Amara Tariq
CS354
Buffenbarger
February 20, 2020

TA1

1.1)

a) A Lexical error detected by the scanner

    char a = ‘aaa’   //’aaa’ is too long to store in a char

b) Syntax error detected by the parcer

    System.out.println(“Syntax Error”)   //no semicolon

c) Static semantic error detected by semantic analysis

    Calling a method that has not been declared in the class that is calling that method

d) Dynamic semantic error detected by code generated by the compiler

    Trying to access an element beyond the size/bounds of the array
    Int array[10];
    array [11] = 404;   //the array size is only set to 10, 11 is out of bounds

e) Error that the compiler can neither catch nor easily generate code to catch

    Having a method and a variable that share the same name
    Int amara = 404;
    Public int amara (int a); //they can't share the same name

1.8)

This sort of dependence management is still accurate, as if one file is modified, it forces the other file to also recompile to make sure they run together. Where the unnecessary work comes up is if the change in the file does not affect or change how the secondary file runs and works. The file, even if uneffected, still has to be recompiled regardless. If file A is modified, and if file B for some reason doesn’t have the code that A depends on, an error can occur since B does not need to be recompiled if its a modification in A, since only file A depends on B. Since A depends on B, and A has a modification, B does not need to be recompiled since A depends on B. Therefore if A’s modifications requires B to be remodified and recompiled, there can be an error. It becomes tedious with a need to recompile both each time to make sure when the source code runs, what it is trying to find in the specific files is actually there and up-to-date with the version that is being ran.

2.1)

a) Strings in C.

    String -> “ (chars | backslashed_chars)* “
    Chars -> a-z|A-z
    Backslashed_chars -> \\ | \” | \n

b) Comments in Pascal.

{Hello! This is a comment in pascal!}
(* Hello! This is also a comment in pascal!*)

c) Numeric constants in C.

Digit ->0|1|2|3|4|5|6|7|8|9
noOneDigits -> 1|2|3|4|5|6|7|8|9
Integers -> noOneDigits digits* (ϵ|U|u|L|l|LL|ll)
Floating_point -> (((0|ϵ) noOneDigits noOneDigits)|ϵ)(ϵ|(. (digit))) (ϵ|((e|E) integers)) (ϵ|F|f|L|l)
Octal_digits -> 0|1|2|3|4|5|6|7
Octal_number -> 0 octal_digit (ϵ|U|u|L|l|LL|ll)
Hexa_digit -> 0|1|2|3|4|5|6|7|8|9|a-f|A-F
hexa_number -> 0 (x|X) hexa_digit (ϵ|U|u|L|l|LL|ll)
Hexa_float -> hexa_number (ϵ| (. hexa_digits)) (p|P) (ϵ|-|+) integers(ϵ|F|f|L|l)
C_num -> integer| octal_number|hexa_number|floating_point|hexa_float

d) Floating-point constants in Ada.

Digit ->0|1|2|3|4|5|6|7|8|9
Extended_digit -> digit| a-f |A-F
Integer -> digit digit*
Ada_integer -> digit ((.|ϵ) digit)*
extended_integer -> ada_integer ((. | ϵ) ada_integer)*
exponent -> (e|E) (+|-|ϵ) integer
extended_exponent -> (e|E) (+|-|ϵ) ada_integer
floating_point -> ((ada_integer ((. ada_integer|ϵ)) | (ada_integer # extended_integer ((. extended_integer)|ϵ) #)) (extended_exponent|ϵ)

e) Inexact constants in scheme.

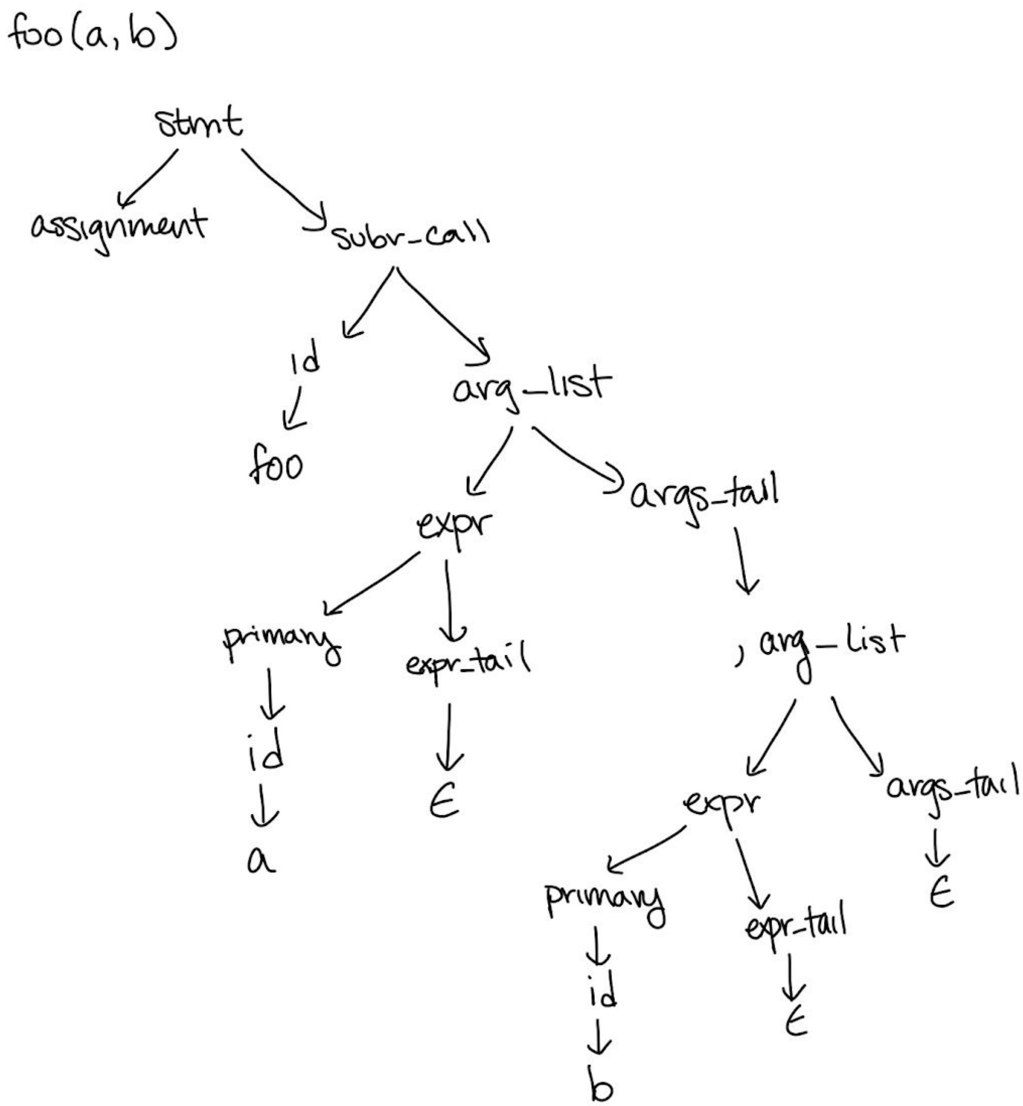
digit -> 0|1|2|3|4|5|6|7|8|9
digit+ # *(. # *) digit*.digit+ # *

f) Financial quantities in American notation.

Nodigit -> 1|2|3|4|5|6|7|8|9
Digit -> 0| Nodigit
group -> , digit digit digit
number -> $**(0| Nodigit (ϵ| digit | digit| digit) group *) (ϵ| . digit digit)

2.13)

a) Construct a parse tree for the input string foo(a,b).



b) Give a canonical (right-most) derivation of this same string

Stmt -> subr_call -> id(foo) (arg_list) -> id(foo) (expr args_tail) ->
id(foo) (expr ,arg_list) -> id(foo) (expr, expr args_tail) -> id(foo) (expr, expr) ->
id(foo) (expr, primary expr_tail) -> id(foo) (expr, primary) -> id(foo) (expr, id(b)) ->
id(foo) (primary expr_tail, id(b)) -> id(foo) (primary, id(b)) -> id(foo) (id(a), id(b))

2.17) extend grammar of figure 2.25 to include if statements and while loops along lines that are suggested in the example (below figure 2.25)

1. *program* → *stmt_list* $$
2. *stmt_list* → *stmt_list stmt*
3. *stmt_list* → *stmt*
4. *stmt* → id := *expr*
5. *stmt* → read id
6. *stmt* → write *expr*
7. *expr* → *term*
8. *expr* → *expr add_op term*
9. *term* → *factor*
10. *term* → *term mult_op factor*
11. *factor* → (*expr*)
12. *factor* → id
13. *factor* → number
14. *add_op* → +
15. *add_op* → -
16. *mult_op* → *
17. *mult_op* → /

Figure 2.25 **LR(1) grammar for the calculator language.** Productions have been numbered for reference in future figures.

```
abs := n
if n < 0 then abs := 0 - abs fi

sum := 0
read count
while count > 0 do
    read n
    sum := sum + n
    count := count - 1
od
write sum
```

If -> factor comp_op factor
While -> factor comp_op factor do stmt_list $$

Comp_op -> <
Comp_op -> <=
Comp_op -> >
Comp_op -> >=
Comp_op -> ==
Comp_op -> !=