

Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.035 Spring 2013

Test I Solutions

I DFAs, NFAs, Regular Expressions and Context Free Grammars

For Questions 1 and 2, if a regular expression or context-free grammar can describe the language then provide one. Otherwise, write “N/A.”

1. [4 points]: The language of matched parentheses.

Solution: CFG:

$$S \rightarrow \epsilon \quad (1)$$

$$S \rightarrow (S)S \quad (2)$$

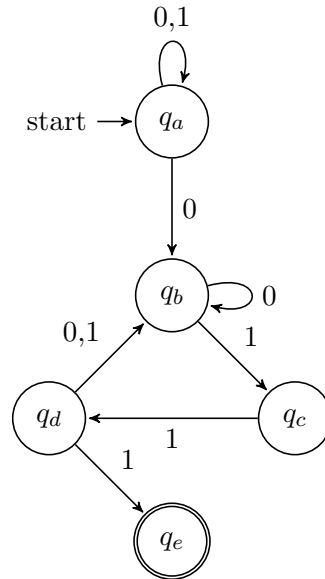
The grammar with $S \rightarrow (S)$ was also accepted since the question wasn't specific.

2. [4 points]: The language of even length strings over the alphabet $\{0, 1\}$. **Solution:**
Regular Expression: $((01)|(10))^*$

3. [4 points]: True or false: NFAs are more powerful (can recognize more languages) than DFAs. If false, explain. If true, give an example of a language that an NFA can parse that a DFA cannot.

Solution: False. Any NFA can be transformed into a DFA, although the DFA make take exponentially more space.

4. [8 points]: Give a regular expression for the following NFA:



Solution: $(0|1)^*00^*11((0|1)0^*11)^*1$ (Other valid regexes accepted)

II Hacking the Grammar

For Questions 5 through 7, consider the following grammar for a language with expressions:

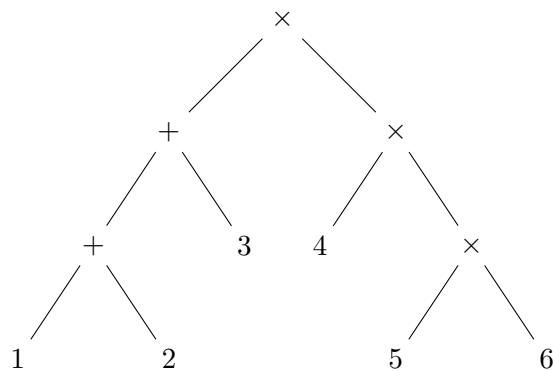
$$E \rightarrow E + E$$

$$E \rightarrow E \times E$$

$$E \rightarrow c$$

Where c is a number token.

5. [11 points]: Hack the grammar to give $+$ higher precedence than \times , to make $+$ left associative, and to make \times right associative. The grammar should produce a parse tree for the string “ $1 + 2 + 3 \times 4 \times 5 \times 6$ ” that reflects the evaluation order $((1 + 2) + 3) \times (4 \times (5 \times 6))$. This evaluation order is also reflected in the following abstract syntax tree:



Solution:

$$E \rightarrow F \times E$$

$$| F$$

$$F \rightarrow F + c$$

$$| c$$

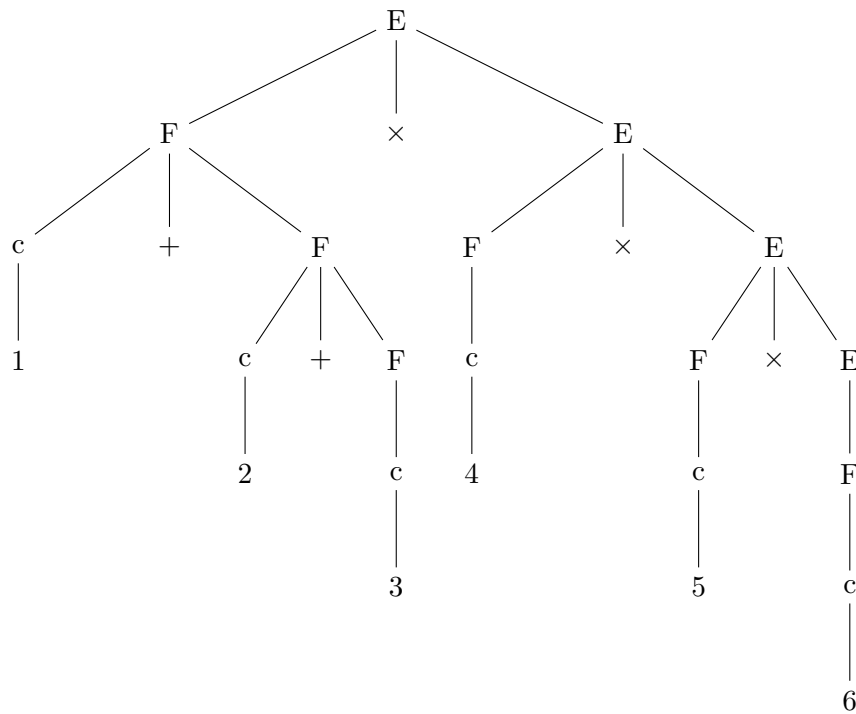
6. [11 points]: Remove left recursion from your answer to Question 5 to make the language parseable by a recursive descent parser with one token of lookahead. Do not worry about maintaining associativity.

Solution:

$$\begin{aligned}
 E &\rightarrow F \times E \\
 &| F \\
 F &\rightarrow c + F \\
 &| c
 \end{aligned}$$

7. [6 points]: Removing left recursion from your grammar leads to weird parse trees. Draw the **parse tree** (not AST) your grammar from Question 6 would produce for the string $1 + 2 + 3 \times 4 \times 5 \times 6$.

Solution:



8. [6 points]: Eliminating Shift-Reduce Conflicts:

Consider the language defined by the following grammar (where S is the only nonterminal):

$S \rightarrow \text{if } a \text{ b}$

$S \rightarrow \text{if } a \text{ b } \textit{else} \text{ c}$

If you give this grammar to a parser generator that produces a shift-reduce parser with no lookahead, then the parser generator will say that there is a shift-reduce conflict. Rewrite the grammar to eliminate the conflict.

Solution:

Oops. Not possible. Points for everyone.

III Implementing Object-Orientation: Descriptors and Symbol Tables

Use the diagram on the next page to answer the following three questions about this fragment of an expression interpreter.

```
class Environment { ... }

abstract class Expression {
    abstract int eval(Environment env);
}

abstract class BinaryExpression extends Expression
{
    Expression op1, op2;
}

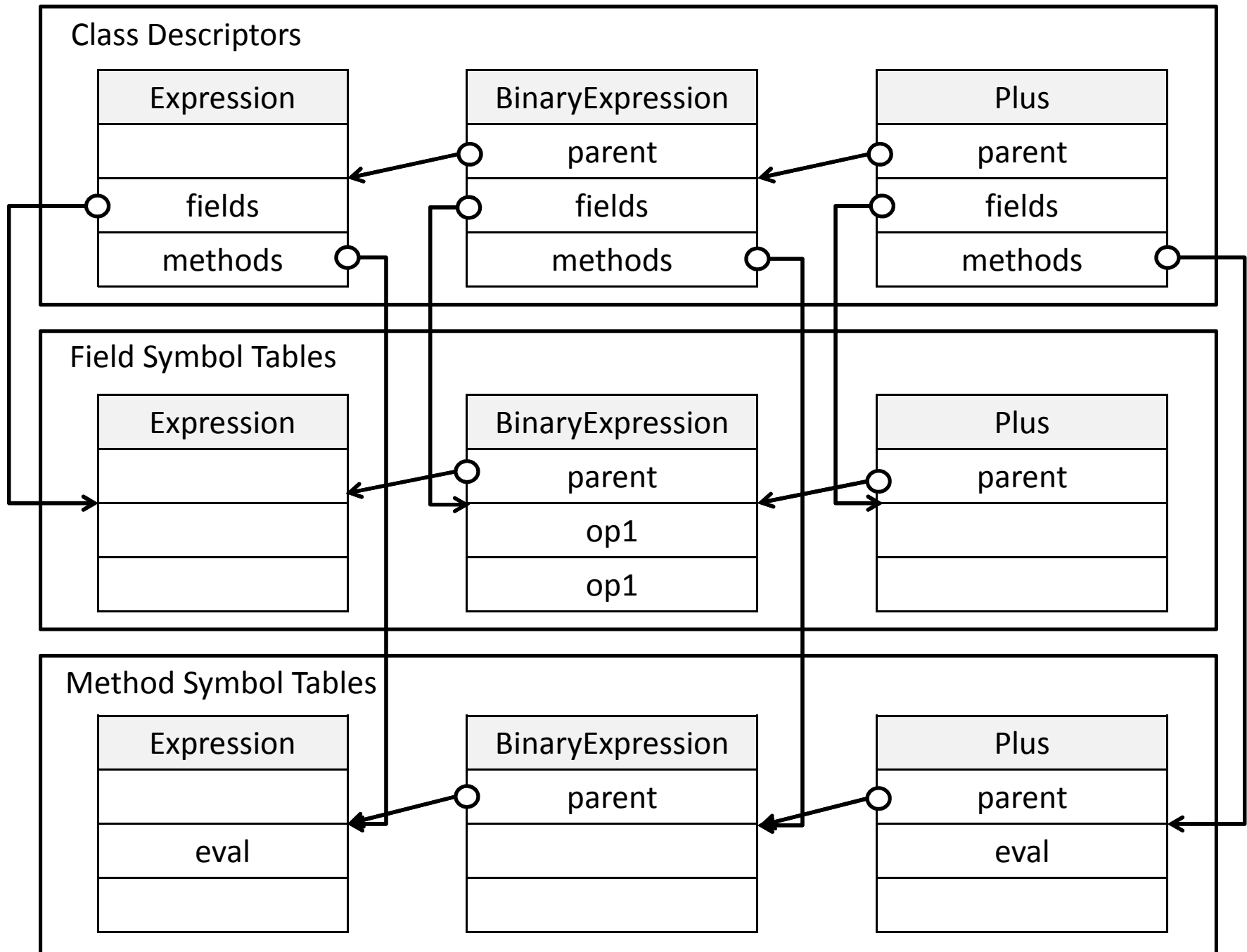
class Plus extends BinaryExpression
{
    int eval(Environment env) { return op1.eval(env) + op2.eval(env); }
}
```

9. [5 points]: Complete the entries of the class descriptors for each class. Use an arrow to connect the entry to a descriptor or symbol table where appropriate.

10. [5 points]: Complete the entries of the field symbol tables for each class. Use an arrow to connect the entry to a descriptor or symbol table where appropriate.

11. [5 points]: Complete the entries of the method symbol tables for each class. Use an arrow to connect the entry to a descriptor or symbol table where appropriate.

12. [5 points]: How does the method descriptor for a method with an `abstract` modifier differ from that of a method without the modifier?



IV Semantic Analysis

For this problem, you will write a semantic analyzer for the following simple language:

$$\begin{aligned} P &\rightarrow \text{Decls } E \\ \text{Decls} &\rightarrow D, \text{Decls} \mid \epsilon \\ D &\rightarrow ID = \text{read_int} \mid ID = \text{read_float} \mid ID = \text{read_string} \\ E &\rightarrow INT \mid FLOAT \mid STRING \mid ID \mid \text{concat}(E, E) \mid E + E \end{aligned}$$

The language consists of a sequence of variable declarations and a single expression consisting of constants (integer, float, and string), variable references, string concatenation, and addition. The keywords `read_int`, `read_float`, and `read_string` read a value of the given type from the user.

Implement a semantic analyzer in pseudo-code for the program element specified in each question. Your implementation should compute the *type attribute* of the given production. For example, the implementations for P and Decls are as follows:

$$P \rightarrow \text{Decls } E$$

```
{ P.type = (Decls.type == "void") ? E.type : Decls.type; }
```

$$\text{Decls} \rightarrow D, \text{Decls}_1$$

```
{ Decls.type = (D.type == "void") ? Decls1.type : D.type; }
```

$$\text{Decls} \rightarrow \epsilon$$

```
{ Decls.type = "void"; }
```

- Use the types “int”, “float”, “str”, and “void”.
- Use the type “err” when the program has a semantic error. Do not throw an exception.
- Use a global symbol table that you can manipulate and access with the functions `void add(string name, string type)` and `string lookup(string name)`. `lookup` returns `null` if the symbol hasn’t been defined.

13. [6 points]: Variable Declaration.

Goal: set `D.type` appropriately.

Semantic Rules:

- Each variable is declared at most once.
- The type of a variable is the type of the value assigned to it from the input.
- Semantically correct declarations have type “void”.

Assume: `ID.value` contains the name of the variable.

```
D → ID = read_int
{
  string t = lookup(ID.value);
  if (t == null) {
    add(ID.value, "int");
    D.type = "void";
  } else {
    D.Type = "err";
  }
}
```

14. [2 points]: Constant Expression.

Goal: set `E.type` appropriately.

Semantic Rule: a constant has its given type (e.g., an integer has type “int”).

```
E → STRING
{
  E.type = "str";
}
```

15. [4 points]: Variable Reference Expression.

Goal: set `E.type` appropriately.

Semantic Rules:

- A referenced variable must be declared.
- The type of a variable reference is the declared type of the variable.

Assume: `ID.value` contains the name of the variable.

```
E → ID
```

```
{
  string t = lookup(ID.value);
  E.type = (t != null) t : "err";
}
```

16. [6 points]: String Concatenation Expression.

Goal: set `E.type` appropriately.

Semantic Rule: string concatenation operates only on string operands.

Assume: `E1.type` and `E2.type` have already been recursively computed by the analyzer.

```
 $E \rightarrow \text{concat}(E_1, E_2)$ 
{
  if (E1.type == "str" && E2.type == "str") {
    E.type = "str";
  } else {
    E.type = "err";
  }
}
```

17. [8 points]: Addition Expression.

Goal: set `E.type` appropriately.

Semantic Rules:

- Addition operates only on integer and float operands.
- If one operand is a float, then the result of the addition is a float.

Assume: `E1.type` and `E2.type` have already been recursively computed by the analyzer.

```
 $E \rightarrow E_1 + E_2$ 
{
  if (E1.type == "err" || E2.type == "err")
  {
    E.type = "err";
  } else if (E1.type == "str" || E2.type == "str") {
    E.type = "err";
  } else if (E1.type == "float" || E2.type == "float") {
    E.type = "float";
  } else {
    E.type = "int";
  }
}
```