BA-1

Hunter Barclay
CS 354
Sept. 30th, 2024

1.1

a.

```
float a = 4.a3f; // ERR
```

b.

```
float new = 4.0f; // ERR
```

c.

```
int a = 2.0f; // ERR
```

d.

```
int[] a = new int[1];
a[1] = 3; // ERR
```

e.

```
// Idea being that num could be larger than the max heap size, causing an allocation of
too much memory. Something that wouldn't be caught by the compiler and prolly not at
runtime until it crashes something.
Scanner sc = new Scanner(System.in);
int num = in.nextInt();
float[] arr = new float[num];
```

I guess? This is a weird question.

1.6

Interpretation and code generation's tree traversals are similar in the sense that they are both traversing the program in tree for to determine what the program is doing. Where they differ is where code generation is essentially reviewing the operations and inserting more to add additional checks whereas interpretation is actually using the operations to perform operations. An interpreter would run across an add operation and send the necessary data for execution (which may include arithmetic overflow check), whereas the code generator would insert new instructions into the program to add those checks (I'm unclear if it actually inserts it into the tree, or if the tree is moot at this point in the compilation).

1.8

`make` would cause unnecessary work in a situation where a file is barely a dependent of a target--i.e. you change a comment inside of a header file, which is included by numerous source files in a C program. This would fail to update dependencies if there are hidden dependencies of dependencies--i.e. header file `a.h` includes `b.h`, which is updated. You would only have make targets for say source file `a.c`, which may only include `a.h` as a dependency and not realize the relationship between `a.h` and `b.h`, resulting in no recompilation.

2.1

For the regex, I'll be using a backslash to indicate that it is actually matching a character versus syntax of the regex when I don't think it's super clear.

a.

Examples:

```
"Hello"
"\tWorld\n"
"'Hi'"
```

```
esc_char -> a | b | e | f | n | r | t | v
        | \ | " | x any_hex any_hex
        | u any_hex any_hex any_hex any_hex

valid -> non_backslash_or_double_quotes
valid -> \ esc_char

string_literal -> " valid* "
```

b.

Examples:

```
(* Hello *)
{ World }
(*****hi*****)
```

```
paren_comm_inter -> not_asterisk | ( asterisk not_paren_close )
paren_comm -> \( \* paren_comm_inter* \* \)

curly_comm -> { not_curly_close* }

comment -> paren_comm | curly_comm
```

c.

Examples:

```
0.345
045
0xa3
3.14
0x4.5p10
```

```
oct_dig -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
dec_dig -> oct_dig | 8 | 9
hex_dig -> dec_dig | a | A | b | B | c | C | d | D | e | E | f | F

unsigned -> u | U
long -> l | L
short -> s | S
float -> f | F
sign -> + | -
dec_int -> ( 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 ) dec_dig*
oct_int -> 0 oct_dig*
hex_int -> 0 ( x | X ) hex_dig+

integer -> ( sign | ^ ) ( dec_int | oct_int | hex_int ) ( unsigned | short | long | ^ )

dec_exp -> ( e | E ) dec_dig+
dec_float -> dec_dig+ . dec_dig* ( dec_exp | ^ )
```

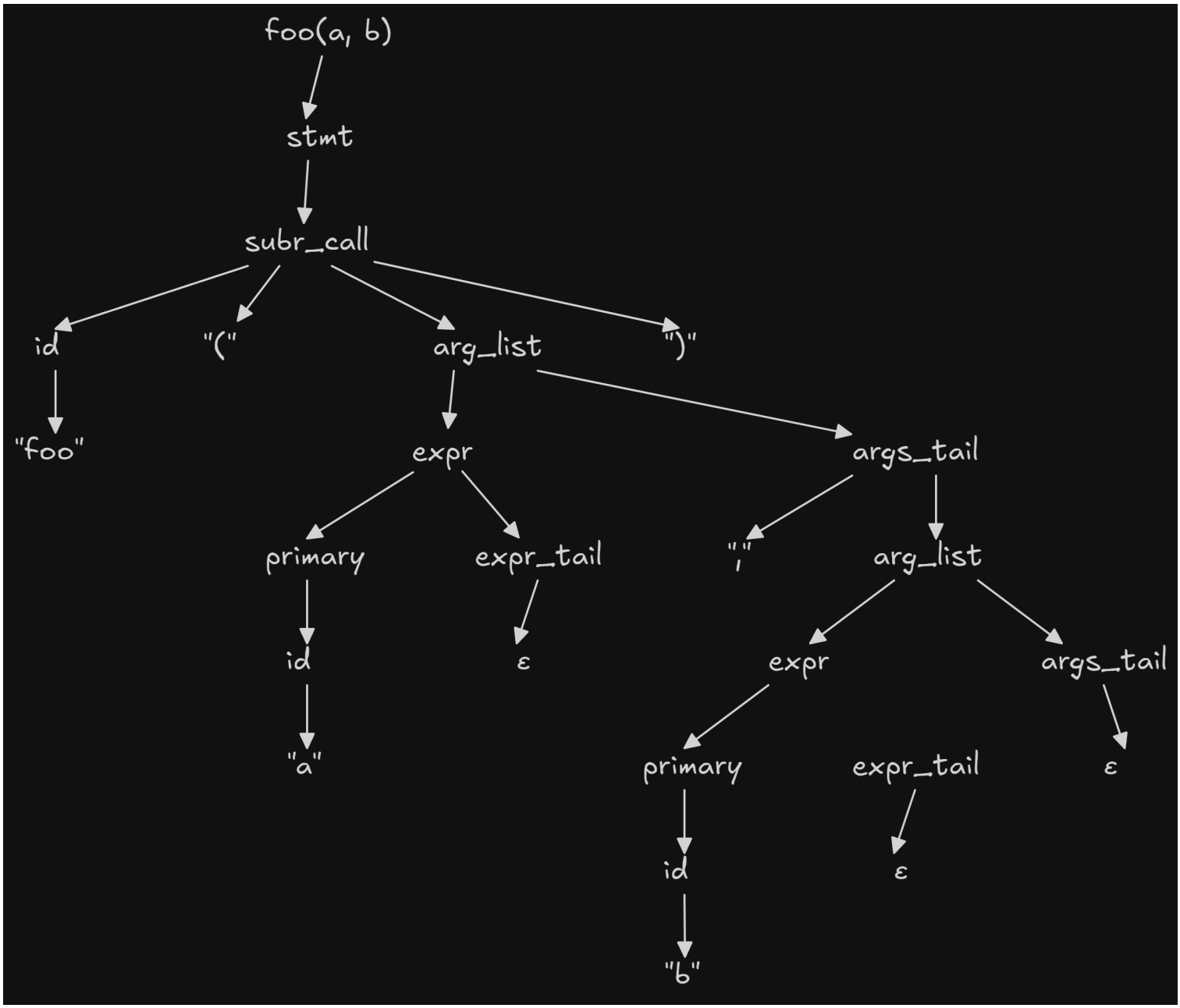
```
                -> dec_dig* . dec_dig+ ( dec_exp | ^ )
                -> dec_dig+ ( dec_exp | ^ )
hex_exp -> ( e | E ) dec_dig+
hex_float -> 0 ( x | X ) hex_dig+ . hex_dig* hex_exp
                -> 0 ( x | X ) hex_dig* . hex_dig+ hex_exp
                -> 0 ( x | X ) hex_dig+ hex_exp

float -> ( sign | ^ ) ( dec_float | hex_float ) ( float | long | ^ )

**FINAL**
numeric_constant -> integer | float
```

2.13

a.



b.

```
foo(a, b)

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

2.17

```
program -> stmt_list $$
stmt_list -> stmt_list stmt
stmt_list -> stmt
stmt -> id := expr
stmt -> read id
stmt -> write expr
stmt -> while cond do stmt_list od
stmt -> if cond then stmt_list if_tail
if_tail -> elif cond then stmt_list if_tail
if_tail -> else stmt_list fi
if_tail -> fi
cond -> expr comp_op expr
expr -> term
expr -> expr add_op term
term -> factor
term -> term mult_op factor
factor -> ( expr )
factor -> id
factor -> number
```

```
add_op -> +
add_op -> -
mult_op -> *
mult_op -> /
comp_op -> >
comp_op -> <
comp_op -> >=
comp_op -> <=
comp_op -> ==
comp_op -> !=
```