CS 4240: Project Phase 1: Front-end: Scanner and Parser and ASTs

8th Oct' 2014

Team: Tyler Bell Brandon Chastain Akash Gangil

Table Of Contents:

1. Project Folder Structure

Tiger/

Tiger.g: Grammar in appropriate LL(1) form along with the lexical spec for Tiger in ANTLR format

sample_test_program.tiger: Sample test program provided with the assignment.
sample_test_program.png: AST of the sample test program.
sample_test_program_parse_tree.jpeg: Parse tree of the sample test program.

test_program.tiger: *Test program submitted by us* utilizing various language constructs with comments.
test_program.png: Test program's AST.
test_program_parse_tree.jpeg: Parse tree of the test program.

output/

TigerParser.java: Generated parser code.
TigerLexer.java: Generated scanner code.
Tiger.tokens: Generated token file.
Tiger.java: Driver for AST generation, outputting sequence of token types and determining if the parse was successful else pointing out the parser and lexer errors.
run.sh: Script which takes a tiger program as a command line argument and outputs the list of tokens, generates the AST png file using the **dot** utility.
***Needs to have dot installed**
clean.sh: Cleans the folder of generated AST image, dot file and classes folder
***.jar:** Jars for antlr runtime, StringTemplate and other libraries.

2. To Run

1. Place the program in the Tiger/output folder.
2. Type ./run.sh <tiger program file>

This will generate the following

- A list of tokens found as they occur in the source (printed in the terminal)
- <name>.dot - A description of the AST of the input program
- <name>.png - The AST for the input program

3. Grammar changes to make it LL(1)

1. *Ambiguity in between the function-declaration and main-function rule because of the keyword void. In the below case, ret-type can evaluate to void too, which creates an ambiguity for the parser to decide which rule to follow (1) or (2).*

(1) <funct-declaration> → <ret-type> function id (<param-list>) begin <block-list> end;

(2) <main-function> → void main ( ) begin <block-list> end;

<ret-type> → void
<ret-type> → <type-id>

Resolution: Use left factoring:

<function-declaration> → VOID <function-definition-void>
<function-declaration> → <type-id> <function-definition-body>

<function-definition-void> → <function-definition-body> | <function-definition-main>

<function-definition-body> → function id (<param-list>) begin <block-list> end ;
<function-definition-main> → main ( ) begin <block-list> end;

2. *Left Recursion in the below rule*

<expr> → <expr> <binary-operator> <expr>
<expr> → <const>
<expr> → <value>
<expr> → (<expr>)

Resolution:

The above rules can be rewritten as

Introducing <expr-new>,

<expr> → (<const> | <value> | (<expr>)) <expr-new>
<expr-new> → ε | <binary-operator> <expr> <expr-new>

3. *Left Recursion in the below rule*

<index-expr> → <index-expr> <index-oper> <index-expr>
<index-expr> → INTLIT
<index-expr> → id

Resolution:

The above rules can be rewritten as

Introducing <index-expr-new>,

<index-expr> → (id | INTLIT) <index-expr-new>
<index-expr-new> → ε | <index-oper> <index-expr> <index-expr-new>

4. *Ambiguity in the below rule,*

```
<value-tail> → [<index-expr>]
<value-tail> → [<index-expr> ] [<index-expr>]
<value-tail> → NULL
```

Resolution:

```
<value-tail> → [<index-expr>] ([<index-expr>])?
```

5. *Ambiguity in the below rule,*

```
<id-list>  → id
<id-list>  → id, <id-list>
```

Resolution: *Introducting, <id-list-tail>*

```
<id-list> → ID <id-list-tail>
<id-list-tail> →  ε | , <id-list>
```

6. *Ambiguity in the below rule,*

```
<type> → <base-type >
<type> → array[INTLIT] of <base-type >
<type> → array [INTLIT][INTLIT] of <base-type>
```

Resolution: *Introduction <array-dimensions> and <array-dimension>*

```
<type> → array <array-dimensions> of <base-type>

<array-dimensions> → <array-dimension> <array-dimension>?
<array-dimension> → [INTLIT]
```

7. *Ambiguity in the below rule:*

```
*<stat-seq> → <stat> (1)
<stat-seq> → <stat> <stat-seq> (2)

**<stat> → <value> := <expr> ;
***<stat> → if <expr> then <stat-seq> endif ;
<stat> →  if <expr > then <stat-seq> else <stat-seq> endif;
<stat> → while <expr> do <stat-seq> enddo;
<stat> → for id := <index-expr> to <index-expr> do <stat-seq> enddo;
<stat> → <optprefix> id( <exprlist> ) ;
<optprefix> → <value> :=
<optprefix> → NULL
<stat> → break;
<stat> → return <expr> ;
<stat> → <block-list> ;
```

There are three problems with the the above set of rules:

1. * Resolution of the <stat-seq> rule is ambiguous between (1) and (2)

2. ** <optprefix> can resolve to <value> :=. Thereby there is ambiguity in resolving stat when we see <value> := between the following rules:

(a) <stat> → <value>:= <expr>
(b) <stat> → <optprefix> id (<exprlist>);

3. ***Resolution of the <stat> rule is ambiguous between (4) and (5)

Resolution:

1. Resolve by left factoring,
   `<stat-seq> → <stat> +`

2. Resolve by left factoring again, Introducing <val-assign>
   ```
   <stat> → <value-assign> (<expr> | id <exprlist>);
   <val-assign> → <value> :=
   <stat> → id (<exprlist>);
   ```

3. Resolve by left factoring. Introducing <if-stmt> <else-stmt>
   ```
   <stat>  → <if-stmt>
   <if-stmt> → if <expr> then <stat-seq> <else-stmt> endif;
   <else-stmt> → ε | else <stat-seq>
   ```

4. Testing

1. *Outputs the list of tokens and states if the parse is successful.*

[01:28 gangil@gangil:~/dev/Tiger/output]./run.sh sample_test_program.tiger
VOID MAIN LPAREN RPAREN BEGIN BEGIN TYPE ID EQ 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 BEGIN FOR ID ASSIGN INTLIT TO INTLIT DO ID ASSIGN ID PLUS ID LBRACK ID RBRACK MULT ID LBRACK ID RBRACK SEMI ENDDO SEMI ID LPAREN ID RPAREN SEMI END SEMI END SEMI END SEMI
Successful Parse

2. *In case of a lexer error states the line number, reason, character at which it occurred.*

[01:30 gangil@gangil:~/dev/Tiger/output]./run.sh sample_test_program.tiger
VOID MAIN LPAREN RPAREN BEGIN BEGIN TYPE ID EQ 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 BEGIN FOR ID ASSIGN INTLIT TO INTLIT DO ID ASSIGN ID PLUS ID LBRACK ID RBRACK MULT ID LBRACK ID RBRACK SEMI ENDDO SEMI ID LPAREN ID RPAREN SEMI END SEMI END SEMI END SEMI
*** Lexer Errors ***
line 5:8 no viable alternative at character '_' **Character at which the error occurred: ['_']**

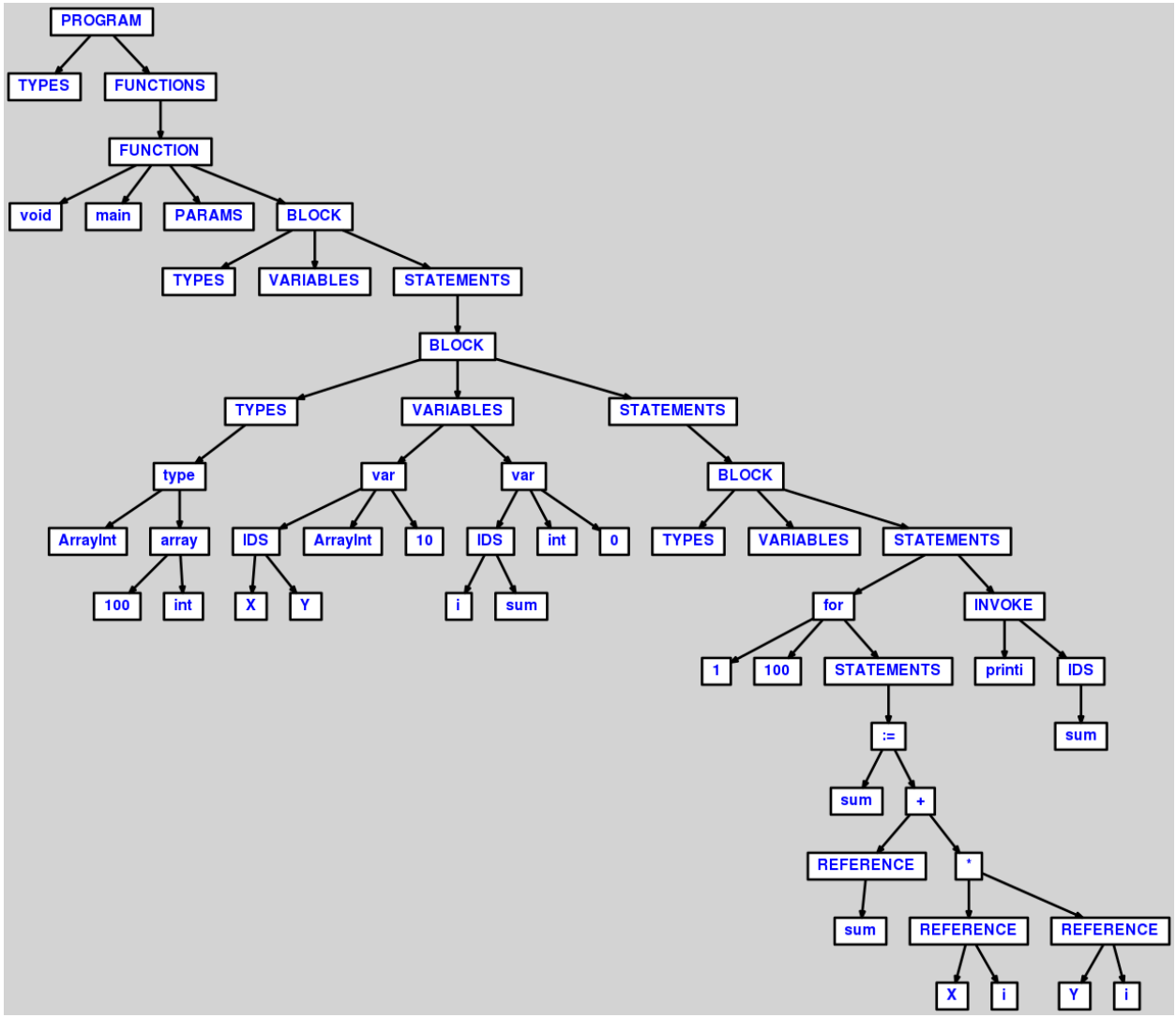
3. *In case of parser error does the same and also tells about the expecting token.*

[01:34 gangil@gangil:~/dev/Tiger/output]./run.sh sample_test_program.tiger
VOID MAIN LPAREN RPAREN BEGIN BEGIN TYPE ID EQ ARRAY LBRACK INTLIT RBRACK OF INT SEMI VAR ID COMMA ID COLON ID ASSIGN INTLIT SEMI VAR ID COMMA ID COLON INT ASSIGN INTLIT BEGIN FOR ID ASSIGN INTLIT TO INTLIT DO ID ASSIGN ID PLUS ID LBRACK ID RBRACK MULT ID LBRACK ID RBRACK SEMI ENDDO SEMI ID LPAREN ID RPAREN SEMI END SEMI END SEMI END SEMI
*** Parser Errors ***
line 7:4 mismatched input 'begin' expecting SEMI **Character at which the error occurred: ['']**

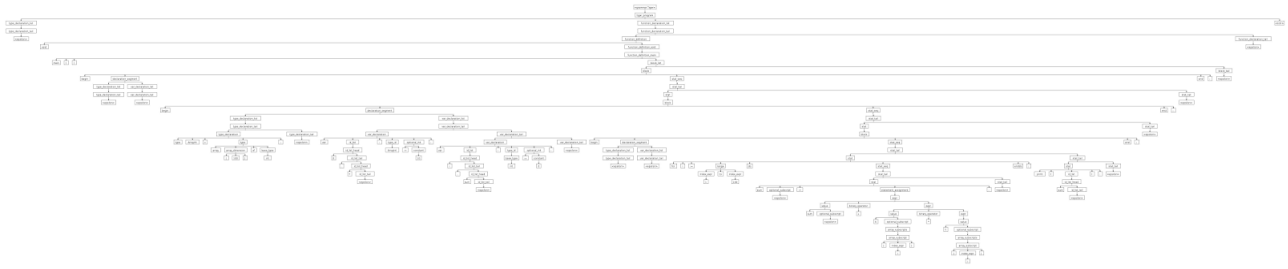
5. LL(1) Grammar Antlr

To make sure our grammar is LL(1) we used the following options:
k=1
backtrack=no

6. Sample Program AST (Tiger/sample_test_program.png)



7. Sample Program Parse Tree (Tiger/sample_test_program_parse_tree.jpeg:)

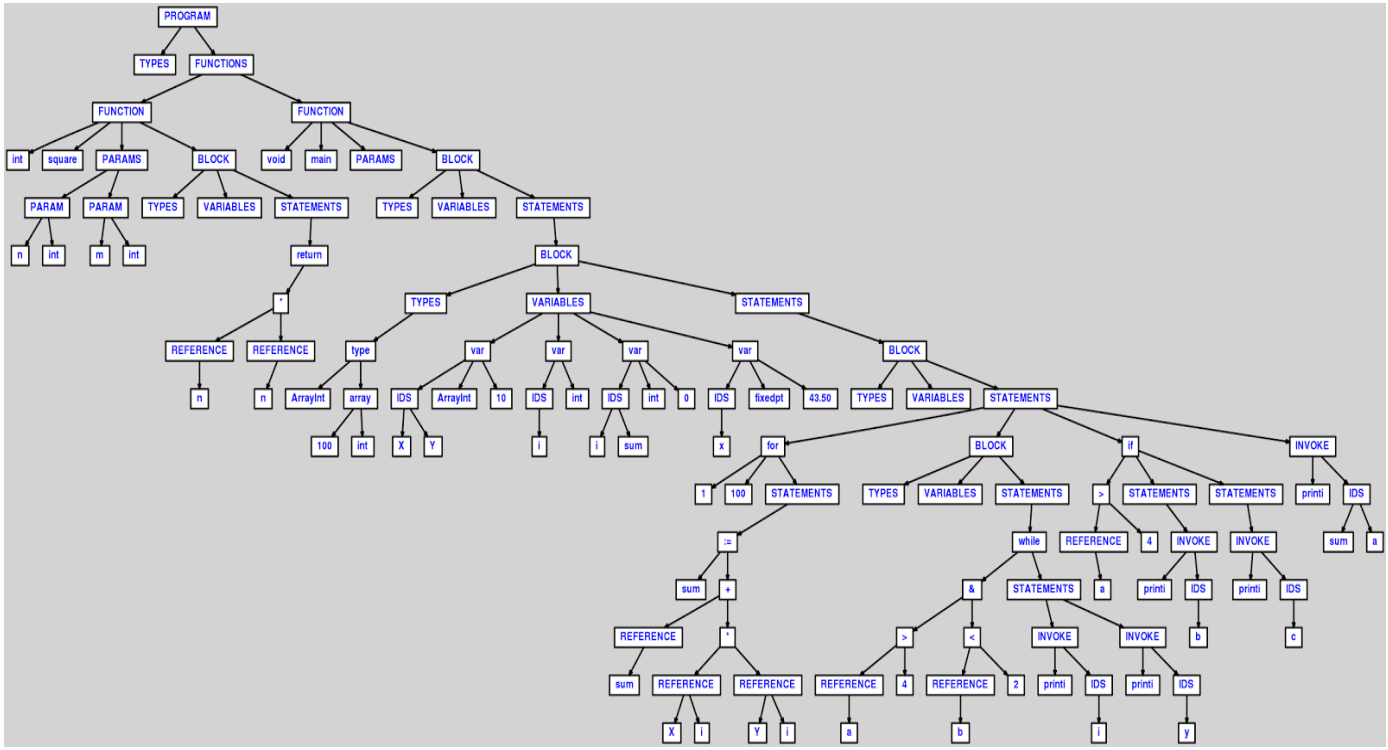


8. Test Program

```
/*Test Program with most of the Tiger grammar constructs*/

/* Helper Function with multiple parameters */
int function square(n:int, m:int)
begin
  begin
    return n*n;
  end;
end;

/*Main Function*/
void main()
begin
  begin
    /*Type declaration*/
    type ArrayInt = array [100] of int; var X, Y : ArrayInt := 10;

    /*Variable Declaration not definition*/
       var i:int;

    /*Integer Variable Declaration*/
    var i, sum : int := 0;

    /*FixedPoint variable declaration*/
    var x: fixedpt := 43.50;

    begin

      /*For loop*/
      for i := 1 to 100 do
        sum := sum + X[i] * Y[i];
      enddo;
      begin
        /*While construct with mutiple predicates
          and multiple statements*/
        while((a>4)  & (b<2)) do
          printi(i);
          printi(y);
        enddo;
      end;

      /* If/Then/Else Construct */
      if(a>4) then
        printi(b);
      else
        printi(c);
      endif;

      /*Function invocation with mutiple args*/
      printi(sum, a);
    end;
```

```
  end;
end;
```

9. Test Program AST (Tiger/test_program.png)



10. Test Program Parse Tree (Tiger/test_program_parse_tree.jpeg)