Tiger Compiler Design Report: Phase I

Luis Pastrana, Ben Templin, Rahul Tholakapalli
CS 4240: Compilers and Interpreters
25 October 2019

Overview

This document outlines the design, use, and features of our Tiger language compiler. This compiler uses the ANTLR framework to specify a grammar for the Tiger language and to generate a parser for this grammar (see Appendix B for the full grammar specification). The compiler itself is written in Java and Make is used to build the compiler from source. At this stage in compiler development, Tiger source code is compiled to an intermediate code representation that is printed to standard out and optionally output to a `.ir` file.

Further sections of this design specification detail how to build and run our compiler, advanced compiler options, and internal design features of the compiler. We also include an appendix containing sample Tiger programs and compiled outputs and an appendix containing the full Tiger grammar specification as used by ANTLR.

The next phase of our compiler implementation will take the intermediate code representation generated in this phase of the compiler and compile that to a program executable on MIPS machines

Contents

Building and Running the Tiger Compiler

Basic Building and Running

To build our compiler from scratch, run `make`. This will use ANTLR to generate the Java files for the lexer and parser; compile the lexer and parser files; and compile the syntax checker, semantic analyzer, IR code generator, and main compiler class.

To compile a Tiger program, run `java TigerCompiler <file-name>`. This will compile the passed in Tiger file. Upon successful compilation, the following will be printed to standard out: the name of the file being compiled, "Successful Parse," tuples of the form `<token_type, token_name>` for each token in the Tiger file, a stream of all token types in the file, "Successful compile," the symbol table generated by the semantic analyzer, and the generated IR code. Additionally, the IR code will be output to the file `<file-name>.ir`.[1] If an error is detected in the parse phase, that token will be discarded, parsing will continue, and all errors will be output to standard out at the end of this phase. Errors detected during the syntax checking and semantic analysis phase will be output immediately on detection and compilation will be stopped. In both error cases, the program will exit with exit code 1.

To clean the directory, run `make clean`. This will remove all ANTLR-generated files, and compiled Java files but will leave files output by the Tiger compiler.

Running Our Testing Suite

We created a testing suite program that runs most of the TA sample programs as well as a few of our own test cases. To run this program, run `java Testing`. Additionally, the Tiger programs for these test cases can be found in the Testing directory within the project. This testing suite only runs tests that should not cause errors because errors cause the JVM to exit in our implementation. The files that should cause an error when tested are:

- `testtypechecking.tiger`
- `testBreakError.tiger`
- `testLetScoping.tiger`

Additional Options

This section details advanced functionality that we have included in our Makefile and compiler. Additionally, a simplified version of this information can be found in README.md in the project directory.

[1] See the section on Advanced Compiler Options to change the outputs of this process.

Advanced Makefile Options

To assist in our internal development of the compiler, we added commands that we frequently used in development and testing to the Makefile.

1. `make compiler`
   a. This option will recompile all Java files in the project directory and can be used to rebuild the compiler without using ANTLR to regenerate the parser and lexer files.
2. `make parser`
   a. This option runs ANTLR to regenerate the parser and lexer Java files. More specifically, it runs the specified `ANTLR_JAR` file on the `GRAMMAR_FILE` with the `ANTLR_FLAGS` defined in the Makefile. It does not compile the parser.
3. `make cleanantlr`
   a. This option removes ANTLR-generated files as defined by the `ANTLR_FILES` variable in the Makefile. As of Phase I, these files are:
      i. `tigerBaseListener.java`
      ii. `tigerBaseVisitor.java`
      iii. `tigerLexer.*`
      iv. `tigerListener.java`
      v. `tigerParser.java`
      vi. `Tiger.interp`
      vii. `tiger.tokens`.
4. `make cleanjava`
   a. This option removes any compiled Java files.
5. `make cleantiger`
   a. This option removes any files output by the Tiger compiler. As of Phase I, this removes all `.ir` files.
6. `make rebuild`
   a. This option rebuilds the compiler from scratch by running `make clean` and `make`. It will remove all ANTLR-generated files, compiled Java files, regenerate the parser and lexer files, and build the compiler.

Advanced Compiler Options

We added these compiler flags to make our compiler easier to use and test. All advanced options are optional and can be used in any order.

1. `-no-print`

   a. This option suppresses all output to standard out except for warnings and error messages.
2. `-no-ir-file`
   a. This option tells the compiler not to generate a `.ir` file during compilation. This option overrides the `-outfile` option if used. If both options are used, a warning will be generated, and the program will be compiled without an output file.
3. `-outfile=<out-file-name>`
   a. This option allows you to specify an output file for the IR code that is generated by the compiler other than `<file-name>.ir`. It is overridden if the `-no-ir-file` flag is used.

Design Internals

This section will outline some of the internal design choices that we made for our compiler project. We used the Visitor paradigm provided by ANTLR to perform semantic analysis, symbol table construction, and IR code generation. We used the Listener paradigm to perform syntax checking and output all tokens in the Tiger program file. We employ a two-pass approach to walking the parse tree, with the first pass being used to build the symbol table and perform semantic analysis and the second pass being used for IR code generation.

Parsing

Design

Parsing is performed performed using the ANTLR-generated parser and lexer, along with a custom error handler that counts the number of errors encountered. Parser functionality is exposed to the overall Tiger compiler through the `TigerParserWrapper`, which allows for the compiler to get the number of errors encountered during lexing and parsing (which include missing or unexpected tokens), get the string of error messages, get the vocabulary for the grammar, print a string of the parse tree, and reset the parser.

Associated Files

- ANTLR-generated Lexer and Parser files
- `TigerErrorHandler.java`
- `TigerParserWrapper.java`

Syntax Checking

Design

To perform syntax checking, we used a Listener to visit all terminal nodes in the parse tree, generate the string of all token types, and generate the tuples mapping token types to tokens. We used the Listener framework for this process because ANTLR provides a `visitTerminal` method that makes it easy for the compiler to quickly visit all terminal nodes.

Associated Files

- `SyntaxChecker.java`

Symbol Table Construction

Design

Our symbol table is constructed as somewhat of a tree with each scope being a node called a `ScopeNode`. In this implementation, each node can have multiple children but only one parent, with a symbol being in-scope if it can be looked up in the current `ScopeNode` or one of its parent nodes. This system also allows for a scope to override a symbol present in its parent scope because symbols are retrieved from the table by first looking in the current scope and then traversing through the parents in order. Each entry in the symbol table is a map with the name of the symbol being the key and a `SymbolData` entry being the value. The `SymbolData` class stores metadata about each symbol in the table, such as the type of the symbol, whether the symbol is an array, and the array size or parameter list, if applicable.

The symbol table is exposed to other elements of the Tiger compiler through the `SymbolTable` class, which provides methods for storing values, looking up values, and creating new scopes. The `SymbolData` class is also exposed to other parts of the compiler because that is what is returned when an entry is looked up in the table. The `ScopeNode` class itself is not exposed to other parts of the compiler and is purely used by the symbol table to manage its entries.

Associated Files

- `ScopeNode.java`
- `SymbolData.java`
- `SymbolTable.java`

Semantic Analysis

Design

The semantic analyzer for the compiler is where the bulk of the work for Phase I of this project is done. The `SemanticChecker` class uses the Visitor paradigm within ANTLR to visit every token in the parsed program on a first pass to add symbols to the symbol table and ensure the program confirms to the semantic requirements of Tiger. This pass also checks the symbol table for duplicate entries or use without declaration.

Additionally, in each visit to a node in the parse tree, the type of that node is returned as that is the only information needed for type checking. One interesting modification that we made to this process is that we add “ array” to the type if the id being referenced is an array that is not being indexed. Additionally, if the id is a function name, the value returned is the return type of the function. This information is used to type check all statements in the program before proceeding to IR code generation. Finally, a stack is employed to track when loops are entered and exited to ensure that break statements only occur within these loops.

Associated Files

- `SemanticChecker.java`

IR Code Generation

Design

The IR code generator also uses the Visitor paradigm to perform a second pass to each node in the Tiger program's parse tree and output the necessary intermediate code representation. During this process, the symbol table from the semantic checking phase is updated with temporary variables created during IR code generation. The `IRCodeGenerator` class also has the necessary functions to generate labels, temporary variables, and emit actual code to our outputs. We used a stack to store labels for while- and for-loops, so that appropriate labels can be emitted where the loops end. Finally, the `IRCodeGenerator` class provides support to write all generated code to an output file.

Associated Files

- `IRCodeGenerator.java`

The Tiger Compiler

Design

The overall `TigerCompiler` class handles argument validation, checks that the passed in file actually exists, and then runs all aspects of the compiler described above. The class first uses a `TigerParserWrapper` to generate the parse tree, checks to see if any errors were generated from the parse tree, passes it to the syntax checker to output the tokens, runs the semantic checker to verify that the program is syntactically correct (and exits if there is an error), prints the symbol table, and runs the IR code generator to finally output the generated code.

Associated Files

- `TigerCompiler.java`

Appendix A: Sample Tiger Programs and Output

General Tiger Example Program

Tiger Program

```
main let
     type ArrayInt = array [100] of int; /*Declare ArrayInt as a new
type */
     var X, Y : ArrayInt := 10; /* Declare vars X and Y as arrays
with initialization */
     var i, sum : int := 0;
in
begin
     while i < 100 do /* while loop for dot product */
               sum := sum + X[i] * Y [i];
     enddo;
end
```

Generated Output

```
Compiling file "tiger_example"

Successful Parse

< MAIN , main >
< LET , let >
< TYPE , type >
< ID , ArrayInt >
< EQUALS , = >
< ARRAY , array >
< LBRACK , [ >
< INTLIT , 100 >
< RBRACK , ] >
< OF , of >
< INT , int >
< SEMI , ; >
< VAR , var >
< ID , X >
< COMMA , , >
< ID , Y >
< COLON , : >
< ID , ArrayInt >
< ASSIGN , := >
< INTLIT , 10 >
< SEMI , ; >
< VAR , var >
< ID , i >
< COMMA , , >
< ID , sum >
< COLON , : >
< INT , int >
< ASSIGN , := >
< INTLIT , 0 >
< SEMI , ; >
< IN , in >
< BEGIN , begin >
< WHILE , while >
< ID , i >
< LESSER , < >
< INTLIT , 100 >
< DO , do >
< ID , sum >
< ASSIGN , := >
< ID , sum >
< PLUS , + >
< ID , X >
< LBRACK , [ >
< ID , i >
< RBRACK , ] >
< MULT , * >
< ID , Y >
< LBRACK , [ >
< ID , i >
< RBRACK , ] >
< SEMI , ; >
< ENDDO , enddo >
< SEMI , ; >
< END , end >

MAIN LET TYPE ID EQUALS ARRAY LBRACK INTLIT RBRACK OF INT SEMI VAR ID
COMMA ID COLON ID ASSIGN INTLIT SEMI VAR ID COMMA ID COLON INT ASSIGN
INTLIT SEMI IN BEGIN WHILE ID LESSER INTLIT DO ID ASSIGN ID PLUS ID
LBRACK ID RBRACK MULT ID LBRACK ID RBRACK SEMI ENDDO SEMI END

Successful Compile

scope 1:
    prints, func, int
    getchar, func, string
    flush, func, int
    X, var, int
    Y, var, int
    i, var, int
    printi, func, int
    sum, var, int
    ArrayInt, type, int
```

```
IR GENERATED CODE:

# start_function main
void main():
assign X, 10
assign Y, 10
assign i, 0
assign sum, 0
label1:
brgeq i, 100, label3
add 1, 0, t1
goto label4
label3:
add 0, 0, t1
label4:
breq t1, 0, label2
load t2, X, i
add sum, t2, t3
load t4, Y, i
mult t3, t4, t5
assign sum, t5
goto label1
label2:
# end_function main
```

Type Check Test

Tiger Program

```
main let
    type First_Int = int;
    type Second_Int = First_Int;
    var X : First_Int := 0;
    var Y : Second_Int;
    var A : int := 0;
    var B : float := 0.1;
in begin
    Y := Y + X;
    A := A + B;
end
```

Generated Output

```
Compiling file "Testing/testtypechecking.tiger"

Successful Parse

< MAIN , main >
< LET , let >
< TYPE , type >
< ID , First_Int >
< EQUALS , = >
< INT , int >
< SEMI , ; >
< TYPE , type >
< ID , Second_Int >
< EQUALS , = >
< ID , First_Int >
< SEMI , ; >
< VAR , var >
< ID , X >
< COLON , : >
< ID , First_Int >
< ASSIGN , := >
< INTLIT , 0 >
< SEMI , ; >
< VAR , var >
< ID , Y >
< COLON , : >
< ID , Second_Int >
< SEMI , ; >
< VAR , var >
< ID , A >
< COLON , : >
< INT , int >
< ASSIGN , := >
< INTLIT , 0 >
< SEMI , ; >
< VAR , var >
< ID , B >
< COLON , : >
< FLOAT , float >
< ASSIGN , := >
< FLOATLIT , 0.1 >
< SEMI , ; >
< IN , in >
< BEGIN , begin >
< ID , Y >
< ASSIGN , := >
< ID , Y >
< PLUS , + >
< ID , X >
< SEMI , ; >
< ID , A >
< ASSIGN , := >
< ID , A >
< PLUS , + >
< ID , B >
< SEMI , ; >
< END , end >
```

```
MAIN LET TYPE ID EQUALS INT SEMI TYPE ID EQUALS ID SEMI VAR ID COLON
ID ASSIGN INTLIT SEMI VAR ID COLON ID SEMI VAR ID COLON INT ASSIGN
INTLIT SEMI VAR ID COLON FLOAT ASSIGN FLOATLIT SEMI IN BEGIN ID
ASSIGN ID PLUS ID SEMI ID ASSIGN ID PLUS ID SEMI END

SEMANTIC ERROR! Error at Y in line 9 character 6 invalid types for +
operator: Expected Second_Int. Got First_Int.
```

Functions in Functions

Tiger Program

```
main let
     var i : int := 1;

     function first ( j : int ) : int begin
     return j;
     end;

     function second ( k : int ) : int begin
     k := first(k);
     return k;
     end;
in begin
     i := second(i);
end
```

Generated Output

```
Compiling file "Testing/functionInFunction.tiger"

Successful Parse

< MAIN , main >
< LET , let >
< VAR , var >
< ID , i >
< COLON , : >
< INT , int >
< ASSIGN , := >
< INTLIT , 1 >
< SEMI , ; >
< FUNC , function >
< ID , first >
< LPAREN , ( >
< ID , j >
< COLON , : >
< INT , int >
< RPAREN , ) >
< COLON , : >
< INT , int >
< BEGIN , begin >
< RETURN , return >
< ID , j >
< SEMI , ; >
< END , end >
< SEMI , ; >
< FUNC , function >
< ID , second >
< LPAREN , ( >
< ID , k >
< COLON , : >
< INT , int >
< RPAREN , ) >
< COLON , : >
< INT , int >
< BEGIN , begin >
< ID , k >
< ASSIGN , := >
< ID , first >
< LPAREN , ( >
< ID , k >
< RPAREN , ) >
< SEMI , ; >
< RETURN , return >
< ID , k >
< SEMI , ; >
< END , end >
< SEMI , ; >
< IN , in >
< BEGIN , begin >
< ID , i >
< ASSIGN , := >
< ID , second >
< LPAREN , ( >
< ID , i >
< RPAREN , ) >
< SEMI , ; >
< END , end >

MAIN LET VAR ID COLON INT ASSIGN INTLIT SEMI FUNC ID LPAREN ID COLON
INT RPAREN COLON INT BEGIN RETURN ID SEMI END SEMI FUNC ID LPAREN ID
COLON INT RPAREN COLON INT BEGIN ID ASSIGN ID LPAREN ID RPAREN SEMI
RETURN ID SEMI END SEMI IN BEGIN ID ASSIGN ID LPAREN ID RPAREN SEMI
END

Successful Compile

scope 1:
    prints, func, int
    getchar, func, string
    flush, func, int
    i, var, int
    printi, func, int
    first, func, int
    second, func, int
    scope 2:
      j, var, int
    scope 3:
      k, var, int


IR GENERATED CODE:

# start_function main
void main():
assign i, 1
# start_function first
int first(int j):
args: j
return j
# end_function first
# start_function second
int second(int k):
args: k
callr k, first, k
return k
# end_function second
callr i, second, i
# end_function main
```

Appendix B: Tiger Grammar Specification for ANTLR

```
grammar tiger;

COMMENT : '/*'(.|'\n')*?'*/' -> skip;

WS : [ \t\r\n]+ -> skip;

MAIN : 'main';

COMMA : ',';

COLON : ':';

SEMI : ';';

LPAREN : '(';

RPAREN : ')';

LBRACK : '[';

RBRACK : ']';

LBRACE : '{';

RBRACE : '}';

PERIOD : '.';

EQUALS : '=';

PLUS : '+';

MINUS : '-';

MULT : '*';

DIV : '/';

EXP: '**';

EQ : '==';

NEQ : '!=';

LESSER : '<';

GREATER : '>';
```

```
LESSEREQ : '<=';

GREATEREQ : '>=';

AND : '&';

OR : '|';

ASSIGN : ':=';

ARRAY : 'array';

RECORD : 'record';

BREAK : 'break';

DO : 'do';

ELSE : 'else';

END : 'end';

FOR : 'for';

FUNC : 'function';

IF : 'if';

IN : 'in';

LET : 'let';

OF : 'of';

THEN : 'then';

TO : 'to';

TYPE : 'type';

VAR : 'var';

WHILE : 'while';

ENDIF : 'endif';

BEGIN : 'begin';

ENDDO : 'enddo';
```

```
RETURN : 'return';

INT : 'int';

FLOAT : 'float';

ID : [a-zA-Z_][a-zA-Z0-9_]*;

INTLIT : [0-9]+;

FLOATLIT: [0-9]+('.'[0-9]+)?([eE][+-]?[0-9]+)?;

tiger_program : MAIN LET declaration_segment IN BEGIN stat_seq END;

declaration_segment : type_declaration_list var_declaration_list
function_declaration_list;

type_declaration_list : type_declaration type_declaration_list | /* NULL */;

var_declaration_list : var_declaration var_declaration_list | /* NULL */;

function_declaration_list : function_declaration function_declaration_list | /* NULL
*/;

type_declaration : TYPE ID EQUALS type SEMI;

type : ARRAY LBRACK INTLIT RBRACK OF type_id | ID | type_id;

type_id : INT | FLOAT;

var_declaration : VAR id_list COLON type optional_init SEMI;

id_list : ID id_list_tail;

id_list_tail : COMMA ID id_list_tail | /* NULL */;

optional_init : ASSIGN constant | /* NULL */;

function_declaration : FUNC ID LPAREN param_list RPAREN ret_type BEGIN stat_seq END
SEMI;

param_list : param param_list_tail | /* NULL */;

param_list_tail : COMMA param param_list_tail | /* NULL */;

ret_type : COLON type;

param : ID COLON type;
```

```
stat_seq : stat stat_seq_tail;

stat_seq_tail : stat stat_seq_tail | /* NULL */;

stat :  IF expr THEN stat_seq else_stat ENDIF SEMI |

        WHILE expr DO stat_seq ENDDO SEMI |

        FOR ID ASSIGN expr TO expr DO stat_seq ENDDO SEMI |

        BREAK SEMI |

        RETURN expr SEMI |

        LET declaration_segment IN stat_seq END SEMI |

        ID id_tail;

id_tail :   ASSIGN assign_tail |

            LBRACK expr RBRACK ASSIGN expr SEMI |

            LPAREN expr_list RPAREN SEMI;

assign_tail: expr SEMI | ID LPAREN expr_list RPAREN SEMI;

lvalue : ID lvalue_tail;

lvalue_tail : LBRACK expr RBRACK | /* NULL */;

else_stat : ELSE stat_seq | /* NULL */;

expr : and_term e_tail; // <expr> is our "<or-term>"

e_tail : OR and_term e_tail | /* NULL */;

and_term : comparison_term and_tail;

and_tail : AND comparison_term and_tail | /* NULL */;

comparison_term: div_term comparison_tail;

comparison_tail : (EQ | NEQ | LESSER | LESSEREQ | GREATER | GREATEREQ) div_term | /*
NULL */;

div_term : mult_term div_tail;

div_tail : DIV mult_term div_tail | /* NULL */;
```

```
mult_term : sub_term mult_tail;

mult_tail : MULT sub_term mult_tail | /* NULL */;

sub_term : add_term sub_tail;

sub_tail : MINUS add_term sub_tail | /* NULL */;

add_term : pow_term add_tail;

add_tail : PLUS pow_term add_tail | /* NULL */;

pow_term : factor pow_tail;

pow_tail : EXP factor pow_tail | /* NULL */;

factor : LPAREN expr RPAREN | constant |  lvalue;

constant : sign INTLIT | sign FLOATLIT;

sign : PLUS | MINUS | /*NULL */;

expr_list : expr expr_list_tail | /* NULL */;

expr_list_tail : COMMA expr expr_list_tail | /* NULL */;
```