Machine Ops

Most RISC like instruction sets support a generic 3 argument family of binary operators

$$\texttt{OP RD, RX, RY} \equiv \; RD \leftarrow RX \;\; OP \;\; RY$$

Where `RD`, `RX`, `RY` are general purpose registers and not necessarily required to be distinct. For example:

$$\texttt{ADD R4, R7, R0} \equiv \; R4 \leftarrow R7 + R0$$

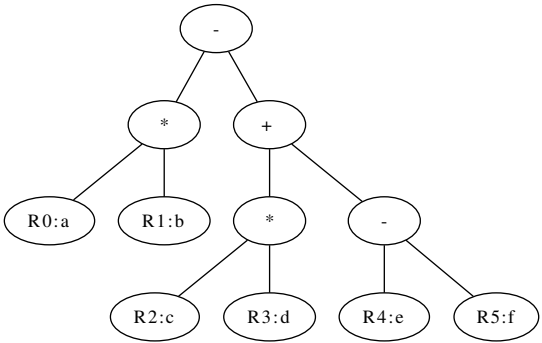
$$\texttt{SUB R10, R10, R3} \equiv \; R10 \leftarrow R10 - R3$$

Naive Register Allocation

Consider this expression tree and the machine operations for its evaluation:

```
LD R0,@a
LD R1,@b
LD R2,@c
LD R3,@d
LD R4,@e
LD R5,@f
MUL R0,R0,R1
MUL R2,R2,R3
SUB R4,R4,R5
ADD R2,R2,R4
SUB R0,R0,R2
```

The leaf values are loaded into six registers, and the arithmetic results are propagated up the tree to the root node.



Naive Register Allocation

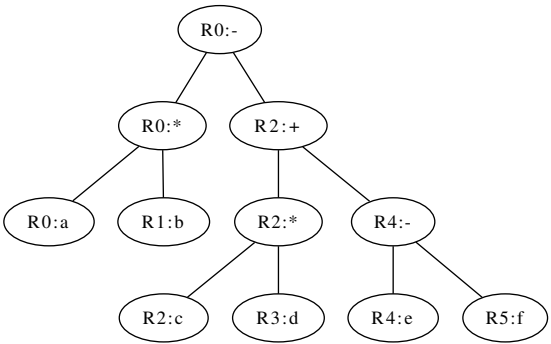
Consider this expression tree and the machine operations for its evaluation:

```
LD R0,@a
LD R1,@b
LD R2,@c
LD R3,@d
LD R4,@e
LD R5,@f
MUL R0,R0,R1
MUL R2,R2,R3
SUB R4,R4,R5
ADD R2,R2,R4
SUB R0,R0,R2
```

The leaf values are loaded into six registers, and the arithmetic results are propagated up the tree to the root node.

The result lands in `R0`.

An inefficient use of registers, we can do the same work with 3.

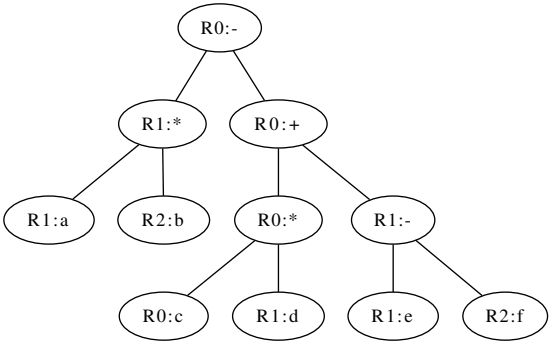


Sethi-Ullman Register Counting

The important observation is that we don't need to load all the registers in one fell swoop. We should work up the tree from the leaves, calculating the "largest" subtree first.

```
LD R0,@c
LD R1,@d
MUL R0,R0,R1
LD R1,@e
LD R2,@f
SUB R1,R1,R2
ADD R0,R0,R1
LD R1,@a
LD R2,@b
MUL R1,R1,R2
SUB R0,R1,R0
```

The right subtree required three registers, we wisely calculated it first, saving the result in `R0`.



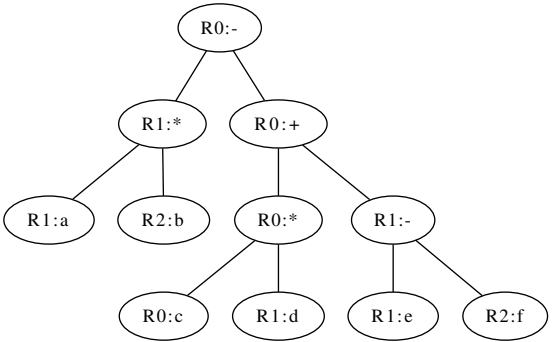
Sethi-Ullman Register Counting

The important observation is that we don't need to load all the registers in one fell swoop. We should work up the tree from the leaves, calculating the "largest" subtree first.

```
LD R0,@c
LD R1,@d
MUL R0,R0,R1
LD R1,@e
LD R2,@f
SUB R1,R1,R2
ADD R0,R0,R1
LD R1,@a
LD R2,@b
MUL R1,R1,R2
SUB R0,R1,R0
```

The right subtree required three registers, we wisely calculated it first, saving the result in `R0`.

Then we reused `R1` and `R2` on the left subtree.

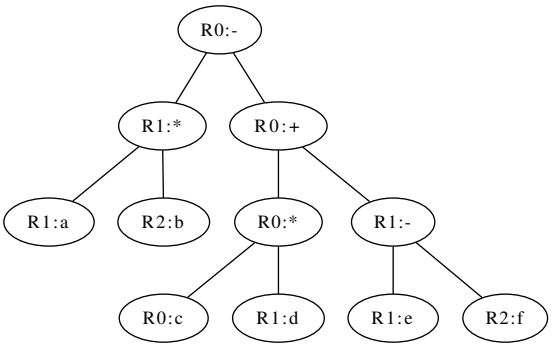


Sethi-Ullman Register Counting

The important observation is that we don't need to load all the registers in one fell swoop. We should work up the tree from the leaves, calculating the "largest" subtree first.

```
LD R0,@c
LD R1,@d
MUL R0,R0,R1
LD R1,@e
LD R2,@f
SUB R1,R1,R2
ADD R0,R0,R1
LD R1,@a
LD R2,@b
MUL R1,R1,R2
SUB R0,R1,R0
```

What is our metric or notion for "larger" subtrees?



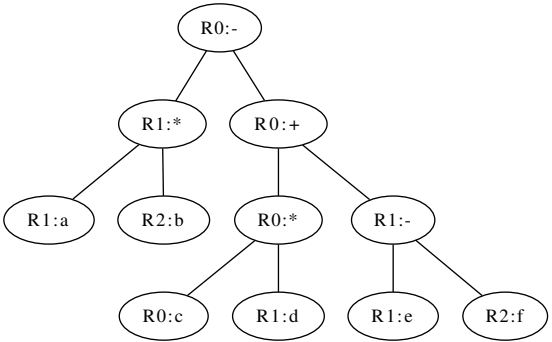
Sethi-Ullman Register Counting

The important observation is that we don't need to load all the registers in one fell swoop. We should work up the tree from the leaves, calculating the "largest" subtree first.

```
LD R0,@c
LD R1,@d
MUL R0,R0,R1
LD R1,@e
LD R2,@f
SUB R1,R1,R2
ADD R0,R0,R1
LD R1,@a
LD R2,@b
MUL R1,R1,R2
SUB R0,R1,R0
```

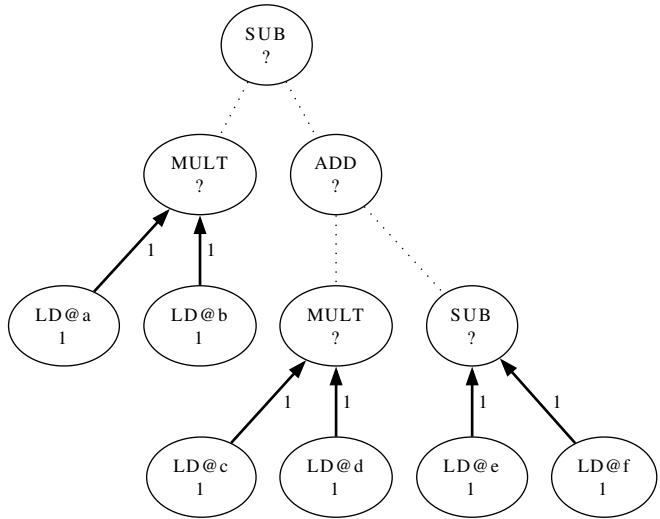
What is our metric or notion for "larger" subtrees?

How do we determine the number of registers needed for an arbitrary tree?



Sethi-Ullman Register Counting Example

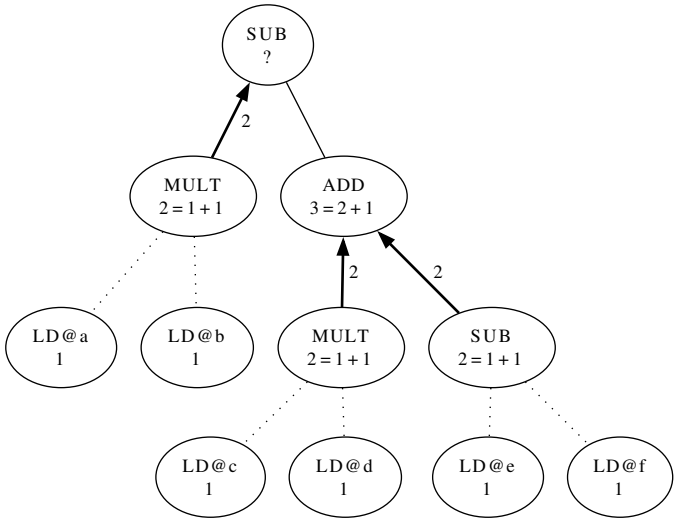
Begin at the leaves of the tree, which are **usually** `LOAD` or immediate operations requiring 1 register.



Sethi-Ullman Register Counting Example

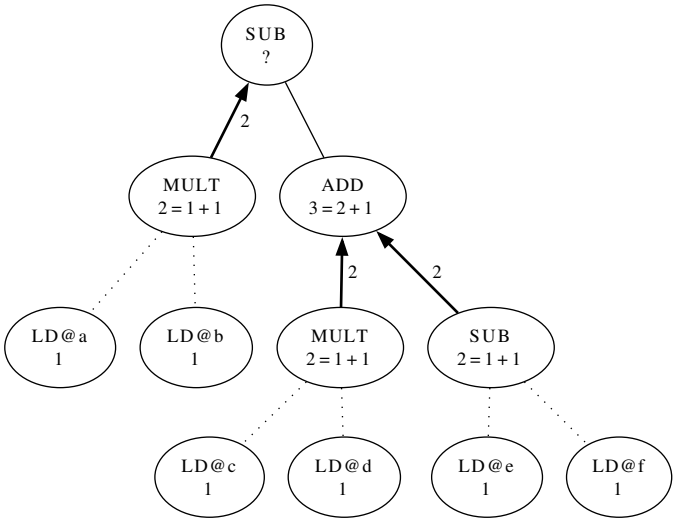
When both left and right trees **require the same number of registers**, there are two cases:

i. If they each require 1 register, we can calculate the result in 2 (eg: the two MULT operations of the tree)



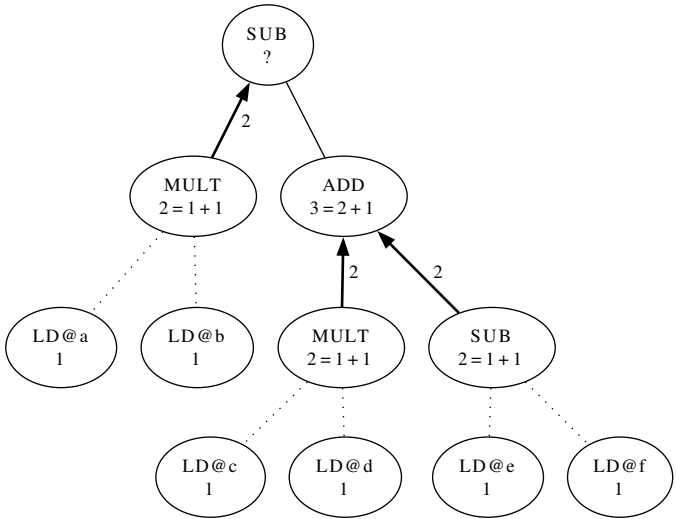
Sethi-Ullman Register Counting Example

ii. If they each require $n > 1$ (but still the same for both trees), then we need $n + 1$ registers for the result.
Consider the ADD operation, it can't be computed with 2 registers, because you need an extra register to hold the LHS MULT result while the RHS SUB is calculated.



Sethi-Ullman Register Counting Example

The two cases for when **both trees require the same number of registers (n)** have the same result: $n+1$ registers are required.



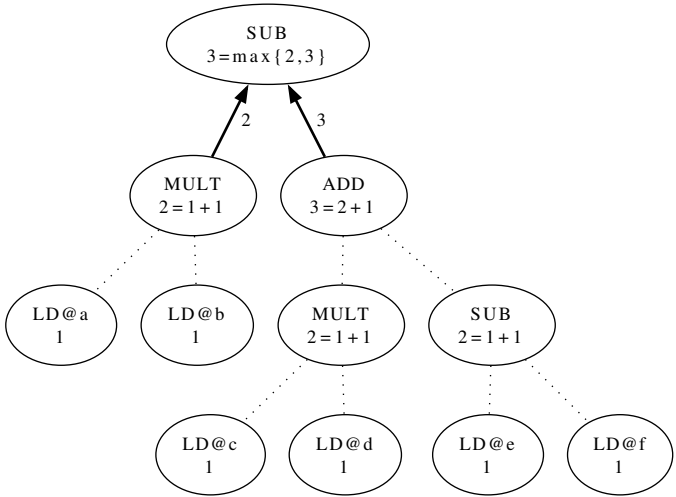
Sethi-Ullman Register Counting Example

When the two subtrees **require a different number of registers (n_{LHS},n_{RHS})**, the tree result can be computed in

$$\max\{n_{LHS},\ n_{RHS}\}$$

registers.

Compute the “larger” subtree first, and you’ll have enough registers remaining to compute the “smaller” sibling.



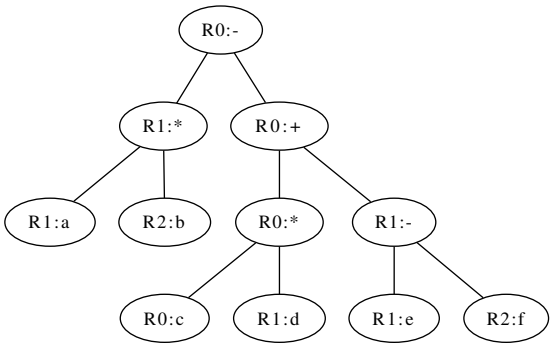
Register Counting → Code Generation (PREVIEW)

During code generation we'll provide an expression tree with a list of **allocatable registers**, eg:

$$\{R0, R1, R2\}$$

and code for calcuating the "larger" (measured by **register count**) tree must be generated by the compiler (executed by the program) first.

So it is wise to retain **register counts** in our expression trees as they are calculated. So the *code generation* visitor pass knows the order to emit each subtree's code.

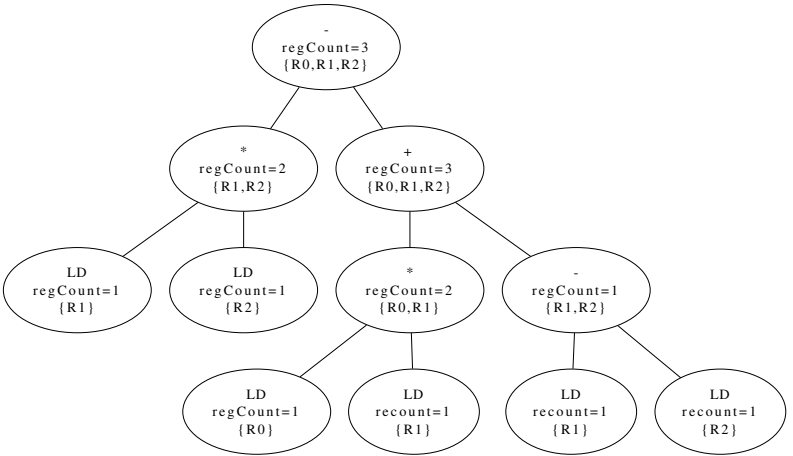


Operation **precedence** is still maintained because it is "encoded" quite naturally by the node's depth in the expression tree.

Register Counting → Code Generation (PREVIEW)

Notice how the result of each node lands in the first register of its register list.
This is a useful feature called **register targeting**.

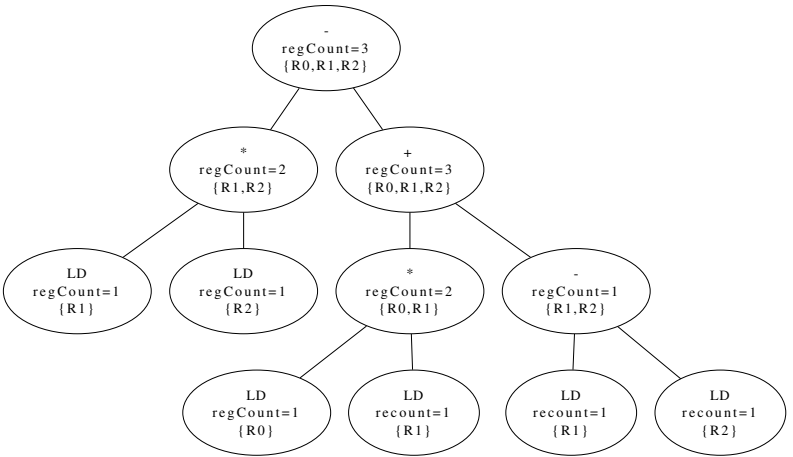
```
LD R0,@c
LD R1,@d
MUL R0,R0,R1
LD R1,@e
LD R2,@f
SUB R1,R1,R2
ADD R0,R0,R1
LD R1,@a
LD R2,@b
MUL R1,R1,R2
SUB R0,R1,R0
```



Register Counting → Code Generation (PREVIEW)

Notice also that the instruction sequence for this expression is ordered by a **post-order** (like) traversal of the expression tree and *processing children by descending **register count*** values.

```
LD R0,@c
LD R1,@d
MUL R0,R0,R1
LD R1,@e
LD R2,@f
SUB R1,R1,R2
ADD R0,R0,R1
LD R1,@a
LD R2,@b
MUL R1,R1,R2
SUB R0,R1,R0
```



`registerNeeds(`*exprTree*`)` — Simple Ops

```
registerNeeds( T )
Calculates the registers required for expression tree T result,
stores value at T.regCount and returns same.

if ( isLeafNode( T ) ) then (
    if ( T.kind is Identifier ) T.regCount ← 1
    if ( T.kind is IntegerLiteral ) T.regCount ← 1
    if ( T.kind is FloatLiteral ) T.regCount ← 1
    ⋮
) else (
    if ( T.kind is BinaryOp ) then (
        registerNeeds( T.leftTree )
        registerNeeds( T.rightTree )
        if ( T.leftTree.regCount = T.rightTree.regCount ) then (
            T.regCount ← T.leftTree.regCount + 1 // Nothin' special about .leftTree
        ) else (
            T.regCount ← max{ T.leftTree.regCount, T.rightTree.regCount }
        )
    ) else if ( T.kind is FunctionCall ) then (
```

`registerNeeds(`*exprTree*`)` — Function Example

```
paramFunction(  x,  m*n+p*q,  B(x+y),  (n+p)*(x-y),  y  )
```

`registerNeeds(`*exprTree*`)` — Function Example

Begin by determining the number of registers required for each parameter expression p_i

	p_1	p_2	p_3	p_4	p_5		param enumeration
`paramFunction(`	`x,`	`m*n+p*q,`	`B(x+y),`	`(n+p)*(x-y),`	`y`	`)`	
	1	3	2	3	1		`registerNeeds(`p_i`)`

`registerNeeds(`*exprTree*`)` — Function Example

Begin by determining the number of registers required for each parameter expression p_i

	p_1	p_2	p_3	p_4	p_5	
`paramFunction(`	`x,`	`m*n+p*q,`	`B(x+y),`	`(n+p)*(x-y),`	`y`	`)`
	1	3	2	3	1	

param enumeration

`registerNeeds(`p_i`)`

Sort by register counts, in descending order, remembering p_i (a stable sort is not required)

p_2	p_4	p_3	p_5	p_1
3	3	2	1	1

Let R be the total number of registers needed to evaluate `paramFunction`, and *rem* be the remaining number of registers available to calculate all remaining parameter values.

$$R = rem = 3$$

`registerNeeds(`*exprTree*`)` — Function Example

p_2	p_4	p_3	p_5	p_1
3	3	2	1	1

$R = rem = 3$

a. We calculate p_2 first, but need to store its result in a register, so now $rem = 2$

`registerNeeds(`*exprTree*`)` — Function Example

p_2	p_4	p_3	p_5	p_1
3	3	2	1	1

$$R = rem = 3$$

a. We calculate p_2 first, but need to store its result in a register, so now $rem = 2$

b. With $rem = 2$, we don't have enough registers to calculate $p_4 \Rightarrow$ increase R and rem sufficiently.

$$R = R + 1 = 4 \qquad rem = rem + 1 = 3$$

Now we can calculate p_4, storing the result into a register leaves $rem = 2$

`registerNeeds(`*exprTree*`)` — Function Example

p_2	p_4	p_3	p_5	p_1
3	3	2	1	1

$R = rem = 3$

a. We calculate p_2 first, but need to store its result in a register, so now $rem = 2$
b. With $rem = 2$, we don't have enough registers to calculate $p_4 \Rightarrow$ increase R and rem sufficiently.

$$R = R + 1 = 4 \qquad rem = rem + 1 = 3$$

Now we can calculate p_4, storing the result into a register leaves $rem = 2$

c. We calculate p_3, storing the result into a register, $rem = 1$

`registerNeeds(`*exprTree*`)` — Function Example

p_2	p_4	p_3	p_5	p_1
3	3	2	1	1

$R = rem = 3$

a. We calculate p_2 first, but need to store its result in a register, so now $rem = 2$
b. With $rem = 2$, we don't have enough registers to calculate $p_4 \Rightarrow$ increase R and rem sufficiently.

$$R = R + 1 = 4 \qquad rem = rem + 1 = 3$$

Now we can calculate p_4, storing the result into a register leaves $rem = 2$

c. We calculate p_3, storing the result into a register, $rem = 1$
d. We calculate p_5, storing the result into a register, $rem = 0$

`registerNeeds(`*exprTree*`)` — Function Example

p_2	p_4	p_3	p_5	p_1
3	3	2	1	1

$R = rem = 3$

a. We calculate p_2 first, but need to store its result in a register, so now $rem = 2$
b. With $rem = 2$, we don't have enough registers to calculate $p_4 \Rightarrow$ increase R and rem sufficiently.

$$R = R + 1 = 4 \qquad rem = rem + 1 = 3$$

Now we can calculate p_4, storing the result into a register leaves $rem = 2$
c. We calculate p_3, storing the result into a register, $rem = 1$
d. We calculate p_5, storing the result into a register, $rem = 0$
e. With $rem = 0$, we can't calculate $p_1 \Rightarrow$ increase R and rem sufficiently.

$$R = R + 1 = 5 \qquad rem = rem + 1 = 1$$

Now we can calculate p_1, the result gets stored into a register.

We'll need 5 registers to calculate the inputs for `paramFunction`.

`registerNeeds(`*exprTree*`)` — Functions

The previous example may need some adjustment for function calling semantics of the target platform, and does not **directly translate** (as in the case of simple binary operations) to the code generation phase.

a. $p > 0$? Arguments passed in registers? Decrement the parameter expression's *regNeeds* by one. Why? At code generation time we'll use **register targeting** by prefixing the *regList* with the appropriate register for parameter passing.

b. At code generation time, we won't traverse the (non register passed) parameter trees in descending *regNeeds* order. Traversal will be dictated by the order arguments should be pushed onto the stack. We'll `PUSH` the head register of *regList* after each parameter is calculated.

registerNeeds.pdf

Tempting but Flawed Optimization

$(a-b)+(c-d)$ will require 3 registers, but

$a-b+c-d$ will require only 2

. . . **and** they are mathematically equivalent. So is this our first optimization for the course?

Tempting but Flawed Optimization

$(a-b)+(c-d)$ will require 3 registers, but

$a-b+c-d$ will require only 2

. . . **and** they are mathematically equivalent. So is this our first optimization for the course?

No: Parenthetical groupings must be respected in expression trees! These are the first (and easiest) tools for the programmer concerned with the creep of numerical errors into critical calculations.

Tempting but Flawed Optimization

$(a-b)+(c-d)$ will require 3 registers, but

$a-b+c-d$ will require only 2

. . . **and** they are mathematically equivalent. So is this our first optimization for the course?

No: Parenthetical groupings must be respected in expression trees! These are the first (and easiest) tools for the programmer concerned with the creep of numerical errors into critical calculations.

You **are** allowed to flip commutative arguments to take advantage of **immediate** operations:

$(1+a) \equiv (a+1)$ or `++i`

These can be efficiently implemented with "immediate arguments" in many machine codes

```
ADD R0,R0,#1
```

Register Groups — bird's eye view

CPUs, operating systems, and executable formats often use **"reserved" registers** for communicating critical values between various components in a computer. Common conventions are returning system call values in a particular register, registers for return values, registers for parameter passing between libraries, as well as hardware configuration and monitoring. We shouldn't use these for expression trees... The compiler writer doesn't have much control over these.

We carve out a small number of general purpose registers we can use (2 or 3) and call them **working registers**, these are always available for use in expression tree generation.

The remaining registers we call **"allocatable"**, meaning their use should be carefully orchestrated during target code generation.

Register Lists for Expression Trees

Our register allocation and code generation algorithm for expression trees (`treeCG`) takes arguments representing the root node of the expression tree, and a list of **allocatable registers**.

Which registers to use? A moderately sophisticated compiler might want some registers for frequently used variables within a single procedure — judicious use of registers like this can boost performance. For this course and our immediate purposes (CZAR!) we'll simply use all the allocatable registers (recall these do not include **work registers**). This simple per-expression approach is called **on-the-fly** register allocation.

What happens if we have really big expression trees and we run out of registers? We use temporary locations in memory (the stack) where we can squirrel away results for expression subtrees until we need them again.

These are the $getTemp()$ and $freeTemp()$ `treeCG` methods in the text's algorithm (figure 13.12); whereas `my version of the algorithm` uses explicit `PUSH`s and `POP`s for SP manipulation.

Register Targeting in `treeCG`

(CG short for Code Generation)

The `treeCG` algorithm has a particularly useful property called **register targeting**, which means the **first register** in the allocatable register list will be the **register holding the result of the expression**.

This **could**[1] be useful when the architecture demands some set of initial function call parameters be passed in registers, such as params one and two of `aFunction(x+3,x/z+2*y,++x)` are passed in R0 and R1.

[1] Gremlins and dragons await naive implementations...

treeCG(*exprTree*, *regList*)

my version of the algorithm

Some of you may prefer the text's pseudo code listings, in which case here is a small "Rosetta Stone" for the TREECG of figure 13.12.

- $head(\cdot)$ and $tail(\cdot)$ as described in my version of the algorithm
- $getTemp()$ and $freeTemp()$ allocate and deallocate space on a stack (probably **the Stack** of the runtime environment, but that's not required per se)
- li, lw and sw are MIPS instructions for LOAD IMMEDIATE, LOAD WORD and STORE WORD
- and GENERATE(·) emits or stores machine instructions for an operation

Alternative Register Lists

An alternative approach to using $getTemp()$ and $freeTemp()$ is to implement a more complex register list elements: a `Register` object that might be either a hardware register or temporary memory location (think of a `Register` base class and subclasses `HardwareReg`, `VirtualReg`).

- `HardwareReg` has a method to generate LD and ST codes without SP "backing"
- `VirtualReg`s load and store onto stack locations, using **working registers** to remain independent of `treeCG` logic.

If there are insufficient **allocatable** registers for an expression tree, then $regList$ is augmented by `VirtualReg`s, and the SP can be modified only twice **per tree evaluation** (PUSH'd at the beginning, POP'd at the end) and virtual reg values on the stack use (SP,Offset) access (much like function parameter access via FP). `treeCG` requires SP manipulations each time a temporary register is needed.

`The evaluation logic` becomes marginally simpler (because complexity has been pushed to register objects).