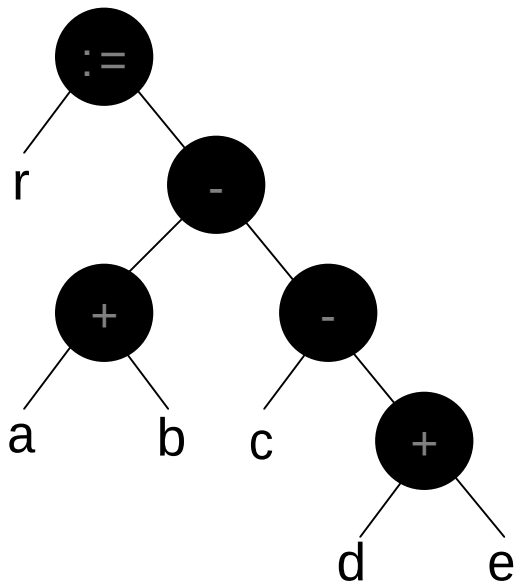
Code Generation: Integrated Methods

- Integration of register allocation with instruction selection for expression trees
- Restriction: Simple machine model:
  - r general purpose registers R_0, ..., R_{r-1}
  - Two-address instructions:
    - $R_i := M[V]$ Load
    - $M[V] := R_i$ Store
    - $R_i := R_i$ op $M[V]$ Compute
    - $R_i := R_i$ op R_j
- Two phases:
  1. Computing register requirements
  2. Generating code, allocating register and temporaries

Example Tree

- Source:
  - r := (a+b)-(c-(d+e))
- Tree:

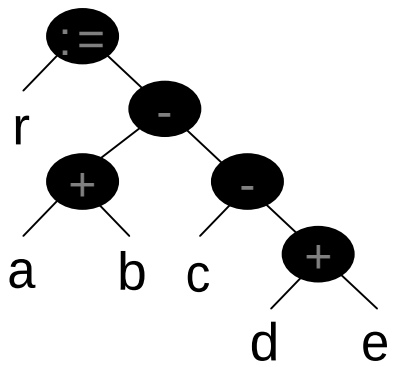


Generated Code

- Given two registers R_0 and R_1, two possible code sequences are:

```
R0    := M[a]
R0    := R0 + M[b]
R1    := M[d]
R1    := R1 + M[e]
M[t1]:= R1
R1    := M[c]
R1    := R1 - M[t1]
R0    := R0 - R1
M[f] := R0
```

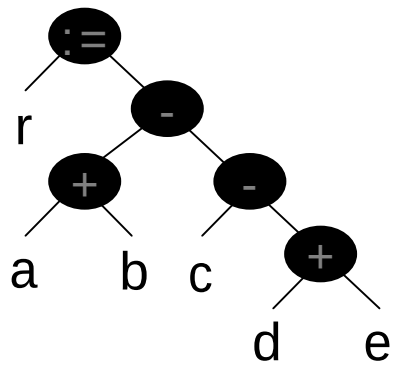
```
R0    := M[c]
R1    := M[d]
R1    := R1 + M[e]
R0    := R0 – R1
R1    := M[a]
R1    := R1 + M[b]
R1    := R1 – R0
M[f] := R1
```



Generated Code

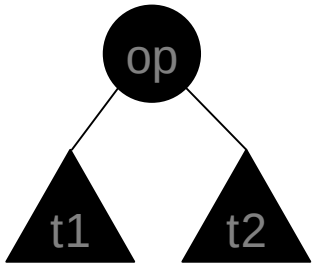
- Left code:
  - stores result for c - (d + e) in a temporary
  - no register available

- Right code:
  - evaluates c - (d + e) first (needs 2 registers)
  - saves one instruction



The Algorithm

- Principle: Given tree t for expression e1 op e2
- t1 needs r1 registers,
  t2 needs r2 registers
- Assume r ≥ r1 > r2:
  - After evaluation of t1:
    - r1-1 registers freed
    - one holds the result
  - t2 gets enough registers to evaluate, hence t can be evaluated in r1 registers
- Assume r1=r2:
  - t needs r1+1 registers to evaluate
- Assume one of t1 or t2 need more than r registers:
  - spill to temporary required



Labeling Phase

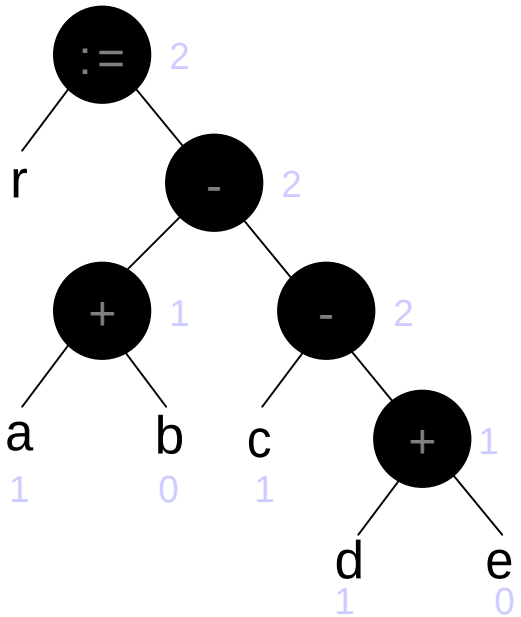
- Labels each node with its register needs

- Bottom up pass:
  - Left leaves labeled with '1' have to be loaded into registers
  - Right leaves labeled with '0' are used as operands
  - Inner nodes:

$$regneed(op(t_1, t_2)) = \begin{cases} max(r_1, r_2), & \text{if } r_1 \neq r_2 \\ r_1 + 1, & \text{if } r_1 = r_2 \end{cases}$$

where $r_1 = regneed(t_1)$, $r_2 = regneed(t_2)$

Example



Generation Phase

- Generates instruction OP for operator *op* in *op(t1,t2)* after generating code for *t1* and *t2*
- Order of *t1* and *t2* depends on their register needs
- Upon execution of the generated Op-instruction: value of *t1* in register
- RSTACK: available registers, initially all registers.
- Before processing *t*, the result register for *t* is top(RSTACK)
- After processing *t*: all registers are available, top(RSTACK) is result register for *t*
- TSTACK: available temporaries

Register Allocation and Instruction Selection by Dynamic Programming

- More complex architecture:
  - r general purpose registers R_0, …, R_{r-1}
  - Instruction formats:
    - $R_i := e$ Compute
    - $R_i := M[V]$ Load
    - $M[V] := R_i$ Store

    where
    - e term with registers and memory cells
    - costs are associated with each instruction
  - Goal: Generate cheapest instruction sequence using no more than r registers
  - Assume contiguous computation of subtrees → only one register required to hold the result
  - Use some instruction selection technique to compute cheapest instruction sequence.

Canonical Recursive Solution

- Assume *e* of instruction *R1:=e* matches tree *t*
- Subtrees of *t* corresponding to memory operands of *e*: they are computed into memory and no registers are occupied after that
- Let *e* have *k* register operands: how to compute the corresponding subtrees $t_1, \ldots, t_k$ into these registers?
- Assume order $i_1, i_2, \ldots, i_k$ and *j* available registers
- t_{i1} has *j* registers available, t_{i2} has *j-1*, t_k has *j-k*

Canonical Recursive Solution

- If this fits *(j-k-regneed(t_{ik})>=0)*, add the minimal costs for computing all subtrees in this way to the costs of *e* to yield the minimal costs for this combination.

- If not enough registers are available, compute enough subtrees into memory, and sum up costs like above.

- Doing this for all potential combinations recomputes the costs for subtrees $\rightarrow$ exponential complexity.

Dynamic Programming

- Convert top-down algorithm into bottom-up algorithm tabulating partial solutions
- Associate cost vector C[0..r] with each node n
  C[0]: cheapest costs for computing t/n into a temporary, C[i] cheapest costs computing t/n into a register using i registers
- Compute cost vector at node n minimizing over all legal combinations of
  - one applicable instruction
  - the cost vectors of the nodes under non-terminal nodes in the applied rule
- What is a legal combination for C[j],j>0?
  - Any combination of generated code for subtrees not needing more than j registers.
- Extract cheapest instruction sequence in a second pass.

Global Register Allocation

- 'Global' means Register allocation across whole procedures / programs.
- Symbolic registers: hold intermediate results and modified variables.
- Two symbolic registers collide, if their contents are live at the same time.
- Colliding symbolic registers cannot be allocated to the same real register.
- Goal: allocate the (unbounded number of) symbolic registers to the fixed number of physical registers without collision.
- A definition of a symbolic register is the computation of an intermediate result of the modification of a variable

Global Register Allocation

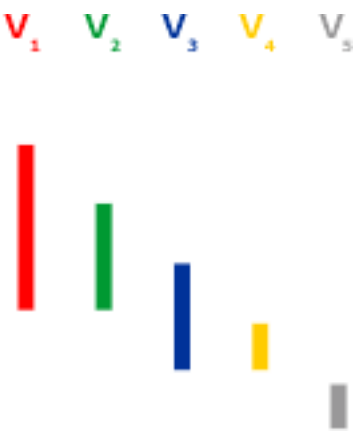
- A use of a symbolic register is a reading access to the corresponding variable or a use of the intermediate value.

- A symbolic register (a variable) r is live at a program point *p*, if there is a program path from the entry node of the procedure to *p* that contains a definition of *r* and there is a path from *p* to a use of *r* on which *r* is not defined. The life range (life span) of a symbolic register *r* is the set of program points at which *r* is live.

- Two life ranges of symbolic registers interfere, if one of them is defined during the life range of the other. The register interference graph is an undirected graph whose nodes are life ranges of symbolic registers and whose
  edges connect the nodes of interfering life ranges.

Life Range and Register Interference

a) $v_1 = Mem[0xAFA0]$
b) $v_2 = Mem[0xAFC0]$
c) $v_3 = v_1 + v_2$
d) $v_4 = v_1 * v_2$
e) $v_5 = v_3 + v_4$
f) return v_5



Graph Coloring

- If *k* physical registers are available, the k-coloring problem must be solved.
- NP-complete for k>2 -> Use heuristics
- Algorithm:
  - If G contains a node n with degree < k:
    - n and its neighbors can be colored with different colors
    - Remove n from G, decreasing the size of G.
    - G is k-colorable, if we arrive at the empty graph.
  - If G is not empty and there exists no node with degree <k:
    - use heuristics to select one node to remove (spilling)
    - modify program inserting spills at definitions and loads at uses
    - reflect changes in graph.

Heuristics for Node Removal

- Degree of the node: high degree causes many deletions of edges.
- Costs of spilling.

Instruction Scheduling

- Definition: Reordering an operation sequence in order to exploit instruction-level parallelism and to minimize pipeline stalls.
  - Complexity: NP-complete.

- Terminology:
  - An operation is a basic machine operation like add, sub, etc.
  - An instruction is a set of machine operations that are issued simultaneously (cf. VLIW).
    Example:

```
r3=r1+r4, r3=r10+r14, r11=dm(i6,m6), r12=pm(i15,m15);
```

Instruction Scheduling

- Scope of instruction scheduling:
  - local acyclic instruction scheduling: reordering operations inside basic blocks. Standard technique: list scheduling.
  - global acyclic instruction scheduling: reordering operations across basic block boundaries but not across loop boundaries. Standard technique: trace scheduling.
  - cyclic instruction scheduling: reordering operations across loop boundaries. Standard technique: software pipelining.

List Scheduling

```
SET data_ready;
int cycle=0;

Insert operations without predecessors in the data
   dependence graph into the data_ready set.

while (data_ready ≠ ∅) do {
   cycle = cycle+1;

   Choose operations from data_ready in priority order and
        insert them into the current cycle, until
   data_ready is empty or the insertion leads to a
   resource conflict.

   Insert all operations into data_ready that can be
   scheduled   in the next cycle without violating data
   dependences.
}
```

List Scheduling

- The priority in which operations from the data ready set are chosen is determined by heuristics.

- Common heuristics: highest-level-first heuristics.
  - The priority of each operation is the length of the longest path in the data dependence graph starting from this operation.

List Scheduling

- Code quality:
  - Standard list scheduling algorithm with highest-level-first exceeds optimal number of instructions (clock cycles) by 8.29% (maximum deviation 21.05%) for Analog Devices SHARC on standard DSP kernels, according to recent studies.
  - Standard list scheduling algorithm with highest-level first heuristics exceeds optimal number of instructions (clock cycles) by 7.29% (maximum deviation 42.85%) for Philips TriMedia TM1000 on standard DSP kernels, according to recent studies.

Exposing More Instruction-Level Parallelism

- Degree of instruction-level parallelism inside basic blocks is limited – typically to 2.
- The available parallelism in contemporary microprocessors grows. Better exploitation by
  - scheduling sequences of consecutive basic blocks
  - scheduling entire loops
  - speculation
- Two kinds of speculation:
  - dynamic: based on hardware branch prediction. In case of mispredicted branch forgetting or undoing effects of speculatively executed instructions.
  - static: generating compensation code for "speculatively placed" instructions.
  - Recently: Static data and control speculation with hardware support to deal with mispredictions (Intel IA-64).