Phase I Report

**Grammar modifications**

The grammar provided initially was not LL(1), as it suffered from left-recursion in multiple rules. Our grammar changed frequently over the course of Phase I, but left-factoring to remove initial recursion occurred relatively early on.

The initial grammar provided is included in the specification PDF – however, the specific rules which suffered from direct left recursion were as follows:

EXPR\* -> CONST | VALUE | EXPR BINARY\_OPERATOR EXPR | ‘(‘ EXPR ‘)’

INDEX\_EXPR -> INTLIT | ID | INDEX\_EXPR INDEX\_OPER INDEX\_EXPR

These were easily left-factored:

EXPR -> (CONST | VALUE | '(' EXPR ')') (BINARY\_OPERATOR EXPR)\*  
INDEX\_EXPR -> (INTLIT | ID) (INDEX\_OPER INDEX\_EXPR)\*

The major problems we had to deal with were mainly confusion over the responsibility in the context of our compiler of lexer versus parser grammars (as can be seen above – all rules were lex rules initially), indirect left-recursion, and input collisions on parser rules.

**Lex rules**

Our grammar incorporates all of the lex rules from the provided specification:

NEQ -> '<>'

LESSER -> '<'

LESSEREQ -> '<='

GREATER -> '>'

GREATEREQ -> '>='

AND -> '&'

OR -> '|'

ASSIGN -> ':='

COMMA -> ','

COLON -> ':'

SEMI -> ';'

LPAREN -> '('

RPAREN -> ')'

LBRACK -> '['

RBRACK -> ']'

PLUS -> '+'

MINUS -> '-'

MULT -> '\*'

DIV -> '/'

EQ -> '='

ID -> ('a'..'z'|'A'..'Z'|'\_') ('a'..'z'|'A'..'Z'|'0'..'9'|'\_')\*

INTLIT -> MINUS? '0'..'9'+

FIXEDPTLIT -> INTLIT+ '.' ('0'..'9')\* | '.' ('0'..'9')+

\* For simplicity, grammar rules specified will omit semicolons and be of the form:   
RULE -> ANTLR\_EXPRESSION

In addition, our final lex grammar also contained a few modifications and additions outside the scope of the specification:

1. Keywords are all tokens in and of themselves. This prevents collision between the ID rule and keywords of Tiger, as the keywords are defined earlier in the grammar and have precedence.

THEN\_KEY -> 'then'ENDIF\_KEY -> 'endif'ELSE\_KEY -> 'else'WHILE\_KEY -> 'while'ENDDO\_KEY -> 'enddo'FOR\_KEY -> 'for'ID\_KEY -> 'id'TO\_KEY -> 'to'DO\_KEY -> 'do'BREAK\_KEY -> 'break'RETURN\_KEY -> 'return'

FUNCTION\_KEY -> 'function'BEGIN\_KEY -> 'begin'END\_KEY -> 'end'VOID\_KEY -> 'void'MAIN\_KEY -> 'main'TYPE\_KEY -> 'type'ARRAY\_KEY -> 'array'OF\_KEY -> 'of'INT\_KEY -> 'int'FIXEDPT\_KEY -> 'fixedpt'VAR\_KEY -> 'var'  
IF\_KEY -> 'if'

1. Comments and formatting escape character sequences are treated as ignored tokens. This prevents their use in other tokens, solving many issues with stray whitespace, tab characters, or return characters treated as part of parse rules.

COMMENT -> '/\*' (options {greedy=false;}:.)\* '\*/' {$channel=HIDDEN;}  
  
TAB -> '\t' {$channel=HIDDEN;}  
NEWLINE -> '\n' {$channel=HIDDEN;}  
CARRAGE\_RET -> '\r' {$channel=HIDDEN;}  
WHITESPACE -> ' ' {$channel=HIDDEN;}

1. All parse rules were edited to make direct reference to lex rules if necessary. For example:   
     
   type -> base\_type | ‘array[‘ INTLIT ‘]’ (‘[‘ INTLIT ‘]’)? ‘of’ base\_type  
     
   type -> base\_type | ARRAY\_KEY LBRACK INTLIT RBRACK (LBRACK INTLIT RBRACK)? OF\_KEY base\_type

**Parse rules**

The parse grammar was heavily modified from the initially provided one for simplicity, terseness, and to solve input collision issues. The full grammar changes are too numerous to list in their entirety here, but below is a list of general strategies which were used:

* **Simplification.** Given a general rule:  
    
  A -> BA  
    
  It is possible to simplify the given expression as the following with no loss of meaning:  
    
  A -> B+  
    
  In addition, if there is a rule B -> λ, then the expression can be further simplified as follows:  
    
  A -> B\*  
    
  Two examples from our grammar follow.  
    
  stat\_seq -> stat stat\_seq\*  
  stat\_seq -> stat+  
    
  type\_declaration\_list -> ( | type\_declaration type\_declaration\_list)  
  type\_declaration\_list -> type\_declaration\*
* **Input collision aversion.** Rules such of the following cause an input collision in an LL(1) grammar:  
    
  A -> (CB | CDE)  
    
  This can be solved with left-factoring:  
    
  A -> C(B | DE)  
    
  Again, two examples from our grammar:  
    
  id\_list -> (ID | ID COMMA id\_list)  
  id\_list -> ID (COMMA id\_list)?  
    
  value\_tail -> LBRACK index\_expr RBRACK  
   | LBRACK index\_expr RBRACK LBRACK index\_expr RBRACK  
   |   
    
  value\_tail -> (LBRACK index\_expr RBRACK (LBRACK index\_expr RBRACK)?)?