Lunds Tekniska Högskola
Datavetenskap

Exam in Compiler Construction

April 25, 2001, time: 14.00-19.00

The total number of points is 40. In addition you can get a maximum of another 8 points from the “seminarieövningarna”. To pass the exam you need 20 points. You should try to solve all the problems in the exam.

Simple solutions result in more points than complex solutions. Names in grammars and implementations should be simple and descriptive rather than non-descriptive or misleading.

The following literature is allowed to be used during the exam: course book (Appel), all additional distributed course material (slides, seminar-exercises, lab descriptions, extra material for the labs, articles on JastAdd and design patterns), and solutions for the lecture exercies. You may also bring your own handwritten notes from the lectures and seminar-exercises.

No more literature than the above is allowed during the exam. For example, you may not use your own lab solutions, old exams, or other books. It is not allowed to use a computer during the exam.

To be allowed to write the exam you must have completed and passed all the lab-exercises.

1 Translation of reference lists

The following example shows an excerpt from a textual database of literature references:

```
article{
   author    = J. Earley,
   title     = "An efficient context-free parsing algorithm",
   journal   = "CACM",
   year      = 1970,
   volume    = 13,
   number    = 2,
   pages     = 94-102
}
book{
   author = Bruce Shriver and Peter Wegner,
   title = "Research Directions in Object-Oriented Programming",
   publisher = "The MIT Press",
   year = 1987
}
book{
   author = A. W. Appel,
   title = "Modern Compiler Implementation in Java",
   publisher = "Cambridge University Press",
   year = 1998,
   ISBN = 0-521-58388-8
}
```

The database format follows the following informal rules: A database consists of a list of references to books or journal articles. If a book or an article has several authors these are separated by the keyword “and”. An author name consists of one or more first names or initials, followed by a last name. Title of articles, books and journals are arbitrary strings. Years are written with exactly 4 digits. Journal volumes, numbers and pages are written with an arbitrary number of digits. The ISBN-numbers are written as an arbitrary mix of digits and dashes. The different items for a reference must appear in the same order as in the example and be separated by commas.

For an article, the items on author, title, journal, and year are mandatory. The items on volume and number are not necessary, but if one of them is present, the other one must also be present. The item on pages is not necessary, but may only be present if volume and number are present. For books, all items are mandatory except for the ISBN item.

a) Design an unambiguous context-free grammar G_1 for the reference lists described above. G_1 must be written in EBNF notation. The example above must be derivable from G_1. Reference lists that break the informal rules described above must not be in the language. Named tokens in G_1 must be described by regular expressions (using either notation of Appel or the JavaCC notation). (6p)

The reference lists are intended to be translated to a number of different formats, for example text formats for different word processors (like LaTeX, or RTF for Word), to markup languages like XML, or to simple unformatted text. To make this easy, the database is parsed and an abstract syntax tree is constructed from which it is easy to translate to the various other formats.

b) Design an object-oriented abstract grammar for representing the reference lists. Use the JastAdd notation (the notation you used in the lab exercises). (4p)

c) Implement a translator that, based on the abstract syntax tree, prints the reference list as unformatted text with one reference per line according to the following example: (Only the last name of the author should be printed; if there is more than one author, only the first one should be printed, followed by the text “et al.”; ISBN numbers should not be printed.)

> Earley: An efficient context-free parsing algorithm, CACM, 13(2):94-102, 1970.
> Shriver et al.: Research Directions in Object-Oriented Programming, The MIT Press, 1987.
> Appel: Modern Compiler Implementation in Java, Cambridge University Press, 1998.

If you need to extract substrings from string variables, you can introduce suitable methods for this. You don’t need to implement those methods. It is sufficient if you give some example and explain what they are intended to do.

Structure your implementation as you like (using either visitors or jadd-files). Explain how your implementation is intended to be called. (6p)

2 Ambiguous grammar

Consider the following context-free grammar, G_2:

```
E → "!" E
E → E "&&" E
E → ID
E → "(" E ")"
```

a) G_2 is ambiguous. Explain what is meant by an ambiguous grammar and give typical examples that show the ambiguities in G_2. (4p)

b) Write a new grammar, G_3, which is equivalent with G_2, but which is unambiguous. G_3 must be written in canonical form. Explain how you have chosen to solve the ambiguities in terms of precedence and/or associativity. (4p)

c) Is your grammar G_3 LL(1) or not? Explain why. (If you did not succeed in constructing G_3 you may instead of answering this question construct an LL(1) table for G_2.) (2p)

3 Bank accounts

Consider the following Java classes. The Java class Account models a simple bank account where you can deposit and withdraw money. If the account is overdraft, that is, more money is withdrawn than what is present in the account, an error message is printed. The class TestAccount is used for testing the code in Account.

```
final class Account {
   int balance = 0;
   void deposit(int amount) {
      balance = balance + amount;
   }
   void withdraw(int amount) {
      if (amount > balance)
         overdraft(amount - balance);
      else
         balance = balance - amount;
   }
   void overdraft(int amount) {
      /* PC */
      System.out.println("An account was overdraft by the following amount: " + amount);
   }
}

final class TestAccount {
   public static void main(String[] args) {
      TestAccount TA = new TestAccount();
      TA.testOverdraft();
   }
   void testOverdraft() {
      Account a = new Account();
      a.deposit(100);
      a.withdraw(150);
   }
}
```

The classes Account and TestAccount are declared as final which means that they can not have subclasses. This allows calls to their methods to be implemented in a similar way as normal procedure calls, that is, with a jump to a label that can be decided during compilation (in contrast to virtual methods where calls are implemented by indirect jumps via a virtual table).

a) Suppose that the execution starts in the method main in the class TestAccount. Draw a sketch of the stack and heap at the point of execution marked by the comment /* PC */. Your skectch should show what objects and frames that are allocated, dynamic and static links (where such are needed), and variables and parameters with integer or reference values (you don't need to draw the vector parameter args, though). The sketch should also show frame pointer, stack pointer, and heap pointer. Explain which static links are needed and which are not needed and why. (6p)

b) Translate the method withdraw to the intermediate code that was presented at lecture F11 (4p)

c) Extend the intermediate code with new instructions that can be used for translating the method testOverdraft. Explain how the new instructions work and translate testOverdraft to the extended intermediate code. (4p)