Improved Local Register Allocation

- Scope
  - Evaluating an arithmetic expression.
  - No reordering of AST using commutative or associative properties.
  - No common subexpression elimination.
- Example: `(A–B)+((C+D)+(E*F))`.
- How many temporaries does it take to generate 3-address code for this expression?

Improved Local Register Allocation

- Scope
  - Evaluating an arithmetic expression.
  - No reordering of AST using commutative or associative properties.
  - No common subexpression elimination.
- Example: `(A-B)+((C+D)+(E*F))`.
- How many temporaries does it take to generate 3-address code for this expression?

```
T1  := A
T2  := B
T3  := T1-T2
T4  := C
T5  := D
T6  := T4+T5
T7  := E
T8  := F
T9  := T7*T8
T10  := T6+T9
T11  := T3+T10
```

Improved Local Register Allocation

- Scope
  - Evaluating an arithmetic expression.
  - No reordering of AST using commutative or associative properties.
  - No common subexpression elimination.
- Example: `(A-B)+((C+D)+(E*F))`.
- How many temporaries does it take to generate 3-address code for this expression?

```
T1  := A
T2  := B
T3  := T1-T2
T4  := C
T5  := D
T6  := T4+T5
T7  := E
T8  := F
T9  := T7*T8
T10  := T6+T9
T11  := T3+T10
```

```
T1  := A
T2  := B
T1  := T1-T2
T3  := C
T4  := D
T3  := T3+T4
T5  := E
T6  := F
T5  := T5*T6
T3  := T3+T5
T1  := T1+T3
```

Improved Local Register Allocation

- Scope
  - Evaluating an arithmetic expression.
  - No reordering of AST using commutative or associative properties.
  - No common subexpression elimination.
- Example: `(A-B)+((C+D)+(E*F))`.
- How many temporaries does it take to generate 3-address code for this expression?

```
T1  := A
T2  := B
T3  := T1-T2
T4  := C
T5  := D
T6  := T4+T5
T7  := E
T8  := F
T9  := T7*T8
T10  := T6+T9
T11  := T3+T10
```

```
T1  := A
T2  := B
T1  := T1-T2
T3  := C
T4  := D
T3  := T3+T4
T5  := E
T6  := F
T5  := T5*T6
T3  := T3+T5
T1  := T1+T3
```

```
T1  := C
T2  := D
T1  := T1+T2
T2  := E
T3  := F
T2  := T2*T3
T1  := T1+T2
T2  := A
T3  := B
T2  := T2-T3
T2  := T2+T1
```

Sethi-Ullman Numbering Algorithm

- [Ref: "The Generation of Optimal Code for Arithmetic Expressions", R. Sethi and J. D. Ullman, *Journal of the ACM* 17(4), pp. 715–728. October 1970.]
- Two-pass algorithm.
  - Pass 1: Recursively determine T_E, the minimum number of temporaries required to evaluate the given expression E.
  - Pass 2: Recursively generate code for E, being supplied with a list of temporary names (i.e., registers) of length $\geq T_E$.
- Doesn't change the amount of computation.
- Will in general reduce register pressure, resulting in fewer GPR spills.

Sethi-Ullman Numbering: Pass 1

- Recursive definition of T_E:

$$
\begin{array}{lr}
E \rightarrow \mathbf{id} & T_E = 1 \\
E \rightarrow \mathbf{unop}\, E_1 & T_E = T_{E_1} \\
E \rightarrow E_1 \mathbf{binop}\, E_2 & T_E = \begin{cases} \max\left(T_{E_1}, T_{E_2}\right), & \text{if } T_{E_1} \neq T_{E_2} \\ 1 + T_{E_1}, & \text{otherwise} \end{cases}
\end{array}
$$

Sethi-Ullman Numbering: Pass 2

- Recursive definition of Codegen(E, Tlist)
  - $E \rightarrow \mathbf{id}$:
    - Emit LOAD E.id.home, first(Tlist)
  - $E \rightarrow \mathbf{unop}\ E_1$:
    - Codegen(E_1, Tlist)
    - EMIT unop.op first(Tlist), first(Tlist)
  - $E \rightarrow E_1\ \mathbf{binop}\ E_2$:
    - **if** $\left(T_{E_1} \geq T_{E_2}\right)$ **then**
      - Codegen(E_1, Tlist)
      - Codegen(E_2, rest(Tlist))
      - Emit binop.op first(Tlist), second(Tlist), first(Tlist)
    - **else**
      - Codegen(E_2, Tlist)
      - Codegen(E_1, rest(Tlist))
      - Emit binop.op second(Tlist), first(Tlist), first(Tlist)