605.629: Programming Languages

Assignment 2

Sabbir Ahmed

1. [30 pts, grammars]

Consider the following unambiguous grammar for expressions,

```
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <term> | <term>
<term> → <term> * <factor> | <factor>
<factor> → ( <expr> ) | <id>
```

(a.) Modify the above BNF to give `+` precedence over `*` and force `+` to be right associative.

Answer:

To give `+` precedence over `*`, we can swap the operators in the grammar. To force `+` to be right associative, we can swap the operands.

```
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> * <term> | <term>
<term> → <factor> + <term> | <factor>
<factor> → ( <expr> ) | <id>
```

(b.) Modify the above BNF to add the `++` and the `--` unary operators of Java.

Answer:

```
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <term> | <term>
<term> → <term> * <factor> | <factor>
<factor> → ( <expr> ) | <id> | <id>++ | <id>--
```

(c.) Modify the above BNF to add a unary minus operator that has higher precedence than either `+` or `*`.

Answer:

```
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <term> | <term>
<term> → <term> * <factor> | <factor>
<factor> → ( <expr> ) | <id> | -<id>
```

---

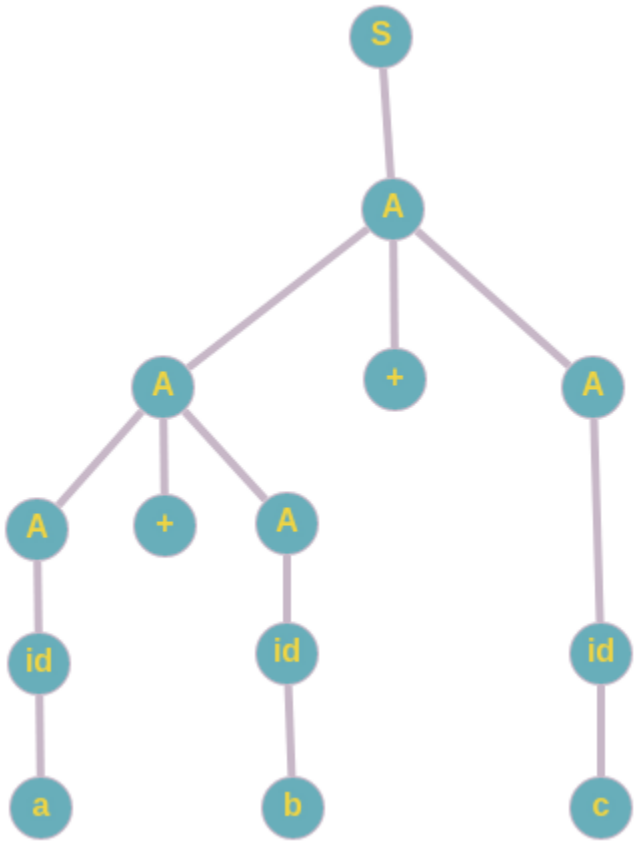
1. [30 pts, grammars]

Prove that the following grammar is ambiguous

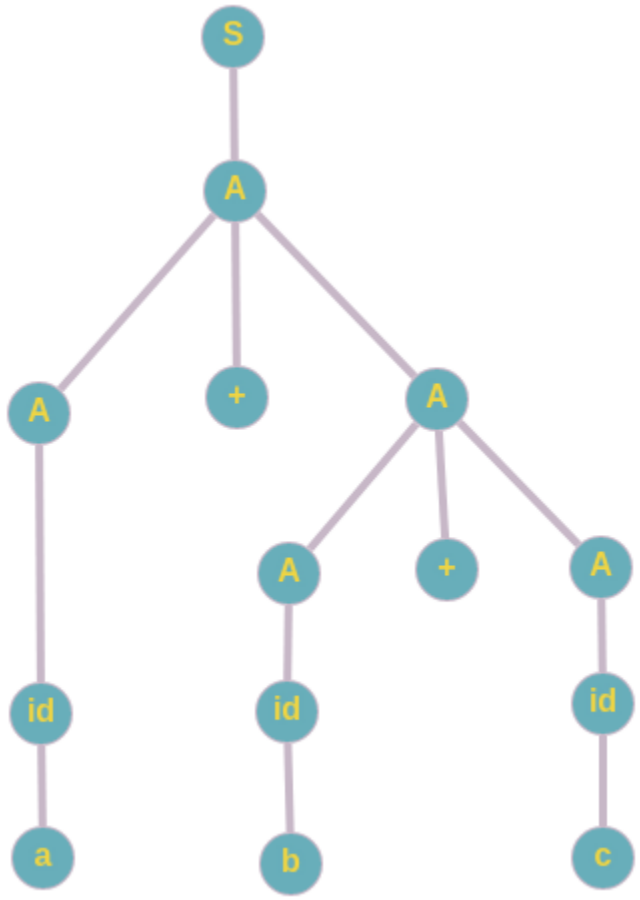
```
<S> → <A>
<A> → <A> + <A> | <id>
<id> → a | b | c
```

Answer:

A grammar is considered ambigious if it can generate a string with more than one parse tree. To generate the string `a + b + c` , the grammar would create the following parse trees:



Tree 1



Tree 2

Since the grammar generated more than one parse tree for `a + b + c`, it is ambiguous.

---

1. [10 pts, ply.lex and ply.yacc to build a lexer/tokenizer]

Use the Python ply library to build a calculator using the resource from the Ply web page (http://www.dabeaz.com/ply/ply.html)
Your lex and yacc scripts must be able to parse the following expression:

```
-1 + (2 * 3 + 4) * -5
```

Answer:

Script `calc.py` is attached separately.

---

1. [30 pts, semantics]

Consider the following grammar:

1. Syntax rule: `<assign> → <var> = <expr>`
   Semantic rule: `<expr>.expected_type ← <var>.actual_type`
2. Syntax rule: `<expr> → <var>[2] + <var>[3]`
   Semantic rule:

```
<expr>.actual_type ←
   if (<var>[2].actual_type = int) and (<var>[3].actual_type = int) then
       int
```

```
    else
        real
    end if
```

   Predicate: `<expr>.actual_type == <expr>.expected_type`
3. Syntax rule: `<expr> → <var>`
   Semantic rule: `<expr>.actual_type ← <var>.actual_type`
   Predicate: `<expr>.actual_type == <expr>.expected_type`
4. Syntax rule: `<var> → A | B | C`
   Semantic rule: `<var>.actual_type ← look-up(<var>.string)`

Modify the attribute grammar in the above BNF above to have: data types cannot be mixed in expressions, but assignment statements need not have the same types on both sides of the assignment operator.

Note that Attribute Grammars have:
`actual_type` - synthesized for `<var>` and `<expr>`
`expected_type` - inherited for `<expr>`
`env` - inherited for `<expr>` and `<var>`

Answer:

To force expressions to not mix data types, we only need to modify the second set of rules:

1. Syntax rule: `<assign> → <var> = <expr>`
   Semantic rule: `<expr>.expected_type ← <var>.actual_type`
2. Syntax rule: `<expr> → <var>[2] + <var>[3]`
   Semantic rule: `<expr>.actual_type ← <var>[2].actual_type`
   Predicate: `<var>[2].actual_type == <var>[3].actual_type`
3. Syntax rule: `<expr> → <var>`
   Semantic rule: `<expr>.actual_type ← <var>.actual_type`
   Predicate: `<expr>.actual_type == <expr>.expected_type`
4. Syntax rule: `<var> → A | B | C`
   Semantic rule: `<var>.actual_type ← look-up(<var>.string)`

The updated predicate rule states the `actual_type` for both of the types around the assignment operator has to be equal. The `actual_type` of `<expr>` then becomes the `actual_type` of `<var>[2]` (which is also the same as `<var>[3]`).
