Homework 5

Name: Reagan McFarland
NETID: rpm141
Date: April 18th, 2022

Problem 1 - LR(1) Parsing

Given the grammar below

```
1 |     S' ::= S
2 |     S  ::= L = R
3 |          | R
4 |     L  ::= *R
5 |          | id
6 |     R  ::= L
```

1. Compute the canonical collection of sets of LR(1) items

- State 0

```
[S' -> .S, $]
[S -> .L=R, $]
[S -> .R, $]
[L -> .*R, =/$]
[L -> .id, =/$]
[R -> .L, $]

On S goto 1
On L goto 2
On R goto 3
On * goto 4
On id goto 5
```

- State 1

```
[S' -> S., $]
```

- State 2

```
[S -> L.=R, $]
[R -> L., $]

On = goto 6
```

- State 3

```
[S -> R., $]
```

- State 4

```
[L -> *.R, =/$]
[R -> .L, =/$]
[L -> .*R, =/$]
[L -> .id, =/$]

On R goto 7
On L goto 8
On * goto 4
On id goto 5
```

- State 5

```
[L -> id., =/$]
```

- State 6

```
[S -> L=.R, $]
[R -> .L, $],
[L -> .*R, $]
[L -> .id, $]

On R goto 9
On L goto 10
On * goto 11
On id goto 12
```

- State 7

```
[L -> *R., =/$]
```

- State 8

```
[R-> L., =/*]
```

- State 9

```
[S -> L = R., $]
```

- State 10

```
[R -> L., $]
```

- State 11

```
[L -> *.R, $]
[R -> .L, $]
[L -> .*R, $]
[L -> .id, $]

On R goto 13
On L goto 10
On * goto 11
On id goto 12
```

- State 12

```
[L -> id., $]
```

- State 13

```
[L -> *R., $]
```

Construct the LR(1) parse table (ACTION & GOTO)

Action table:

State	=	*	id	$
0		4	5	
1				accept
2	6			R → L
3				S → L = R
4		4	5	
5	L → id			L → id
6		11	12	
7	L → *R			L → *R
8	R → L			R → L
9				S → L = R
10				R → L
11		11	12	
12				L → id
13				L → *R

GOTO table:

State	S'	S	L	R
0		1	2	3
1				
2				
3				
4			8	7
5				
6			10	9
7				
8				
9				

State	S'	S	L	R
10				
11			10	13

3. Is the grammar LR(1) or not? Justify your answer

This grammar **IS** LR(1) because we have no conflicts in our ACTION and GOTO tables. We always know exactly which step to take depending on the input we parse and the state we are at by looking at most 1 symbol look ahead.

4. If the grammar is LR(1), show the behavior LR(1) parser on input `*id = id`, i.e., show stack content, current input, and selected action for each move of the machine

```
[s0], .*id = id $, next input + goto s4
[s0, *, s4], *.id = id $, next input + goto s5
[s0, *, s4, id, s5], *id. = id $, reduce L -> id
[s0, *, s4, L], *id. = id $, goto s8
[s0, *, s4, L, s8], *id. = id $, reduce R -> L
[s0, *, s4, R], *id. = id $, goto s7
[s0, *, s4, R, s7], *id. = id $, reduce L -> * R
[s0, L], *id. = id $, goto s2
[s0, L, s2], *id. = id $, next input + goto s6
[s0, L, s2, =, s6], *id=. id $, next input + goto s12
[s0, L, s2, =, s6, id, s12], *id=id. $, reduce L -> id
[s0, L, s2, =, s6, L], *id=id. $, goto s10
[s0, L, s2, =, s6, L, s10], *id=id. $, reduce R -> L
[s0, L, s2, =, 6, R], *id=id. $, goto s9
[s0, L, s2, =, 6, R, 9], *id=id. $, reduce S -> L = R
[s0, S], *id=id. $, goto s1
[s0, S, s1], $, accept
```

Problem 2 - LR(0)

Show that the above grammar (Problem 1) is not LR(0). Note that it is sufficient to show one state where there is a conflict (Hint: you don't need to enumerate ***all*** states)

There is a conflict at State 0 if there is no lookahead:

```
[S -> L.=R, $]
[R -> L., $]
```

Without at least one symbol of look ahead, the parser generator does not know to reduce R → L. unless it knows that there is not a = symbol directly after. This is not possible to check with LR(0) since there is no look ahead, while it is possible in LR(1) since there is one symbol of look ahead.

Problem 3 - Type Systems

Assume a type system with the following inference rules

- Rule 1: $E \vdash e1 : integer\ E \vdash e2 : integer \Rightarrow E \vdash (e1 + e2) : integer$
- Rule 2: $E \vdash e : \alpha \Rightarrow E \vdash (\&e) : pointer(\alpha)$
- Rule 3: $E \vdash e : pointer(\alpha) \Rightarrow E \vdash *e : \alpha$

Assuming that variable **a** and constant **3** are of type integer, and variable **b** is of type boolean. Use the inference rules to determine the types of the following expressions. Note: if a proof does not exist, the type system reports a type error

1. `&a`

Initially, *E = { a : integer }*

- Using rule 2, we can deduce *E ⊢ a : integer ⇒ E ⊢ &a : pointer(integer)*

Therefore, the type of `&a` is `pointer(integer)`

2. `&b`

Initially, *E = { b : boolean }*

- Using rule 2, we can deduce *E ⊢ b : boolean ⇒ E ⊢ &b : pointer(boolean)*

Therefore, the type of `&b` is `pointer(boolean)`

3. `(&a + 5)`

Initially, *E = { a : integer, 5 : integer }*

- Using rule 2, we can deduce *E ⊢ a : integer ⇒ E ⊢ &a : pointer(integer)*
- We add this to our type environment, leaving us with *E = { a : integer, &a : pointer(integer), 5 : integer }*
- The expression is attempting to do an add, but the only inference rule we have for addition requires that 2 integers are on either side of the + operator, but we have a *integer* and *pointer(integer)*, so we cannot do anything.

Therefore, the type of `(&a + 5)` is `type error`

4. `*a`

Initially, *E = { a : integer }*

- Rule 3 is what we will have to use for dereference, but it requires that *E ⊢ e : pointer(α)* in order to deduce the type of `*a`. But, our α in this case is of type *integer* as states in our type environment, which means this is not allowed.

Therefore, the type of `*a` is `type error`

5. `&3`

Initially, *E = { 3 : integer }*

- Using rule 2, we can deduce *E ⊢ 3 : integer ⇒ &3 : pointer(integer)*

Therefore, the type of `&3` is `pointer(integer)`

6. `*(a + b)`

Initially, *E = { a : integer, b : boolean }*

- Using order of operations, we need to deduce the type of `a + b` before we can deduce the type of the dereference operation. However, our rule 1 requires that *e1* and *e2* must be of type integer, while we have *E ⊢ a : integer E ⊢ b : boolean*, which means we cannot deduce the type of this.

Therefore, the type of `*(a + b)` is `type error`

7. `&&a`

Initially, $E = \{ a : integer \}$

- Using rule 2, we can deduce $E \vdash a : integer \Rightarrow \&a : pointer(integer)$
- Using rule 2, we can deduce $E \vdash \&a : integer \Rightarrow \&\&a : pointer(pointer(integer))$

Unlike rule 3, rule 2 has no constraints on the type α in the first part of the inference rule. Therefore, the type of `&&a` is `pointer(pointer(integer))`