Phase I Report

**Left recursion**

The grammar provided initially was not LL(1), as it suffered from left recursion and collision 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)?)?
* **Function call support.** Our grammar expands support for function calls, which was not included in the original grammar specification:  
    
  stat -> ID ((value\_tail ASSIGN expr\_list) | (func\_call\_tail)) SEMI  
  func\_call\_tail -> LPAREN func\_param\_list RPAREN  
  func\_param\_list -> (expr (COMMA expr)\*)?  
    
  expr -> (constval | ID (value\_tail | func\_call\_tail) | LPAREN expr RPAREN) (binop\_p0 expr)?  
    
  This adds support for (1) standard void function calls, as well as (2) function parameter chaining and (3) the use of a function return value as an expression:  
    
  1. printi(5);  
  2. printi(fact(5))  
  3. var result : int := (fact(1) + 2);
* **Operator precedence.** This was implemented primarily in the expr parse rule, and works based on the fact that our parser chooses a parse rule based on the longest match. Because of this inherent property, we implemented an operator parsing chain with the highest precedence operators at the bottom – which would naturally be matched first.  
    
  expr -> (constval | ID (value\_tail | func\_call\_tail) | LPAREN expr RPAREN) (binop\_p0 expr)

binop\_p0: (AND | OR | binop\_p1) // Lowest precedence

binop\_p1: (EQ | NEQ | LESSER | GREATER | LESSEREQ | GREATEREQ | binop\_p2)

binop\_p2: (MINUS | PLUS | binop\_p3)

binop\_p3: (MULT | DIV) // Highest precedence

* **Major design decisions.** Certain problems inherent to the structure of the grammar forced us to change certain aspects of the parse rules to avoid collision. A few rules needed to be expanded, factored, and removed in their parent calling rule in order to avoid collision with other rules:  
    
  Removal of opt\_prefix:  
    
  stat -> opt\_prefix ID LPAREN expr\_list RPAREN SEMI | func\_call  
  opt\_prefix -> value ASSIGN  
  value -> ID value\_tail  
  func\_call -> ID LPAREN func\_param\_list RPAREN  
    
  stat -> ID ((value\_tail ASSIGN expr\_list) | (func\_call\_tail)) SEMI  
  func\_call\_tail -> LPAREN func\_param\_list RPAREN  
  ~~opt\_prefix ->~~   
    
  Removal of main\_function rule to support void functions:  
    
  tiger\_program -> type\_declaration\_list funct\_declaration\_list main\_function  
  funct\_declaration\_list -> funct\_declaration\*

funct\_declaration -> ret\_type FUNCTION\_KEY ID LPAREN PARAM\_LIST RPAREN BEGIN\_KEY block\_list END\_KEY SEMI  
ret\_type -> VOID\_KEY | type\_id

main\_function -> VOID\_KEY MAIN\_KEY LPAREN RPAREN BEGIN block\_list END\_KEY SEMI  
  
tiger\_program -> type\_declaration\_list funct\_declaration\_list

funct\_declaration\_list -> funct\_declaration\*

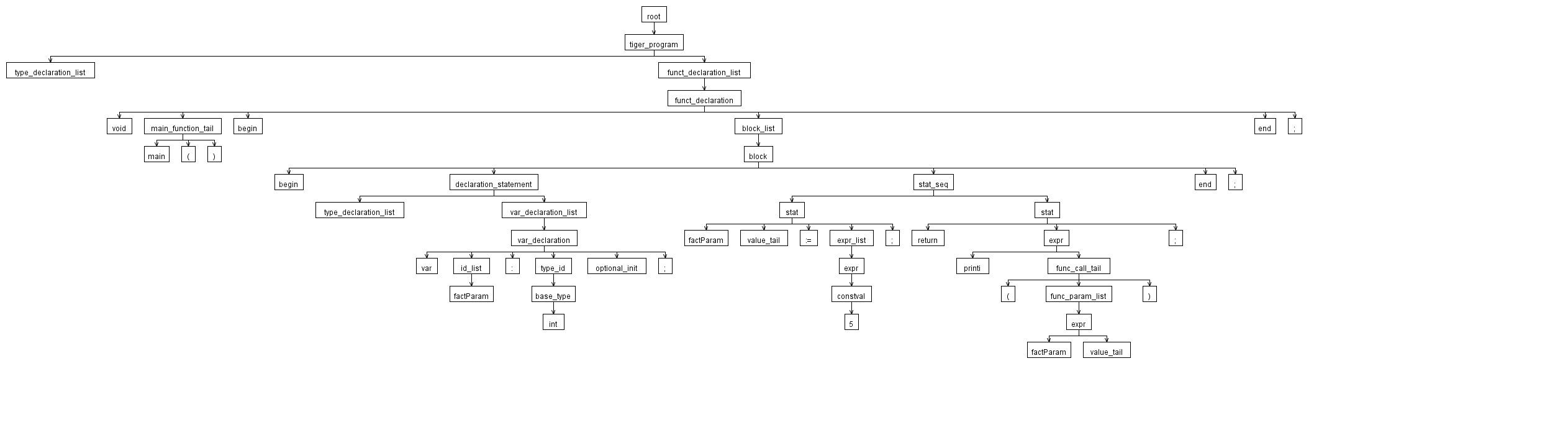
funct\_declaration -> ((type\_id funct\_declaration\_tail) | (VOID\_KEY (funct\_declaration\_tail | main\_function\_tail))) BEGIN\_KEY block\_list END\_KEY SEMI

funct\_declaration\_tail -> FUNCTION\_KEY ID LPAREN param\_list RPAREN

main\_function\_tail -> MAIN\_KEY LPAREN RPAREN  
~~main\_function ->~~

**Testing**

The initial phase of testing was ongoing and was primarily grammar checks for collision or recursion, and was done solely in ANTLRWorks. The following phase tested our grammar against a series of test programs we wrote to see if our grammar outputted a valid parse tree.   
  
The next page contains a tiny Tiger test program, followed by its parse tree against our grammar.



\* A full test program and parse tree is too large to reproduce here, but is included as part of the phase I deliverable.

/\* Variable declaration and subroutine call \*/

void main()

begin

begin

var factParam : int;

factParam := 5;

return printi(factParam);

end;

end;

**Lex/parse error generation**

Upon encountering a lexical error (i.e. a token was encountered that is not part of the language lex grammar), our lexer will output a message similar to the following example:  
  
Given token beginkey:  
  
Error At Line 3: ' 'beginerror (extraneous input 'beginerror' expecting BEGIN\_KEY)

As the correct token (begin) is meant to start a code block, further input is expected to produce parse errors. This is true in the case of our parser, and outputs multiple errors before reaching EOF:  
  
Error At Line 5: 'v'ar factParam : int; (required (...)+ loop did not match anything at input 'var')  
Error At Line 5: var 'f'actParam : int; (mismatched input 'factParam' expecting END\_KEY)  
Error At Line 5: var factParam ':' int; (mismatched input ':' expecting FUNCTION\_KEY)  
Error At Line 5: var factParam : 'i'nt; (mismatched input 'int' expecting BEGIN\_KEY)  
...