TDDD16 Compilers and Interpreters

TDDB44 Compiler Construction

Code Generation

Peter Fritzson, Christoph Kessler,
IDA, Linköpings universitet, 2009.

Code Generation

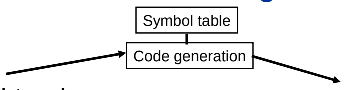
Requirements for code generation

- Correctness
- High code quality
- Efficient use of the resources of the target machine
- Quick code generation (for interactive use)
- Retargetability (parameterization in target machine spec.)

In practice:

- Difficult to generate good code
- Simple to generate bad code
- There are code generator generators ...

Intermediate Code vs. Target Code



Intermediate code

- High-level IR
  e.g. Abstract Syntax Tree (AST)
- Medium-level IR
  e.g. Control flow graph of complex operations (calls, array refs left)
- Low-level IR
  e.g. Quadruples, DAGs

Lowering the IR

Target code

- Very low-level IR
  (using target instructions only)
- Assembler code / Object code
  - Absolute machine code
  - Relocatable machine code

  (often generate asm (text) code and use an assembler tool to convert this to binary (object) code – easier, but slower compile

---

- Code for abstract stack machine
  e.g. Postfix code
- Code for concrete stack machine
  e.g. JVM byte code

Absolute vs. Relocatable Target Code

Absolute code

- Final memory area for program is statically known
- Hard-coded addresses
- Sufficient for very simple (typically, embedded) systems

☺ fast
☹ no separate compilation
☹ cannot call modules from other languages/compilers

Relocatable code

- Needs relocation table and relocating linker + loader
  or run-time relocation in MMU (memory management unit)

☺ most flexible

Stack Machines vs. Register Machines

Generate code for C assignment **A = B * C;**

where A, B *global*, C *local* var.

	B value	**B value**	**B*C** value
Stack:	**C** addr	**C** value	
	step 1	step 2	step 3

On a stack machine:

```
PUSH  _A   // static address of A
PUSH  _B   // static address of B
LOAD        // dereference _B
PUSH  fp   // stack frame ptr reg
ADD   #4  // C at stack adr. FP+4
   (step1 above)
LOAD       // load C value (step 2)
MUL  // multiply two stack values
   (step 3 above)
STORE  // store via address of A
```

On a register machine:

```
LDCONST  _A, R1
LDCONST  _B, R2
LOAD  (R2), R2  // dereference _B
ADD     FP, #4, R3  // addr. of C
LOAD (R3), R3  // dereference &C
MUL     R2, R3, R2
STORE  R2, (R1)
```

3 Main Tasks in Code Generation

- **Instruction Selection**
  - Choose set of instructions equivalent to IR code
  - Minimize (locally) execution time, # used registers, code size
  - Example: INCRM #4(fp)   vs.   LOAD #4(fp), R1
    ADD R1, #1, R1
    STORE R1, #4(fp)
- **Instruction Scheduling**
  - Reorder instructions to better utilize processor architecture
  - Minimize temporary space (#registers, #stack locations) used, execution time, or energy consumption
- **Register Allocation**
  - Keep frequently used values in registers (limited resource!)
    - Some registers are reserved, e.g. sp, fp, pc, sr, retval …
  - Minimize #loads and #stores   (which are expensive instructions!)
  - ***Register Allocation***: Which variables to keep when in some register?
  - ***Register Assignment***: In which particular register to keep each?

Machine Model (here: a simple register machine)

- **Register set**
  - E.g. 32 general-purpose registers R0, R1, R2, …
    some of them reserved (sp, fp, pc, sr, retval, par1, par2 …)
- **Instruction set** with different addressing modes
  - Cost (usually, time / latency)
    depends on the operation and the addressing mode
  - Example: PDP-11 (CISC), instruction format *OP src, dest*

Source operand	Destination address	Cost
register	register	1
register	memory	2
memory	register	2
memory	memory	3

Two Example Machine Models

- Simple CISC machine model (CISC = Complex Instruction Set Computer). src, dest can be either memory or register
  - MOVE src, dest (or LOAD src, reg; STORE reg, dest)
  - OP src, dest
- Simple RISC machine model (RISC = Reduced Instruction Set Computer)
  - LOAD reg, mem
  - STORE mem, reg
  - OP reg, reg, reg // **Operations only between registers**

Example: A = B + C;

```
1.  MOVE   _B, R0      ; 2
    ADD    _C, R0      ; 2
    MOVE   R0, _A      ; 2    →  total cost = 6

2.  MOVE   _B, _A      ; 3
    ADD    _C, _A      ; 3    →  total cost = 6

3.  (B already in R2, C already in R3,  C in R3 not used later)
    ADD    R2, R3      ; 1
    MOVE   R3, _A      ; 2    →  total cost = 3

4.  (B already in R2, C in R3 and not needed later, A will be kept in R3)
    ADD    R2, R3      ; 1    →  total cost = 1
```

There is a lot to be gained with good register allocation!

Some Code Generation Algorithms

- Macro-expansion of IR operations (quadruples)
- "Simple code generation algorithm" (textbook Section 8.6)
- Code generation for expression trees (textbook Section 8.10)
  - Labeling algorithm [Ershov 1958] [Sethi, Ullman 1970]
- Code generation using pattern matching
  - For trees: Aho, Johnsson 1976 (dynamic programming), Graham/Glanville 1978 (LR parsing), Fraser/Hanson/Proebsting 1992 (IBURG tool), …
  - For DAGs: [Ertl 1999], [K., Bednarski 2006] (DP, ILP)

Macro Expansion of Quadruples

- Each quadruple is translated to a sequence of one or several target instructions that performs the same operation.

☺ very simple, quick to implement

☹ bad code quality

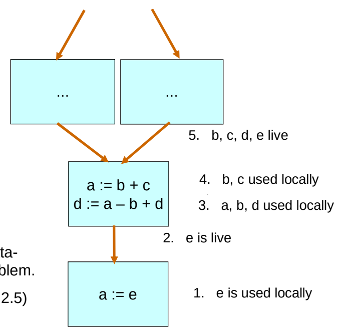
- Cannot utilize powerful instructions/addressing modes that do the job of several quadruples in one step
- Poor usage of registers

Simple Code Generation Algorithm (1)

- Input: Basic block graph (quadruples grouped in BB's)
- **Principle:**
  Keep a (computed) value in a register as long as possible, and move it to memory only
  1. if the register is needed for another calculation
  2. at the end of a basic block.
- A variable x is **used locally** after a point p if x's value is used within the block after p before an assignment to x (if any) is made.
- All variables (except temporaries) are assumed to be **live** (may be used later before possibly being reassigned) after a basic block.

BB3:
(ADD, a, b, **x**)
…
(MUL, **x**, y, t1)
…
(ASGN, t1, 0, **x**)
…

”is used locally” and ”live”



”live variables”

is a backward data-flow analysis problem.

(Textbook Sec 9.2.5)

Simple Code Generation Algorithm (2)

reg(R): current content (variable) stored in register R

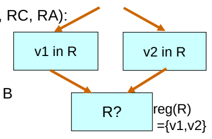
adr(A): list of addresses (”home” memory location, register) where the *current* value of variable A resides

Generate code for a quadruple Q = (op, B, C, A): (op a binary oper.)

- (RB, RC, RA) ← getreg(Q); // selects registers for B, C, A – see later
- If reg(RB) != B
  generate **LOAD** B, RB; reg(RB) ← B; adr(B) ← adr(B) U { RB }
- If reg(RC) != C
  generate **LOAD** C, RC; reg(RC) ← C; adr(C) ← adr(C) U { RC }
- generate **OP** RB, RC, RA (where **OP** implements op)
  adr(A) ← { RA }; // old value of A in memory is now obsolete
  reg(RA) ← A;
- If B and/or C no longer used locally and are not live after the current basic block, free their registers RB, RC (update reg, adr)

After all quadruples in the basic block have been processed,
generate **STORE**s for all non-temporary var's that only reside in a register

Simple Code Generation Algorithm (3)



getreg (quadruple Q = (op, B, C, A)) determines (RB, RC, RA):

- Determine RB:
  - If adr(B) contains a register RB and reg(RB) == B, then use RB.
  - If adr(B) contains a register RB with reg(RB) != B or adr(B) contains no register at all:
    - Use an empty register as RB if one is free.
    - If no register is free: Select any victim register R that does not hold C.
    - V ← { v: R in adr(V) } (there *may* be several v's, due to control flow)
      - If for all v in V, adr(v) contains some other place beyond R: OK, use R for RB.
      - If C in V, retry with another register R instead.
      - If A in V: OK, use R=RA for RB.
      - If all v in V not live after this basic block and not used locally after Q: OK, use R for RB
      - Otherwise: for each v in V,
        - generate **STORE** R, v; // **spill** register R contents to memory
        - adr(v) = adr(v) – { R } U { &v };
- Determine RC: similarly

Prefer candidates R that require least spills

Simple Code Generation Algorithm (4)

- Determine RA: similarly, where…
  - Any R with reg(R) = { A } can be reused as RA
  - If B is not used after Q, and reg(RB) = { B }, can use RA=RB.
  - (similarly for C and RC)

Example

Generate code for this basic block in pseudo-quadruple notation:

```
T1 := a + b;
T2 := c + d;
T3 := e + f;
T4 := T2 * T3;
g  := T1 – T4;
```

Initially, no register is used.

Assume a, b, c, d, e, f, g are live after the basic block, but the temporaries are not

Machine model: RISC machine model, only operations between registers, but only 3 registers R0, R1, R2

(see whiteboard)

Solution (NB – several possibilities, dep. on victim selection)

```
1.  LOAD  a, R0         // now adr(a) = { &a, R0 }, reg(R0)={a}
2.  LOAD  b, R1
3.  ADD   R0, R1, R2  // now adr(T1) = {R2},  reg(R2)={T1}
    // reuse R0, R1 for c, d, as a, b still reside in memory
    // use R0 for T2, as c still available in memory.
4.  LOAD  c, R0
5.  LOAD  d, R1
6.  ADD   R0, R1, R0   // now adr(T2) = {R0},  reg(R0)={T2}
    // reuse R1 for e, need a register for f – none free! Pick victim R0
7.  STORE  R0, 12(fp)   // spill R0 to memory - a stack location for T2, e.g. at fp+12
8.  LOAD  e, R1
9.  LOAD  f,  R0
10. ADD   R1, R0, R1   // now adr(T3) = {R1},  reg(R1)={T3}
11. LOAD  T2, R0        // reload T2 to R0
12. MUL   R0, R1, R0   // T4 in R0
13. SUB   R2, R0, R2   // g in R2
14. STORE R2, g
```

```
T1 := a + b;
T2 := c + d;
T3 := e + f;
T4 := T2 * T3;
g  := T1 – T4;
```

14 instructions,

including 9 memory accesses

(2 due to spilling)

Example – Slightly Reordered

Generate code for this basic block in pseudo-quadruple notation:

T2 := c + d;
T3 := e + f;
T4 := T2 * T3;
T1 := a + b;
g := T1 – T4;

Moving T1 := a + b; here does not modify the semantics of the code. (Why?)

Initially, no register is used.

Assume a, b, c, d, e, f, g are live after the basic block,
but the temporaries are not

Machine model: as above, but only 3 registers R0, R1, R2

(see whiteboard)

Solution for Reordered Example

T2 := c + d;
T3 := e + f;
T4 := T2 * T3;
T1 := a + b;
g := T1 – T4;

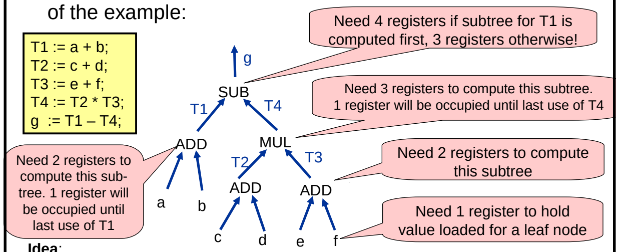
```
1.  LOAD  c, R0
2.  LOAD  d, R1
3.  ADD   R0, R1, R2   // now adr(T2)={R2}, reg(R2)={T2}
      // reuse R0 for e, R1 for f:
4.  LOAD  e, R0
5.  LOAD  f, R1
6.  ADD   R0, R1, R0   // now adr(T3) = {R0}, reg(R0)={T3}
      // reuse R0 for T4:
7.  MUL   R0, R1, R0   // now adr(T4)={R0}, reg(R0)={T4}
      // reuse R1 for a, R2 for b, R1 for T1:
8.  LOAD  a, R1
9.  LOAD  b, R2
10. ADD   R1, R2, R1   // now adr(T1) = {R1}, reg(R1)={T1}
      // reuse R1 for g:
11. SUB   R1, R0, R1   // g in R1
12. STORE R1, g
```

12 instructions,
including 7 memory accesses
No spilling! Why?

Explanation

- Consider the **data flow graph** (here, an expression **tree**) of the example:

T1 := a + b;
T2 := c + d;
T3 := e + f;
T4 := T2 * T3;
g := T1 – T4;


Need 4 registers if subtree for T1 is computed first, 3 registers otherwise!
Need 3 registers to compute this subtree. 1 register will be occupied until last use of T4
Need 2 registers to compute this subtree. 1 register will be occupied until last use of T1
Need 2 registers to compute this subtree
Need 1 register to hold value loaded for a leaf node
g
SUB
T1
T4
ADD
MUL
T2
T3
ADD
ADD
a
b
c
d
e
f

Idea:

For each subtree T(v) rooted at node v:
How many registers do we need (at least) to compute T(v) without spilling?

Call this number label(v) (a.k.a. "Ershov numbers")

If possible, at any v, code for "heavier" subtree of v (higher label) should come first.

Generating Code from Labeled Expression Trees Labeling Algorithm [Ershov 1958] (textbook Section 8.10)

- Yields **space-optimal code** (proof: [Sethi, Ullman 1970]) (using a minimum #registers **without spilling**, or min. # stack locations) for expression **trees**.
  - (Time is fixed as no spill code is generated.)
- The problem is NP-complete for expression DAGs! [Sethi'75]
  - Solutions for DAGs: [K., Rauber '95], [K. '98], [Amaral et al.'00]
- If #machine registers exceeded: Spill code could be inserted afterwards for excess registers, but not necessarily (time-) optimal then…

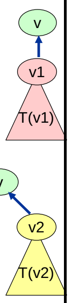
2 phases:

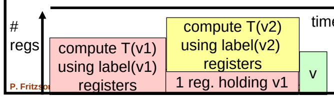
- **Labeling phase**
  - bottom-up traversal of the tree
  - computes label(v) recursively for each node v
- **Code generation phase**
  - top-down traversal of the tree
  - recursively generating code for heavier subtree first

Labeling Phase for RISC Machine Model

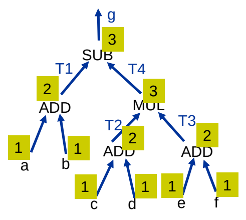
Bottom-up, calculate the register need for each subtree T(v):

- If v is a **leaf node**,
  label(v) ← 1
- If v is a **unary operation** with operand node v1,
  label(v) ← label(v1)
- If v is a **binary operation** with operand nodes v1, v2:
  m ← max(label(v1), label(v2));
  if (label(v1) = label(v2)) label(v) ← m + 1;
  else label(v) ← m;




#
regs
time
compute T(v1) using label(v1) registers
compute T(v2) using label(v2) registers
1 reg. holding v1
v

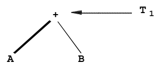
Example – Labeling Phase RISC Machine Model



Labeling Phase for CISC Machine Model

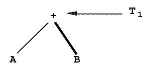
Bottom-up, calculate the register need for each subtree T(v):

- If **n** is a left leaf
  ⇒ **LABEL(n):= 1**



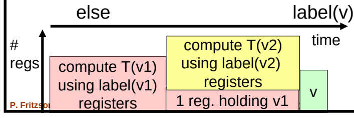
`MOVE A,R0`

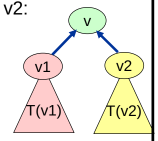
- If **n** is a right leaf
  ⇒ **LABEL(n):=** 0



`ADD B,R0`

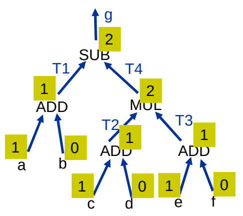
- If v is a **unary operation** with operand node v1,
  label(v) ← label(v1)
- If v is a **binary operation** with operand nodes v1, v2:
  m ← max(label(v1), label(v2));
  if (label(v1) = label(v2)) label(v) ← m + 1;
  else label(v) ← m;





Example – Labeling Phase
CISC Machine Model

The CISC Machine model needs fewer registers than RISC since it can perform operations with an operand in memory



Code Generation Phase
RISC Machine Model

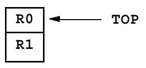
- Register stack **freeregs** of currently free registers, initially full
- Register assignment function **reg** from values to registers, initially empty

```
gencode( v ) {   // generate space-opt. code for subtree T(v)
 ▪ If v is a leaf node:
    1. R ← freeregs.pop();  reg(v) ← R;  // get a free register R for v
    2. generate( LOAD v, R );
 ▪ If v is a unary node with operand v1 in a register reg(v1)=R1:
    1. generate( OP  R1, R1 );    reg(v) ← R1;
 ▪ If v is a binary node with operands v1, v2 in reg(v1)=R1, reg(v2)=R2:
    1. if (label(v1) >= label(v2))   // code for T(v1) first:
              gencode( v1 );
              gencode( v2 );
       else                          // code for T(v2) first:
              gencode( v2 );
              gencode( v1 );
    2. generate( OP  R1, R2, R1 );
    3. freeregs.push( R2 );       // return register R2, keep R1 for v
}
```

Code Generation Phase – More Detailed for CISC Machine with memory-register ops

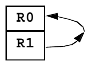
Data structures:

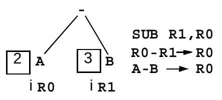
- **RSTACK:**the register stack, initialised with all available registers.
- **TSTACK**:Stack for temporary variables.



Procedures:

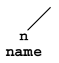
- **Gencode(n):**
  - Recursive procedure which generates code for sub-trees with root **n**.
  - The result is placed in **RSTACK[TOP]**
- **Swap(RSTACK)**
  - swaps the top two elements at the top of the stack



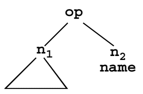


CISC Gencode(n) – 5 different cases (0 – 4) depending on the register needs for the sub-trees

- **Case 0:** n = left leaf ⇒
  Print('LOAD ', name, RSTACK[TOP]);

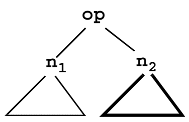


- **Case 1:** If **n** is a node with children **n1** and **n2** (left and right children, resp.) and LABEL(n2)= 0 ⇒
  Gencode(n1);
  Print(op, name, RSTACK[TOP]);



Gencode – case 2

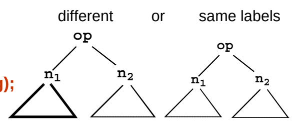
- **Case 2:**
  If **1 ≤ LABEL(n1)< LABEL(n2) and LABEL(n1)< r** where **r** is the number of registers in the machine.
  Swap(RSTACK);
  Gencode(n2);
  savereg := Pop(RSTACK);
  Gencode(n1);
  Print(op, savereg, RSTACK[TOP]);
  Push(savereg, RSTACK);
  Swap(RSTACK);



Gencode – case 3 and case 4

- **Case 3:** If **1 ≤ LABEL(n2)≤ LABEL(n1) and LABEL(n2)< r** where **r** is the number of registers in the machine.

```
Gencode(n1);
savereg := Pop(RSTACK);
Gencode(n2);
Print(op, RSTACK[TOP], savereg);
Push(savereg,RSTACK);
```



- **Case 4:** Both **n1** and **n2** have register needs ≥ **r** ⇒ store the result in the temporary stack **TSTACK**.

```
Gencode(n2); { recursive call }
T := Pop(TSTACK);
Print('STORE ', RSTACK[TOP], T);
Gencode(n1);
Print(op, T, RSTACK[TOP]);
Push(T, TSTACK);
```

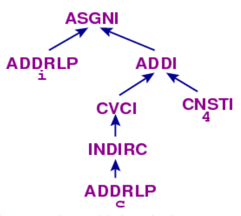
Remarks on the Labeling Algorithm

- Still one-to-one or one-to-many translation from quadruple operators to target instructions
- The code generated by gencode() is **contiguous** (a subtree's code is never interleaved with a sibling subtree's code).
  - E.g., code for a unary operation v immediately follows the code for its child v1.
  - Good for space usage, but sometimes bad for execution time on pipelined processors!
  - There are expression DAGs for which a non-contiguous code exists that uses fewer registers than any contiguous code for it. [K., Rauber 1995]
- The labeling algorithm can serve as a heuristic (but not as optimal algorithm) for DAGs if gencode() is called for common subexpressions only at the first time.

Towards Code Generation by Pattern Matching

- Example: Data flow graph (expression tree) for i = c + 4
  - in LCC-IR (DAGs of quadruples) [Fraser,Hanson'95]
  - i, c: local variables

```
{ int i; char c; i=c+4; }
```



Intermediate code in quadruple form:

(Convention: last letter of opcode gives result type: **I**=int, **C**=char, **P**=pointer)

(**ADDRLP**, i, 0, t1)	// t1 ← fp+4; addr i
(**ADDRLP**, c, 0, t2)	// t2 ← fp+12; addr c
(**INDIRC**, t2, 0, t3)	// t3 ← M(t2); c value
(**CVCI**, t3, 0, t4)	// convert char to int
(**CNSTI**, 4, 0, t5)	// create int-const 4
(**ADDI**, t4, t5, t6)	
(**ASGNI**, t6, 0, t1)	// M(t1) ← t6; store i

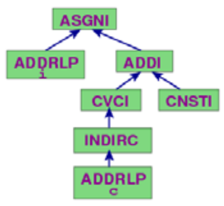
Pattern Matching Idea

- Given: Tree fragment of the intermediate code
- Given: Tree fragment describing a target machine instruction
- If the intermediate code tree fragment match the target machine instruction tree fragment, generate code with that instruction
- This method generates better code, since a single target machine instruction can match a whole tree fragment in the intermediate code

Recall: Macro Expansion

- For the example tree:
  - s1, s2, s3...: "symbolic" registers (allocated but not assigned yet)
  - Target processor has delayed load (1 delay slot)

```
{ int i; char c; i=c+4; }
```



naive instruction selection, arranged by a postorder traversal:

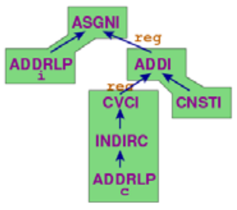
```
addi fp,#4,s1     ! ADDRLP(i)
addi fp,#8,s2     ! ADDRLP(c)
load 0(s2),s3     ! INDIRC
nop               ! ", delay slot
                  ! CVCI
addi R0,#4,s4     ! CNSTI R0 assumed 0
addi s3,s4,s5     ! ADDI
store s4,0(s1)    ! ASGNI
```

(needs 7cc) cc - compute cycles

Using Tree Pattern Matching...

- Utilizing the available addressing modes of the target processor, 3 instructions and only 2 registers are sufficient to cover the entire tree:

```
{ int i; char c; i=c+4; }
```



pattern–matching instruction selection, arranged by a postorder traversal:

```
load 8(fp),s3     ! ADDRFP+INDIRC+CVCI
nop               ! ", delay slot

addi s3,#4,s4     ! CNSTI+ADDI

store s4,4(fp)    ! ADDRLP(i)+ASGNI
```

(needs 4cc)

Code Generation by Pattern Matching

- Powerful target instructions / addressing modes may cover the effect of several quadruples in one step.
- For each instruction and addressing mode,
  define a pattern that describes its behavior in terms of quadruples resp. data-flow graph nodes and edges
  (usually limited to tree fragment shapes: **tree pattern**).
- A pattern **matches** at a node v
  if pattern nodes, pattern operators and pattern edges coincide with a tree fragment rooted at v
- Each instruction (tree pattern) is associated with a **cost**,
  e.g. its time behavior or space requirements
- **Optimization problem**: Cover the entire data flow graph (expression tree) with matching tree patterns such that each node is covered *exactly once*, and the accumulated cost of all covering patterns is minimal.

Tree Grammar (Machine Grammar)
(E.g. to be used for code gen pattern maching by parsing)

The target processor is described by a tree grammar $G = (N, T, s, P)$

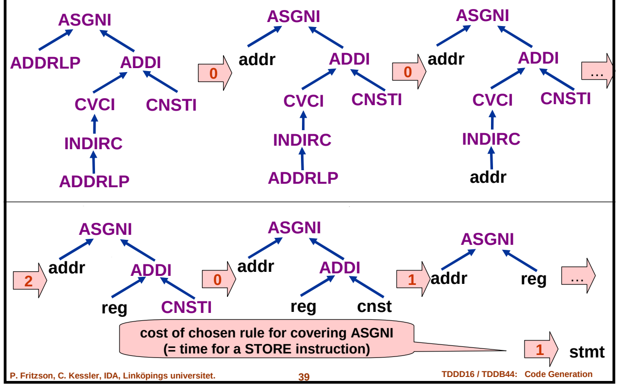
nonterminals N = { stmt, reg, con, addr, mem, ... } (start symbol is stmt)

terminals T = { CNSTI, ADDRLP, ... }

production rules P:

	target instruction for pattern	cost
reg → ADDI(reg, con)	`addi %r,%c,%r`	1
reg → ADDI(reg, reg)	`addi %r,%r,%r`	1
stmt → ASGNI(addr, reg)	`store %r,%a`	1
stmt → ASGNI(reg, reg)	`store %r,0(%r)`	1
reg → ADDRLP	`addi fp,#%d,%r`	1
addr → ADDRLP	`%d(fp)`	0
reg → addr	`addi %a,%r`	1
reg → INDIRC(addr)	`load %a,%r; nop`	2
reg → INDIRC(reg)	`load 0(%r),%r; nop`	2
reg → CVCI(INDIRC(addr))	`load %a,%r; nop`	2
reg → CVCI(INDIRC(reg))	`load 0(%r),%r; nop`	2
con → CNSTI	`%d`	0
reg → con	`addi R0,#%c,%r`	1

Derivation Using an LR Parser:



Some Methods for Tree Pattern Matching

- Use a LR-parser for matching [Graham, Glanville 1978]
  - ☺ compact specification of the target machine
    using a context-free grammar (”machine grammar”)
  - ☺ quick matching
  - ☹ not total-cost aware
    (greedy local choices at reduce decisions → suboptimal)
- Combine tree pattern matching with dynamic programming for total cost minimization (→ More details in TDDC86 course)
  [Aho, Ganapathi, Tjiang ’89] [Fraser, Hanson, Proebsting’92]
- An LR parser is stronger than what is really necessary for matching tree patterns in a tree.
  - Right machine model is a **tree automaton**
    = a finite automaton operating on input *trees*
    rather than flat strings [Ferdinand, Seidl, Wilhelm ’92]
- By Integer Linear Programming [Wilson et al.’94] [K., Bednarski ’06]