IN4303 — Compiler Construction

Exam

April 04, 2011

---

1. Answer every question on its own page (A sheet has four pages).
2. Put your name, your student ID and the number of the question on top of each page.
3. If you need more than one sheet to answer a question, number your pages and state the overall number of pages for this question on the first page.
4. Take care of your time. This exam has 13 questions, for a total of 150 points. Try to answer a question worth 10 points in 10 minutes.
5. Keep your answers short and precise. Don't waste your time on essay writing.
6. Hand in the answers together with this form (including the questions).

Good luck!

Name: _____ Student ID: _____

Question	Points	Score
Language	15	
Formal grammars	10	
Syntax trees	10	
Term rewriting	10	
Static analysis	10	
Java Virtual Machine	10	
Polymorphism	10	
Calling conventions	10	
Liveness analysis	20	
Register allocation	20	
Garbage collection	5	
Lexical analysis	10	
LL parsing	10	
Total	150	

Question 1: Language (15 points)

(a) According to Edward Sapir, what is language? (5)

Solution: Language is a purely *human* and *non-instinctive* method of *communicating* ideas, emotions, and desires by means of a *system* of *voluntarily* produced *symbols*.

(b) What is the formal language $L(G)$ specified by a formal grammar G? Give a definition in English. (5)

Solution: The set of all *words* over the *alphabet* of G which are derivable from the *start symbol* of G by *repeatedly applying* production rules of G. A production rule is applied by *replacing* its left-hand side with its right-hand side.

(c) Where are aspects from Sapir's definition of language reflected in the definition of $L(G)$? Which aspects are not reflected at all? (5)

Solution: Symbols are reflected in the alphabet, system is reflected in the grammar, and production of symbols is reflected in derivation. Its not reflected that language is human and non-instinctive, that its used for communication, and that production is voluntary.

Question 2: Formal grammars (10 points)

Let G_1 be a formal grammar with nonterminal symbols S, and P, terminal symbols '**f**', '**x**', ',', '(' and ')', start symbol S, and the following production rules:

$$\begin{aligned} S &\rightarrow \mathbf{f}\ (\ P\ ) \\ P &\rightarrow \mathbf{x} \\ P &\rightarrow P\ ,\ \mathbf{x} \\ P &\rightarrow \mathbf{x}\ ,\ P \end{aligned}$$

(a) Is G_1 context-free? Why (not)? (1)

Solution: Yes, because all production rules are of the form $N \times (N \cup \Sigma)^*$.

(b) Describe the language defined by G_1 in English. (2)

Solution: The language consists of all applications of a function symbol **f** to one or more parameters **x**. Parameters are separated by commas and surrounded by parentheses.

(c) Give a left-most derivation for the sentence $\mathbf{f}(\mathbf{x},\mathbf{x},\mathbf{x})$ according to G_1. (3)

Solution: $S \Rightarrow \mathbf{f}\ (\ P\ ) \Rightarrow \mathbf{f}\ (\ P\ ,\ \mathbf{x}\ ) \Rightarrow \mathbf{f}\ (\ P\ ,\ \mathbf{x}\ ,\ \mathbf{x}\ ) \Rightarrow \mathbf{f}\ (\ \mathbf{x}\ ,\ \mathbf{x}\ ,\ \mathbf{x}\ )$

(d) Use $\mathbf{f}(\mathbf{x},\mathbf{x})$ as an example to explain why G_1 is ambigous. (4)

Solution: There are two different left-most derivations for the same word.
$S \Rightarrow \mathbf{f}\ (\ P\ ) \Rightarrow \mathbf{f}\ (\ P\ ,\ \mathbf{x}\ ) \Rightarrow \mathbf{f}\ (\ \mathbf{x}\ ,\ \mathbf{x}\ )$ and $S \Rightarrow \mathbf{f}\ (\ P\ ) \Rightarrow \mathbf{f}\ (\ \mathbf{x}\ ,\ P\ ) \Rightarrow \mathbf{f}\ (\ \mathbf{x}\ ,\ \mathbf{x}\ )$

Question 3: Syntax trees (10 points)

(a) Why do we need syntax trees when constructing compilers? (2)

Solution: Semantic analysis and code generation require knowledge about the structure of a sentence. Syntax trees capture this structure.

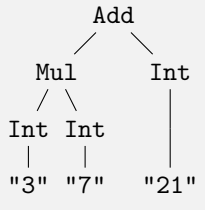
(b) What are the fundamental differences between parse trees and abstract syntax trees? (3)

Solution: In parse trees, inner nodes are labeled with nonterminal symbols. Leave nodes are labeled with terminal symbols, which form the derived sentence. Abstract syntax trees abstract over injective production rules and over those terminal symbols which do not convey information. Instead, they convey information in constructor labels at nodes.

(c) How can we represent trees as terms? Illustrate your explanation with an example. (5)

Solution: The leaves of trees are represented as constants. Nodes with subtrees are represented as terms with subterms. Labels of these nodes become the constructor of these terms.

The leaves of the example tree become constants `"3"`, `"7"` and and `"21"` . Their parent nodes become terms with a constructor `Int` and a constant as a subterm. The node labeled with `Mul` combines the terms for its subtrees to `Mul(Int("3"),Int("7"))`. Finally, the root node combines the terms for its subtrees to `Add(Mul(Int("3"),Int("7")),Int("21"))`.



Question 4: Term rewriting (10 points)

Stratego provides a strategy `inverse` with the following implementation:

```
inverse(|a): []     -> a
inverse(|a): [x|xs] -> <inverse(|[x|a])> xs
```

(a) Explain the semantics of `inverse` in English. (2)

Solution: `inverse` rewrites a list to a new list. The resulting list has the same elements as the original list, but in reversed order.

(b) What is the result of applying `inverse(|[])` to the term `[1,2,3]`? Show each step of computation. (4)

Solution:
```
<inverse(|[])> [1,2,3] => <inverse(|[1])> [2,3] =>
<inverse(|[2,1])> [3] => <inverse(|[3,2,1])> [] => [3,2,1]
```

(c) Based on the definition of `inverse`, explain how an accumulator is used. (4)

Solution: An accumulator stores a temporary result of an ongoing computation. `inverse` uses its term parameter to accumulate the inverted list. When the strategy is called, the accumulator should be initialised with the empty list. The second rule accumulates the inverted list by prepending the current head of the list to the so far accumulated list. It calls `inverse` recursively on the tail of the list with the new accumulator. The first rule returns the accumulated list as the result of the inversion.

Question 5: Static analysis (10 points)

(a) How does static analysis contribute to a compiler w.r.t. its architecture? (2)

Solution: Static analysis takes place after parsing and before code generation. It works on the abstract syntax tree provided by the parser and provides additional information about name bindings and types to the code generator. It furthermore checks constraints and reports corresponding errors to the user.

(b) Explain the generic approach of performing static analysis in rename/map/project/check phases. Use the example of type checking MiniJava. (8)

Question 6: Java Virtual Machine (10 points)

Execute the bytecode instructions of `A/main()V`, starting with an empty stack:

`A/main()V`

```
aload_0
bipush 5
iconst_4
isub
invokevirtual A/m(I)V
```

`A/m(I)V`

```
    goto l2
l1: iinc 1 -1
l2: iload_1
    ifne l1
    return
```

Hint: `iinc 1 -1`

```
iload_1
ldc -1
iadd
istore_1
```

The initial value of local variable 0 is `4242 4103`, pointing to an object of class `A`. Show stacks and local variables after each instruction.

Solution:

`aload_0`

stack	locals
&A	&A

`bipush 5`

stack	locals
&A	&A
5	

`iconst_4`

stack	locals
&A	&A
5	
4	

`isub`

stack	locals
&A	&A
1	

`invokevirtual A/m(I)V`

stack	locals
	&A

stack	locals
	&A
	1

`goto l2`
`iload_1`

stack	locals
1	&A
	1

`ifne l1`

stack	locals
	&A
	1

`iinc 1 -1`

stack	locals
	&A
	0

`iload_1`

stack	locals
0	&A
	0

`ifne l1`

stack	locals
	&A
	0

`return`

stack	locals
	&A

Question 7: Polymorphism (10 points)

(a) Identify three examples of polymorphism in the following Java expression: (6)

```
"1" + ((2 + 4) + 3.5)
```

Which kinds of polymorphism do they represent? Explain the differences.

Solution: The `+` operator is overloaded (ad-hoc polymorphism): It either performs a string concatenation (1st occurence), an integer addition (2nd occurence), or a floating point addition (3rd occurrence). These are two different kinds of overloading. The first and second occurence perform different operations on different types, while the second and third occurrence perform similar operations on different types.

The result of `(2 + 4)` is converted (implicit coercion, ad-hoc polymorphism) from an integer value into a floating point value. Similarly, the result of `((2 + 4) + 3.5)` is converted into a string value.

With operator overloading, the types of the operands determine which operation is performed. With type coercion, there is no operation that can handle the operand types, i.e. there is no operation that can handle an integer and a floating point operand or a string and a floating point operand. Instead, the operand is converted to a type an operation can handle.

(b) Explain the difference between method overloading and method overriding. Illustrate your explanation with an example in Java. (4)

Solution: Method overloading and overriding is about different methods with the same name. Method overloading can take place in unrelated classes, in the same class, and in classes related by inheritance. In unrelated classes, there are no constraints on the parameter or return types of the overloaded methods (e.g. `A.m()` and `B.m()` or `A.m()` and `B.m(A)`). In the same class, the parameter types of methods with the same name need to be different (e.g. `B.m()` and `B.m(A)`). The same holds for classes related by inheritance (e.g. `B.m()` and `C.m(A)`).

In contrast, method overriding can only take place in classes related by inheritance. The method in the subclass needs to have the same and the same parameter types (at least in Java) and a covariant return type (subtype of the overridden method's return type). In the example, `C.m(A)` overrides `B.m(A)` (same parameter types, covariant return type) but overloads `B.m()` (different parameter types).

```
class A {
  public int m() {...}
}
```

```
class B {
  public int m() {...}
  public B m(A a) {...}
}
```

```
class C extends B {
  public C m(A a) {...}
}
```

Question 8: Calling conventions (10 points)

A compiler translates a function call and a function body to the following instructions for a register-based machine:

function call

```
mov  AX 21
mov  DX 42
call _f@8
```

function body

```
push BP
mov  BP SP
add  AX DX
pop  BP
ret
```

(a) Which calling convention do these instructions follow? (1)

Solution: FASTCALL

(b) What are the benefits of this calling convention? (1)

Solution: Passing parameters in registers avoids memory access and reduces stack frame sizes.

(c) How are calls handled by callers and by callees according to this convention? Base your explanation on the given instructions. (8)

Solution: The caller passes the first parameters in registers (`mov AX 21`, `mov DX 42`) and pushs remaining parameters right-to-left on the stack (not in this example). The callee saves the old base pointer on the stack (`push BP`) and initialises the new one (`mov BP SP`). Next, it saves registers which it needs to preserve (not in this example). Finally, it leaves the result in `AX` (`add AX DX`), restores the registers (not in this example) and the base pointer (`pop BP`), and cleans the stack (not in this example), before it returns (`ret`).

Question 9: Liveness analysis (20 points)

Consider the following intermediate code:

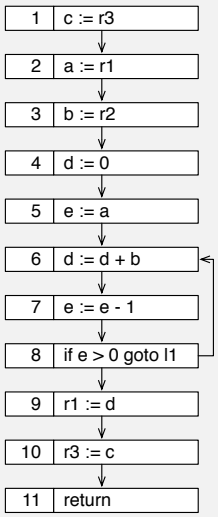
```
     c := r3
     a := r1
     b := r2
     d := 0
     e := a
l1:  d := d + b
     e := e - 1
     if e > 0 goto l1
     r1 := d
     r3 := c
     return
```

(a) Construct the control graph. (2)

(b) Calculate successor nodes, defined variables, and used variables for each node in the control graph. (3)

(c) Assume `r1` and `r3` to be live-out on the return instruction. Calculate live-ins and live-outs for each node in the control graph. Present your results in a table. (15)

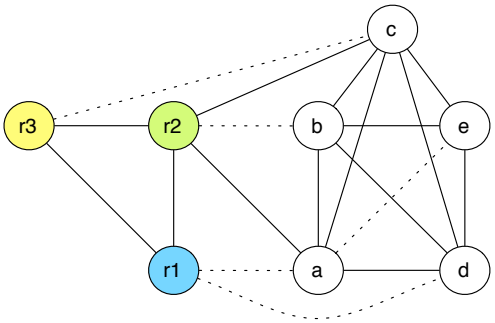
Solution:



node	succ	def	use	out	in	out	in
11				r1 r3	r1 r3	r1 r3	r1 r3
10	11	r3	c	r1 r3	r1 c	r1 r3	r1 c
9	10	r1	d	r1 c	c d	r1 c	c d
8	6, 9		e	c d	c d e	b c d e	b c d e
7	8	e	e	c d e	c d e	b c d e	b c d e
6	7	d	b d	c d e	b c d e	b c d e	b c d e
5	6	e	a	b c d e	a b c d	b c d e	a b c d
4	5	d		a b c d	a b c	a b c d	a b c
3	4	b	r2	a b c	r2 a c	a b c	r2 a c
2	3	a	r1	r2 a c	r1 r2 c	r2 a c	r1 r2 c
1	2	c	r3	r1 r2 c	r1 r2 r3	r1 r2 c	r1 r2 r3

Question 10: Register allocation (20 points)

You have to colour the following interference graph with three colours (`r1`, `r2`, `r3` are precoulored):

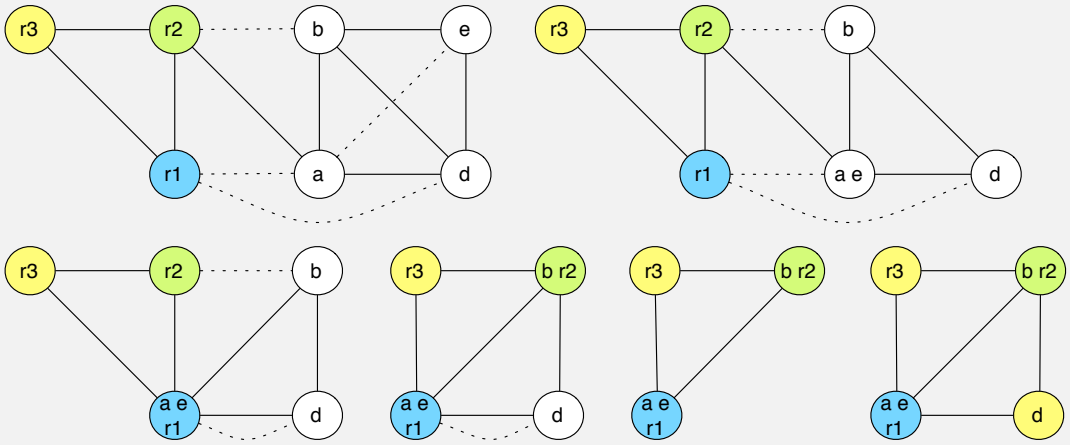


(a) Should the next step be a *spill*? Why (not)? (5)

Solution: We cannot simplify, since all nodes are either precoloured or move-related. But we can coalesce `a` and `e` (all neighbours of `e` of significant degree are neighbors of `a`, George).

(b) Spill node `c` and continue the graph colouring until you can decide if this spill is an actual one. (12)

Solution: We can coalesce `a` and `e`, since the resulting node has no neighbours of significant degree ($0 < 3$, Briggs). For the same reason, we can coalesce `a e` and `r1` as well as `b` and `r3`. We cannot coalesce `a e r1` with `d` since they interfere, but we can simplify `d`. Now we have only precoloured nodes and can bring back nodes, starting with `d`.



(c) Is node `c` an actual spill? (1)

Solution: Yes, since we cannot assign any colour to `c`.

(d) Perform the spill on the intermediate code from the previous question. (2)

Solution:

```
     c1 := r3
     M[cloc] := c1
     a := r1
     b := r2
     d := 0
     e := a
l1:  d := d + b
     e := e - 1
     if e > 0 goto l1
     r1 := d
     c2 := M[cloc]
     r3 := c2
     return
```

Question 11: Garbage collection (5 points)

(a) Explain the general ideas behind garbage collection by reference counting. (3)

Solution: For each element on the heap, a counter is maintained. At each assignment, the counter of the olde reference is decreased and the counter of the new reference increased. Heap elements with a reference count of 0 can be collected when memory for a new element needs to be allocated.

(b) What are the disadvantages of this strategy? (2)

Solution: Heap elements with cyclic references are not collected. Furthermore, the additional instructions for counting slow down the execution.

Question 12: Lexical analysis (10 points)

Let G_2 be a formal grammar with nonterminal symbols S and D, terminal symbols '**b**', '**0**' and '**1**', start symbol S, and the following production rules:

$$\begin{aligned} S &\rightarrow \mathbf{b}\ D \\ D &\rightarrow \mathbf{0}\ D \\ D &\rightarrow \mathbf{1}\ D \\ D &\rightarrow \mathbf{0} \\ D &\rightarrow \mathbf{1} \end{aligned}$$

(a) Is G_2 regular? Why (not)? (1)

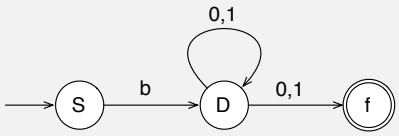
Solution: Yes. All rules have only a single nonterminal symbol on their left-hand side and either a single terminal symbol or a terminal symbol followed by a nonterminal symbol on their right-hand side.

(b) Describe the language defined by G_2 in English. (2)

Solution: The language consists of all words starting with **b**, followed by one or more digits **0** or **1**, i.e. binary numbers marked with a leading **b**.

(c) Turn G_2 *systematically* into a finite automaton. (4)

Solution:



(d) Use G_2 to generate a word with at least five letters. Show each derivation step. Use the automaton to recognise this word. Enumerate the states passed during the recognition. (3)

Solution:

$$S \Rightarrow \mathbf{b}\ D \Rightarrow \mathbf{b\ 1}\ D \Rightarrow \mathbf{b\ 1\ 0}\ D \Rightarrow \mathbf{b\ 1\ 0\ 1}\ D \Rightarrow \mathbf{b\ 1\ 0\ 1\ 0}$$

$$S \xrightarrow{\mathbf{b}} D \xrightarrow{\mathbf{1}} D \xrightarrow{\mathbf{0}} D \xrightarrow{\mathbf{1}} D \xrightarrow{\mathbf{0}} f$$

Question 13: LL parsing (10 points)

Let G_3 be a formal grammar with nonterminal symbols S, T, E and E', terminal symbols '**x**', '+' and '$', start symbol S, and the following production rules:

$$\begin{aligned} S &\rightarrow E\ \$ \\ E &\rightarrow T\ E' \\ E' &\rightarrow +\ T\ E' \\ E' &\rightarrow \\ T &\rightarrow \mathbf{x} \end{aligned}$$

(a) Construct an LL(0) parse table for the grammar. Calculate FIRST and FOLLOW sets as needed. (8)

Solution:

	nullable	FIRST	FOLLOW
S	no	$\mathbf{x}$	
E	no	$\mathbf{x}$	\$
E'	yes	$+$	\$
T	no	$\mathbf{x}$	$+$ \$

	$\mathbf{x}$	$+$	\$
S	$S \to E$ \$		
E	$E \to T\ E'$		
E'		$E' \to +\ T\ E'$	$E' \to$
T	$T \to \mathbf{x}$		

(b) Use the parse table to recognise the sentence $\mathbf{x} + \mathbf{x}$. Show the stack and the remaining input after each step. (2)

Solution:

stack	input
S	$\mathbf{x} + \mathbf{x}$\$
E\$	$\mathbf{x} + \mathbf{x}$\$
TE'\$	$\mathbf{x} + \mathbf{x}$\$
$\mathbf{x}E'$\$	$\mathbf{x} + \mathbf{x}$\$
E'\$	$+\mathbf{x}$\$
$+TE'$\$	$+\mathbf{x}$\$
TE'\$	$\mathbf{x}$\$
$\mathbf{x}E'$\$	$\mathbf{x}$\$
E'\$	\$
\$	\$