LEIDEN UNIVERSITY MEDICAL CENTER

Parsing the HGVS nomenclature

Jeroen F. J. Laros
Department of Human Genetics
Center for Human and Clinical Genetics

HGVS nomenclature

HGVS stands for Human Genome Variation Society.

Nomenclature for the description of sequence variations.

Used as input for Mutalyzer, a program running on a web server that checks the correctness of a variation.

- To assist people in denoting a variation.
- To check already stored variations (in LOVD for example).
- To analyse the effect of a mutation.
- etc.

Variation checking

Must be done on two levels:

- Syntactic checking.
- Semantic checking.

Syntactic checking is to check whether the variation is correct (in principle):

- Is the reference sequence given?
- If a gene symbol is given, is it between parenthesis?
- If a transcript variant is given, is it preceded by "_v"?
- etc.

Variation checking

Semantic checking is to check whether the variation is also correct within the context.

- If a substitution “A>T” is given at a certain position, is there really an “A” in the reference sequence?
- Are two adjacent deletions given? (if so, we have to group them).
- A large delins is given, might it be an inversion?
- etc.

Semantic checking can only be done after the syntactic check.

The problem

The HGVS nomenclature is capable of describing many different variations:

One simple variation.
NG_008660.1(CAPN3):c.496delC

More simple variations on one allele.
NG_008660.1(CAPN3):c.[496delC;500G>T;510A>C]

A simple insertion.
NG_008660.1(CAPN3):c.496_497insAGGGACT

The problem

The variations can become complex:

An insertion of a slice of an other accession number.
NG_008660.1(CAPN3):c.496_497insNM_000059:n.4_104

The same insertion as above, but now with a variation within the insertion.
NG_008660.1(CAPN3):c.496_497insNM_000059:n.4_104{100G>A}

These expressions can become of arbitrary length (and complexity).

The problem

We want to extract the data from the input string in two steps:

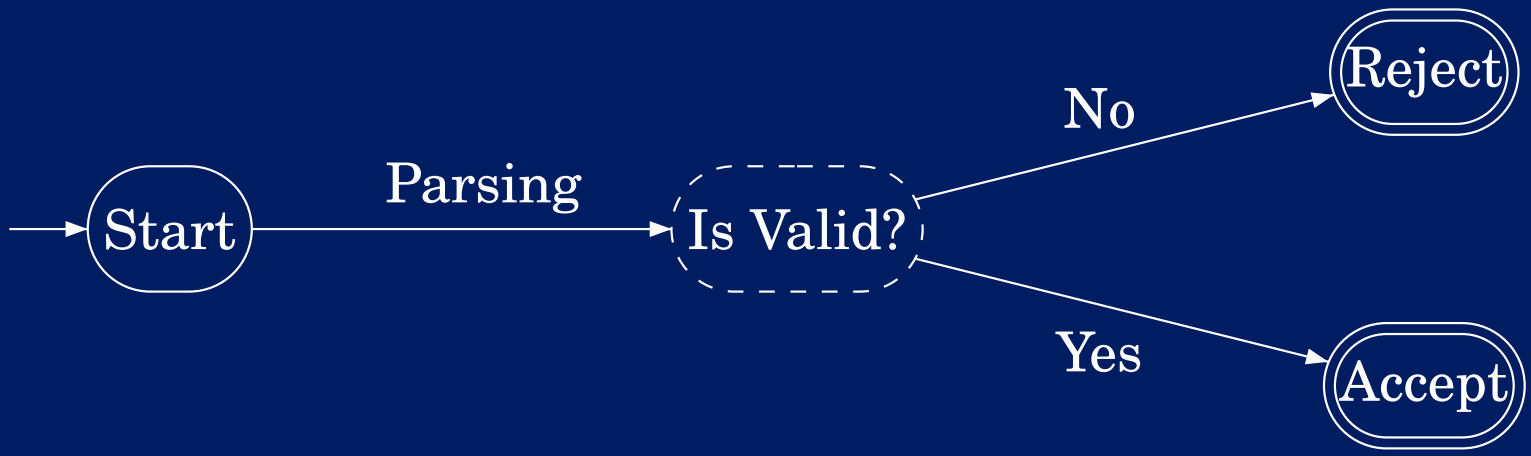
- Recognition of an input string.
  - Optionally we can give information upon rejection.
- Extraction of data from the input string into a format the computer can understand (a parse tree).
  - This parse tree is used in combination with the context for semantic checking.

For these two functions, we need a *parser*.

Two functions of a parser

Syntactic checking:

First of all, we want a yes/no answer to see if the input is valid.



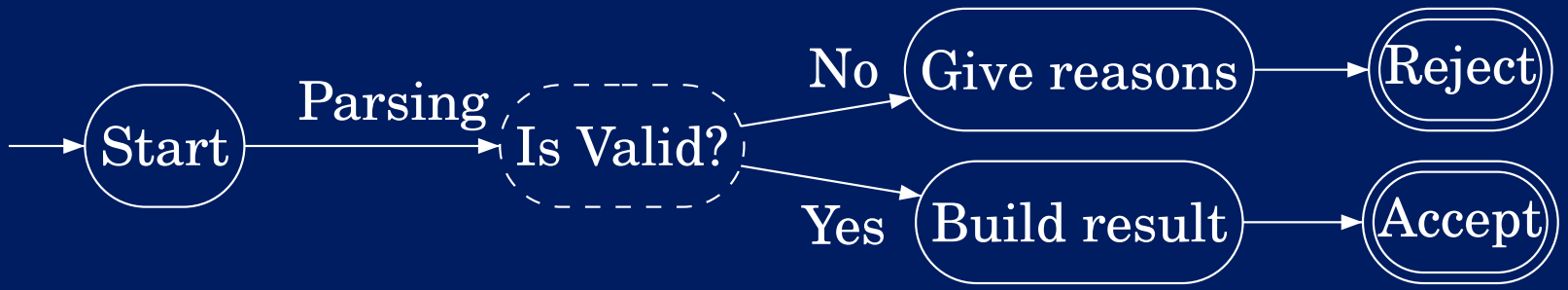
NG_008660.1(CAPN3):c.496delC → Yes.

NG_008660.1(CAPN3)c.496delC → No.

Two functions of a parser

NG_008660.1(CAPN3):c.496delC → Yes.

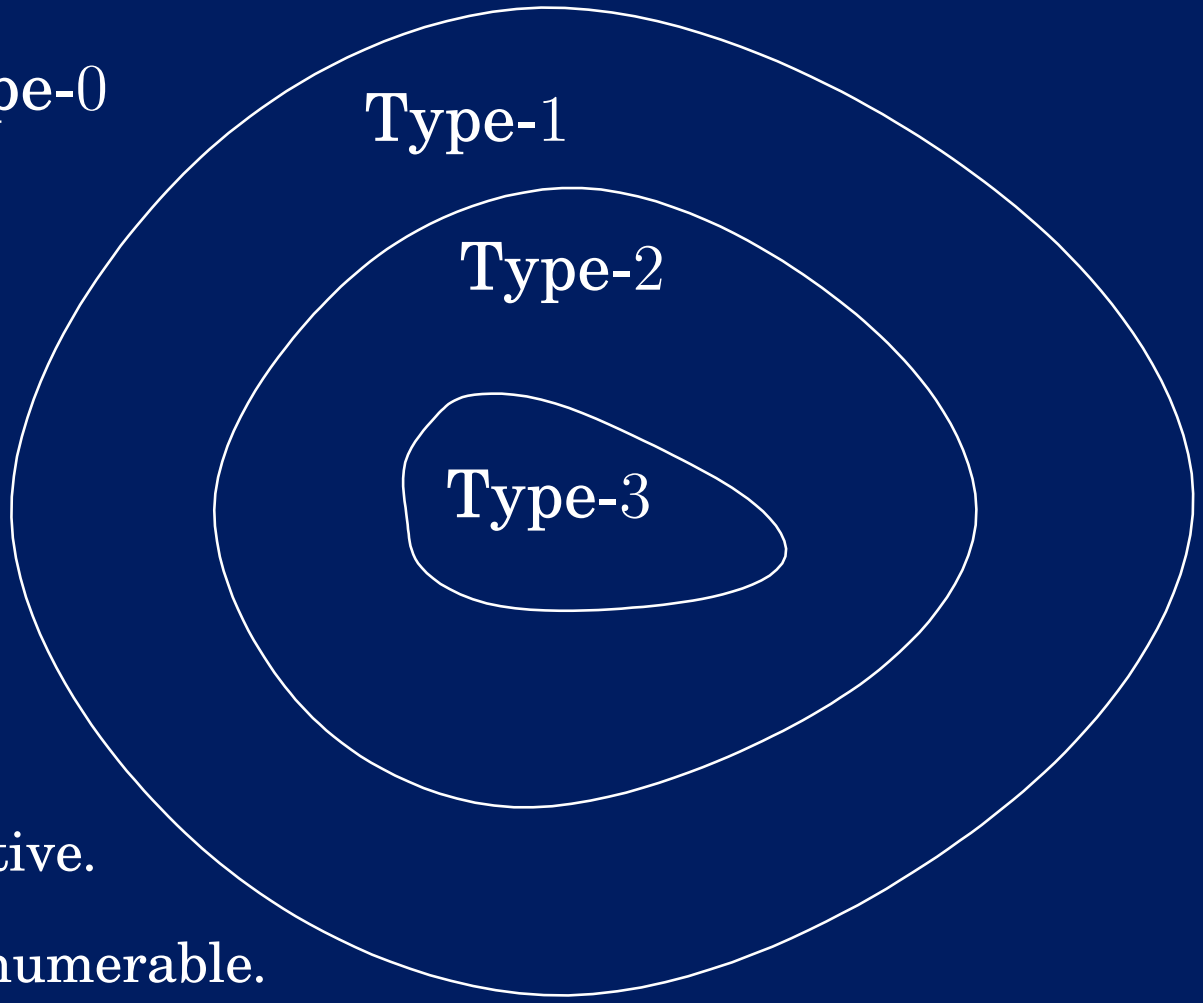
Accession Number: NG_008660 (version 1)
Gene: CAPN3
Reference type: coding
Mutation:
- Position: 496
- Type: deletion



NG_008660.1(CAPN3)c.496delC → No.

Reason: Missing ‘:’

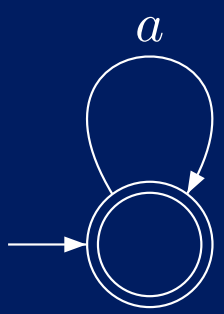
What kind of parser?



- Type-3: Regular.
- Type-2: Context-free.
- Type-1: Context-sensitive.
- Type-0: Recursively enumerable.

Type-3 parsers

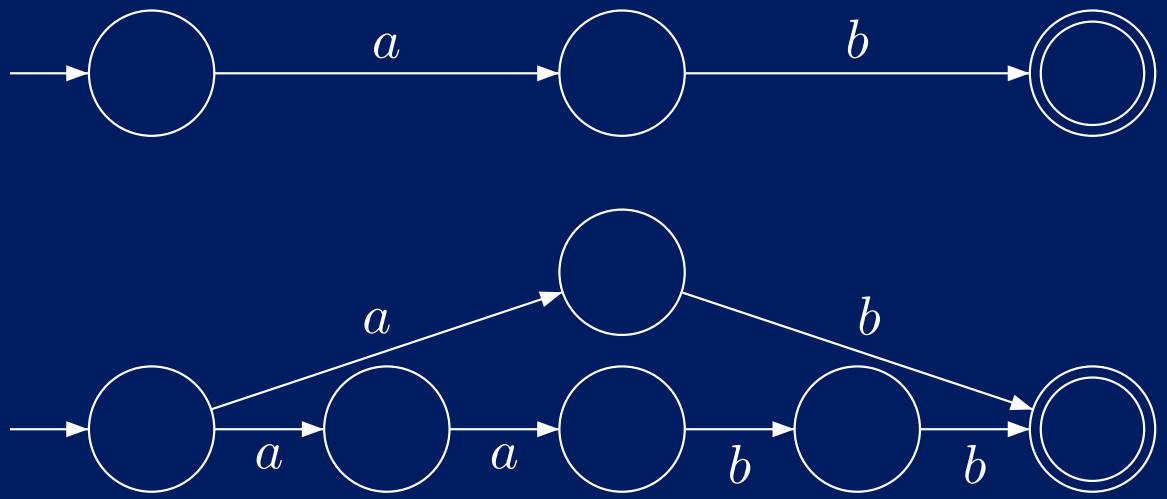
Regular expressions, capable of recognising strings of unbounded length, for example, the string aaa and the string $aaaaaaaaaa$ are both recognised by the regular expression a^*.



Regular expressions can be visualised as finite automata, the graph depicted above is such an automaton.

Type-3 parsers

Though regular expressions can recognise complicated strings (pattern matching), they are incapable of "counting" (the string $a^n b^n$ can not be recognised for each n).



This limitation disqualifies this class of languages when we want to use nested expressions.

Type-2 parsers

These parsers are commonly used in compilers and interpreters of programming languages, e.g., Perl, C, Java, etc.

The restriction for these types of parsers is that the language is context-free.

Because of the nesting, the HGVS nomenclature is actually a context-free language. So we need a parser of this type to interpret the given string.

An other problem

The nomenclature is not static (yet).

Because of this, we shall need frequent alterations in our parser, to reflect the newly added (or deleted) rules.

We need a way to make the rules insightful.

Example: A calculator

Suppose we want to parse the input of a calculator that is capable of using variables:

$a = 1$

$b = 3$

$c = a * 2$

$d = b/c$

Somehow, we need to catch these expressions in general rules.

Example: A calculator

integer	$\rightarrow$	$\ldots \mid$ -2 $\mid$ -1 $\mid 0 \mid 1 \mid 2 \mid \ldots$
varname	$\rightarrow$	$a \mid b \mid c \mid \ldots \mid z$
equals	$\rightarrow$	$=$
operator	$\rightarrow$	$+ \mid - \mid * \mid /$
operand	$\rightarrow$	integer $\mid$ varname
unaryoperation	$\rightarrow$	operand
binaryoperation	$\rightarrow$	operand operator operand
operation	$\rightarrow$	unaryoperation $\mid$ binaryoperation
expression	$\rightarrow$	varname equals operation

This notation is BNF, short for Backus-Naur Form.
('|' means 'or').

The nomenclature rules (a sample)

Ref	→	((RefSeqAcc | GeneSymbol) ‘:’)? RefType?
Extent	→	PtLoc ‘_’ (‘o’? (RefSeqAcc | GeneSymbol) ‘:’)? PtLoc
RangeLoc	→	Extent | ‘(‘ Extent ‘)’
Loc	→	PtLoc | RangeLoc
FarLoc	→	(RefSeqAcc | GeneSymbol) (‘:’ RefType? Extent)?
Subst	→	PtLoc Nt ‘>’ Nt
Del	→	Loc ‘del’ (Nt^+ | Number)?
Dup	→	Loc ‘dup’ (Nt^+ | Number)? Nest?
AbrSSR	→	PtLoc Nt^+ ‘(’ Number ‘_’ Number ‘)’
VarSSR	→	(PtLoc Nt^+ ‘[’ Number ‘]’) | (RangeLoc ‘[’ Number ‘]’)

Avoid ambiguity

If there are more than one paths to evaluate an expression, the language is ambiguous, so it might mean more than one thing.

Note that more than one expression for the same thing **is** allowed.

Ambiguous languages can not be parsed by a Type-2 parser any more (and they are difficult to interpret for humans as well).

We were forced to made a couple of changes to the nomenclature to avoid this problem.

LUMC

Translation to code

Fortunately, since there are many programming languages and new programming languages are developed each day, there are tools available to generate a parser, once the language has been defined.

Parser generators make a programmer able to use the language definition (BNF) almost directly as code. This makes the code highly flexible and maintainable.

Translation to code

```
Ref = Optional((RefSeqAcc ^ GeneSymbol) + Suppress(':')) +
      Optional(RefType)
Extent = Group(PtLoc("PtLoc"))("StartLoc") +
         Suppress('_') + Group(Optional(Group(Optional('o')
         + (RefSeqAcc ^ GeneSymbol) + Suppress(':') +
         Optional(RefType)))("OptRef") +
         PtLoc("PtLoc"))("EndLoc")
RangeLoc = Extent ^ (Suppress('(') + Extent +
           Suppress(')'))
Loc = Group(PtLoc("PtLoc"))("StartLoc") ^ RangeLoc
FarLoc = (RefSeqAcc ^ GeneSymbol) + Optional(Suppress(':')
         + Optional(RefType) + Extent)
Subst = Group(PtLoc("PtLoc"))("StartLoc") +
        Nt + Literal('>')("MutationType") + Nt
```

The parser as an acceptor

```
> Parser "NG_008660.1(CAPN3):c.[496delC"
Error: Expected "]" (at char 29), (line:1, col:30)
NG_008660.1(CAPN3):c.[496delC
                            ^
NG_008660.1(CAPN3):c.[496delC]

> Parser "NG_008660.1(CAPN3_v1)c.496delC"
NG_008660.1(CAPN3_v1)c.496delC
Error: Expected ":" (at char 21), (line:1, col:22)
NG_008660.1(CAPN3_v1)c.496delC
                      ^
NG_008660.1(CAPN3_v1):c.496delC
```

The parse tree

```
> Parse "NG_008660.1(CAPN3_v1):c.[496delC;500A>T]"
```

RefSeqAcc:	NG_008660
RefType:	c
Version:	1
Gene Symbol:	CAPN3
Transcript variant:	1

RawVar	
- Position:	496
- Type:	del

RawVar	
- Position:	500
- Type:	>
- From:	A
- To:	T

Questions?

Acknowledgements:

- Peter Taschner
- Johan den Dunnen