CS352 - Compilers: Principles and Practice
Midterm Exam
2003 Oct 15

Instructions: Read carefully through the entire exam first, and plan your time accordingly. Note the relative weights of each segment, as a percentage of the total exam score.

This exam is **closed book, closed notes**. You may *not* refer to any books or other materials during the exam.

Write your answers on this exam. You may use both sides of the page.

When you are done, present your completed exam and your student identification to the instructor or proctors at the head table. If leaving before the exam period is concluded, please leave as quietly as possible as a courtesy to your neighbors.

Name:

Student Number:

Signature:

1. (PARSING: 50%) Consider the following grammar:

$$
\begin{array}{lllllll}
P & \rightarrow & E & & & & \\
E & \rightarrow & \texttt{int} & & E & \rightarrow & E * E \\
E & \rightarrow & E + E & & E & \rightarrow & E \,/\, E \\
E & \rightarrow & E - E & & E & \rightarrow & E \,\%\, E
\end{array}
$$

(a) (5%) Is this grammar ambiguous? Why or why not?

The grammar is ambiguous because even simple expressions like, "1 + 2 * 3" have multiple possible parses.

(b) (15%) Give an equivalent $LL(1)$ grammar that enforces left associativity and operator precedence. (HINT: The "%" or "modulus" operator has the same precedence level as division.) (OTHER HINT: You can use ϵ-productions to make the next question easier.)

$$
\begin{array}{rcl}
P & \rightarrow & E \\
E & \rightarrow & F\ E' \\
E' & \rightarrow & \texttt{+}\ E \\
E' & \rightarrow & \texttt{-}\ E \\
E' & \rightarrow & \epsilon \\
F & \rightarrow & \texttt{int}\ F' \\
F' & \rightarrow & \texttt{*}\ F \\
F' & \rightarrow & \texttt{/}\ F \\
F' & \rightarrow & \texttt{\%}\ F \\
F' & \rightarrow & \epsilon
\end{array}
$$

(c) (10%) Construct a table showing *nullable*, *FIRST*, and *FOLLOW* sets for all of the nonterminals used in your grammar in 1B.

γ	nullable(γ)	FIRST(γ)	FOLLOW(γ)
P	no	`int`	\$
E	no	`int`	\$
E'	yes	$+, -$	\$
F	no	`int`	$+, -, \$$
F'	yes	$*, /, \%$	$+, -, \$$

(d) (10%) Construct the $LL(1)$ predictive parsing table for your grammar. (HINT: It should have one row for each nonterminal in your grammar, and one column for each of the six terminals [$+, -, *, /, \%$,`int`] plus the special end-of-file terminal, \$.)

	`int`	`(+|-)`	`(*|/|%)`	`$`
P	$P \to E$			
E	$E \to FE'$			
E'		$E' \to (+\|-)E$		$E' \to \epsilon$
F	$F \to$ `int` F'			
F'		$F' \to \epsilon$	$F' \to (*\|/\|\%)F$	$F' \to \epsilon$

(e) (5%) Complete the table below, showing the production expansions that result from running the LL(1) predict(a|i)ve parsing table on the input string,

```
3 * 4 % 5 + 6
```

Use the "•" to show the location of the cursor in the input string as the parser consumes tokens. The first and last steps in the table are already given for you.

`•3 * 4 % 5 + 6$`	$P \rightarrow E$
`•3 * 4 % 5 + 6$`	$P \rightarrow FE'$
`3 •* 4 % 5 + 6$`	$P \rightarrow \texttt{int}\ F'E'$
`3 * •4 % 5 + 6$`	$P \rightarrow \texttt{int}\ {*FE'}$
`3 * 4 •% 5 + 6$`	$P \rightarrow \texttt{int} * \texttt{int}\ F'E'$
`3 * 4 % •5 + 6$`	$P \rightarrow \texttt{int} * \texttt{int}\ \%FE'$
`3 * 4 % 5 •+ 6$`	$P \rightarrow \texttt{int} * \texttt{int}\ \%\ \texttt{int}\ F'E'$
`3 * 4 % 5 •+ 6$`	$P \rightarrow \texttt{int} * \texttt{int}\ \%\ \texttt{int}\ E'$
`3 * 4 % 5 + •6$`	$P \rightarrow \texttt{int} * \texttt{int}\ \%\ \texttt{int}\ +E$
`3 * 4 % 5 + •6$`	$P \rightarrow \texttt{int} * \texttt{int}\ \%\ \texttt{int}\ +FE'$
`3 * 4 % 5 + 6•$`	$P \rightarrow \texttt{int} * \texttt{int}\ \%\ \texttt{int} + \texttt{int}\ F'E'$
`3 * 4 % 5 + 6•$`	$P \rightarrow \texttt{int} * \texttt{int}\ \%\ \texttt{int} + \texttt{int}\ E'$
`3 * 4 % 5 + 6$•`	$P \rightarrow \texttt{int} * \texttt{int}\ \%\ \texttt{int} + \texttt{int}$

(f) (5%) Draw the abstract parse tree that should result from the expansion above.

```
        +
       / \
      %   7
     / \
    *   5
   / \
  3   4
```

2. (VISITORS, INTERPRETERS: 10%) Consider anew our long-suffering grammar from the previous problem. Our parser builds Abstract Syntax Trees using the `Absyn` classes shown below:

```
public abstract class Absyn implements Hospitable
{   public Token begin, end;
    Absyn(Token begin, Token end)
    {   this.begin = begin;   this.end = end;   }
    public abstract  public int  accept(InterpVisitor v);
}
public abstract class Exp extends Absyn
{   Exp(Token begin, Token end)
    {   super(begin,end);   }
    public abstract void accept(Visitor v);
}
abstract public class BinExp extends Exp
{   public Exp left, right;
    public BinExp(Token begin, Token end, Exp lft, Exp rgt)
    {   super(begin, end);   left = lft;   right = rgt;   }
}
```

There are concrete subclasses of `BinExp` called `AddExp`, `SubExp`, `DivExp`, `MulExp` and `ModExp` which correspond to the binary operation +, -, /, *, and % respectively. There is a concrete subclass of `Exp` called `IntExp` for representing integer terminals. Each of the concrete subclasses of `Absyn` implements the `accept` method demanded by the `Hospitable` interface as follows:

```
  public int  accept(InterpVisitor v)     { return v.visit(this); }
```

The `InterpVisitor` interface is shown below:

```
public interface InterpVisitor
{   public int visit(Program n);
    public int visit(AddExp n);
    public int visit(SubExp n);
    public int visit(IntExp n);
    public int visit(MulExp n);
    public int visit(ModExp n);
    public int visit(DivExp n);
}
```

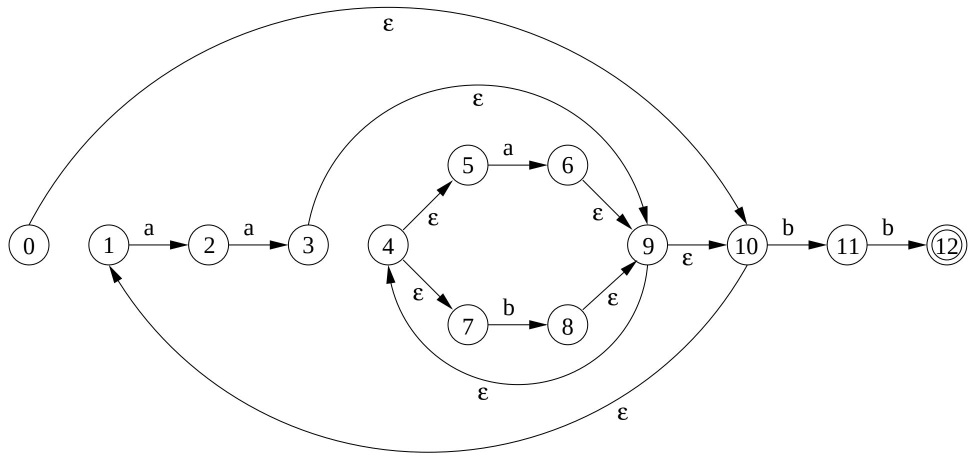
On the next page, complete the `Interpreter` class, a concrete subclass of `InterpVisitor` which returns the integer result of evaluating the mathematical expression represented by a given abstract parse tree. Two of the `visit` methods are already implemented for you.

```
public class Interpreter implements InterpVisitor
{
    public int visit(Program n) {   return n.e.accept(this);   }

    public int visit(IntExp n)
    {   return new Integer(n.value.toString()).intValue();   }

    // Write rest of visit() methods down here!


    public int visit(AddExp n)  {   return n.left.accept(this)
                                        +  n.right.accept(this);   }
    public int visit(SubExp n)  {   return n.left.accept(this)
                                        -  n.right.accept(this);   }
    public int visit(MulExp n)  {   return n.left.accept(this)
                                        *  n.right.accept(this);   }
    public int visit(ModExp n)  {   return n.left.accept(this)
                                        %  n.right.accept(this);   }
    public int visit(DivExp n)  {   return n.left.accept(this)
                                        /  n.right.accept(this);   }

}
```

3. (FINITE AUTOMATA: 20%) Consider the following regular expression:

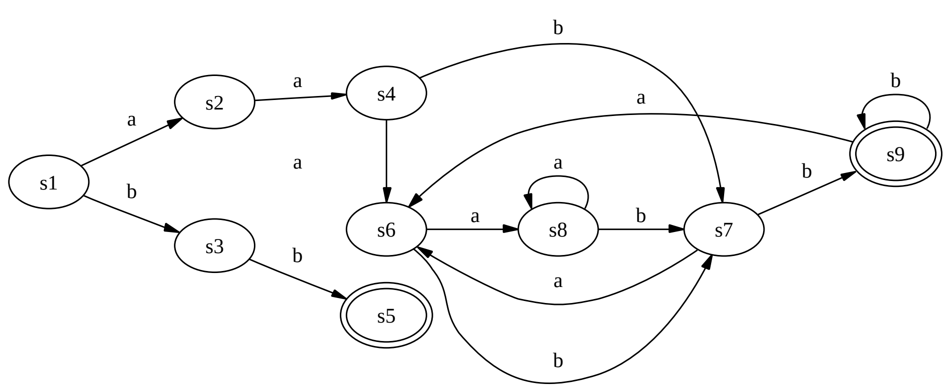
$(a\ a\ (b \mid a)^*)^* b\ b$

(a) (NFA: 10%) Construct a Nondeterministic Finite Automaton (NFA) for this expression. Take care to label start and accept states.



(b) (DFA: 10%) Construct the Deterministic Finite Automaton (DFA) for this expression. Take care to label start and accept states. (HINT: Fewer than 10 states are required!)

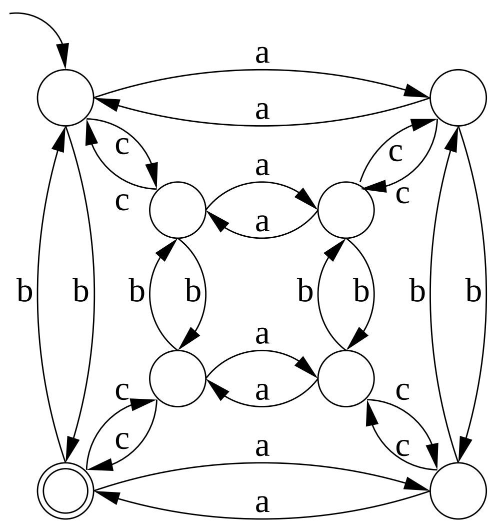
		a	b
S_1	{0,1,10}	S_2	S_3
S_2	{2}	S_4	
S_3	{11}		S_5
S_4	{1, 3, 4, 5, 7, 9, 10}	S_6	S_7
S_5^*	$\{12\}^*$		
S_6	{1, 2, 4, 5, 6, 7, 9, 10}	S_8	S_7
S_7	{1, 4, 5, 7, 8, 9, 10, 11}	S_6	S_9
S_8	{1, 2, 3, 4, 5, 6, 7, 9, 10}	S_8	S_7
S_9^*	$\{1, 4, 5, 7, 8, 9, 10, 11, 12\}^*$	S_6	S_9



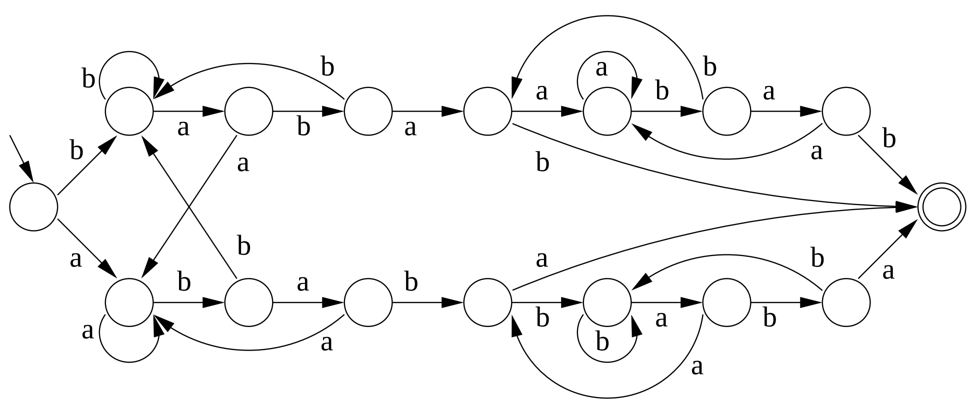
(With redundancy elimination, S_4, S_6, and S_8 can be merged.)

4. (MORE FINITE AUTOMATA: 10%) Write DFA's that recognize the following languages. Take care to label start and accept states.

   (a) (5%) $\{w \in \{a, b, c\}^* : w$ has an even number of a's, an odd number of b's, and an even number of c's. Consider zero to be even.$\}$



   (b) (5%) $\{w \in \{a, b\}^* : w$ has both $baba$ and $abab$ as a substring $\}$



5. (TYPES: 10%) Consider the class declarations below:

```
class Q5
{   String str = null;
    boolean y1 = false;
    Boolean y2 = new Boolean(false);
    int foo(int x, int y)     {  ...  }
    Q5  bar(Q5 q, boolean b)  {  ...  }
    static int x1 = 1;
    static int x2 = 2;
    public static void main(String[] args) {  ...  }
}

class P5
{   ...
    Q5  bar(Q5 q, boolean b)  {  ...  }
}
```

(a) (5%) Show the type representation of class `Q5`. (HINT: Use the "record of records" notation described in lecture and the slides, or the tagged record output seen in the project and MJ.)

```
CLASS(Q5,
  null,
  RECORD (
    FIELD (String str),
    FIELD (boolean y1),
    FIELD (Boolean y2)),
  RECORD (
    FIELD ((Q5,int,int)->int foo),
    FIELD ((Q5,Q5,boolean)->Q5 bar)),
  RECORD (
    FIELD (int x1),
    FIELD (int x2)),
  RECORD (
    FIELD ((String[])->void main)),
  ...)
```

(b) (5%) Can the function type for class `P5`'s method `bar()` be coerced to the function type for `Q5`'s method `bar()`? Why or why not?

No. The implicit `this` parameters of the non-static methods are unrelated, and therefore cannot be coerced.