Parsing Idris

A part of the Tools and Tactics for Idris project

Ahmad Salim Al-Sibahi (asal@itu.dk)

IT University of Copenhagen
Supervisors:
David Raymond Christiansen (drc@itu.dk)
Dr. Peter Sestoft (sestoft@itu.dk)

December 7, 2013

1 Introduction

Idris[1] is a strict pure dependently-typed programming language inspired by Haskell[2] and ML-style languages[3]. As a dependently-typed language Idris allows types to be indexed by values such as the natural numbers, in addition to being parametrised over other types like other functional languages. This makes Idris suitable for reasoning formally about programs, and makes it suitable for a variety of applications such as writing embedded domain specific languages (EDSLs) because many errors can be reported at compile time.

Yet all this flexibility comes at a price. Since there is not distinction between, the grammar is vastly complicated. Moreover, the focus on EDSLs has brought in more features such as syntactic extensions, which makes reasoning about the complete grammar impossible statically. Partly because of this and partly because of the indentation sensitive Haskell-like syntax; Idris is not a particularly trivial language to parse.

As such one must carefully evaluate the way the grammar is written, and write the parser in such way that there is little room for ambiguity. Furthermore, the source code must be tracked carefully in order to provide sensible errors in all parts of the compiler pipeline, and that external tools can use this to provide additional features such as clickable source code.

A parser written in the Parsec[4] library already exist for Idris. Yet because of all the advanced features of Idris, and the lack of time spent on the parser to accommodate these features, error locations and messages are highly inaccurate.

In this paper I will outline my work regarding restructuring of the Idris parser such that the grammar is more sensible, the parser provides better errors and that the source code is tracked more accurately. In Section 2 I will discuss the Idris grammar and what particular difficulties there can be when trying to parse it. Section 3 discusses the particular parser technology used, and the advantages of using it. Section 4 describes the specific effort put into improving the grammar and parser of Idris. I will evaluate the solution in Section 5 and discuss future work in Section 6. Lastly I conclude in Section 7.

2 Idris Syntax

The syntax of Idris is heavily inspired by the syntax of Haskell; with some modifications to better accommodate the dependent types-based programming paradigm and EDSLs.

In this section I will use examples to highlight some of the features that makes creating a parser non-trivial.

Figure 1 showcases a simple inductive data declaration for natural numbers, and a pattern-matching definition for a function 'add' which computes the sum of two naturals. The only noticeable difference in syntax from an ordinary Haskell program is that ':' is used for type declarations instead of '::'.

```
data N =
    zer
  | suc N

add : N -> N -> N
add zer     m = m
add (suc n) m = suc (add n m)
```

Figure 1: Simple Idris program

Similarly to Haskell, Idris allows one to define custom operators with user-defined fixity and precedence. Because of the lack of distiction between types and terms in Idris; these operators can now appear anywhere in the program, even at type level. This can be seen in Figure 2 where the data type for existential types is using the operator '***' as a name.

Another thing the reader might notice is that both the '***' operator and a lambda declaration were allowed in the type signature for 'take_while'.

Lastly, the reader should notice that much of the syntax presented is indentation sensitive; for example the data constructors and the 'where'-block.

```
infix 8 ***

data (***) : (a : Type) -> (b : a -> Type) -> Type where
  exists : {a : Type} -> {b : a -> Type} -> (w : a) -> (p : b w) -> a *** b

infixr 5 :::

data Vector : (n : N) -> (a : Type) -> Type where
  nil   : {a : Type}                              -> Vector zer a
  (:::) : {n : N} -> {a : Type} -> a -> Vector n a -> Vector (suc n) a

take_while : (p : a ->  Bool) -> Vector n a -> N *** (\ m => Vector m a)
take_while p nil       = exists zer nil
take_while p (x ::: xs) with (take_while p xs)
  | exists p ys = if p x then exists (suc m) (x ::: ys) else eps
  where eps = exists zer nil
```

Figure 2: Operators and lambdas

For improved readability of EDSLs, Idris allows further expansion of the syntax. In Figure 3 there has been defined two syntactic extensions, one for the 'take_while' function and one for the existential data type. The syntax definitions (line 1-2) can define new keywords or syntactic markers (specified as either simple identifiers or quoted in a string), capture other expressions which are delimited in square brackets (e.g. '[vect]' in the example), or capture names which are delimited in curly braces (e.g. '{x}') that can be used for binding. This makes syntactic extensions quite powerful, and the only thing one cannot seemingly extend the language with is recursive grammar definitions.

At lines 10-11 in Figure 3 there is a clause which uses these syntactic extensions as any other built-in syntax. Rather than being baked into Idris, many things like 'if'-expressions are in practice defined using syntax extensions in the library.

```
syntax take' [vect] as {x} while [body] = take_while (\x => body) vect
syntax "∃" {x} "->" [ty] = _ *** (\x => ty)

isEven : N -> Bool
isEven zer     = True
isEven (suc x) with (isEven x)
  | True  = False
  | False = True

testSyntax : ∃ m -> Vector m N
testSyntax = take' suc (suc zer) ::: zer ::: suc zer ::: nil as i while isEven i
```

Figure 3: Syntax extensions

3 Trifecta

Trifecta[5] is a monadic parser combinator library created by Edward Kmett, with focus on providing good tools for error reporting and incremental parsing.

In this section I will explain what parser combinators are, and how Trifecta as a library is designed such that the combinators provide the aforementioned features.

3.1 Parser Combinators

Traditionally there are two ways of creating a parser: either by hand-writing one, or by giving the grammar as input to a parser generator such as yacc[6] and generating one.

Parser combinator libraries[7, Chapter 16][8] provides aid to creating hand-written parsers by providing, often in the form of an EDSL, a set of generic higher-order functions called parser combinators.

Parser combinators produce new parsers by taking other parsers as input, and provides features as alternation, sequencing, choice, repetition and cursor movement.

3.1.1 Combinators by example

```
landCode = string "+45"

phoneNumber = option "" landCode *> count 8 digit

username = many alphaNum

login = do id <- try phoneNumber <|> username
           string "-"
           password <- some alphaNum
           return (id, password)
```

Figure 4: Parsing a login

Figure 4 shows a simple example of a parser for parsing login information given in the following format "id–password". In this format id is either a phone number or a self-chosen user-name and password is a string of alphanumerical characters.

The choice for id is created by using the '<|>' combinator, which tries parsing the first alternative and if it fails, without consuming any input, it will try parsing the second alternative. Since there is overlap between the definition of phone number and user-name parser, the phone number parser must be wrapped in a try-combinator which ensures that input can be restored by backtracking such that no input is consumed and other alternatives can be tried. Because the parser is monadic, sequencing multiple parsers is conveniently available using do-notation.

In the example there are three repetition parser combinators used: 'count' which specifies an exact number of repetitions, 'many' which specifies zero or more repetitions and 'some' which specifies one or more repetitions; although many more are usually available in such libraries.

Finally, one can observe that the phone number parser uses an combinator 'option' which allows specification of an optional part of the input and the combinator '$*>$' which ignores the result of the left operand and returns the result of the right operand.

3.2 Features of Trifecta

While there are many parser combinator libraries for Haskell, Trifecta provides the following desirable features:

Incremental parsing When working with external tools the input might change quite often. It is therefore desirable if only the relevant part of the input is parsed again, and Trifecta achieves this by using monoidal parsing[9]. This feature provide baseline support for semantic highlighting, code completion and other types of parser-dependent tooling.

Error reporting Trifecta has a focus on nice diagnostics and pretty printing. If a parser fails, it doesn't just show what it expected but also nicely formats and points to the place of error. Moreover, Trifecta provides utilities to highlight semantic errors in the code using the same type of pretty printing it uses for syntactic errors.

4 Solution

In many ways rewriting the parser from Parsec to Trifecta is non-trivial. Nonetheless, I will outline in this section my effort on improving the parser by highlighting the interesting parts of my work, namely formalization of the grammar and improvement on the error reporting and locating.

4.1 Formalising the Grammar

Part of the challenge of writing a new parser for Idris is that, at the time, there were no formalization of grammar in Idris. This entailed two issues:

1. There was no official reference regarding the Idris syntax. This meant that it was hard to find the syntactically correct way to write a program and in case of error one could either:

    - try to isolate the bug in a structured fashion and try to guess how it should be corrected

- or try to use a lot of time analysing the parser code and see what the legal syntax should have been

2. There were some grammatical inconsistencies, such that some programs were falsely accepted (e.g. programs that were missing a '}') and some didn't behave as expected (e.g. proof-blocks did support verbose but not indentation-style syntax, unlike do-blocks which supported both).

The formalization to an official BNF grammar was further complicated by the fact that Idris supports an indentation-based syntax and user defined extensions. As such the resulting grammar is incomplete since it is impossible to formalize all possible syntax extensions, and is not totally sound because it does not reflect all of the lexical properties of indentation. The latter could be improved by specifying the grammar in an extended BNF syntax like the one used by Adams[10], but at the time of analyses there were too many inconsistencies regarding indentation in order to completely reflect the actual syntax and thus this was omitted.

In any case the grammar should provide a useful, covering and understandable overview of the syntax.

The actual translation process was done by carefully analysing the existing parser structure, noting any seeming inconsistencies and correcting them in the grammar. The final result is available in Appendix A.

4.2 Documenting for Failures

Another issue there was with the existing parser was that grammar rules were improperly documented; so instead of error messages showing what legal types of grammar it expected, it instead showed a list of tokens that can appear somewhat random for the intended user. For example if a user had mistyped something in the start of an expression, it would show all operators, keywords and marker symbols that were possible at that point instead of just showing that an expression was expected.

Therefore some of the work done on the parser was to provide a human-readable string explanation for each grammar rule using the '<?>' combinator. Figure 5 shows an example of such annotation.

```
{-- | Parses an accessibilty modifier (e.g. public, private) --}
accessibility :: IdrisParser Accessibility
accessibility = do reserved "public";   return Public
            <|> do reserved "abstract"; return Frozen
            <|> do reserved "private";  return Hidden
            <?> "accessibility␣modifier"
```

Figure 5: Documenting Parser Rules

4.3 The Parser with Fear of Commitment

One of the more challenging parts of the grammar re-factoring process was that many of the rules were written in such way that they didn't commit to any of the branches in a list of alternatives.

This meant that each time there was an error in one of the alternatives, the parser had to backtrack all the way back to the starting point.

In addition to being exponentially slow and memory requiring, in the case that none of the branches succeeded it will show a location at where it was before parsing the specific grammar rule instead of the location where the possible error happens. In the case of top-level declarations, this can be multiple lines and columns away from the actual location, and finding the correct location could be extremely hard for the programmer.

Figure 6 shows one of the rules, were the reader may observe that all alternatives are wrapped in the try-combinator meaning it will never commit to any alternative in case of an error.

```
decl' :: SyntaxInfo -> IdrisParser PDecl
decl' syn =    try fixity
           <|> try (fnDecl' syn)
           <|> try (data_ syn)
           <|> try (record syn)
           <|> try (syntaxDecl syn)
           <?> "declaration"
```

Figure 6: Non-committing Parser Rule

To fix this usually two things are required:

1. Reordering of rules such that less ambiguous rules comes before more ambiguous rules

2. Minimizing the span of the try-combinator such that only the ambiguous part of covered, i.e. usually until a keyword or marking character

Figure 7 shows the committing version of the rule shown in Figure 6, where some of the alternatives has been reordered to avoid backtracking. It should be noted that this was a simple example of such re-factoring, and more complex rules such as the one for data declaration required significantly more effort.

```
decl' :: SyntaxInfo -> IdrisParser PDecl
decl' syn =    fixity
           <|> syntaxDecl syn
           <|> fnDecl' syn
           <|> data_ syn
           <|> record syn
           <?> "declaration"
```

Figure 7: Committing Parser Rule

4.4 Improving Source Tracking

In the old parser there was tracking of source code in the abstract syntax tree, but some of the tracking was highly inaccurate and was missing column information.

In some elements like atomic identifiers or references, the location was first retrieved after the token was parsed; This meant that if there was a lot of white space after the reference, the location would have been at the end of the white space, instead of near the token. For these tokens, a correction was made such that the location was retrieved *before* a token was parsed. This additionally had the benefit of allowing easier access to highlighting relevant identifiers by simply selecting all text from the stored location to white space.

Furthermore, all the abstract syntax trees at various stages in the pipeline were altered to provide column information such that multiple items on the same line were distinguishable.

5 Evaluation

In this section I will highlight some of the improvements that were discovered and fixed in the new parser, and what challenges there still are because of the way the grammar is constructed. Additionally I will highlight the improvements in error handling and the reception by the community both non-empirically and in terms of reported bugs.

5.1 Syntactic Corrections and Challenges

The following bugs in the syntax was corrected:

1. Proof and tactic blocks allow indented-style syntax, similarly to do blocks
2. Documentation comments can be nested inside existing comments
3. Where is optional on instances
4. Tactic sequences can hold more than two tactics
5. Using and parameter blocks can be empty
6. Checks that parentheses are matched

The only challenging item in the current implementation of the parser is the parsing of nested parenthesized expressions, which currently is exponentially slow. This is due to ambiguity in grammar where operator slices e.g. '(\\ x)' are hard to distinguish from lambdas '(\ y => y * x)' and thus all parenthesized expression require full lookahead.

To solve this issue, the grammar needs to be rewritten such that there is less ambiguity between the various kinds of parenthesised expressions.

5.2 Error Report Improvements

As a showcase for error reporting improvements, I will highlight a couple of examples where a simple mistake resulted in some uninformative error report by the old parser but where the new parser shows a clear placement and presentation of error.

```
module ErrorReport

data List : Type -> Type where
  Nil :: List a
  Cons :: a -> List a -> List a
```

(a) File with error

```
"./error-report.idr" (line 3, column 11):
unexpected ':'
expecting "infixl", "infixr", "infix", ...
```

(b) Old parser report

```
./error-report.idr:4:8: error: expected: type signature
  Nil :: List a
        ^
```

(c) New parser report

Figure 8: Improvements in Error Reporting

Figure 8 shows an Idris program where the user had accidentally used '::' for data constructor type signatures similarly to Haskell type signatures, instead of ':' used in Idris type signature.

As can be observed in the figure, the old parser reports that the error is at line 3 (where type is declared, not the constructor) and suggests that it expected one token from a long list of unrelated ones.

The new parser substantially improves the error location by pointing directly at the place of error and showing a part of the code that was affected. Furthermore it only suggests that the type signature should be fixed, which is the correct error in this case.

```
module ErrorReport

test : Int -> Int -> Int
test x y
= let z = x + y in
do x +
```

(a) File with error

```
"./error-report2.idr" (line 4, column 6):
unexpected "x"
expecting ":" or operator
```

(b) Old parser report

```
./error-report2.idr:7:1: error: unexpected
    EOF, expected: expression
<EOF>
^
```

(c) New parser report

Figure 9: Improvements in Error Reporting, cont.

Figure 9 shows similarly that the old parser error report were basically unusable. The user had accidentally forgot to finish the expression of an infix operator and instead of complaining that it had missed the rest of the operation; it instead complains at the start of the function clause saying that it didn't expect the argument given.

Contrarily, the new parser correctly highlights that the parser didn't expect end-of-file but an expression and the user can immediately correct this bug instead of hunting down the error at the completely wrong place.

5.3 Community Reception

The parser described in this paper has officially been included in the Idris language, and is now the solely used parser.

The reception has been generally positive by the community and in the word of the language designer, it has been described as following in the changelog: "New parser implementation with more precise errors."

Initially there has been less than 10 bugs reported, many of which already existed in the previous parser, but were undiscovered due to bad error messages. However, the number of bugs reported regarded parsing has decreased substantially and the last reported bug was reported more than a month ago since writing this report.

NOTE: Add results for HTML clickable code thing if there is time

6 Future Work

While there has been lot of effort improving the parser, not all of the advantages of the parser technology was utilized. In the future work, I would suggest the following improvements that could be made:

Spanning the source code Currently the source code is only tracked by a cursor in the abstract syntax tree, a better solution would entail tracking the full span of a structure in the AST. This would allow easier access to pretty printing and error highlighting the affected code.

Utilising incremental parsing Trifecta supports incremental parsing, and it could provide many useful features in regards to tooling. As part of future work it could be of great interest to utilize this feature to allow for things like semantic highlighting and code completion.

Refactoring semantics out of the parser Currently the parser does a lot of semantic checking inside the parser structure, in order to ensure that some things are correct by construction.

For example implicit parameters are not allowed in function arguments and the parser currently disallows writing implicit argument syntax in the type signatures where function arguments are expected.

This results in a parser error saying that it didn't expect the implicit syntax, and while this is correct it might be confusing for a new user. As such, many of these semantic checks should be moved out of the parser and refactored in a separate syntax checking step which provide better error report for such situations.

7 Conclusion

In this project I have successfully formalised the grammar and rewritten the parser for the Idris language. I did this by analysing the existing parser, fixing any inconsistencies and rewritten the new parser in such way the error reporting was improved and locations were tracked better.

I also conclude that the resulting work has been received well by the community, and that it can be seen as a great success that the resulting work now forms the basis for the Idris language syntax and parsing.

Finally, while there were improvements to be made, changing the parser technology to Trifecta has provided us with a great start for improving tooling and semantic error reporting.

References

[1] Brady, E.: Idris, a general-purpose dependently typed programming language: Design and implementation. Journal of Functional Programming **23** (9 2013) 552–593

[2] Marlow, S., et al.: Haskell 2010 language report. `http://www.haskell.org/onlinereport/haskell2010` (2010)

[3] Milner, R.: The definition of standard ML: revised. The MIT press (1997)

[4] Leijen, D., Meijer, E.: Parsec: Direct style monadic parser combinators for the real world. (2001)

[5] Kmett, E.: Trifecta official page. `http://ekmett.github.io/trifecta/`

[6] Johnson, S.C.: Yacc: Yet another compiler-compiler. Volume 32. Bell Laboratories Murray Hill, NJ (1975)

[7] O'Sullivan, B., Goerzen, J., Stewart, D.: Real World Haskell. 1st edn. O'Reilly Media, Inc. (2008)

[8] Hutton, G., Meijer, E.: Monadic parser combinators. (1996)

[9] Kmett, E.: Iterators, parsec and monoids – a parsing trifecta

[10] Adams, M.D.: Principled parsing for indentation-sensitive languages: revisiting landin's offside rule. In: Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM (2013) 511–522

A Idris Formalized Grammar

```
{- Header and Imports are currently parsed separatly -}
Main ::= ModuleHeader Import* Prog;

{-
  shortcut notation:
    ~CHARSEQ = complement of char sequence (i.e. any character except CHARSEQ)
    RULE? = optional rule (i.e. RULE or nothing)
    RULE* = repeated rule (i.e. RULE zero or more times)
    RULE+ = repeated rule with at least one match (i.e. RULE one or more times)
    RULE! = invalid rule (i.e. rule that is not valid in context, report meaningful error in case)
    RULE{n} = rule repeated n times
-}

EOL_t ::= '\n'  |EOF_t;

StringChar_t* = {- Any valid Haskell string character or escape code -};
StringLiteral_t ::= '"' StringChar_t* '"';

DocCommentMarker_t ::= '|' | '^';

DocComment_t ::=   '--' DocCommentMarker_t ~EOL_t* EOL_t
                 | '{-' DocCommentMarket_t ~'-}'* '-}'
                 ;

SingleLineComment_t ::= '--' EOL_t
                  |     '--' ~DocCommentMarker_t ~EOL_t* EOL_t
                  ;

MultiLineComment_t ::=
  '{--}'
  | '{-' ~DocCommentMarker_t InCommentChars_t
  ;

InCommentChars_t ::=
  '-}'
  | MultiLineComment_t InCommentChars_t
  | ~'-}'+ InCommentChars_t
  ;


Whitespace_t ::=
  SimpleWhitespace_t
  | SingleLineComment_t
  | MultiLineComment_t
  ;

Identifier_t ::= ['a'-'z''A'-'Z']['a'-'z''A'-'Z''0'-'9''_''.']+;
Operator_t ::= [':''!''#''$''%''&''*''+''.''/''<''=''>''?''@''\\''^''|''-''~']+

ModuleHeader ::= 'module' Identifier_t ';'?;

Import ::= 'import' Identifier_t ';'?;

Prog ::= Decl* EOF;

Decl ::=
    Decl'
  | Using
  | Params
  | Mutual
  | Namespace
  | Class
  | Instance
  | DSL
  | Directive
  | Provider
  | Transform
  | Import!
  ;

Decl' ::=
    Fixity
  | FnDecl'
  | Data
  | Record
  | SyntaxDecl
  ;

SyntaxDecl ::= SyntaxRule;

SyntaxRuleOpts ::= 'term'  |'pattern';

SyntaxRule ::=
  SyntaxRuleOpts? 'syntax' SyntaxSym+ '=' TypeExpr Terminator;

SyntaxSym ::=   '[' Name_t ']'
            |  '{' Name_t '}'
            |  Name_t
            |  StringLiteral_t
            ;

FnDecl ::= FnDecl';

{- NOTE: Check compatible options -}
FnDecl' ::=
```

```
  DocComment_t? FnOpts* Accessibility? FnOpts* FnName TypeSig Terminator
  | Postulate
  | Pattern
  | CAF
  ;

Postulate ::=
  DocComment_t? 'postulate' FnOpts* Accesibility? FnOpts* FnName TypeSig Terminator
  ;

Using ::=
  'using' '(' UsingDeclList ')' OpenBlock Decl+ CloseBlock
  ;

Params ::=
  'parameters' '(' TypeDeclList ')' OpenBlock Decl+ CloseBlock
  ;

Mutual ::=
  'mutual' OpenBlock Decl+ CloseBlock
  ;

Namespace ::=
  'namespace' identifier OpenBlock Decl+ CloseBlock
  ;

Fixity ::=
  FixityType Natural_t OperatorList Terminator
  ;

FixityType ::=
  'infixl'
  | 'infixr'
  | 'infix'
  | 'prefix'
  ;

OperatorList ::=
  Operator_t
  | Operator_t ',' OperatorList
  ;

MethodsBlock ::= 'where' OpenBlock FnDecl* CloseBlock
           ;

Class ::=
  DocComment_t?  Accessibility? 'class' ConstraintList? Name ClassArgument* MethodsBlock?
  ;

ClassArgument ::=
   Name
   | '(' Name ':' Expr ')'
   ;

Instance ::=
  'instance' InstanceName? ConstraintList? Name SimpleExpr* MethodsBlock?
  ;

InstanceName ::= '[' Name ']';

FullExpr ::= Expr EOF_t;

Expr ::= Expr';

Expr' ::= {- External (User-defined) Syntax -}
      |   InternalExpr;

InternalExpr ::=
  App
  | MatchApp
  | UnifyLog
  | RecordType
  | SimpleExpr
  | Lambda
  | QuoteGoal
  | Let
  | RewriteTerm
  | Pi
  |DoBlock
  ;

Name ::= IName_t;

OperatorFront ::= (Identifier_t '.')? '(' Operator_t ')';

FnName ::= Name  |OperatorFront;

Accessibility ::= 'public' | 'abstract' | 'private';

FnOpts ::= 'total'
  | 'partial'
  | 'implicit'
  | '%' 'assert_total'
  |'%' 'reflection'
  | '%' 'specialise' '[' NameTimesList? ']'
  ;

NameTimes ::= FnName Natural?;
```

```
NameTimesList ::=
  NameTimes
  | NameTimes ',' NameTimesList
  ;

CaseExpr ::=
  'case' Expr 'of' OpenBlock CaseOption+ CloseBlock;

CaseOption ::=
  Expr '=>' Expr Terminator
  ;


{- NOTE: Consider using OpenBlock CloseBlock for proofs and tactics -}
ProofExpr ::=
  'proof' OpenBlock Tactic'* CloseBlock
  ;

TacticsExpr :=
  'tactics' OpenBlock Tactic'* CloseBlock
;

Tactic' ::=
  Tactic Terminator
;


SimpleExpr ::=
  '![' Term ']'
  | '?' Name
  | % 'instance'
  | 'refl' ('{' Expr '}')?
  | ProofExpr
  | TacticsExpr
  | CaseExpr
  | FnName
  | List
  | Comprehension
  | Alt
  | Idiom
  | '(' Bracketed
  | Constant
  | Type
  | '_|_'
  | '_'
  | {- External (User-defined) Simple Expression -}
  ;

Bracketed ::=
  ')'
  | Expr ')'
  | ExprList ')'
  | Expr '**' Expr ')'
  | Operator Expr ')'
  | Expr Operator ')'
  | Name ':' Expr '**' Expr ')'
  ;



ListExpr ::=
  '[' ExprList? ']'
;

ExprList ::=
  Expr
  | Expr ',' ExprList
  ;

Alt ::= '(|' Expr_List '|)';

Expr_List ::=
  Expr'
  | Expr' ',' Expr_List
  ;

HSimpleExpr ::=
  '.' SimpleExpr
  | SimpleExpr
  ;

MatchApp ::=
  SimpleExpr '<==' FnName
  ;

UnifyLog ::=
  '%' 'unifyLog' SimpleExpr
  ;

App ::=
  'mkForeign' Arg Arg*
  | SimpleExpr Arg+
  ;

Arg ::=
  ImplicitArg
  | ConstraintArg
  | SimpleExpr
  ;
```

```
ImplicitArg ::=
  '{' Name ('=' Expr)? '}'
  ;

ConstraintArg ::=
  '@{' Expr '}'
  ;

RecordType ::=
  'record' '{' FieldTypeList '}';

FieldTypeList ::=
  FieldType
  | FieldType ',' FieldTypeList
  ;

FieldType ::=
  FnName '=' Expr
  ;

TypeSig ::=
  ':' Expr
  ;

TypeExpr ::= ConstraintList? Expr;

Lambda ::=
    '\\' TypeOptDeclList '=>' Expr
  | '\\' SimpleExprList  '=>' Expr
  ;

SimpleExprList ::=
  SimpleExpr
  | SimpleExpr ',' SimpleExprList
  ;

RewriteTerm ::=
  'rewrite' Expr ('==>' Expr)? 'in' Expr
  ;

Let ::=
  'let' Name TypeSig'? '=' Expr 'in' Expr
| 'let' Expr'                '=' Expr' 'in' Expr

TypeSig' ::=
  ':' Expr'
  ;

QuoteGoal ::=
  'quoteGoal' Name 'by' Expr 'in' Expr
  ;

Pi ::=
    '|'? Static? '('           TypeDeclList ')' DocComment '->' Expr
  | '|'? Static? '{'           TypeDeclList '}'            '->' Expr
  |              '{' 'auto'    TypeDeclList '}'            '->' Expr
  |              '{' 'default' TypeDeclList '}'            '->' Expr
  |              '{' 'static'               '}' Expr'      '->' Expr
  ;

ConstraintList ::=
    '(' Expr_List ')' '=>'
  | Expr              '=>'
  ;

UsingDeclList ::=
  UsingDeclList'
  | NameList TypeSig
  ;

UsingDeclList' ::=
  UsingDecl
  | UsingDecl ',' UsingDeclList'
  ;

NameList ::=
  Name
  | Name ',' NameList
  ;

UsingDecl ::=
  FnName TypeSig
  | FnName FnName+
  ;

TypeDeclList ::=
    FunctionSignatureList
  | NameList TypeSig
  ;

FunctionSignatureList ::=
    Name TypeSig
  | Name TypeSig ',' FunctionSignatureList
  ;

TypeOptDeclList ::=
    NameOrPlaceholder TypeSig?
  | NameOrPlaceholder TypeSig? ',' TypeOptDeclList
```

```
  ;

NameOrPlaceHolder ::= Name | '_';

Comprehension ::= '[' Expr '|' DoList ']';

DoList ::=
    Do
  | Do ',' DoList
  ;

Do' ::= Do Terminator;

DoBlock ::=
  'do' OpenBlock Do'+ CloseBlock
  ;

Do ::=
    'let' Name  TypeSig? '=' Expr
  | 'let' Expr'                '=' Expr
  | Name  '<-' Expr
  | Expr' '<-' Expr
  | Expr
  ;

Idiom ::= '[|' Expr '|]';

Constant ::=
    'Integer'
  | 'Int'
  | 'Char'
  | 'Float'
  | 'String'
  | 'Ptr'
  | 'Bits8'
  | 'Bits16'
  | 'Bits32'
  | 'Bits64'
  | 'Bits8x16'
  | 'Bits16x8'
  | 'Bits32x4'
  | 'Bits64x2'
  | Float_t
  | Natural_t
  | String_t
  | Char_t
  ;

Static ::=
  '[' static ']'
;

Record ::=
    DocComment Accessibility? 'record' FnName TypeSig 'where' OpenBlock Constructor Terminator CloseBlock;

DataI ::= 'data' | 'codata';

Data ::= DocComment? Accessibility? DataI FnName TypeSig ExplicitTypeDataRest?
       | DocComment? Accessibility? DataI FnName Name*   DataRest?
       ;

Constructor' ::= Constructor Terminator;

ExplicitTypeDataRest ::= 'where' OpenBlock Constructor'* CloseBlock;

DataRest ::= '=' SimpleConstructorList Terminator
           | 'where'!
           ;

SimpleConstructorList ::=
    SimpleConstructor
  | SimpleConstructor '|' SimpleConstructorList
  ;

Constructor ::= DocComment? FnName TypeSig;
SimpleConstructor ::= FnName SimpleExpr* DocComment?

Overload' ::= Overload Terminator;

DSL ::= 'dsl' FnName OpenBlock Overload'+ CloseBlock;

OverloadIdentifier ::= 'let' | Identifier;

Overload ::= OverloadIdentifier '=' Expr;

Pattern ::= Clause;

CAF ::= 'let' FnName '=' Expr Terminator;

ArgExpr ::= HSimpleExpr | {- In Pattern External (User-defined) Expression -};

ImplicitOrArgExpr ::= ImplicitArg | ArgExpr;

RHS ::= '='            Expr
      |  '?='  RHSName? Expr
      | 'impossible'
      ;

RHSName ::= '{' FnName '}';
```

```
RHSOrWithBlock ::= RHS WhereOrTerminator
                | 'with' SimpleExpr OpenBlock FnDecl+ CloseBlock
                ;

Clause ::=                                                                   WExpr+ RHSOrWithBlock
       |   SimpleExpr '<=='  FnName                                                 RHS WhereOrTerminator
       |                     FnName ConstraintArg* ImplicitOrArgExpr*    WExpr* RHSOrWithBlock
       |   ArgExpr Operator ArgExpr                                      WExpr* RHSOrWithBlock
       ;


WhereOrTerminator ::= WhereBlock  |Terminator;

WExpr ::= '|' Expr';

WhereBlock ::= 'where' OpenBlock Decl+ CloseBlock;

Codegen ::= 'C'
        |   'Java'
        |   'JavaScript'
        |   'Node'
        |   'LLVM'
        |   'Bytecode'
        ;

Totality ::= 'partial' | 'total'

StringList ::=
  String
  | String ',' StringList
  ;

Directive ::= '%' Directive';

Directive' ::= 'lib'      CodeGen String_t
           |   'link'     CodeGen String_t
           |   'flag'     CodeGen String_t
           |   'include'  CodeGen String_t
           |   'hide'     Name
           |   'freeze'   Name
           |   'access'   Accessibility
           |   'default'  Totality
           |   'logging'  Natural
           |   'dynamic'  StringList
           |   'language' 'TypeProviders'
           ;

Provider ::= '%' 'provide' '(' FnName TypeSig ')' 'with' Expr;

Transform ::= '%' 'transform' Expr '==>' Expr

Tactic ::= 'intro' NameList?
       |   'intros'
       |   'refine'      Name Imp+
       |   'mrefine'     Name
       |   'rewrite'     Expr
       |   'equiv'       Expr
       |   'let'         Name ':' Expr' '=' Expr
       |   'let'         Name           '=' Expr
       |   'focus'       Name
       |   'exact'       Expr
       |   'applyTactic' Expr
       |   'reflect'     Expr
       |   'fill'        Expr
       |   'try'         Tactic '|' Tactic
       |   '{' TacticSeq '}'
       |   'compute'
       |   'trivial'
       |   'solve'
       |   'attack'
       |   'state'
       |   'term'
       |   'undo'
       |   'qed'
       |   'abandon'
       |   ':' 'q'
       ;

Imp ::= '?' | '_';

TacticSeq ::=
    Tactic ';' Tactic
  | Tactic ';' TacticSeq
  ;

{- Open Block Close Block should match i.e. either  {}or indentation -}

OpenBlock ::= '{'
          | {- Same Indentation Level Or Greater -}
          ;

CloseBlock ::= '}'
           | {- Smaller Indentation Level Than Before -}
           ;

Terminator ::= ';'
          |  {- Smaller inden than before -}
          |  {- '|', ')' or '}' is at start of input -}
```

```
            |  eof
            ;

Float_t ::= {- Float literal similar to Haskell -}
          ;

IName_t ::= {- Any valid identifier except keywords possibly prefixed with a namespace -}
       ;

Integer_t ::= {- Integer literal similar to Haskell -}
       ;

Char_t ::= {- Char literal similar to Haskell -}
       ;

Natural_t ::= {- Natural number literal i.e. [1-9]*[0-9] -}
           ;
```