David Benas
Sean Collins
Casey Evanish

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:

```
COMMA -> ','
COLON -> ':'
SEMI -> ';'
LPAREN -> '('
RPAREN -> ')'
LBRACK -> '['
RBRACK -> ']'
PLUS -> '+'
MINUS -> '-'
MULT -> '*'
DIV -> '/'
EQ -> '='
```

```
NEQ -> '<>'
LESSER -> '<'
LESSEREQ -> '<='
GREATER -> '>'
GREATEREQ -> '>='
AND -> '&'
OR -> '|'
ASSIGN -> ':='
```

```
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`

David Benas
Sean Collins
Casey Evanish

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.

```
FUNCTION_KEY -> 'function'        THEN_KEY -> 'then'
BEGIN_KEY -> 'begin'              ENDIF_KEY -> 'endif'
END_KEY -> 'end'                  ELSE_KEY -> 'else'
VOID_KEY -> 'void'                WHILE_KEY -> 'while'
MAIN_KEY -> 'main'                ENDDO_KEY -> 'enddo'
TYPE_KEY -> 'type'                FOR_KEY -> 'for'
ARRAY_KEY -> 'array'              ID_KEY -> 'id'
OF_KEY -> 'of'                    TO_KEY -> 'to'
INT_KEY -> 'int'                  DO_KEY -> 'do'
FIXEDPT_KEY -> 'fixedpt'          BREAK_KEY -> 'break'
VAR_KEY -> 'var'                  RETURN_KEY -> 'return'
IF_KEY -> 'if'
```

2. 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;}
```

3. 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
```

David Benas
Sean Collins
Casey Evanish

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)?)?
  ```

David Benas
Sean Collins
Casey Evanish

- **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
```

David Benas
Sean Collins
Casey Evanish

- **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.

David Benas
Sean Collins
Casey Evanish

```
/* Variable declaration and subroutine call */
void main()
begin
    begin
        var factParam : int;
        factParam := 5;
        return printi(factParam);
    end;
end;
```


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

David Benas
Sean Collins
Casey Evanish

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)
...
```