Sethi–Ullman algorithm

From Wikipedia, the free encyclopedia
(Redirected from Sethi-Ullman algorithm)

In computer science, the **Sethi–Ullman algorithm** is an algorithm named after Ravi Sethi and Jeffrey D. Ullman, its inventors, for translating abstract syntax trees into machine code that uses as few registers as possible.

Overview [edit]

When generating code for arithmetic expressions, the compiler has to decide which is the best way to translate the expression in terms of number of instructions used as well as number of registers needed to evaluate a certain subtree. Especially in the case that free registers are scarce, the order of evaluation can be important to the length of the generated code, because different orderings may lead to larger or smaller numbers of intermediate values being spilled to memory and then restored. The Sethi–Ullman algorithm (also known as **Sethi–Ullman numbering**) fulfills the property of producing code which needs the least number of instructions possible as well as the least number of storage references (under the assumption that at the most commutativity and associativity apply to the operators used, but distributive laws i.e. $a*b+a*c=a*(b+c)$ do not hold). Please note that the algorithm succeeds as well if neither commutativity nor associativity hold for the expressions used, and therefore arithmetic transformations can not be applied. The algorithm also does not take advantage of common subexpressions or apply directly to expressions represented as general directed acyclic graphs rather than trees.

Simple Sethi–Ullman algorithm [edit]

The **simple Sethi–Ullman algorithm** works as follows (for a load-store architecture):

1. Traverse the abstract syntax tree in pre- or postorder
    1. For every non-constant leaf node, assign a 1 (i.e. 1 register is needed to hold the variable/field/etc.). For every constant leaf node (RHS of an operation – literals, values), assign a 0.
    2. For every non-leaf node *n*, assign the number of registers needed to evaluate the respective subtrees of *n*. If the number of registers needed in the left subtree (*l*) are not equal to the number of registers needed in the right subtree (*r*), the number of registers needed for the current node *n* is max(l, r). If *l* == *r*, then the number of registers needed for the current node is *r* + 1.
2. Code emission
    1. If the number of registers needed to compute the left subtree of node *n* is bigger than the number of registers for the right subtree, then the left subtree is evaluated first (since it may be possible that the one more register needed by the right subtree to save the result makes the left subtree spill). If the right subtree needs more registers than the left subtree, the right subtree is evaluated first accordingly. If both subtrees need equal as much registers, then the order of evaluation is irrelevant.

Example [edit]

For an arithmetic expression $a=(b+c+f*g)*(d+3)$, the abstract syntax tree looks like this:

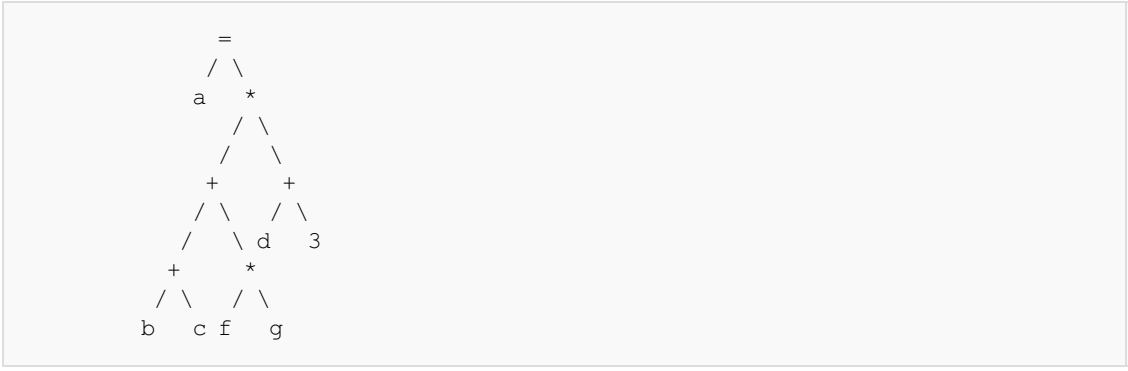
Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

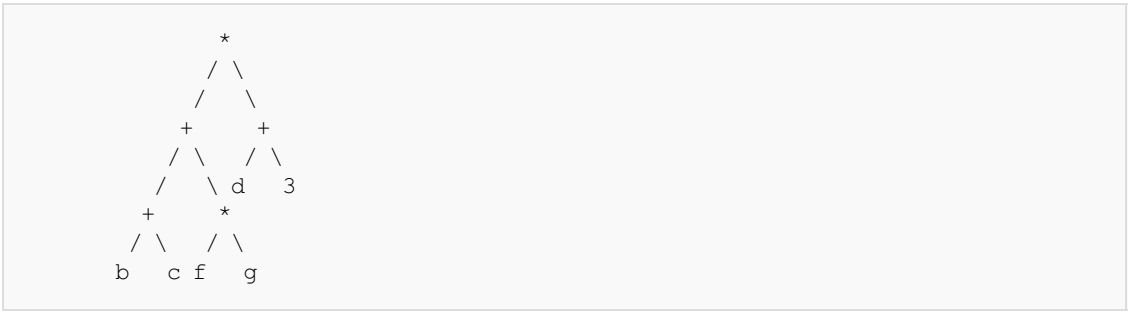
Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

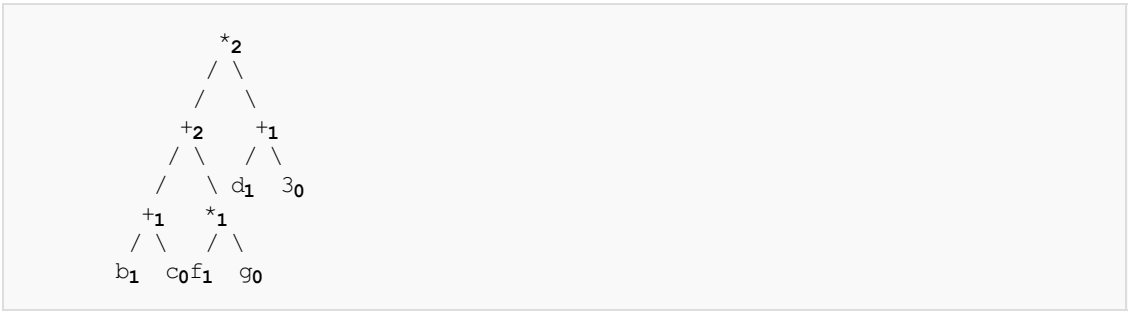
Languages
فارسی
日本語
Polski
Српски / srpski
Edit links



To continue with the algorithm, we need only to examine the arithmetic expression $(b + c + f * g) * (d + 3)$, i.e. we only have to look at the right subtree of the assignment '=':



Now we start traversing the tree (in preorder for now), assigning the number of registers needed to evaluate each subtree (note that the last summand in the expression $(b + c + f * g) * (d + 3)$ is a constant):



From this tree it can be seen that we need 2 registers to compute the left subtree of the '*', but only 1 register to compute the right subtree. Nodes 'c' and 'g' do not need registers for the following reasons: If T is a tree leaf, then the number of registers to evaluate T is either 1 or 0 depending whether T is a left or a right subtree(since an operation such as add R1, A can handle the right component A directly without storing it into a register). Therefore we shall start to emit code for the left subtree first, because we might run into the situation that we only have 2 registers left to compute the whole expression. If we now computed the right subtree first (which needs only 1 register), we would then need a register to hold the result of the right subtree while computing the left subtree (which would still need 2 registers), therefore needing 3 registers concurrently. Computing the left subtree first needs 2 registers, but the result can be stored in 1, and since the right subtree needs only 1 register to compute, the evaluation of the expression can do with only 2 registers left.

Advanced Sethi–Ullman algorithm [edit]

In an advanced version of the **Sethi–Ullman algorithm**, the arithmetic expressions are first transformed, exploiting the algebraic properties of the operators used.

See also [edit]

- Strahler number, the minimum number of registers needed to evaluate an expression without any external storage

References [edit]

- Sethi, Ravi; Ullman, Jeffrey D. (1970), "The Generation of Optimal Code for Arithmetic Expressions", *Journal of the Association for Computing Machinery* **17** (4): 715–728, doi:10.1145/321607.321620.

External links [edit]

- Code Generation for Trees

Categories: Compiler construction | Graph algorithms