1.

(a) I talked with Danny and he helped me to figure out question 3 part c.

(b) I acknowledge this.

2.

Two examples of semantic rules are:

1. C allows operands of many types to appear in an expression while Ada does not
2. C requires no tun time check while Java check as many as possible

The reason why we need semantic rules:

Usually the syntax of a language might not be fully described by context free grammar, but these can be checked easily in semantic analysis. Semantic allows us to enforce rules that go beyond mere form, and it provides the information we need in order to generate an equivalent output program.

3.

Given

<expr> -> number <expr\_tail>

<expr\_tail> -> - number <expr\_tail> | e

(a)

A close up of text on a white background

Description automatically generated

(b)

If it’s an attribute grammar,

expr.numbers := 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

expr\_tail.numbers := 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

or We can use const to define numbers, such as:

expr ->const expr\_tail

expr\_tail -> - const expr\_tail | E

then, 9 - 4 - 3 would be:

A picture containing object

Description automatically generated

4.

(a)

<Program> -> <FuncDef><FuncCall>

<FuncDef> -> <FuncName>’(’<al>’)’

<FuncCall> -> <FuncName>’(‘<parms>’)’

< FuncName> -> <id>

<parms> -> <id> <parms>

<parmst> -> ‘,’ <id> <parmst>

<al> -> <type><id> <alt>

<alt> -> ’,’ <type><id> <alt>

<type> -> ‘bool’ | ‘int’

<id> -> <letter><number>

<letter> -> a|b|c|d…….|z

<number> -> 0|1|2…|9

(b)

<Program> -> <FuncDef><FuncCall>

<FuncDef> -> <FuncName>’(’<al>’)’

* <FuncDef>.name := <FuncName>.name
* <FuncDef>.num := <al>.num

<FuncCall> -> <FuncName>’(‘<parms>’)’

* < FuncCall >.name := <FuncName>.name
* < FuncCall >.num := <parms>.num

< FuncName> -> <id>

* <FuncName>.name := <id>.val

<parms> -> <id> <parms>

* <parms>.num := 1 + <parmst>.num

<parmst> -> ‘,’ <id> <parmst>

* <parmst>.num := 1

<al> -> <type><id> <alt>

* <al>.num := 1 + <alt>.num

<alt> -> ’,’ <type><id> <alt>

* <alt>.num := 1

<type> -> ‘bool’ | ‘int’

<id> -> <letter><number>

* <id>.val := <letter>.val + <number>.val

<letter> -> a|b|c|d…….|z

<number> -> 0|1|2…|9

(c)

Because the decoration of the parse tree has the attribute value, so we can use

< FuncDef >.num and < FuncCall >.num to check if the parameters number of function call equals the parameters number of function declaration.