CS 4240 Project Phase 1 Report

Team 1: Currell Berry, Scott Vermeyen, Max Virgil

I. How to Run Grammar through ANTLRWorks

In order to handle outputting detailed error messages on the command line, we have created our own Java classes that add additional functionality to ANTLR's CommonTokenStream class. A reference to one of these classes is inside the @members block in our **Tiger.g** grammar file. **Tiger.g** is our re-written grammar in ANTLR format, and it also includes additional markup to generate an AST.

Unfortunately, ANTLRWorks will not recognize the new classes we have added. So, we created **TigerAntlrWorks.g**. This is simply **a copy of Tiger.g with the @members block removed**. By removing the @members block, ANTLRWorks will no longer complain about references to unknown classes.

There are two options to run our grammar in the ANTLRWorks IDE in order to view parse trees and ASTs:

Option 1 - using TigerAntlrWorks.g (easiest)

1. Open TigerAntlrWorks.g in ANTLRWorks
2. This file is ready for ANTLRWorks as-is.

Option 2 - using Tiger.g

1. Open Tiger.g in ANTLRWorks
2. Manually comment out "@members" block. This file will now be ready to be used by ANTLRWorks.

II. How to Run Grammar through Command Line

TigerMain.java contains the main function to run our parser on the command line. When run from the command line, the parser generated from Tiger.g has additional functionality to output more detailed error messages. There is also a debug option (-d, or --debug) available to print token types as they are consumed by the parser.

If Tiger.g has been modified while following steps from Option 2 in Part I, the @members block will need to be uncommented to successfully compile the code.

1. Add antlr-3.5.2-complete.jar to the classpath. Then, run **java org.antlr.Tool Tiger.g** to generate the lexer and parser, and run **javac *.java** to compile all files.

2. Run **java TigerMain -d < [path to tig file of your choice]**

This will use the generated parser/scanner .class files to parse the tig file. The -d (or --debug) option prints token types to stdout as they are consumed. This is an optional flag. For example, to run our large program sample with debug mode on, use the following command:

java TigerMain -d < LargeProgram.tig

III. Changes to Make Grammar LL(1)

The first changes that were made to the grammar were related to the <expr> and <index_expr> rules. Originally, the Tiger spec grouped all operators into the <binary-operator> rule, so we regrouped operators according to their precedence.

Old Rule

binary_operator: ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘=’ | ‘<>’ | ‘<’ | ‘>’ | ‘<=’ | ‘>=’ | ‘&’ | ‘|’ ;

New Rules

mult_operator : '*' | '/' ;
add_operator : '+' | '-' ;
compare_operator : '=' | '<>' | '<' | '>' | '<=' | '>=' ;
and_operator : '&' | '|' ;

Next, we changed the <expr> rule to meet requirements for operator precedence and left-associativity. These changes also removed the ambiguity and left-recursion that was previously present. We used EBNF notation here, as we do in many other rules, since ANTLR supports it.

Old Rule

expr: expr binary_operator expr | literal | value | '(' expr ')' ;

New Rules

expr : term4 (and_operator term4)* ;
term4 : term3 (compare_operator term3)* ;
term3 : term2 (add_operator term2)* ;
term2 : term1 (mult_operator term1)* ;
term1 : literal | value | '(' expr ')' ;

Similar changes were made to <index-expr>.

Old Rules

index_expr: index_expr index_oper index_expr | INTLIT | ID ;
index_oper: ‘+’ | ‘-’ | ‘*’ ;

New Rules

index_expr : index_term (add_operator index_term)* ;
index_term : index_factor ('*' index_factor)* ;
index_factor : INTLIT | ID ;

We then looked at rules that obviously had alternatives that could begin with the same token, meaning these decisions could not be made using an LL(1) parser using one token look-ahead. We used left-factoring in these cases, along with shorthand notation provided by ANTLR to indicate implicit intermediate rules.

Old Rules

id_list: ID | ID ‘,’ id_list ;
stat_seq: stat | stat stat_seq ;

value_tail: ‘[‘ index_expr ‘]’ | ‘[‘ index-expr ‘]’ ‘[’ index-expr ‘]’ | <NULL> ;
type
: base_type
| ARRAY ‘[‘ INTLIT ‘]’ OF base_type
| ARRAY '[' INTLIT ']' '[' INTLIT ']' OF base_type
;
stat: IF expr THEN stat_seq ENDIF ‘;’ ;
stat: IF expr THEN stat_seq ELSE stat_seq ENDIF ‘;’ ;

New Rules
id_list : ID (',' ID)* ;
stat_seq: stat+ ;
value_tail: ‘[‘ index_expr ‘]’ (‘[‘ index-expr ‘]’)? | <NULL> ;
type: base_type | ARRAY '[' INTLIT ']' ('[' INTLIT ']')? OF base_type ;
stat: IF expr THEN stat_seq (ELSE stat_seq ENDIF | ENDIF) ';' ;

The <stat> rule had additional LL(1) issues that needed to be resolved. Upon seeing an ID at the start of a <stat>, it is unclear of whether or not it is the beginning of an assignment or a function call. Also, when seeing an ID after ‘:=’, it was unclear whether that ID belonged to an expression or was the name of a function. So, we created several intermediate rules so that an LL(1) parser could handle this cases.

Old Rules
stat: value ‘:=’ expr ‘;’ ;
stat: opt_prefix ID ‘(‘ exprlist ‘)’ ‘;’ ;

New Rules
stat: function_call_or_assignment ;
function_call_or_assignment: ID (function_args | value_tail ':=' expr_or_function_call) ';' ;
function_args: '(' expr_list ')’ ;
expr_or_function_call: ID (expr_with_start_id | function_args) | expr_no_start_id ;

Finally, we turned on ANTLR options for “k=1” and “backtrack=no”, and this helped us identify the last issue keeping our grammar from being LL(1). Like other rule rewrites explained above, we needed to use left-factoring to remove ambiguity when peeking at a VOID token, since both function declarations and the main declaration could begin with “void”.

Old Rule
tiger_program : type_declaration_list funct_declaration_list main_function EOF;

New Rules
tiger_program: type_declaration_list funct_declaration_list_then_main EOF ;
funct_declaration_list_then_main
: VOID (funct_declaration_tail funct_declaration_list_then_main | main_function_tail)
| (ID|INT|FIXEDPT) funct_declaration_tail funct_declaration_list_then_main
;

IV. Sample Programs

Output for our two small programs are shown below. In our submission, we also have **LargeProgram.tig**, which is our large Tiger program.

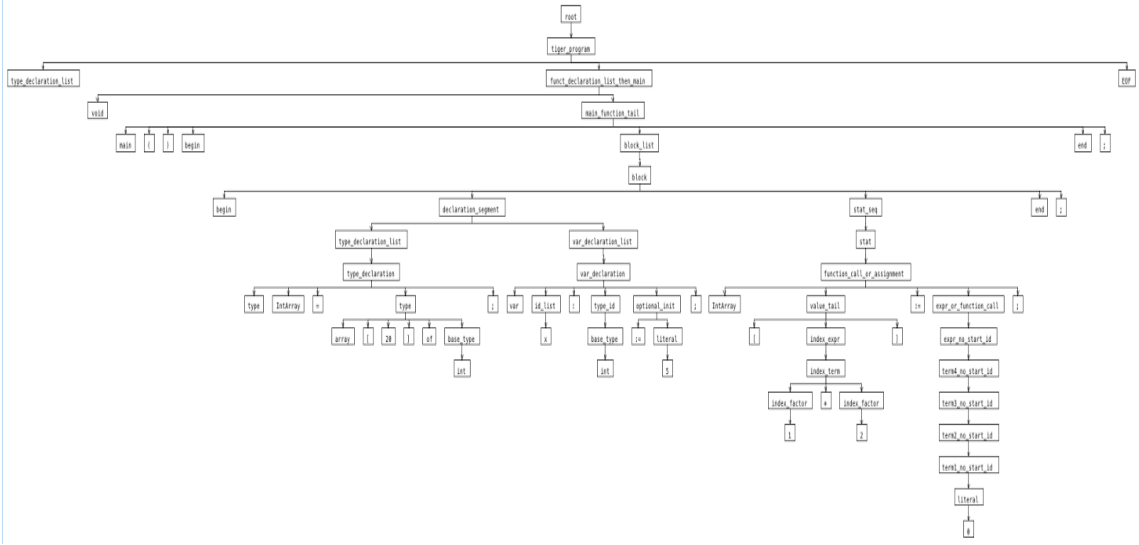
SmallSuccessfulProgram.tig

```
void main() begin
  begin
    type IntArray = array[20] of int;
    var x : int := 5;
    IntArray[1*2] := 0;
  end;
end;
```

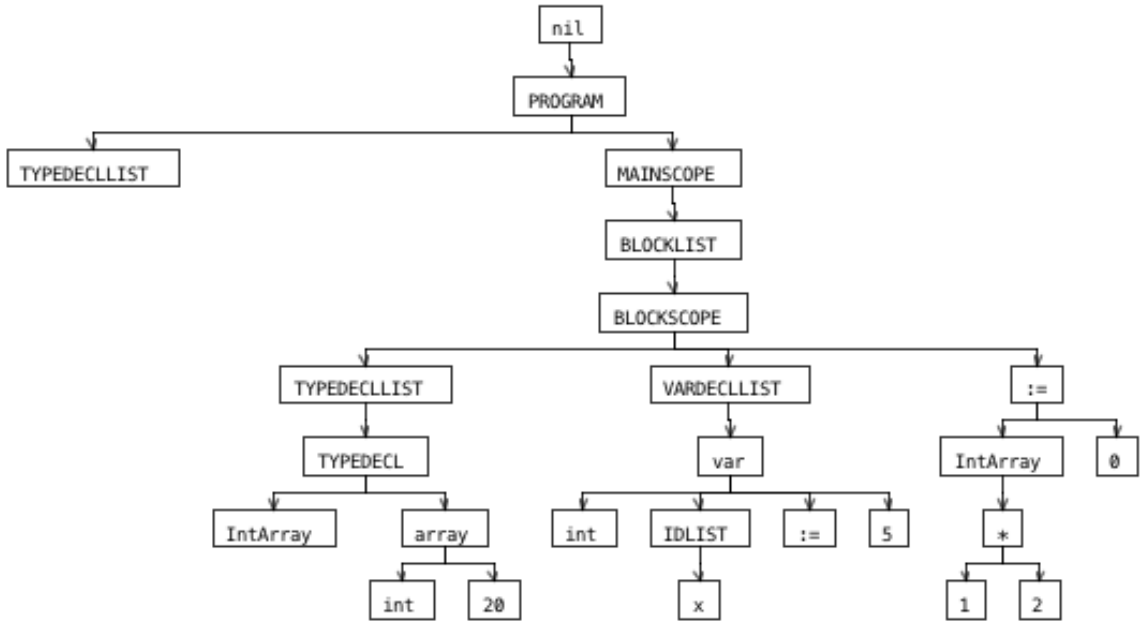
Sample output:

```
========================
Reading input from stdin (to read from file, redirect stdin)...
Running in debug mode (tokens will be printed as parser consumes them)...
Parsing input...

Tokens consumed by parser:

VOID MAIN LPAREN RPAREN BEGIN BEGIN TYPE ID EQ ARRAY LBRACK INTLIT RBRACK OF INT SEMI VAR ID COLON INT ASSIGN INTLIT
SEMI ID LBRACK INTLIT MULT INTLIT RBRACK ASSIGN INTLIT SEMI END SEMI END SEMI <EOF>

Successful parse!
========================
```

Parse Tree (open ParseTree.jpg for a better view):



Abstract Syntax Tree:



SmallUnsuccessfulProgram.tig
Line 3: arrays cannot be of type Float (or any ID).
Line 4: type must be assigned to var x first.
Line 5: missing semicolon.
Line 7: missing semicolon.

```
void main() begin
  begin
    type FloatArray = array[20][20] of Float;
    var x := 5;
    IntArray[1*2] := 0
  end;
end
```

Sample output:

```
==========================
Reading input from stdin (to read from file, redirect stdin)...
Parsing input...

line 3:39 mismatched input 'Float' expecting set null
        type FloatArray = array [ 20 ] [ 20 ] of Float
                                                 ^
line 4:10 mismatched input ':=' expecting COLON
        var x :=
              ^
line 6:2 extraneous input 'end' expecting SEMI
        end
        ^
line 8:0 mismatched input '<EOF>' expecting SEMI
        <EOF>
        ^


Unsuccessful parse
==========================
```

Sample output with debug mode on:

```
==========================
Reading input from stdin (to read from file, redirect stdin)...
Running in debug mode (tokens will be printed as parser consumes them)...
Parsing input...

Tokens consumed by parser:

VOID MAIN LPAREN RPAREN BEGIN BEGIN TYPE ID EQ ARRAY LBRACK INTLIT RBRACK LBRACK INTLIT RBRACK OF

line 3:39 mismatched input 'Float' expecting set null
        type FloatArray = array [ 20 ] [ 20 ] of Float
                                                 ^
ID SEMI VAR ID

line 4:10 mismatched input ':=' expecting COLON
        var x :=
              ^
ASSIGN INTLIT SEMI ID LBRACK INTLIT MULT INTLIT RBRACK ASSIGN INTLIT END

line 6:2 extraneous input 'end' expecting SEMI
        end
        ^
SEMI END

line 8:0 mismatched input '<EOF>' expecting SEMI
        <EOF>
        ^
<EOF> <EOF>

Unsuccessful parse
==========================
```