Overall Compiler Structure – Control Structures - Stage 2

Stage2 adds three new control structures to allow conditional and repeated execution of Pascallite source code.

1. **if-then-else** statement
2. **while-do** statement
3. **repeat-until** statement

The first statement, which permits conditional execution of code, actually appears in two forms, with and without an **else** clause, while the second and third statements, employed for looping, have only one form. With their inclusion, Pascallite will almost become a viable language for simple but nontrivial programs.

Several new concepts will be introduced as well. First, all of these new statements are *compound* statements. As such, you must be able to properly detect the end of each statement which may be arbitrarily far from its beginning. This problem is further compounded by the fact that these statements may be nested inside one another arbitrarily deep.

Pascallite Language Definition - Stage 2

1.	Seven new keywords are added: **if, then, else, repeat, while, do, until** All of these keywords identify clauses of the three control structures augmenting Pascallite. No new tokens are added to Pascallite here.

Pascallite Grammar Stage 2

The context-free grammar for stage2 features along with the selection set of each alternative production is given below. A production is not repeated unless it has somehow changed to accommodate the new language features. *In several cases, selection sets of productions not listed must be augmented*. That task is left to you.

Revised Productions:

```
1. BEGIN_END_STMT  →  'begin' EXEC_STMTS 'end'                          {'begin'}
                      (
                         '.'                                              {'.'}
                         | ';'                                            {';'}
                      )

2. EXEC_STMT       →  ASSIGN_STMT                                  {NON_KEY_ID}
                   →  READ_STMT                                        {'read'}
                   →  WRITE_STMT                                      {'write'}
                   →  IF_STMT                                            {'if'}
                   →  WHILE_STMT                                      {'while'}
                   →  REPEAT_STMT                                    {'repeat'}
                   →  NULL_STMT                                           {';'}
                   →  BEGIN_END_STMT                                  {'begin'}
```

New Productions:

```
3. IF_STMT         →  'if'  EXPRESS  'then'  EXEC_STMT  ELSE_PT          {'if'}

4. ELSE_PT         →  'else'  EXEC_STMT                                {'else'}
                   →   ε                    {'end',NON_KEY_ID,';','until','begin',
                                             'while','if','repeat','read','write'}

5. WHILE_STMT      →  'while'  EXPRESS  'do'  EXEC_STMT               {'while'}

6. REPEAT_STMT     →  'repeat'  EXEC_STMTS  'until'  EXPRESS  ';'    {'repeat'}

7. NULL_STMT       →  ';'                                                 {';'}
```

The only context-sensitive constraint on programs not expressed by the grammar is that the value of the expression in an **if**, **while**, or **repeat** statement must be of type *boolean*.

if statement – both forms

The semantics of the **if** statement is shown below. You should be familiar with this nearly universal statement.

`'if'`	
`express`	evaluate `express`
`'then'`	branch to L_1 if `express` is false
`EXEC_STMT`	execute `EXEC_STMT`
	L_1:

(a) without **else** clause

`'if'`	
`express`	evaluate `express`
`'then'`	branch to L_1 if `express` is false
`EXEC_STMT`	execute `EXEC_STMT`
`'else'`	branch to L_2
	L_1:
`EXEC_STMT`	execute `EXEC_STMT`
	L_2:

(b) with **else** clause

The **if** statement presents an unusual situation. If you carefully examine the `ELSE_PT` production, you will note that part of one selection set is missing. The productions should actually be:

```
ELSE_PT    →  'else'  EXEC_STMT                                 {'else'}
           →   ε                     {'end',NON_KEY_ID,';','until','begin',
                                      'while','if','repeat','read','write',
                                                                    'else'}
```

The `'else'` has been omitted from the selection set of production 4. Note that with the full selection set listed, a key conflict results; hence, the grammar is not strictly LL(1), as promised. Here is an explanation of why `'else'` has been left out of production 4's selection set. The **if** statement suffers from a classic ambiguity, called the "dangling **else**" problem, which manifests itself in the following code segment:

```
if  p  then  if  q  then  r  else  s                                   (i)
```

Both *p* and *q* are *boolean* predicates, while *r* and *s* are statements. Which **if** does the "dangling" **else** match? The question becomes clearer when you consider two alternative reformattings of (i) above.

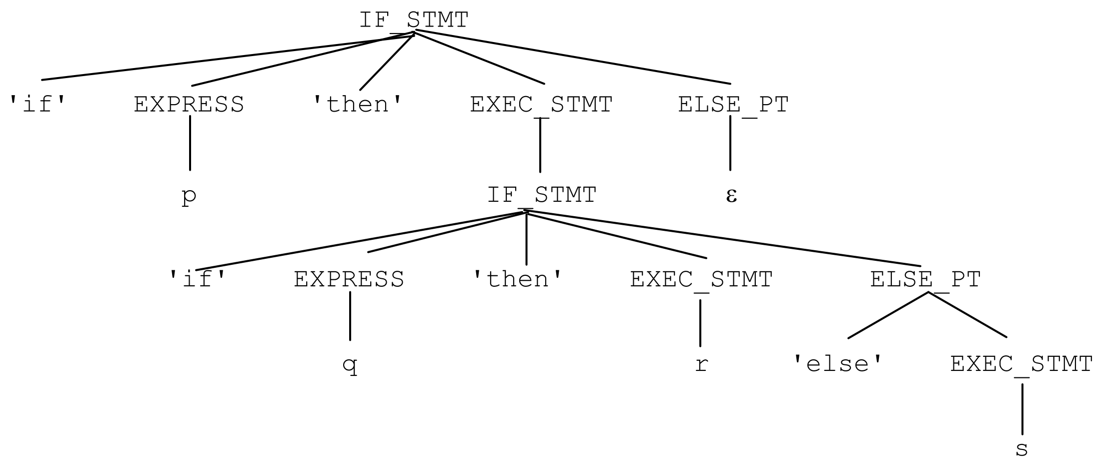
```
if  p  then                                                            (ii)
  if  q  then  r
  else  s
```

and

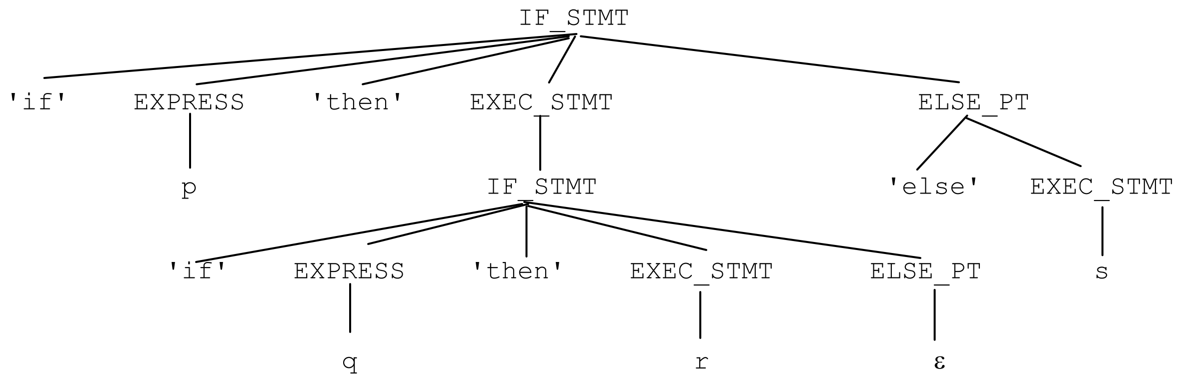
```
if  p  then                                                           (iii)
  if  q  then  r
else  s
```

The first binds (in appearance) the **else** to the inner and closest **if**, while the second binds the **else** clause to the outer and farthest **if**. The parse trees for these two bindings are shown below. The grammar is ambiguous, with these two trees expressing the nature of the ambiguity.

The ambiguity in the *language* must be resolved first. This will dictate how to resolve the ambiguity in the grammar. Pascallite, like most languages, opts for the interpretation shown in the parse tree (ii). No known language opts for the parse tree (iii). The grammar must allow only for this interpretation, but unfortunately, there is no unambiguous grammar for Pascallite which has this optional **else** clause. You should rely upon an alternative which is occasionally taken by compiler writers—to force the choice of one production over another by deleting elements from selection sets, production 4 being an embodiment of this strategy. A willingness to resolve selection set conflicts in this manner significantly increases the class of grammars to which the LL(1) parsing algorithm is applicable, even though these grammars are not strictly LL(1).



parse tree (ii)



parse tree (iii)

while statement

The **while** statement provides for conditional repetitive execution of a single (possibly compound) statement. When the expression following `'while'` is evaluated, if it is false, control resumes following the end of the **while** statement. If it is true, the statement following `'do'` is executed. When its execution is complete, control returns back to the point where the expression is again evaluated, and the entire process is repeated until the expression eventually turns false (or else the program is in an infinite loop).

`'while'`	L_1:
`express`	evaluate `express`
`'do'`	branch to L_2 if `express` is false
`EXEC_STMT`	execute `EXEC_STMT`
	branch to L_1
	L_2:

repeat statement

The **repeat** statement is quite similar to the **while** statement in that both are used for looping, but with two major differences: the predicate is evaluated *after* the loop body has been executed, not *before*, as in a **while** statement. Consequently, the loop body (statements between `'repeat'` and `'until'`) is always executed at least once, even if the predicate is initially false. For a **while** statement, this is not true. If the predicate is initially false, the loop body (statement following `'do'`) is skipped entirely. The second difference is that the **while** loop is executed repeatedly until the predicate is false, while the **repeat** statement is executed repeatedly until the predicate is true.

`'repeat'`	L_1:
`EXEC_STMTS`	execute `EXEC_STMTS`
`'until'`	
`express`	evaluate `express`
`';'`	branch to L_1 if `express` is false

Pascallite Translation Grammar Stage 2

Revised Productions:

```
1.  BEGIN_END_STMT   →  'begin' EXEC_STMTS 'end'
                        ('.'x | ';'x)   Code('end',x)

2.  EXEC_STMT        →  ASSIGN_STMT
                     →  READ_STMT
                     →  WRITE_STMT
                     →  IF_STMT
                     →  WHILE_STMT
                     →  REPEAT_STMT
                     →  NULL_STMT
                     →  BEGIN_END_STMT
```

New Productions:

```
3.  IF_STMT          →  'if'  EXPRESS  'then'
                        Code('then',PopOperand())
                        EXEC_STMT  ELSE_PT

4.  ELSE_PT          →  'else'  Code('else',PopOperand())
                        EXEC_STMT  Code('post_if',PopOperand())
                     →   ε         Code('post_if',PopOperand())

5.  WHILE_STMT       →  'while'  Code('while')  EXPRESS  'do'
                        Code('do',PopOperand())  EXEC_STMT
                        Code('post_while',PopOperand(),PopOperand())

6.  REPEAT_STMT      →  'repeat'  Code('repeat')  EXEC_STMTS
                        'until'  EXPRESS  Code('until',PopOperand(),PopOperand())
                        ';'

7.  NULL_STMT        →  ';'
```