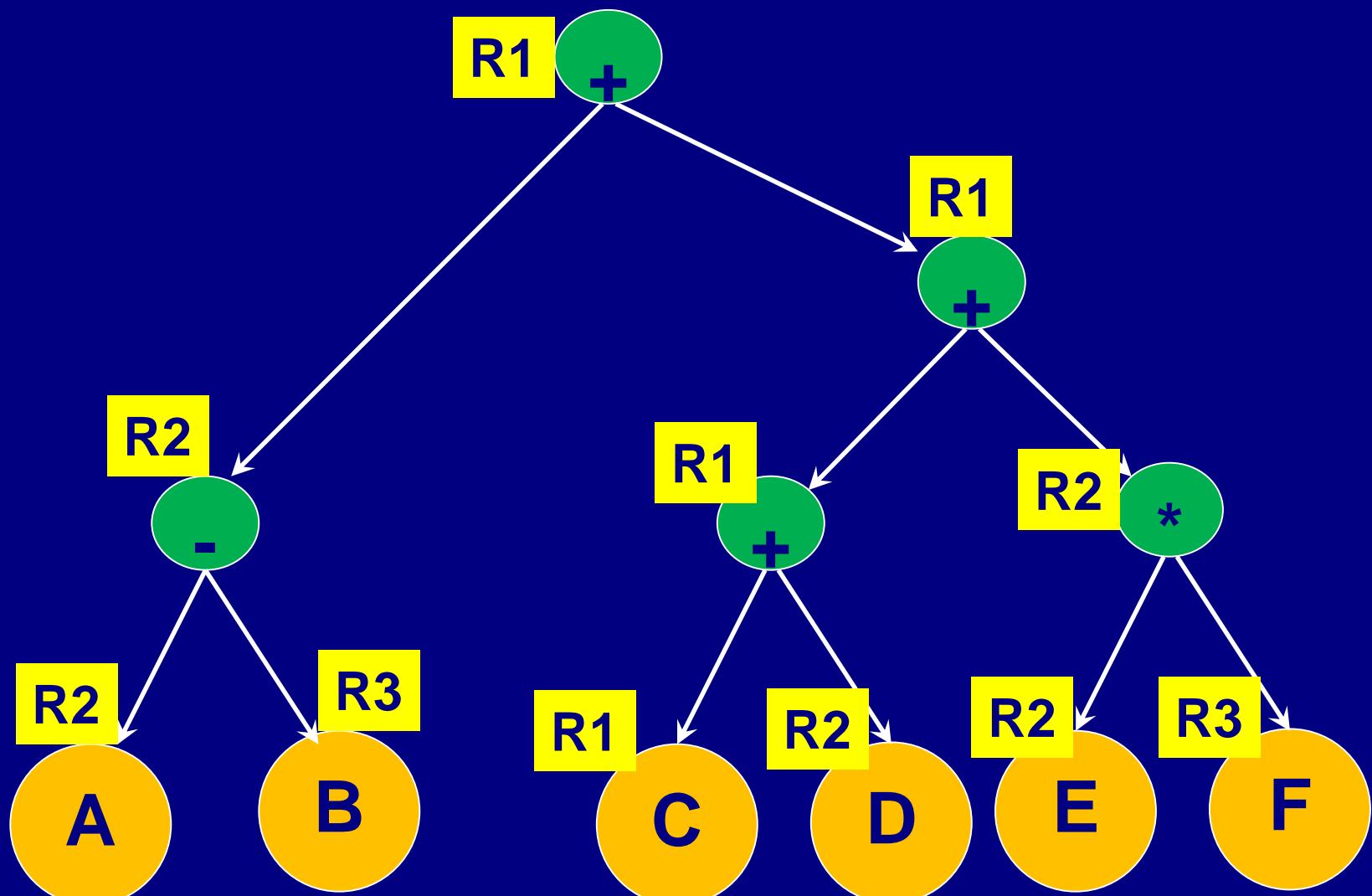
What is the minimum number  
of registers used?



# Translating Expression Trees



# Translating Expression Trees

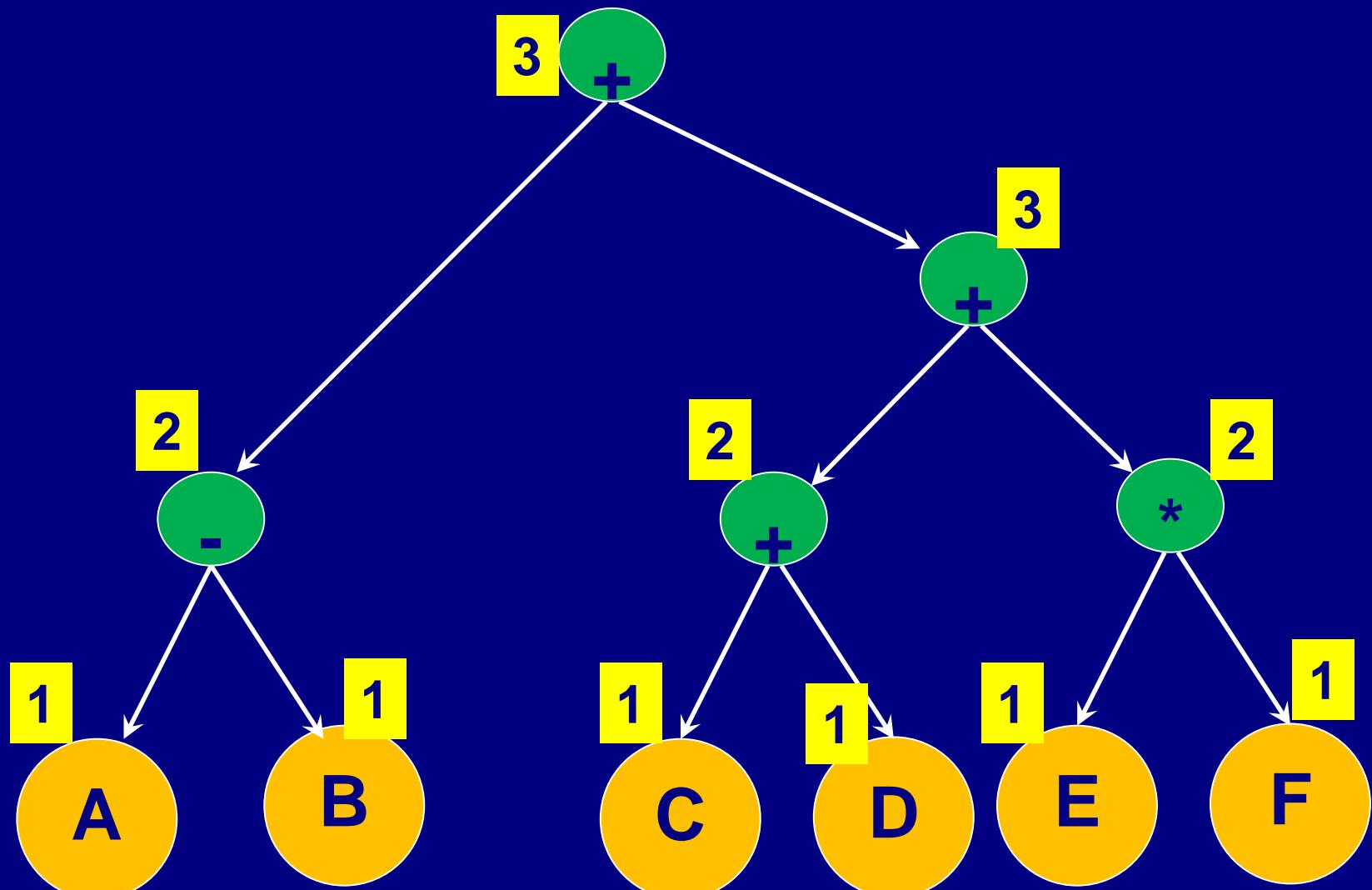


# Sethi-Ullman/Ershov Numbering

Determines the minimum number of registers needed to evaluate any expression **trees**

```
Procedure registerNeeds(T)
If (T.kind == ID or T.kind == const)
    { T.regcount = 1}
else
    {call registerNeeds(T.leftchild);
     call registerNeeds(T.rightchild);
if (T.leftchild.regcount == T.rightchild.regcount)
    { T.regcount = T.rightchild.regcount +1; }
else
    {T.regcount = MAX (T.leftchild.regcount,
     T.rightchild.regcount);}
end
```

# Sethi-Ullman/Ershov Numbering



# Sethi-Ullman Numbering Code Gen

## TreeCG

- We use regcount labeling to drive code gen
- It generates code to evaluate the tree, leaving the result in the first register on the list.
- If TreeCG is given too few registers, it spills a register to the frame (i.e. activation record)
- If left child has a regcount > right child it evaluates left child first, then the right child otherwise, it evaluate the right child, then the left child.

# Other Considerations

*Can the immediate value fit in inst ?*

- ◆ Some machines allow immediate values to be specified in instructions.
- ◆ Some operators are associative, for example,  
 $(a+b) + c == a + (b+c)$   
Applying associative rule, registers could be reduced, for example:  
 $(a+b)+(c+d)$  requires 3 registers  
but  $a+b+c+d$  requires only 2 registers

However, this is not true for FP arithmetic due to rounding issues

# Insufficiently Supply of Registers

- Assume we have  $r$  registers available
- If a node  $N$  with a label  $k$ ,  $k > r$ , we handle each side of the tree separately, and store the result of *larger subtree* (i.e. spill) to AR, and load back the result from AR before node  $N$  is evaluated.

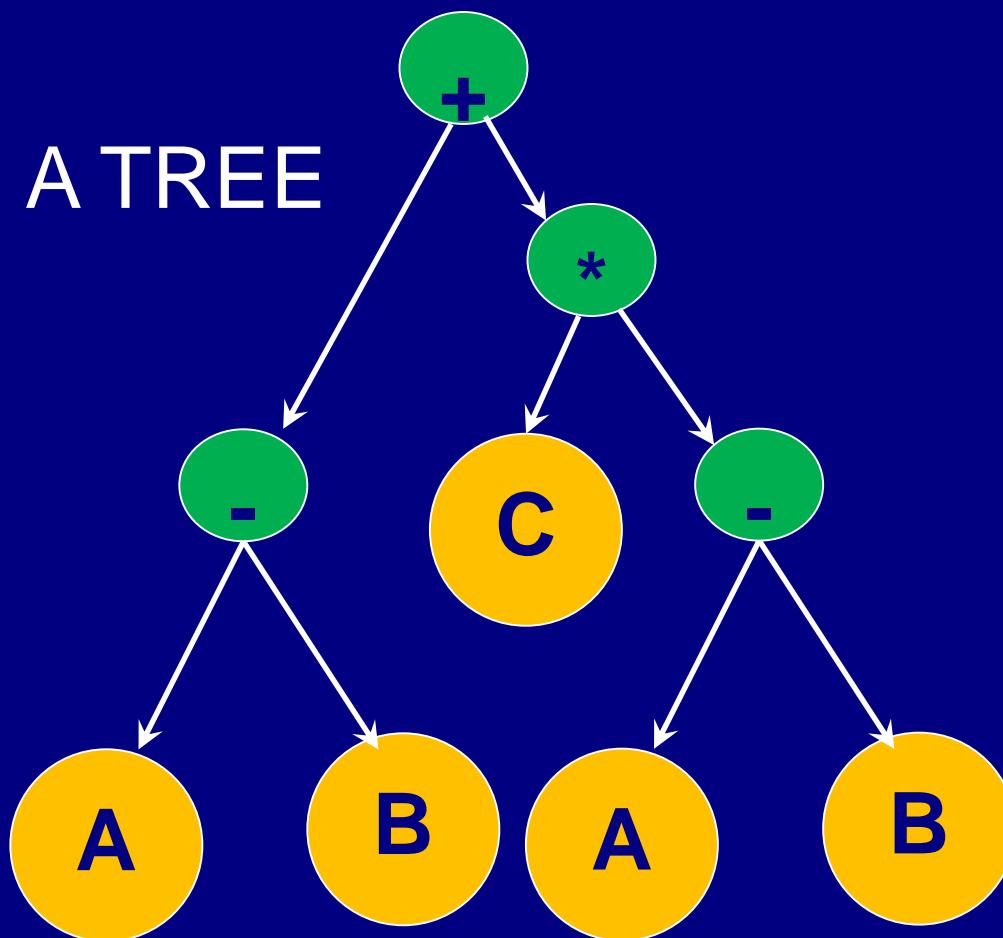
# CSE (Common Sub-Expressions)

- ◆  $(A+B) + (A+B)*C$   
 $(A+B)$  here is called a common sub-expression
- ◆ A CSE can be allocated to a register to avoid redundant computations.
- ◆ An expression tree captures no CSEs, each operand has only one consumer. A CSE has more than one consumers.
- ◆ A DAG (Directed Acyclic Graph) supports CSEs
- ◆ Value numbering is an efficient algorithm to identify CSEs in building a DAG

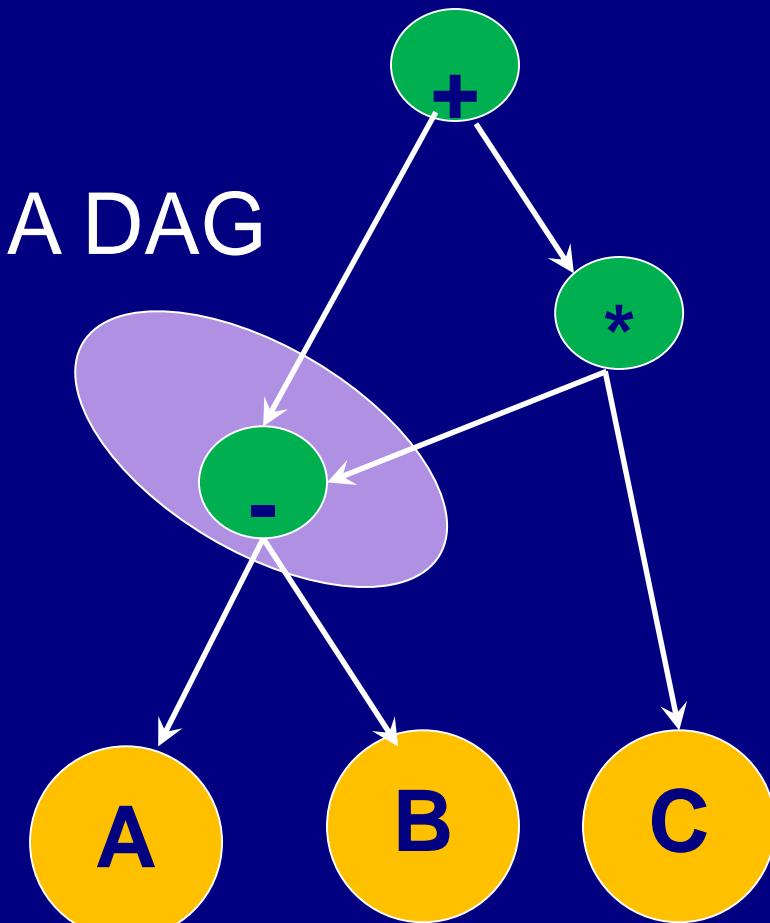
# CSE (Common Sub-Expressions)

◆  $(A-B) + (A-B)^*C$

A TREE



A DAG



# Local Value Numbering

- Each variable and expression is assigned a unique number
- When the compiler sees a variable or expression, check if it is already assigned a number.
  - If so, use the number for that value
  - If not, assign a new number
- Same number means same value (i.e. an index of an array).

# Local Value Numbering

## ■ Example:

$$A = B + C$$

$$D = B + C$$

$$E = B$$

$$F = E + C$$

Numbers assigned

$B \rightarrow \#1$

$C \rightarrow \#2$

$B+C \rightarrow \#1+\#2 \rightarrow \#3$

$A \rightarrow \#3$

$D \rightarrow \#3$

$E \rightarrow \#1$

$E+C \rightarrow \#1+\#2 \rightarrow \#3$

$F \rightarrow \#3$

# Value Numbering Implementation

## ■ Example:

$i = i + j;$

$k = (i + j) * k;$

...

$x = (i + j) * k + 2$

index	type	left	right
1	ID		i
2	ID		j
3	+	1	2
4	=	1	3
5	ID		k
6	*	3	5

Value number means the *index* of the array

# Value Numbering Implementation

## ■ Example:

$i = i + j;$

$k = (i + j) * k;$

...

$x = (i + j) * k + 2$

index	type	left	right
1	ID		i
2	ID		j
3	+	1	2
4	=	1	3
5	ID		k
6	*	3	5

How to locate the expression  $(i + j)$  quickly in the table? Hint: this is a sequential table

Hash – take (+, i, j) and generate a hash value

# Automatic Instruction Selection

## ■ Instruction selection

- Relatively easy for RISC machines
- Complex for CISC machines
  - Wide variety of addressing modes, operands do not need to be in registers, ... etc
- A challenge to retargetable compilers

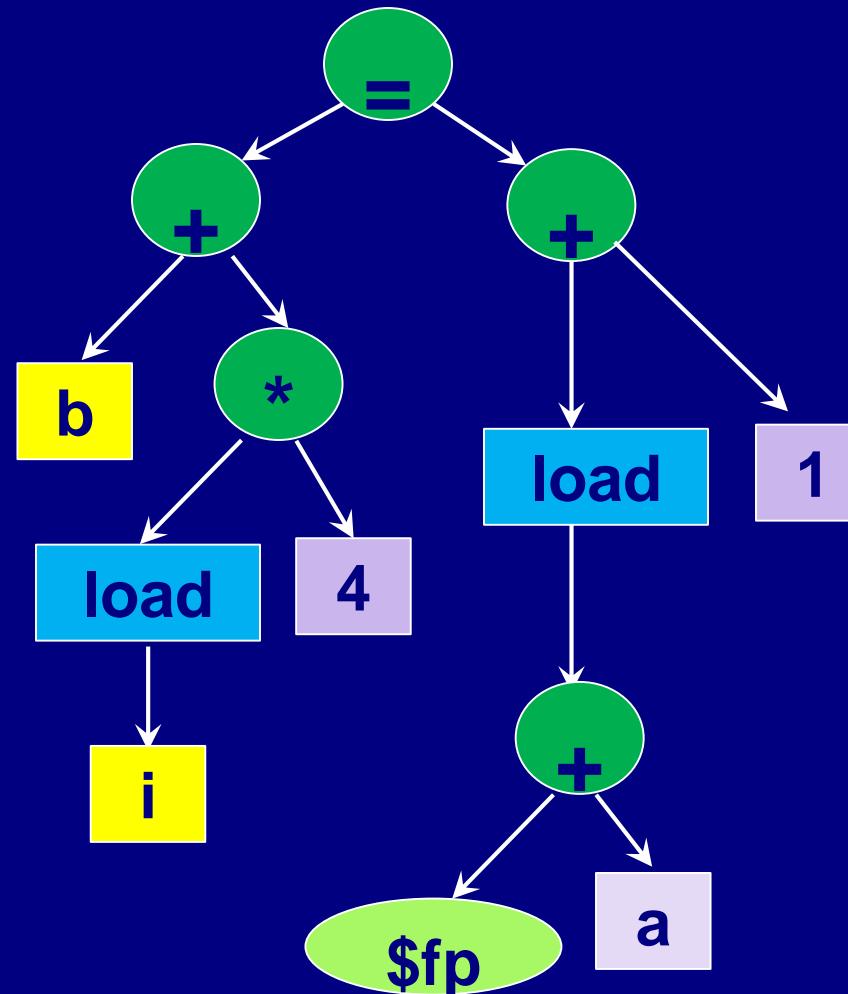
## ■ Idea:

- Generate source code into a low level IR code sequence (down to register level), represented as trees
- Each target machine instruction is also represented as an individual subtree
- Code generation turns into a tree matching problem like parsing

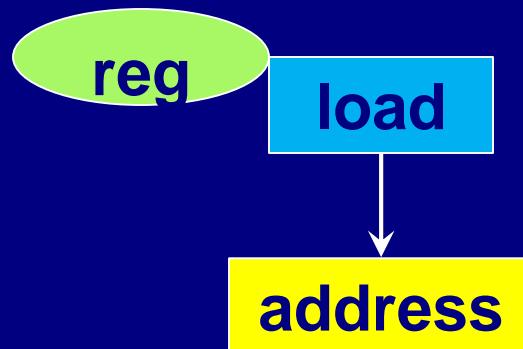
# Automatic Instruction Selection

■ Example  $b[i] = a + 1$

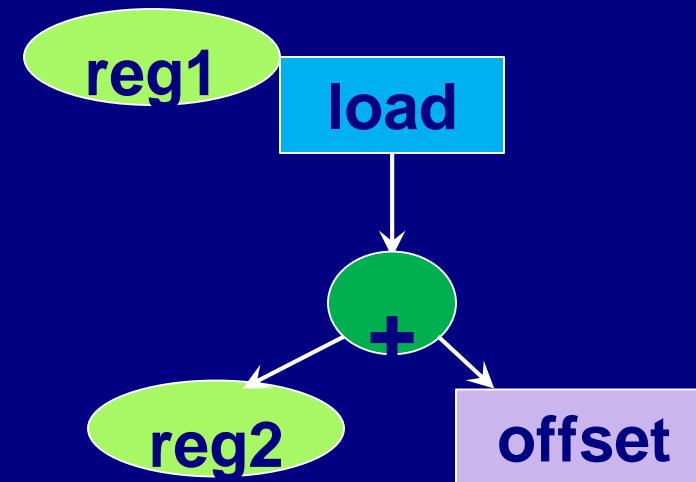
Low level  
IR tree



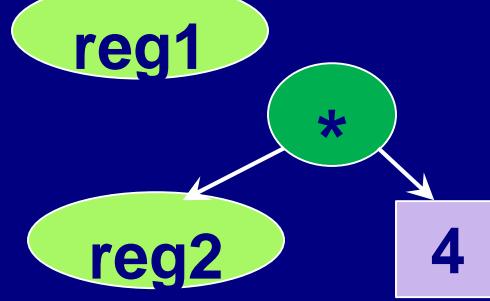
# IR Tree Patterns for MIPS Instructions



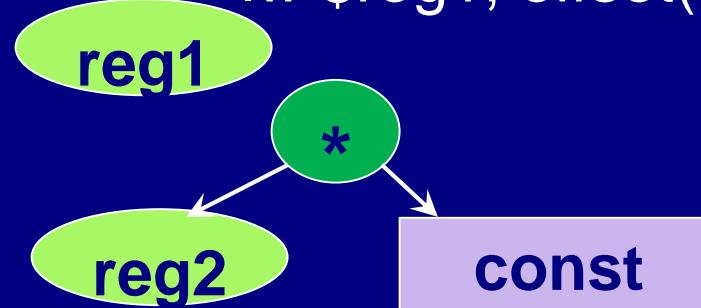
lw \$reg, address



lw \$reg1, offset(\$reg2)

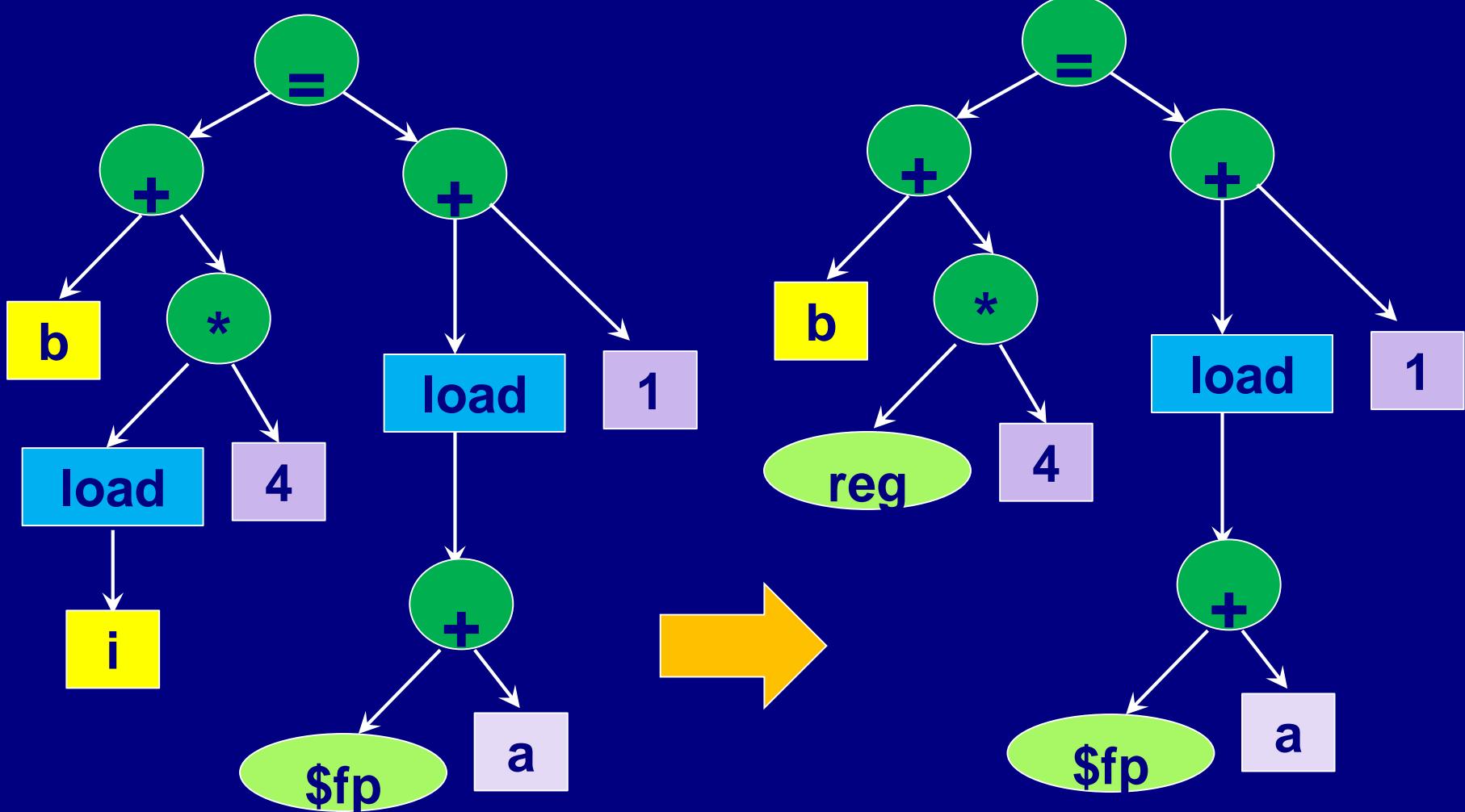


sll \$reg1, \$reg2, 2



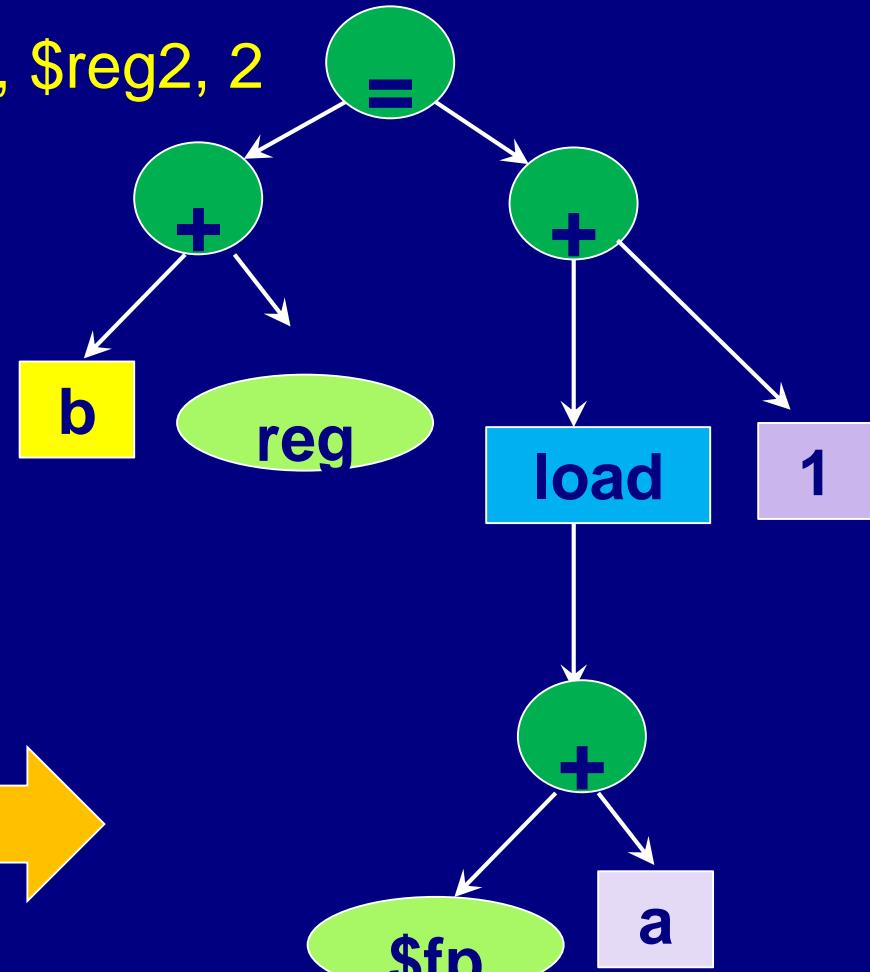
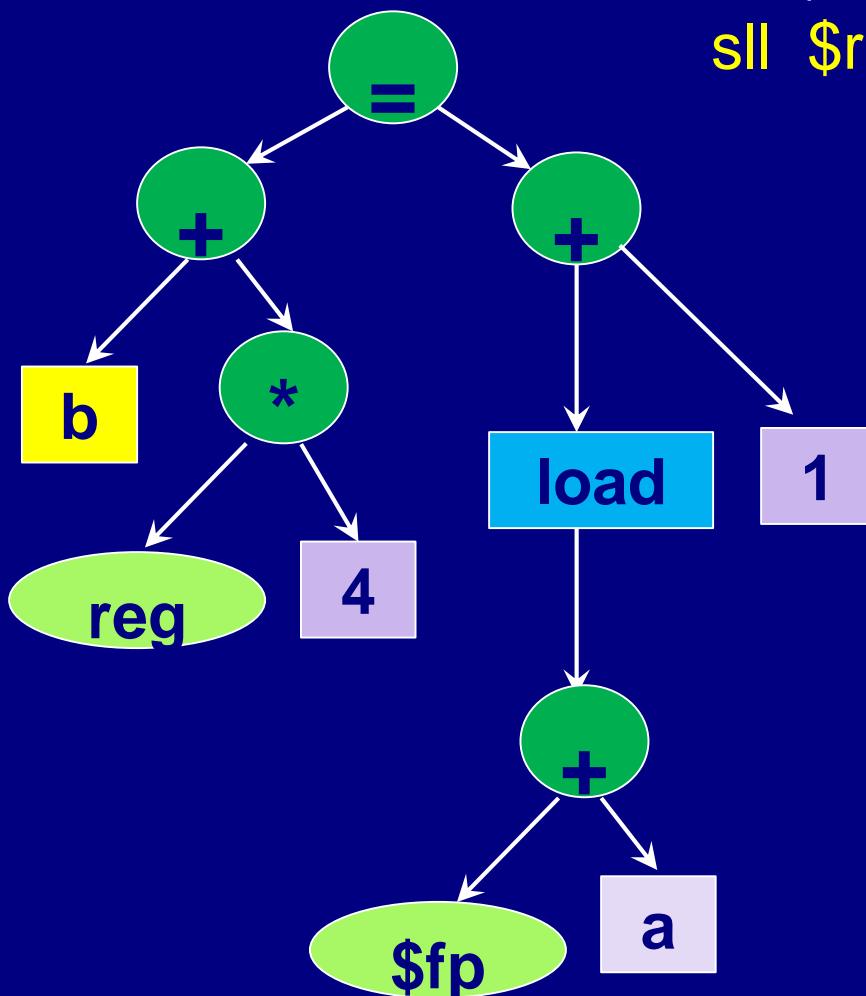
mul \$reg1, \$reg2, const

lw \$reg, i

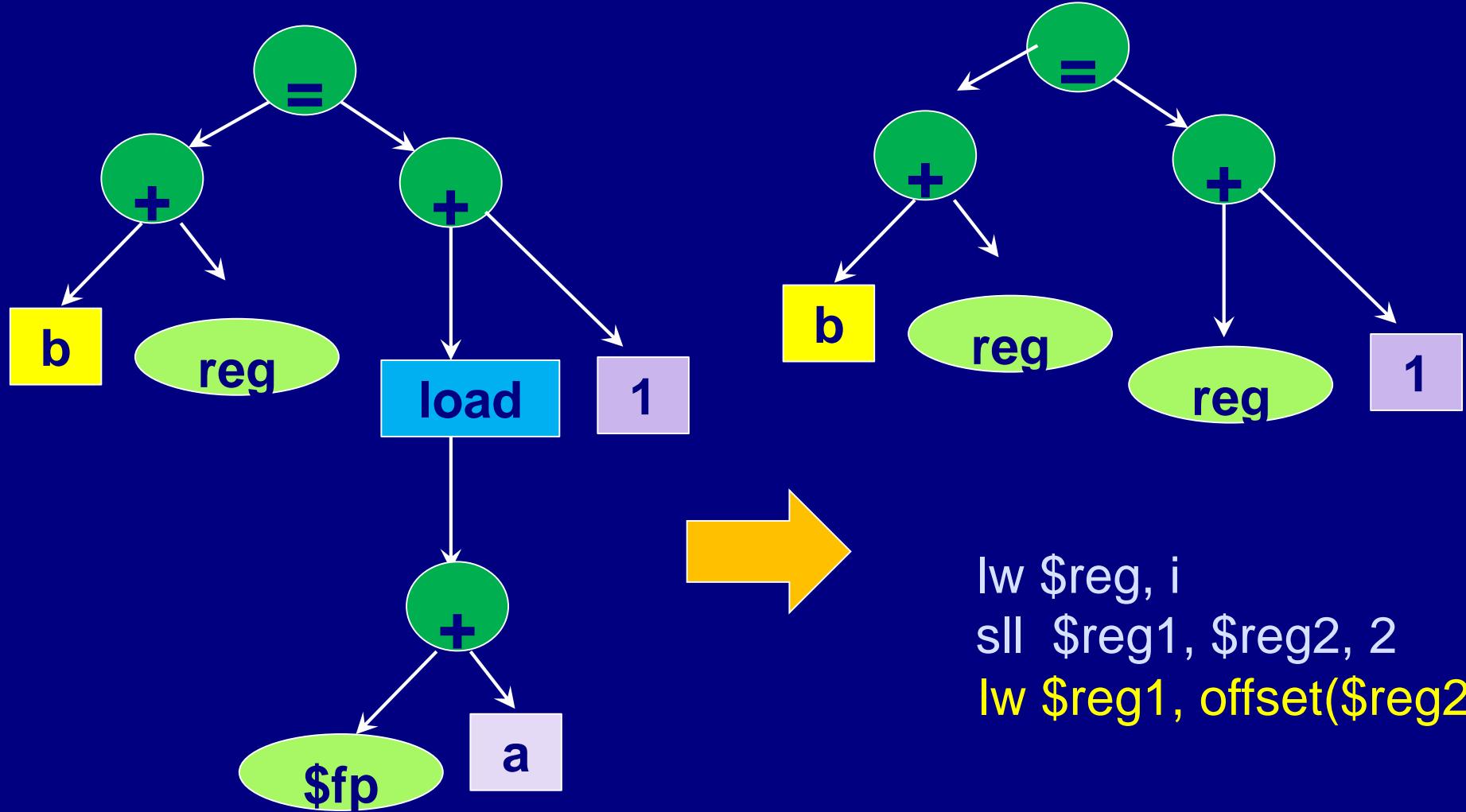


Similar to bottom-up parsing

lw \$reg, i  
sll \$reg1, \$reg2, 2



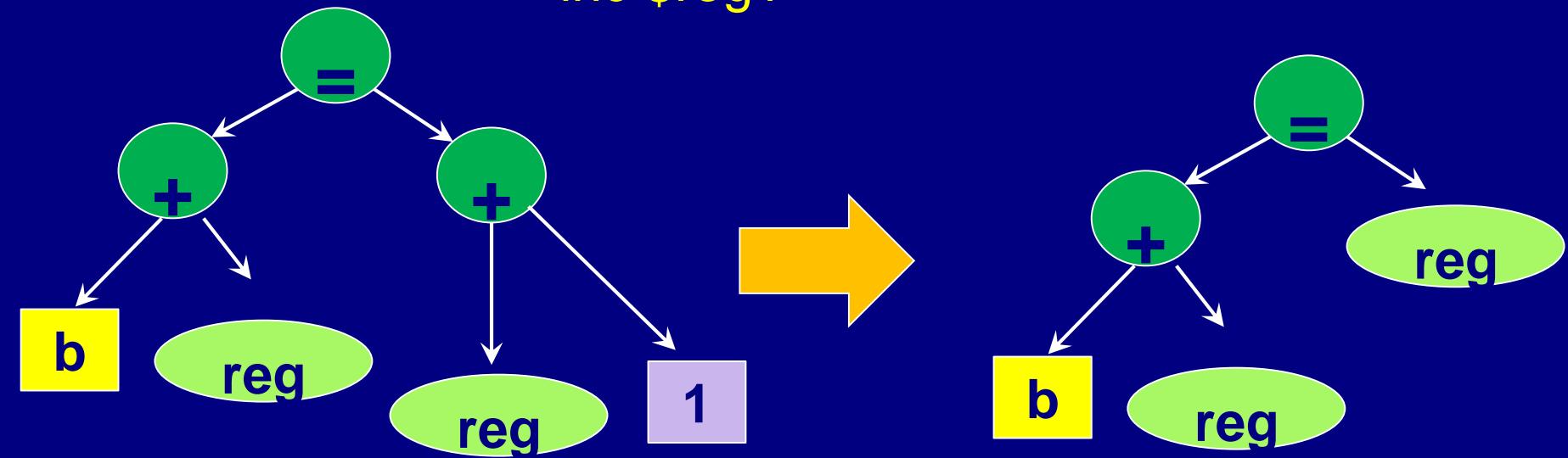
Similar to bottom-up parsing



lw \$reg, i  
sll \$reg1, \$reg2, 2  
lw \$reg1, offset(\$reg2)

Similar to bottom-up parsing

lw \$reg, i  
sll \$reg1, \$reg2, 2  
lw \$reg3, offset(\$reg4)  
inc \$reg1



Similar to bottom-up parsing

# Search for Least-Cost Tree Pattern

- More than one instruction sequence can implement the same construct.
  - In terms of IR trees, different reductions of the same tree yielding different code sequences.
  - So how do we obtain the least expensive code sequence?
- Both BURS (Bottom-Up Rewriting System) and TWIG use dynamic programming to find the least cost cover for a subtree. (see textbook chapter 13.5)