`[PROVE]`

17th December 2020

1 Introduction

This document is work in progress. The purpose of this document to introduce notation and vocabulary in order to facilitate efficient communication about this project. Underlying concepts are explained verbally and/or formally in order to detect any possible misconceptions of the matter. Furthermore this document could serve as an introduction to the project, if somebody wants to join. Especially with respect to this last point it might be useful to include a short discussion of the following through points at the beginning of this document (this is a TODO list):

- a short verbal explanation of the purpose and form of the `[PROVE]` language
- a list of terms we have agreed upon to describe certain entities, e.g. statement, formula, formulator, variable, constant, identifier, ... (we might have to have a look at our agreements again)
- the meaning of an implication and an equality
- to be continued ...

At the moment this document starts off right, where we ended our last discussion, so no great explanations of things that we have agreed upon are present at the moment.

2 Terminology

- statement
- formula
- formulator
- identifier
- variable
- constant
- implication
- equality

3 Syntax

3.1 EBNF

3.1.1 A first sketch

The syntax of the [PROVE] language can conveniently be described by means of the extended Backus-Naur form[1] (EBNF), which is a meta syntax notation defined in the ISO/IEC 14977 [2] standard. Using an EBNF enables us to recursively and unambiguously describe nested patterns by grouping syntactic elements to so called $\langle\texttt{production}\rangle$s. It must always be possible to 'unpack' a production in a unique and finite way, resulting in a (sequence of) *terminal* symbols, indicated by quotation marks. In order to correctly interpret the [PROVE]-EBNF it is sufficient to understand the meaning of the following meta characters:

$\vert$	indicates 'or'
$(\dots)$	groups productions and/or terminals together; only useful in combination with $\vert$
$[\dots]$	indicates optionality
$\{\dots\}$	indicates zero or more repetitions

Using this notation the [PROVE] language can be described by the following EBNF:

$$\begin{aligned}
\langle\texttt{formula}\rangle &= \left\{\langle\texttt{statement}\rangle\langle\texttt{formulator}\rangle\middle|\langle\texttt{formulator}\rangle\langle\texttt{statement}\rangle\right\}\\
\langle\texttt{statement}\rangle &= \texttt{"["}\left(\langle\texttt{formula}\rangle\middle|\langle\texttt{string}\rangle\middle|\langle\texttt{statement}\rangle\right)\texttt{"]"}\left\{\texttt{"["}\ \left(\langle\texttt{formula}\rangle\middle|\langle\texttt{string}\rangle\middle|\langle\texttt{statement}\rangle\right)\ \texttt{"]"}\ \right\}\\
\langle\texttt{formulator}\rangle &= \texttt{"="}\middle|\texttt{"=>"}\middle|\langle\texttt{string}\rangle\\
\langle\texttt{string}\rangle &= \langle\texttt{symbol}\rangle\left\{\langle\texttt{symbol}\rangle\right\}\\
\langle\texttt{symbol}\rangle &= \left(\text{"a"}\middle|\text{"b"}\middle|\dots\middle|\text{"z"}\middle|\text{"A"}\middle|\text{"B"}\middle|\dots\middle|\text{"Z"}\middle|\text{"0"}\middle|\text{"1"}\middle|\dots\middle|\text{"9"}\middle|\text{"+"}\middle|\text{"-"}\middle|\text{"/"}\middle|\text{"*"}\middle|\text{"\%"}\middle|\text{"\^{}"}\middle|\text{"\&"}\middle|\text{"."}\middle|\text{"?"}\middle|\text{"!"}\right)
\end{aligned}$$

Every EBNF needs to have an initial production as a starting point. It is convention that this distinguished production is to be listed first, so in this case, $\langle\texttt{formula}\rangle$ is the initial production. One might notice that a $\langle\texttt{formula}\rangle$ might be empty and a $\langle\texttt{statement}\rangle$ may contain such an empty $\langle\texttt{formula}\rangle$. We can reason about whether this makes sense or not, and it might be necessary to revise these definitions, but currently I just regard these situations as edge cases with no harmful implications.

3.1.2 Examples

Here are some examples of productions and some motivations for the design of the syntax. We will work our way up from bottom to top through the EBNF.

- A ⟨`symbol`⟩ can be any letter, number or one of the 10 special characters as indicated in the EBNF. Examples of valid ⟨`symbol`⟩s are `a`, `T`, `7`, `&`.
- A ⟨`string`⟩ is a sequence of one or more symbols. Examples of valid ⟨`string`⟩s are: `a7we`, `T`, `76`, `&q`. Note that `T` is a ⟨`string`⟩ and a ⟨`symbol`⟩ at the same time. This does not mean that the grammar is ambiguous. A single `T` will always be interpreted in this way, but it depends on the context, whether we are interested in the fact of `T` being a ⟨`string`⟩ or `T` being a ⟨`symbol`⟩.
- A ⟨`formulator`⟩ is either a ⟨`string`⟩ or any one of the terminals `=` or `=>`. Examples of valid ⟨`formulator`⟩s are `a7we`, `T`, `=`, `=>`. Note that `=T` is not a valid ⟨`formulator`⟩. Seeing that `T` can be a ⟨`formulator`⟩ (comprised of a ⟨`string`⟩) in one case and just a ⟨`string`⟩ (not packed inside a ⟨`formulator`⟩) in another case, one might worry about ambiguity. However, having a close look at the EBNF, we see that ⟨`formulator`⟩s only occur in between ⟨`statement`⟩s and ⟨`string`⟩s only within ⟨`statement`⟩s. The user of the language has great freedom in naming formulators and identifiers (TODO: define formulators and identifiers in this document, i.e. not the productions, but what they represent), so it is up to the user to choose them wisely. It could for example be confusing to name a formulator `7`, but `add7` makes intuitively sense.
- A ⟨`statement`⟩ is a concatenation of one or more pairs of square brackets each containing either a ⟨`formula`⟩, a ⟨`string`⟩ or a ⟨`statement`⟩. Examples of valid ⟨`statement`⟩s are `[]`, `[a][b][[a]in[b]]`, `[[]]`, `[[a]or[b]]`.
- A ⟨`formula`⟩ is comprised of at least one ⟨`statement`⟩ and at least one ⟨`formulator`⟩. The order of occurence does not matter. Examples of valid ⟨`formula`⟩e are `[a]in[b]`, `[a]=[b][c]`, `[[a]a7we[b]]=>[c]`.

3.2 Improving the sketch

As intuitive and beautiful (due to its simplicity) this first sketch might look, there are a few problems with it. Firstly a proof might be given in form of a statement or in form of a formula. Secondly, when parsing a ⟨`statement`⟩ we might have to look far ahead after encountering the leading `[` in order to determine, whether we are handling a ⟨`formula`⟩ or a ⟨`statement`⟩. One way to address this issue is, to start parsing a ⟨`formula`⟩, after reading a `[`, but then, when not finding a ⟨`formulator`⟩, not to report an error, but to proceed with the steps that would follow when parsing a statement. To encode this behaviour in the EBNF and to address the issue of having the proof as a whole in form of a statement or a formula, we can introduce a helper production ⟨`expr`⟩(for expression) containing either a ⟨`formula`⟩ or a ⟨`statement`⟩.

$$
\begin{aligned}
\langle\texttt{expr}\rangle &= \langle\texttt{statement}\rangle \Big| \langle\texttt{formula}\rangle \\
\langle\texttt{formula}\rangle &= \Big\{ \langle\texttt{statement}\rangle\langle\texttt{formulator}\rangle \Big| \langle\texttt{formulator}\rangle\langle\texttt{statement}\rangle \Big\} \\
\langle\texttt{statement}\rangle &= \text{"["} \Big(\langle\texttt{expr}\rangle \Big| \langle\texttt{string}\rangle \Big) \text{"]"} \Big\{ \text{"["} \ \Big(\langle\texttt{expr}\rangle \Big| \langle\texttt{string}\rangle \Big) \ \text{"]"} \ \Big\} \\
\langle\texttt{formulator}\rangle &= \text{"="} \Big| \text{"=>"} \Big| \langle\texttt{string}\rangle \\
\langle\texttt{string}\rangle &= \langle\texttt{symbol}\rangle \Big\{ \langle\texttt{symbol}\rangle \Big\} \\
\langle\texttt{symbol}\rangle &= \Big(\text{"a"} \Big| \text{"b"} \Big| \dots \Big| \text{"z"} \Big| \text{"A"} \Big| \text{"B"} \Big| \dots \Big| \text{"Z"} \Big| \text{"0"} \Big| \text{"1"} \Big| \dots \Big| \text{"9"} \Big| \text{"+"} \Big| \text{"-"} \Big| \text{"/"} \Big| \text{"*"} \Big| \text{"\%"} \Big| \text{"\^{}"} \Big| \text{"\&"} \Big| \text{"."} \Big| \text{"?"} \Big| \text{"!"} \Big)
\end{aligned}
$$

4 Scanning

Briefly discuss the tokenisation.

```c
typedef enum {
    TOK_EOF,      /* end-of-file */

    TOK_LBRACK,   /* left bracket */
    TOK_RBRACK,   /* right bracket */

    TOK_IMPLY,    /* implication */
    TOK_EQ,       /* equality */
    TOK_STR       /* string */
} TType;
```

5 Semantic

When generating a couple of syntactically correct snippets of `[PROVE]`code, one realises that not all code adhering to the rules of the EBNF makes sense - at least not without introducing more conventions. Consider the following code:

```
[a]and[b]=[c]=>[b][c]
```

The use of `=` and `=>` is ambiguous in this context, since `=` could refer to everything on its right hand side or just to `[c]`. Similarly `=>` could refer to everything to its left hand side or just to `[c]`. This problem could be addressed by introducing an order of precedence (e.g. non reserved formulators, `=`, `=>`). However, the implications of this approach should be carefully considered. Precedence can easily be encoded in the EBNF, without changing the set of valid code patterns. By doing so we could elegantly encode a part of the language's meaning in its syntactic structure. As tempting as this idea might be, I can see a negative side effect coming along with this approach. Introducing precedence between formulators will set brackets *implicitly.* Since the visibility and meaning of identifiers (variables/constants) depends on the depth of nesting of statements (i.e. on brackets), the approach of precedence might lead to counter-intuitive code. Therefore I recommend against the precedence approach.

Instead I suggest the following semantic rules, which will address this and other issues after the code is confirmed to be syntactically correct:

- `=`, `=>` and other formulators must not be mixed
- new identifiers must only be introduced before the first `=>` in a formula
- `=>` must not appear at the end of a ⟨`statement`⟩
- `=` may appear only once in a ⟨`formula`⟩ and must not appear at the beginning nor at the end of a ⟨`formula`⟩

6 Parsing

Briefly discuss the parser functions. Maybe.

7 Representation

7.1 Example: Russel's Paradox

A first example, we considered is the following proof of Russel's Paradox. The purpose of this [PROVE]code is to provide a valid proof as a starting point for investigating further, which rules a data type representation should follow and which properties it must have.

```
[A] [ [x] => [ [[x]in[A]] = [[[x]in[x]]=>[False]] ] ] =>
[ [A] => [[[A]in[A]] = [[[A]in[A]]=>[False]]] ] =>
[ [[A]in[A]] = [[[A]in[A]]=>[False]] ] =>
[ [[A]in[A]] =>
        [[[A]in[A]]=>[False]] =>
        [False]] =>
[ [[A]in[A]] =>
[False]] =>
[ [A]in[A] ] =>
[False]
```

In the following subsections, we will showcase a possible representation of the structure of this proof as a graph and the C implementation of this graph.

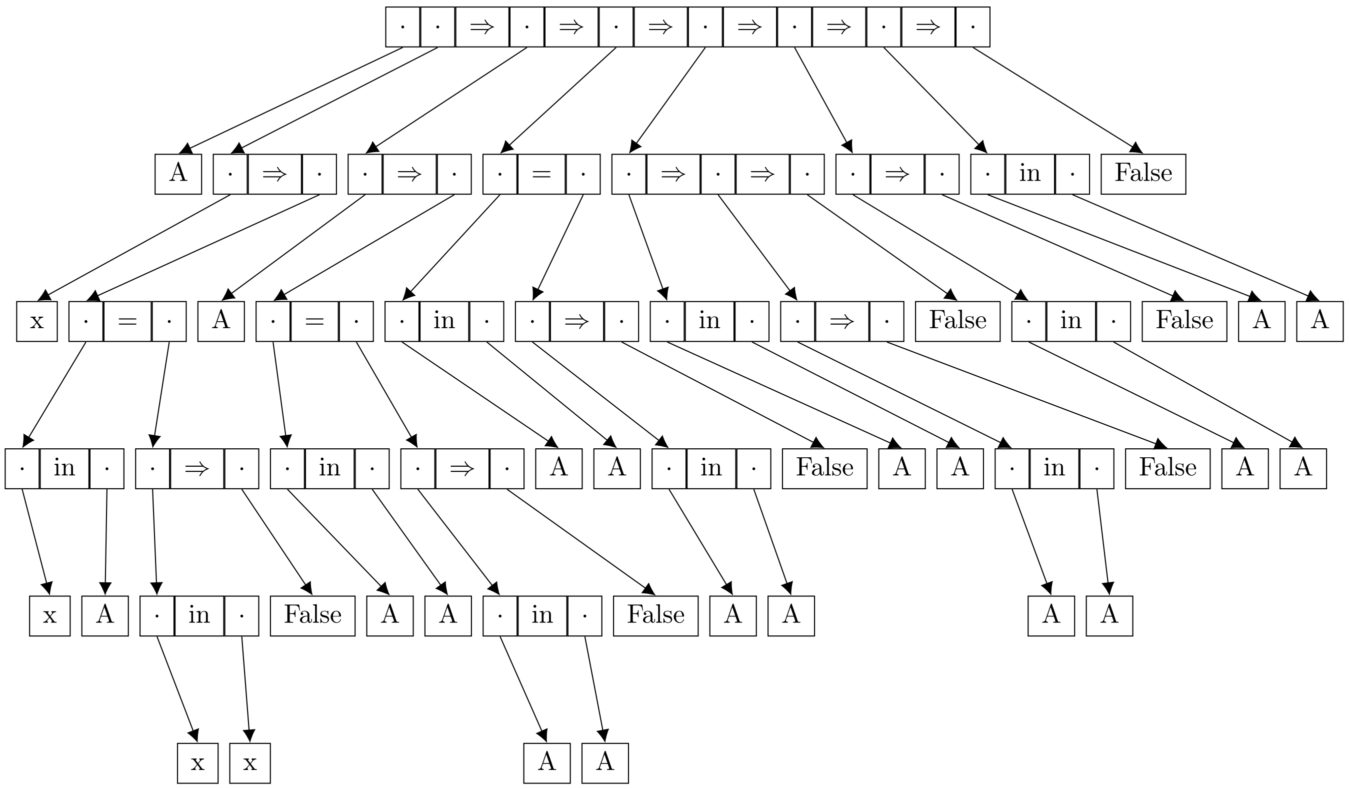
7.2 Graph representation

7.2.1 Nodes

Here comes a picture to illustrate the structure as well as the corresponding C code.

```c
typedef struct Pnode {
    struct pnode* parent;
    struct pnode* child;
    struct pnode* left;
    struct pnode* right;
    char* symbol;
    Flags flags;
    int varcount;
} Pnode;
```

7.2.2 Graph



7.3 Flags

Here comes some explanation of the purpose of each flag as well as the corresponding C code.

```c
typedef enum {
    FLAG_START = 0,
    FLAG_IMPL = 1,
    FLAG_EQTY = 2,
    FLAG_FMTR = 4,
    FLAG_ASMP = 8
} Flags;
```

7.3.1 Differences between implication and equality

Maybe a table would be helpful here.

7.4 The variable counter

Explain the purpose of the variable counter.

8 Creation of the graph

Discuss this code:

```
#define IS_CONST_NODE(pnode) (pnode->child->symbol != NULL)
#define IS_LEAF_NODE(pnode) (pnode->symbol != NULL)
#define IS_LEFTMOST_NODE(pnode) (pnode->left == NULL)
#define IS_START_NODE(pnode) (pnode->flags == FLAG_START)

#define SET_IMPL(pnode) (pnode->flags &= FLAG_IMPL)
#define SET_EQTY(pnode) (pnode->flags &= FLAG_EQTY)
#define SET_FMTR(pnode) (pnode->flags &= FLAG_FMTR)
#define SET_ASMP(pnode) (pnode->flags &= FLAG_ASMP)

#define UNSET_ASMP(pnode) (pnode->flags &= ~FLAG_ASMP)
```

9 Verification

- only `=>` triggers verification
- `=>` is an assumption (i.e. does not trigger verification), if statements to the left contain *new* constants but no variables

References

[1] https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_form

[2] https://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf

[3] https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_Form