Comparing Parser Construction Techniques

Bas van Gijzel
b.m.vangijzel@student.utwente.nl

ABSTRACT

Today most recursive descent parsers are generated by providing grammars and generating parsers according to these grammars. An alternative approach to constructing parsers consists of parser combinators, which do not need a separate step to generate the parser, and furthermore claim to be clear and simple in use. Despite these claimed advantages, parser combinators have not been widely adopted and are rarely actually compared to parser generators.

This paper introduces two of the contemporary implementations, ANTLR and Parsec, along with a novel implementation, called Tinadic Parsing, based on the combination of the techniques used in parser generators and parser combinators. The three implementations are compared and evaluated by the implementation of increasingly extended examples.

Keywords

parser generators, parser combinators, novel parsing technique

1. INTRODUCTION

A common activity within computer science is the analyzing of tokens, called *parsing*. Contemporary parsers are constructed by means of parser generators, and to a lesser extent by parser combinators. Although parser generators are the standard approach to constructing parsers, they are not always the easiest method. Parser combinators have long claimed to be more intuitive and easier in use than their generating counterpart [12, 11, 21].

There are numerous existing parser generator and combinator libraries, including recent and industry strength libraries [26, 40, 22, 35]. The comparative advantages of each library, or even the advantages of the different techniques are still quite unclear. Comparisons between libraries are rarely made, and if made they are between the same techniques, and the comparisons are based on benchmarks [21], not on the ease of use or other criteria. This raises the question:

> How do parser generators and parser combinators compare concerning usability and readability?

This paper will compare three instances of the previously mentioned parsing techniques. Namely ANTLR [26, 27], a popular representative of parser generators, Parsec [21, 22], a popular parser combinator library written in Haskell [28], and finally Tinadic Parsing, a novel approach combining features of both parser generators and combinators. Therefore:

> What are the comparative advantages and disadvantages of ANTLR, Parsec and Tinadic Parsing concerning usability and readability?

By answering this question we will be able to give exploratory results for the main research question. The method of research used in this paper for comparison is to program similar parsers in the three parser construction instances and to compare these parsers on readability and usability. The similar parsers will implement increasingly complex examples containing (arithmetic) expressions. The parser sets will be evaluated after the implementations. All the implementations along with driver classes and test files can be found online [7].

In this section parser generators, parser combinators and a hybrid approach are introduced. In Section 2 we will discuss recent research and some problems. In Section 3 ANTLR, Parsec and Tinadic Parsing are introduced by a simple parsing example. In Section 4 this example will be extended to more advanced expressions containing different operator associativities and precedences. In Section 5 the expressions example is extended with lexing and a possible implementation of the off-side rule is given for each parser implementation. In Section 6 we will answer our research questions and draw conclusions. Finally in Section 7 possible future work and research is laid out.

1.1 Parser Generators

The standard approach for constructing parsers is to use a parser generator. Parser generators such as ANTLR and JavaCC [40], are tools that generate parser code based on a grammar used as input, where the most common format used for describing grammars is EBNF (Extended Backus–Naur Form) [32]. Parser generators can generate parsers that work on pre-processed streams of tokens, which can be generated by a separate lexer, or can integrate a lexer in the input grammar.

1.2 Parser Combinators

A flexible approach, mostly used in functional programming, is to produce parsers by defining primitive parsers and *parser combinators* [21, 13, 3, 17, 34], and using these to construct more complex parsers.

A primitive parser (for example, a parser that parses only one letter) is modeled as a function that takes a sequence of symbols and returns a list of successes [41]. A parser combinator is then modeled as a higher-order function (a function that can take functions as argument(s)). Some examples of common combinators, similar to the EBNF operators, are sequencing, choice and repetition. A parser that takes two letters in sequence is then produced by combining the one letter parser with a sequencing combinator and another one letter parser. This approach of combining primitive parsers by means of parser combinators makes it possible to easily construct more complex parsers in an intendedly EBNF-like notation.

11th Twente Student Conference on IT, Enschede 29th June, 2009

1.3 A Hybrid Approach

In this section a novel approach of constructing parsers is considered. An implementation using this approach is Tinadic Parsing, which has been developed at the University of Twente by Jan Kuper [18]. The implementation is written in Amanda [1], a programming language with syntax very similar to Miranda [37] and Haskell.

This alternative approach is a combination of the techniques used in parser generators and parser combinators. A grammar is considered a *function* that maps a non-terminal to a list of possible terminals/non-terminals. Presenting this function in a lambda notation (a notation for defining anonymous functions) resembles the standard EBNF form of a grammar to a large extent, as can be seen in Listing 4. Hence, a parser generator is a higher-order function that takes a grammar as an argument and delivers a parser for that grammar. While parsing, the functional character of this grammar is exploited. Operators used in describing the grammar are similar to those used in EBNF operators. The definition of these operators resembles parser combinators.

2. RELATED WORK

Recent research on parser generators has seen an increased interest in extensible syntax, namely in the research of Parsing Expressions Grammars (PEGs) [5, 6, 9]. PEGs use an alternative approach inspired by the parser combinator paradigm [5] to describe context-free grammars. Interestingly this use of syntactic predicates and longest-match disambiguation has been implemented in the newer versions of ANTLR [25].

The shortcomings of early parser combinators [11, 2, 41] have been mentioned in several papers [17, 33, 36, 30]. The speed and memory consumption of these libraries were often super-linear in complexity and thus unsuitable for large inputs. Error messages were sometimes absent or only conveyed limited information. The more recent libraries are better suited for larger inputs and resulting error messages have improved. A benchmark has been made by Ljunglöf [23], his thesis contains a thorough comparison of parsing techniques within functional programming on an algorithmic level. No existing libraries are used for this comparison however.

Despite the earlier research on parser combinators, few comparisons have been made between the newer generation of parser combinators and parser generators. Tinadic Parsing is an approach that takes a combination of both parsing techniques and therefore makes it quite suitable for comparison.

3. SIMPLE PARSING

In the following subsections the parser construction implementations will be demonstrated and then explained by implementation of the EBNF grammar in Listing 1.

Listing 1: EBNF grammar for simple expressions

```
expr   ::= factor (op factor)*
op     ::= '+' | '-'
factor ::= number | '(' expr ')'
number ::= ('0' | '1' | ... | '9')+
```

The | stands for choice, * and + are the familiar EBNF operators meaning zero or more and one or more repetitions.

3.1 ANTLR

ANTLR is a parser generator that can generate recursive descent parsers from an EBNF like grammar description. Interpretation and code generation is mostly done by generating intermediate output, namely an abstract syntax tree (AST), and processing that output by using a tree walker. A small example of a (partial) ANTLR lexer/parser grammar implementing the simple expressions is given in Listing 2.

Listing 2: Simple expressions in ANTLR

```
gram   : expr EOF;

expr   : factor (op factor)*;
op     : '+' | '-';
factor : NUMBER  | '(' expr ')';

NUMBER : ('0'..'9')+;
```

This grammar contains both lexer tokens (*NUMBER*) and non-terminals (*gram*, *expr*, *op* and *factor*). The '0'..'9' is just a short hand for the similar EBNF expression. * and + are the standard EBNF operators that denote repetition and *EOF* is the end of file token. After ANTLR processes this grammar it will generate a separate lexer and parser.

3.2 Parsec

As a parser combinator library, Parsec uses a set of primitive parsers which can be made into more complex parsers by combining these primitive parsers with parser combinators. Parsers constructed in Parsec always return a value (a string, a number, etc.), and parsers can thus easily be built as a combined parser/interpreter. The intermediate values or possible error messages are passed through the parsers by use of monadic constructs [21, 42].

The previous expression example is partially worked out in Listing 3. The example purposefully does not return a useful value to mimic the previous example. Only an error message or the value () is returned as final parse result.

Listing 3: Simple expressions in Parsec

```
gram   = expr >> eof

expr   = factor >> many (op >> factor)

op     = string "+" <|> string "-"

factor = many1 digit <|>
         string "(" >> expr >> string ")"
```

Non-terminals (*gram*, *expr*, *op* and *factor*) are parsers defined as combinations of parsers. *Many* and *many1* are equivalent to the * and + operators, <|> is (predictive) choice, *string*, *digit* and *eof* are predefined parsers, the first being able to parse a given string, the second capable of parsing a digit, and the final one tries to parse the end of the file. The results are combined with a monadic >> operator [42], which intuitively reads like this: throw away the return value of the previous parser and try to parse the remaining symbols.

3.3 Tinadic Parsing

Tinadic Parsing is a hybrid approach of parser generators and combinators, the technique uses combinators written in the underlying implementation language Amanda while still retaining an EBNF like notation. In Listing 4 the simple expression example is implemented in the lambda notation explained in section 1.3.

Listing 4: Simple expressions in Tinadic Parsing

```
exprgrammar =
   Expr   -> [[Factor, <*>[ Op, Factor]]]

 | Op     -> [[Check (member "+-")]]

 | Factor -> [[<+>[digit]]
             ,[Token '(', Expr, Token ')']]

digit = Check (member "0123456789")
```

Exprgrammar is just the name of the function the grammar represents, the non-terminals (*Expr*, *Op*, *Factor*, *Digit*) and terminals (*op* and *digit*) are just data constructors. The EBNF-like operators (<+> and <*>) and *Check* and *Token* are also data constructors of the same type, but they are reserved constructors used to direct the parse function. Choice is implicitly implemented by the use of a list of lists as can be seen in the definition of Factor.

3.4 Conclusions

The simple expressions example already illustrates the differences in syntax of the three implementations; ANTLR has a notation closest to EBNF, Parsec uses a functional notation and Tinadic Parsing uses lambda notation with a large amount of square brackets and therefore loses a bit of clarity. An unintuitive necessary addition for ANTLR and Parsec is the explicit matching on the end of the file. Finally, the Parsec example is a bit contrived due to the explicit throwing away of the parse results; a combined parser/interpreter would result in a clearer and shorter implementation.

4. ARITHMETIC EXPRESSIONS

A recurring problem in parsing, is the parsing of expressions with different precedence levels and associativities. For example, multiplication has precedence over addition, and ^ is a right associative operator. This information can be encoded in (at least) two different ways, as described in the next subsections. After that we will show the benefits of both approaches by showing parser implementations in the aforementioned parser techniques.

The parsers constructed all have different types of results. The default output of ANTLR is an abstract syntax tree, Tinadic Parsing returns a concrete syntax tree and Parsec's default output is direct interpretation of the parse result. Again we will demonstrate the three approaches by providing interpreters for all three techniques.

Implicit Associativity

Although expressions with priorities are quite common, they are not naturally described through the standard formalism, EBNF. To describe n different precedence levels in EBNF, we would need to add n+1 non-terminals to the grammar. Furthermore, associativity of operators has to be encoded through left recursion for left associative operators (left recursion has to be rewritten in most parser generators), and right recursion for right associative operators [10]. An EBNF formalizing arithmetic expressions can be seen in Listing 5.

Listing 5: EBNF of Expressions

```
expr   ::= expr addop term | term
term   ::= term multop pow | pow
pow    ::= factor (expop pow)?
factor ::= number | '(' expr ')'

addop  ::= '+' | '-'
multop ::= '*' | '/'
expop  ::= '^'

number ::= ('0' | '1' | ... | '9')+
```

For illustration, + and * are assumed to be left associative.

Explicit Associativity

Instead of only using an EBNF like formalism, we can use a table or list to ease the encoding of associativities and precedences of operators. An example of this approach can be seen in Listing 6. Although these techniques for describing precedence and associativity since long existed and used in the past [29, 10, 15], they are rarely implemented in modern conventional approaches to parsing, not even in popular parser construction software such as ANTLR [25]. This partly due to the easier translation into corresponding parsers in dynamical and functional languages [20].

Listing 6: EBNF with precedences and associativities

```
Left associative: +, -, *, /
Right associative: ^
Priority 1: +,-
Priority 2: *, /
Priority 3: ^

expr   ::= factor (op factor)*
factor ::= number | '(' expr ')'

op     ::= '+' | '-' | '*' | '/' | '^'

number ::= ('0' | '1' | ... | '9')+
```

As shown, Listing 6 encodes associativity and precedence by explicitly stating them (instead of implicit encoding in the grammar). A higher number corresponds to a higher priority.

4.1 ANTLR

ANTLR defines operator precedence and right associativity similar to the standard EBNF constructs. Left associativity has to be encoded differently because of the lack of support of left recursion. ANTLR partially solves this problem by defaulting associative operators to left associative. Thus if an EBNF with already implicitly encoded operator associativity and precedence is available, left recursion has to be removed, but after that further translation to an ANTLR parser grammar is trivial [25].

To interpret the ANTLR parse results, the grammar is extended with tree annotations which will make the parser output an AST. The ANTLR grammar with tree annotations is shown in Listing 7.

Listing 7: Parsing Expressions in ANTLR

```
gram   : expr EOF!;

expr   : term (addop^ term)*;
term   : pow (multop^ pow)*;
pow    : factor (expop^ pow)?;
factor : NUMBER | '('! expr ')'!;

addop  : '+' | '-';
multop : '*' | '\';
expop  : '^';

NUMBER : ('0'..'9')+;
```

The bang (*!*) operator indicates tokens that can be left out as an AST node, ^ results in a parent node with the other nodes as children.

After processing arithmetic expressions with the above parser grammar, an AST is generated. This AST can subsequently be processed by using a tree walker, generated from an ANTLR tree grammar. A tree walker walks the AST by following defined patterns of nodes, and executes corresponding actions. Actions can be in any language supported by ANTLR.

In Listing 8, the default ANTLR action programming language, Java [8], is used.

Listing 8: Interpreting Expressions in ANTLR

```
gram returns [int value]
 : expr {$value=$expr.value;};

expr returns [int value]
 : ^('+' a=expr b=expr)
     {$value = a + b;}

 | ^('-' a=expr b=expr)
     {$value = a - b;}

 | ^('*' a=expr b=expr)
     {$value = a * b;}

 | ^('/' a=expr b=expr)
     {$value = a / b;}

 | ^('^' a=expr b=expr)
     {$value = (int)Math.pow(a,b);}

 | NUMBER
     {$value =
      Integer.parseInt($NUMBER.text);};
```

As can be seen the tree grammar imports the *NUMBER* token from the parser grammar. Furthermore the ^ followed by one or more nodes is a pattern match on a tree structure, which after matching executes the actions within the curly braces. The non-terminals *gram* and *expr* are parameterised by an integer value. This value is accessed by a dollar sign ($) and is returned after each pattern match. This propagates back to the root of the tree and is used as final result of the tree walker.

4.2 Parsec

Parsec has multiple possibilities for the encoding of precedence and associativity of operators. One being a rewrite from an EBNF with implicit operator information to a corresponding combination of parsers, analogous to what we did in the ANTLR example. Another being a predefined structure called a *buildExpressionParser* which uses a table driven approach for explicitly defining operator information. We will demonstrate the second approach by building a combined lexer/parser/interpreter which again processes arithmetic expressions. Furthermore, Parsec can directly manipulate results from parsers and therefore the implementation in Listing 9 combines the lexing, parsing and interpreting in one step.

Listing 9: Expressions in Parsec

```
gram  = do x <- expr
           eof
           return x

expr :: Parser Integer
expr  = buildExpressionParser table factor

table = [[op "^" (^) AssocRight]

        ,[op "*" (*) AssocLeft,
          op "/" div AssocLeft]

        ,[op "+" (+) AssocLeft,
          op "-" (-) AssocLeft]]

       where op s f assoc = Infix
        (do{ string s; return f}) assoc

factor = (do i <- many1 digit
             return (read i))
         <|>
         (do  string "("
              y <- expr
              string ")"
              return y)
```

Again *gram*, *expr* and *factor* are parser combinations corresponding to the non-terminals used in Listing 6. The do notation can be seen as a sequencing of parsers, for example *gram* first applies the *expr* parser, stores the result in x, applies the *eof* (end of file) parser, and finally returns a parser which will result in *x*. *buildExpressionParser* is a predefined function in Parsec which takes a table *table* containing operators and precedence information, and a basic expression term *factor*. The table of operators defines precedence by position in the list; the higher in the list the higher the priority. Associations *AssocLeft* and *AssocRight* are predefined data constructors. Finally *factor* tries to parse a sequence of digits (which is returned as a string) and tries to convert it to an Int by using read. If that fails it tries to parse a parenthesized expression and returns the result of that expression.

4.3 Tinadic Parsing

Here we will demonstrate Tinadic Parsing by again implementing the arithmetic expressions example. Similar to Parsec, because of the lack of a generation step, Tinadic Parsing can define new constructs in the underlying implementation language without having to change the underlying parsing framework. Thus Tinadic Parsing could add support for a *buildExpressionParser* like used in the Parsec example, or still use an EBNF like syntax like used in the implementation given in Listing 10.

Listing 10: Parsing Expressions in Tinadic Parsing

```
exprgrammar =
   Expr   -> [[Term, <*>[Addop, Term]]]

 | Term   -> [[Pow, <*>[Multop, Pow]]]

 | Pow    -> [[Factor, <?>[Expop, Pow]]]

 | Factor -> [[Number]
              ,[Token '(', Expr, Token ')']]

 | Number -> [[<+>[digit]]]

 | Addop  -> [[Check (member "+-")]]

 | Multop -> [[Check (member "*/")]]

 | Expop  -> [[Check (member "^")]]

digit = Check (member "0123456789")
```

Similar to the ANTLR parser the above parser uses implicit encoding of operator precedences and associativities. The operator <*?*> denotes optional tokens or non-terminals, similar to the EBNF operator *?*. After applying this grammar function and a token stream to the parser a parse tree is returned. In Listing 11 an interpreter is built by defining an evaluation function in the underlying implementation language of the parser, Amanda. The evaluation function uses helper functions defined in Appendix A.

Listing 11: Interpreting Expressions in Tinadic Parsing

```
eval :: parseTree -> num
eval (ParseNode Expr   xs)  = chainl xs lst
eval (ParseNode Term   xs)  = chainl xs lst
eval (ParseNode Pow    xs)  = chainl xs lst
eval (ParseNode Factor [x]) = eval x
eval (ParseNode Number xs)  = digits xs

lst = [("+", +), ("-", minus),
       ("*", *), ("/", /), ("^", ^)]
```

parseTree is an algebraic datatype with one or more *ParseLeaf*s containing the spelling (char) for a token, or a non-terminal in a *ParseNode*. *chainl* is a function that folds a list containing at least one term, followed by zero or more times an operator and a term. This corresponds to the same structure used in the parser grammar. Furthermore, *digits* converts the spelling of all the tokens hanging below the *ParseNode Number* into a number.

4.4 Conclusions

Arithmetic Operators

The encoding of arithmetic operators in ANTLR takes quite some effort due to the necessary rewriting of left recursion and the extra non-terminals needed to add precedence levels for operators. This can make subsequent additions to the grammar, such as an operator with a new precedence level, hard to oversee and error prone. For Parsec the encoding of operators is easier implemented by means of a *buildExpressionParser*, as shown in the previous code example. Left recursion is still a problem, although this is partially solved by predefined parsers and combinators such as *chainl* and *buildExpressionParser*. Tinadic Parsing can use both the approach of ANTLR by defining a grammar function alike to the EBNF, or it can use the approach of Parsec and define your own buildExpressionParser for further use.

Interpretation

In ANTLR, the separation in a parsing and interpreting phase by using intermediate output such as an AST makes it easier to reason about it. Subsequent changes in the parser or tree structure can still however affect the interpreting phase. In Parsec parsers do not return parse trees or AST's but immediate values; this can be used to immediately interpret the result. Although this shortens the code it can be quite hard to change afterwards. Tinadic Parsing uses parse trees instead of AST's, this means the interpreter is more based on the actual structure of the parser instead of a generated tree. Changes made in the parser, therefore severely affect the interpreter.

5. LEXING AND THE OFF-SIDE RULE

This section will extend the previous expressions example by allowing whitespace between tokens. We will use this opportunity to first demonstrate some simple lexing capabilities of the parser construction techniques. Lexing is quite easy when constrained to the grouping and discarding of symbols; however when confronted with meaningful whitespace as used for the lexing of the off-side rule [19] this changes severely. The off-side rule is used to avoid curly braces and other explicit sectioning of code. An example of the off-side rule as used in Amanda can be seen in the Appendix in Listing 16. A function is defined by a left hand side containing the function name (compare) and zero or more arguments (x and y). A function containing guards (the if statements at the end), should equally indent all equals tokens. Another good example is the use of indentation blocks in Python [31]. We will suggest possible solutions for the implementation of the off-side rule in this section.

5.1 ANTLR

ANTLR can use a separate grammar for the lexer and parser or use a combined grammar as shown previously. The most natural approach, if the problem permits, is to use a combined grammar as used below.

5.1.1 Lexing

Below in Listing 12 is an extension of the arithmetical expressions example, where whitespace between tokens is allowed. For illustration purposes the operators are defined as separate lexer tokens to demonstrate features of the lexer.

Listing 12: Lexing/Parsing Expressions in ANTLR

```
tokens {
 PLUS  = '+';
 MINUS = '-';
 MULT  = '*';
 DIV   = '/';
 EXP   = '^';
}

gram   : expr EOF!;

expr   : term (addop^ term)*;
term   : pow (multop^ pow)*;
pow    : factor (expop^ pow)?;
factor : NUMBER | '('! expr ')'!;

addop  : PLUS | MINUS;
multop : MULT | DIV;
expop  : EXP;

NUMBER : ('0'..'9')+;

WS : ( ' ' | '\t' | '\r' | '\n' )+
     { skip(); } ;
```

Whitespace is discarded in the lexer by using the *skip* command whenever whitespace is encountered. Tokens such as the arithmetical operators and *NUMBER* will be tokenized by the lexer before the parser phase.

5.1.2 The Off-Side Rule

A common way to deal with the off-side rule is to add explicit *INDENT* and *DEDENT* tokens (such as curly braces) while processing (or before processing) the token stream [25]. If the lexer matches whitespace that is further off the side than the previous statement an *INDENT* token should possibly be emitted. The emitting of *DEDENT* tokens is similar, but harder due to the possibility of multiple dedentations, which would force the emitting of multiple *DEDENT* tokens. Beside the extra tokens to be added in the token stream, a lot of bookkeeping such as column numbers for each token and current indentations should be stored. Some of the problems of this approach in ANTLR will be discussed in the conclusion.

5.2 Parsec

Parsec has multiple approaches for lexing: the first is the usage of a seperate scanner, the second is the usage of lexeme parsers. A separate (handwritten) scanner should be used when the token stream has to be pre-processed. The second approach is quite different; instead of defining separate scanner functions and a parser, the lexing and parsing is combined in one phase by using lexeme parsers.

5.2.1 Lexing

In Listing 13 the extended arithmetic expression example is implemented using these lexeme parsers. The lexeme parsers are defined using a language definition (which is a record), containing information such as commenting rules, whitespace and operator names. A lexeme parser always consumes trailing whitespace (as defined in the record) guaranteeing the next parser starts at the correct input [20].

Listing 13: Lexing/Parsing Expressions in Parsec

```
lexer :: P.TokenParser ()
lexer = P.makeTokenParser
        (emptyDef
        { reservedOpNames =
         ["*","/","+","-", "^"]
        })

whiteSpace = P.whiteSpace lexer
natural    = P.natural    lexer
parens     = P.parens     lexer
reservedOp = P.reservedOp lexer

gram   = do whiteSpace
            x <- expr
            eof
            return x

expr :: Parser Integer
expr = buildExpressionParser table factor

table = [[op "^" (^) AssocRight]

        ,[op "*" (*) AssocLeft,
          op "/" div AssocLeft]

        ,[op "+" (+) AssocLeft,
          op "-" (-) AssocLeft]]

       where op s f assoc = Infix
        (do{ reservedOp s; return f}) assoc

factor = natural <|> parens expr
```

lexer is defined using an empty language definition as a start record and updating the *reservedOpNames* field. The lexeme parsers *whiteSpace*, *natural*, *parens* and *reservedOp* are then extracted from the resulting *lexer* record and defined at toplevel for clarity and efficiency. *whitespace* is used to consume leading whitespace to start *expr* at the correct input, *natural* parses a natural number, *parens* parses a parenthesized expression and *reservedOp* parses reserved operators.

5.2.2 The Off-Side Rule

Parsec can use a similar approach to ANTLR, by preprocessing the token stream and adding *INDENT* and *DEDENT* tokens. But an approach better suited for use in parser combinators would be to define so called block parsers. Parsec passes position and possibly user defined state information along through the parse process by using monads. Thereby making it possible to direct the parse process based on state information. A block parser would be defined as a parser depending on passed state information and could be reused as a building block in other combinations of parsers. Examples of this approach can be found in a paper by Hutton and Meijer [12], in parser libraries such as uulib [39] and in Listing 17.

5.3 Tinadic Parsing

Tinadic Parsing uses an approach to lexing similar to ANTLR. As seen in the previous Tinadic Parsing examples, lexing can be integrated in the parsing phase or it can be used in a separate phase as will demonstrate below.

5.3.1 Lexing

Analogous to ANTLR, we will do lexing in a separate phase to demonstrate capabilities of the lexer. The arithmetic expression example is worked out in Listing 14. The Tinadic Parsing lexer grammar is again described as function in lambda notation. This grammar and the character stream are applied as arguments to the lexer which will deliver a tokenized stream.

Listing 14: Lexing/Parsing Expressions in Tinadic Parsing

```
|| Lexer part
tokengrammar =
  Num    -> [<+>[digit]]

| Addop  -> [addopChar]

| Multop -> [multopChar]

| Expop  -> [expopChar]

| Sep    -> [<+>[separator]]

addopChar   = CheckChar (member "+-")
multopChar  = CheckChar (member "*/")
expopChar   = CheckChar (member "^")
separator   = CheckChar (member " ")


|| Parser part
expressiongrammar =
  Expr   -> [[Term, <*>[Addop, Term]]]

| Term   -> [[Pow, <*>[Multop, Pow]]]

| Pow    -> [[Factor, <?>[Expop, Pow]]]

| Factor -> [[Num]
             ,[lBrack, Expr, rBrack]]

| Num    -> [[CheckToken ((c,s) -> c=Num)]]

| Addop  -> [[CheckToken ((c,s) -> c=Addop)]]
| Multop -> [[CheckToken ((c,s) -> c=Multop)]]
| Expop  -> [[CheckToken ((c,s) -> c=Expop)]]

lBrack  = Token "("
rBrack  = Token ")"
lSquare = Token "["
rSquare = Token "]"
comma   = Token ","
```

tokengrammar is the grammar function that will be applied as argument to the lexer. *Num*, *Addop* and other lexer tokens are from the same alphabet as used in the parser grammar function. *CheckChar* is a data constructor which will direct the lexer to check for equal characters, and if equal deliver a matching token. *CheckToken* works analogous to the *CheckChar* data constructor but on token equality.

5.3.2 The Off-Side Rule

Tinadic Parsing has an implementation of the off-side rule combining the previously mentioned state passing and parser blocks. In the lexer phase tokens are augmented with line and column numbers to enable the parser to depend on that information. The parser grammar function contains parser blocks will check for correct line and column information on the following tokens. A snippet of an implementation of Amanda-style guarded function definitions (Listing 16) is shown in Listing 17 in the Appendix.

For clarity and brevity, function definitions are assumed to require guards.

As can be seen a function definition is composed of a left hand side, *Lhs*, containing one or more identifiers (*idf*) representing the function name and possibly multiple arguments. This is followed by one or multiple *IfClause*s, and an optional otherwise clause (*OtherClause*). An *IfClause* starts with an equals sign, *eqToken*, which should start at an indentation, followed by an expression (*Expr*) and a comma, an *ifToken* and ending in an expression with a dedent.

5.4 Conclusions

While ANTLR has no trouble discarding whitespace in the lexer phase, it is much harder to direct the parse process based on amounts of whitespace. This is a hurdle in the design for a possible implementation for the off-side rule in ANTLR. ANTLR's lexer is mostly designed for efficiency, the standard classes therefore do not provide support for multiple token emissions and would force the user to override the standard token streams and other classes uses by the lexer. Beside this, the use of the lexer in a combined parser grammar is intuitively integrated and readable.

Lexeme parser can easily be created in Parsec by supplying a language definition containing commenting rules and other language specifics. This approach delegates a lot of subtleties of lexing to the internal working of the lexeme parsers, thereby relieving the user. Analogous to this approach is the definition of block parsers, where the internal handling of the off-side rule is hidden in indent and dedent building blocks. The lexeme bases parser approach suffers though, when the complexity of the needed lexing is higher than the capabilities of the lexeme parsers. This would force the user to manually write a scanner or to define his own state information to pass through the parse process. Furthermore the use of lexemes is very similar to the use primitive parsers, making it easier to combine lexeme parsers and primitive parsers.

Lexing in Tinadic Parsing is very similar in use to the parsing; the lexer and parser grammar are both defined in lambda notation and data constructors can be shared. The parser grammar function can be defined quite cleanly by using block parsers similarly to Parsec; although the internal handling and the definition of these block parsers is still very complex.

6. CONCLUSIONS

In this paper we have demonstrated advantages and disadvantages of three parser construction implementations: ANTLR, Parsec and Tinadic Parsing. Notation-wise each implementation has its strengths; ANTLR and Tinadic Parsing are close to EBNF notation, while Parsec uses a functional approach with combinators similar to EBNF operators. Thus, the transcribing of an EBNF grammar to a parser would be the easiest in ANTLR and Tinadic Parsing.

However, when developing a language with arithmetic expressions, containing operators with different precedence levels and associativities, the encoding to EBNF, and therefore the transcribing to an ANTLR grammar or Tinadic Parsing lambda notation, is less natural. The table based approach in Parsec solves this problem more directly by being able to state precedence and associativity instead of needing a series of grammar transformations. Constructs such as a table parser are easier defined in Parsec and Tinadic Parsing due to the lack of a generation step. Non generating parser libraries can create new primitive constructs without having to redefine the existing parser constructs. This would for example enable Tinadic Parsing, analogously to Parsec, to implement a table based parser by using existing parsing primitives or the expression power of the underlying implementation language, Amanda.

Interpretation of parse results varies greatly in the three implementations; while ANTLR uses an AST as intermediate output, therefore making it possible to use a tree walker, Parsec often uses direct interpretation of the parse results, giving a concise implementation, and finally Tinadic Parsing uses a concrete syntax tree, which shows the full parsing process. ANTLR's approach is suited well for larger grammars, although changes in the parser grammar might affect the tree structure and therefore affect the interpreter. Parsec's direct interpretation works well for small grammars, but when a separate step output step is needed, the user would be forced to rewrite output to his/her own algebraic data type. Finally Tinadic Parsing interpreter is even more affected by changes in the parser due to the greater changes in a concrete syntax tree, for larger grammars the parse tree can give more information though.

Average lexer and parser usage in each implementation, does not require a big shift in notation or thinking, thereby increasing usability. Usability of the lexer for heavier use such as the implementation of the off-side rule is different though. ANTLR's lexer for instance, lacks standard support for the implementation of the off-side rule. This is more naturally solved in Parsec and Tinadic Parsing by using a block parser based approach.

Concluding, given an EBNF an implementation in ANTLR and to lesser extent, Tinadic Parsing, would be the most straightforward. Parsec and Tinadic Parsing, due to the ability to define new parsers and combinators from the existing building blocks and implementation language, are best suited for constructing parsers and interpreters that need a more flexible approach, such as a language with an off-side rule.

7. FUTURE RESEARCH

The results delivered here are still exploratory, an obvious extension to this work would be more thorough testing based on formal criteria and an extension of the existing testing framework [7]. Such an extension could of course benchmark speed and memory consumption, or compare the ability to parse a certain class of grammars.

There are also other approaches to building parsers or compilers that can be considered, such as the use of attribute grammars [38, 4, 16] or the use of alternative parser construction interfaces. The uu-parsinglib [35, 34] for example, uses the applicative interface [24], which is a less powerful than the monadic interface used for the Parsec examples, but possibly more intuitive. This applicative interface can also be used as interface for constructing parsers in Parsec. An interesting combination of the monadic and applicative interfaces is demonstrated by Wallace [43] to create a parser able to return a partial parse and therefore possibly reducing space complexity.

Acknowledgments

I would like to thank Jan Kuper for the suggestion of this subject, his great enthusiasm throughout the project and finally his help with Tinadic Parsing. I would like to thank Michael Weber for keeping pressure on the bachelor project. For helpful comments on my paper I would like to thank Charl, Jorrit, Pascal, Lester and Annemiek.

REFERENCES

[1] Dick Bruin. Amanda (version 2.04) [software], 2004. http://www.engineering.tech.nhl.nl/engineering/personeel/bruin/data/.

[2] William H. Burge. *Recursive programming techniques / William H. Burge*. Addison-Wesley Pub. Co., Reading, Mass., 1975.

[3] Jeroen Fokker. Functional parsers. In Jeuring and Meijer [14], pages 1–23.

[4] Jeroen Fokker and S. Doaitse Swierstra. Abstract interpretation of functional programs using an attribute grammar system. Technical Report UU-CS-2007-049, Department of Information and Computing Sciences, Utrecht University, 2007.

[5] Bryan Ford. *Packrat Parsing: a practical linear-time algorithm with backtracking*. PhD thesis, Massachusetts Institute of Technology, Sep 2002.

[6] Bryan Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Symposium on Principles of Programming Languages*, pages 111–122. ACM Press, 2004.

[7] Bas van Gijzel. Code depository, 2009. http://fmt.cs.utwente.nl/~michaelw/projects/vgijzel/.

[8] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman, Amsterdam, 3 edition, June 2005.

[9] Robert Grimm. Better extensibility through modular syntax. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 38–51, New York, NY, USA, 2006. ACM.

[10] David R. Hanson. Compact recursive-descent parsing of expressions. *Softw. Pract. Exper.*, 15(12):1205–1212, 1985.

[11] Graham Hutton. Higher-order Functions for Parsing. *Journal of Functional Programming*, 2(3):323–343, July 1992.

[12] Graham Hutton and Erik Meijer. Monadic Parser Combinators. Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.

[13] Graham Hutton and Erik Meijer. Monadic Parsing in Haskell. *Journal of Functional Programming*, 8(4):437–444, July 1998.

[14] Johan Jeuring and Erik Meijer, editors. *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text*, volume 925 of *Lecture Notes in Computer Science*. Springer, 1995.

[15] Steven C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[16] Donald E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145, June 1968.

[17] Pieter Koopman and Rinus Plasmeijer. Efficient combinator parsers. In *In Implementation of Functional Languages, LNCS*, pages 122–138. Springer-Verlag, 1999.

[18] Jan Kuper. Tinadic parsing, 2006. Unpublished work.

[19] P. J. Landin. The next 700 programming languages. *Commun. ACM*, 9(3):157–166, 1966.

[20] D. J. P. Leijen. *Parsec, a fast combinator parser*, 2001.

[21] D. J. P. Leijen and H. J. M. Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-35, Department of Information and Computing Sciences, Utrecht University, 2001.

[22] Daan Leijen. Parsec (version 2.1.0.1) [software], 2008. http://hackage.haskell.org/cgi-bin/hackage-scripts/package/parsec-2.1.0.1.

[23] Peter Ljunglöf. *Pure Functional Parsing – an advanced tutorial*. Licenciate thesis, Department of Computer Science, Gothenburg University and Chalmers University of Technology, April 2002.

[24] Conor Mcbride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, 2008.

[25] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Bookshelf, 2007.

[26] Terence Parr. Antlr (version 3.1.2) [software], February 2009. http://www.antlr.org.

[27] Terence J. Parr and Russell W. Quong. Antlr: A predicated-ll(k) parser generator. *Software Practice and Experience*, 25:789–810, 1995.

[28] Simon Peyton Jones. *Haskell 98 Language and Libraries: the Revised Report*. 2003.

[29] Vaughan R. Pratt. Top down operator precedence. In *POPL '73: Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 41–51, New York, NY, USA, 1973. ACM.

[30] Niklas Röjemo. Efficient parser combinators.

[31] Guido van Rossum. *The Python Language Reference Manual*. Network Theory Ltd., September 2003.

[32] R. S. Scowen. Extended bnf - a generic base standard. www.cl.cam.ac.uk/~mgk25/iso-14977-paper.pdf.

[33] S. Doaitse Swierstra. Combinator parsers: From toys to tools. In *Electronic Notes in Theoretical Computer Science*, volume 41. Elsevier Science Publisher, 2001.

[34] S. Doaitse Swierstra. Combinator parsing: A short tutorial. Technical Report UU-CS-2008-044, Department of Information and Computing Sciences, Utrecht University, 2008.

[35] S. Doaitse Swierstra. uu-parsinglib (version 2.2.0) [software], 2009. http://hackage.haskell.org/cgi-bin/hackage-scripts/package/uu-parsinglib.

[36] S. Doaitse Swierstra and Luc Duponcheel. Deterministic, error-correcting combinator parsers. In *Advanced Functional Programming*, pages 184–207. Springer-Verlag, 1996.

[37] D. A. Turner. Miranda: a non-strict functional language with polymorphic types. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 1–16, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[38] Utrecht University. Utrecht university attribute grammar (version 0.9.10) [software], 2009. `http://www.cs.uu.nl/wiki/bin/view/HUT/AttributeGrammarSystem`.

[39] Utrecht University. uulib (version 0.9.10) [software], 2009. `http://hackage.haskell.org/cgi-bin/hackage-scripts/package/uulib`.

[40] Sreeni Viswanadha and Sriram Sankar. Javacc (version 4.2) [software], February 2009. `https://javacc.dev.java.net/`.

[41] Philip Wadler. How to replace failure by a list of successes. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 113–128, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[42] Philip Wadler. Monads for functional programming. In Jeuring and Meijer [14], pages 24–52.

[43] Malcolm Wallace. Partial parsing: Combining choice with commitment. *Implementation and Application of Functional Languages: 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers*, pages 93–110, 2008.

APPENDIX A: CODE

Listing 15: Interpreter Helper Functions

```
digits :: [parseTree] -> num
digits xs = digitsh xs 0
  where digitsh []                  n = n
        digitsh ((ParseLeaf x):xs) n = digitsh xs (n*10 + (atoi [x]))

chars :: [parseTree] -> [char]
chars [] = []
chars ((ParseLeaf x):xs) = x : (chars xs)

chainl :: [parseTree] -> [([char], (num -> num -> num))] -> num
chainl [x]      lst = eval x
chainl (x:y:zs) lst = (evalop y lst) (eval x) (chainl zs lst)

evalop :: parseTree -> [([char], (num -> num -> num))] -> (num -> num -> num)
evalop (ParseNode _ x)  lst = lookup (chars x) lst
 where lookup x []          = error "no existing op"
       lookup x ((y,z):yzs) = z, if x=y
                            = lookup x yzs, otherwise

minus x y = x - y
```

Listing 16: Amanda-style function definitions

```
compare :: num -> num -> [char]
compare x y = "first argument is smaller" , if x < y
            = "arguments are equal"       , if x = y
            = "second argument is smaller", otherwise
```

Listing 17: Off-Side Rule Example

```
...
| Def         -> [[Lhs, <+>[IfClause], <?>[OtherClause]]]
| Lhs         -> [[idf, <*>[idf]]]
| IfClause    -> [[eqToken<$>addIndent, Expr, comma, ifToken, Expr<$>removeIndent]]
| OtherClause -> [[eqToken<$>addIndent, Expr, comma, otherToken<$>removeIndent]]
...
```