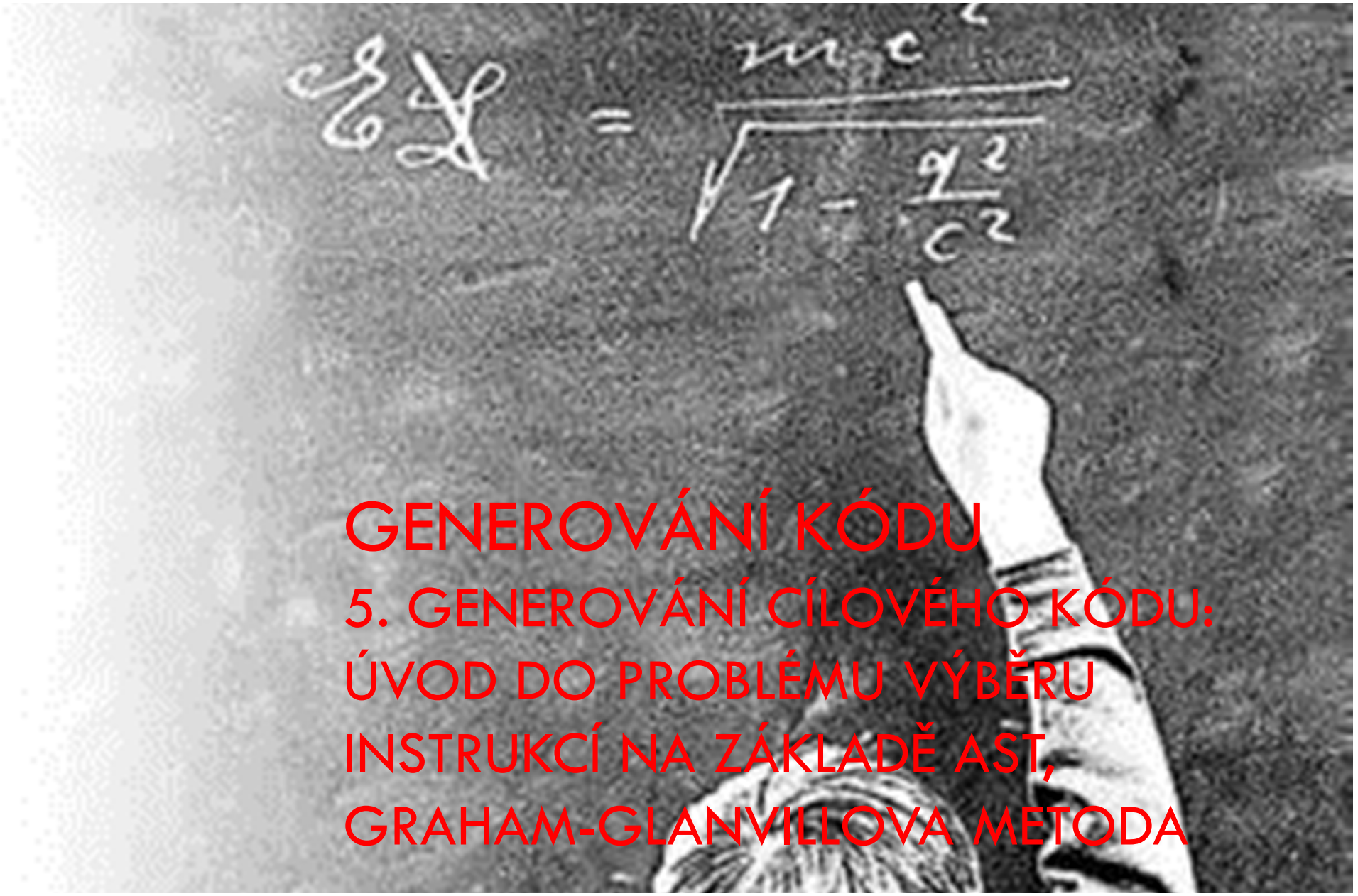


GENEROVÁNÍ KÓDU

5. GENEROVÁNÍ CÍLOVÉHO KÓDU: ÚVOD DO PROBLÉMU VÝBĚRU INSTRUKCÍ NA ZÁKLADĚ AST, GRAHAM-GLANVILLOVA METODA

2011 Jan Janoušek
MI-GEN

OPPA PRAHA PRAGUE PRAGA PRAG EVROPSKÁ UNIE
Evropský sociální fond
Praha & EU:
Investujeme do vaší budoucnosti

PROPERTIES OF TARGET LANGUAGE, COSTS

Simple target machine model - instructions

There are many kinds of real processors (of types CISC, RISC,…).
We consider a simple model with the most common instructions.

- Registers r0, r1, …, r_n-1

- Most common types of instructions:

  - Load operations:
    - **LD dst, addr** dst=addr
    - LD r, x
    - LR r1,r2

  - Store operations:
    - ST addr, r x=r

Simple target machine model – instructions

- Computation operations of the form OP dst, src1,src2
  - OP can be ADD, SUB
  - for example SUB r1,r2,r3 r1=r2-r3
- Unconditional jumps BR L (BR stands for branch)
- Conditional jumps
  - Bcond r,L cond stands for a common test
  - for example BLTZ r,L causes jump to label L if the value in register r is less than zero

Simple target machine model – address modes

Addresses in instructions can be:

- variable name x or related – memory location of x
- indexed address – LD r1, x(r2) r1=contents (x+contents(r2))
  - for example for arrays or for following pointers
- indirect addressing mode *r1
  - for example LD r1, *r2 r1=contens(contents(r2))
- constants addressing modes – for constants
  - LD r1, #100 r1=100

PROGRAM AND INSTRUCTION COSTS

- A **cost** is associated with running program. The cost depends on what aspect of a program we are interested (running time, size, power comsuption,…).

- Often particular instructions have their costs. Examples:
  - LD r0, r1 copying the contents between registers, cost=1 because no additional memory is required
  - LD r0, M loads the contents from memory M into r0, cost=2
  - LD r0, *100(r2) cost=2

GENERATOR OF THE TARGET CODE

GENERATOR OF THE TARGET CODE

- a critical part of the compiler,
- **tricky** because for a particular CPU instruction set and other its properties (registers, pipeling, …) the task to generate a best possible code might be unresolvable or NP-hard in general,
- hard to debug because the output is assembly code,
- specific for each architecture supported by the compiler.

Tasks of the target code generator

- Tasks to be performed by the target code generator
  - **Instructions selection**
  - **Registers allocation**
  - **Instruction scheduling** (ILP – instruction-level parallelism: pipelines, vector processors SIMD, super-scalar processors MIMD,…)

Types of the target code generator

- Code generators can be:
  - hand-written
    - difficult for retargeting and future modifications
    - can be optimized especially for the given target architecture – may generate almost optimal target code
  - generated
    - based on a description of the translation of the IR to the target language - compiler backend generators
    - several principles used in practice
    - the best ones generate codes almost as if it were produces by hand-written
  - some parts are hand-written, other generated

Instruction selection

Basic approaches to instruction selections

- Naïve approch
  - 1:N – N target instructions are generated from each instruction of IR; instructions to be generated are fixed for each IR instruction (see course BI-PJP and generating target code by traversing AST)
  - simple, but very poor and inefficient code

- Conventional approach

  - M:N – N instructions from M instructions
  - Mapping of IR to instructions of the target language
  - Combinatorial problem – heuristics must be used

Some main methods of instruction selection

1980 *Graham* and *Glanville*:
Based on bottom-up shift-reduce (LR) parser

1984 *Davidson* and *Fraser*:
macro-expansion of trees to naive Register Transfer Lists,
followed by declarative code improvement (GCC).

1989 *TWIG - Aho, Ganapathi, Tjiang*:
tree pattern matching and dynamic programming.

1992 *BURG - Fraser, Henry, Proebsting*:
BURS (Bottom-Up Rewrite System).

1994 *Ferdinand, Seidl, Wilhelm*: Tree automata for code select.

History of instruction selection

Many formalisms are used for describing configuration files of instruction selection generators:

- Context-free grammars (Graham-Glanville based approaches)
- Register Transfer Lists (Davidson and Fraser based approaches)
- Rewriting systems (Fraser, Henry, Proebsting)
- Tree regular grammars (Ferdinand, Seidl, Wilhelm)
- …

Instruction selection from AST

We present ideas of the main methods for selecting instructions from a given AST

Naïve approch

Generating code by traversing AST
See BI-PJP course

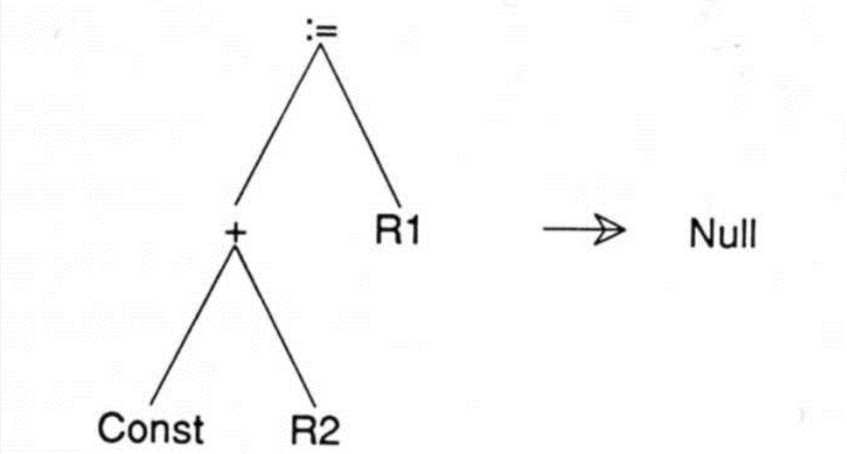
Conventional approach - Target instructions as tree patterns

Target instructions as tree patterns

- The IR is an AST (or DAG).
- AST is to be translated into a sequence of target language instructions.
- Each target language instruction is represented by corresponding tree patterns.
- Tree pattern matching is performed so that the IR would by completely covered by patterns of instructions.
- Because of variety of target language instructions, most machines have many ways to do the same thing.
- If we use code generator generator it is desired that the generated code would be competetive with a code generated by hand-code generators. The problem is that the selection of the optimal instructions can be unresolvable in general, or is often NP-hard – using heuristics is necessary.

Example – instruction `ST const(R2), R1`

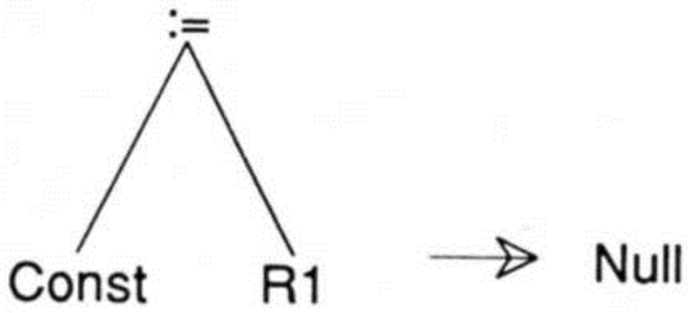
- Example: Consider instruction `ST const(R2), R1`
- Register R1 is stored into indexed memory (const+R2)
- Tree pattern of this instruction:



- If the entire subtree is completely matched, it can be replaced by Null

Example – instruction `ST const, R1`

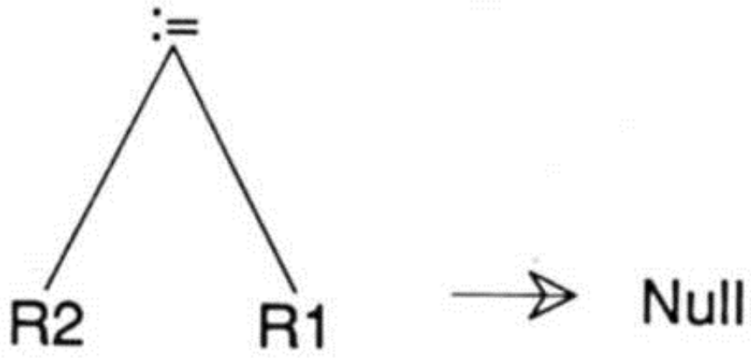
- Example: Consider instruction `ST const, R1`
- Register R1 is stored into memory (const)
- Tree pattern of this instruction:



- If the entire subtree is completely matched, it can be replaced by Null

Example – instruction `ST 0(R2), R1`

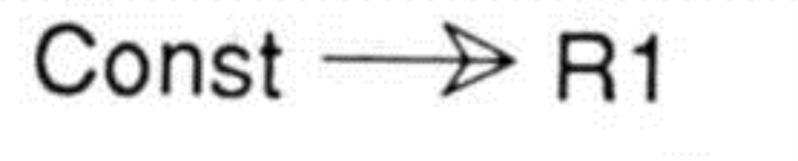
- Example: Consider instruction `ST 0(R2), R1`
- Register R1 is stored into memory (R2)
- Tree pattern of this instruction:



- If the entire subtree is completely matched, it can be replaced by Null

Example – instruction `LD #const, R1`

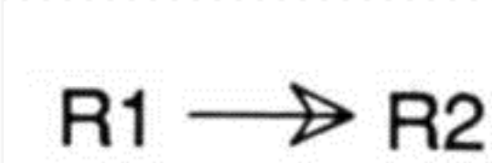
- Example: Consider instruction `LD #const, R1`
- A constant #const is loaded into register R1
- Tree pattern of this instruction:



- If the entire subtree is completely matched and its result is stored in the register R1

Example – instruction `LR R2, R1`

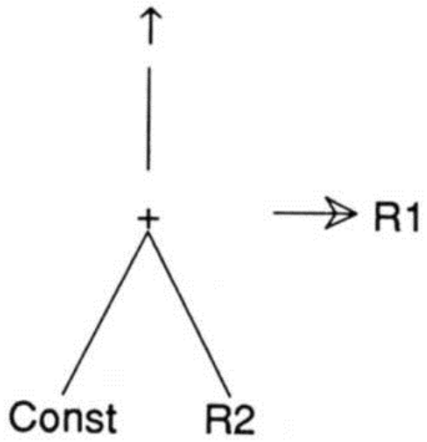
- Example: Consider instruction `LR R2, R1`
- A value is moved form register R1 into register R2
- Tree pattern of this instruction:



- If the entire subtree is completely matched and its result is stored in the register R2

Example – instruction `LD const(R2), R1`

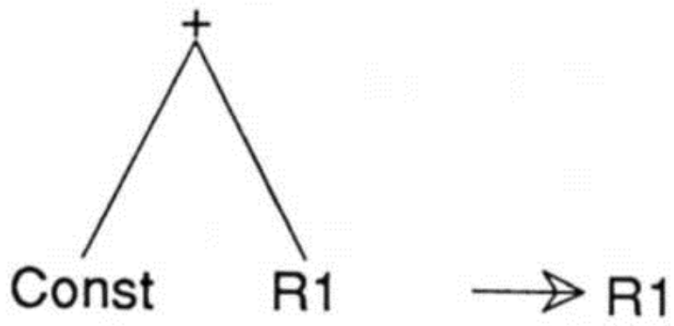
- Example: Consider instruction `LD const(R2), R1`
- A value from indexed memory is loaded into register R1
- Tree pattern of this instruction:



- If the entire subtree is completely matched and its result is stored in the register R1

Example – instruction `ADD #const, R1`

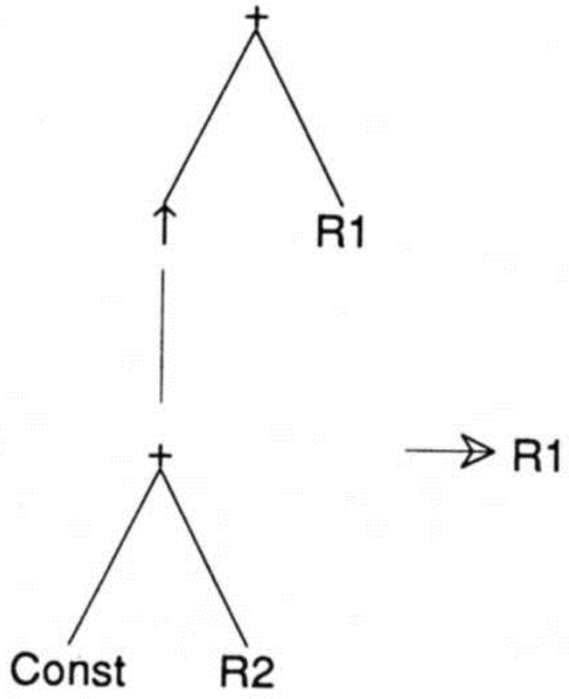
- Example: Consider instruction `ADD #const, R1`
- A constant #const is added to register R1
- Tree pattern of this instruction:



- If the entire subtree is completely matched and its result is stored in the register R1

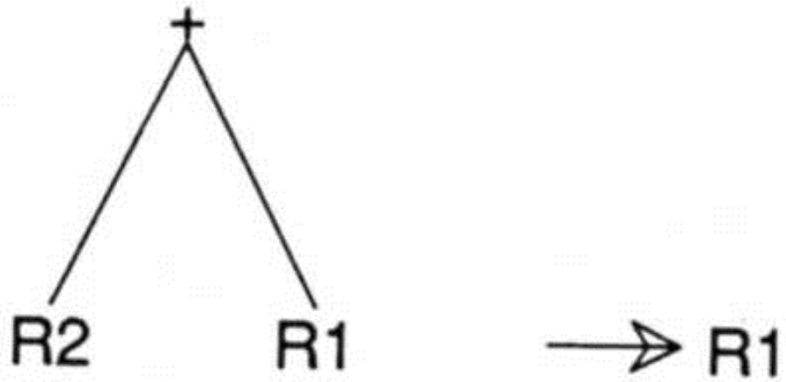
Example – instruction `ADD const(R2), R1`

- Example: Consider instruction `ADD const(R2), R1`
- The value from an indexed memory is added to register R1
- Tree pattern of this instruction:



Example – instruction `ADD R2, R1`

- Example: Consider instruction `ADD R2, R1`
- The value from register R2 is added to register R1
- Tree pattern of this instruction:



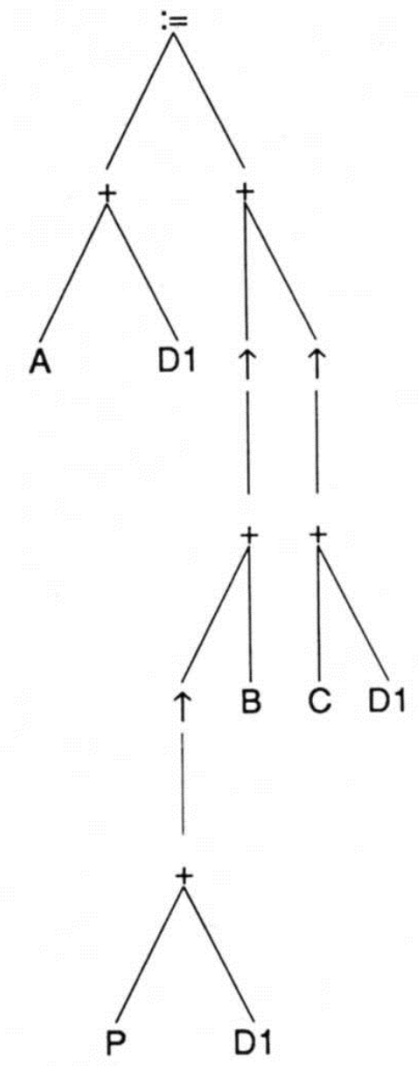
- If the entire subtree is completely matched and its result is stored in the register R1

Example, contd.

- Generate target code for statement
  A= *P.B + C. IR:

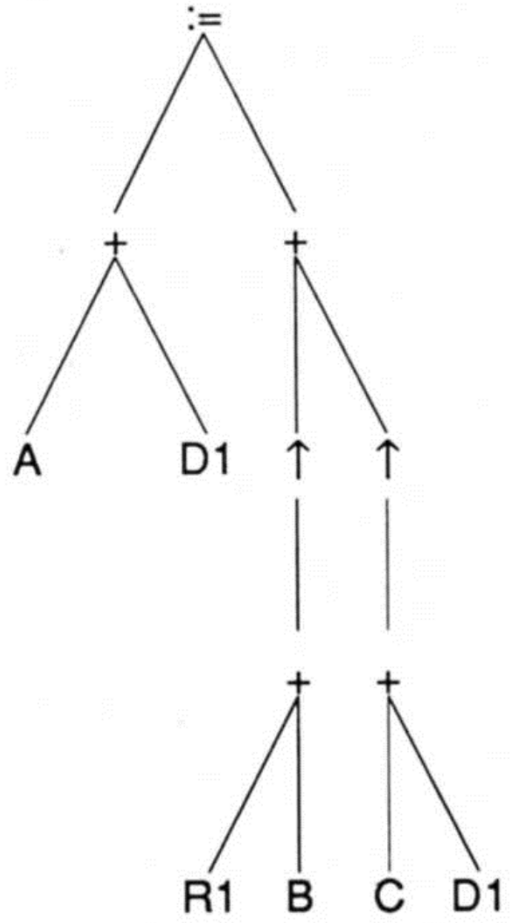
D1 is the register containing the beginning address of the corresponding activation record. D1 is not allowed to be overwritten.

The task is to match subtrees until Null is produced.



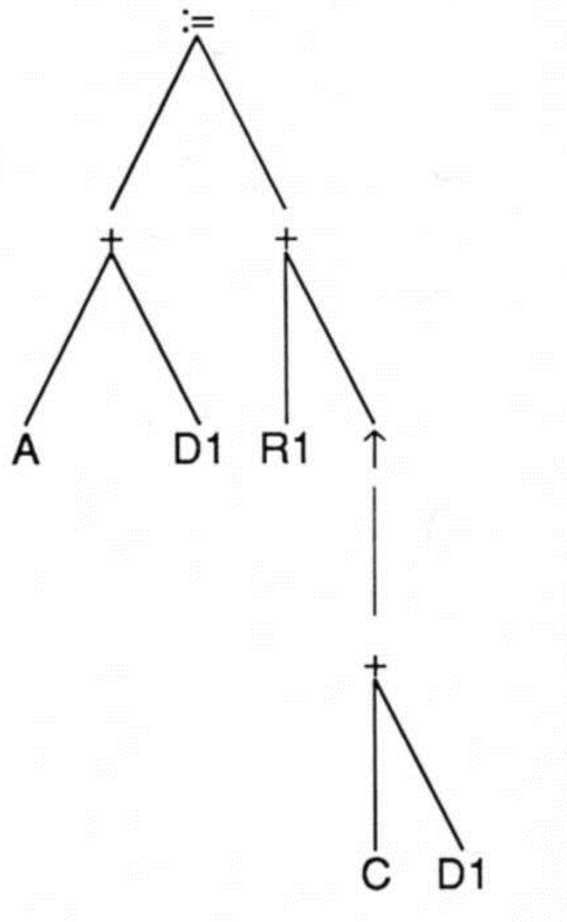
Example, contd.

- `LD P(D1),R1`



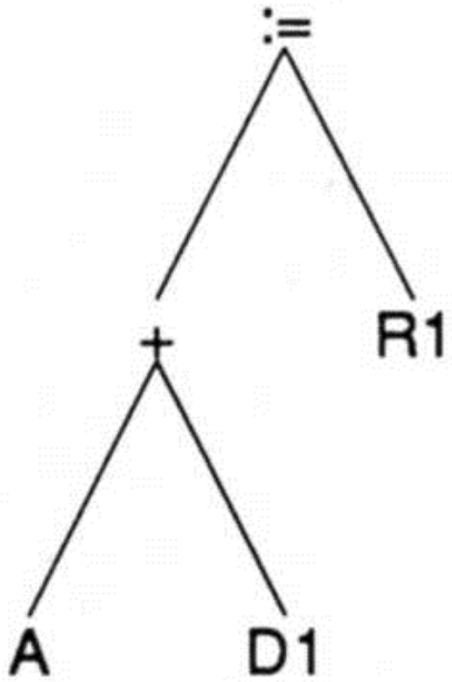
Example, contd.

- `LD B(R1),R1`



Example, contd.

- `ADD C(D1),R1`



Example, contd.

- `ST A(D1),R1`

Null

Example, contd.

- The target code produced is:

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

- However, other code could have been produced, if we match the expressions in the tree in a different order:

```
LD P(D1),R1
LD B(R1),R1
LD C(D1),R2          ; here is the change
ADD R2,R1            ; here is the change
ST A(D1),R1          ; question - what is better?
```

Notes.

- Generally, the target instructions selector tries to find a way to reduce the tree to Null.
- If a one-pass approach is used, the selector must use heuristics to decide which way of code to produce, sometimes selecting an instruction does not lead to a possible code and the generator must even do backtracking.

- Many methods exist and many of them do more than one passes over the tree.

Instruction selection from trees (using tree pattern matching and tiling IR trees)

General principles once again:

- Work on AST or on DAG
- Use patterns to describe semantics of target instructions
- Cover (tile) the AST with possibly smallest/cheapest set of patterns

Graham-Glanville technique

Graham-Glanville technique - main properties

- Context-free grammar based one-pass method.
- Grammar rules are constructed for prefix notations of patterns of each machine instruction.
- LR(0) parser for the grammar is constructed.
- The parser reads the AST in prefix notation.
- 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 often resolved by some heuristics so that the minimal number of target instructions would be generated.

LR parser - recapitulation

- Bottom-up context-free parsing, LR parsing is its deterministic version
- See example on the board
- Contd…