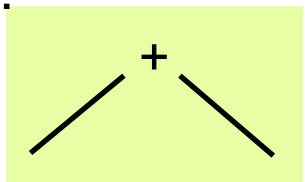
Lecture 9
Code Generation
Instruction Selection

Instruction Selection

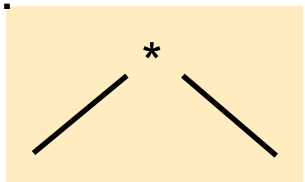
- *Mapping IR trees to assembly instructions*
  - Tree patterns
  - Retain virtual registers (temps)
- *Algorithms for instruction selection*
  - Maximal munch
  - Dynamic programming
  - Tree grammars
- *Instruction selection for the tiger compiler*
  - Generic assembly instructions (Assem.instr)

Instructions and Tree Patterns

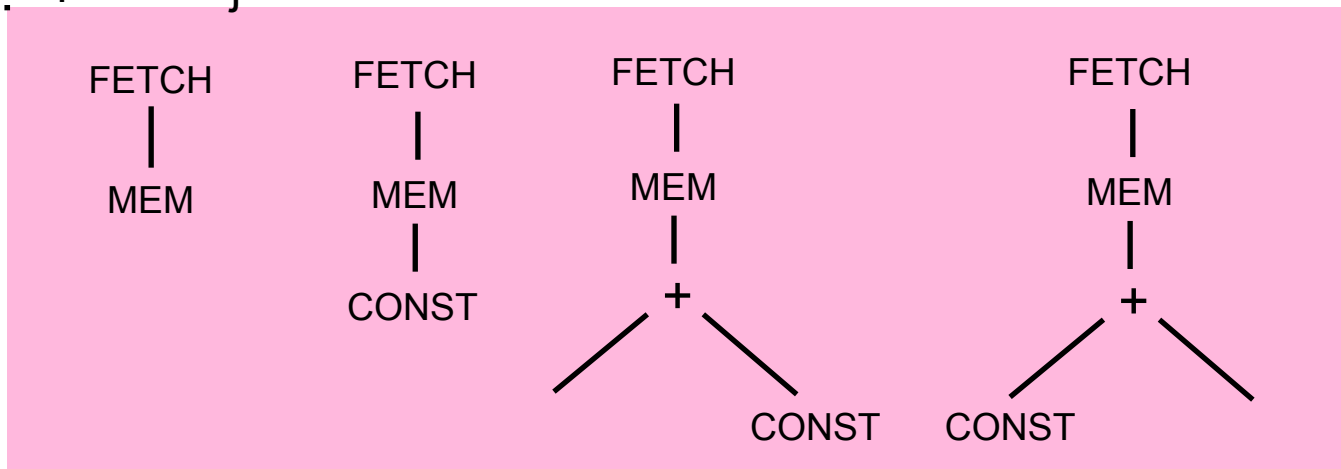
ADD $r_i \leftarrow r_j + r_k$



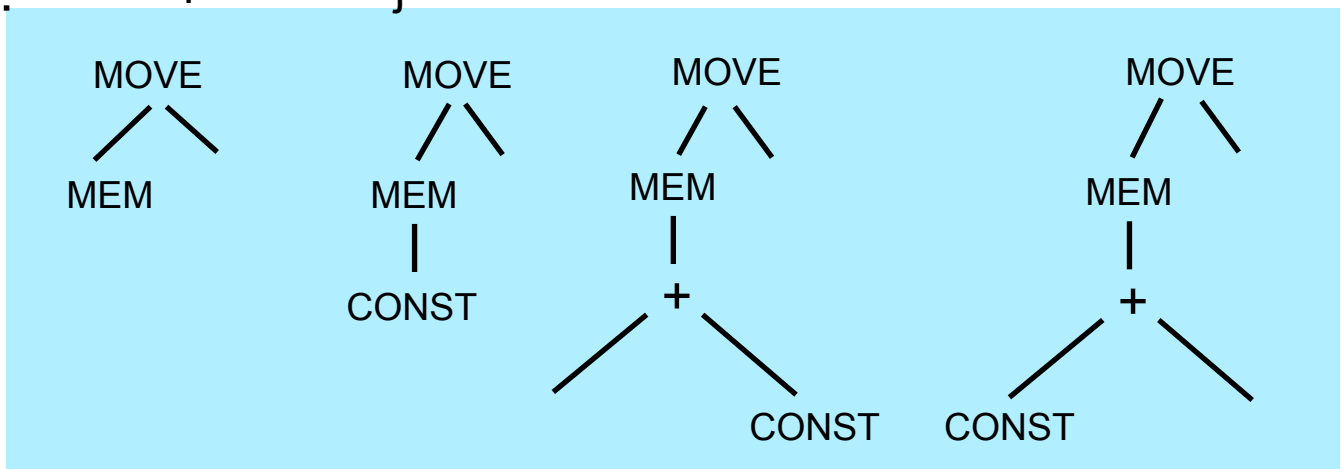
MUL $r_i \leftarrow r_j * r_k$



LOAD $r_i \leftarrow M[r_j + c]$



STORE $M[r_i + c] \leftarrow r_j$

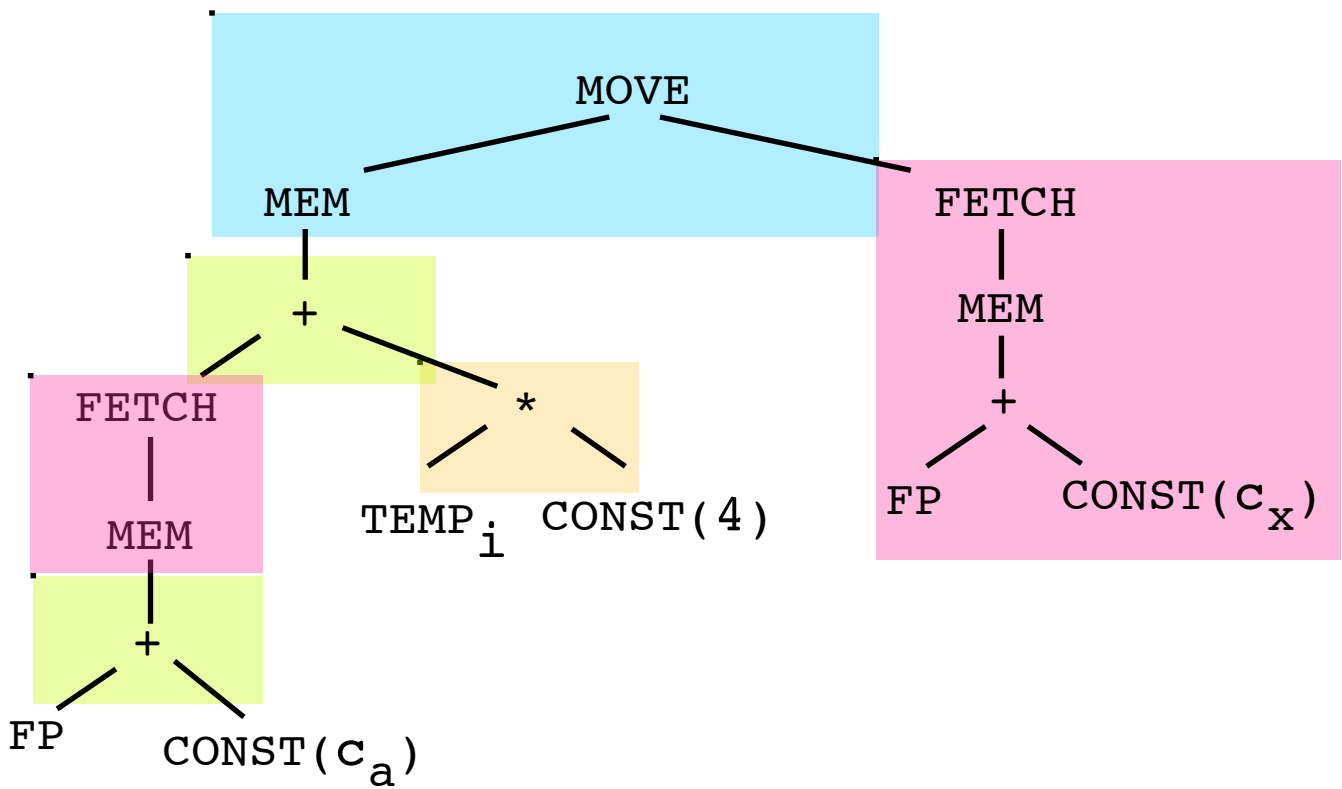


Tiling with Instruction Patterns

a[i] := x
where i = $TEMP_i$

$a = M[fp + c_a] = c_a(\$fp)$

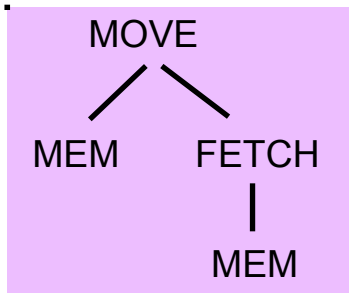
$x = M[fp + c_x] = c_x(\$fp)$

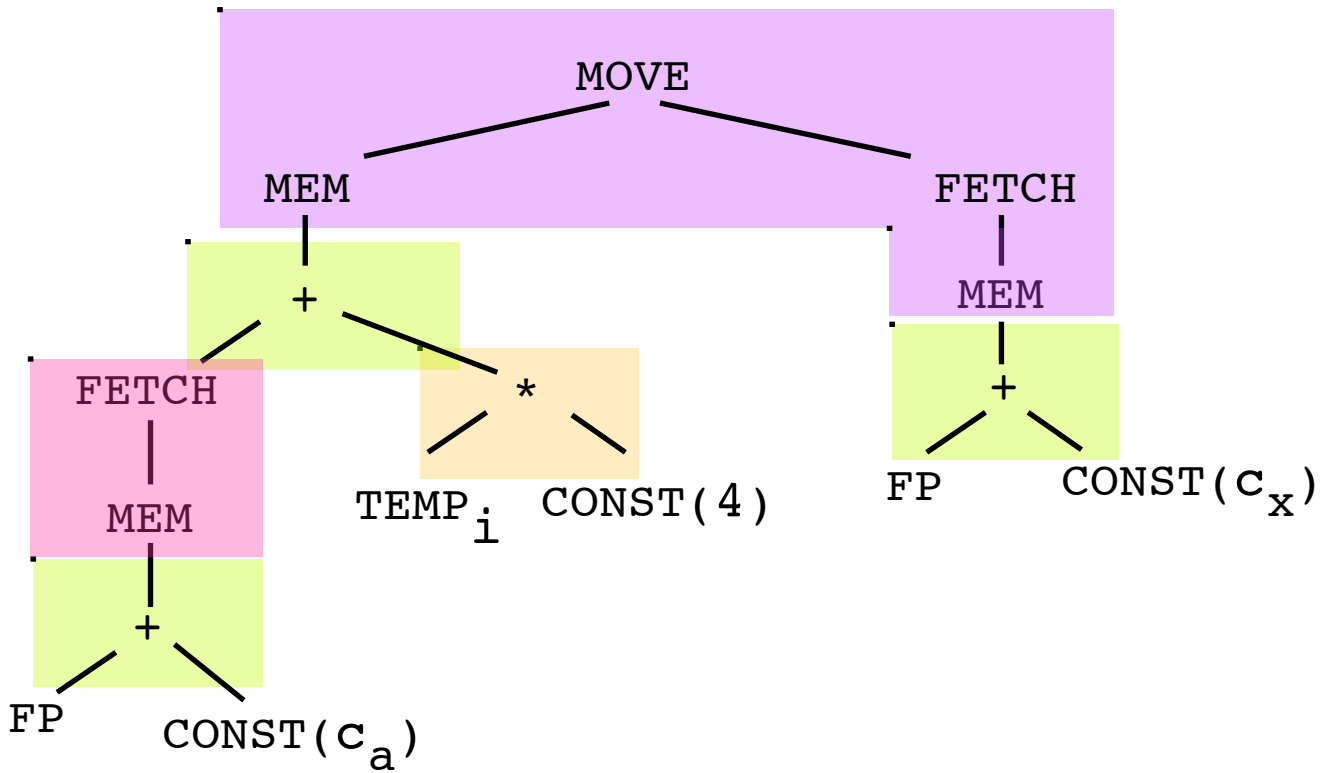


Tiling with Instruction Patterns

`a[i] := x`

MOVEM $M[r_i] \leftarrow M[r_j]$





Optimal, Optimum Tilings

A tiling of an IR tree with instruction patterns is *optimum* if the set of patterns has the least "cost" by some measure, e.g.:

1. the smallest number of patterns (producing the shortest instruction sequence, thus smallest code size)
2. total number of cycles to execute, thus fastest code

A tiling is *optimal* if no two adjacent tiles can be combined into a single tile of lower cost.

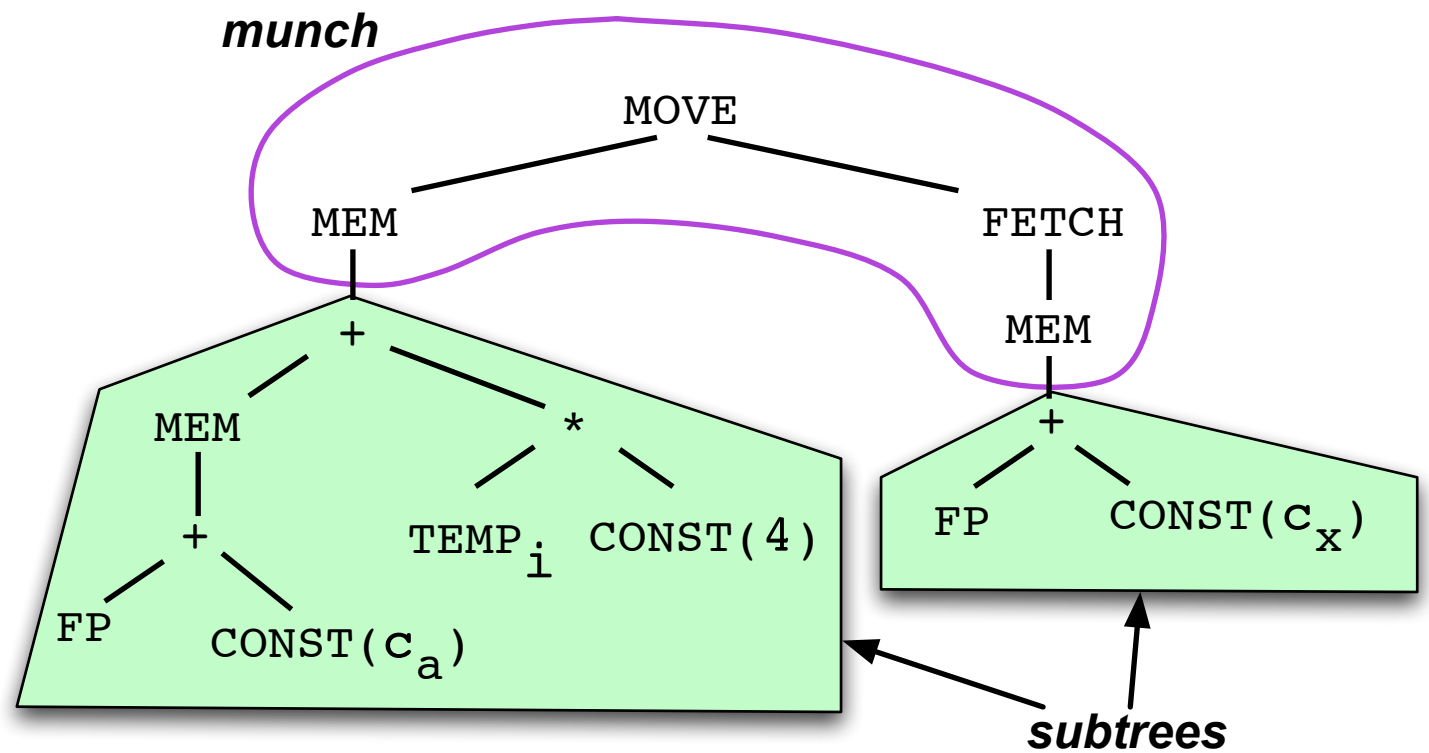
optimum $\Rightarrow$ *optimal*

Maximal Munch

1. Choose the “largest” tile that matches at the root of the IR tree (this is the *munch*)

2. Recursively apply maximal much at each subtree of this munch.

3. Generate the instruction for the first munch, using source and destination registers produced by processing the subtrees.



Implementing Maximal Munch

ML pattern matching is natural way to implement maximal munch.

```
fun munchStm(MOVE(MEM(BINOP(PLUS,e1,CONST i)),e2) =
    (munchExp e1; munchExp e2; emit “STORE”)
  | munchStm(MOVE(MEM(BINOP(PLUS,CONST i,e1)),e2) =
    (munchExp e1; munchExp e2; emit “STORE”)
  | munchStm(MOVE(MEM e1,FETCH(MEM e2))) =
    (munchExp e1; munchExp e2; emit “STORE”)
  | ...

fun munchExp(BINOP(PLUS,e1,CONST i)) =
    (munchExp e1; emit “ADDI”)
  | munchExp(BINOP(PLUS,CONST i,e1)) =
    (munchExp e1; emit “ADDI”)
  | munchExp(BINOP(PLUS,e1,e2)) =
    (munchExp e1; munchExp e2; emit “ADD”)
  | ...
```

Dynamic Programming

Assume that we have a cost function assigning a cost (size, time) to each instruction. We want to compute

Tile(S) = (T, c)

where T is an optimal tiling and c is its cost.

Try each top-level tile that matches at root of S: say P_1, P_2, P_3

For each of these tiles, there is a set of residual subtrees:

$P_1 \Rightarrow \{S_{1,1}, S_{1,2}\}$

$P_2 \Rightarrow \{S_{2,1}, S_{2,2}, S_{2,3}\}$

$P_3 \Rightarrow \{S_{3,1}, S_{3,2}\}$

For each subtree, recursively compute the best tiling and its cost:

Tile($S_{i,j}$) = ($T_{i,j}$, $c_{i,j}$)

Choose $T = P_i(T_{i,1}, ..., T_{i,n})$ such that $c = c_{i,1} + ... + c_{i,n}$ is minimal.

Tree Grammars

We don't use things like ML-Burg any more, because it is really over-kill --- maximal munch does just as well and is easier to write. The problems with burg are the following:

- It does not do input rewriting, therefore you either have to code up all the commutative rules, or ensure the input is in some canonical form (immediates as second operand, etc). This results in a lot of rules, and corresponding semantic actions that need to be written.

- One ends up inventing many terminals to represent different sized immediates, e.g, INT0, INT4, INT8, INT255, ... etc. INT0, INT4, and INT8 may be used to represent the corresponding constants, and INT255 is anything between 0..255. At this point a lot of hand crafting is required in the rules and semantic actions.

- Instruction selection is just the beginning of the story. Constant propagation/ folding, spilling, code motion, and others may open new opportunities for better instruction selection. Thus there is little point in trying to be optimal in the initial instruction selection phase.

Lal George, author or MLRISC

Maximal Munch for Tiger

How to represent (MIPS) instructions?

```
datatype instr
  = OPER of
      {assem: string,
       dst: temp list,
       src: temp list,
       jump: label list option}
  | LABEL of {assem: string, lab: label}
  | MOVE of
      {assem: string,
       dst: temp,
       src: temp}
```

Instructions

Example instructions:

```
sw $ta 4($tb)

  OPER{assem = "sw `s0, 4(`s1)",
       dst = [],
       src = [temp_a,temp_b],
       jump = NONE}

j L0

  LABEL{assem = "L0", lab = label_L0}

move $ta, $tb

  MOVE{assem = "move `d0, `s0",
       dst = temp_a,
       src = temp_b}
```

Formating Instructions

```
val format : (temp -> string) -> instr -> string
```

```
format sayTemp
 (OPER{assem = "sw `s0, 4(`s1)",
       dst = [],
       src = [tempa,tempb],
       jump = NONE})
```

`s0 ⇒ sayTemp(nth(src,0)) = "$t3"
`s1 ⇒ sayTemp(nth(src,1)) = "$fp"

Result: "sw $t3, 4($fp)"

sayTemp maps template variables (by number) to names of actual registers

Munch cases

```
| munchStm (T.CJUMP (rop, e1, e2, t, f)) =
  1. rop translates to a corresponding conditional
     branch opcode
  2. arguments e1 and e2 are munched by munchExp,
     returning two source temps (src1, src2)
  3. there are two jump labels, t and f

 OPER{assem =
        concat[oper, " `s0, `s1, ", Temp.labelToString t],
      src = [src0,src1],
      dst = [],
      jump = SOME [t, f]}

After register allocation, this might format to:

 “bgt $t1, $t3, L5”
```

Munch cases

```
| munchStm (T.MOVE (T.TEMP d, T.FETCH (T.MEM saddr))) =
```

This could translate simply to

```
  OPER{assem = "lw `d0, `s0",
       src = [munchExp saddr], dst = [d],
       jump = NONE}
```

But some special cases of saddr can be optimized using addressing modes:

```
  saddr = BINOP(PLUS, e, CONST c)
  saddr = BINOP(PLUS, CONST c, e)
    ⇒ “lw $t2 c($t3)”

  saddr = BINOP(MINUS, e, CONST c)
    ⇒ “lw $t2 -c($t3)”
```

Questions

1. Do we need a munchExp case for *ESEQ*?

```
  | munchExp (T.ESEQ (stm, exp)) =
```

2. Do we need a *munchLexp* function? If not, why not?