CSci 450, Org. of Programming Languages

Chapter 10
Expression Language Syntax and Semantics

H. Conrad Cunningham

7 November 2017

Contents

Advisory: The HTML version of this document may require use of a browser that supports the display of MathML. A good choice as of November 2017 is a recent version of Firefox from Mozilla.

TODO:

- Add chapter outcomes, etc.
- Make sure the examples and exercises to use a consistent set of features
- Consider various functions on ASTs – size, determine free variables, convert to infix and prefix concrete syntax
- Consider an alternative evaluator that uses substitution in a separate phase
- Consider adding exercises for the concrete syntax and abstract syntax
- Add a section on the REPL (Read-Evaluate-Print-Loop) textual interface

10.1 Chapter Introduction

The case study in this and the following two chapters (Chapters 10-12) examines how we can represent and process simple arithmetic expressions using Haskell. We call the language used in this case study the ***Expression Language***.

Chapter 10 examines two different concrete syntaxes for expressions written as text and an abstract syntax represented as an algebraic data type. Chapter 11 considers the parsing of the syntaxes.

Chapter 10 explores direct interpretation (evaluation) of the abstract syntax trees (ASTs) and manipulation of the trees to simplify or differentiate the expressions.

Chapter 12 looks at a simple Stack Virtual Machine with an instruction set represented as another algebraic data type and how to translate (i.e., compile), how to execute the machine, and how to translate the abstract syntax trees to sequences of instructions.

We will extend it with other features in later chapters.

TODO: Add outcomes

10.2 Concrete Syntax

The Expression Language can be represented in human-readable text strings in forms similar to traditional mathematical and programming notations. The structure of these textual expressions is called the *concrete syntax* of the expressions.

In this case study, we examine two possible concrete syntaxes: a familiar infix syntax and a (probably less familiar) parenthesized prefix syntax.

But, first, let's consider how we can describe the syntax of a language.

10.2.1 Grammars

We usually describe the syntax of a language using a *grammar*.

Formally, a grammar consists of a tuple (V, T, S, P), where

- V is a finite set of *variable* (or *nonterminal*) symbols
- T is a finite set of *terminal* symbols (called the *alphabet*)
- $S \in V$ is the *start* (or *goal*) symbol
- P is a finite set of *production* rules
- V and T are disjoint

Production rules describe how the grammar transforms one sequence of symbols to another. The rules have the general form

$$x \to y$$

where x and y are sequences of symbols from $V \cup T$ such that x has length of at least one symbol.

A *sentence* in a language consists of any finite sequence of terminal symbols that can be generated from the start symbol of a grammar by a finite sequence of productions from the grammar.

We call a sequence of productions that generates a sentence a *derivation* for that sentence.

Any intermediate sequence of symbols in a derivation is called a *sentential form*.

The *language* generated by the grammar is the set of all sentences that can be generated by the grammar.

10.2.1.1 Context-free grammars and BNF

To express the syntax of programming languages, we normally restrict ourselves to the family of *context-free grammars* (and its subfamilies). In a context-free grammar (CFG), the production rules have the form

$$A \to y$$

where $A \in V$ and y is a sequence of zero or more symbols from $V \cup T$. This means that an occurence of nonterminal A can be replaced by the sequence x.

We often express a grammar using a metalanguage such as the *Backus-Naur Form (BNF)*.

For example, consider the following BNF description of a grammar for the unsigned binary integers:

```
<binary> ::= <digit>
<binary> ::= <digit> <binary>
<digit>  ::= '0'
```

```
<digit>  ::= '1'
```

The nonterminals are the symbols shown in angle brackets: `<binary>` and `<digit>`.

The terminals are the symbols shown in single quotes: `'0'` and `'1'`.

The production rules are shown with a nonterminal on the left side of the metasymbol `::=` and its replacement sequence of nonterminal and terminal symbols on the right side.

Unless otherwise noted, the start symbol is the nonterminal on the left side of the first production rule.

For multiple rules with the same left side, we can use the `|` metasymbol to write the alternative right sides concisely. The four rules above can be written as follows:

```
<binary> ::= <digit> | <digit> <binary>
<digit>  ::= '0' | '1'
```

We can also use the extended BNF metasymbols:

- `{` and `}` to denote that the symbols between the braces are repeated zero or more times
- `[` and `]` to denote that the symbols between the brackets are optional (i.e., occur at most once)

10.2.1.2 Derivations

Consider a derivation of the sentence `101` using the grammar for unsigned binary numbers above.

- Start symbol – `<binary>`
- Apply rule 2 – `<digit> <binary>`
- Apply rule 2 – `<digit> <digit> <binary>`
- Apply rule 3 – `<digit> 0 <binary>`
- Apply rule 4 – `1 0 <binary>`
- Apply rule 1 – `1 0 <digit>`
- Apply rule 4 – `1 0 1`

This is not the only possible derivation for `101`. Let's consider a second derivation of `101`.

- Start symbol – `<binary>`
- Apply rule 2 – `<digit> <binary>`
- Apply rule 4 – `1 <binary>`
- Apply rule 2 – `1 <digit> <binary>`
- Apply rule 3 – `1 0 <binary>`
- Apply rule 1 – `1 0 <digit>`

- Apply rule 4 – `1 0 1`

The second derivation applies the same rules the same number of times, but it applies them in a different order. This case is called the *leftmost derivation* because it always replaces the leftmost nonterminal in the sentential form.

Both of the above derivations can be represented by the *derivation tree* (or *parse tree*) shown in Figure 1. (The numbers below the nodes show the rules applied.)

10.2.1.3 Regular grammars

The grammar above for binary numbers is a special case of a context-free grammar called a *right-linear grammar*. In a right-linear grammar, *all* productions are of the forms

$A \to xB$
$A \to x$

where A and B are nonterminals and x is a sequence of zero or more terminals. Similarly, a *left-linear grammar* must have *all* productions of the form:

$A \to Bx$
$A \to x$

A grammar that is either right-linear or left-linear is called a *regular grammar*. (Note that *all* productions in a grammar must satisfy either the right- or left-linear definitions. They cannot be mixed.)

We can recognize sentences in a regular grammar with a simple "machine" (program) – a *deterministic finite automaton* (dfa).

In general, we must use a more complex "machine" – a *pushdown automaton (PDA)*– to recognize a context-free grammar.

We leave a more detailed study of regular and context-free grammars to courses on formal languages, automata, or compiler construction.

Now let's consider the concrete syntaxes for the Expression language–first infix, then prefix.

10.2.2 Infix syntax

An *infix syntax* for expressions is a syntax in which most binary operators appear between their operands as we tend to write them in mathematics and in programming languages such as Java and Haskell. For example, the following are intended to be valid infix expressions:

```
3
-3
x
```

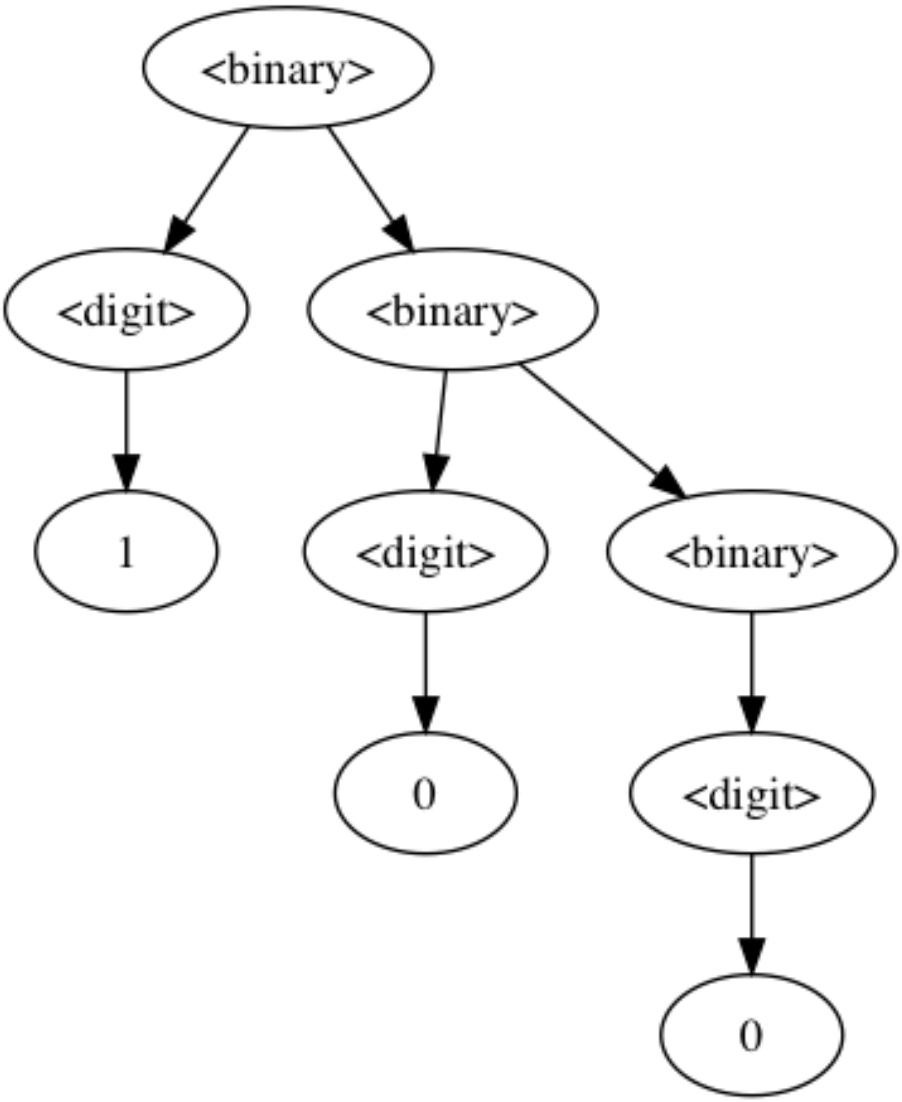

Figure 1: Derivation (parse) tree for binary number 101

```
1+1
x + 3
(x + y)  * (2 + z)
```

For example, we can present the concrete syntax of our core Expression Language with the grammar below. Here we just consider expressions made up of decimal integer constants; variable names; binary operators for addition, subtraction, multiplication, and division; and parentheses to delimit nested expressions.

We express the upper levels of the infix expression's syntax with the following context-free grammar where `<expression>` is the start symbol.

```
<expression> ::= <term> { <addop> <term> }
<term>       ::= <factor> { <mulop> <factor> }
<factor>     ::= <var> | <val>
                 | '(' <expression> ')'
<val>        ::= [ '-' ] <unsigned>
<var>        ::= <id>
<addop>      ::=  '+'  |  '-'
<mulop>      ::=  '*'  |  '/'
```

Normally we want operators such as multiplication and division to bind more tightly than addition and subtraction. That is, we want expression `x + y * z` to have the same meaning as `x + (y * z)`. To accomplish this in the context-free grammar, we position `<addop>` in a higher-level grammar rule than `<mulop>`.

We can express the lower (lexical) level of the expression's grammar with the following production rules:

```
<id>         ::=  <firstid>  |  <firstid> <idseq>
<idseq>      ::=  <restid>   |  <restid> <idseq>
<firstid>    ::=  <alpha>    | '_'
<restid>     ::=  <alpha>    | '_'  | <digit>
<unsigned>   ::=  <digit>    |  <digit> <unsigned>
<digit>      ::=  any numeric character
<alpha>      ::=  any alphabetic character
```

The variables `<digit>` and `<alpha>` are essentially terminals. Thus the above is a regular grammar. (We can also add the rules for recognition of `<addop>` and `<mulop>` and rules for recognition of the terminals `(`, `)`, and `-` to the regular grammar.)

We assume that identifiers and constants extend as far to the "right" as possible. That is, an `<id>` begins with an alphabetic or underscore character and extends until it is terminated by some character other than an alphabetic, numeric, or underscore character (e.g., by whitespace or special character). Similarly for `<unsigned>`.

Otherwise, the language grammar ignores whitespace characters (e.g., blanks, tabs, and newlines). The language also supports end of line comments, any

characters on a line following a `--` (double dash).

We can use a *parsing* program (i.e., a *parser*) to determine whether a concrete expression (e.g., `1 + 1`) satisfies the grammar and to build a corresponding *parse tree*.

Aside: In a previous subsection, we use the term derivation tree to refer to a tree that we construct from the root toward the leaves by applying production rules from the grammar. We usually call the same tree a parse tree if we construct it from the leaves (a sentence) toward the root.

Figure 2 shows the parse tree for infix expression `1 + 1`. It has `<expression>` at its root. The children of a node in the parse tree depend upon the grammar rule application needed to generate the concrete expression. Thus the root `<expression>` has either one child–a `<term>` subtree–or three children– a `<term>` subtree, an `<addop>` subtree, and an `<expression>` subtree.

If the parsing program returns a boolean result instead of building a parse tree, we sometimes call it a *recognizer* program.

10.2.3 Prefix syntax

An alternative is to use a *parenthesized prefix syntax* for the expressions. This is a syntax in which expressions involving operators are of the form

```
( op operands )
```

where `op` denotes some "operator" and `operands` denotes a sequence of zero or more expressions that are the arguments of the given operator. This is a syntax similar to the language Lisp.

In this syntax, the examples from the subsection on the infix syntax can be expressed something like:

```
3
3
x
(+ 1 1)
(+ x 3)
(* (+ x y) (+ 2 z))
```

We express the upper levels of a prefix expression's syntax with the following context-free grammar, where `<expression>` is the start symbol.

```
<expression> ::=  <var> | <val> | <operexpr>
<var>        ::=  <id>
<val>        ::=  [ "-" ] <unsigned>
<operexpr>   ::=  '(' <operator> <operandseq> ')'
<operandseq> ::=  { <expression> }
<operator>   ::=  '+'  |  '*'  |  '-'  |  '/' | ...
```

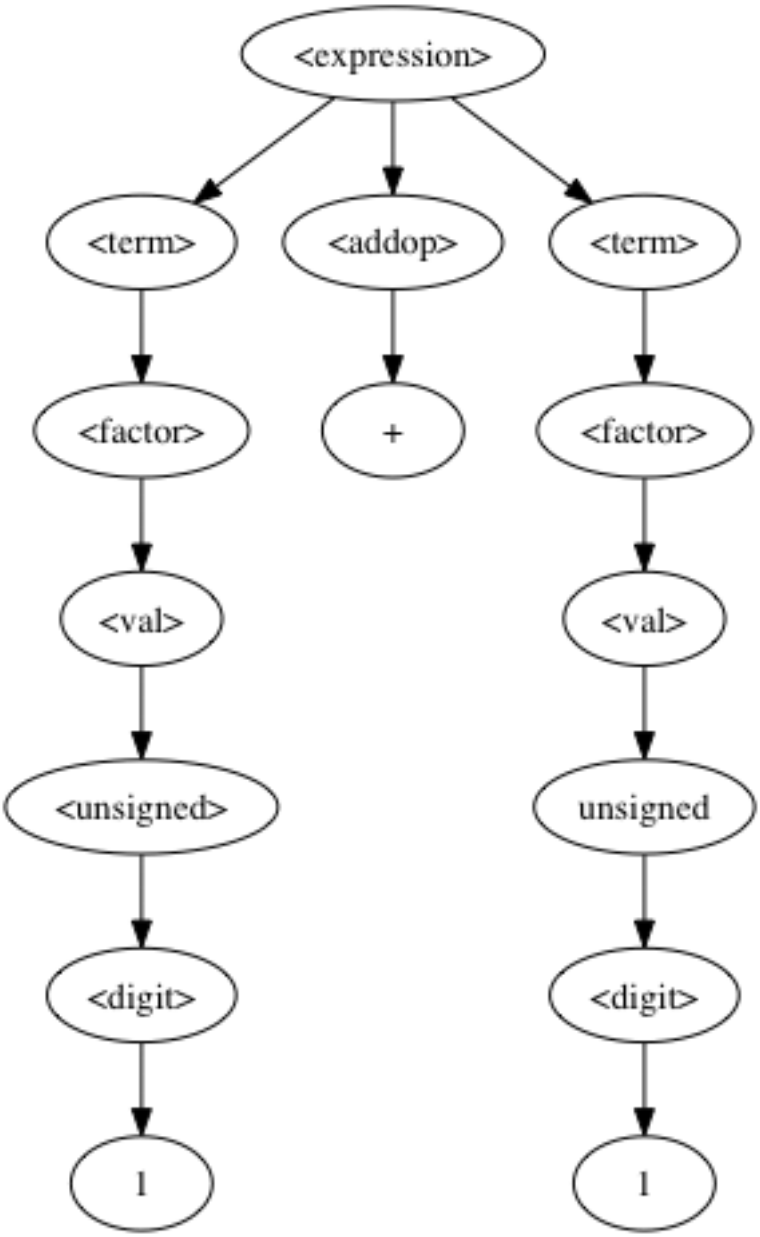

Figure 2: Parse tree for infix 1 + 1

We can express the lower (lexical) level of the expression's grammar with basically the same regular grammar as with the infix syntax. (We can also add the rule for recognition of `<operator>` and for recognition of the terminals `(`, `)`, and `-` to the regular grammar

The parse tree for prefix expression `(+ 1 1)` is shown in Figure 3.

Because the prefix syntax expresses all operations in a fully parenthesized form, there is no need to consider the binding powers of operators. This makes parsing easier.

The prefix also makes extending the language to other operators–and keywords–much easier. Thus we will primarily use the prefix syntax in this and other cases studies.

We return to the problem of parsing expressions in a later subsection.

10.3 Abstract Syntax

The *abstract syntax* of an expression seeks to represent only the essential aspects of the expression's structure, ignoring nonessential, representation-dependent details of the concrete syntax.

For example, parentheses represent structural details in the concrete syntaxes given in the previous subsections. This structural information can be represented directly in the abstract syntax; there is no need for parentheses to appear in the abstract syntax.

We can represent arithmetic expressions conveniently using a tree data structure, where the nodes represent operations (e.g., addition) and leaves represent values (e.g., constants or variables). This representation is called a *abstract syntax tree* for the expression.

10.3.1 Abstract syntax tree data type

In Haskell, we can represent an abstract syntax trees using algebraic data types. Such types often enable us to express programs concisely by using pattern matching.

For the Expression Language, we define the `Expr` data type (in the `AbSynExpr` module) to describe the abstract syntax tree.

```
import Values ( ValType, Name )

data Expr = Add Expr Expr
          | Sub Expr Expr
          | Mul Expr Expr
          | Div Expr Expr
```

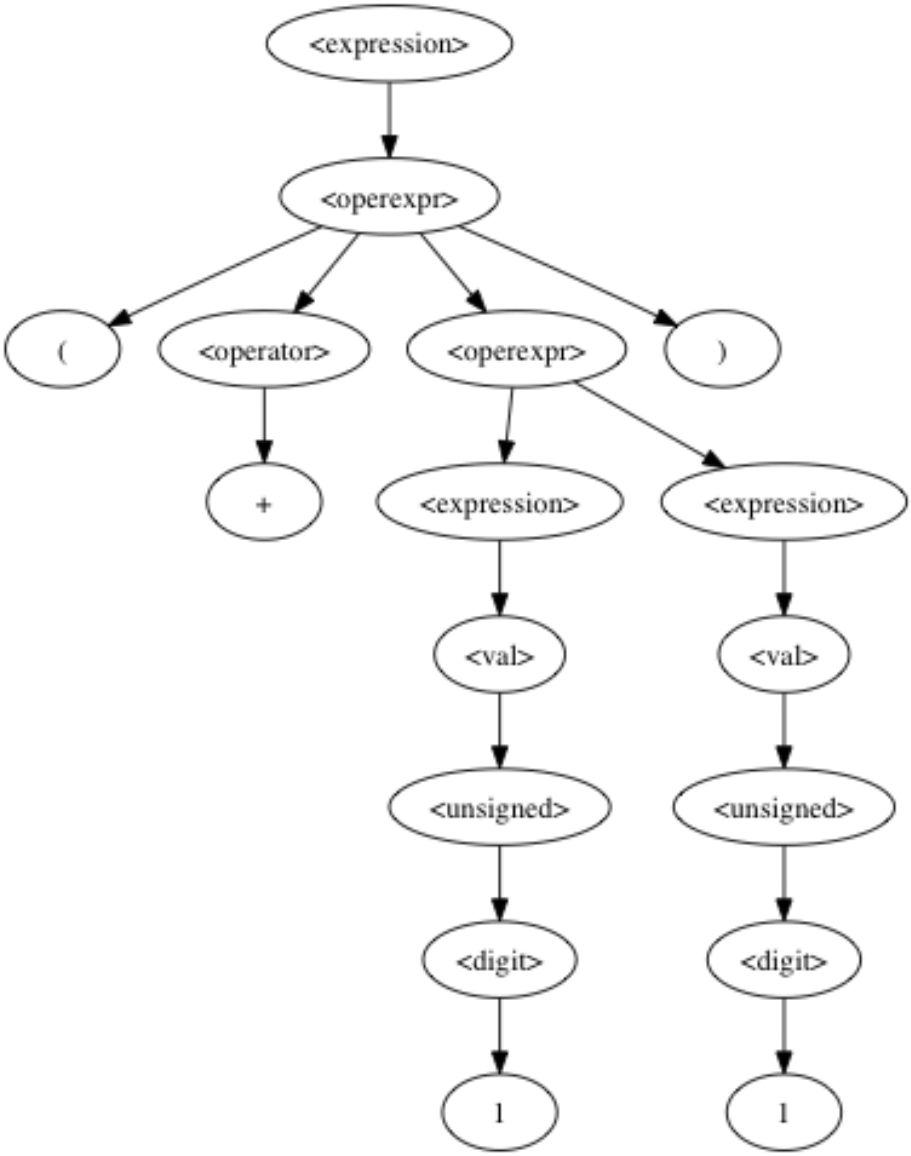

Figure 3: Parse tree for prefix (+ 1 1)

```
          | Var Name
          | Val ValType
            -- deriving Show?

instance Show Expr where
    show (Val v)   = show v
    show (Var n)   = n
    show (Add l r) = showParExpr "+" [l,r]
    show (Sub l r) = showParExpr "-" [l,r]
    show (Mul l r) = showParExpr "+" [l,r]
    show (Div l r) = showParExpr "/" [l,r]

showParExpr :: String -> [Expr] -> String
showParExpr op es =
    "(" ++ op ++ " " ++ showExprList es ++ ")"

showExprList :: [Expr] -> String
showExprList es = Data.List.intercalate " " (map show es)
```

Above in type `Expr`, the constructors `Add`, `Sub`, `Mul`, and `Div` represent the addition, subtraction, multiplication, and division, respectively, of the two operand subexpressions, `Var` represents a variable with a name, and `Val` represents a constant value.

Note that this abstract syntax is similar to the (Lisp-like) parenthesized prefix syntax described in the previous subsection.

We make type `Expr` an instance of class `Show`. We do not derive or define an instance of the `Eq` class because direct structural equality of trees may not be how we want to define equality comparisons.

We can thus express the example expressions from the Concrete Syntax subsection as follows:

```
Val 3                   -- 3
Val (-3)                -- -3
Var "x"                 -- x
Add (Val 1) (Val 1)     -- 1+1
Add (Var "x") (Val 3)   -- x + 3
                        -- (x + y) * (2 - z)
Mul (Add (Var "x") (Var "y")) (Sub (Val 2) (Var "z"))
```

Figures 5 and 6 show abstract syntax trees for two example expressions above.

In the following chapter on parsing, we develop parsers for both the prefix and infix syntaxes. Both parsers construct abstract syntax trees using the algebraic data type `Expr`.

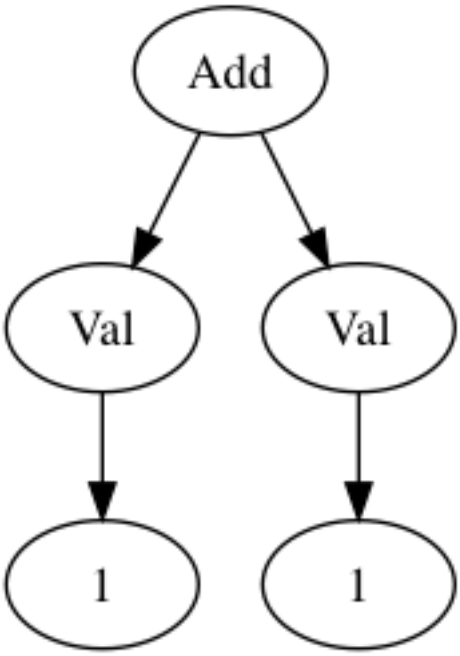

Figure 5: Abstract syntax tree for `1 + 1` and `(+ 1 1)`

10.3.2 Values and variable names

The Expression Language restricts values to `ValType`. The `Values` module indirectly defines this type synonym to be `Int`.

The abstract syntax allows a name to be represented by any string (i.e., type alias `Name`, which is defined to be `String` in the `Values` module). We likely want to restrict names to follow the usual "identifier" syntax. The parser for the concrete syntax should enforce this restriction. Or we could define Haskell functions to parse and construct identifiers, such as the functions below.

```
import Data.Char ( isAlpha, isAlphaNum )

getId :: String -> (Name,String)
getId []          = ([],[])
getId xs@(x:_)
    | isFirstId x = span isRestId xs
    | otherwise   = ([],xs)
where
    isFirstId c = isAlpha c    || c == '_'
    isRestId  c = isAlphaNum c || c == '_'
```

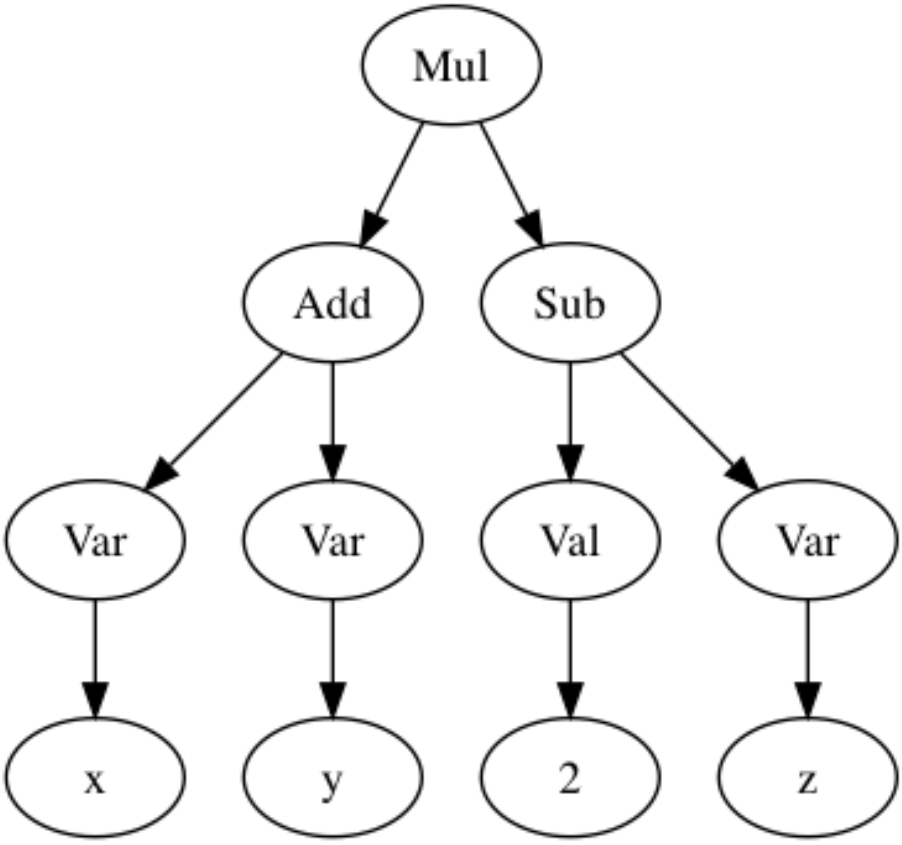

Figure 6: Abstract syntax tree for `(x + y) * (2 - z)` and `(* (+ x y) (- 2 z))`

```
identifier :: String -> Maybe Name
identifier xs =
    case getId xs of
        (xs@(_:_),[]) -> Just xs
        otherwise     -> Nothing
```

The `getId` function takes a string and parses an identifier at the beginning of the string. A valid identifier must begin with an alphabetic or underscore character and continue with zero or more alphabetic, numeric, or underscore characters.

The `getId` function uses the higher order function `span` to collect the characters that form the identifier. This function takes a predicate and returns a pair, of which the first component is the prefix string satisfying the predicate and the second is the remaining string.

In the following chapter, we examine how to parse an expression's concrete syntax to build an abstract syntax tree.

10.4 Associative Data Structures

TODO: Modify this subsection to account for the introduction of this concept in the Algebraic Data Type chapter.

In language processing, we often need to associate some key (e.g., a variable name) with its value. There are several names for this type of data structure – *associative array*, *dictionary*, *map*, *symbol table*, etc.

As we saw in a previous chapter, an *association list* is a simple list-based implementation of this concept. It is a list of pairs in which the first component is the *key* (e.g., a string) and the second component is the *value* associated with the key.

The Prelude function `lookup`, shown below, searches an association list for a key and returns a `Maybe` value. If it finds the key, it wraps the associated value in a `Just`; if it does not find the key, it returns a `Nothing`.

```
lookup :: (Eq a) => a -> [(a,b)] -> Maybe b
lookup _   []   =  Nothing
lookup key ((x,y):xys)
    | key == x  =  Just y
    | otherwise =  lookup key xys
```

For better performance with larger dictionaries, we can replace an association list by a more efficient data structure such as a `Data.Map.Map`. This structure implements the dictionary structure as a size-balanced tree. It provides a `lookup` function with essentially the same interface.

Of course, imperative languages might use a mutable *hash table* to implement a dictionary.

10.5 Semantics

Consider the evaluation of the Expression Language's abstract syntax trees as defined above.

10.5.1 Environments

To evaluate an expression, we must determine the current value of each variable occurring in the expression. That is, we must evaluate the expression in some *environment* that associates the variable names with their values.

For example, consider the expression `x + 3`. It might be evaluated in an environment that associates the value `5` with the variable `x`, written `{ x -> 5 }`. The evaluation of this expression yields the value `8`.

The environment `{ x -> 5 }` can be expressed in a number of ways in Haskell. Here we choose to represent it as a simple association list as follows:

```
[("x",5)]
```

This list associates a variable name in the first component with its integer value in the second component.

Looking up a key in an association list is an `O(n)` operation where `n` denotes the number of key-value pairs.

A good alternative to the association list is a `Map` from the `Data.Map` library. It implements the dictionary as a size-balanced tree, thus its `lookup` function is an `O(log n)` operation.

In the Expression Language implementation, we encapsulate the representation of the environment in the `Environments` module. This module exports the following type synonym and functions:

```
type AnEnv a = [(Name,a)]

newEnv     :: AnEnv a
toList     :: AnEnv a -> [(Name,a)]
getBinding :: Name -> AnEnv a -> Maybe a
hasBinding :: Name -> AnEnv a -> Bool
newBinding :: Name -> a -> AnEnv a -> AnEnv a
setBinding :: Name -> a -> AnEnv a -> AnEnv a
bindList   :: [(Name,a)] -> AnEnv a -> AnEnv a
```

For the purposes of our evaluation program, we can then define a specific environment with the type synonym `Env` in the `EvalExpr` module:

```
import Values       ( ValType, Name, defaultVal )
import AbSynExpr    ( Expr(..) )
import Environments ( AnEnv, Name, newEnv, toList, getBinding,
```

```
                          hasBinding, newBinding, setBinding, bindList )

type Env = AnEnv ValType
```

10.5.2 Values of AST nodes

We express the *semantics* (i.e., meaning) of the various Expression Language expressions (i.e., nodes of the AST) as follows.

- `Val c` evaluates to the constant (`NumType`) value `c`.
- `Var n` evaluates to the value of variable `n` in the environment, generating an error if the variable is not defined.
- `Add l r` evaluates to the sum of the values of the expression trees `l` and `r`.
- `Sub l r` evaluates to the difference between the values of the expression trees `l` and `r`.
- `Mul l r` evaluates to the product of the values of the expression trees `l` and `r`.
- `Div l r` evaluates to the quotient of the values of the expression trees `l` and `r`.

10.5.3 Evaluation function

We can thus define a Haskell evaluation function (i.e., interpreter) for the Expression Language as follows.

This function in the `EvalExpr` module does a *post-order traversal* of the abstract syntax tree, first computing the values of the child subexpressions and then computing the value of of a node. The value is returned wrapped in an `Either`, where the `Left` constructor represents and error message and the `Right` constructor a good value.

```
import Values       ( ValType, Name, defaultVal )
import AbSynExpr    ( Expr(..) )
import Environments ( AnEnv, Name, newEnv, toList, getBinding,
                      hasBinding, newBinding, setBinding, bindList )
type EvalErr = String
type Env     = AnEnv ValType

eval :: Expr -> Env -> Either EvalErr ValType
eval (Val v) _   = Right v
eval (Var n) env =
    case getBinding n env of
        Nothing -> Left ("Undefined variable " ++ n)
```

```
        Just i  -> Right i
eval (Add l r) env =
    case (eval l env, eval r env) of
        (Right lv, Right rv) -> Right (lv + rv)
        (Left le,  Left re ) -> Left (le ++ "\n" ++ re)
        (x@(Left le),  _   ) -> x
        (_,     y@(Left le)) -> y
eval (Sub l r) env =
    case (eval l env, eval r env) of
        (Right lv, Right rv) -> Right (lv - rv)
        (Left le,  Left re ) -> Left (le ++ "\n" ++ re)
        (x@(Left le),  _   ) -> x
        (_,     y@(Left le)) -> y
eval (Mul l r) env =
    case (eval l env, eval r env) of
        (Right lv, Right rv) -> Right (lv * rv)
        (Left le,  Left re ) -> Left (le ++ "\n" ++ re)
        (x@(Left le),  _   ) -> x
        (_,     y@(Left le)) -> y
eval (Div l r) env =
    case (eval l env, eval r env) of
        (Right _,  Right 0 ) -> Left "Division by 0"
        (Right lv, Right rv) -> Right (lv `div` rv)
        (Left le,  Left re ) -> Left (le ++ "\n" ++ re)
        (x@(Left le),  _   ) -> x
        (_,     y@(Left le)) -> y
```

Consider an example with a simple `main` function below (that could be added to the `EvalExpr` module) that evaluates the example expressions from a previous section. (See the extended `EvalExpr` module.)

```
main =
  do
    let env = [("x",5), ("y",7),("z",1)]
    let exp1 = Val 3           -- 3
    let exp2 = Var "x"         -- x
    let exp3 = Add (Val 1) (Val 2)     -- 1+2
    let exp4 = Add (Var "x") (Val 3)   -- x + 3
    let exp5 = Mul (Add (Var "x") (Var "y"))
                   (Add (Val 2) (Var "z")) -- (x + y) * (2 + z)
    putStrLn ("Expression: " ++ show exp1)
    putStrLn ("Evaluation with x=5, y=7, z=1:  "
              ++ show (eval exp1 env))
    putStrLn ("Expression: " ++ show exp2)
    putStrLn ("Evaluation with x=5, y=7, z=1:  "
              ++ show (eval exp2 env))
    putStrLn ("Expression: " ++ show exp3)
```

```
putStrLn ("Evaluation with x=5, y=7, z=1:  "
          ++ show (eval exp3 env))
putStrLn ("Expression: " ++ show exp4)
putStrLn ("Evaluation with x=5, y=7, z=1:  "
          ++ show (eval exp4 env))
putStrLn ("Expression: " ++ show exp5)
putStrLn ("Evaluation with x=5, y=7, z=1:  "
          ++ show (eval exp5 env))
```

It first computes its value in the environment `{ x -> 5, y -> 7 }` and then computes its derivative relative to `x` and then to `y`.

```
Expression: 3
Evaluation with x=5, y=7, z=1:  Right 3
Expression: x
Evaluation with x=5, y=7, z=1:  Right 5
Expression: (+ 1 2)
Evaluation with x=5, y=7, z=1:  Right 3
Expression: (+ x 3)
Evaluation with x=5, y=7, z=1:  Right 8
Expression: (* (+ x y) (+ 2 z))
Evaluation with x=5, y=7, z=1:  Right 36
```

10.6 Read-Evaluate-Print Loop (REPL)

TODO: Write this section

- *Read* expression from command line
- *Evaluate* expression after parsing
- *Print* resulting value
- *Loop* back for next expression

10.7 Simplification

An expression may be more complex than necessary. We can simplify it, especially with the intention "optimizing" its evaluation.

An operation whose operands are constants can be simplified by replacing it by the appropriate constant. For example, `Add (Val 3) (Val 4)` is the same as `Val 7`.

Similarly, we can take advantages of an operation's identity element and other mathematical properties to simplify expressions. For example, `Add (Val 0) (Var "x")` is the same as `Var "x"`.

We can thus define a skeletal function `simplify` as follows. As with `eval`, the `simplify` function traverses the abstract syntax tree using a post-order traversal.

```
simplify :: Expr -> Expr
simplify (Add l r) =
    case (simplify l, simplify r) of
        (Val 0, rr)    -> rr
        (ll, Val 0)    -> ll
        (Val x, Val y) -> Val (x+y)
        (ll, rr)       -> Add ll rr
simplify (Mul l r) =
    case (simplify l, simplify r) of
        (Val 0, rr)    -> Val 0
        (ll, Val 0)    -> Val 0
        (Val 1, rr)    -> rr
        (ll, Val 1)    -> ll
        (Val x, Val y) -> Val (x*y)
        (ll, rr)       -> Mul ll rr
simplify t@(Var _) = t
simplify t@(Val _) = t
```

In an exercise, you are asked to complete the development of this function.

See the incomplete Process AST module `ProcessAST` for the sample code in this subsection and the following one.

10.8 Symbolic Differentiation

Suppose that we redefine the `Expr` type to support double precision floating point (i.e., `Double`) values.

Then let's consider symbolic differentiation of the arithmetic expressions. Thinking back to our study of differential calculus, we identify the following rules for differentiation:

- The derivative of a sum is the sum of the derivatives.
- The derivative of a product of two operands is the sum of the product of (a) the first operand and the derivative of the second and (b) the second operand and the derivative of the first.
- The derivative of some variable `v` is 1 if differentiation is relative to `v` and is 0 otherwise.
- The derivative of a constant is 0.

We can directly translate these rules into a skeletal Haskell function that uses the above data types, as follows:

```haskell
deriv :: Expr -> Name -> Expr
deriv (Add l r) v = Add (deriv l v) (deriv r v)
deriv (Mul l r) v = Add (Mul l (deriv r v)) (Mul r (deriv l v))
deriv (Var n)   v
      | v == n    = Val 1
deriv _         _ = Val 0
```

10.9 Source Code and Module Dependencies

An Expression Language interpreter consists of 7 modules with the module dependencies shown in Figure 6.

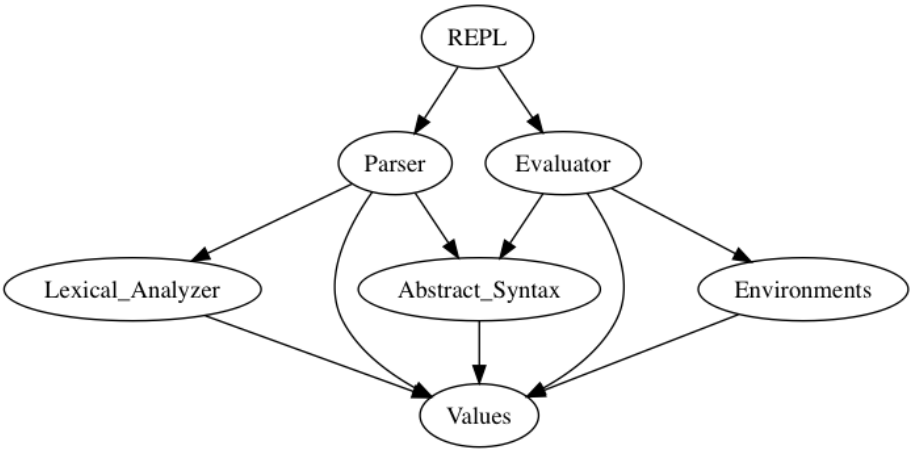

Figure 6: Expression Language module dependencies

The Haskell source code for the Expression Language modules discussed in this chapter are linked below:

- *Evaluator* module `EvalExpr`
- *Abstract_Synax* module `AbSynExpr`
- *Environments* module `Environments`
- *Values* module `Values`

In addition, the code for the partially implemented *Process AST* module includes the skeleton `simplify` and `deriv` code:

- Incomplete Process AST module `ProcessAST` (simplify and deriv)

This module is "wrapper" for the `EvalExpr` module currently.

The source code for the remaining Expression Language modules are discussed in the next chapter. These are linked below:

- *Lexical_analyzer* module `LexExpr` common to both prefix and infix parsers

- Prefix syntax
  - Recursive descent *Parser* module for prefix language `ParsePrefixExpr`
  - *REPL* module for prefix syntax `PrefixExprREPL`
- Infix syntax
  - Recursive descent *Parser* module for infix language `ParseInfixExpr`
  - *REPL* module for infix syntax `InfixExprREPL`

The main entry points for use of the interpreters are the REPLs.

10.10 Exercises

1. Extend the abstract syntax tree data type `Expr` (in the Abstract Syntax module) to add new operations `Neg` (negation), `Min` (minimum), `Max` (maximum), and `Exp` (exponentiation).

   ```
   data Expr = ...
             | Neg Expr
             | Min Expr Expr
             | Max Expr Expr
             | Exp Expr Expr
             ...
               deriving Show
   ```

   Then extend the `eval` function (in the `EvalExpr` module) to add these new operations with the following informal semantics:

   - `Neg e` negates the value of `e` – `Neg (Val 1)` is `(Val (-1))`
   - `Min l r` returns the smaller value of `l` and `r`
   - `Max l r` returns the larger value of `l` and `r`
   - `Exp l r` raises `l` to the poewr `r`

2. Extend the `simplify` function to support `Sub`, `Div`, and the new operations given in the previous exercise.

   This function should simplify the abstract syntax tree by evaluating subexpressions involving only constants (not evaluating variables) and handling special values like identity and zero elements.

3. Extend the `simplify` function from the previous exercise in other ways. For example, take advantage of mathematical properties such as associativity (`(x + y) + z = x + (y + z)`), commutativity (`x + 1 = 1 + x`), and idempotence (`x min x = x`).

4. Extend the abstract syntax tree data type `Expr` to include the binary operators `Eq` (equality) and `Lt` (less-than comparison), logical unary operator `Not`, and the ternary conditional expression `If` (if-then-else).

```
data Expr = ...
          | Eq  Expr Expr
          | Lt  Expr Expr
          | Not Expr
          | If  Expr Expr Expr
          ...
            deriving Show
```

   Then extend the `eval` function to implement these new operations.

   This extended language does not have Boolean values. We represent "false" by integer 0 and "true" by a nonzero integer, canonically by

   1.

   We can express the informal semantics of the new Expression Language expressions as follows:

   - `Eq l r` evaluates to the value 1 if `l` and `r` have the same value and to 0 otherwise.
   - `Lt l r` evaluates to the value 1 if the value of `l` is smaller than the value of `r` and to 0 otherwise.
   - `Not i` evaluates to 1 if `i` is zero and evaluates to 0 if `i` is nonzero.
   - `If c l r` first evaluates `c`; if `c` is nonzero, the `if` evaluates to the value of `l`; otherwise the `if` evaluates to the value of `r`.

   Note: The constants `falseVal` and `trueVal` and the functions `boolToVal` and `valToBool` in the `Values` module may be helpful. (The intention of the `Values` module is to keep the representation of the values hidden from the rest of the interpreter. In particular, these constants and functions these are to help encapsulate the representation of booleans as the underlying values.)

5. Develop an object-oriented program (e.g., in Java) to carry out the same functionality as the `Expr` data type and `eval` function described in this chapter. That is, define a class hierarchy that corresponds to the `Expr` data type and use the message-passing style to implement the needed classes and instances.

6. Extend the object-oriented program from the previous exercise to implement the `Neg`, `Min`, `Max`, and `Exp` as described in an earlier exercise.

7. Extend the object-oriented program from the previous exercise to implement the `Eq`, `Lt`, `Not`, and `If` as described in another earlier exercise.

8. Extend the object-oriented program above to implement simplification.

9. For this exercise, redefine the `Expr` data type above to hold `Double` constants instead of `Int`. In addition to `Add`, `Mul`, `Sub`, `Div`, `Neg`, `Min`, `Max`, and `Exp`, extend the data type and `eval` function to include the trigonometric operators `Sin` and `Cos` for sine and cosine.

10. Using the extended `Double`-version of `Expr` from the previous exercise, extend function `deriv` to support all the operators in the data type.

10.11 Acknowledgements

I initially developed this case study for the Haskell-based offering of CSci 556 (Multiparadigm Programming) in Spring 2017. I continued work on it during Summer and Fall 2017. I based this work, in part, on ideas from:

- the Scala-based Expression Tree Calculator case study from my *Notes on Scala for Java Programmers* (which was itself adapted from the tutorial Scala for Java Programmers in Spring 2016)
- the Lua-based Expression Language 1 and Imperative Core interpreters I developed for the Fall 2016 CSci 450 course
- Samuel N. Kamin's textbook *Programming Languages: An Interpreter-based Approach*, Addison Wesley, 1990 and my work to implement three (Core, Lisp, and Scheme) of these interpeters in Lua in 2013
- Sections 1.2, 3.3, and 5.1 of Peter Linz's textbook *An Introduction to Formal Languages and Automata*, Fifth Edition, Jones and Bartlett, 2017
- Section 1.3 of Peter Sestoff's *Programming Language Concepts*, Springer, 2012.
- the Wikipedia articles on Regular Grammar, Context-Free Grammar, Backus-Naur Form, and Abstract Syntax

I maintain these notes as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the notes to HTML, PDF, and other forms as needed, The HTML version of this document may require use of a browser that supports the display of MathML.

10.12 References

[**Kamin 1990**] Samuel N. Kamin. *Programming Languages: An Interpreter-Based Approach*, Addison-Wesley, 1990.

[**Linz 2017**] Peter Linz. *An Introduction to Formal Languages and Automata*, Fifth Edition, Jones and Bartlett, 2017. (Especially sections 1.2, 3.3, and 5.1)

[**Schinz-Haller 2017**] Michel Schinz and Philipp Haller. A Scala Tutorial for Java Programmers, accessed February 2016.

[**Sestoft 2012**] Peter Sestoft. *Programming Language Concepts*, Springer, 2012.
[**Wikipedia 2017**] Wikipedia articles "Regular Grammar", "Context-Free Grammar", "Backus-Naur Form", "Lexical Analysis", "Parsing", "LL Parser", "Recursive Descent Parser", and "Abstract Syntax".

10.13 Terms and Concepts

TODO