Colin Gabrielson
CIS 425 Fall '14
Prof. Zena Ariola

Assignment 1

4. 11)

(a)

Assume we evaluate g(e1 , e2) by starting to evaluate g, e1 , and e2 in parallel, where g is the function defined above. Is it possible that one process will have to wait for another to complete? How can this happen?

if e1 = n (operator) m, then the value of e1 will need to be evaluated before parts of g, like (if = x = 0) and (else if x+y = 0). Otherwise the x value in those statements is unknown.

(b)

Now, suppose the value of e1 is zero and evaluation of e2 terminates with an error. In the normal (i.e., eager) evaluation order that is used in C and other common languages, evaluation of the expression g(e1 , e2) will terminate in error. What will happen with lazy evaluation? Parallel evaluation?

With lazy evaluation the function will not terminate with an error, since if x =0 will evaluate to true and then 1 will return. With parallel evaluation the error will be evaluated, since everything is evaluated whether or not the relevant if statements return true. Assuming that the parallel evaluation will terminate when it sees an error, then this is what will happen even though if x = 0 still evaluates to true

(c)

Suppose you want the same value, for every expression, as lazy evaluation, but you want to evaluate expressions in parallel to take advantage of your new pocket-sized multiprocessor. What actions should happen, if you evaluate g(e1 , e2) by starting g, e1 , and e2 in parallel, if the value of e1 is zero and evaluation of e2 terminates in an error?

|A|
Each expression will still be evaluated in parallel so it will be seen that e2 terminates with an error (and all other evaluations as well will be seen.) However, since the path of e1 is taken since (if x = 0) is true, then an error will not terminate and "then 1" will still return

(d)

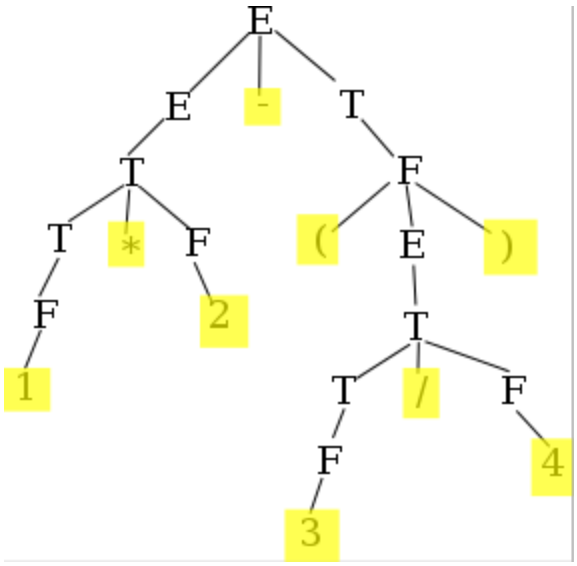
Suppose now that the language contains side effects. What if e1 is z and e2 contains an assignment to z; can you still evaluate the arguments of g(e1 , e2) in parallel? How? Or why not?

The side effect is relevant since if e2 contains an assignment to z, then this causes the evaluation of e1 to be indeterminate (this is a race condition). This sheds light on why, in functional programming variables do not change value.
We could still evaluate the arguments in parallel by instead of changing the value of z in e2 we would copy the value into a new variable, z' and alter that instead of z. Then, if the path of e2 is returned instead of e1, we would use the value of z', e.i. return it or whatever is necessary.

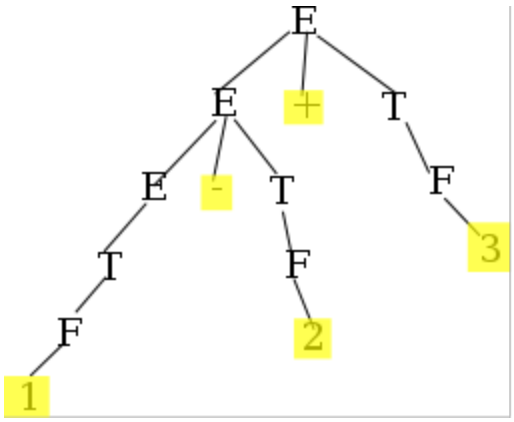
2)

E ::= E + T | E - T | T
T ::= T * F | T / F | F
F ::= num | (E)

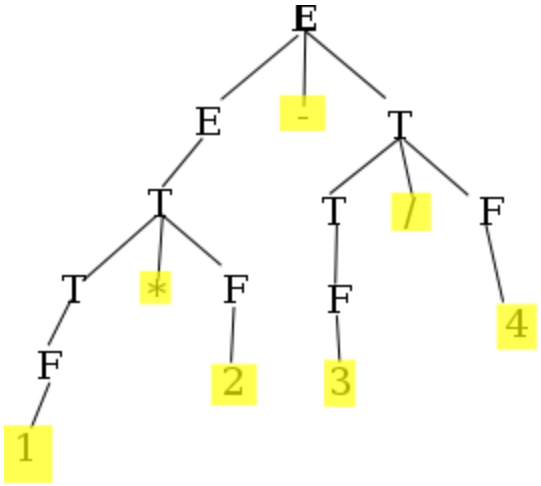
a. 1 * 2 - (3 / 4)



b. 1 - 2 + 3



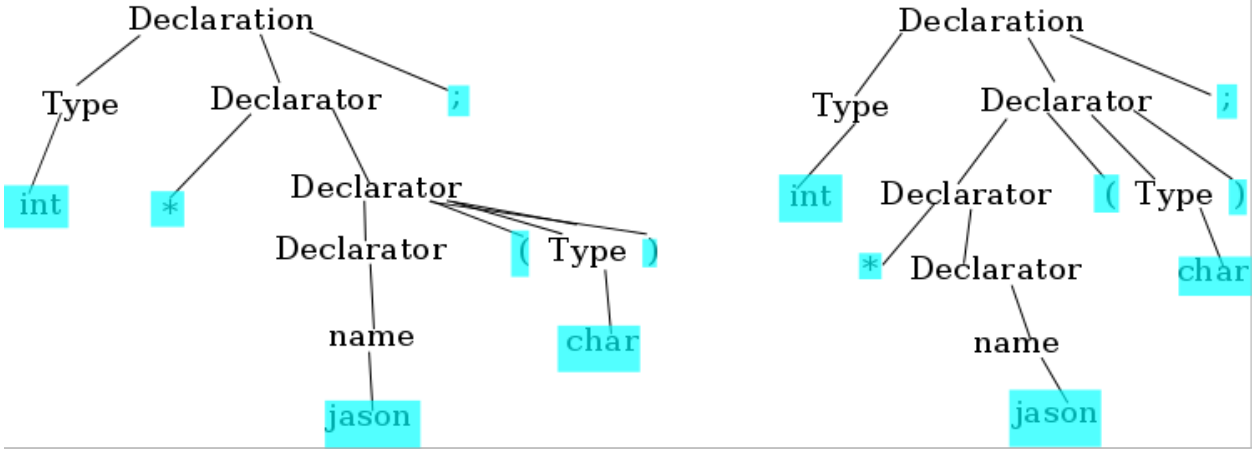
c. 1 * 2 - 3 / 4



3)

Here is a statement for which I will build two equivalent parse trees.

int * jason (char) ;



If there exists more than one non-isomorphic parse tree for a given string, then the grammar is ambiguous. ■

4)

Let L be the language over the alphabet {a,b,c}, which consists of at least three consecutive b's (For example, L includes the strings bbb and abcbbba, but not abbab). Write a BNF grammar for L.

L ::= bbb | Ebbb | bbbE | EbbbE
E ::= a | b | c | Ea | aE | Eb | bE | Ec | cE

5)
a)
<x + y, σ> → <σ(x) + σ(y), σ> → <2 + 3, σ> → <5, σ>

Explanation:
To get from: <x + y, σ> → <σ(x) + σ(y), σ>
We first use the rule: <a1 + a2,σ> → <a1' + a2,σ'> to evaluate
<x + y, σ> → <σ(x) + y, σ>. Next we use the rule: <x,σ> → <σ(x),σ>
to evaluate <σ(x) + y, σ> → <σ(x) + σ(y) , σ>

To get from: <σ(x) + σ(y), σ> → <2 + 3, σ>
because: σ(x) = 2 and σ(y) = 3

Lastly, <2 + 3, σ> → <5, σ>
from the rule: <n + m,σ> → <p,σ>

b) <x = x +3, σ> → <x = 1 +3, σ> → <x = 4, σ> → <4, put(σ,x,4)>

Explanation:
To get from: <x = x +3, σ> → <x = 1 +3, σ>
We use the rule: <a1 + a2,σ> → <a1' + a2,σ'> to evaluate
<x = x +3, σ> → <x = σ(x) +3, σ>, and because: σ(x)
<x = 1 +3, σ>

To get from: <x = 1 +3, σ> → <x = 4, σ>
We use the rule: <n + m,σ> → <p,σ>

Lastly, <x = 4, σ> → <4, put(σ,x,4)>

from the rule: $\langle x = n, \sigma \rangle \rightarrow \langle n, Put(\sigma,x,n) \rangle$

c) $\langle (x = 3) + x, \sigma \rangle \rightarrow \langle (3, Put(\sigma,x,3) + x, \sigma \rangle \rightarrow \langle (3) + 3, \sigma) \rangle \rightarrow \langle 6,\sigma \rangle$

First $\langle (x = 3) + x, \sigma \rangle \rightarrow \langle 3, Put(\sigma,x,3) + x, \sigma \rangle$
from the rule: $\langle x = n, \sigma \rangle \rightarrow \langle n, Put(\sigma,x,n) \rangle$

Lastly, $\langle (3) + 3, \sigma \rangle \rightarrow \langle 6,\sigma \rangle$
from the rule: $\langle n + m,\sigma \rangle \rightarrow \langle p,\sigma \rangle$

d) $\langle x = (x = x + 3) + (x = x + 5), \sigma \rangle \rightarrow \langle x = (x = \sigma(x) + 3) + (x = x + 5), \sigma \rangle$
Because: $\langle a1 + a2,\sigma \rangle \rightarrow \langle a1' + a2,\sigma' \rangle$
$\langle x = (x = \sigma(x) + 3) + (x = x + 5), \sigma \rangle \rightarrow \langle x = (x = 1 + 3) + (x = x + 5) \rangle$
Because: $\sigma(x) = 1$
$\langle x = (x = 1 + 3) + (x = x + 5) \rangle \rightarrow \langle x = (x = 4) + (x = x + 5) \rangle$
Because: $\langle n + m,\sigma \rangle \rightarrow \langle p,\sigma \rangle$
$\langle x = (x = 4) + (x = x + 5) \rangle \rightarrow \langle x = (x = 4) + (x = \sigma(x) + 5) \rangle$
Because: $\langle a1 + a2,\sigma \rangle \rightarrow \langle a1' + a2,\sigma' \rangle$
$\langle x = (x = 4) + (x = \sigma(x) + 5) \rangle \rightarrow \langle x = (x = 4) + (x = 4 + 5) \rangle$
Because: $x = 4$
$\langle x = (x = 4) + (x = 4 + 5) \rangle \rightarrow \langle x = (x = 4) + (x = 9) \rangle$
Because: $\langle n + m,\sigma \rangle \rightarrow \langle p,\sigma \rangle$
$\rightarrow \langle x = (x = 4) + (x = 9) \rangle \rightarrow \langle x = 13 \rangle$