

# Tentamen i

## KOMPILATORTEKNIK

Kurskod:	D7011E
Datum:	2010- 03 - 26
Skrivtid:	5 timmar
Totalt antal uppgifter:	8
Totalt antal poäng:	40
Jourhavande lärare:	Johan Nordlander
Telefon:	070 - 620 74 61
Tillåtna hjälpmedel:	Engelskt lexikon

---

1. (5 marks)

- a) Explain what the terms *source language*, *intermediate language* and *target language* mean in the context of a compiler.
- b) In what way do the languages above relate to the language used for *implementing* a compiler?
- c) What is the difference between *compiling* a program and *interpreting* it?

2. (5 marks)

- a) What makes *regular expressions* unsuitable for defining the full syntax of most programming languages?
- b) How come regular expressions still have a role to play in many programming language definitions?
- c) Deterministic finite automata (DFAs) can be used to implement analyzers for languages defined by regular expressions. What is the main difference between a DFA and the more complex parser implementations that are used for analyzing the syntax of general programming languages?

3. (5 marks)

Consider the following grammar for a simple functional language  $L$  (where ID and INTEGER, as well as strings in quotes, are considered to be terminals):

```
Program ::= Fundef Program
         | 'print' Exp
Fundef  ::= ID '(' Params ')' '=' Exp ';' // function definition
Params ::= ID
         | Params ',' Params *
Exp     ::= ID
         | INTEGER
         | Exp '+' Exp *
         | Exp '**' Exp *
         | '(' Exp ')'
         | ID '(' Args ')' // function call
         | 'ifzero' Exp 'then' Exp 'else' Exp *
Args    ::= Exp
         | Args ',' Exp
```

The productions marked with asterisks all give rise to shift/reduce conflicts when the grammar is processed by an LALR parser generator tool like Yacc/Jacc. Discuss for each case what the conflict means in terms of parse-trees alternatives, and which of the following responses you find most appropriate:

1. Relying on the default parser behavior that favors shift over reduce.
2. Adding precedence directives (show which ones).
3. Rewriting the grammar while keeping the syntax intact (describe how).
4. Redefining language  $L$  to use a different concrete syntax (describe how).

You may assume that established operator precedences and associativities should apply.

4. (5 marks)

- a) Rewrite the grammar for language  $L$  above such that it becomes more suitable for an implementation that uses the *top-down (recursive decent)* parsing technique.
- b) Show the concrete compiler code that parses an  $Exp$  with this technique, using Java or Python or some other established language. You may assume that the first element in the incoming token stream can be tested by boolean functions of the form  $tokenIsXXX()$  for each token or token type  $XXX$ , and that the token stream is advanced by the imperative function  $nextToken()$ . The  $Exp$  parser does not have to build any abstract syntax tree; all it should do is to return normally if the examined token stream belongs to language  $L$ , or throw an exception (call  $error()$ ) otherwise.

5. (5 marks)

A type system for language  $L$  would not be very interesting per se, since computations in  $L$  only work on the integer type. However,  $L$  nevertheless allows two distinct kinds of identifiers to be defined – parameters and functions – and the function identifiers are moreover distinguished by the number of arguments they take. It is therefore recommendable that a compiler for  $L$  performs a static analysis that checks that

1. only defined parameters are used as identifier expressions.
2. only defined functions are being called, and only with the right number of arguments.

Formalize such a static analysis for  $L$  by means of a logical inference system providing the judgments

$$E \vdash Exp \text{ OK} \quad \text{and} \quad E \vdash Program \text{ OK}$$

where the environment  $E$  is a simple mapping from defined identifiers to the number of arguments each identifier takes (note that a parameter identifier takes 0 arguments!) and 'OK' is just a redundant reminder about what the judgments are supposed to mean.

You may assume that function definitions in  $L$  are *non-recursive*; i.e., the right-hand side of each definition only sees function names introduced in *previous* definitions. However, should you rather prefer to define a system where functions are indeed recursive (a slightly more complex task), you are welcome to do so instead.

6. (5 marks)

Draw a picture that describes the layout of a method activation record (stack frame) for an object-oriented language like MiniJava running on a stack-based architecture like the IA32. The picture should clearly show where the following identifier categories are stored: method parameters (including the special pointer *this*), method-local variables, instance variables of the current object (i.e., object *this*), and code for the methods of the current object. For the last part you may assume that the implementation uses the virtual function-table (*vtable*) technique for accessing methods.

7. (5 marks)

Describe in detail the algorithm for constructing the graph of *basic blocks* in a sequence of instructions, where each instruction is either a *label*, a *jump* to a label (might be conditional), or a non-branching instruction like *load* or *add*. Use informal language for your description, but be as precise as you can. The goal is that a person who has never heard of the basic block concept should be able to follow the algorithm without difficulty.

8. (5 marks)

Give short but clear definitions of the following terms:

- a) Common subexpression elimination
- b) Copy propagation
- c) Dead code elimination
- d) Peephole optimization
- e) Instruction selection