CS352 - Compilers: Principles and Practice
Midterm Exam
2004 Oct 20

"You think you know when you can learn,
are more sure when you can write,
even more when you can teach,
but certain when you can program."
– *Epigrams in Programming*, Alan J. Perlis

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.

Hats should be removed, or worn brim backwards, to ensure that the proctors have an unobstructed view of your eyes.

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:

$$
\begin{array}{lll}
S & \rightarrow & S \; ; \; S \\
S & \rightarrow & \texttt{<ID> =} \; E \\
S & \rightarrow & \texttt{print <ID>}
\end{array}
\qquad\qquad
\begin{array}{lll}
E & \rightarrow & E \; + \; E \\
E & \rightarrow & E \; * \; E \\
E & \rightarrow & \texttt{<ID>} \\
E & \rightarrow & \texttt{<NUM>}
\end{array}
$$

where `<ID>` is an identifier starting with a letter and consisting of letters, digits, and underscores, and `<NUM>` is a number consisting of the digit "`0`" or any non-zero digit followed by a sequence of digits ranging from "`0`" to "`9`".

(a) (15%) Give an equivalent $LL(1)$ grammar that enforces left associativity and operator precedence. (HINTS: Use ϵ-productions to make the next question easier. One solution has 8 non-terminals and 13 productions; yours may differ.)

$$
\begin{array}{lll}
P & \rightarrow & S \; S' \\
S & \rightarrow & \texttt{<ID> =} \; E \\
S & \rightarrow & \texttt{print <ID>} \\
S' & \rightarrow & \texttt{;} \; S \; S' \\
S' & \rightarrow & \epsilon \\
E & \rightarrow & T \; E' \\
E' & \rightarrow & \texttt{+} \; E \\
E' & \rightarrow & \epsilon
\end{array}
\qquad\qquad
\begin{array}{lll}
T & \rightarrow & F \; T' \\
T' & \rightarrow & \texttt{*} \; T \\
T' & \rightarrow & \epsilon \\
F & \rightarrow & \texttt{<ID>} \\
F & \rightarrow & \texttt{<NUM>}
\end{array}
$$

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

γ	nullable(γ)	FIRST(γ)	FOLLOW(γ)
P	no	`<ID>`, `print`	\$
S	no	`<ID>`, `print`	`;`, \$
S'	yes	`;`	\$
E	no	`<ID>`, `<NUM>`	`;`, \$
E'	yes	`+`	`;`, \$
T	no	`<ID>`, `<NUM>`	`+`, `;`, \$
T'	yes	`*`	`+`, `;`, \$
F	no	`<ID>`, `<NUM>`	`*`, `+`, `;`, \$

(c) (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 [<NUM>, <ID>, print, +, *, ;] plus the special end-of-file terminal, \$.)

	<NUM>	<ID>	print	+	*	;	\$
P		$P \to S\ S'$	$P \to S\ S'$				
S		$S \to$ <ID> = E	$S \to$ print <ID>				
E	$E \to T\ E'$	$E \to T\ E'$					
T	$T \to F\ T'$	$T \to F\ T'$					
F	$F \to$ <NUM>	$F \to$ <ID>					
S'						$S' \to$; $S\ S'$	$S' \to \epsilon$
E'				$E' \to$ + E		$E' \to \epsilon$	$E' \to \epsilon$
T'				$T' \to \epsilon$	$T' \to * T$	$T' \to \epsilon$	$T' \to \epsilon$

(d) (5%) Given the input string below, draw the abstract parse tree that should result.

```
x = 5 + 4 * 3 ; print x
```

```
        ;
       / \
      =   print x
     / \
    x   +
       / \
      5   *
         / \
        4   3
```

(e) (10%) Complete the table below. On the right, show the production expansions that result from running the LL(1) predictive parsing table on the input string from the previous question. On the left, show which tokens, if any, are consumed by the productions. (We 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; the name you have chosen for your goal non-terminal may be different.

`•x = 5 + 4 * 3 ; print x$`	P
`•x = 5 + 4 * 3 ; print x$`	$S\ S'$
`x = •5 + 4 * 3 ; print x$`	<ID> = $E\ S'$
`x = •5 + 4 * 3 ; print x$`	<ID>= $T\ E'\ S'$
`x = •5 + 4 * 3 ; print x$`	<ID>= $F\ T'\ E'\ S'$
`x = 5 •+ 4 * 3 ; print x$`	<ID>= <NUM> $T'\ E'\ S'$
`x = 5 •+ 4 * 3 ; print x$`	<ID>=<NUM> $E'\ S'$
`x = 5 + •4 * 3 ; print x$`	<ID>=<NUM> + $E\ S'$
`x = 5 + •4 * 3 ; print x$`	<ID>=<NUM>+ $T\ E'\ S'$
`x = 5 + •4 * 3 ; print x$`	<ID>=<NUM>+ $F\ T'\ E'\ S'$
`x = 5 + 4 •* 3 ; print x$`	<ID>=<NUM>+<NUM> $T'\ E'\ S'$
`x = 5 + 4 * •3 ; print x$`	<ID>=<NUM>+<NUM> * $T\ E'\ S'$
`x = 5 + 4 * •3 ; print x$`	<ID>=<NUM>+<NUM>* $F\ T'\ E'\ S'$
`x = 5 + 4 * 3 •; print x$`	<ID>=<NUM>+<NUM>* <NUM> $T'\ E'\ S'$
`x = 5 + 4 * 3 •; print x$`	<ID>=<NUM>+<NUM>*<NUM> $E'\ S'$
`x = 5 + 4 * 3 •; print x$`	<ID>=<NUM>+<NUM>*<NUM> S'
`x = 5 + 4 * 3 ; •print x$`	<ID>=<NUM>+<NUM>*<NUM> ; $S\ S'$
`x = 5 + 4 * 3 ; print x•$`	<ID>=<NUM>+<NUM>*<NUM>; print <ID> S'
`x = 5 + 4 * 3 ; print x$•`	<ID>=<NUM>+<NUM>*<NUM>;print <ID>

2. (VISITORS, INTERPRETERS: 20%) Consider anew our long-suffering grammar from the previous problem. Our parser builds Abstract Syntax Trees using the `Absyn` classes shown below:

```
public class Program implements Visitable
{  public java.util.LinkedList stms = new java.util.LinkedList();
   public int  accept(IntVisitor v) { return v.visit(this); }
}
public abstract class Exp implements Visitable
{  public abstract int  accept(IntVisitor v);
}
public abstract class Stm implements Visitable
{  public abstract int  accept(IntVisitor v);
}
```

There are concrete subclasses called `AddExp`, `MulExp` and `AssignStm` (each with fields `Exp left`, and `Exp right`,) `NumExp` and `IdExp` for representing integer and identifier terminals (with fields `int num` and `String id`, respectively,) and `PrintStm` (with field `Exp e`.) Each of the concrete abstract syntax tree classes in the `Absyn` package implements the `accept` method demanded by the `Visitable` interface as follows:

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

The `IntVisitor` interface is shown below:

```
public interface IntVisitor
{ public int visit(Program p);
  public int visit(PrintStm  s);
  public int visit(AssignStm s);
  public int visit(AddExp e);
  public int visit(MulExp e);
  public int visit(NumExp e);
  public int visit(IdExp  e);
}
```

(a) (10%) On the next page, complete the `Interpreter` class, a concrete subclass of `IntVisitor` which runs the input program and prints the result of each `print` statement on a line by itself. Two of the `visit` methods are already implemented for you.

```
public class Interpreter implements IntVisitor
{ java.util.Hashtable environment = new java.util.Hashtable();

  public int visit(Program p)
  { java.util.Iterator i = p.stms.iterator();
    while (i.hasNext())
      {
        Stm stm = (Stm)i.next();
        if (i.hasNext()) stm.accept(this);
        else return stm.accept(this);
      }
    return 0;
  }

  public int visit(PrintStm  s)
  { int x = s.e.accept(this);
    System.out.println(x);
    return x;
  }
// Write rest of visit() methods down here!

  public int visit(AssignStm s)
  { int x = s.right.accept(this);
    environment.put(((IdExp)s.left).id, new Integer(x));
    return x;
  }

  public int visit(AddExp e)
  {   return e.left.accept(this) + e.right.accept(this);   }

  public int visit(MulExp e)
  {   return e.left.accept(this) * e.right.accept(this);   }

  public int visit(NumExp e) { return e.num; }

  public int visit(IdExp  e)
  {
    Integer i = (Integer)environment.get(e.id);
    return i.intValue();
  }
} // Interpreter
```

(b) (5%) Assuming that the abstract parse trees passed to our interpreter are grammatically correct, there is still at least one major *semantic* error that can be caught by this interpreter. What is it?

Reference to undefined identifier.

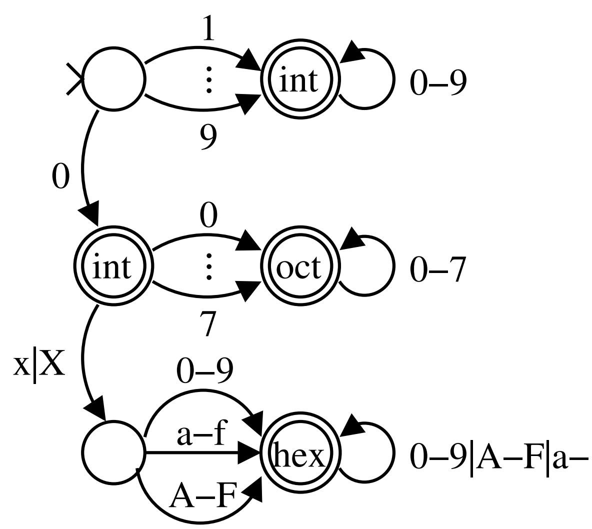
(c) (5%) What code should be inserted into the interpreter to detect this semantic error? Indicate which `visit()` method should be modified, and what code should be inserted.

```
public int visit(IdExp  e)
{ Integer i = (Integer)environment.get(e.id);
  if (null == i)
    {
      // This detects the error.  What we do here is a design
      //  decision.  Many answers are possible, but the code
      //  below prints an error message and returns a safe
      //  value so that interpretation does not stop.
      System.err.println("ERROR: Unbound identifier, "
                          + e.id + ".");
      return 0;
    }
  else return i.intValue();
}
```

3. (LEXICAL ANALYSIS: 10%)

Recall the definition of integer literals in our compiler for this semester: Integer literals consist of the digit "0", or a non-zero digit followed by an arbitrary sequence of digits "0" to "9" inclusive. Octal literals must start with a leading "0", and continue with one or more digits in the range "0" to "7". Hexadecimal literals are prefixed with "0x" or "0X", and continue with one or more digits plus the upper and lower-case letters "A" through "F" and "a" through "f".

(a) (5%) Draw a single Deterministic Finite Automaton that accepts any input string that is one of the three allowable kinds of integer literals.



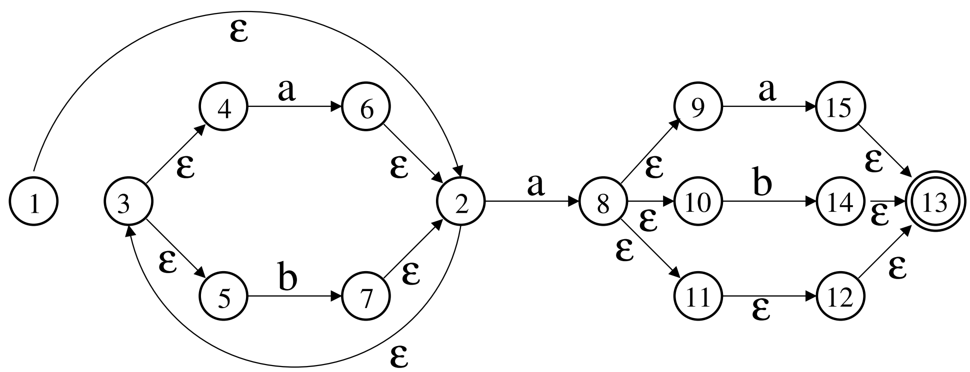
(b) (5%) Given the start of a JavaCC lexical declaration below, fill in the rules for JavaCC to recognize as terminals the three allowable kinds of integer literals.

```
TOKEN :
{
  < INT: ( ["1"-"9"] (["0"-"9"])* | "0" ) >
| < OCT: ( "0"["0"-"7"] (["0"-"7"])* ) >
| < HEX: ( "0x"(["0"-"9"]|["A"-"F"]|["a"-"f"])
           (["0"-"9"]|["A"-"F"]|["a"-"f"])*) >
```

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

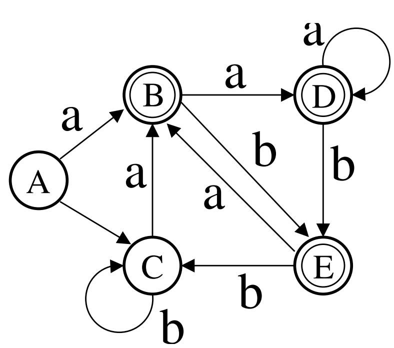
$$(a \mid b)^* \; a \; (a \mid b \mid \epsilon)$$

(a) (Nondeterministic Finite Automata: 5%) Construct an NFA for this expression using *Thompson's Method* as shown in the textbook and in lecture. Take care to label start and accept states.



(b) (Deterministic Finite Automata: 10%) Using the *Subset Construction* method, build the corresponding DFA for this expression. Take care to label start and accept states.

	a	b
A {1,2,3,4,5}	B	C
B {2,3,4,5,6,8,9,10,11,12,13}	D	E
C {2,3,4,5,7}	B	C
D {2,3,4,5,6,8,9,10,11,12,13,15}	D	E
E {2,3,4,5,7,13,14}	B	C



(c) (Minimization: 5%) Optimize your DFA from above to reduce it to the minimum number of states.