Concept Grammar Language

by Sven Nilsen, 2016

Abstract:
In this paper I construct a Concept Grammar Language that creates a pseudo formal semantics for formal languages. The idea is to communicate thoughts regarding orthogonal meaning of inter related grammar. It can be used to explain how languages are extended with new concepts and why this results in greater combinatorial complexity.

A grammar is like showing the shape of a language. Syntactic grammars is a detailed tree that maps words to some data structure. Semantic grammars are more challenging to show, because you do not want to just refer to a rule, but explain how this makes a sentence valid.

My idea is to use big capital letters, such as `A`, `B`, `F` and `X`, to refer to some classes of symbols. The classes can be used in mathematical axioms. This gives meaning to sentences.

For example, to explain that adding two natural numbers leads to a greater or equal number:

```
A + A = B
A <= B
```

It means that both numbers of class `A` are less or equal than numbers of class `B`.

Then, if somebody see `A + A = B` and wonders whether it is `A + (A = B)` or `(A + A) = B`, you can show them this is the order from top to bottom:

1. A = A
2. A + A

Therefore, `A + A = B` should be read as `(A + A) = B`.

This tells them how to read a sentence, but not how to understand equality and addition.

To explain the detailed internals of language you need a more powerful language. This is out of scope for this paper. What I want to show is technical terms and ways of thinking for creating the fundament for new and powerful languages.

Type agnostic

The grammar concept language is type agnostic. Different classes can be of different types, but does not have to. The same class can consist of different types when referring to different parts of grammar.

Relationships between types and variables can be encoded into the language you describe. Therefore, one is referring to symbols in general, or symbols accepted by the language.

What sentences are accepted depends on the concepts constructed by the language. A concept language is meant to show how the concepts are constructed, by using simpler ones to describe more advanced concepts.

Strong vs weak

These are technical terms for defining externality of operators:

- “strong” means the operator is well defined within the meaning of the same language
- “weak” means the operator is encoded and interpreted in some external language

For example, using the concept of natural numbers:

$$A = A$$

Is a strong operator because it is well defined how equality between natural numbers behave. There is no need to referring to some external concept.

Assuming that the concept of natural numbers is unknown, then:

$$A = A$$

Is a weak operator, because understanding fully what it means requires an external concept.

A weak operator does not mean that the operator does not behave well. It must make sense somehow, but the justification for how it makes sense is unknown.

Class

A class has set properties, but is also ordered. It can have tree like properties by being defined over a more complex grammar. In general terms, it refers to a collection of symbols.

Indices are used when there are more than one symbols of the same class that refer to different parts of a grammar:

$$(A_0, A_1, A_2)$$

This can represent a vector of 3 dimensions.

Advanced index notation

This notation is for advanced transformations of sentences.

One can describe a pattern by using letters instead of number when indexing:

$(A_i,^i)$

The pattern above matches:

(a)
(a, b)
(a, b, c)
etc.

Each index that is lowered is matched against a lifted index. The raised index is placed such that one can understand how the pattern is repeated.

Pattern	**Description**	**Examples**
(P) | [P]	Parenthesis, square brackets	(0) [0]
$\{\{A_i\}^i\}$	Separated by new line	1 2
$[\{A_i\}^i]$	Group	[1 2]
$A_i,^i \mid A_i;^i \mid A_i+^i \mid ...$	Separate by sign	1,2,3
A_i^i	Separate by empty space	1 2 3
A_i^{-i}	Reverse order	3 2 1
$F_i(A)^i$	Nested function calls	a(b(c(x)))
$[(A_i\ A_j),^{ij}]$	Pair up indices	[(1 2), (3 4)]
$[A_i^{ij}, A_j, A_i^{-ij}]^j$	Entangling indices	[1 2 3, 4, 3 2 1]
$[A_i,]^i$	Repeat same pattern	[1, 1, 1]
$[A_{ij}^j,^i]$	Matrix	[1 2 3, 4 5 6, 7 8 9]

A matrix or tensor with many indices is easier to understand if you reduce them like this:

$[A_{ijk}^{k\ j},;^i]$
$[B_{ij},^j;^i]$
$[C_i;^i]$
[D]

Then you go in the other direction:

[a]
[a; a; a]
[a, a, a; a, a, a; a, a, a]
[a a a, a a a; a a a, a a a]

For example:

[1 2 3, 4 5 6; 7 8 9, 10 11 12]

Ambiguous cases

Consider the following pattern:

$$A_i{}^{i\,j}_{j}$$

It is not clear how to interpret:

1 2 3 4

As one of the following ways:

((1 2) (3 4))
((1) (2 3) (4))
((1) (2) (3 4))
etc.

Therefore, for separator signs they are only allowed once inside a parenthesis or separated by newlines.

Example: Order of a function does not matter

This can be expressed with two different classes:

$$\mathrm{and}(A, B) = \mathrm{and}(B, A)$$

Or using a single class with indices:

$$\{\mathrm{and}(A_i, A_j) = \mathrm{and}(A_j, A_i)\}^{ij}$$

Example: Paths and equations

This is an example from path semantics, where a path between two functions corresponds to an equation:

$$F_0([F_1]\ X) \rightarrow [F_2]\ F_3(X)$$

$$F_2(F_0(X)) = F_3(F_1(X))$$

This demonstrates the concept, but does not explain:

- how to deal with multiple arguments
- whether the sentences are interpreted separately or together

Here is a more accurate description:

$$\{\{F_n(\{[F_j]^{jn}\ X_i\},^{ip}) \rightarrow [F_k]^{kn}\ F_m(X_i,^{ip})^{mn}\}^{p}$$
$$\{F_k(F_n(X_i,^{ip}))^{-kn} = F_m(F_j(X_i,^{ip})^{-jn})^{mn}\}^{p}$$

$\}^n$

The entanglement of indices such as “jn” makes it possible to carry over symbols from one group to another.

This still does not deal with reversible paths, tuple paths etc. Path semantics is *complex*!

Combination of concepts

So far we have not used this notation for more than just another way of binding values to grammar. The real power comes from interpreting the same symbols in more than one way, and then superimpose those interpretations on top of each other.

As an example, let us examine how a normal programming language usually works. The most basic concept is the one of computation, where we replace some symbols with others:

$$F \rightarrow X$$

This operation is Turing complete. If this concept is lifted to the level where we reason about programs, then another concept can be introduced that represents computation as a relation between symbols:

$$F(X) \rightarrow X$$

or

$$F \rightarrow \{X \rightarrow X\}$$

Repeating this process gives us Lambda calculus. A function might take another function as a first class citizen and return another function:

$$F(A \rightarrow B, B \rightarrow C) \rightarrow \{A \rightarrow C\}$$

Most programming languages have these concepts:

$F \rightarrow X$	(computation, axiom, return value)
$F(X)$	(function arguments)
$F(X_0) \rightarrow X_1$	(function closure)

The full meaning of a sentence are all the concepts that it expresses. When you see the following in a programming language there are several concepts built into it:

$$foo(a) \rightarrow b$$

`foo` - function name
`a` – function argument
`b` - return value
`(a) → b`