

Midterm Review Sheet

CS 5363, Fall 2007, Dr. Jeffery von Ronne

Objectives to be covered on the midterm by topic

1. Overview of Compilation and PL Implementation

(a) compilation and interpretation (Scott Ch. 1)

- i. **understand and be able to explain how compilation and interpretation differ**
- ii. *understand why compilation is usually faster than interpretation*
- iii. know what bootstrapping is
- iv. understand how interpreters and compilers can be combined to implement a language like Java

(b) structure and phases of a compiler (Cooper, 1.4, 1.5)

- i. **know the major phases of a modern optimizing compiler are and what their purposes are**
- ii. **understand the role of the scanner in a compiler**
- iii. **understand the role of the parser in a compiler**
- iv. *know what passes make up the back-end of a typical, modern optimizing compiler*
- v. **understand why it is advantage to split compilers between a front-end and a back-end**
- vi. understand why it is advantage to split compilers between a front-end, an optimization phase, and a back-end

2. Syntax (Scott, Ch. 2)

(a) language classes (regular, LL(1), LR(1), context free)

- i. **know that all LR(1) languages are context free**
- ii. **know that all LL(1) languages are also LR(1), and context free**
- iii. **know that all regular languages are also LL(1), LR(1), and context free**
- iv. *understand the distinction between the a class of languages and the classes of grammars that describe them (know that a language is regular/LL(1)/LR(1)/context free if there exists a regular/LL(1)/LR(1)/context free grammar for it)*
- v. *know the difference between an LL(k) and an LL(1) grammar*
- vi. *know the difference between an LR(k) and an LR(1) grammar*
- vii. know that there is an LR(1) equivalent grammar for every language described by an LR(k) grammar
- viii. **know that most scanners are based on regular expressions describing the different types of tokens**
- ix. **know that most parsers are based on grammars that describe LR languages**

- (b) regular expressions
 - i. **know that every regular language can be described equivalently with a regular expression that generates it, an NFA that recognizes it, a DFA that recognizes it, or a regular grammar that generates it**
 - ii. **know that it is possible to mechanically convert between regular expressions, NFA's, and DFA's**
 - iii. **know that there are tools (e.g., lex) that can generate executable code to recognize arbitrary regular expressions using constant time for each character**
 - iv. know that there are tools (e.g., grep) that can search files for arbitrary regular expressions
- (c) finite automata, table-driven scanners
 - i. **know what a deterministic finite automata (DFA) is**
 - ii. **know what a non-deterministic finite automata (NFA) is**
 - iii. **understand the difference between a DFA and an NFA**
 - iv. **be able to simulate the execution of a DFA on some string**
 - v. **be able to convert simple regular expression into an NFA**
 - vi. be able to convert an NFA into a DFA
 - vii. *be able to convert a DFA into an executable program in a high-level programming language*
- (d) BNF, parse trees, and ambiguous grammars
 - i. **know the difference between a terminal and a non-terminal symbol**
 - ii. **understand what a grammar production represents**
 - iii. **know what a parse tree is and understand how they represent productions**
 - iv. *understand how precedence can be reflected in a parse tree*
 - v. *understand how associativity can be reflected in a parse tree*
 - vi. *know what an ambiguous grammar is*
 - vii. know that the same language can often be described by both ambiguous and unambiguous grammars
 - viii. understand why unambiguous grammars are desirable
 - ix. **be able to read, write, and understand BNF grammars**
 - x. **given a sentence in a language described by a simple BNF grammar, be able to draw a parse tree for that sentence**
- (e) top-down vs. bottom up parsing
 - i. **know that most top-down parsers are based on LL(1) grammars**
 - ii. **know that bottom-up parsers can be generated from LR(k), LALR(k), and SLR(k) grammars**
 - iii. *know that most bottom-up parsers are generated from LALR(1) grammars*

- (f) recursive descent parsers
 - i. **know what a recursive descent parser is**
 - ii. **understand how a recursive descent parser can be derived from an LL(1) grammar**
 - iii. **know what FIRST, FOLLOW, and PREDICT sets are**
 - iv. **understand why common prefixes are a problem for recursive descent parsers**
 - v. **know what left-recursion is and why recursive descent parsers have trouble with left-recursion**
 - vi. **given an LL(1) grammar, be able to compute the FIRST, FOLLOW, and PREDICT sets**
 - vii. **given a LL(1), be able to write a recursive descent parser for it**
 - viii. **given a left-recursive grammar, be able to derive a right-recursive grammar that describes the same language**
 - ix. *given a grammar with common prefixes, be able to left-factor it*
- (g) bottom up table-driven parsers
 - i. *know that for many LR(k) languages there are also SLR(k) and LALR(k) grammars that can be parsed using simpler parsers.*
 - ii. *understand how SLR parsers works conceptually*
 - iii. *know what an LR item is*
 - iv. **given the a description of the states and transitions of a CFSM that can parse a simple SLR(1) grammar, be able to derive the SLR(1) parse table**
 - v. **given a simple, SLR(1), and parse table, be able to trace the parsing of a simple sentence**

3. Names, Scopes, and Activation Records (Scott, Ch. 3)

- (a) variable lifetimes and storage management
 - i. **know what static, stack, and heap storage are**
 - ii. **understand the implications of stack storage for variable lifetimes**
 - iii. **given a sample program in a C-like language, be able to identify dangling pointers due to local variables leaving scope**
- (b) nested static scopes
 - i. **understand the difference between dynamic and static scoping**
 - ii. **given a simple program, be able to identify which declaration would be associated with a given reference based on static scoping rules**
 - iii. recursive procedures and activation records
 - iv. **know what an activation record is and what can be put in it**
 - v. **understand how an activation record pointer and control links can be used to maintain a stack of activation records**

- vi. *understand why it is necessary to store the return address in the activation record*
 - vii. *understand why it can be necessary to store parameters, local variable, and temporaries on the stack*
 - viii. **given a simple program, be able to simulate the maintenance of control links in activation records during function calls**
 - ix. **be able to simulate the passing of parameters through activation records during procedure calls**
 - x. *given a simple program, be able to simulate the maintenance of local variable and temporary storage in activation records during function calls*
 - xi. *given a simple program, be able to simulate the maintenance of return address in activation records during function calls*
- (c) nested procedures, procedure closures, and access links
- i. **know what an access link is**
 - ii. *understand how access links are used to access non-local variables in language with nested procedures*
 - iii. *know what a closure is*
 - iv. *understand why closures are necessary in language with first-class functions*
 - v. **be able to simulate the initialization of an activation record's access link from a closure during a procedure call**
 - vi. *know why function pointers (rather than closures and access links) are sufficient for C*

bold denotes important objective, expected to be satisfied

italics denotes objectives, expected of 'A' students

normal font denotes tertiary objectives that are also desirable