Topic 8: Instruction Selection

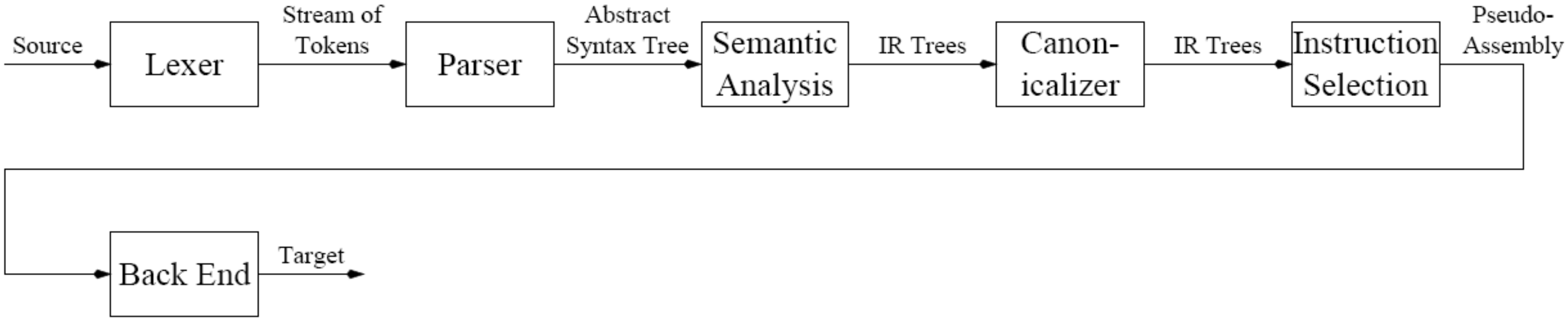
COS 320

Compiling Techniques

Princeton University
Spring 2016

Lennart Beringer

Instruction Selection



Instruction Selection

- Process of finding set of machine instructions that implement operations specified in IR tree.
- Each machine instruction can be specified as an IR tree fragment $\longrightarrow$ *tree pattern*
- Goal of instruction selection is to cover IR tree with non-overlapping tree patterns.

Our Architecture

- Load/Store architecture (arithmetic instructions operate on registers)
- Relatively large, general purpose register file
  - Data or addresses can reside in registers (unlike Motorola 68000)
  - Each instruction can access any register (unlike x86)
- r_0 always contains zero.
- Each instruction has latency of one cycle.
- Execution of only one instruction per cycle.

Our Architecture

Arithmetic:

ADD	$r_d = r_{s1} + r_{s2}$
ADDI	$r_d = r_s + c$
SUB	$r_d = r_{s1} - r_{s2}$
SUBI	$r_d = r_s - c$
MUL	$r_d = r_{s1} * r_{s2}$
DIV	$r_d = r_{s1} / r_{s2}$

Memory:

LOAD	$r_d = M[r_s + c]$
STORE	$M[r_{s1} + c] = r_{s2}$
MOVEM	$M[r_{s1}] = M[r_{s2}]$

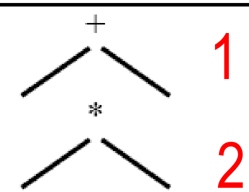
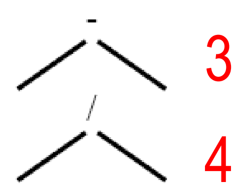
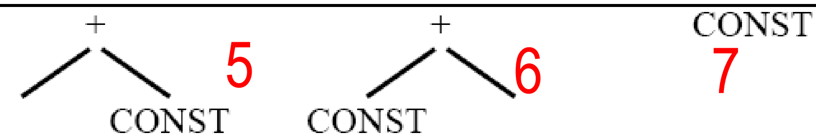
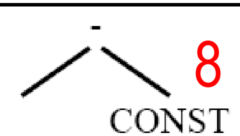
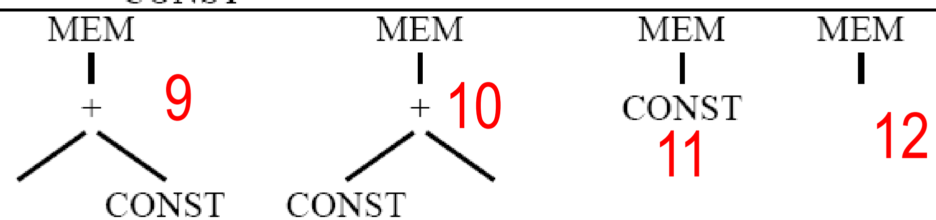
Pseudo-ops

Pseudo-op - An assembly operation which does not have a corresponding machine code operation. Pseudo-ops are resolved during assembly.

MOV	$r_d = r_s$	ADDI	$r_d = r_s + 0$
MOV	$r_d = r_s$	ADD	$r_d = r_{s1} + r_0$
MOVI	$r_d = c$	ADDI	$r_d = r_0 + c$

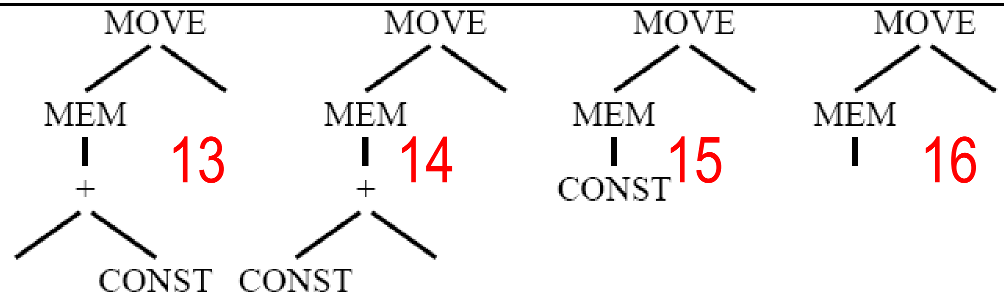
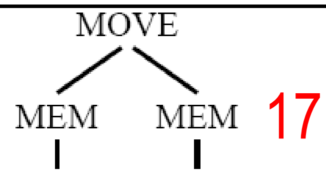
(Pseudo-op can also mean assembly directive, such as `.align`.)

Instruction Tree Patterns

Name	*Effect*	*Trees*
—	r_i	TEMP
ADD MUL	$r_i \quad r_j + r_k$ $r_i \quad r_j \times r_k$	
SUB DIV	$r_i \quad r_j \quad r_k$ $r_i \quad r_j / r_k$	
ADDI	$r_i \quad r_j + c$	
SUBI	$r_i \quad r_j \quad c$	
LOAD	$r_i \quad M[r_j + c]$	

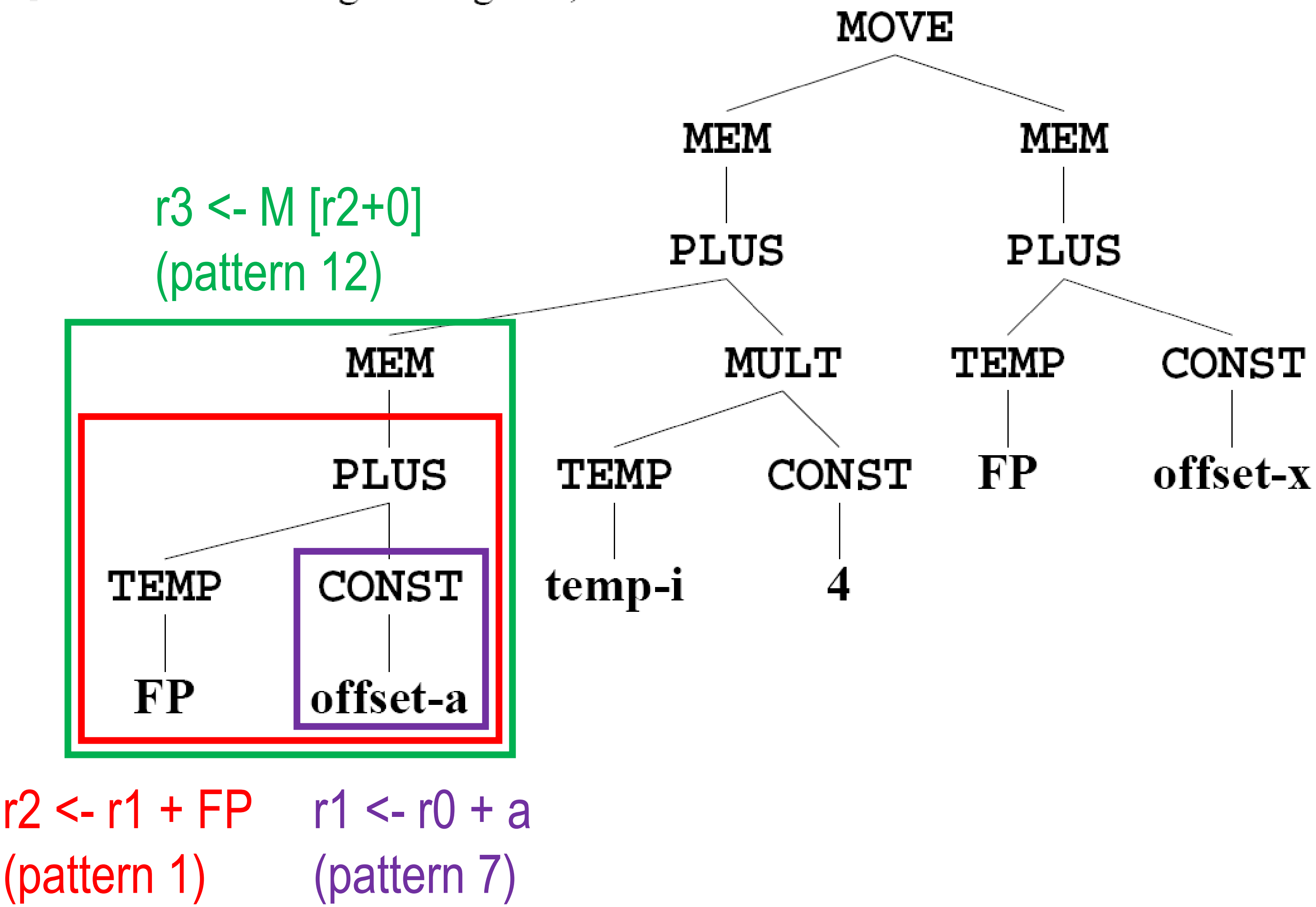
(rule numbers for reference later on)

Instruction Tree Patterns

<table>
<tr><td>STORE</td><td>$M[r_j + c]$</td><td>r_i</td><td>
</td></tr>
<tr><td>MOVEM</td><td>$M[r_j]$</td><td>$M[r_i]$</td><td>
</td></tr>
</table>

Example

`a[i] :=` x assuming i in register, a and x in stack frame.



Individual Node Selection

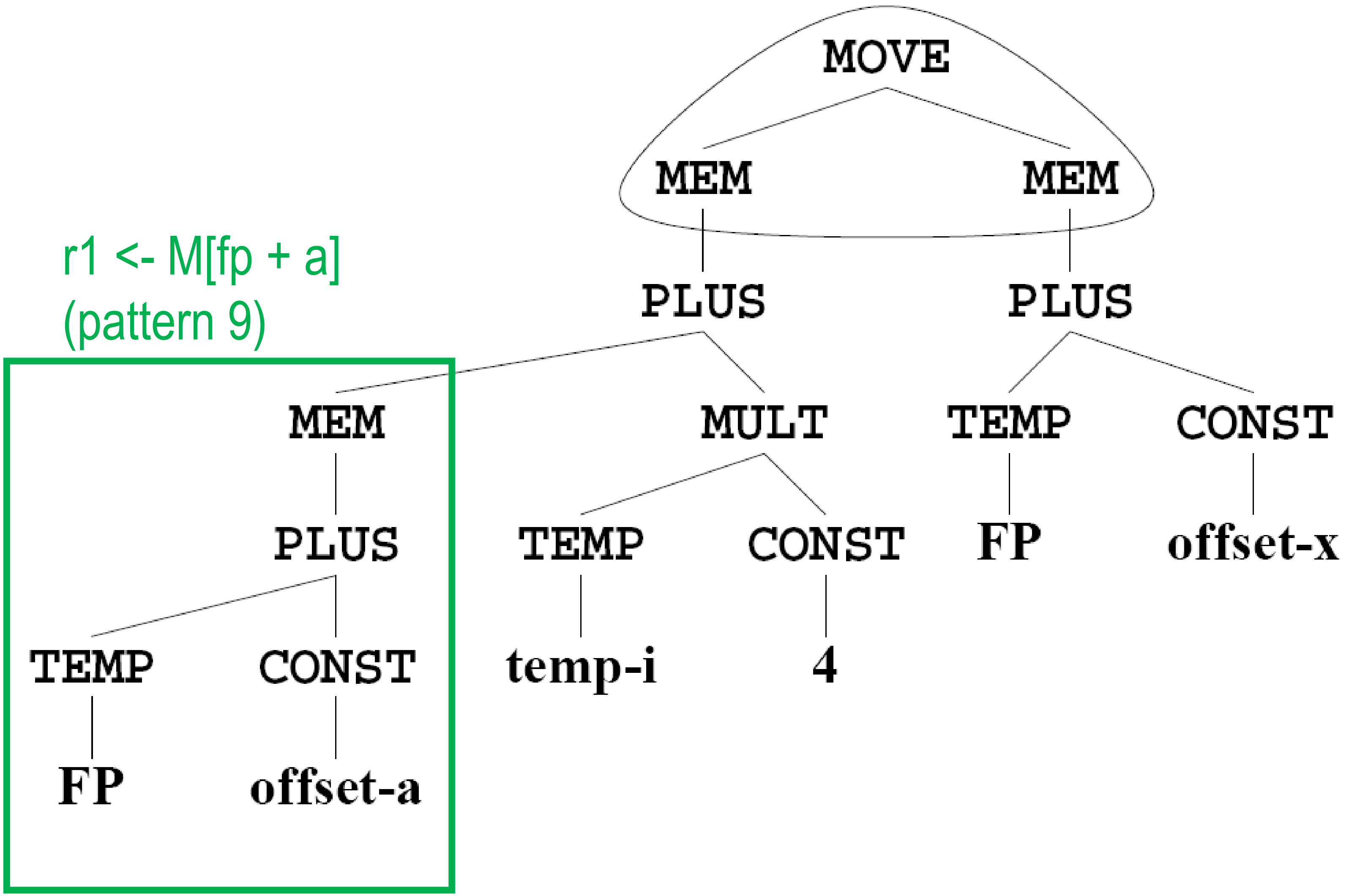
Covering of tree yields sequence of instructions (“inside out/bottom-up”)

No register allocation yet

```
ADDI  r1 = r0 + offset_a
ADD   r2 = r1 + FP
LOAD  r3 = M[r2 + 0]

ADDI  r4 = r0 + 4
MUL   r5 = r4 * r_i

ADD   r6 = r3 + r5

ADDI  r7 = r0 + offset_x
ADD   r8 = r7 + FP
LOAD  r9 = M[r8 + 0]

STORE M[r6 + 0] = r9
```

9 registers, 10 instructions

Tiling not unique



Code resulting from yet another tiling

```
ADDI   r1 = r0 + offset_a
ADD    r2 = r1 + FP
LOAD   r3 = M[r2 + 0]

ADDI   r4 = r0 + 4
MUL    r5 = r4 * r_i

ADD    r6 = r3 + r5

ADDI   r7 = r0 + offset_x
ADD    r8 = r7 + FP
MOVEM  M[r6] = M[r8]
```

Saves a register ($9 \to 8$) and an instruction ($10 \to 9$).

Node Selection

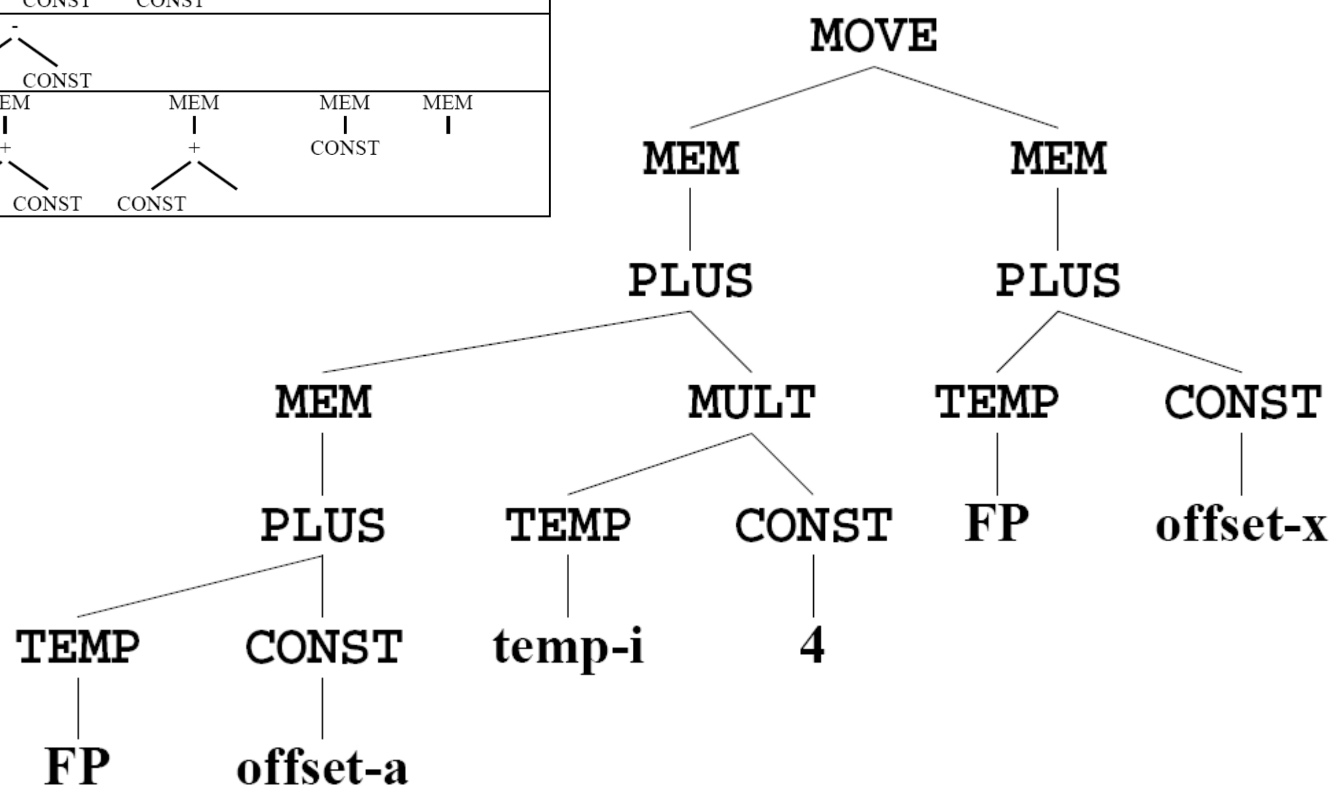
- There exist many possible tilings - want tiling/covering that results in instruction sequence of *least cost*
  - Sequence of instructions that takes least amount of time to execute.
  - For single issue fixed-latency machine: fewest number of instructions.
- Suppose each instruction has fixed cost:
  - *Optimum Tiling*: tiles sum to lowest possible value - globally "the best"
  - *Optimal Tiling*: no two adjacent tiles can be combined into a single tile of lower cost - locally "the best"
  - Optimal instruction selection easier to implement than Optimum instruction selection.
  - Optimal is roughly equivalent to Optimum for RISC machines.
  - Optimal and Optimum are noticeably different for CISC machines.
- Instructions are not self-contained with individual costs.

Optimal Instruction Selection: Maximal Munch

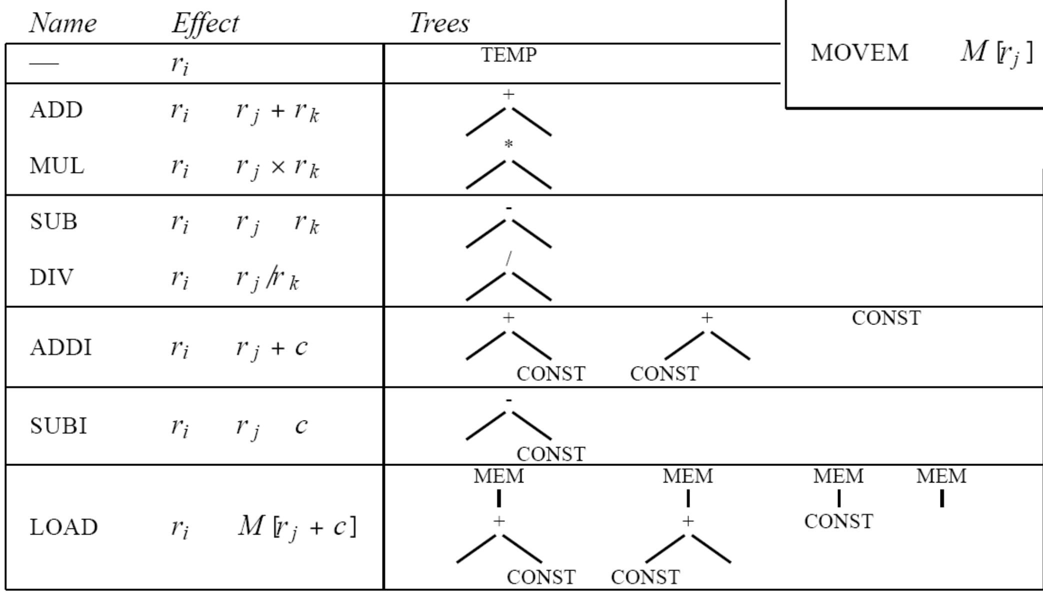
- Cover root node of IR tree with largest tile t that fits (most nodes)
  - Tiles of equivalent size $\Rightarrow$ arbitrarily choose one.
- Repeat for each subtree at leaves of t.
- Generate assembly instructions in reverse order - instruction for tile at root emitted last.

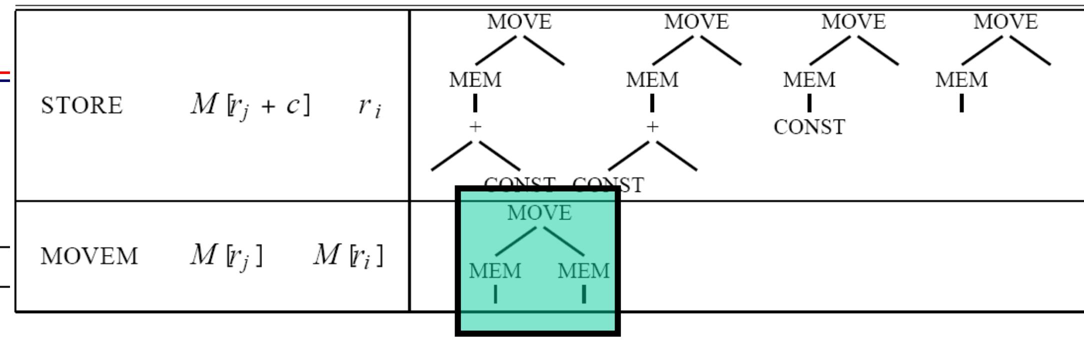
Maximal Munch

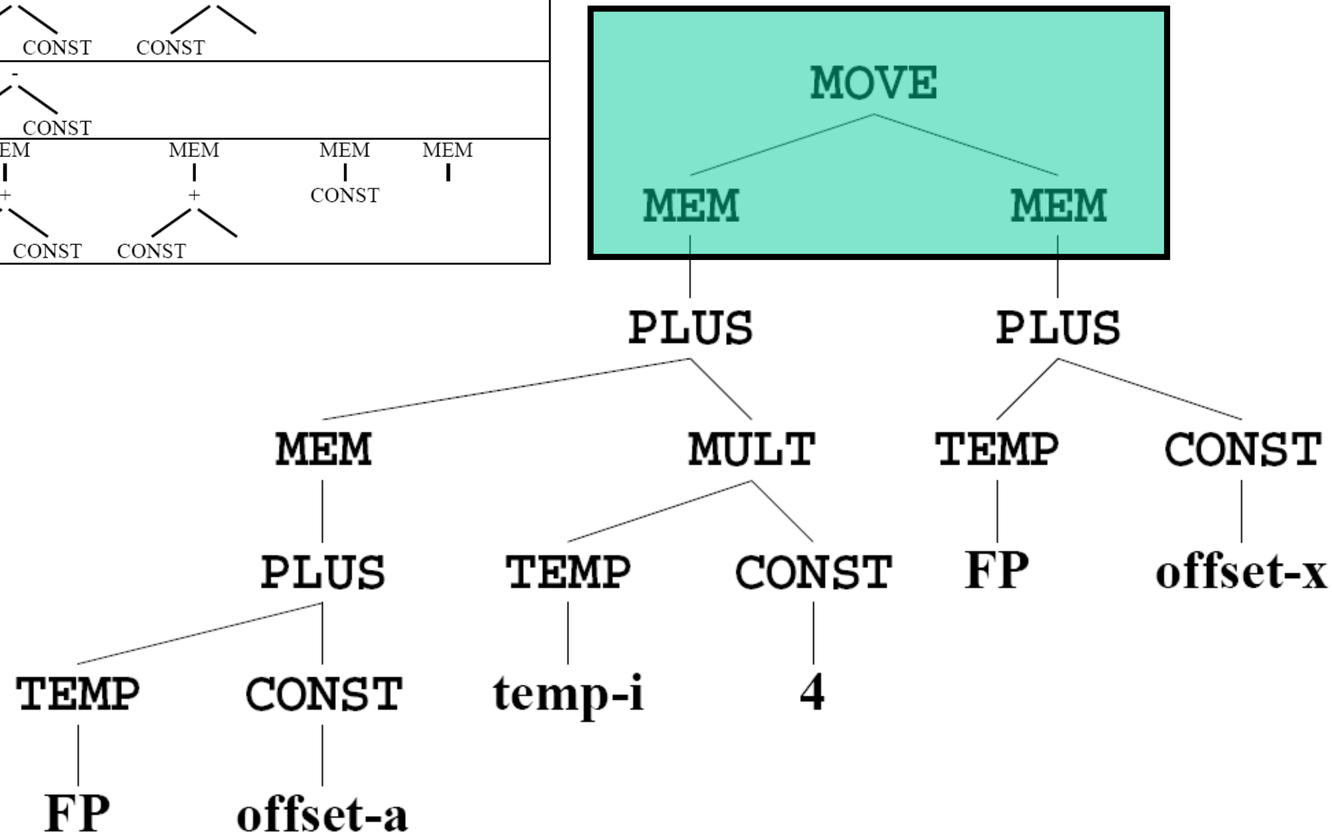
Name	Effect		Trees
—	r_i		TEMP
ADD	r_i	$r_j + r_k$	+
MUL	r_i	$r_j \times r_k$	*
SUB	r_i	$r_j \quad r_k$	-
DIV	r_i	r_j / r_k	/
ADDI	r_i	$r_j + c$	+ CONST; + CONST; CONST
SUBI	r_i	$r_j \quad c$	- CONST
LOAD	r_i	$M[r_j + c]$	MEM + CONST; MEM + CONST; MEM CONST; MEM
STORE	$M[r_j + c]$	r_i	MOVE MEM + CONST; MOVE MEM + CONST; MOVE MEM CONST; MOVE MEM
MOVEM	$M[r_j]$	$M[r_i]$	MOVE MEM MEM



Maximal Munch

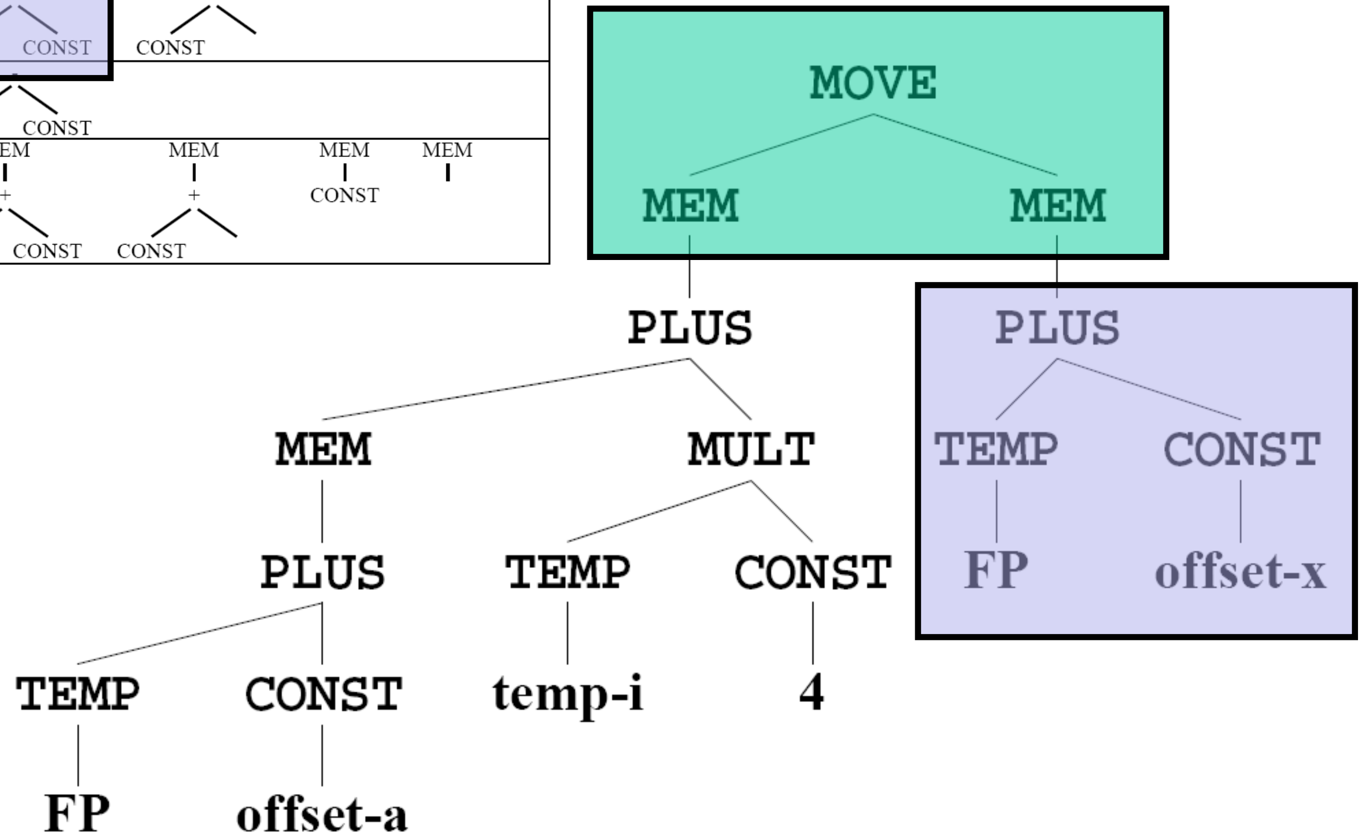






Maximal Munch

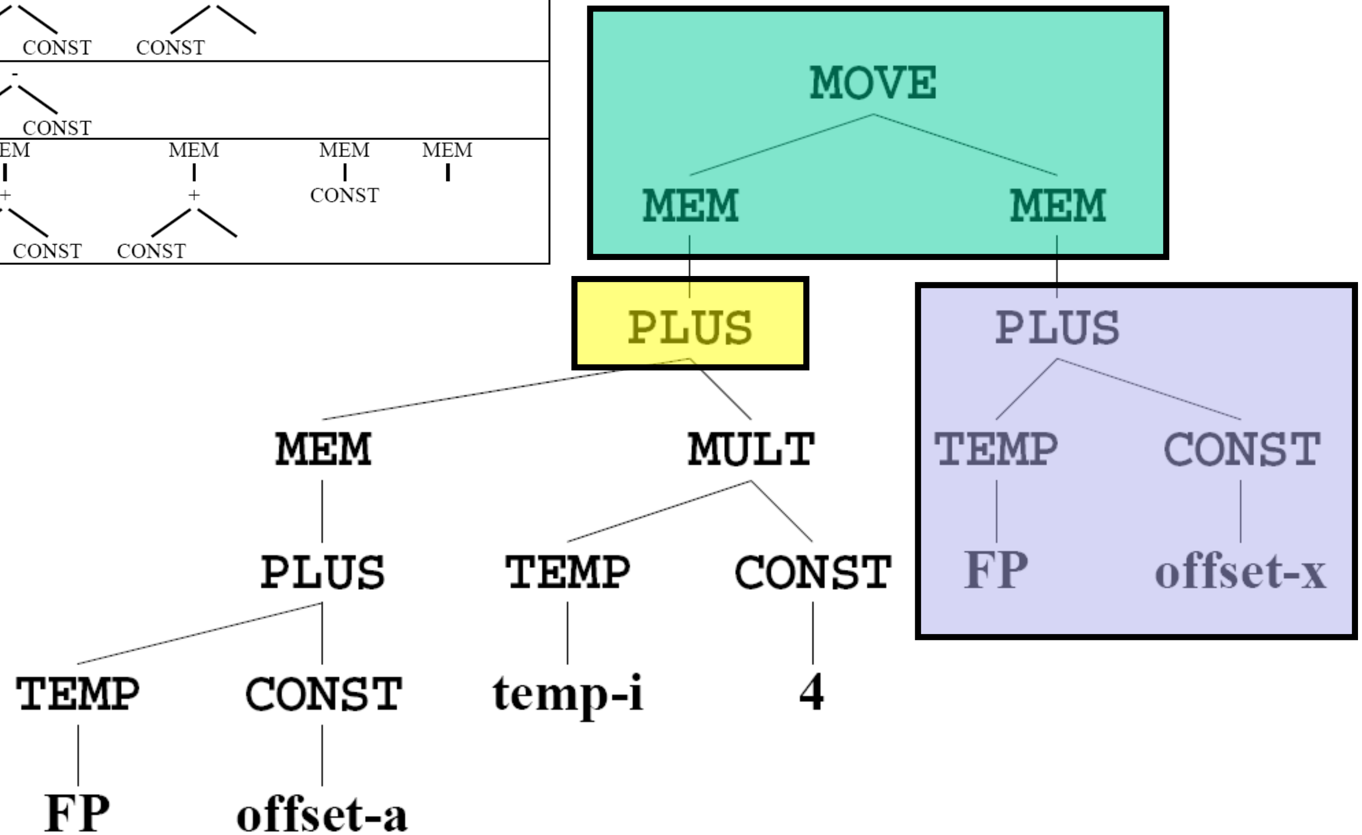
Name	*Effect*	*Trees*
—	r_i	TEMP
ADD	r_i $r_j + r_k$	+
MUL	r_i $r_j \times r_k$	*
SUB	r_i r_j r_k	-
DIV	r_i r_j / r_k	/
ADDI	r_i $r_j + c$	+(_, CONST); +(CONST, _); CONST
SUBI	r_i r_j c	-(_, CONST)
LOAD	r_i $M[r_j + c]$	MEM(+(_, CONST)); MEM(+(CONST, _)); MEM(CONST); MEM
STORE	$M[r_j + c]$ r_i	MOVE(MEM(+(_, CONST)), _); MOVE(MEM(+(CONST, _)), _); MOVE(MEM(CONST), _); MOVE(MEM, _)
MOVEM	$M[r_j]$ $M[r_i]$	MOVE(MEM, MEM)



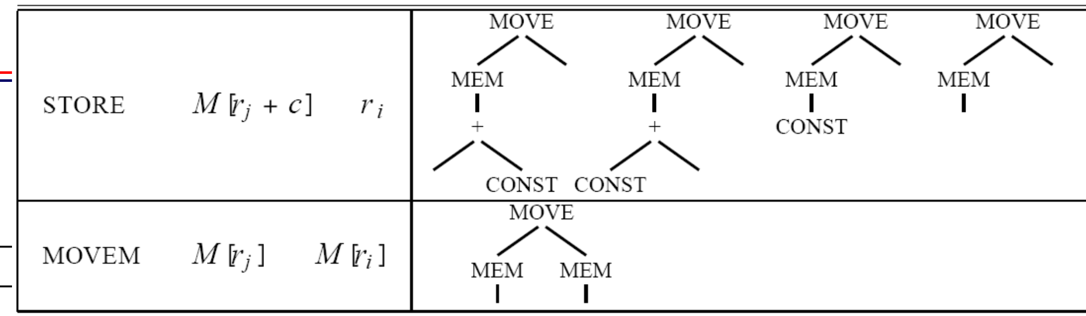
Maximal Munch

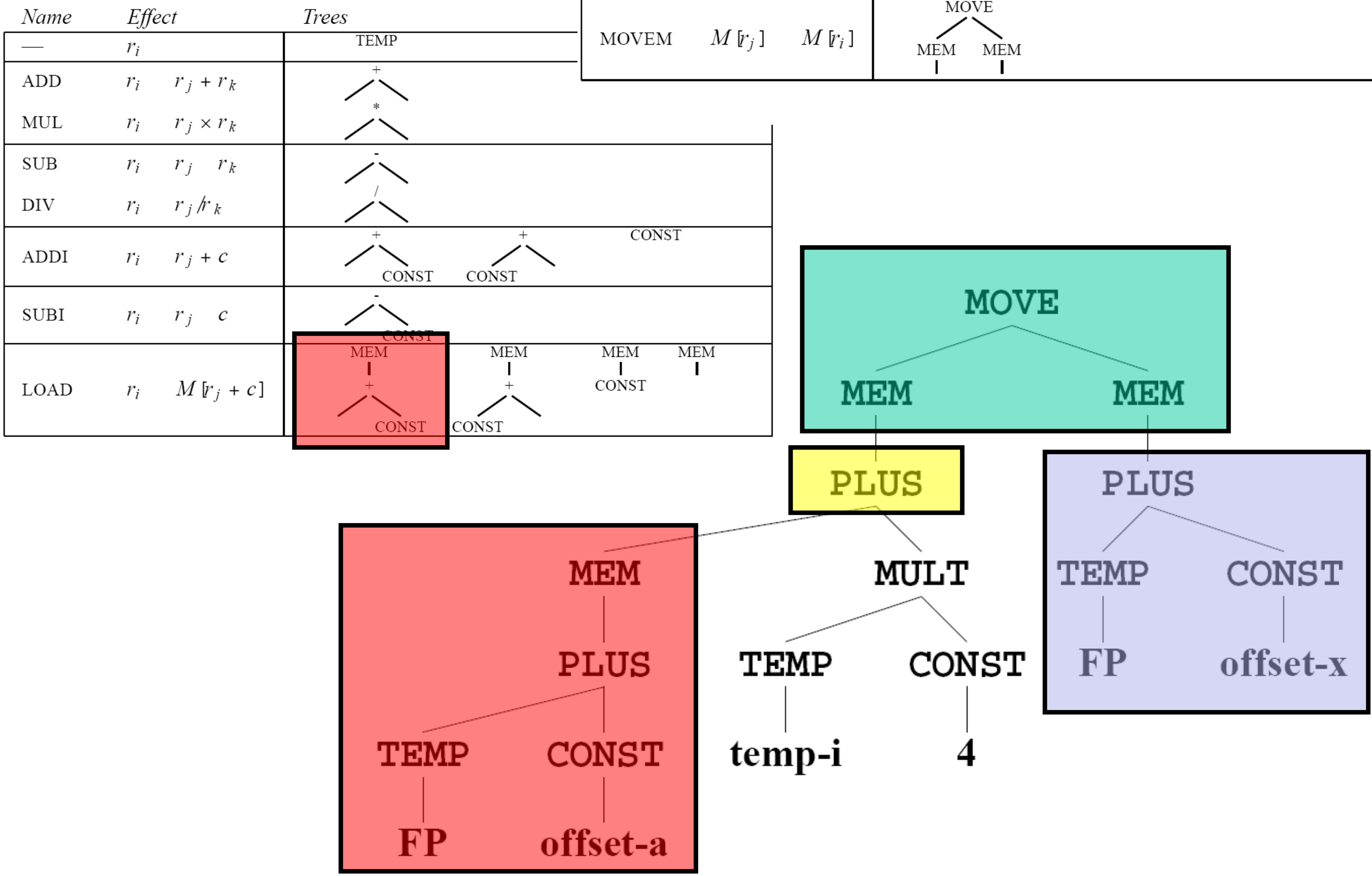
Name	Effect		Trees
—	r_i		TEMP
ADD	r_i	$r_j + r_k$	+
MUL	r_i	$r_j \times r_k$	*
SUB	r_i	$r_j \quad r_k$	-
DIV	r_i	r_j / r_k	/
ADDI	r_i	$r_j + c$	+ (CONST); + (CONST); CONST
SUBI	r_i	$r_j \quad c$	- (CONST)
LOAD	r_i	$M[r_j + c]$	MEM(+(CONST)); MEM(+(CONST)); MEM(CONST); MEM

STORE	$M[r_j + c]$	r_i	MOVE(MEM(+(CONST))); MOVE(MEM(+(CONST))); MOVE(MEM(CONST)); MOVE(MEM)
MOVEM	$M[r_j]$	$M[r_i]$	MOVE(MEM, MEM)



Maximal Munch

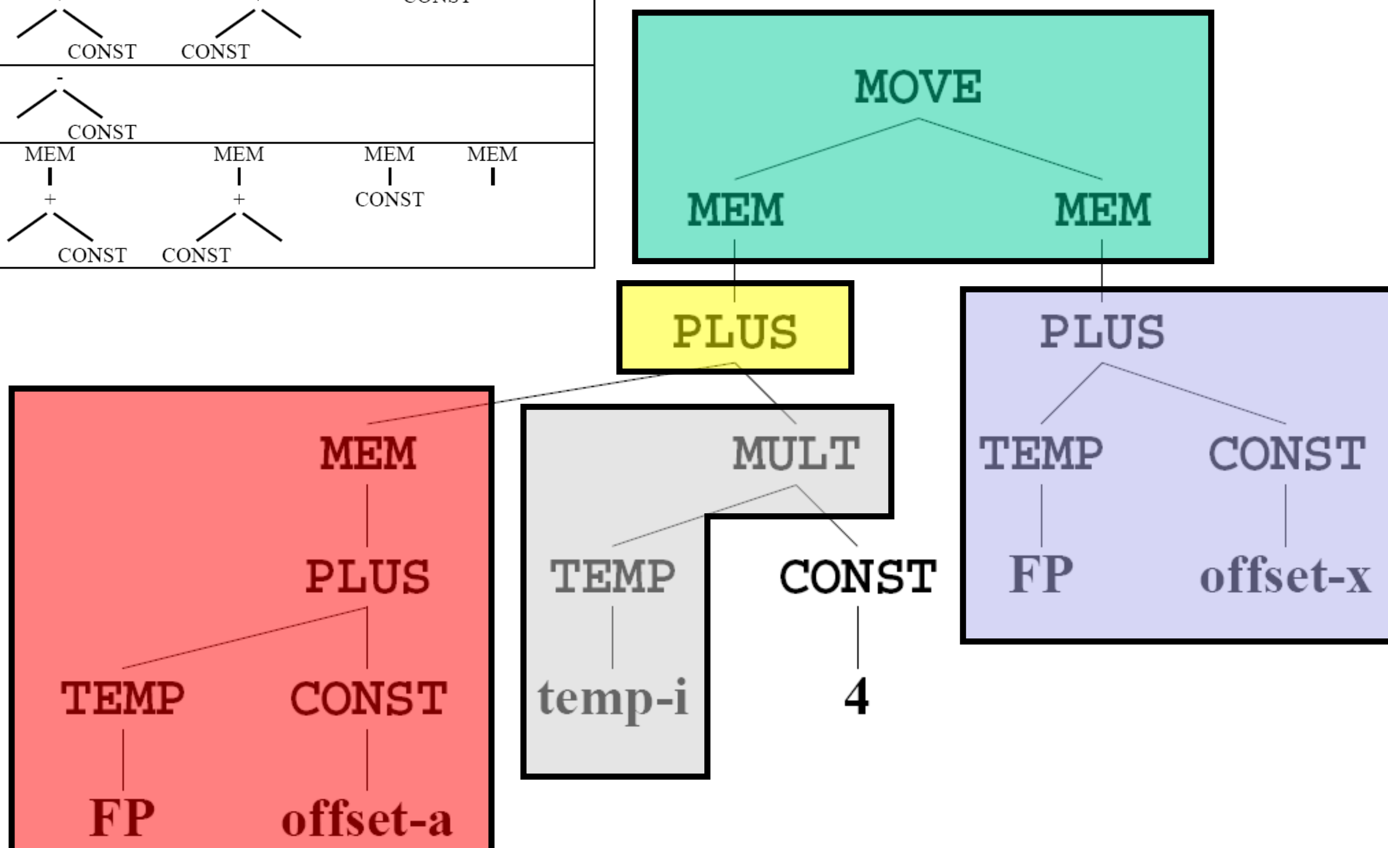




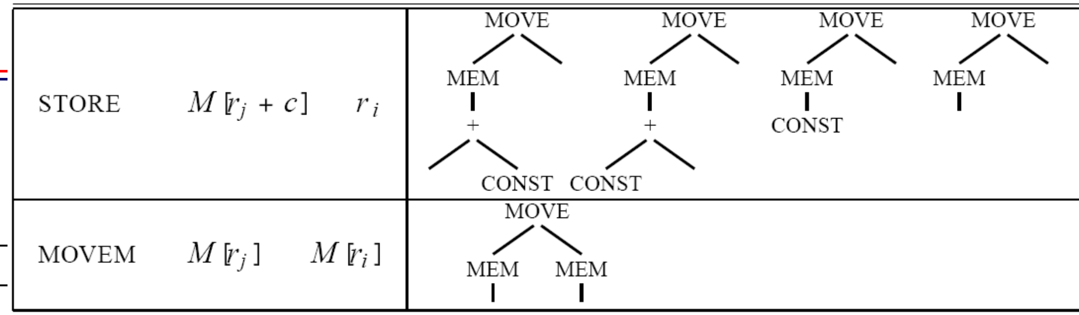
Maximal Munch

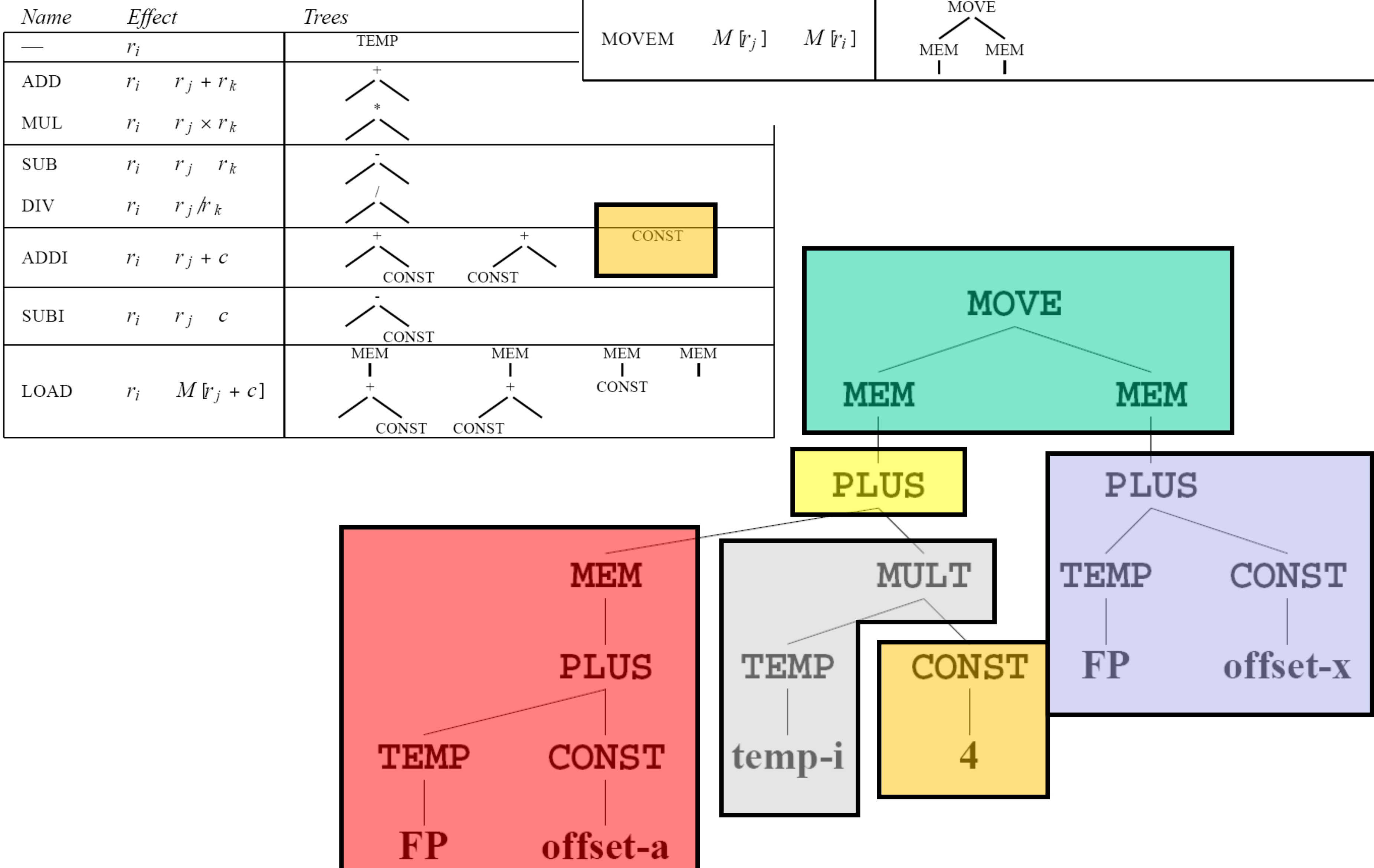
Name	Effect	Trees
—	r_i	TEMP
ADD	$r_i \quad r_j + r_k$	+
MUL	$r_i \quad r_j \times r_k$	*
SUB	$r_i \quad r_j \quad r_k$	-
DIV	$r_i \quad r_j / r_k$	/
ADDI	$r_i \quad r_j + c$	+ CONST; + CONST; CONST
SUBI	$r_i \quad r_j \quad c$	- CONST
LOAD	$r_i \quad M[r_j + c]$	MEM + CONST; MEM + CONST; MEM CONST; MEM

STORE	$M[r_j + c] \quad r_i$	MOVE MEM + CONST; MOVE MEM + CONST; MOVE MEM CONST; MOVE MEM
MOVEM	$M[r_j] \quad M[r_i]$	MOVE MEM MEM



Maximal Munch





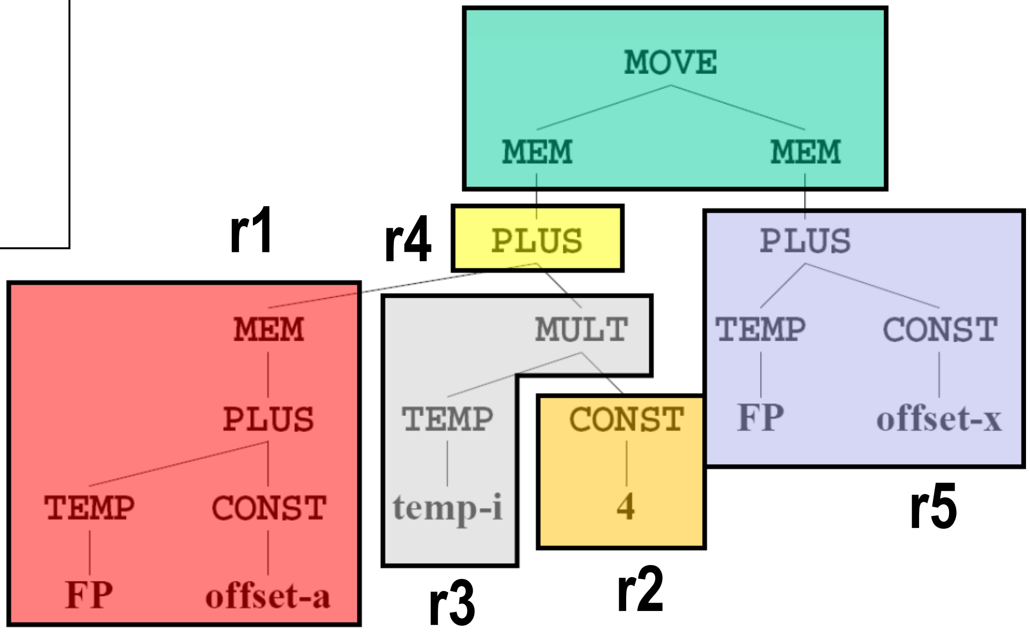
Maximal Munch

To obtain code, assign registers and emit instructions, bottom-up:

```
LOAD     r1  = M[FP + offset_a]

ADDI     r2 = r0 + 4
MUL      r3 = r_i * r2

ADD      r4 = r1 + r3

ADDI     r5 = FP + offset_x
MOVEM  M[r4] = M[r5]
```

5 registers, 6 instructions

optimize register usage
in later phase…



Assembly Representation

```
structure Assem = struct
  type reg = string
  type temp = Temp.temp
  type label = Temp.label

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

Codegen

```
fun codegen(frame)(stm: Tree.stm):Assem.instr list =
let
  val ilist = ref(nil: Assem.instr list)
  fun emit(x) = ilist := x::!ilist
  fun munchStm: Tree.stm -> unit
  fun munchExp: Tree.exp -> Temp.temp
in
  munchStm(stm);
  rev(!ilist)
end
```

Statement Munch

```
fun munchStm(
  T.MOVE(T.MEM(T.BINOP(T.PLUS, e1, T.CONST(c))), e2)
          ) =
     emit(Assem.OPER{assem="STORE M['s0 + " ^
                                int(c) ^ "] = 's1\n",
                     src=[munchExp(e1), munchExp(e2)],
                     dst=[],
                     jump=NONE})
  | munchStm(T.MOVE(T.MEM(e1), T.MEM(e2))) =
     emit(Assem.OPER{assem="MOVEM M['s0] = M['s1]\n"
                     src=[munchExp(e1), munchExp(e2)],
                     dst=[],
                     jump=NONE})
  | munchStm(T.MOVE(T.MEM(e1), e2)) =
     emit(Assem.OPER{assem="STORE M['s0] = 's1\n"
                     src=[munchExp(e1), munchExp(e2)],
                     dst=[],
                     jump=NONE})
...
```

Expression Munch

```
and munchExp(T.MEM(T.BINOP(T.PLUS, e1, T.CONST(c)))) =
      let
        val t = Temp.newtemp()
      in
        emit(Assem.OPER{assem="LOAD 'd0 = M['s0 +" ^
                                      int(c) ^ "]\n",
                                src=[munchExp(e1)],
                                dst=[t],
                                jump=NONE});
        t
      end
```

Expression Munch

```
  | munchExp(T.BINOP(T.PLUS, e1, T.CONST(c))) =
      let
        val t = Temp.newtemp()
      in
        emit(Assem.OPER{assem="ADDI 'd0 = 's0 +" ^
                                      int(c) ^ "\n",
                                src=[munchExp(e1)],
                                dst=[t],
                                jump=NONE});
        t
      end
...
  | munchExp(T.TEMP(t)) = t
```

Optimum Instruction Selection

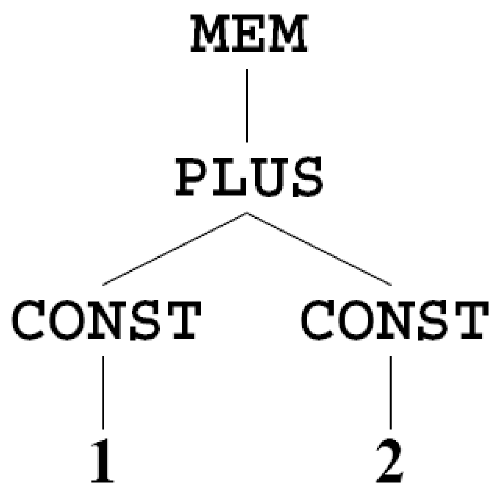
- Find optimum solution for problem (tiling of IR tree) based on optimum solutions for each subproblem (tiling of subtrees)
- Use Dynamic Programming to avoid unnecessary recomputation of subtree costs.
- *cost* assigned to *every* node in IR tree
  - Cost of best instruction sequence that can tile subtree rooted at node.
- Algorithm works bottom-up (Maximum Munch is top-down) - Cost of each subtree s_j (c_j) has already been computed.
- For each tile t of cost c that matches at node n, cost of matching t is:

$$c_t + \sum_{\text{all leaves } i \text{ of } t} c_i$$

- Tile is chosen which has minimum cost.

Optimum Instruction Selection – Example

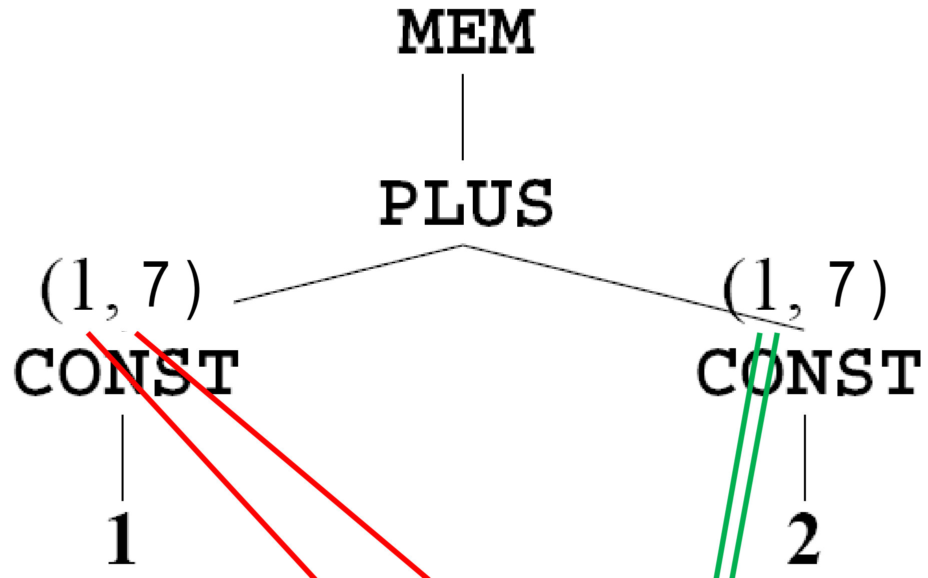
```
MEM(BINOP(PLUS, CONST(1), CONST(2)))

MEM(PLUS(CONST(1), CONST(2)))
```



Optimum Instruction Selection – Example

Step 1: Find cost of root node

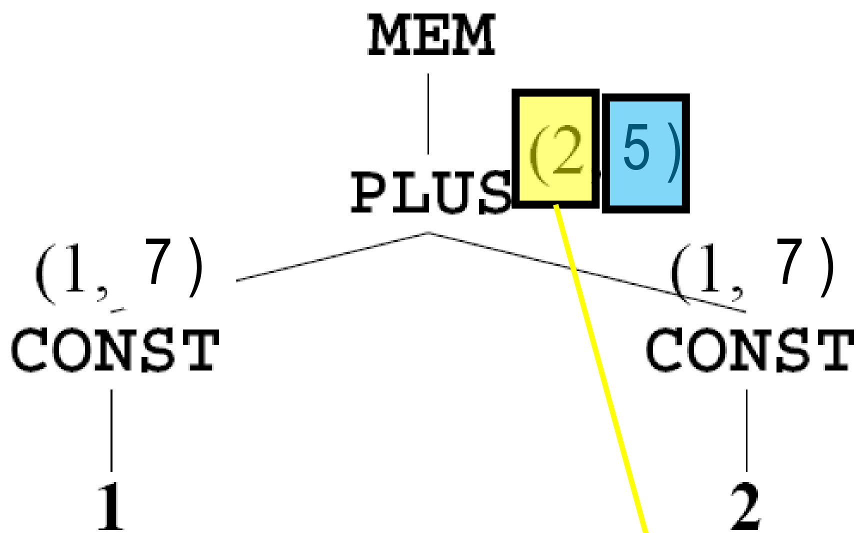
(a,b): a is minimum cost, b is corresponding pattern number



Consider PLUS node:

Pattern	Cost	Leaves Cost	Total
(2) PLUS(e1, e2)	1	2	3
(6) PLUS(CONST(c), e1)	1	1	2
(5) PLUS(e1, CONST(c))	1	1	2

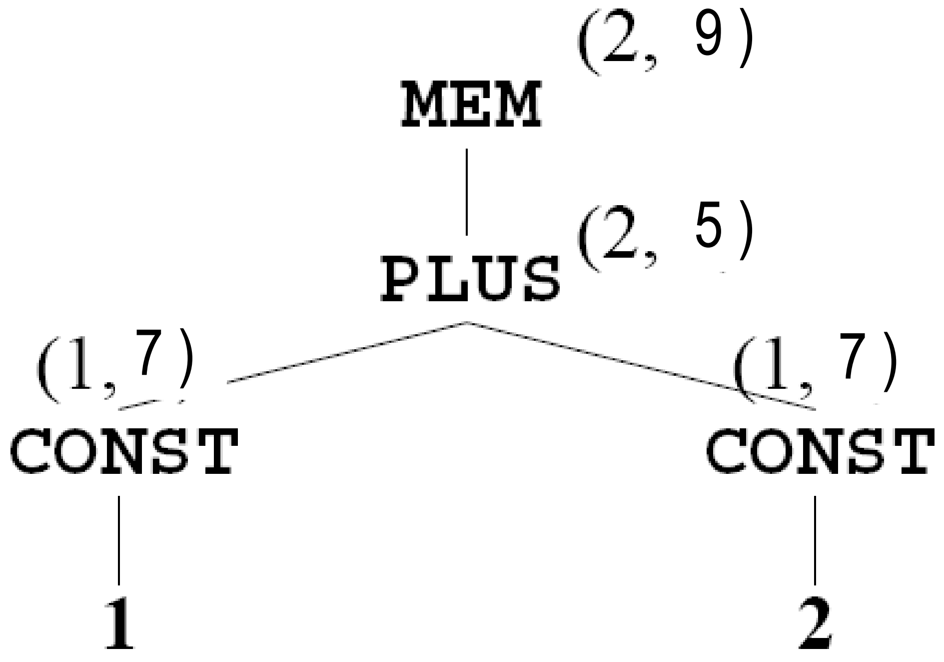
Optimum Instruction Selection – Example



Consider MEM node:

Pattern	Cost	Leaves Cost	Total
(12) MEM(e1)	1	2	3
(9) MEM(PLUS(e1, CONST(c))	1	1	2
(10) MEM(PLUS(CONST(c), e1)	1	1	2

Optimum Insruction Selection – Example

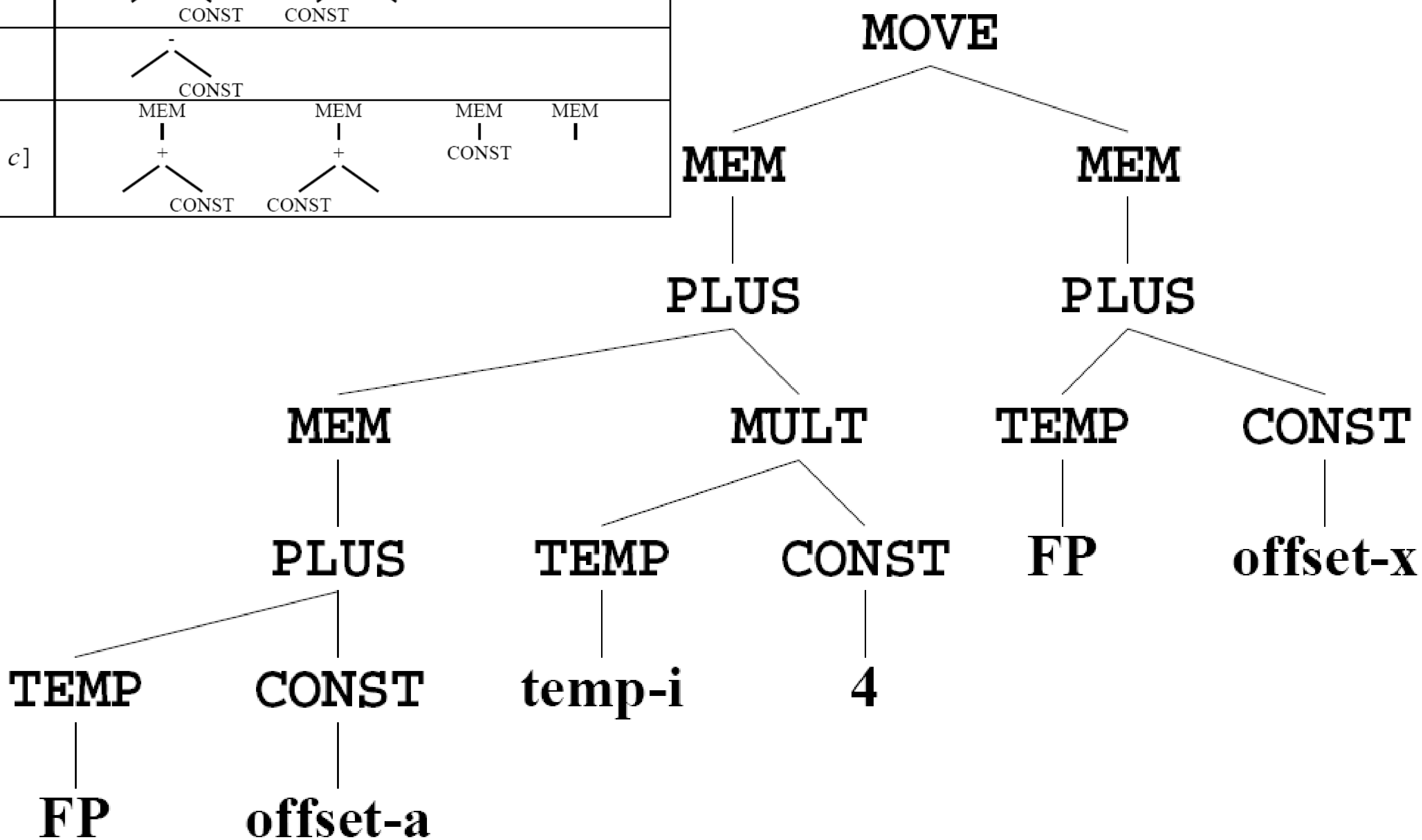


Step 2: Emit instructions

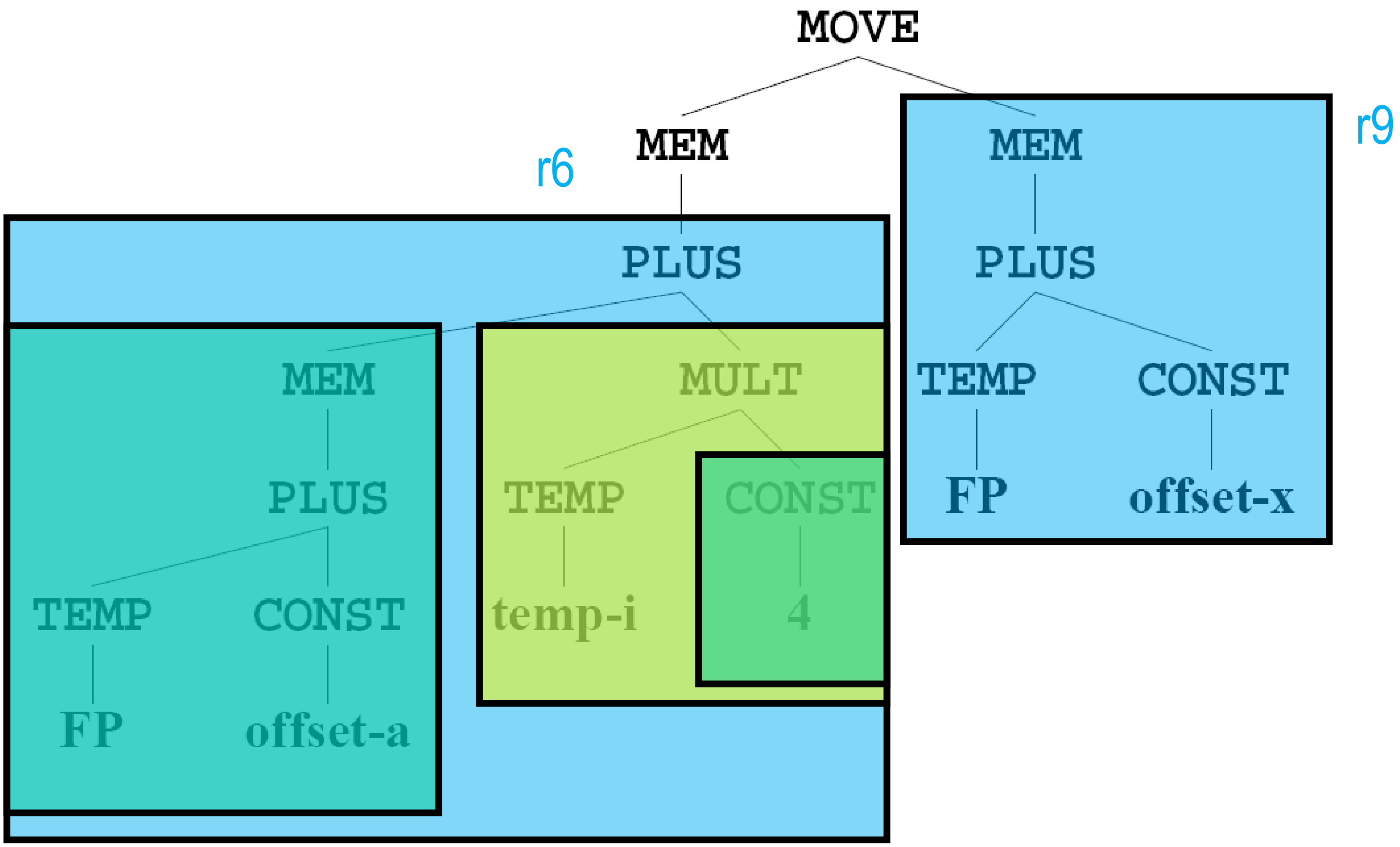
```
ADDI r1 = r0 + 1
LOAD r2 = M[r1 + 2]
```

Optimum Instruction

Name	*Effect*		*Trees*
—	r_i		TEMP
ADD	r_i	$r_j + r_k$	+
MUL	r_i	$r_j \times r_k$	*
SUB	r_i	$r_j \quad r_k$	-
DIV	r_i	r_j / r_k	/
ADDI	r_i	$r_j + c$	+ CONST; + CONST; CONST
SUBI	r_i	$r_j \quad c$	- CONST
LOAD	r_i	$M[r_j + c]$	MEM + CONST; MEM + CONST; MEM CONST; MEM
STORE	$M[r_j + c]$	r_i	MOVE MEM + CONST; MOVE MEM + CONST; MOVE MEM CONST; MOVE MEM
MOVEM	$M[r_j]$	$M[r_i]$	MOVE MEM MEM



Optimum Instruction Selection – Big Example



Optimum Instruction Selection – Big Example

```
LOAD  r3 = M[FP + offset_a]

ADDI  r4 = r0 + 4
MUL   r5 = r4 * r_i

ADD   r6 = r3 + r5

LOAD  r9 = M[FP + offset_x]
STORE M[r6] = r9
```

5 registers, 6 instructions

Optimal tree generated by Maximum Munch is also optimum...