CS 411 Midterm Feb 2008

Exam number:		P1:	/ 9	P4:	/ 10
		P2:	/ 4	P5:	/ 9
Student Name:		P3:	/ 7	P6:	/ 8
				P7:	/ 8
	SUBTOTALS:		/ 20		/ 35
Student ID number:	TOTAL:				**/ 55**

READ THIS CAREFULLY BEFORE PROCEEDING

DO NOT TURN THE PAGE UNTIL WE TELL YOU TO DO SO

CAUTION -- Candidates suspected of any of the following, or similar, dishonest practices shall be immediately dismissed from the examination and shall be liable to disciplinary action:

a) Making use of any books, papers or memoranda.

b) Speaking or communicating with other candidates.

c) Purposely exposing written papers to the view of other candidates. The plea of accident or forgetfulness shall not be received.

Instructions

1. This exam contains 7 problems.
2. Each candidate should be prepared to produce his/her student-ID.
3. READ AND OBSERVE THE FOLLOWING RULES:
   - No candidate shall be permitted to ask questions of the invigilators except in case of supposed errors or ambiguities in exam questions.
   - No candidate shall be permitted to enter the examination room after the expiration of one half hour, or to leave during the first half hour of the examination.
   - No candidate shall be permitted to leave during the last half hour of the examination. Please remain in your seat until your paper has been collected.
   - Smoking is not permitted during examinations.
4. You are given 80 minutes to complete this exam.
5. Give concise answers. Long answers will be looked down upon with particular disregard - you will get points deducted for fluff. Put all your answers on the exam. Do any rough work on the back of the exam pages. All rough work must be handed in.

1. There are three parts to the specification of a programming language. Describe in a few sentences what each of them specifies. (9 pts)

   a. Syntax (3 pts)

      Specifies the textual structure of grammatically correct programs. Usually this constitutes of two parts. One part specifies "word structure" or "lexical" structure and is usually described in terms for regular expressions. A second part specifies how the "words" are arranged into higher-level "parse tree" structures. This is usually described using some form of context free grammar formalism such as BNF or EBNF.

   b. Contextual constraints (3 pts)

      Specifies additional constraints that well-formed programs (i.e. programs that are acceptable as input to the compiler and for which semantics is properly defined) must obey. These constrains, typically scope and typing rules, are specified separately from the syntax because they cannot be expressed with context free grammars.

   c. Semantics (3 pts)

      Specifies the meaning of well-formed programs, i.e. what behavior does a program generate when it is executed.

2. Put the following algorithms in order from least powerful to most powerful: LR(0), LR(1), SLR, LALR(1). (4 pts)

   LR(0), SLR, LALR(1), LR(1)

3. Sablecc parser generator algorithm (7 pts)

   a. Which of the above 4 algorithms mentioned in question 2 is the one used by sablecc? (2 pts)

      LALR

   b. Is this a bottom-up or a top-down algorithm? (2 pts)

      Bottom-up

   c. Focussing on high-level issues that matter to someone using a parser generator tool (rather than details of the algorithms), briefly dicuss what the potential advantages/disadvantages are of bottom-up versus a top-down parser generators. (3 pts)

      Bottom-up

      - Harder to debug
      + more expressive => grammar does not need to be modified (as much) and this results in more intuitive parse tree structure.

      Top-down

      + Easier to debug and more intuitive.
      + Can be implemented manually
      - less expressive => may require extensive changes to the grammar to make it parsable. This results in less intuitive parse tree structure.

4. For each of the following statements indicate whether they are true or false. Explain your answer briefly. (10 pts)

a) EBNF is a more expressive notation than BNF, therefore some languages that can be defined using EBNF cannot be defined using BNF notation.

False, EBNF is more expressive only in the sense that it is more convenient to use, but all EBNF specifications can be converted into equivalent BNF specifications.

b) Every language that can be defined by a grammar (BNF) can also be defined by a regular expression.

False, regular expression languages can only specify languages that do not have "self embedding".

c) All of the following formalisms are equally powerfull in the sense that they can be used to define exactly the same class of languages: regular expressions, DFA, NDFA, NDFA-epsilon.

True, they are all equivalent in that they can recognize the same class of languages, called "regular languages".

d) The difference between statements and expressions in the Java language is that statements do not need to be type-checked but expressions do.

False, although statements do not have a type, we still need to type check the expressions they contain, and verify whether those expressions are of the correct type given the context in which they occur. For example, the test expression of a while statement should be of type boolean.

e) Although C is a statically scoped language, its stack frames do not need to contain a static link.

True, C does not have nested procedures. Therefore, if C procedures would have a static link, each static link would just point to the global scope, so it would be redundant.

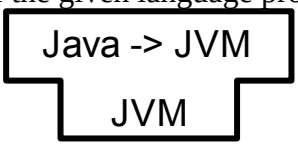
5. Tombstone Diagrams (9 pts)

You are given a standard Java SDK (downloaded from Sun). The SDK provides a bytecode compiler called **javac** in the form of a number of .class files containing Java bytecode. It also provides a JVM implementation, in the form of a single **java.exe** binary file that can be run directly on your x86-based windows box. You are also provided with the source code of a Java compiler called **myjavac**. This compiler takes a Java program (i.e. a number of .java source files) and compiles it directly to an x86 .exe file. **Myjavac** is itself implemented in the Java language.

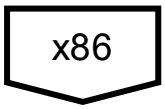
a. Draw tombstone diagram components for all of the given language processors (3 pts)

java.exe:

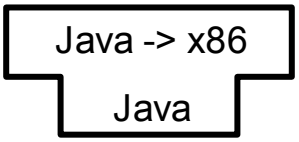
javac:



x86 processor:



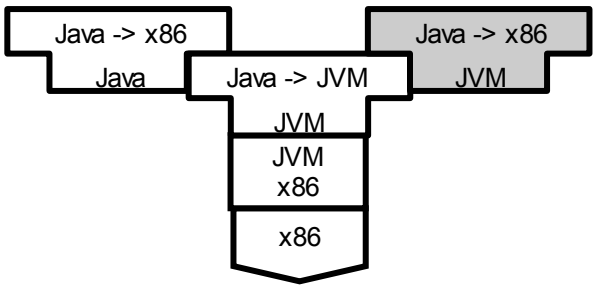
myjavac (source code):



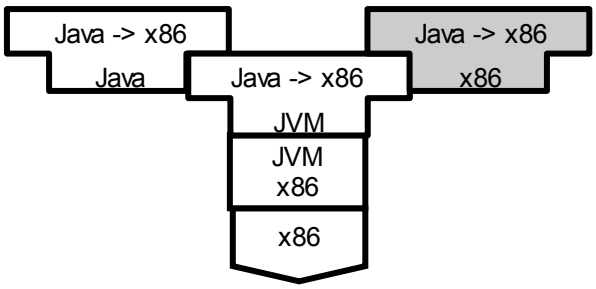
b. Draw a series of tombstone diagrams showing how you can bootstrap the myjavac compiler, using the Sun compiler and JVM. This should end with a diagram showing myjavac running natively on the x86 machine, compiling itself into native code. (6 pts)

1) Compile myjavac into JVM with javac
2) Compile myjavac into x86 using compiled myjavac from step 1
3) We now have a version of myjavac running natively on x86, capable of compiling itself

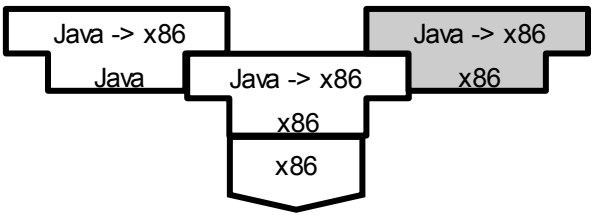
1)



2)



3)



6. Below is a grammar for a simple expression language. (8pts)

```
S   ::= E "$"
E   ::= E "+" T
      | T
T   ::= T "*" F
      | F
F   ::= "i"
      | "i" "(" E ")"
      | "1"
      | "(" E ")"
```

a. what is the starting state of the LR0 DFA (list the LR0 items in that state) (3pts)

```
S   ::= ● E "$"
E   ::= ● E "+" T
E   ::= ● T
T   ::= ● T "*" F
T   ::= ● F
F   ::= ● "i"
F   ::= ● "i" "(" E ")"
F   ::= ● "1"
F   ::= ● "(" E ")"
```

b. what state will the automaton be in after shifting an **`i`** token from the starting state? (3pts)

```
F   ::= "i" ●
F   ::= "i" ● "(" E ")"
```

c. Is this language LR 0? Explain your answer. (2pts)

No, there is a shift-reduce conflict:

```
F   ::= "i" ●    is a "reduce" item.

F   ::= "i" ● "(" E ")" is a "shift" item.
```

7. Below is a sablecc lexer and parser specification for a language called "simply typed lambda calculus". This language should be familiar to you from 311. It differs from the lambda calculus that you know (e.g. from 311 and Scheme) in that it is statically typed (all variable declarations are explicitly associated with a type). (8 pts)

```
Helpers
    letter = ['a'..'z'] | ['A'..'Z'] ;      digit = ['0'..'9'] ;
    cr = 13;                                lf = 10;  tab = 9;

Tokens
    lpar = '(' ;              col = ':' ;
    rpar = ')' ;              arrow = '->';
    lambda = 'lambda';        int = 'int' ;
    ident   = letter (letter|digit)* ;
    int_lit = digit digit* ;
    whitespace = (' ' | cr | lf | tab)* ;

Ignored Tokens

    whitespace ;

Productions

    program = exp ;

    exp = {lambda} lpar lambda ident col type rpar exp
        | {var_ref} ident
        | {apply} lpar [rator]:exp [rand]:exp rpar
        | {int} int_lit
        ;

    type
        = {function} [arg]:primary_type arrow [return]:type
        | {int} int
        | {primary} primary_type
        ;

    primary_type
       = {int} int
       | {parens} lpar type rpar
       ;
```

Note: This is the "nicest" solution, not the simplest solution.

When we run sablecc on this file the following error is reported:

```
shift/reduce conflict in state [stack: TLpar TLambda TIdent TColon PType TArrow PType *] on TArrow in {
       [ PType = PType * TArrow PType ] (shift),
       [ PType = PType TArrow PType * ] followed by TArrow (reduce)
}
```

a. Explain what this error means and why it arises (4pts)

This means the parser cannot chose whether to shift or reduce when it sees an arrow token after a PType.
This arises because the grammar is ambigous about how to parse something like
int -> int -> int: should the parsetree be like (int->int)->int or int->(int->int) ?

b. Fix this problem by marking-up the sablecc file above (e.g. cross-out what you want to change, and write your changes to the right, or below the existing definitions) (4pts)