Exercise walk-through

11.5 Parsing

University Computer Science

Parsing: table plus stack

1. **Trace** E -> T Ep, Ep -> + T Ep | empty, T -> id: id + id parses in eight stack snapshots, suite-asserted.
2. **Conflicts** A cell with two productions means one lookahead cannot choose - the grammar is not LL(1).
3. **Tail** E -> E+T loops the stack; E -> T Ep makes the loop a tail - associativity moves to the AST build.

Parse table and stack trace

the LL(1) table

nonterm	lookahead	production
E	id	T Ep
Ep	+	+ T Ep
Ep	$	empty

FIRST/FOLLOW fill the rows;
conflicts reject the grammar

left recursion E -> E+T loops the stack; the tail trick fixes it:
E -> T Ep, Ep -> + T Ep | empty

the stack trace of id + id

```
1. E
2. T Ep
3. id Ep
4. Ep
5. + T Ep
6. T Ep
7. id Ep
8. Ep
9. (empty)
```

eight moves, one tree, one symbol of lookahead

The LL(1) table and the eight-snapshot stack trace of id + id.

Exercise 2.1

2.1 Give FIRST(E), FIRST(E') and FIRST(T') for the grammar in the box. [3]

Grammar

```
E → T E'
E' → + T E' | (empty)
T → F T'
T' → * F T' | (empty)
F → ( E ) | id
```

Exercise 2.1

2.1 Give FIRST(E), FIRST(E') and FIRST(T') for the grammar in the box. [3]

Grammar

```
E → T E'
E' → + T E' | (empty)
T → F T'
T' → * F T' | (empty)
F → ( E ) | id
```

2.1

FIRST(E) = {(, id}

FIRST(E') = {+, ε}

FIRST(T') = {*, ε}

Exercises 2.2·2.3

2.2 Give FOLLOW(E') and FOLLOW(T'). [2]

Grammar

```
E → T E'
E' → + T E' | (empty)
T → F T'
T' → * F T' | (empty)
F → ( E ) | id
end marker: $
```

2.3 Remove the left recursion from the rules shown. [2]

Grammar

```
E → E + T | T
T → T * F | F
```

Exercises 2.2·2.3

2.2 Give FOLLOW(E') and FOLLOW(T'). [2]

```
Grammar
E → T E'
E' → + T E' | (empty)
T → F T'
T' → * F T' | (empty)
F → ( E ) | id
end marker: $
```

2.2

FOLLOW(E') = FOLLOW(E) = {), $}

FOLLOW(T') = FOLLOW(T) = {+,), $}

2.3 Remove the left recursion from the rules shown. [2]

```
Grammar
E → E + T | T
T → T * F | F
```

2.3

E → T E', E' → + T E' | ε

T → F T', T' → * F T' | ε

Exercise 2.4

2.4 Trace a shift-reduce parse of `id * id` with the grammar shown. [3]

Grammar

```
E → E + T | T
T → T * F | F
F → id
```

Exercise 2.4

2.4 Trace a shift-reduce parse of `id * id` with the grammar shown. [3]

Grammar

```
E → E + T | T
T → T * F | F
F → id
```

2.4

shift `id`, reduce F, reduce T

shift `*`, shift `id`, reduce F

reduce T → T * F, reduce E → T, accept

Exercise 3.1 (a)

3.1 Use the LL grammar shown, with tokens `id + * ( ) $`.

(a) Fill the LL(1) table. [4]

	id	+	*	(	)	$
E						
E’						
T						
T’						
F						

Exercise 3.1 (a)

3.1 Use the LL grammar shown, with tokens `id + * ( ) $`.

(a) Fill the LL(1) table. [4]

	id	+	*	(	)	$
E						
E’						
T						
T’						
F						

(a) E: `id`, `(` → T E’; E’: `+` → + T E’; `)`, `$` → ε; T: `id`, `(` → F T’; T’: `*` → * F T’; `+`, `)`, `$` → ε; F: `id` → id; `(` → (E)

Exercise 3.1 (b)

3.1 Use the LL grammar shown, with tokens `id + * ( ) $`.

(b) What shows that the grammar is LL(1)? [1]

Exercise 3.1 (b)

3.1 Use the LL grammar shown, with tokens `id + * ( ) $`.

(b) What shows that the grammar is LL(1)? [1]

(b) no cell has two rules

Exercise 3.1 (c)

3.1 Use the LL grammar shown, with tokens `id + * ( ) $`.

(c) Write a recursive-descent function for E'. [2]

Exercise 3.1 (c)

3.1 Use the LL grammar shown, with tokens `id + * ( ) $`.

(c) Write a recursive-descent function for E'. [2]

(c) `def Eprime(): if next == '+': match('+'); T(); Eprime(); else: return` (ε) which is right when next is `)` or `$`

Exercise 3.2 (a)

3.2 A parser for `E → E + E | E * E | id` uses precedence rules, as shown.

(a) The stack holds `E + E` and the next token is `*`. Shift or reduce? Why? [2]



Exercise 3.2 (a)

3.2 A parser for `E → E + E | E * E | id` uses precedence rules, as shown.

(a) The stack holds `E + E` and the next token is `*`. Shift or reduce? Why? [2]



(a) shift: `*` binds more tightly, so `E * E` must be built first

Exercise 3.2 (b)

3.2 A parser for `E → E + E | E * E | id` uses precedence rules, as shown.

(b) The stack holds `E * E` and the next token is `+`. Shift or reduce? [1]

Exercise 3.2 (b)

3.2 A parser for `E → E + E | E * E | id` uses precedence rules, as shown.

(b) The stack holds `E * E` and the next token is `+`. Shift or reduce? [1]

(b) reduce `E * E` first

Exercise 3.2 (c)

3.2 A parser for `E → E + E | E * E | id` uses precedence rules, as shown.

(c) How do tools such as yacc or bison settle these conflicts? [2]

Exercise 3.2 (c)

3.2 A parser for `E → E + E | E * E | id` uses precedence rules, as shown.

(c) How do tools such as yacc or bison settle these conflicts? [2]

(c) the grammar file declares precedence and associativity, e.g. `%left '+'` then `%left '*'`; the tool uses them to pick shift or reduce in each conflict

Exercise 3.3 (a)

3.3 An online judge parses beginners' code. A parser that stops at the first error gives one message; a parser with **panic-mode recovery** tries to go on. A submission is shown.

(a) Describe panic-mode recovery. [2]

```
submission
1 x = (3 + ;
2 y = 2;
```

Exercise 3.3 (a)

3.3 An online judge parses beginners' code. A parser that stops at the first error gives one message; a parser with **panic-mode recovery** tries to go on. A submission is shown.

(a) Describe panic-mode recovery. [2]

submission

```
1 x = (3 + ;
2 y = 2;
```

(a) on an error, skip tokens until a synchronising token such as `;` or `}`, pop the parser state back to a matching rule, and continue

Exercise 3.3 (b)

3.3 An online judge parses beginners' code. A parser that stops at the first error gives one message; a parser with **panic-mode recovery** tries to go on. A submission is shown.

(b) Say what each kind of parser reports for this code. [2]

Exercise 3.3 (b)

3.3 An online judge parses beginners' code. A parser that stops at the first error gives one message; a parser with **panic-mode recovery** tries to go on. A submission is shown.

(b) Say what each kind of parser reports for this code. [2]

(b) stop-at-first: "line 1: expected a number or `(` before `;`"; panic mode reports the same, skips to `;`, and parses line 2 without errors

Exercise 3.3 (c)

3.3 An online judge parses beginners' code. A parser that stops at the first error gives one message; a parser with **panic-mode recovery** tries to go on. A submission is shown.

(c) Give one risk of going on after an error. [2]

Exercise 3.3 (c)

3.3 An online judge parses beginners' code. A parser that stops at the first error gives one message; a parser with **panic-mode recovery** tries to go on. A submission is shown.

(c) Give one risk of going on after an error. [2]

(c) later messages may be caused by the first error, not by real mistakes, which confuses beginners

Exercise 3.3 (d)

3.3 An online judge parses beginners' code. A parser that stops at the first error gives one message; a parser with **panic-mode recovery** tries to go on. A submission is shown.

(d) Recommend an error policy for beginners. [2]

Exercise 3.3 (d)

3.3 An online judge parses beginners' code. A parser that stops at the first error gives one message; a parser with **panic-mode recovery** tries to go on. A submission is shown.

(d) Recommend an error policy for beginners. [2]

(d) show the first error clearly, with the line, column and the expected token; after recovery show at most two more, and mark them "may be caused by the first"