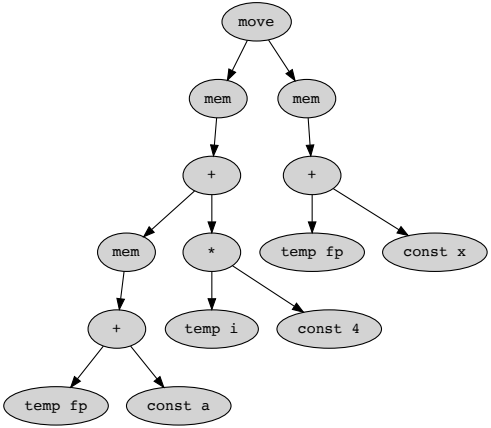
Compiler Construction

Instruction Selection

Problem Statement

How would you translate **a[i] := x** where `x` is frame resident, and `i` is not?

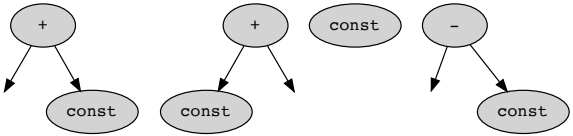


Tree Patterns

- Translation from Tree to Assembly corresponds to *parsing a tree*.
- Looking for a covering of the tree, using tiles.
- The set of tiles corresponds to the instruction set.

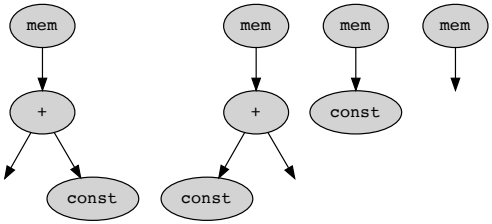
Tiles

Missing nodes are plugs for *temporaries*:
tiles read from temps, and create temps.

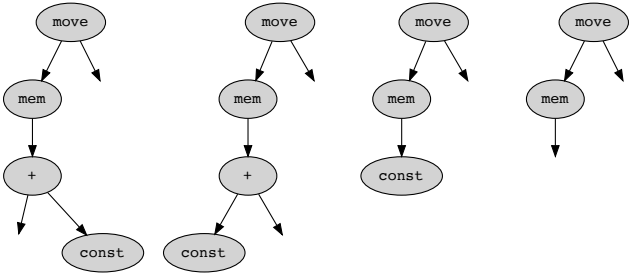


Some architectures rely on a special register to produce 0.

Tiles: Loading $\texttt{load}\ r_i \leftarrow M[r_j + c]$

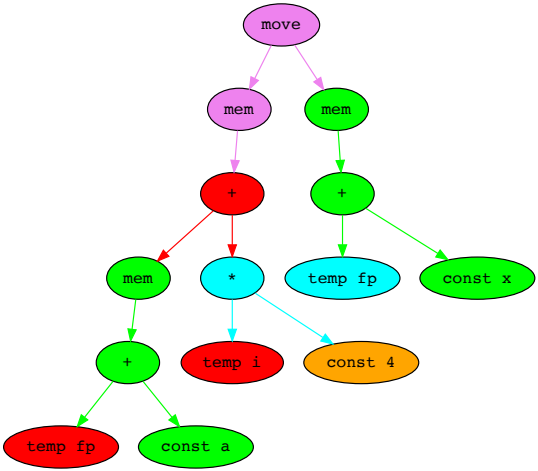


Tiles: Storing $\texttt{store}\ \texttt{M}[\texttt{r}_\texttt{j} + \texttt{c}] \leftarrow \texttt{r}_\texttt{i}$



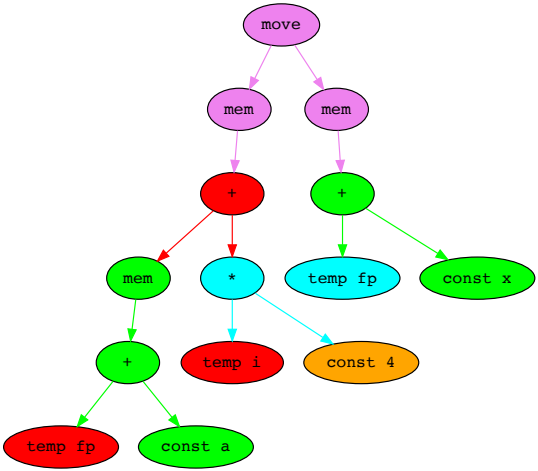
Simple Instruction: Translation 1

```
load t17 <- M[fp + a]
addi t18 <- r0 + 4
mul t19 <- ti * t18
add t20 <- t17 + t19
load t21 <- M[fp + x]
store M[t20+0] <- t21
```



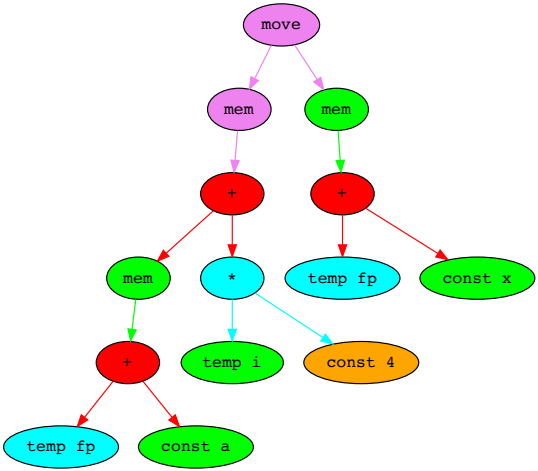
Simple Instruction: Translation 2

```
load t17 <- M[fp + a])
addi t18 <- r0 + 4
mul t19 <- ti * t18
add t20 <- t17 + t19
addi t21 <- fp + x
movem M[t20]
              <- M[t21]
```



Simple Instruction: Translation 3

```
addi t17 <- r0 + a
add t18 <- fp + t17
load t19 <- M[t18 + 0]
addi t20 <- r0 + 4
mul t21 <- ti * t20
add t22 <- t19 + t21
addi t23 <- r0 + x
add t24 <- fp + t23
load t25 <- M[t24 + 0]
store M[t22 + 0]
            <- t25
```



Translating a Simple Instruction

- There is always a solution (provided the instruction set is reasonable)
- there can be several solutions
- given a cost function, some are better than others:
  - some are locally better, *optimal coverings* (no fusion can reduce the cost),
  - some are globally better, *optimum coverings*.

Algorithms for Instruction Selection (1/2)

Maximal Munch Find an optimal tiling.

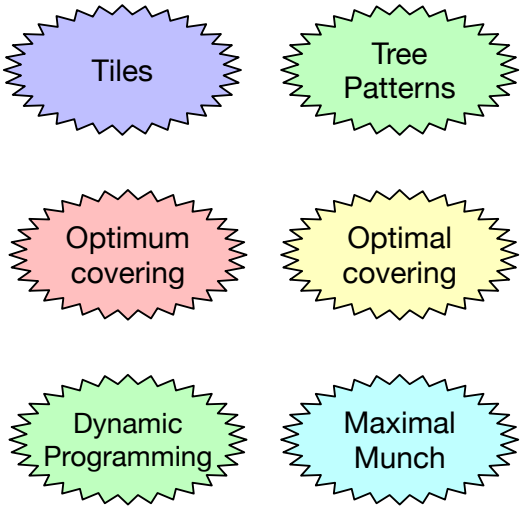
- Top-down strategy.
- Cover the current node with the largest tile.
- Repeat on subtrees.
- Generate instructions in reverse-order after tile placement.

Algorithms for Instruction Selection (2/2)

Dynamic Programming Find an optimum tiling.

- Bottom-up strategy.
- Assign cost to each node.
- Cost = cost of selected tile + cost of subtrees.
- Select a tile with minimal cost and recur upward.
- Implemented by code generator generators (Twig, Burg, iBurg, MonoBURG, ...).

Summary


Tiles
Tree Patterns
Optimum covering
Optimal covering
Dynamic Programming
Maximal Munch