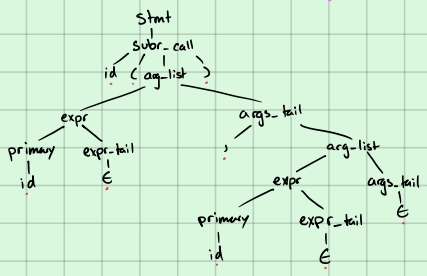
Colby Frison — CS 3323 || Homework 2 — 2/4/2025

2.13

Consider the following grammar:

```
stmt       ⟶ assignment
           ⟶ subr_call
assignment ⟶ id := expr
subr_call  ⟶ id ( arg_list )
expr       ⟶ primary expr_tail
expr_tail  ⟶ op expr
           ⟶ ϵ
primary    ⟶ id
           ⟶ subr_call
           ⟶ ( expr )
op         ⟶ + | - | * | /
arg_list   ⟶ expr args_tail
args_tail  ⟶ , arg_list
           ⟶ ϵ
```

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



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

Step	Derivation
1	sint → subr.call
2	subr.call → id (arg.list)
3	id (arg.list) → foo (arg.list)
4	foo (arg.list) → foo (expr args.tail)
5	foo (expr args.tail) → foo (primary expr.tail args.tail)
6	foo (primary expr.tail args.tail) → foo (id expr.tail args.tail)
7	foo (id expr.tail args.tail) → foo (a expr.tail args.tail)
8	foo (a expr.tail args.tail) → foo (a ε args.tail)
9	foo (a ε args.tail) → foo (a , arg.list)
10	foo (a , arg.list) → foo (a , expr args.tail)
11	foo (a , expr args.tail) → foo (a , primary expr.tail args.tail)
12	foo (a , primary expr.tail args.tail) → foo (a , id expr.tail args.tail)
13	foo (a , id expr.tail args.tail) → foo (a , b expr.tail args.tail)
14	foo (a , b expr.tail args.tail) → foo (a , b ε args.tail)
15	foo (a , b ε args.tail) → foo (a , b ε)

c) Prove that the grammar is not LL(1).

"primary → id | subr_call | (expr)" In this example primary can can be a variable or an variable or a function, causing ambiguity.

d) Modify the grammar so that it is LL(1).

Change: primary → id | subr_call | (expr)
subr_call → id (arg_list)

to: primary → id primary_tail | (expr)
primary_tail → (arg_list) | ϵ
→ := expr

also refactor arg_list:
arg_list → expr arg_list_tail
arg_list_tail → , expr arg_list_tail | ϵ

2.17

Extend the grammar of Figure 2.25 to include `if` statements and `while` loops, along the lines suggested by the following examples:

```
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
```

Your grammar should support the six standard comparison operations in conditions, with arbitrary expressions as operands. It should also allow an arbitrary number of statements in the body of an `if` or `while` statement.

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* ⟶ /

(4–6: *stmt*)

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

Stmt → id := expr
→ read id
→ write expr
→ if condition then stmt_list fi ϵ
→ while condition do stmt_list od ϵ
(end markers)

Condition → expr rel_op expr

rel_op → >
→ <
→ <=
→ >=
→ ==
→ !=

Created if & while stmt

Created conditions for if & while statements along with the operators for comparison

---

2.18

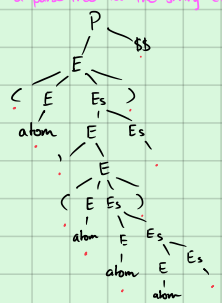
Consider the following LL(1) grammar for a simplified subset of Lisp:

```
P  ⟶ E $$
E  ⟶ atom
   ⟶ ' E
   ⟶ ( E Es )
Es ⟶ E Es
   ⟶
```

(a) What is FIRST(*Es*)? FOLLOW(*E*)? PREDICT(*Es* ⟶ ϵ)?
(b) Give a parse tree for the string `(cdr '(a b c)) $$`.
(c) Show the left-most derivation of `(cdr '(a b c)) $$`.
(d) Show a trace, in the style of Figure 2.21, of a table-driven top-down parse of this same input.
(e) Now consider a recursive descent parser running on the same input. At the point where the quote token (`'`) is matched, which recursive descent routines will be active (i.e., what routines will have a frame on the parser's run-time stack)?

a)
- First(Es) = { atom, ', (}
- Follow(E) = { $$,) }
- Predict(Es → ϵ) = {) }

b) Give a parse tree for the string (cdr '(a b c)) $$



c) left most derivation of (cdr '(abc)) $$

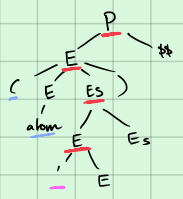
P → E $$
→ (E Es) $$
→ (cdr Es) $$
→ (cdr E Es) $$ ϵ
→ (cdr ' E Es) $$
→ (cdr ' (E Es)) $$
→ (cdr ' (a Es)) $$
→ (cdr ' (a E Es)) $$
→ (cdr ' (a b Es)) $$
→ (cdr ' (a b E Es)) $$
→ (cdr ' (a b c Es)) $$
→ (cdr ' (a b c)) $$

d) Show a trace of a table-driven top-down parse

Step	Stack	Input	Action
1	P	(cdr '(a b c))$$	start program
2	E $$	(cdr '(a b c))$$	predict P → E $$
3	(E Es) $$	cdr '(a b c) $$	predict E → (E Es)
4	E Es $$	cdr '(a b c) $$	match ()
5	Atom Es $$	cdr '(a b c) $$	predict E → atom
6	Es $$	'(a b c) $$	match atom(cdr)
7	E Es $$	'(a b c) $$	predict E → E Es
8	'E Es $$	'(a b c) $$	predict E → 'E
9	E Es $$	(a b c) $$	match '
10	(E Es) Es $$	(a b c) $$	predict E → 'E
11	E Es Es $$	a b c $$	match ()
12	atom Es Es $$	a b c $$	predict E → atom
13	a Es Es $$	a b c $$	predict E → a
14	Es Es $$	b c $$	match a
15	E Es Es $$	b c $$	predict E → E Es
16	atom Es Es $$	b c $$	predict E → atom
17	b Es Es $$	b c $$	predict E → b
18	Es Es $$	b c $$	match b
19	E Es Es $$	c $$	predict E → E Es
20	atom Es Es $$	c $$	predict E → atom
21	c Es Es $$	c $$	predict E → c
22	Es Es $$	$$	match c
23	Es $$	$$	predict Es → ϵ
23	$$	$$	predict Es → ϵ
25			predict $$ → ϵ

e) When the token ' is matched the following routines are active

P E Es & another E
represents terminals
represents active routines when ' is matched.



2.40) What do you think of python's use of indentation to delimit control constructs? Would you expect this convention to make programs construction easier or harder

python uses indentation to format its code instead of { } or begin & end statements

Ex. python:

```
x = input()
if x > 1 :
    print("hello")
print('world')
```

x is 0 → world
x is 3 → hello world

c++

```
int x = 0;
cin >> x;
if (x > 1) {
    std :: cout << "hello";
}
std:: cout << "world";
```

you know what I mean

this approach has some advantages & disadvantages:

Advantages:
- Fewer tokens / less clutter → without the extra keywords the scanner can be slightly better
- Readability - with the simpler syntax the code becomes far more consistent

Downsides:
- It can be hard to debug when the error is occuring in tokens that aren't visible (whitespace) as a misalignment can cause errors in code, but often can be hard to diagnose

So overall, python's approach of using whitespace as delimeters has numerous advantages: Readability, less tokens, consistency, but it also has some potential deal breaking disadvantages: white space sensitivity