Mälardalen University
School of innovation, design and engineering

TEN1 DVA339 2020

Compiler theory, take-home examination

Thursday, Tuesday June 9, 2020, 14:10 - 18:30
Contact: Daniel Hedin, 021-107052 (15:00 - 17:00)

OBSERVE! You are *not* allowed to cooperate with, receive help from or give help to other people. Apart from this restriction all other aids are allowed.

Handing in the exam is done via Canvas as *one* pdf, word or txt file before the end of the exam. Exams handed in late will not be graded.

The exam consists of 48 points distributed over 6 questions. Answers must be given in English or Swedish and should be clearly justified.

Grades: 3: 50%, 4: 70%, 5: 90% (corresponding to ECTS E, C and A).

- Explain all solutions. A correctly explained solution with minor mistakes may render full points.
- Start each question on a new page and only write on one side of the page.
- Please make sure your exam is sorted with the answers in the natural order.
- Write down any assumptions you make.

Regular expressions and lexing (6p)

1) Why do lexers typically employ the two strategies: **(2+2+2p)**

- longest match, and
- order of definition

and what is the rationale for each of them, i.e., explain and exemplify what could go wrong if they were not used. (Note: there are three parts in this question: why the use of strategies in general, and the two strategies).

Grammars and parsing (16p)

2) Show how to put the grammar in Table 1 on LL(1) form using llanalyze. **(4+2p)**

$$\begin{array}{lcl} E & ::= & E \text{ ”+” } E \mid \text{”(” } E \text{ ”)”} \mid \text{”[” } \lambda \mid Es \text{ ”]”} \mid NUM \\ Es & ::= & E \mid E \text{ ”,” } Es \end{array}$$

Table 1: Simple grammar

Where NUM represents numbers, and terminals are marked with ””. You should explain every step you take and why. For full points you should hand in a grammar file that passes llanalyze.

3) Extend the grammar in Table 1 to be able to interact with the lists, i.e., to be able to write programs of the form below (one example per line) and suggest an abstract syntax for the extended grammar in C# as a collection of classes. (Note: there are two parts in this question: the extension and the abstract syntax in C#.) **(4+6p)**

```
Head [1,2]
Head (Tail [1,2,3])
Tail (Head [[1,2], [3,4]])
1 : 2 : []
1 : [2,3]
Head (1 : [2,3])
Tail (1 : [2,3])
```

A quick note on the extension: `Head` returns the first element of a list, `Tail` returns the list without the first element, and : (Cons) is an operator that takes an element, and a list and adds the element as the first element of the list. None of the operations modify their arguments, i.e., : does not add the element to the list, but rather creates a new list. See questions 4) and 5) for examples.

Type checking (10p)

4) Given the type language τ ::= num | [τ] that defines the types of the language found in Table 1 (and the extension) implement a type system using the abstract syntax you made in question 3). You can either use pseudocode or C#. Use your intuition for a language with lists and the examples given below. **(10p)**

```
[1,2] has type [num]
[[1,2], [3,4]] has type [[num]]
Head [1,2] has type num
Head (Tail [1,2,3]) has type num
Tail (Head [[1,2], [3,4]]) has type [num]
1 : 2 : [] has type [num]
1 : [2,3] has type [num]
Head (1 : [2,3]) has type num
Tail (1 : [2,3]) has type [num]

[1, [1]] is type incorrect
Head 1 is type incorrect
Tail 1 is type incorrect
```

Code generation and Evaluation. (16p)

5) Implement an evaluator using the abstract syntax you made in question 3). You can use either pseudocode or C#. Use your intuition for a language with lists and the examples given below. You can assume that the programs to be evaluated are type correct. Think about how to implement this efficiently. How should your represent the lists in order to make the operations fast? **(10p)**

```
Head [1,2] evaluates to 1
Head (Tail [1,2,3]) evaluates to 2
Tail (Head [[1,2], [3,4]]) evaluates to [2]
1 : 2 : [] evaluates to [1,2]
1 : [2,3] evaluates to [1,2,3]
Head (1 : [2,3]) evaluates to 1
Tail (1 : [2,3]) evaluates to [2,3]
```

6) Reversecompile the following Trac42 program and show the correspondance between the resulting program and the original Trac42 code. For full points it is important that you clearly mark which parts of the resulting program come from which parts of the Trac 42 code. **(6p)**

```
0   BSR 26
1   END
2   [f]
3     LINK
4       RVALINT 3(FP)
5       PUSHINT 0
6     EQBOOL
7     BRF 13
8       LVAL 4(FP)
9       PUSHINT 1
10      ASSINT
11      UNLINK
12      RTS
13    LVAL 4(FP)
14      RVALINT 2(FP)
15      DECL 1
16        RVALINT 3(FP)
17        PUSHINT 1
18      SUB
19      RVALINT 2(FP)
20      BSR 2
21      POP 2
22    MULT
23    ASSINT
24    UNLINK
25    RTS
26  [main]
27    LINK
28    DECL 1
29      LVAL -1(FP)
30      DECL 1
31      PUSHINT 3
32      PUSHINT 10
33      BSR 2
34      POP 2
35    ASSINT
```