Structured Type System

Abstract:
...
As every theory needs some syntax form to express its elements, a road to a theory about theories leads through a syntax defining land, so structured type system, in the first place, provides a flexible generalized text parser that builds up internal abstract syntax trees (AST) from input data. The other aspect of theory about theories inevitably covers the meaning of input data. This is called semantics, and this is the point where structured type system provides a possibility to define deeper connections between syntactic elements of AST-s. For this purpose, structured type system uses a kind of functions known from functional programming paradigm. These functions are able to process any data corpus, being natural or artificial language translation, which in turn happens to be just enough for running any complexity task used to analyze existing and calculate new data from an input.
...

Table Of Contents

1. Introduction

So, what is structured type system really? It all began with an attempt to construct a formal system for describing knowledge about the Universe. It turned out that it is a complex task and after a lot of tries, and a lot of failures, finally something useful came out: a kind of **structured type system**, that could be used as a basis for a solid declarative programming language that can describe dynamic states of the Universe.

Structured type system could be also a bouncing board for artificial intelligence related tasks such are analyzing and concluding knowledge about the Universe. To program true artificial intelligence, we need a way to store the knowledge that comes in forms of theories. Therefore, we need a general theory about theories, and that is what structured type system is about to offer. We want to support scientific use for deduction, induction, and automatic problem solver, but we also want to support the real world use, that is programming computer applications. Actually, structured type system possesses some thrilling properties for developers like automatic language translation (transpiling) and support for easy syntax extending of arbitrary programming languages. From a regular programming aspect, structured type system could be used for programming, building new programming languages, or placing new data exchange standards. From a scientific aspect, structured type system could be used for explaining the Universe, and for seeking new knowledge about it. I guess the point is: if structured type system could be used for explaining the Universe, then structured type system could be used for expressing the real world situations managed by usual computer applications. Our task is to make it suitable for both purposes.

As every theory needs some syntax form to express its elements, a road to a theory about theories leads through a syntax defining land, so structured type system, in the first place, provides a flexible generalized text parser that builds up internal abstract syntax trees (AST) from input data. The other aspect of theory about theories inevitably covers the meaning of input data. This is called semantics, and this is the point where structured type system provides a possibility to define deeper connections between syntactic elements of AST-s. For this purpose, structured type system uses a kind of functions known from functional programming paradigm. These functions are able to process any data corpus, being natural or artificial language translation, which in turn happens to be just enough for running any complexity task used to analyze existing and calculate new data from an input.

Using syntactic analysis and semantic transformations, we can build up whole theories, as well as whole programming languages (in this paper, a theory notion is isomorphic to a programming language notion), from the same, already known seed point, assuming that the seed point is expressive enough to hold our theories (programming languages) defined in structured type system.

The important aspect of structured type system is that for using it, we have to have an external seed point that can stand for any existing theory (or a computer language like Assembler, Javascript, HTML, SVG…). Once we have a usable seed point defined outside of structured type system, theories (thus, programming languages) can build up a translation pyramid, from more abstract higher level expressions at the bottom, over more basic expressions in the middle, to the very seed point expressions at the top of the pyramid. Finally, externally we deal only with the top of the pyramid, while structured type system plays a role of translation manager that doesn't really have to know what the expressions from the top of pyramid really mean. Structured type system only knows how to gradually translate our higher level theory expressions to the seed form expressions, and that is just enough for us. You can think of structured type system as of a universal compiler that translates higher level programming languages, over intermediate languages, to an assembler. Finally, the computer processor already knows how to deal with the assembler and it doesn't care for the actual translation path, opposite to the compiler that knows how to translate the language expressions to the assembler but doesn't need to know how to actually interpret the assembler.

2. Structured Type System Language

2.1. about structured type system

Structured type system is inspired by Backus-Naur form (BNF) language, and extends it by structure and function enhancements. It represents a rich and expressive type system, aiming to be able to express any form of expressions, accompanied by support for additional operations that can be made over these expressions. Structured type system grammar distinguishes syntactic part from semantic part of expressions it can describe.

Outlines of the syntactic part are sequence (analogous to product known from type theory) and alternation (analogous to sum known from type theory) types, wrapped inside implication (analogous to… well, this is something new, I think) types that introduce a structure to types. Syntactic part can be used to form: (1) parsed expressions which are read as streams of characters, or (2) typed expressions which are read as sequences of expressions.

Outlines of semantic part introduce function formation, where functions provide the meaning to syntactic elements. The question that arises is: *"What is really a meaning of an expression?"* In structured type system, the answer is: *"A meaning of an expression is its representation in terms of another expression for which the meaning is already known."* In structured type system, meanings of expressions are represented by chained functions from one expression to another. These meanings could be used for translating expressions through different languages, or for calculating outcomes of specific operations. It turns out that any operation, such is a math addition (i.e. 2 + 2), is merely a translation from operation parameters to operation result (i.e. 2 + 2 is being translated to 4). Readers familiar with functional programming paradigm are aware of algorithmic completeness of using functions to perform arbitrary complex calculations over arbitrary data.

Structured type system also provides a support for dependent types known from type theory. Dependent types are implemented in a special way, by pairing an expression that has to be recognized by the type system, to an expression that is expected to be found at another place. The relation is that the later expression depends on the former one.

We will see in the rest of this chapter how to utilize structured type system to form compact and expressive parts and wholes of programming and theory systems. The reader is required to be familiar with notions of parsing texts and functional programming, as the rest of this chapter gets extremely technical regarding these notions.

2.2. the grammar language

The **grammar language of structured type system** consists of atomic, compound, or dependent terms, together with optional comments:

- **Term** is any of the following:
  - **Atomic term** is any of the following:
    - **Constant:** expression enclosed between two single quotes, double quotes or back quotes
    - **Regular expression:** expression enclosed between two forward slashes
    - **New symbol:** expression that starts with alpha or _ character, followed by alphanumeric or _ characters.
    - **Reference:** a group, or expression prefixed with @, followed by a dot delimited path to a new symbol
  - **Compound term** is any of the following:
    - **Alternation:** expression in a form of *"A | B | ..."*, where *A, B, ...* are terms
    - **Sequence:** expression in a form of *"A, B, …"*, where *A, B, …* are terms

    - **Implication:** expression in a form of *"A <- B"*, where *A* is a new symbol and *B* is a term
    - **Function:** expression in a form of *"A => B"*, where *A* and *B* are terms
    - **Group:** expression in a form of *"(A)"* where *A* is a term
    - **Assigning a value:** expression in a form of: *"A (B)"*, where *A* is a term, and *B* is a specification of *A*. Braces around *B* are mandatory syntax element here, as well as delimiting whitespace

  When not explicitly emphasized by braces, compound terms are grouped by their precedence order. Precedence of grouping is shown in ascending order in the list above. Alternation terms have the lowest while assigning a value the highest precedence order. To emphasize grouping, any term can be surrounded by braces, even if surrounding is not needed at the place of its appearance
- **Dependent term** is a term that is paired with another term to the left of its appearance in a sequence or a function, and is written in one of two forms:
    - **Dependent pair:** *"(... | A -> [i] | …), …, (... | [i] -> B | …)"*, where *A* and *B* are atomic or compound terms and *i* is unique integer
    - **Dependent function:** *"(... | A -> [i] | …) => … => (... | [i] -> B | …)"*, where *A* and *B* are atomic or compound terms and *i* is unique integer

  Dependent term operator *"->"* has higher precedence order than any operator of compound terms
- **Comment** is an expression that begins with *"//"* and ends with an end of line, or expression that begins with *"/*"* and ends with *"*/"*. Comments can be injected anywhere within a grammar body

2.3. putting pieces together

It is time to get familiar with building terms and assigning values in structured type system. In this section, we will talk about syntax, semantics and dependent terms. When we say "syntax", "semantics", and "dependent terms", we are expressing it in a way of analyzing other languages, not structured type system, whose synthesis is the matter of the whole paper. Learning how to deal with these three important segments should be enough to move on to more complex examples like building theories, and general or domain-specific programming languages. Point your ears and let's begin. It's going to be ruff in this section.

2.3.1. syntax formations

We are preparing to learn how to syntactically construct different forms expressions. The meaning of the word "syntax" is just about blind expression formation, not about actual expression meaning. We will introduce rules by which we can recognize when expressions are not well formed, and by which we can internally structurize well formed expressions, preparing them for further analysis or computation.

Probably the simplest example of a valid structured type is a constant. Constants are parsed exactly as noted between quotes, only without the quotes. Example:

```
"John"
```

This grammar example parses assigned value (we will get there a bit later) only if it says exactly `John`, otherwise, it reports an error. Equivalent to Javascript strings, there are three types of quotes in structured type system: single quote, double quote, and backquote. Single and double quotes are used when parsing a single line constant, while backquotes are used when a constant spans through multiple lines. To use a relevant quote as a part of a constant value, we have to escape it with a backslash.

Regular expressions are embraced with "/" sign on the both sides. They behave exactly like Javascript regular expressions and are in fact string patterns used to match a certain variable type of an assigned value. For example:

```
/[0-9]+/
```

succeeds in parsing any size integer number. If you are not familiar with Javascript regular expressions, try searching the Internet for a detailed explanation of their syntax, while we move further to alternations. **Alternations** denote optional parsing possibilities of assigning values. For example:

```
"car" | "ship " | "plane"
```

succeeds whether an assigned value says car, ship, or plane. **Sequences** stand for what their name says: sequences of inputs. Let's use the previous example in a sequence:

```
"Vehicle", ":", ("car" | "ship " | "plane")
```

This grammar succeeds on assigned values: Vehicle:car, Vehicle:ship, or Vehicle:plane. Note that we had to use grouping braces, as sequences have higher precedence order than alternations. **Implications** are used when we want to provide some structure for assigned value. For example, we could use the following grammar:

```
Titled <- (
    Title <- ("Mr." | "Mrs." | "Miss"),
    WhiteSpace <- /\s+/,
    Surname <- /[A-Z][a-z]*/
)
```

To parse an assigned value Mrs. Robinson and to structure it internally in the following abstract syntax tree (AST):

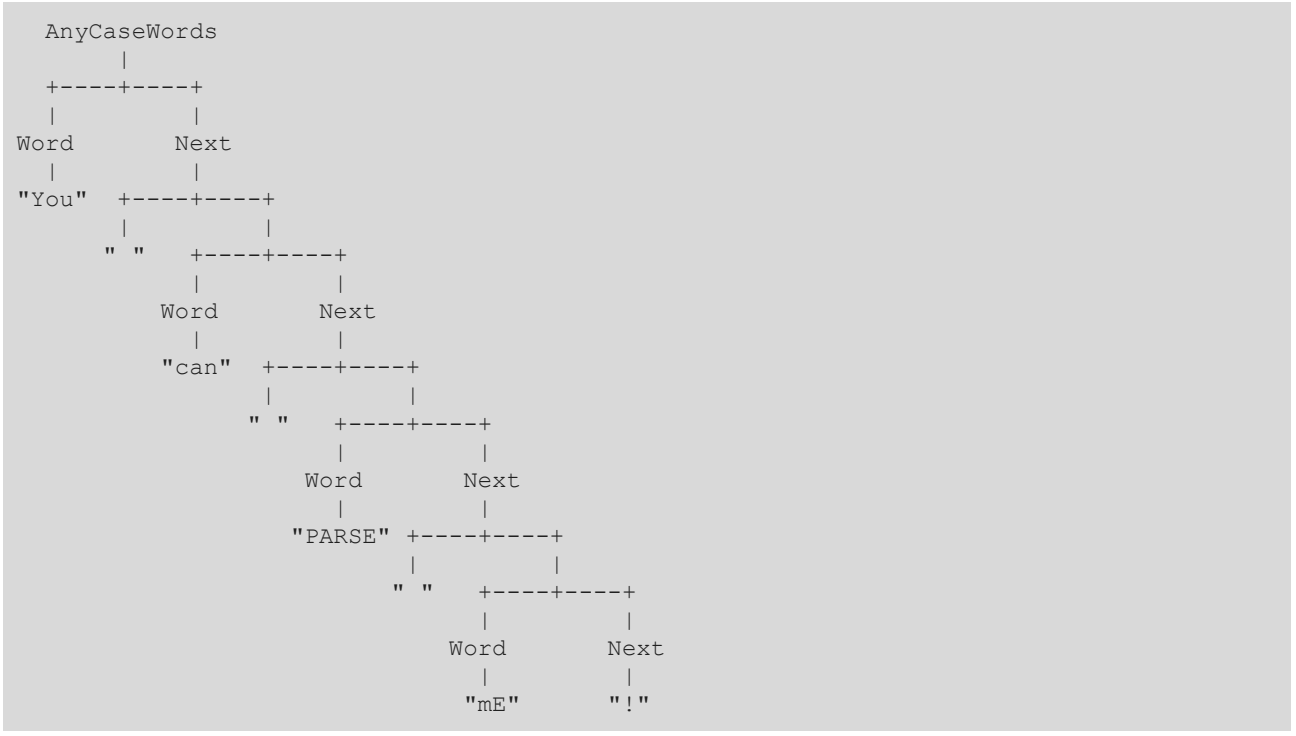
```
          Titled
             |
  +----------+------------+
  |          |            |
Title     Whitespace   Surname
  |          |            |
"Mrs."      " "       "Robinson"
```

As we can see, the left side symbol of an implication becomes a parent of the right side, when generating AST from assigned value. This parent-child relation from an AST follows parent-child relation graphically observed from the corresponding term. Using implications naturally leads to structuring terms which, combined with arbitrarily used white spaces and new lines for inlining expressions, brings a certain level of readability of terms written in structured type system.

Often we want to reach the structure defined elsewhere in a term, and then we use **references** written prefixed by @, followed by a dot delimited path to a wanted symbol. Term example:

```
AnyCaseWords <- (
    Word <- /[A-z]+/,
    Next <- (
        (/\s+/, @AnyCaseWords) |
        "!"
    )
)
```

in combination with assigned value: You can PARSE mE! creates the following AST:



In this case, we recursively reach AnyCaseWords symbol until "!" character is found at the end of the assigned value. **Extracting symbols from braces** can be useful to have a more concise arrangement of symbols in a term. It is done simply by appending a brace group by a dot delimited path to a symbol we want to extract. Extracted symbol becomes the topmost symbol in the place of extracting. In an example of date expression:

```
/*
This is an example of extracting symbols from braces. It is used
to form expressions with some special editing capabilities.
*/
(
    Number    <- /[0-9]+/ |
    Delimiter <- "/" |
    Date <- (
        Day         <- @Number,
        @Delimiter,
        Month       <- @Number,
        @Delimiter,
        Year        <- @Number
    )
).Date // this is where we extract a symbol contained in braces
```

if we want to change the delimiter, it is enough to touch the term at just one place, right at the definition of a delimiter. As Date symbol becomes the topmost symbol, Number and Delimiter do not interfere with parsing. In the same example, we can also see a use of multi line and single line comments. Comments are ignored by structured type system and their content is used only to textually describe parts of grammars to potential readers.

We have seen how to use basic constructs to define terms with different levels of expressiveness. We can say that compound terms are really just wrappers around terminal values that introduce some structure in terms and resulting AST-s.

Let us proceed by **assigning values** to symbols that is managed simply by stating a value inside braces right after the symbol, delimited by a whitespace. If we want assigned values to be parsed as a stream of characters, we have to previously assign the whole type to built-in symbol @Parsed.

```
@Parsed (
    (
        Integer <- /[0-9]+/ |
        Decimal <- (@Integer, ".", @Integer)
    ).Decimal
) // we assigned a type to the symbol @Parsed

...

x <- @Decimal (123.456) // assigning a parsed value of decimal number
```

Take a deep look at the previous example... A deep look... Oh, right there! Yes, there, you saw it well, we just mixed up a definition of a parsed type and one of its parsed values in the same source code! Please don't ask me about the low level implementation, but I checked it out: it works, you'll have to trust me. All right, all right, I'll tell you how to do it, but you don't tell a living soul about it: internally we process value assignments at the second pass, right after we parsed the whole source code, saving the assignments for later parsing. We repeat the whole process recursively in cases of multiple sub-level assignments. For that reason, braces inside assignment values always have to be balanced, sorry for that, but we have to know in advance where a parsed expression ends. If we have to use unbalanced open or closing brace, we have to escape it with a backslash. And let me tell you another secret: I was just joking about keeping a secret about this solution. Finally, It is not such a big deal, anyone can understand it if I can. Ok then, we can mix term types and their parsed values in the same source code, so let's proceed further now.

The other way of assigning values is assigning as values in a sequence. To do this, we just leave out the @Parsed symbol, and the term is ready for receiving a sequence. This mode of an assignment is similar to parsed mode, but we have to delimit assigned sub-values by a comma and optional whitespace. The other difference is that we can also optionally specify which parent symbol a specified sub-value belongs to:

```
(
    Integer <- /[0-9]+/ |
    IntegerList <- (
        Item <- @Integer,
        Next <- (@IntegerList | @None)
    )
).IntegerList

...

L1 <- @IntegerList (1, 2, 3)

...

L2 <- @IntegerList (@Item (1), @Next (@Item (2), @Next (@Item (3), @None)))
```

Note that symbol L1 completely equals to L2 after internal processing. Built in @None symbol stands for an empty alternation, and can be used to terminate lists, just like its analogous built in symbol @Unit that stands for an empty sequence. But in the case of using @Unit, we don't have to explicitly write it down at the end of the assignment sequence, which looks like a better way of sequence termination. Although the previous example is not the right case, the specification of parent symbols in some cases increases readability of assigned value. To show this, we can easily imagine a record like a symbol Person and its assignment examples:

```
Person <- (
```

```
    Name    <- /[A-z]+/,
    Surname <- /[A-z]+/
)

...
P1 <- @Person (John, Doe)
...
P2 <- @Person (@Name (John), @Surname (Doe))
...
```

where P1 and P2 contain exactly the same data, but with P2 we are sure what is a name and what is a surname without looking at the definition of Person. Every symbol also can be assigned by **multiple values**, in which case we write values in a form of alternation. It is possible to write:

```
...
P3 <- @Person ((John | Joe), Doe)
```

in which case the name of P3 represents a set of values, the set with elements John and Joe. This is an important aspect of structured type system that, as we will see later in more advanced examples, opens possibilities of processing sets of data, especially when utilizing data ambiguity to create sets.

Structured type system can also deal with assigned value **ambiguities** in a way analogous to handling multiple values. As we can see, in the following example:

```
Vehicle <- (
    Plane <- /[A-z]+/ |
    Ship  <- /[A-z]+/
)
```

it is clear that expected value for Plane and for Ship is of the same form, thus in a case where we would write:

```
V1 <- @Vehicle (Jenny)
```

the internal interpretation would be the same as we would write:

```
V1 <- @Vehicle (@Plane (Jenny) | @Ship (Jenny))
```

which means that the symbol V1 would in both cases contain exactly the same set of data, namely the set of two elements: @Plane (Jenny) and @Ship (Jenny).

With that, we will conclude this section. I hope I didn't scare you out with all of the complexities, but they are important and necessary for the functionality of structured type system, which could be used to do some exceptional things, as we will see later in more profound and complete examples. We learned how to form alternations and sequences, how to use implications to structurize symbols, how to extract specific symbols from braces, we learned how to assign values to symbols and how to understand what is going on when encountering ambiguities. In the next section, we will get down to the real business, and that is dealing with semantics. Things are just about to become interesting, I promise.

2.3.2. semantic formations

The notion "semantics of an expression" stands for a meaning of an expression. Remember when we said that a meaning of an expression is its representation in some other syntax form for which we already know how to interpret it? In structured type system, functions actually do this magic. Functions provide meanings

to syntactic elements, by connecting their parts to allow translation from one to another, more or less informative syntax form that we are defining along.

To define functions in structured type system, we have to know what parts functions consist of. In structured type system, every function consists of its typed input and its typed output. Input and output are typed so we don't ruin the integrity of expressions from which we use functions. When we want to use a function, we make a call to it. We provide some input, the function crunches and turns over some data, and then puts the final output to the place where we made the call to the function. The type of the output restricts function use only to those places where exactly the same type or its more specified type is expected. Lastly, if we want to reach a function from inner or outer places of its definition (and we usually want), a function in structured typed system has to be a part of an implication (the same implication from the section 2.3.1.) to some symbol which will be used as an anchor to call the function.

We form functions by operator `=>`, which defines an input on its left and an output on its right side. In examples from this section we will assume the existence of types `@Int` and `@Bool` that stand for integers and booleans respectively. Simple integer arithmetic and boolean comparison algebra are also assumed to be a part of the language base. Let's begin with a simple function that arithmetically adds two integers:

```
Add <- (                              // this is an anchor of a function
    (a <- @Int, b <- @Int) => @Int (  // this is a type of the function
        @a + @b                       // this is its assigned return value
    )
)

...
x  <- @Add (2, 4)           // x now holds an integer 6
...
y1 <- @Add (@x, 8)          // y1 now holds an integer 14
...
y2 <- @Add (@Add (2, 4), 8) // y2 now holds an integer 14
...
```

In this example, we nested the function body under the anchor `Add` which reflects the name of the function. Function parameters (input) take a sequence of two integers and returns an integer as the function result (output). The returning type is enriched by a value assignment, such that, in this case, calculates a sum of parameters. The function in later use is reached by its anchor, to which we apply parameters by value assignment, while the function behaves in a way expected from an expression that can be used in places where the type of the result allows. Functions written this way (with an anchor) can be passed around by a reference too:

```
...
a <- @Add      // a becomes a reference to function @Add
...
x <- @a (2, 4) // @a behaves exactly like @Add; x now holds an integer 6
```

One of the very useful functions may be a function `Case` that takes a choice of condition-result sequence pairs and returns an alternation of those results for which the condition is true. Although the definition is fairly simple, it does the expected job of filtering results:

```
Case <- (
    (Condition <- @Bool, Result <- @Any) => (
        @True => @Result |
        @False => @None
    ) (@Condition)
)

...
```

```
truths <- @Case (
    (1 < 2,  "one is less than two" ) |
    (1 == 1, "one is equal to one"  ) |
    (1 < 0,  "one is less than zero")
) // symbol truths now holds: "one is less than two" | "one is equal to one"
```

We already described @None and @Unit built-in symbols and now we are using the third built-in symbol from a general type triad: @Any, which stands for any expression conceivable in structured type system. @Any symbol is generally used where any type of expression is expected to be filled in later value assignment. As an intermediate result of Case function, we use a nested alternation of functions @True => @Result and @False => @None to be matched against assigned value of @Condition after its calculation. If @Condition matches @True, @Result is returned, otherwise, an empty alternation @None is used to ignore the result. Symbol truths shows a use and the intended behavior of @Case function.

Readers not familiar with functional programming paradigm (if any of them are still reading) may find it hard to imagine all the job that can be done by functions, but here is an example of emulating a loop by a recursion. We form a factorial function, known from math:

```
// factorial function definition
Fact <- (
    (Param <- @Int) => @Int (
        @Case (
            (@Param == 0, 1                          ) |
            (@Param > 0,  @Param * @Fact (@Param - 1))   // recursive call @Fact (...)
        )
    )
)

...

x <- @Fact (4) // x now holds a value 24
```

Where @Case function is showing its usability. It picks an arithmetic product of @Param with a recursive call to @Fact, putting it on the internal stack and moving to a level below. @Param gets decremented and the process repeats, until @Param reaches zero, when the recursion is terminated by returning 1. On the way back, while returning gradually from all the recursive calls, the function calculates the remembered product from each stack item, while eliminating calculated items. By the time the stack gets empty, our factorial expression is calculated. Huh, this was some nasty sentence construction, but it holds, as far as I can see, and I hope you understood it.

As you might guess, especially if you are familiar with functional programming paradigm, function chaining is also possible and that is called currying. Currying can be used for partial parameter application. As an example, let's rewrite our Add function to the curried form:

```
Add <- (
    (a <- @Int) => (b <- @Int) => @Int (@a + @b)
)

...

x  <- @Add (2)      // x now holds a function (b <- @Int) => @Int (2 + @b)
...
y1 <- @x (4)        // y1 now holds an integer 6
...
y2 <- @x (8)        // y2 now holds an integer 10
...
z  <- @Add (2) (10) // z now holds an integer 12
```

Function operator by default associates to the right, but if we want it to associate to the left, we can emphasize it by braces. With currying, we can apply parameters one slice at the time. The slice gets remembered, the slice awaiting iterator increments, so the function is waiting for the next slice in the chain. When the function reaches the chain end, iterator holds at the last result. As we can see with symbol z, we can also assign a chain of values at once. There is also some little something with assigning all of the chain of values at once: if we chain exactly the same number of assignments, which matches the number of curried function elements, the slice awaiting iterator doesn't change. This is an intended property of structured type system to open a possibility of defining function results in a form of semantic tables. To illustrate it, let's define a complete semantic table of basic Boolean constants and operators, all being members of a new symbol Boolean:

```
Boolean <- (
    True |
    False |
    And <- (
        (
            (@Boolean, @Boolean) => @Boolean
        ) (
            (@True,    @True   )    (@True)  |
            (@True,    @False  )    (@False) |
            (@False,   @True   )    (@False) |
            (@False,   @False  )    (@False)
        )
    ) |
    Or <- (
        (
            (@Boolean, @Boolean) => @Boolean
        ) (
            (@True,    @True   )    (@True) |
            (@True,    @False  )    (@True) |
            (@False,   @True   )    (@True) |
            (@False,   @False  )    (@False)
        )
    ) |
    Not <- (
        (
            @Boolean => @Boolean
        ) (
            (@True)     (@False) |
            (@False)    (@True)
        )
    )
)

...

b  <- @Boolean (@Or (@False, @Not (@And (@True, @False)))) // b now holds @True
```

We will remember this definition of Boolean and use it a bit later, while we move to the next example. Now we want to build a parsed version of boolean algebra. First, we will define a syntax that just accepts correct assignment value and rejects erroneous ones. We will call it BoolLang:

```
// bare bone BoolLang syntax definition

@Parsed (
    BoolLang <- (
        BinaryOp <- (
            And <- (@BinaryOp, "and", @UnaryOp) |
            Or  <- (@BinaryOp, "or",  @UnaryOp) |
            UnaryOp <- (
                Not <- ("not", @Primary) |
```

```
                Primary <- (
                    /\s*/, //regular expression for whitespace
                    (
                        Group <- ("(", @BinaryOp, ")") |
                        "true" |
                        "false"
                    ),
                    /\s*/  //regular expression for whitespace
                )
            )
        )
    )
)
```

Next, we will extend this syntax by functions that actually do Boolean calculations using Boolean base from example mentioned before. To show a useful property of structured type system, we will intentionally leave out the definition of Not operator, and fill it in later, as an additional syntax extension to BoolLang. Example:

```
// rewriting BoolLang to introduce semantic calculations by Boolean function base
// and leaving out Not operator definition to be implemented as a syntax extension

@Parsed (
    BoolLang <- (
        BinaryOp <- (
            (
                And <- (Left <- @BinaryOp, "and", Right <- @Primary)
            ) => @Boolean.And (@And.Left, @And.Right) |
            (
                Or <- (Left <- @BinaryOp, "or", Right <- @Primary)
            ) => @Boolean.Or (@Or.Left, @Or.Right ) |
            (
                Primary <- (
                    /\s*/,
                    Value <- (
                        Group <- ("(", In <- @BinaryOp, ")") => @In |
                        "true" => @Boolean.True |
                        "false" => @Boolean.False
                    ),
                    /\s*/
                )
            ) => @Primary.Value
        )
    ).BinaryOp // this extraction is essential for the future extensibility
               // of BoolLang. It is used to correctly line up this reference
)

...

b1 <- @BoolLang (true and (false or true)) // b1 now holds @Boolean.True

...
```

This example shows an implementation of a little boolean calculator (but without "not" operator!) that accepts customized Boolean algebra syntax and returns a correct result. And here comes the interesting part I promised you earlier: further extending a syntax of languages defined in structured type system. We will additionally define "not" operator outside of BoolLang that will fit into our BoolLang language exactly where it belongs to like it was defined there from the beginning:

```
...

@Parsed (
    (
```

```
        Not <- ("not", Param <- @BoolLang)
    ) => @Boolean.Not (@Param)
)

...

b2 <- @BoolLang (true and (not true)) // b2 now holds @Boolean.False
```

Now, isn't this just what every decent artificial language needs: syntax extensibility? In the example, we just introduced a parsed "not" operator that, because its function type results by @Boolean.Not function (which in turn results by @Boolean.True or @Boolean.False), fits exactly everywhere where @Boolean.True or @Boolean.False is expected. And that means it fits into our @BoolLang language too because both @Boolean.True and @Boolean.False are stated as results under Primary symbol, right beside "true" and "false" constants. In short, you can call functions from any place where their result is expected. Whether functions are parsed or regular, their result is what matters at the end, just because a result equals semantics. This is a very important property of structured type system, so please make sure you understand it well before proceeding any further.

There is one more interesting thing we have to see in this section. Somewhere earlier we mentioned using structured type system ambiguity handling mechanism to form sets. This is exactly what we are going to do now. We will define a function Range that takes two integers and returns a set that holds every integer greater than or equal to the first and less than or equal to the second parameter:

```
Range <- (
    (x <- @Int, y <- @Int) => (
        (
            // begin of ambiguity handler
            SetToY <- (
                @Int |
                (p <- @Int) => @Case (@p < @y, @SetToY (@p + 1)) // possible recursion
            )
            // end of ambiguity handler
        ) (@x) // applying a single parameter to the ambiguity handler
    )
)

...

s10 <- @Range (0, 10) // s10 now contains a choice of integers from 0 to 10
```

We use a symbol SetToY that, once assigned by an integer, matches against both of its two alternations. The first alternation just takes the integer as it is, while the second alternation is a function whose only parameter is an integer. Once the parameter is applied to this function (making an ambiguity, along with the first alternation), it returns a recursive call to the very same ambiguity handler SetToY, extending itself by incremented value while the @Case call is satisfied by checking if the parameter is still below the top bound. The recursion terminates when the last element from recursion reaches the top bound. SetToY symbol is a very nice example of expressivity of structured type system, where we combine implication which nests an alternation of a single element and a function that happens to be recursive multiplicator by consuming input ambiguity. The rest of formalities about passing x and y parameters from Range to SetToY are arranged in the similar way we saw in examples before this one.

This concludes the section 2.3.2. I hope you liked at least some of behaviors of structured type system shown here. We learned how to define functions, how to combine it with implications as anchors, how parameters are applied to functions (even to functions in curried form) and we constructed our first parsed Boolean calculator language from which we learned how to extend any expression defined in structured type system. Seeing syntactic extensibility from this side of knowledge, It seems just like the most natural thing that exists, but some of the most important programming languages of today seem not to be aware of this

simplicity. I hope that this fact would be changed as soon as possible. Lastly, we saw a set formation style that uses ambiguous alternations in hope to inspire lucid solutions made in structured type system in the future.

As a formality that we can't avoid when dealing with types, we left dependent terms for the last section of this chapter. As you might already expect, we also prepared some new and somewhat unusual solutions to this specific area, so let's give them a chance. These solutions could be useful at the end of the day.

2.3.3. dependent terms

Yes, terms may be dependent, but not on drugs like some kind of drug addicts, yet in a more subtle correlation: terms may depend on each other. Actually, this implementation of dependent terms came to me as a result of an effort to represent parse forests in a human readable way. Parse forest is something like abstract syntax tree, but it involves ambiguities, resulting in a more complex structure than plain AST. I had a half of solution solved by introducing alternations right into AST-s, to deal with ambiguities, but this half covered only branching out to ambiguous terms to the right. What I also needed was gathering ambiguous terms back to single branch to the right, so we don't have to repeat the same terms over and over beside each branch, to the end of the parsed text. Let me show the essence of this problem in an example:

```
Crowd <- (
    Person <- /[A-z]+/,
    Gender  <- ("Male" | "Female")
) (
    (John,   Male  ) |
    (Jane,   Female) |
    (Jenice, Female) |
    (Joe,    Male  ) |
    (Jill,   Female)
)
```

What we see here is repeating a gender beside each name. This doesn't seem so bad when we deal with a short term such is gender, but imagine if we, instead of gender had an endless e-sausage term we have to repeatedly write over and over again, wasting our time and space needed for the whole term representation, while listening to a worm of doubt that whispers slowly again and again: that's just not it… Well, let's make it right this time, if not for another reason, then just because we don't like whispering worms (no offense to other kinds of worms). We will introduce a pair of dependent term operations `… -> [i]` and `[i] -> …`, which will serve as a continuation anchoring mechanism. Let's see how the example looks now:

```
Crowd <- (
    Person <- /[A-z]+/,
    Gender  <- ("Female" | "Male")
) (
    (
        (
            John   -> [2] |
            Jane   -> [1] |
            Jenice -> [1] |
            Joe    -> [2] |
            Jill   -> [1]
        ),
        (
            [1] -> Female |
            [2] -> Male
        )
    )
)
```

Voilé! Now we can use any size e-sausages without having to repeat their content. Dependent terms, being a part of a sequence, form a construction named **dependent pair**. We can use them anywhere in terms of

structured type system (assuming that they are parts of a sequence in this case), affecting even types of expressions, not only values. For example, we can form a term Vehicle in the following way:

```
Vehicle <- (
    Type <- (
        Plane -> [1] |
        Ship  -> [2]
    ),
    Name <- /[A-z]+/,
    (
        [1] -> (MaximumAltitude <- @Number) |
        [2] -> (HydroplaningAbility <- @Boolean)
    )
)
```

Now Vehicle's third sequence element depends on the first sequence element, which should be considered when assigning future values. However, analyzing further, dependent terms could be a part of the left and the right side of some function, where a function domain affects a function codomain. Example:

```
VehicleProperty <- (
    (
        VehicleType <- (
            Plane -> [1] |
            Ship  -> [2]
        )
    ) => (
        [1] -> (MaximumAltitude <- @Number) |
        [2] -> (HidroplaningAbility <- @Boolean)
    )
)
```

In this case, a codomain of VehicleProperty function depends on its domain VehicleType. Dependent terms that affect functions are forming a construction known as **dependent function**. Dependent terms in a dependent function are not obligated to be directly on the sides of the same function, so they can span through multiple chains of functions.

At the end, by having some insights in theory of dependent types background, from deforestation of parse forest solution, we got an extension to structured type system, which enhances its typing expressivity. We might find it useful in some definitions, so we will include dependent terms in the definition of structured type system.

With this section, we conclude the exposure of structured type system. This should be enough to build up some more serious examples of general or domain specific languages with the most important property of being able to be extended by custom syntax extensions afterward.

3. Conclusions and further work

Structured type system, in spite of what I expected in the beginning of my research, is closer to a programming language than to some scientific theory about theories. I guess that being formal enough to be able to process data by a computer has its price: a complexity of expressions.

In a hope that this solution is complete, the further work is expected to take a place at specific implementation of deduction, induction and natural language processing by structured type system. This should open doors of making an online solution that could be used for holding a general knowledge base in examples of school programs for students. Such a knowledge base should have a property of solving math, physics, chemistry, and other natural sciences tasks, as well as being able to construct more or less complex

answers to questions posed about social sciences, all of which may be enabled only if crowdsourced support would take a swing.