

**CS 405/505 PROGRAMMING LANGUAGES**  
**TOPICS TO BE COVERED ON EXAM #1**  
**September 21, 2005**

The exam will cover *everything we have discussed* from Chapters 1-4 in the text and all presented materials, but emphasizing the material covered in class (e.g. very little of Chapter 2 was explicitly mentioned, from Chapter 3, we covered only syntax and attribute grammars, while topics of Chapter 4 were covered in greater detail (note that we did not cover LR parsing at the level of detail given in the text)). The style of the exam will be very flexible, possibly consisting of fill in the blank, true or false (possibly with justification), multiple choice, matching, short answer (e.g. definitions or listing), and discussion questions. You will not be asked to work detailed problems such as writing any actual code (i.e. code in a specific language like Java - you should be able to write pseudo-code for a JLex token specification or recursive descent parser). You should have some ideas about the theory behind implementation of programming languages (e.g. what really does happen in lexical analysis). Some smaller problems such as drawing a parse/syntax tree are likely. In general, you should know how to solve the problems given in assignments even though you may not have to actually solve such problems on the exam.

A brief outline of the topics we have covered is described below. (This list is intended to be as complete as possible but may not be all inclusive.)

- *Programming Languages - Syntax.* Grammar models of syntax were discussed, particularly the context-free/Backus Naur Form grammar (BNF), extended BNF and syntax graphs. Topics included parse trees and syntax trees, and the limitations of context-free grammars in describing context-sensitive issues such as declarations.
- *Programming Languages - Semantics.* We discussed static and dynamic semantics. Attribute grammar was discussed as a formal method for static semantics (mainly).
- *Compilers and Interpreters.* The various modules were discussed and you should know what all of them do. We especially concentrated on lexical and syntax analysis, discussing recursive descent parsing as one way of accomplishing syntax analysis. The relationship between implementation strategies and theory should be known (e.g. regular expressions vs. lexical tokens and formal syntax vs. syntax analysis). You should know the differences between compiling and interpreting.

## CS 405/505 EXAM #1 SAMPLE QUESTIONS

1. The string representation of a token is called a(n) \_\_\_\_\_.
2. True or False. Context-sensitive syntax refers to that part of a context-free grammar that requires nondeterminism to be parsed.
3. List the phases of a *compiler*. Discuss in detail what each phase does and how the phases interact with each other. (You may draw a diagram to assist in this discussion if you wish.) Explain how an *interpreter* is different from a compiler.
4. A C++ class declaration is described in Extended Backus-Naur form as:

```

class-specifier ::=
    class identifier [ : identifier ] { member-declaration {member-declaration} }
member-declaration ::= identifier identifier function-definition
    | int identifier function-definition
    
```

Note that this syntax contains terminal symbols  $\{$  and  $\}$  which are different from the extended BNF meta-symbols  $\{$  and  $\}$ , and distinguished by underlining. Write an equivalent syntax graph and a recursive descent parser for this construction. Assume the existence of production rules for function-definition and corresponding parsing procedure `FUNCTION_DEFINITION`, and a scanning procedure `GET_TOKEN` to set a global string variable `TOKEN` to the token type of the next token. Furthermore, assume that `TOKEN` is already set to the first token when parsing begins.

5. Consider the attribute grammar below:

```

<string> ::= <a-n> <b-n> <c-n>
    <b-n> . count ← <a-n> . count
    <c-n> . count ← <a-n> . count
<a-n> ::= a
    <a-n> . count ← 1
    | <a-n>1 a
    <a-n> . count ← <a-n>1 . count + 1
<b-n> ::= b
    Condition: <b-n> . count = 1
    | <b-n>1 b
    <b-n>1 . count ← <b-n> . count - 1
<c-n> ::= c
    Condition: <c-n> . count = 1
    | <c-n>1 c
    <c-n>1 . count ← <c-n> . count - 1
    
```

- (a) Indicate which attributes are inherited, which are synthesized, and which are intrinsic.
- (b) Draw the attributed parse tree for the string "aaabbbccc", showing clearly the order of evaluation of all attributes.