Compilation Methods SS 2013 - Assignment 5

Kastens, Pfahler, 13. Juni 2013

Exercise 1 (Storage Mapping for Arrays)

The following declaration appears in a Pascal program:

```
var rain: array [1..12, 2000..2029] of integer;
```

a) Draw the memory layout in row-major order.
b) Let's assume a target architecture that uses 32 bit integers (4 bytes) and requires integers to be aligned on 4 byte boundaries.

   What is the index map (see Slide 305) for the declared array? Use a Horner scheme with all constant parts of the array descriptor determined at compile time.
c) Give intermediate code in the style of Slide 316 for the following statement:

```
rain[month, 2011] := 13;
```

   Compiled code for our target architecture uses register `R6` to store the base address of the current activation record on the run-time stack. Assume that variable `month` is stored at offset 8 and that the array starts at offset 12. The assignment is represented as a node of type `assign`. Its left subtree must compute the address of the target location; its right subtree must represent the value to be assigned. Assume compile time evaluation of all operations on constant operands.

Exercise 2 (Code Sequences for Control Statements)

Design code sequences for `for`-loops and `do-while`-loops in C. Example sequences for different kinds of loops appear on Slide 315.

Exercise 3 (Code Selection by Pattern Matching in Trees)

a) The following set of translation patterns describes a simple RISC processor:

No.	Operator	Operands	Result	Code	Costs
1	iconst	const	-> IConst	./.	0
2	iconst	const	-> IReg	move #const, IReg	1
3	var	address	-> IReg	addr #adress, IReg	2
4	iadd	$IReg_1$, $IReg_2$	-> $IReg_3$	add $IReg_1$, $IReg_2$, $IReg_3$	2
5	iadd	$IReg_1$, IConst	-> $IReg_2$	add $IReg_1$, #const, $IReg_2$	2
6	deref	$IReg_1$	-> $IReg_2$	load $IReg_1$, $IReg_2$	3
7	assign	$IReg_1$, $IReg_2$	-> Stmt	store $IReg_1$, $IReg_2$	2

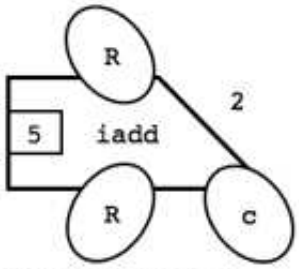
   Assume that the following source code

```
{ int v[8];
  struct { int x, y; } p;
  p.y = v[7] + 9;
}
```

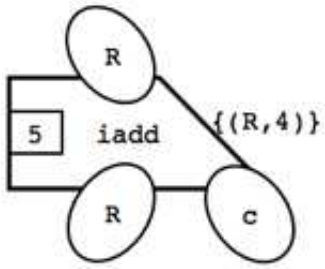
   has been translated to the intermediate code tree given below. Use the set of patterns to cover that tree. Apply two different strategies:

   1. Choose locally optimal solutions to cover the tree in a single bottom-up traversal through the tree.
   2. Apply the 2-pass-strategy that makes its decisions based on costs of whole subtrees (Slide 320).

   Fill out the prepared tree form according to the following examples:

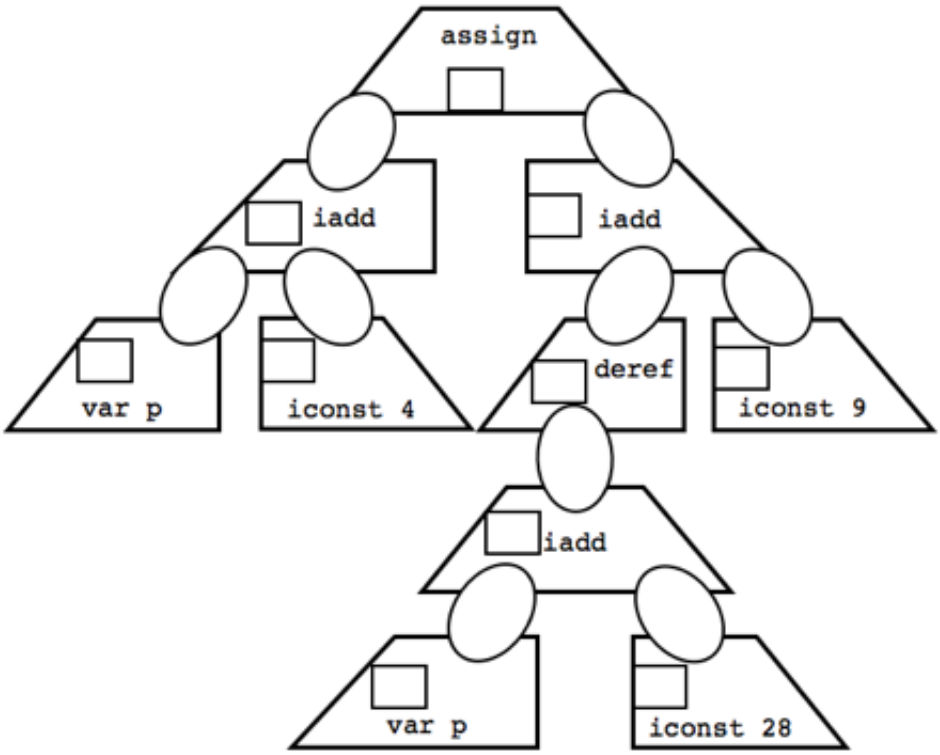


Example for annotation:
pattern 5, cost 2 in 1-pass strategy,

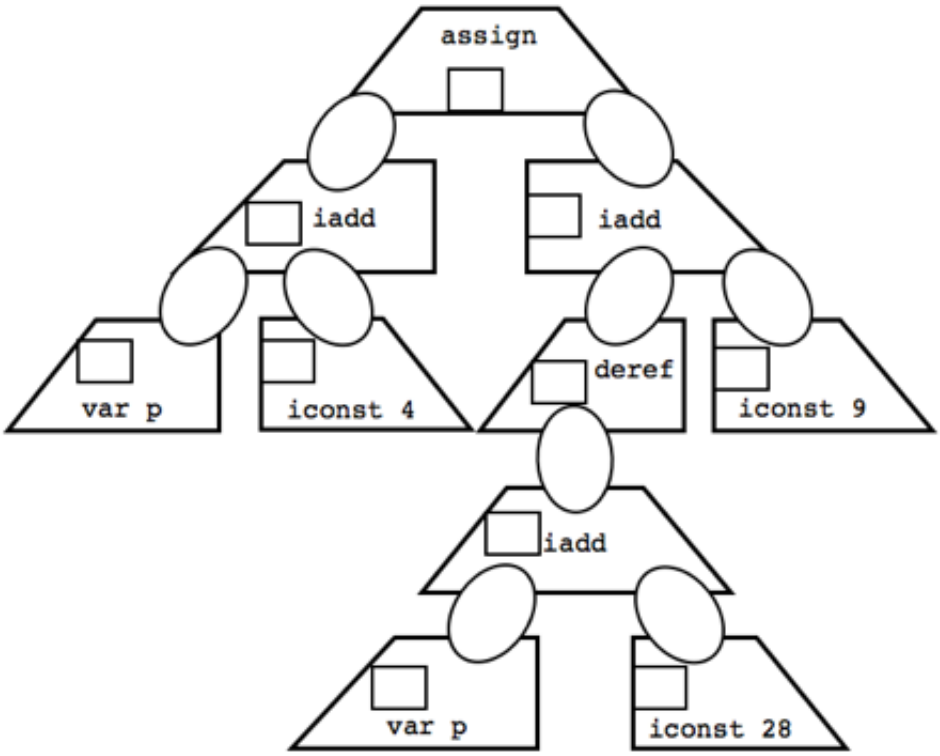


or annotation with pairs set in 2-pass strategy

Fill in the annotations for the 1-pass strategy:



Fill in the annotations for the 2-pass strategy:



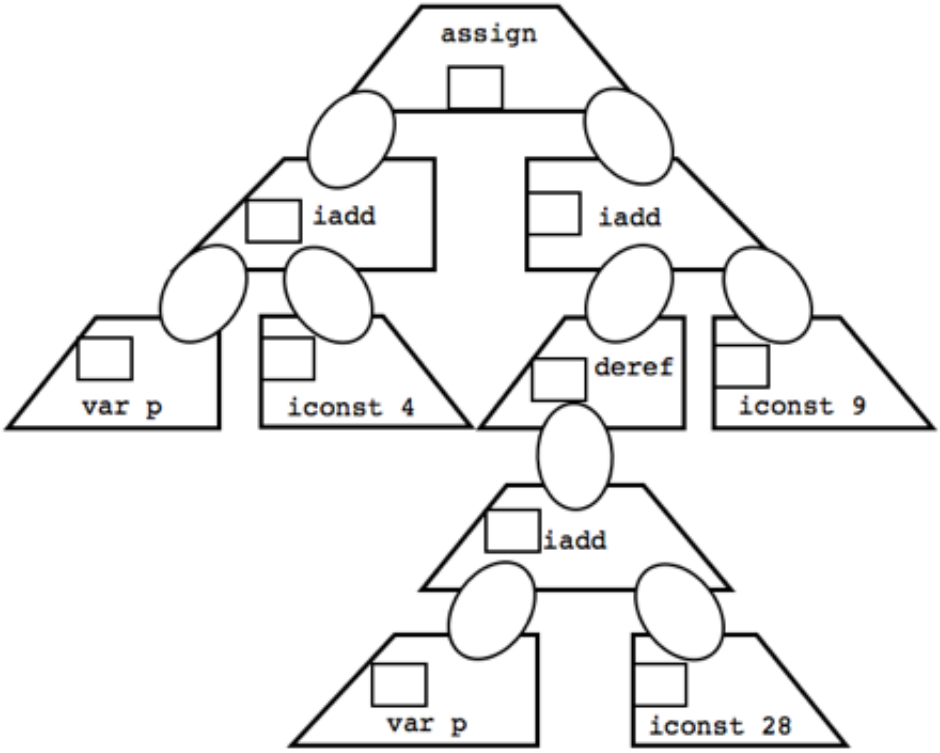
Is there a difference in the tree covers?

b)

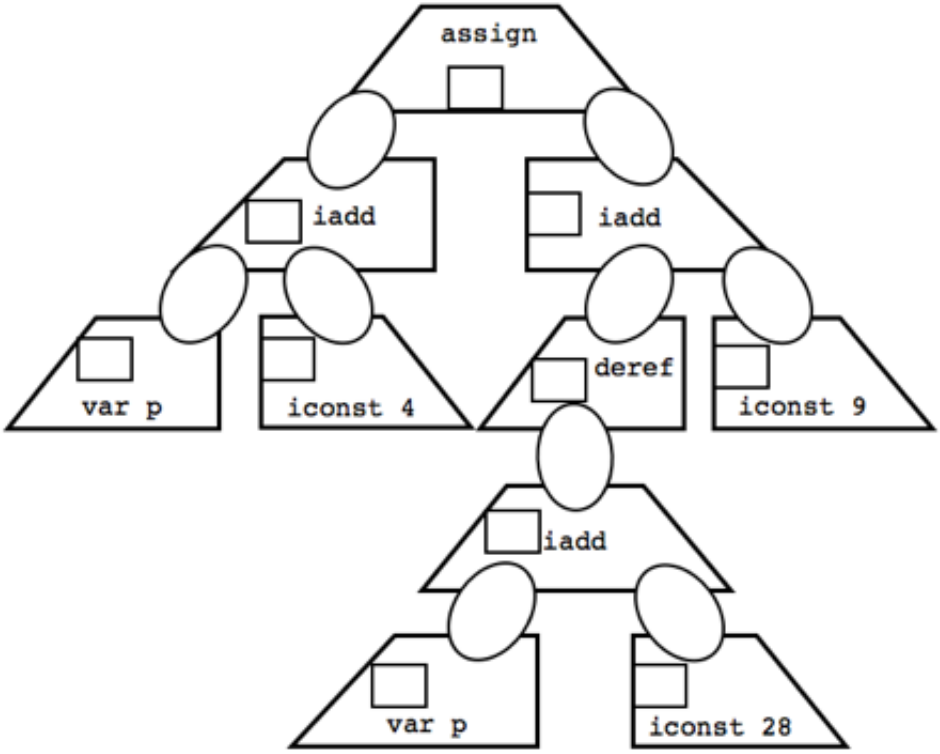
Consider 3 additional tree patterns:

No.	Operator	Operands	Result	Code	Costs
8	iadd	$IReg_1$, $IReg_2$	-> RegSum	./.	0
9	deref	RegSum	-> $IReg_3$	load $IReg_1$ + $IReg_2$, $IReg_3$	3
10	assign	RegSum, $IReg_3$	-> Stmt	store $IReg_1$ + $IReg_2$, $IReg_3$	2

The new instructions compute a to be accessed memory address as the sum of the contents of two registers. Apply both tree cover strategies again. Fill in the annotations for the 1-pass strategy:



Fill in the annotations for the 2-pass strategy:



Is there a difference in the tree covers?

c)

LAB/HOME:

The directory `blatt5/risc` contains the above set of translation patterns in a form that is suitable for the bottom-up rewriting generator `BURG`. The file `risc.burg` contains a specification of the tree patterns and a test program that computes a cost optimal cover of a given intermediate code tree.

To compute a cost optimal cover of the example tree, use `BURG` to build a tree automaton:

```
./burg -I risc.burg > risc.c
```

Afterwards compile and link the resulting C source code. (The file `util.c` supports tree construction, it is included and linked.)

```
cc -o risc risc.c
```

Start the resulting program `risc` and compare its output with your solution of part (a).

Note: The supplied `burg` executable works on Linux computers only.

d)

LAB/HOME:

Extend the pattern specification in file `risc.burg` with the additional "RegSum" tree patterns described above. These patterns represent the machine instructions

`load IReg`$_1$ `+IReg`$_2$ `,IReg`$_3$

and

`store IReg`$_1$ `+IReg`$_2$ `,IReg`$_3$

respectively.

Regenerate the tree automaton from the modified specification. What cost optimal tree cover does the new automaton find?