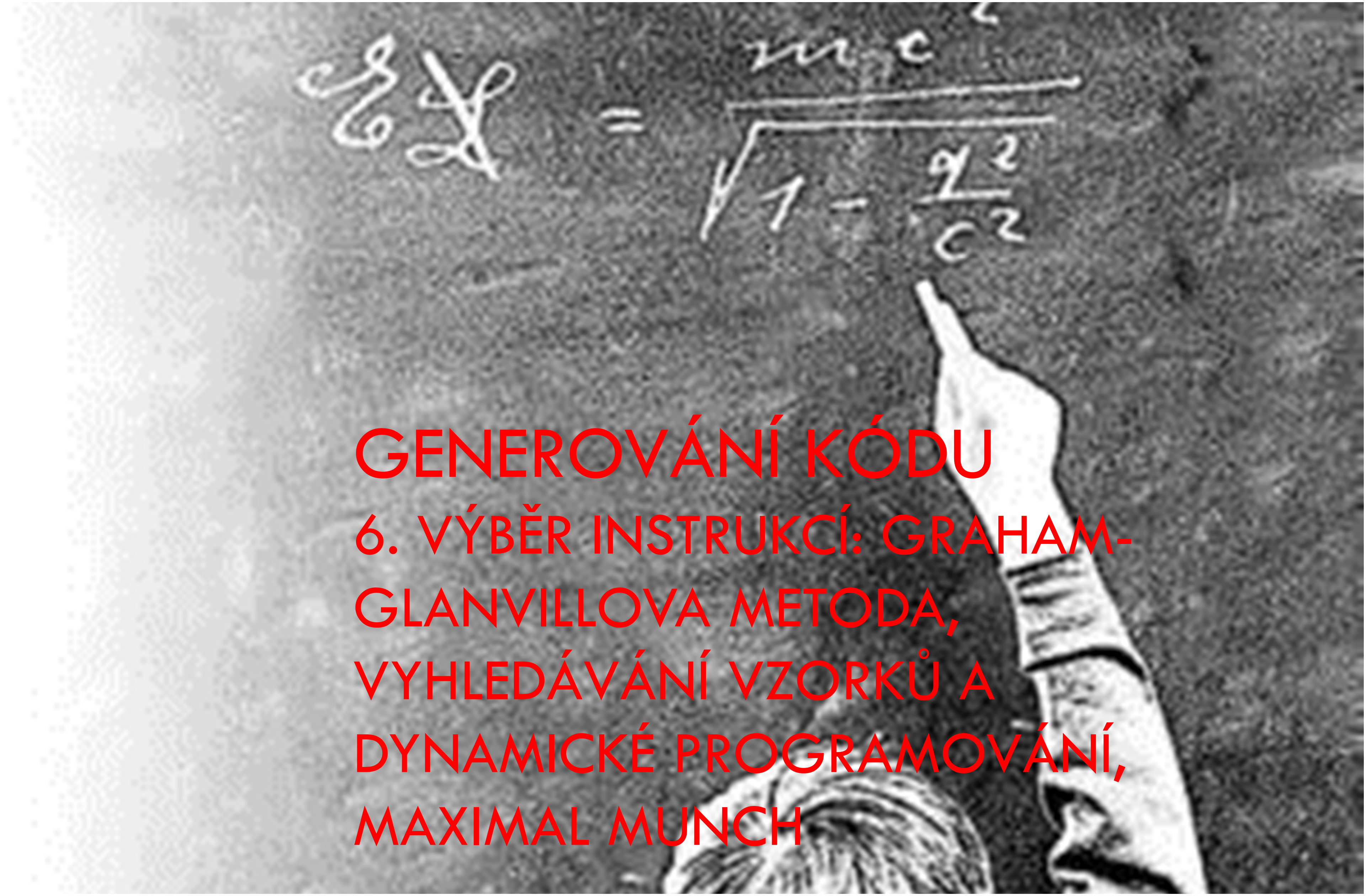


GENEROVÁNÍ KÓDU

6. VÝBĚR INSTRUKCÍ: GRAHAM-GLANVILLOVA METODA, VYHLEDÁVÁNÍ VZORKŮ A DYNAMICKÉ PROGRAMOVÁNÍ, MAXIMAL MUNCH

2011 Jan Janoušek
MI-GEN

Evropský sociální fond
Praha & EU:
Investujeme do vaší budoucnosti

GRAHAM-GLANVILLE METHOD IN DETAILS

Graham-Glanville technique -recapitulation

- The basic method which uses context-free grammars.
- Grammar rules are constructed for prefix notations of patterns of all machine instructions.
- LR(0) parser for the grammar is constructed.
- Reduction by a rule means that the corresponding instruction is selected.
- The created grammar is unambiguous, which means that the constructed LR(0) parser contains many parse conflicts (nondeterminisms).
- The conflicts are resolved by some heuristics so that the minimal number of target instructions would be generated.

Example, contd.

- The IR from our example in prefix notation:

```
:= + A D1 + | + | + P D1 B | + C D1
```

```
Note. Symbol | represents indirection.
```

- A context-free translation grammar which corresponds to the tree patterns is constructed:

```
Null -> := + const R2 R1   {ST const(R2),R1}
Null -> := + R2 const R1   {ST const(R2),R1} ;commutative!
Null -> := const R1        {ST const,R1}
Null -> := R2 R1           {ST 0(R2),R1}
```

Example, contd.

```
R1 -> := const                {LD #const,R1}
R1 -> | + const R2            {LD const(R2),R1}
R1 -> | + R2 const            {LD const(R2),R1}
R2 -> := R1                   {LR R2,R1}

R1 -> + | + const R2 R1       {ADD const(R2),R1}
R1 -> + | + R2 const R1       {ADD const(R2),R1}
R1 -> + R1 | + const R2       {ADD const(R2),R1}
R1 -> + R1 | + R2 const       {ADD const(R2),R1}

R1 -> + const R1              {ADD #const,R1}
R1 -> + R1 const              {ADD #const,R1}

R1 -> + R2 R1                 {ADD R2,R1}
R1 -> + R1 R2                 {ADD R2,R1}

+rules for the other registers
```

Example, contd.

- LR(0) parser is constructed.
- For the construction of LR(0) parser, see additional slides (see subject MI-SYP).
- The resulting parser contains conflicts.

Resolving conflicts

- The conflicts are resolved so that a minimal number of instructions would be generated. Therefore, if there is a conflict (ie. more instructions can be selected) the instruction with the biggest pattern is preferred.

  - **In case of shift-reduce conflict: shift is preferred.**

  - **In case of reduce-reduce conflict: an instruction with the biggest right-hand side is preferred.**

Example, contd.

String `:= + A D1 + | + | + P D1 B | + C D1` is read. The following sequence of transitions is performed:

(pushdown store, input)

```
(#,  := + A D1 + | + | + P D1 B | + C D1)├
(# :=,  + A D1 + | + | + P D1 B | + C D1)├
(# := +,  A D1 + | + | + P D1 B | + C D1)├
(# := + A,  D1 + | + | + P D1 B | + C D1)├
(# := + A D1,  + | + | + P D1 B | + C D1)├
(# := + A D1 +,  | + | + P D1 B | + C D1)├
(# := + A D1 + |,  + | + P D1 B | + C D1)├
(# := + A D1 + | +,  | + P D1 B | + C D1)├
(# := + A D1 + | + |,  + P D1 B | + C D1)├
(# := + A D1 + | + | +,  P D1 B | + C D1)├
```

Example, contd.

```
(#  :=  +  A  D1  +  |  +  |  +  P,  D1  B  |  +  C  D1)├
(#  :=  +  A  D1  +  |  +  |  +  P  D1,  B  |  +  C  D1)├  ;  LD  P(D1),R1
(#  :=  +  A  D1  +  |  +  R1,  B  |  +  C  D1)├
(#  :=  +  A  D1  +  |  +  R1  B,  |  +  C  D1)├
(#  :=  +  A  D1  +  |  +  R1  B  |,  +  C  D1)├
(#  :=  +  A  D1  +  |  +  R1  B  |  +,  C  D1)├
(#  :=  +  A  D1  +  |  +  R1  B  |  +  C,  D1)├
(#  :=  +  A  D1  +  |  +  R1  B  |  +  C  D1,  ε)├     ;  LD  C(D1),R2
(#  :=  +  A  D1  +  |  +  R1  B  R2,  ε)├             ;  ADD  B(R1),R2
(#  :=  +  A  D1  R2,  ε)├                             ;  ST  A(D1),R2
(#  Null,  ε)
```

Example, contd.

The produced code:

```
LD P(D1),R1
LD C(D1),R2
ADD B(R1),R2
ST A(D1),R2
```

However, it is not optimal (we use two registers, one register suffices):

```
LD P(D1),R1
LD B(R1),R1
ADD C(D1),R1
ST A(D1),R1
```

SELECTING INSTRUCTIONS BY TREE PATTERN MATCHING AND DYNAMIC PROGRAMMING

Introduction

- Code selection problem has been presented as a problem of tiling an IR tree by tree patterns, which correspond to particular machine instructions.

- As we have seen, we can assign **a cost** to each pattern.

- Graham-Glanville technique tries to cover the tree by a largest possible patterns and therefore to generate minimal number of instructions. But it does not follow an optimal tiling.

- The computing of tiling a tree with the minimal cost (**the total cost of all used patterns is to be the minimal one**) can be done by combination of tree pattern matching and dynamic programming:

Introduction

- Another bottom-up method
- **Produces optimal tiling**
- **Suitable for CISC processors, which has rather compilcated instructions**

Note on tree pattern matching

- The task of tree pattern matching (TPM) is **to find all occurences of given tree patterns in an input subject tree.** Many TPM methods have been proposed:
  - Basic TPM algorithms: Hofmann, ODonell 1982
  - TPM using automata:
    - Finite tree automata
    - Standard pushdown automata reading a linear notation of the tree (arbology)
- All these methods can be used in the selection code method in question

The Dynamic Programming Algorithm

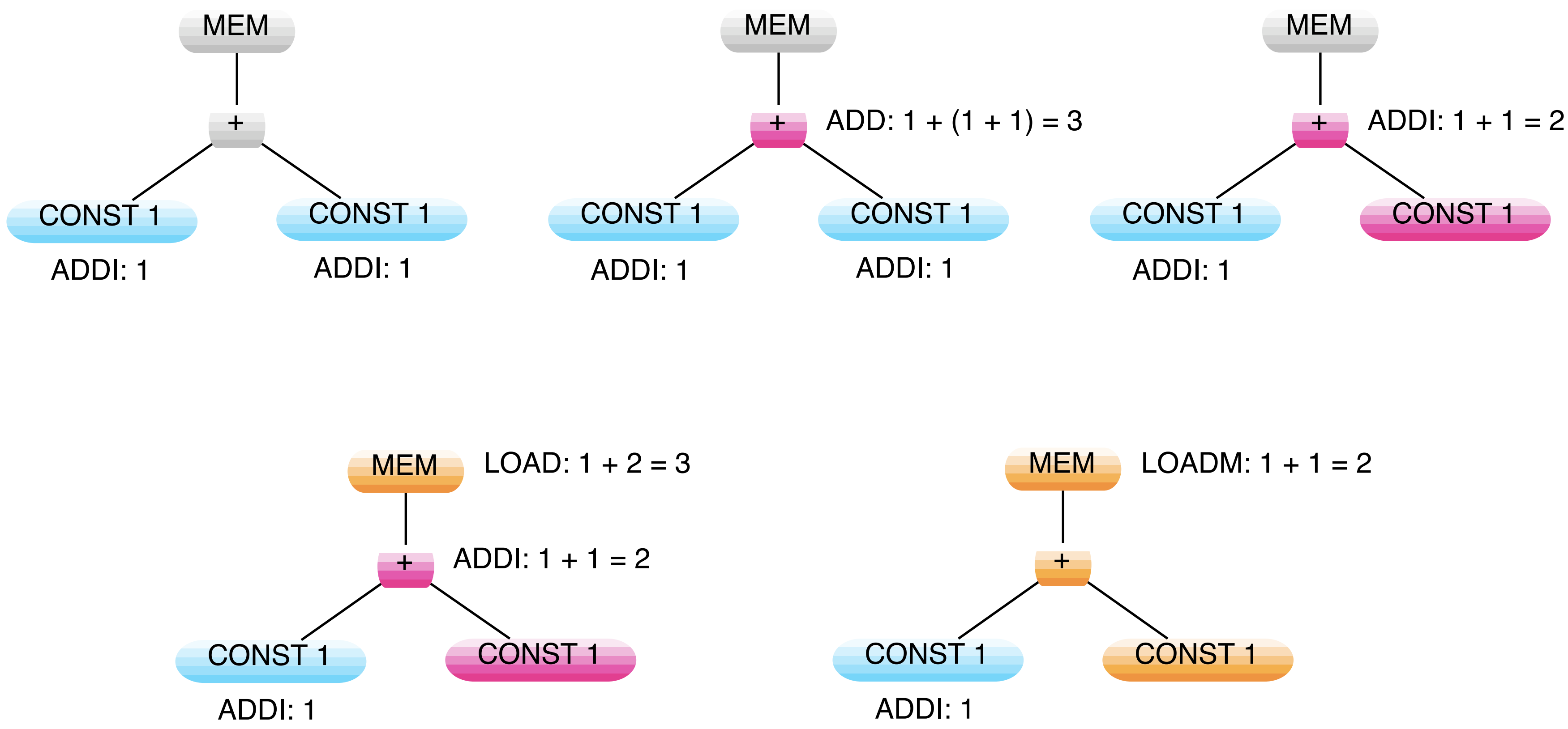
- We compute a minimal **cost** to every node in the tree in a bottom-up fashion.

- **Bottom-up traversal:**
  - for each tile *t* of cost c that matches at node *n*
  - *ci* = cost of each subtree corresponding to the leaves of *t*
  - cost of *n* = c $+\sum ci$

- **2nd traversal:**

Traverse the tree using costs and associated instructions to generate the target code.

Dynamic programming - example



MAXIMAL MUNCH METHOD

Maximal munch method

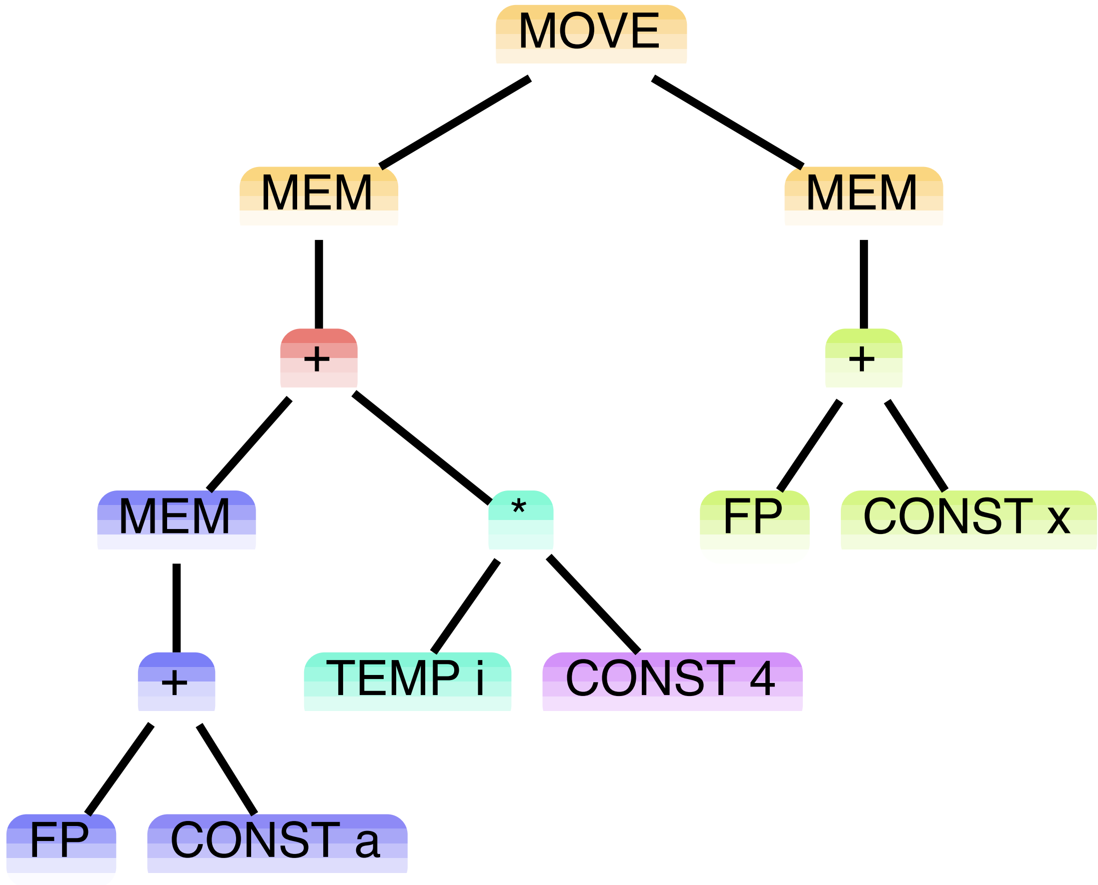
- Construct tree patterns for each machine instruction
- Order pattern trees by size, largest first
- Top-down method, starting at the root node, find the largest pattern that fits.
- Cover the root node, and perhaps several other nodes, with this tile. Repeat for each subtree.
- Traverse tree top-down and emit instructions (in reverse order)

Maximal munch

- Easy to understand and to implement (pattern matching).
- top-down method
- Fast method, good result with a RISC instruction set, but does not produce an optimal tiling in general.

Maximal munch - example

Maximal munch - example



- $r_1 \leftarrow M[fp + a]$
- $r_2 \leftarrow r_0 + 4$
- $r_2 \leftarrow r_i \times r_2$
- $r_1 \leftarrow r_1 + r_2$
- $r_2 \leftarrow fp + x$
- $M[r_1] \leftarrow M[r_2]$