Kennesaw State University

College of Computing and Software Engineering

Department of Computer Science

CS4308 Concepts of Programming Languages W01

CPL Assignment, Deliverable 2

Michael Epps

mepps5@students.kennesaw.edu

November 3, 2020

Problem Statement

The objective of this part of the CPL project is to implement a parser for the Ada grammar subset we have chosen. This parser will use the scanner developed in the last part of the project to read a stream of tokens. These tokens are then in turn to be parsed out into statements and expressions representing the grammar subset chosen. The primary goal of this parser is to build an Abstract Syntax Tree, which is a tree structure representing the input read as a series of statement and expression nodes on the tree. If the tree is found to not be valid for the grammar then the parser is to error out and give the user indication on what is wrong with their code. Provided that the tree is valid, the tree is to be written out to a file for evaluation.

To accomplish this various languages are allowed to be used: C, C++, Java, or Python. The program must be written to language standards, and include appropriate comments throughout the source code. The program will read in source from a source file, and output the finished product to a new file.

Summary

To accomplish this problem's objective the first thing was to create the pieces of the parser that would come together. The first of these pieces is the TokenStream. This abstracts away the actual scanner from parsing. Instead the parser only sees the current and next token, and can only advance this token stream. This stream also provides various helper functions to ensure the current and next tokens are of a certain type. This abstraction will prove very powerful in parsing as will be discussed soon.

The next piece was the IdentifierTable. This class provides a repository of known identifiers and information about them such as type and scope. One of the most powerful parts of this table is how it handles scope. A stack is implemented to represent the current scope. As the parser parses scopes are added and the the stack grows. Once these scopes end the stack shrinks. This allows identifiers to know what scope they belong to and allows multiple identifiers of the same name to be created in separate scopes.

The last piece of the parser that was created was the sub-parsers. These sub-parsers are responsible for parsing a single type of statement or expression. For instance, Procedure statements are entirely handled by the ProcedureStatementParser. These parsers take in the main parser and token stream and parse out the single statement or expression they care about and then return. The primary parser maps certain token to these statements such that when the token is read it knows which sub-parser to call. For example, if the parser encounters the "procedure" token, then it will lookup the sub-parser in the mapping and call the sub-parser to parse the procedure statement. This is very helpful as it seperates all the specific code needed to parse the many

statement and expressions into their own classes and functions.

With all of these pieces, the parser was able to be created. On initialization all of the sub-parser mappings are initialized, and the identifier table is created with a global scope. The token stream is also initialized such that the current and next tokens are read.

The parser itself is a recursive descent parser implementing Pratt parsing. This allows it to easily deal with things such as operator precedence and sub-statements/expressions. For instance: When parsing a procedure statement, the formal parameter declarations are handled by a recursive call to parse. This way the procedure parser does not need to know how to parse these declarations, just that they should exist. The parser is also constructing the syntax tree from these parsed statements and expressions at this time.

When an error occurs, such as an unexpected token or statement parsed, the parser will stop parsing any further. Errors are bubbled up to the top level of the parser and printed for the user to see and handle.

The language used to implement this project is Java. No external libraries or APIs were used to create this project. Only the standard library implementation provided in OpenJDK-11 is used to implement the parser. The parser itself is located in the Parser.java file in src/epps/ada/lexing/parsing/. All related source files to the parser itself are found in src/epps/ada/lexing/parsing/ and stc/epps/ada/lexing/errors

To run the program as is execute the following commands in the workspace:

javac -d ./build -cp ./src/epps/ada ./src/epps/ada/Ada.java
java -cp ./build Ada input.adb

Detailed Description

To begin, the grammar recognized currently by the scanner/parser is as follows:

```
upper_case_letter ::= "A" | "B" | "C" | "...etc..." | "Z"

lower_case_letter ::= "a" | "b" | "c" | "...etc..." | "z"

identifier_letter ::= upper_case_letter | lower_case_letter

digit ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8
    " | "9"

identifier ::= identifier_letter { [ "_" ] (
    identifier_letter | digit ) }

numeral ::= digit { [ "_" ] digit }

graphic_character ::= identifier_letter | digit

character_literal ::= "'" graphic_character "'"

string_element ::= "pair of quotation mark" |
    graphic_character

string_literal ::= "quotation mark" { string_element } "
    quotation mark"

direct_name ::= identifier | operator_symbol

name ::= direct_name | character_literal

parent_unit_name ::= name

defining_identifier ::= identifier

defining_program_unit_name ::= [ parent_unit_name "." ]
    defining_identifier

subtype_mark ::= name

defining_operator_symbol ::= operator_symbol

defining_designator ::= defining_program_unit_name |
    defining_operator_symbol

subprogram_specification ::= ( "procedure"
    defining_program_unit_name [ formal_part ] )
```

```
subprogram_declaration ::= subprogram_specification ";"

parameter_association ::= ( expression | name )

actual_parameter_part ::= "(" parameter_association { ","
    parameter_association } ")"

procedure_name ::= name

procedure_call_statement ::= ( procedure_name | prefix ) [
    actual_parameter_part ] ";"

assignment_statement ::= variable_name ":=" expression ";"

expression ::= relation

relation ::=
      ( simple_expression [
      ( "=" | "/=" | "<" | "<=" | ">" | ">=" )
      simple_expression ] )

simple_expression ::= [ ( "+" | "-" ) ] term
      { ( "+" | "-" | "&" ) term }

term ::= factor { ( "*" | "/"  ) factor }

factor ::= primary

primary ::= numeric_literal | string_literal | name | ( "("
    expression ")" )

formal_part ::= "(" parameter_specification { ";"
    parameter_specification } ")"

parameter_specification ::=
      defining_identifier_list ":"
      ( ( subtype_mark ) )
      [ ":=" default_expression ]

subtype_mark ::= name
```

This subset of grammar is currently designed to be able to parse very simple programs made of procedures, declarations, and assignements such as:

```
 procedure test
(
    paramOne : Integer := 10;
    paramTwo : Integer := 10;
    paramThree : Integer
) is
    paramFive : Integer := 4;
begin

    paramOne := paramOne * 2;
    paramFour := paramOne + paramTwo;
    paramFive := (paramOne + 5) * paramThree;

end test;
```

TokenStream.java

The first piece of the parser to examine is the TokenStream. As stated this is a stream that abstracts the scanner output into an easy to use form. Users of this class simply have to worry about the current and next tokens provided by the TokenStream. In addition, the token stream can be shared across multiple users and passed up and down the parsing call stack so that the state of the stream is always correct. This class also provides many helper methods such as "currentMustBe" in order to assert that the current token is a token of a certain pattern.

Below is the class definition for the TokenStream.java.

```
/**
 * A TokenStream represents a stream of parsed tokens from a
    scanner.
 */
public class TokenStream {

    /**
     * The scanner the TokenStream reads from
     */
    private Scanner tokenScanner;

    /**
     * The currently processed token
     */
    private Token current;

    /**
     * The next token to be processed;
     */
    private Token next;

    /**
     * Creates a new TokenStream that reads from the
    provided scanner
     */
    public TokenStream(Scanner tokenScanner) throws
    ParsingException {
        this.tokenScanner = tokenScanner;

        this.advance();
        this.advance();
    }


    /**
     * Advances the token stream by one token
```

```
     */
    public void advance() throws ParsingException {
        this.current = this.next;
        this.next = this.tokenScanner.nextToken();
    }

    /**
     * Get the pattern of the current token
     * @return The pattern of the current token
     */
    public TokenPattern currentPattern() {
        return this.current != null ? this.current.
    getPattern() : TokenPattern.EOF;
    }

    /**
     * Get the pattern of the next token
     * @return The pattern of the next token
     */
    public TokenPattern nextPattern() {
        return this.next != null ? this.next.getPattern() :
    TokenPattern.EOF;
    }

    /**
     * Checks to see if the current token matches the
    pattern provided
     * @param pattern The pattern to check against
     * @return Does the current token match the provided
    pattern
     */
    public boolean isCurrent(TokenPattern pattern) {
        return this.currentPattern() == pattern;
    }


     /**
     * Checks to see if the next token matches the pattern
    provided
     * @param pattern The pattern to check against
     * @return Does the next token match the provided
    pattern
     */
    public boolean isNext(TokenPattern pattern) {
        return this.nextPattern() == pattern;
    }

    /**
     * Requires that the current token matches the provided
    pattern.
```

```
     * If it does not then an ExpectedSymbolException is
    thrown.
     */
    public void currentMustBe(TokenPattern pattern) throws
    ExpectedSymbolException {
        if (!this.isCurrent(pattern)) {
            throw new ExpectedSymbolException(pattern, this.
    currentLiteral());
        }
    }

    /**
     * Requires that the next token matches the provided
    pattern.
     * If it does not then an ExpectedSymbolException is
    thrown.
     */
    public void nextMustBe(TokenPattern pattern) throws
    ExpectedSymbolException {
        if (!this.isNext(pattern)) {
            throw new ExpectedSymbolException(pattern, this.
    nextLiteral());
        }
    }

    /**
     * Requires that the current token DOES NOT match the
    provided pattern.
     * If it does not then an UnexpectedSymbolException is
    thrown.
     */
    public void currentMustNotBe(TokenPattern pattern)
    throws UnexpectedSymbolException {
        if (this.isCurrent(pattern)) {
            throw new UnexpectedSymbolException(this.
    currentLiteral());
        }
    }

    /**
     * Requires that the next token DOES NOT match the
    provided pattern.
     * If it does not then an UnexpectedSymbolException is
    thrown.
     */
    public void nextMustNotBe(TokenPattern pattern) throws
    UnexpectedSymbolException {
        if (this.isNext(pattern)) {
            throw new UnexpectedSymbolException(this.
    nextLiteral());
```

```
        }
    }

    /**
     * @return The literal of the current token
     */
    public String currentLiteral() {
        return this.current != null ? this.current.
    getLiteral() : "";
    }

    /**
     * @return The literal of the next token
     */
    public String nextLiteral() {
        return this.next != null ? this.next.getLiteral() :
    "";
    }

}
```

IdentifierTable.java

The next piece of the parser is the IdentifierTable. This table keeps track of identifiers and information about them, such as scope and type. This information is simply stored in a list. The powerful part of this table is the use of scopes to determine if an identifier is already declared in the current scope. When the parser enters a new scope the stack is pushed to, and then when it exits that scope the stack is popped from. This allows multiple identifiers of the same name to be defined as long as they exists in separate scopes.

Below is the definition for the IdentifierTable.java followed by the definition for Scope.java

```
public class IdentifierTable {

    // Temporary declaration of these types here for easy
    access;
    public static final Type BUILT_IN_TYPE_PROCEDURE = new
    Type("procedure");
    public static final Type BUILT_IN_TYPE_INTEGER = new
    Type("Integer");
    public static final Type BUILT_IN_TYPE_FLOAT = new Type(
    "Float");

    /**
     * The identifiers that have been declared;
     */
    private List<IdentifierInformation> identifierInfos;

    /**
     * A set of all the types that have been defined;
     */
    private Set<Type> knownTypes;

    /**
     * Represent the scope as a stack that can be pushed and
     popped. When the stack
     * is empty, we are in global scope. Example stack would
     be: procedure "hello"
     * -> for loop -> for loop. This would allow variables
    local to only the inner
     * for loop, outer for loop, the procedure "hello", or
    globally.
     */
    private Scope currentScope;

    public IdentifierTable() {
        this.identifierInfos = new ArrayList<>();
        this.knownTypes = new HashSet<>();
```

```
        this.currentScope = new Scope();

        // Add built-in types
        this.addKnownType(BUILT_IN_TYPE_PROCEDURE);
        this.addKnownType(BUILT_IN_TYPE_INTEGER);
        this.addKnownType(BUILT_IN_TYPE_FLOAT);
    }

    /**
     * Attempts to declare the identifier in the current
    scope as the type provided.
     *
     * @param identifier - The identifier to declare
     * @param type       - The type of the identifier
     * @throws ParsingException - Thrown if the identifier
    cannot be declared in
     *                           this scope or with this type
     */
    public void declareIdentifier(IdentifierExpression
    identifier, Type type) throws ParsingException {

        // If the type doesnt exist then we cant declare
    this identifier
        if (!doesTypeExist(type)) {
            throw new UnknownTypeException(identifier, type)
    ;
        }

        // If the identifier is already declared in this
    scope then we cant declare this
        // identifier, again
        if (identifierDeclaredInScopeAlready(identifier)) {
            throw new IdentifierRedeclarationException(
    identifier);
        }

        IdentifierInformation identifierInformation = new
    IdentifierInformation(identifier, type, currentScope);
        this.identifierInfos.add(identifierInformation);
    }

    /**
     * Adds a type to the known types set
     *
     * @param type The type to add
     */
    public void addKnownType(Type type) {
        this.knownTypes.add(type);
    }
```

```
/**
 * Is the type provided a known type
 *
 * @param type The type to check
 * @return Is the type provided a known type
 */
private boolean doesTypeExist(Type type) {
    return this.knownTypes.stream().filter(type::equals)
.findAny().isPresent();
}

/**
 * Checks to see if there exists any declared
identifiers that equal the passed
 * in identifier tith scopes that are included in the
current scope.
 *
 * @param identifier The identifier to check
 * @return Is this identifier declared already in this
scope
 */
private boolean identifierDeclaredInScopeAlready(
IdentifierExpression identifier) {
    return this.identifierInfos.stream().filter(
identifierInfo -> identifierInfo.getIdentifier().equals(
identifier))
            .map(IdentifierInformation::getScope).
anyMatch(scope -> scope.isIncludedIn(currentScope));
}

/**
 * Push the provided scope onto the current scope.
 *
 * @param scope The scope to push
 */
public void pushScope(IdentifierExpression scope) {
    this.currentScope.pushScope(scope);
}

/**
 * Pop the top layer of the current scope.
 */
public void popScope() {
    this.currentScope.popScope();
}

/**
 * IdentifierInformation is simply a class to hold
various information about an
 * identifier
```

```
     */
    private class IdentifierInformation {
        private IdentifierExpression identifier;
        private Type type;
        private Scope scope;

        private IdentifierInformation(IdentifierExpression
    identifier, Type type, Scope scope) {
            this.identifier = identifier;
            this.type = type;
            this.scope = scope.copy();
        }

        public IdentifierExpression getIdentifier() {
            return identifier;
        }

        public Type getType() {
            return type;
        }

        public Scope getScope() {
            return scope;
        }
    }

    /**
 * Represent the scope as a stack that can be pushed and
    popped. When the stack
 * is empty, we are in global scope. Example stack would be:
     procedure "hello"
 * -> for loop -> for loop. This would allow variables local
     to only the inner
 * for loop, outer for loop, the procedure "hello", or
    globally.
 */
public class Scope {

    private Stack<IdentifierExpression> stack;

    public Scope() {
        this.stack = new Stack<>();
    }

    /**
     * Checks to see if this scope is either equal to or a
    parent scope of the
     * scope. A global scope is included in all other scopes
    .
     */
```

```
public boolean isIncludedIn(Scope otherScope) {
    // Special case: If the stack is empty then this is
a global scope.
    // A global scope is implicitity included in all
other scopes.
    if (this.stack.size() == 0) {
        return true;
    }

    // If the other scope is smaller then its impossible
 to be included in it.
    if (this.stack.size() > otherScope.stack.size()) {
        return false;
    }

    // Since we know this scope is always smaller or
equal to the other scope,
    // we will never go out of bounds here.
    for (int i = 0; i < this.stack.size(); i++) {
        if (!this.stack.get(i).equals(otherScope.stack.
get(i))) {
            return false;
        }
    }

    return true;
}

@Override
public boolean equals(Object obj) {
    if (obj == null || !(obj instanceof Scope)) {
        return false;
    }

    Scope otherScope = (Scope) obj;

    if (this.stack.size() != otherScope.stack.size()) {
        return false;
    }

    for (int i = 0; i < this.stack.size(); i++) {
        if (!this.stack.get(i).equals(otherScope.stack.
get(i))) {
            return false;
        }
    }

    return true;
}
```

```
    /**
     * Creates a new Scope that has the same scope as this
    one.
     *
     * @return the newly created Scope
     */
    public Scope copy() {
        Scope copy = new Scope();
        for (IdentifierExpression identifierExpression :
    this.stack) {
            copy.stack.push(identifierExpression);
        }

        return copy;
    }

    /**
     * Adds a another layer to the scope.
     *
     * @param identifierExpression The layer to add to the
    scope
     */
    public void pushScope(IdentifierExpression
    identifierExpression) {
        this.stack.push(identifierExpression);
    }

    /**
     * Removes the top most layer from the scope;
     */
    public void popScope() {
        this.stack.pop();
    }

}
}
```

ProcedureStatementParser.java

The last piece needed to build this parser is the sub-parsers mentioned before. These each handle a single statement or expression. Each will read from the token stream and create their respective node in the tree.

The definition of one of these parsers, ProcedureStatementParser.java is listed below

```
public class ProcedureStatementParser implements
    StatementParser<ProcedureStatement> {

    @Override
    public ProcedureStatement parse(Parser parser,
    TokenStream tokenStream) throws ParsingException {
        // We are currently at the "procedure" token, ensure
     that the next token is a
        // identifier
        // so that the procedure has a name.
        tokenStream.nextMustBe(TokenPattern.IDENTIFIER);
        tokenStream.advance();

        // Parse out the next expression
        Expression expression = parser.parseExpression(0);

        // If the expression isnt an identifier for some
    reason, throw an error
        if (!(expression instanceof IdentifierExpression)) {
            throw new InvalidExpressionException(expression)
    ;
        }

        IdentifierExpression procedureName = (
    IdentifierExpression) expression;

        // Push this scope onto the identifier table
        parser.getIdentifierTable().pushScope(procedureName)
    ;

        // Declare the name identifier as a procedure type
        parser.getIdentifierTable().declareIdentifier(
    procedureName, IdentifierTable.BUILT_IN_TYPE_PROCEDURE);

        tokenStream.advance();

        // Parse out the list of formal parameters in the
    parentheses, if there are any
        List<DeclarationStatement> parameters = new
    ArrayList<>();
```

```
    if (tokenStream.isCurrent(TokenPattern.
SYMBOL_PAREN_LEFT)) {
        parameters = parseDeclarationStatementList(
parser, tokenStream);
        // The last formal parameter must not end with a
 semicolon
        tokenStream.currentMustNotBe(TokenPattern.
SYMBOL_SEMICOLON);
        tokenStream.currentMustBe(TokenPattern.
SYMBOL_PAREN_RIGHT);
        tokenStream.advance();
    }

    // Start parsing the "is" block
    tokenStream.currentMustBe(TokenPattern.KEYWORD_IS);

    // If we arn't at the begin block then we are
declaring local variables
    List<DeclarationStatement> locals = new ArrayList
<>();
    if (!tokenStream.isNext(TokenPattern.KEYWORD_BEGIN))
 {
        locals = parseDeclarationStatementList(parser,
tokenStream);
        tokenStream.currentMustBe(TokenPattern.
SYMBOL_SEMICOLON);
    }

    tokenStream.advance();

    // Start parsing the "begin" block
    tokenStream.currentMustBe(TokenPattern.KEYWORD_BEGIN
);
    BeginStatement begin = (BeginStatement) parser.
parseStatement();

    // If the identifier at the end of the begin block
is not the same as the
    // procedure's identifier
    // then throw an error.
    if (!begin.getBelongsTo().equals(procedureName)) {
        throw new ParsingException("expected to find " +
 procedureName + ", but found " + begin.getBelongsTo());
    }

    // Entire statement must end with a semi-colon
    tokenStream.currentMustBe(TokenPattern.
SYMBOL_SEMICOLON);
    tokenStream.advance();
```

```
        // Pop the procedure scope off
        parser.getIdentifierTable().popScope();
        return new ProcedureStatement(procedureName,
parameters, locals, begin);
    }

    /**
     * Parses out a list of declaration statements
     *
     * @param parser      The parser to use
     * @param tokenStream The token stream to use
     * @return A list of declaration statements
     * @throws ParsingException Thrown if any errors occur
while parsing the
     *                          declaration statements
     */
    private List<DeclarationStatement>
parseDeclarationStatementList(Parser parser, TokenStream
tokenStream)
            throws ParsingException {

        List<DeclarationStatement> declarations = new
ArrayList<>();
        do {
            tokenStream.advance();

            // Attempt to parse a declaration statement
            try {
                Statement statement = parser.parseStatement
();
                if (statement instanceof
DeclarationStatement) {
                    declarations.add((DeclarationStatement)
statement);
                } else if (statement != null) {
                    // If the statement is not a declaration
 statement we are in an error state
                    throw new UnexpectedStatementException(
statement);
                }
            } catch (ExpectedSymbolException e) {
                // This is a special case such that the last
 declaration statement does not require a semi-colon
                if (!tokenStream.isCurrent(TokenPattern.
SYMBOL_PAREN_RIGHT)) {
                    throw e;
                }
            }
            // Continue as long as we find a semi-colon
ending the current declaration statement
```

```
            // followed by an identifier starting the next
    one
        } while (tokenStream.isCurrent(TokenPattern.
    SYMBOL_SEMICOLON) && tokenStream.isNext(TokenPattern.
    IDENTIFIER));

        return declarations;
    }

    @Override
    public boolean allowedAtTopLevel() {
        return true;
    }

    @Override
    public boolean allowedBelowTopLevel() {
        return false;
    }
}

public class ProcedureStatement implements Statement {

    private IdentifierExpression name;
    private List<DeclarationStatement> parameters;
    private List<DeclarationStatement> locals;
    private BeginStatement begin;

    public ProcedureStatement(IdentifierExpression name,
    List<DeclarationStatement> parameters,
            List<DeclarationStatement> locals,
    BeginStatement begin) {
        this.name = name;
        this.parameters = parameters;
        this.locals = locals;
        this.begin = begin;
    }

    @Override
    public String toString() {
        StringBuilder builder = new StringBuilder();
        builder.append("procedure ");
        builder.append(name);
        if (parameters.size() > 0) {
            builder.append(" (\n");
            parameters.forEach(p -> {
                builder.append(p);
                builder.append("\n");
            });
            builder.append(")");
        }
```

```
            if (begin != null) {
                builder.append(" is \n");
                locals.forEach(l -> {
                    builder.append(l);
                    builder.append("\n");
                });
                builder.append(begin);
            }
            builder.append(";");
            return builder.toString();
        }

        @Override
        public void addChildren(List<Node> children) {
            children.add(name);
            children.addAll(parameters);
            children.addAll(locals);
            children.add(begin);
        }

    }
```

Parser.java

Now that we have all the pieces created we can define the actual parser. As mentioned, this is a recursive descent parser implementing Pratt parsing. This method of parsing allows easy handling of typically hard problems such as operator grouping and precedence. The main portion of this parser is in the actual mapping of tokens to sub-parsers. The only real work the parser does is this mapping. All the actual parsing is done through the sub-parsers. The parser will continue parsing until either an error occurs or the end of the stream.

Below is the definition of Parser.java

```
/**
 * The Parser is parser based on Pratt Parsing. It is a
   recursive descent parser.
 * The parser will read from a stream of tokens, and match
   the token with a given set of sub-parsers.
 * These sub-parsers will then parse out the correct
   statement or expression for the token.
 * Once the token stream is finished the parser will have a
   completed syntax tree of the program.
 * If any error occurs the parser will stop parsing on the
   token and return.
 */
public class Parser {

    /**
     * The token stream the parser reads from
     */
    private TokenStream tokenStream;

    /**
    * The identifier table the parser uses to store
    identifier information
    */
    private IdentifierTable IdentifierTable;

    /**
    * Mapping of Token Patterns to parsers that will parse
    out statements
    */
    private Map<TokenPattern, StatementParser<?>>
    statementParsers;

    /**
    * Mapping of Token Patterns to parsers that will parse
    out most expressions
    */
```

```
    private Map<TokenPattern, ExpressionParser<?>>
expressionParsers;

    /**
     * Mapping of Token Patterns to parsers that will parse
out infixed expressions
     */
    private Map<TokenPattern, InfixExpressionParser>
infixParsers;

    /**
     * Mapping of Token Patterns to operator precedences.
Only contains patterns with precendence.
     * All other tokens are assumed to have the highest
precedence.
     */
    private Map<TokenPattern, Integer> precedences;

    /**
     * Determines if we are currently in the global scope.
     * Used to prevent things such as procedure statements
being defined inside
     * of other procedure statements.
     */
    private boolean atTopLevel;

    /**
     * Create a new Parser and fill its parser mappings with
 values
     * @param tokenStream The token stream this parser will
read from
     */
    public Parser(TokenStream tokenStream) {
        this.tokenStream = tokenStream;
        this.IdentifierTable = new IdentifierTable();

        this.statementParsers = new HashMap<>();
        this.statementParsers.put(TokenPattern.IDENTIFIER,
new IdentifierStatementParser());
        this.statementParsers.put(TokenPattern.
KEYWORD_PROCEDURE, new ProcedureStatementParser());
        this.statementParsers.put(TokenPattern.KEYWORD_BEGIN
, new BeginStatementParser());

        this.expressionParsers = new HashMap<>();
        this.expressionParsers.put(TokenPattern.IDENTIFIER,
(p, s) -> {
            return new IdentifierExpression(s.currentLiteral
());
        });
```

```
        this.expressionParsers.put(TokenPattern.NUMERAL, (p,
 s) -> {
            return new NumeralExpression(Float.parseFloat(s.
currentLiteral().replaceAll("_", "")));
        });
        this.expressionParsers.put(TokenPattern.SYMBOL_MINUS
, new PrefixExpressionParser());
        this.expressionParsers.put(TokenPattern.
SYMBOL_PAREN_LEFT, new GroupedExpressionParser());

        InfixExpressionParser infixExpressionParser = new
InfixExpressionParser();
        this.infixParsers = new HashMap<>();
        this.infixParsers.put(TokenPattern.SYMBOL_PLUS,
infixExpressionParser);
        this.infixParsers.put(TokenPattern.SYMBOL_MINUS,
infixExpressionParser);
        this.infixParsers.put(TokenPattern.SYMBOL_DIVISION,
infixExpressionParser);
        this.infixParsers.put(TokenPattern.
SYMBOL_MULTIPLICATION, infixExpressionParser);
        this.infixParsers.put(TokenPattern.SYMBOL_EQUALITY,
infixExpressionParser);
        this.infixParsers.put(TokenPattern.SYMBOL_INEQUALITY
, infixExpressionParser);
        this.infixParsers.put(TokenPattern.SYMBOL_LESSTHAN,
infixExpressionParser);
        this.infixParsers.put(TokenPattern.
SYMBOL_GREATERTHAN, infixExpressionParser);

        this.precedences = new HashMap<>();
        this.precedences.put(TokenPattern.SYMBOL_EQUALITY,
1);
        this.precedences.put(TokenPattern.SYMBOL_INEQUALITY,
 1);
        this.precedences.put(TokenPattern.SYMBOL_LESSTHAN,
2);
        this.precedences.put(TokenPattern.SYMBOL_GREATERTHAN
, 2);
        this.precedences.put(TokenPattern.SYMBOL_PLUS, 3);
        this.precedences.put(TokenPattern.SYMBOL_MINUS, 3);
        this.precedences.put(TokenPattern.SYMBOL_DIVISION,
4);
        this.precedences.put(TokenPattern.
SYMBOL_MULTIPLICATION, 4);

    }

    /**
     * Parse out the entire program. Will run until either
```

```
EOF is reached or an error occurs
     * @return The parsed Syntax Tree
     * @throws RuntimeException Thrown when any error occurs
    while parsing
     */
    public Program parse() throws RuntimeException {

        Program program = new Program();

        try {
            // Parse until EOF
            while (!this.tokenStream.isCurrent(TokenPattern.
EOF)) {
                // As this is recursive parsing, if we are
here we are in the global scope
                this.atTopLevel = true;
                Statement statement = this.parseStatement();
                if (statement != null) {
                    program.addStatement(statement);
                }
            }
        } catch (Exception e) {
            throw new RuntimeException(e);
        }

        return program;
    }

    /**
     * Parses the next statement found.
     * @return The next statement that was parsed
     * @throws ParsingException Thrown if any error occurs
    while parsing the statement
     */
    public Statement parseStatement() throws
    ParsingException {
        // Get the parser for the next statement
        StatementParser<?> statementParser = this.
    statementParsers.get(this.tokenStream.currentPattern());

        if (statementParser != null) {
            // If we in global scope and the statement is
    not allowed in global scope (I.E. Declarations) or
            // if we are not in global scope and the
    statement requires global scope (I.E. Procedures)
            // then throw a new error.
            if ((atTopLevel && !statementParser.
    allowedAtTopLevel())
                    || (!atTopLevel && !statementParser.
    allowedBelowTopLevel())) {
```

```
            throw new ParsingException("statement not
allowed at this level");
        }

        // If we are at this point then we are not in
global scope anymore
        this.atTopLevel = false;

        // Actually parse the next statement
        Statement statement = statementParser.parse(this
, this.tokenStream);
        if (statement == null) {
            throw new ParsingException("invalid
statement");
        }
        return statement;
    }

    return null;
}

/**
 * Parses out the next expression using the defined
expression parsers.
 * @param precedence The current precedence of the
expression being parsed
 * @return The parsed expression
 * @throws ParsingException Thrown if any error occurs
while parsing the expression
 */
public Expression parseExpression(int precedence) throws
 ParsingException {
    // Get the expression parser for this
    ExpressionParser<?> prefixParser = this.
expressionParsers.get(this.tokenStream.currentPattern());
    if (prefixParser == null) {
        return null;
    }

    // Parse the expression
    Expression leftExpression = prefixParser.parse(this,
 this.tokenStream);
    // If the expression is not a semicolon and the next
 expression has a lower precedence than current this is a
 infix expression
    while (!this.tokenStream.isNext(TokenPattern.
SYMBOL_SEMICOLON) && precedence < this.nextPrecedence())
{
        // Get the infix parser for the next token
        InfixExpressionParser infixParser = this.
```

```
infixParsers.get(this.tokenStream.nextPattern());
        if (infixParser == null) {
            return leftExpression;
        }

        this.tokenStream.advance();
        // Parse the infix expression
        leftExpression = infixParser.parse(this, this.
tokenStream, leftExpression);
    }

    return leftExpression;
}

/**
 * Parses an list of expressions seperated by commas
 * @return The list of expressions
 * @throws ParsingException Thrown if any expression
errors while parsing
 */
public List<Expression> parseExpressionList() throws
ParsingException {
    List<Expression> expressions = new ArrayList<>();

    do {
        // If the token is a comma then we advance to
the next expression
        if (this.tokenStream.isCurrent(TokenPattern.
SYMBOL_COMMA)) {
            this.tokenStream.advance();
        }
        // Parse the next expression
        Expression expression = this.parseExpression(0);
        if (expression != null) {
            expressions.add(expression);
        }
        this.tokenStream.advance();

        // Continue until the token is not a comma
anymore
    } while (this.tokenStream.isCurrent(TokenPattern.
SYMBOL_COMMA));

    return expressions;
}

/**
 * Looks up the precedence for the current token, 0 if
the token is not in the mapping
 */
```

```
    public int currentPrecedence() {
        return this.precedences.getOrDefault(this.
    tokenStream.currentPattern(), 0);
    }


    /**
     * Looks up the precedence for the next token, 0 if the
    token is not in the mapping
     */
    private int nextPrecedence() {
        return this.precedences.getOrDefault(this.
    tokenStream.nextPattern(), 0);
    }

    public IdentifierTable getIdentifierTable() {
        return IdentifierTable;
    }
}
```

Results

The results of this parser are quite great. Here is the syntax tree created from parsing the example code listed at the top of the detailed description:

```
- Program
    - ProcedureStatement
        - IdentifierExpression
        - DeclarationStatement
            - IdentifierExpression
            - Type
            - NumeralExpression
        - DeclarationStatement
            - IdentifierExpression
            - Type
            - NumeralExpression
        - DeclarationStatement
            - IdentifierExpression
            - Type
        - DeclarationStatement
            - IdentifierExpression
            - Type
            - NumeralExpression
        - BeginStatement
            - AssignmentStatement
                - IdentifierExpression
                - InfixExpression
                    - IdentifierExpression
                    - OperatorExpression
                    - NumeralExpression
            - AssignmentStatement
                - IdentifierExpression
                - InfixExpression
                    - IdentifierExpression
                    - OperatorExpression
                    - IdentifierExpression
            - IdentifierExpression
    - ProcedureStatement
        - IdentifierExpression
        - DeclarationStatement
            - IdentifierExpression
            - Type
        - BeginStatement
            - IdentifierExpression
    - ProcedureStatement
        - IdentifierExpression
        - BeginStatement
            - IdentifierExpression
```

Limitations and Possible Improvements

The main limitation of the parser currently is that there is no way to call other procedure or create functions. This means the program is limited to procedures, declarations, assignments, and basic operator expressions. One way to improve this system would be to expand the functionality to be able to call other procedures or functions.

Some issues are also not checked, such as if the expression on the RHS of the assignment statement matches the type of the LHS. These issues need to be addressed in order to have a functioning program.

The other limitation that needs to be fixed is that the parser provides no indication on line number or column number that an error occurs on. For small programs this is not a big deal, but going forward this should be addressed.

Conclusion

Overall, the parser is working as detailed. The parser is written in a way that is very easily extensible, and will be extended going into the final part of this project. I am very pleased with how the overall design of the parser is done. It feels very natural and easy to add additional sub-parsers to the system. As each parser is only responsible for its children it is not needed to have complex management of the token stream. The issues outlined above will need to be addressed before continuing onto the interpreter, but I do not think it will take much time to address them.

Given more time I really would like to expand this parser to include more of the grammar, but am aiming to have procedure and functions working with basic statements while creating the interpreter.

References *Ball, T. (2017). Writing an interpreter in Go. CreateSpace*
Sebesta, R. W. (2019). Concepts of programming languages. NY, NY: Pearson.