UKRAINIAN CATHOLIC UNIVERSITY

MASTER THESIS

Generation of code from text description with syntactic parsing and Tree2Tree model

Author:
Anatolii STEHNII

Supervisor:
Dr. Rostyslav HRYNIV

A thesis submitted in fulfillment of the requirements for the degree of Master of Computer Science

in the

Department of Computer Sciences
Faculty of Applied Sciences

Lviv 2017

Declaration of Authorship

I, Anatolii STEHNII, declare that this thesis, titled "Generation of code from text description with syntactic parsing and Tree2Tree model" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

UKRAINIAN CATHOLIC UNIVERSITY

Abstract

Faculty of Applied Sciences

Master of Computer Science

Generation of code from text description with syntactic parsing and Tree2Tree model

by Anatolii STEHNII

Software development requires vast knowledge of different programming tools which cannot be kept in human memory. Therefore software developers often formulate their task in human language to query online knowledge bases like StackOverflow to get short snippets of code. In this work, we explored the way of code generation from natural language description and prepared web API for Python which translates NL descriptions to short snippets of code. Our model implements sequence-to-sequence model with recursive encoder and uses syntactic trees instead of plain sequence on input. Results have not outperformed current state-of-the-art performance. However, presented Tree2Tree model has potential in other applications and this work makes a solid base for a further research.

Acknowledgements

I would like to thank Vsevolod Dyomkin and Rostyslav Hryniv who supervised and directed my research for this thesis. Special thanks to Artem Chernodub for hints about neural network implementation.

This work would not be possible without computational resources for a neural network training provided by ArviLab. Thanks to CoreNLP and EasyCCG developers, who provided great tools for a natural language parsing. And thanks to PyTorch team for an easy and handy framework.

Last, but not least I am grateful to Ukrainian Catholic University and Oleksii Molchanovskyi personally for the first master program in data science in Ukraine. I also want to thank Grammarly, Inc for covering my tuition fees.

Contents

List of Figures

List of Tables

List of Abbreviations

AST	**A**bstract **S**yntax **T**ree
BiLSTM	**Bi**directional **L**ong **S**hort-**T**erm **M**emory
BLEU	**B**ilingual **E**valuation **U**nderstudy
CCG	**C**ombinatory **C**ategorial **G**rammar
CFG	**C**ontext **F**ree **G**rammar
DNN	**D**eep **N**eural **N**etwork
IDE	**I**ntegrated **D**evelopment **E**nvironment
LSTM	**L**ong **S**hort-**T**erm **M**emory
MT	**M**achine **T**ranslation
NMT	**N**eural **M**achine **T**ranslation
NL	**N**atural **L**anguage
OOV	**O**ut **O**f **V**ocabulary
PBG	**P**hrase **B**ased **G**rammar
PCFG	**P**robabilistic **C**ontext-**F**ree **G**rammar
ReLU	**Re**ctified **L**inear **U**nit
RNN	**R**eccurent **N**eural **N**etwork
RvNN	**R**ecursive **N**eural **N**etwork
Seq2Seq	**S**equence-to-**S**equence
Seq2Tree	**S**equence-to-**T**ree
SRvN	**S**imple **R**ecursive **N**etwork
Tree2Tree	**T**ree-to-**T**ree

List of Symbols

$\cdot$	matrix multiplication
$\circ$	entrywise multiplication
$[x1, x2]$	concatenation of vectors $x1$ and $x2$
$p(y\|x)$	probability of y given x
P	vector of probability distribution
τ	abstract syntax tree
η	node in natural language syntax tree
$x^{(t)}$	LSTM input vector for sequence step t
$h^{(t)}$	LSTM hidden state vector for sequence step t
$c^{(t)}$	LSTM memory cell vector for sequence step t
$f^{(t)}$	LSTM forget gate vector for sequence step t
$i^{(t)}$	LSTM input gate vector for sequence step t
$u^{(t)}$	LSTM memory cell input vector for sequence step t
$o^{(t)}$	LSTM output gate vector for sequence step t
$h_e^{(t)}$	encoder embedding vector for encoding step t
H_e	matrix of encoder embeddings
$h_d^{(t)}$	decoder embedding vector for decoding step t
H_d	matrix of decoder embeddings
$\alpha^{(t)}$	attention vector for decoding step t
$\phi^{(t)}$	context vector for decoding step t
$a^{(t)}$	AST generation action for step t
$r^{(t)}$	AST production rule for step t
$v^{(t)}$	AST terminal token for step t
$e(v)$	one hot encoding for vocabulary item v
w	word embedding vector
W	linear transformation weights
b	linear transformation bias
$f_{pointer}$	pointer network function
f_{lstm}	Long Short-Term Memory single step function
$softmax$	normalized exponential function
$tanh$	hyperbolic tangent function

To my patient and loving family

Chapter 1

Introduction

1.1 Motivation

Software development is often described as a knowledge-intensive field (Robillard, 1999). Implementation and maintenance of enterprise software systems require broad knowledge of various programming languages and application of programming interfaces. While in 2002 to create a website a developer had to know HTML/CSS, PHP and MySQL, in 2017 this requires knowledge about frontend ecosystem, backend frameworks, and different NoSQL query languages. Documentation becomes a bottleneck while solving simple tasks, especially for new developers. For that reason software development involves regular use of search engines and Q&A databases (Treude, Barzilay, and Storey, 2011). Code snippets from crowdsourced resources like StackOverflow are adopted and reused in other projects. Developers often seek to find existing examples of working code to solve regular tasks instead of writing and testing it from scratch (Brandt et al., 2010). And to find corresponding code snippet, the software developers first formulate its description as a query for search engine (Brandt et al., 2009).

However, web search is a time-consuming task, which causes interruptions of the coding process. As an alternative, the code description could be translated directly to code. Such translation tool would reduce the burden of remembering the details of a particular language or API and allow a developer to use his or her time for more creative aspects of development. That was a motivation of the present work: we wanted to develop a model of description-to-code translation, which could transform informal instrcutions to actual language specific implementation.

1.2 Goals of the master thesis

1. To explore previous examples of code generation tools.
2. To train a Description2Code syntactic model and compare its performance to previous works.
3. To develop code generation plugin for PyCharm IDE.

1.3 Thesis structure

This work is structured as follows: In chapter 2 we have a review of related publications and presented a comparison with previous code generation projects. In chapter 3 we provided a theoretical background for methods we used in this work. In chapter 4 we explained the idea of a Tree2Tree model with all details about its structure and implementation. In chapter 5 we described data preprocessing, explained the model implementation details and presented evaluation results. And finally, in chapter 6 we drawn the conclusion and set the points for a further research.

Chapter 2

Related works

2.1 Automatic programming

A problem of translation of high-level specifications to low-level instructions was recognized at the early years of computational industry. It addresses the important goal of computer science and artificial general intelligence — to shift the burden of requirements understanding and instructions implementation from a human to a machine. The notion of *automatic programming* was first established in FORTRAN compiler in 1957 (Backus et al., 1957) and used as a prototype of a high-level programming concept. Later the automatic programming split into two complementary approaches: bottom-up and top-down (Balzer, 1985). In the first approach, a specification language is developed as a set of high-level functions and modules. And in the second approach, informal specification language is translated to a formal level, which can be compiled automatically. While the first approach was a background for high-order programming languages and frameworks, the second approach gave raise to an automatic code generation field.

First attempts to build automatic programming system addressed the roles of symbolic evaluations, deduction and programming knowledge in the programming process. That was coherent with symbolic artificial intelligence, a dominant paradigm in artificial intelligence research from the mid-1950s until the late 1980s (Haugeland, 1989). Green, 1969 and Lee, Waldinger, and Chang, 1974 was focused on the use of theorem-prover to produce the programs. In 1976, the PSI program synthesis system (Green, 1976; Green et al., 1977) was concerned with coding a high-level program knowledge from requirements collected via a dialog with a user. It used a set of expert modules to build a program model from natural language and generate a code for this model. For example, one of the generator modules was PECOS (Barstow, 1979), which used a set of symbolic rules to design abstract algorithms like a sorting or a path finding.

Another automated coding attempt was made in 1978 with project SAFE (Balzer, Goldman, and Wile, 1978). It used the semantic parsing to resolve ambiguity in informal specifications and translate them into a symbolic representation. While SAFE was a laboratory prototype designed to solve a limited set of tasks, its results were used in further automated programing researches like a specification language Gist (Balzer, 1985) and automatic requirement derivation system SPECIFIER (Miriyala and Harandi, 1991).

Although these works have given a great advance in ideas about knowledge representation and informality translation, they have major flaws. Symbolic approach to artificial intelligence was criticized (Harnad, 1990; McDermott, 1987) for a symbolic grounding problem and problems with uncertainty representation. Dreyfus, 1994 argued that symbols and formal rules could not catch unconscious instincts which form the human intelligence. Therefore further research of automatic programming addressed this problem with statistical machine learning methods like neural networks and a representation learning.

2.2 Deep learning

Deep neural networks have two major advantages for the natural language modeling and translation. The first is *representation learning* (Bengio, Courville, and Vincent, 2013), which allows to transform data into the representation which contains important features for a current task. This feature allows to create knowledge representations of informal instructions automatically, without complex preparations. And the second is the ability to approximate statistical distribution prior to some conditions (White, 1992), which allows to deal with uncertainty in the language interpretation.

With invention of word embeddings (Bengio et al., 2003) sequences of informal instructions became a possible input for neural models. Great advance in language modeling was introduced by *recurrent neural networks* (Gers and Schmidhuber, 2001; Hochreiter and Schmidhuber, 1997; Jozefowicz et al., 2016; Sundermeyer, Schlüter, and Ney, 2012) which were able to capture features encoded in a sequential structure of the natural language. *Sequence-to-sequence models* (Sutskever, Vinyals, and Le, 2014) with recent novel methods like *attention technique* (Bahdanau, Cho, and Bengio, 2014; Jean et al., 2014; Luong, Pham, and Manning, 2015) allowed to surpass results of phrase-based machine translation in production level systems like Google Translation (Wu et al., 2016).

Code generation with neural models was interpreted as a neural machine translation problem and used an established approach — sequence-to-sequence model with attention. Ling et al., 2016 proposed an architecture of latent predictor networks with structured attention for generation of Magic: The Gathering and HearthStone cards implementation from card description in Java and Python. Chen et al., 2016 used a latent attention to generate If-Then recipes for natural descriptions for IFTTT.com dataset. A remarkable idea of pointer networks introduced by Vinyals, Fortunato, and Jaitly, 2015 allow to re-use parts of an input sequence in an output. It was used by Zhong, Xiong, and Socher, 2017 along with the reinforcement learning for a generation of SQL queries from informal questions (Bhoopchand et al., 2016).

2.3 Snippet generation

Instead of end-to-end translation of complex high-order requirements to a software system into compiled instructions, code generation could be used as a handy *snippet generation* tool. For example, Little and Miller, 2009 proposed a keyword based generation of Java code and implemented it in an integrated development environment plugin for Eclipse. Gvero and Kuncak, 2015 addressed the problem of description-to-code generation as a machine translation task and used probabilistic context-free grammar model to generate a list of ranked code expressions for a developer. Codehint (Galenson et al., 2014) is another plugin for Eclipse which explores Java virtual machine execution state to build a search space of possible expressions for a given description and then to select the most likely one using the statistical model built offline from existing projects. NaturalJava (Price et al., 2000) uses decision trees to infer a Java abstract syntax tree from English sentences. Allamanis et al., 2015 created bimodal models of language and code to generate a code from a description and reversed for C#.

Another common approach in the code generation is a creation of a code for a predefined context. In other words, creation a code for defined input and output of a module. These solutions allow to fill gaps and create connections in the program template. Raychev, Vechev, and Yahav, 2014 used statistical language models to synthesize a code for holes in application programming interfaces with a most likely sequence of statements. Jha et al., 2010 proposed code generation approach with I/O oracle, constrained to input-output behavior and a set of available components. Search in the expression space performed with off-the-shelf Satisfiability Modulo Theory solvers. Domain-specific solution StreamBit (Solar-Lezama et al., 2005) shifts the task of most complex and error sensitive bit-level operations from a programmer to a code generator. A developer only needs to write a sketch — a domain-specific description which then gets translated to a C implementation. Srivastava, Gulwani, and Foster, 2010 used verification tools to perform a proof-theoretic program synthesis with a given input-output functional specification and a specification of the synthesized programs looping structure.

2.4 Semantic language models

A semantic model of the natural language is a meaning representation which could be understood by the machine. Tasks like natural language understanding or paraphrase detection heavily relies on semantic models like abstract meaning representation (Banarescu et al., 2013) or combinatory categorial grammar (Clark and Curran, 2007). These models also could be domain specific formal representation like a natural language database interface (Berant et al., 2013; Zettlemoyer and Collins, 2012), instructions for a robot (Artzi and Zettlemoyer, 2013) or a smart home instructions (Quirk, Mooney, and Galley, 2015). Since a semantic meaning representation already contains formal instructions, we believe that their use for code generation task could significantly improve the result.

Until recently in the neural modeling of natural language dominated an idea that neural networks do not require any information about a syntactic structure of sentences. Outstanding results of deep neural networks in the representation learning and the structure parsing suggested to use a plain sequence of words as an input for a neural model and allow it to learn a representation of all other important features backpropagating a gradient of error. However, recent results of syntactic structures usage for the machine translation (Chen et al., 2017) and the reading comprehension (Xie and Xing, 2017) outperform the result of Seq2Seq models. To parse syntactic structures, *recursive neural networks* (Goller and Kuchler, 1996; Socher et al., 2011) and *recurrent neural networks grammar* (Dyer et al., 2016) were used. Recursive neural networks also have shown good results for the semantic (Tai, Socher, and Manning, 2015) and dependency parsing (Zhu et al., 2015).

Another approach of a syntax structure usage was addressed by Dong and Lapata, 2016; Rabinovich, Stern, and Klein, 2017; Yin and Neubig, 2017. In these works an augmented decoder was used to generate a tree (syntax tree for syntactic parsing or abstract syntax tree for a code generation) instead of a sequence. These models reduced model search space by inferring prior knowledge about target language syntax into their information flow.

2.5 Comparison with other works

Keyword programming approach (Little and Miller, 2009) is tailored to a Java syntax and requires substantial work to be reused in other languages. The same problem has plugin anyCode, described in Gvero and Kuncak, 2015. Plugin Codehint (Galenson et al., 2014) implemented a handy user experience for code generation. It uses specially marked comments as a place marker to insert generated code. However, its approach of usage Java virtual machine state as exploration space could not be transferred to dynamic typed languages like Python. The model described in our work does not require to run a program to generate a target code. And, it is language agnostic and could be trained for any language requiring only an appropriate dataset of code lines and descriptions.

Architecture described in Zhong, Xiong, and Socher, 2017 shows an interesting approach with pointer networks for reuse of description parts, for example variables or function names. However, it is domain specific, since it is a natural language database interface, thus it can not be compared with general use code generation. Chen et al., 2016 and Ling et al., 2016 also described only domain specific code generation.

Yin and Neubig, 2017 proposed to use a vanilla long short-term memory encoder and an augmented long short-term memory decoder with additional neural connections to reflect the structure of abstract syntax tree. This idea has a great potential as it infers the code model in its structure. In our work the above approach is modified by using Tree-LSTM as encoder with parsed semantic trees as input. We believe, that usage of syntactic tree information in the model can enhance the instructions understanding and improve the model results.

Chapter 3

Background information and theory

3.1 Abstract syntax tree

The *abstract syntax tree* is a tree representation of an abstract structure of programming code. For each expression or statement in the code, abstract syntax tree assigns the corresponding node. Abstract syntax tree could not contain all the details of the underlying code like parentheses or indentation, but its structure allows to interpret a code execution process unambiguously. The established grammar of abstract syntax tree reduces search space of the model and implies the validity of generated code.

The Python *abstract grammar* contains a set of production rules, composed of a head node and multiple child nodes. For example, in fig. 3.1 the first rule is used to generate the assignment of statement result to a variable. It consists of the head node of type `Assign` and two child nodes of types `Name` and `Call`, respectively. Non-terminal nodes (the blue ones) defines a structure of the target code, while terminal nodes (the green ones) refer to symbol tokens, like variables, constants or operations. Full Python 3.6 abstract grammar can be found in appendix A.

[1]Picture is created in Python library showast.

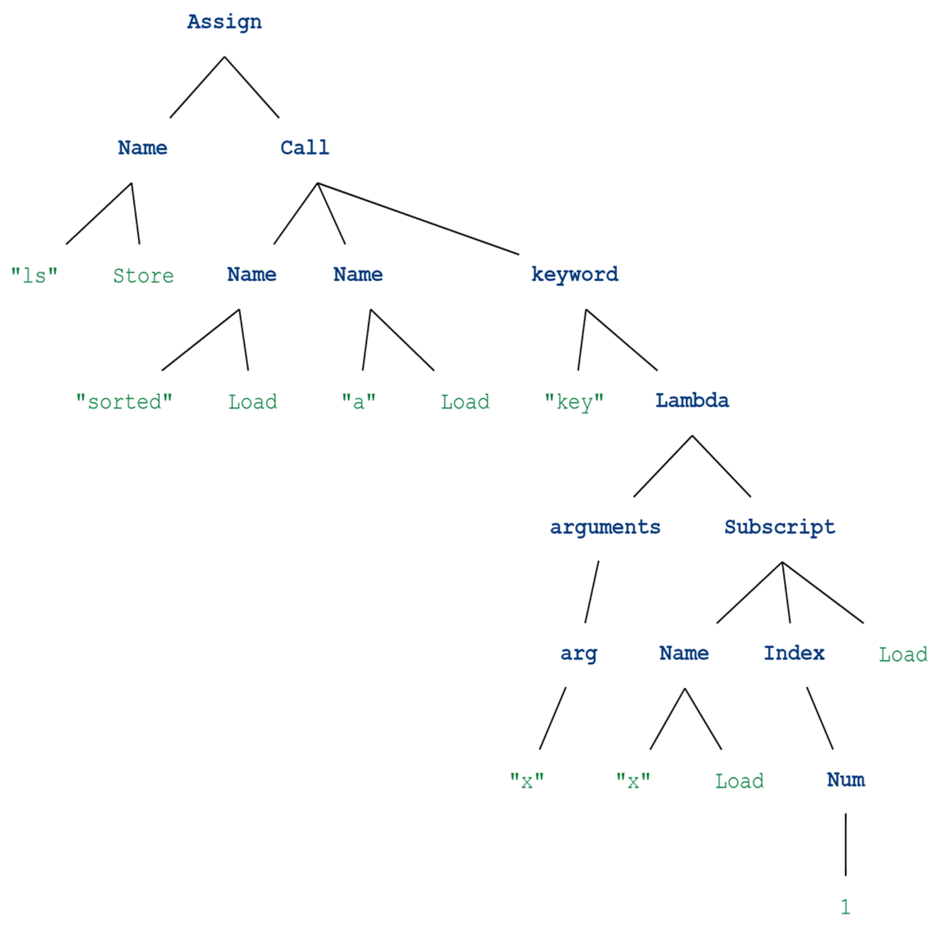

FIGURE 3.1: Abstract syntax tree of code snippet `ls = sorted(a, key=lambda x: x[1])`[1].

3.2 syntactic parsing

3.2.1 Constituency parsing

Theory of sentence analysis is derived from the idea that the words of a sentence seem to combine into patterns and structures. Each word is classified as a different in terms of its function in a sentence. According to this function, to each word there can be assigned its lexical category or part of speech, like a noun (`N`), an adjective (`A`), or a verb (`V`).

This system could be extended to the level of syntactic categories, combining words or other syntactic categories with a similar function into phrases. Each phrase is characterized by the properties of a headword that it includes. For example, the headword of the subject of a sentence is a noun, so it is classified as a noun phrase (`NP`). A verb is regarded as the head of the sentence predicate, so predicate is classified as a verb phrase (`VP`). Rules which describe how lexical categories can be combined into syntactic categories is called *rewriting rules*. For example, a noun phrase category may be described as the parent of adjective and noun categories, and it may be represented by a rule of the following form:

```
NP => A N
```

A formalization of such a rule-based sentence structure system is called a *phrase-based grammar* or a *constituency grammar*. A grammar is defined by the following constituents: V_N is a non terminal vocabulary, which contains the lexical and syntactic category labels; V_T is a terminal vocabulary and it contains a set of words. P identifies a collection of the production rules of the grammar. To illustrate this concept, we present a short grammar to parse a sentence "good dogs like cats" to its syntactic representation. The grammar for this example contains the following vocabulary:

$$\begin{aligned} V_N &= S, NP, VP, A, N, V \\ V_T &= cats, dogs, like, good \end{aligned} \tag{3.1}$$

The production rules for this grammar would be following:

```
S => NP VP
NP => A N
NP => N
VP => V NP
N => dogs
N => cats
V => like
A => good
```

A graphic representation of the syntactic tree of this sentence can be seen in fig. 3.2.

The phrase-based grammar might also be referred as the *context-free grammar*, because it provides clear mechanism for combining a phrases from their constituents without usage of sentence context.

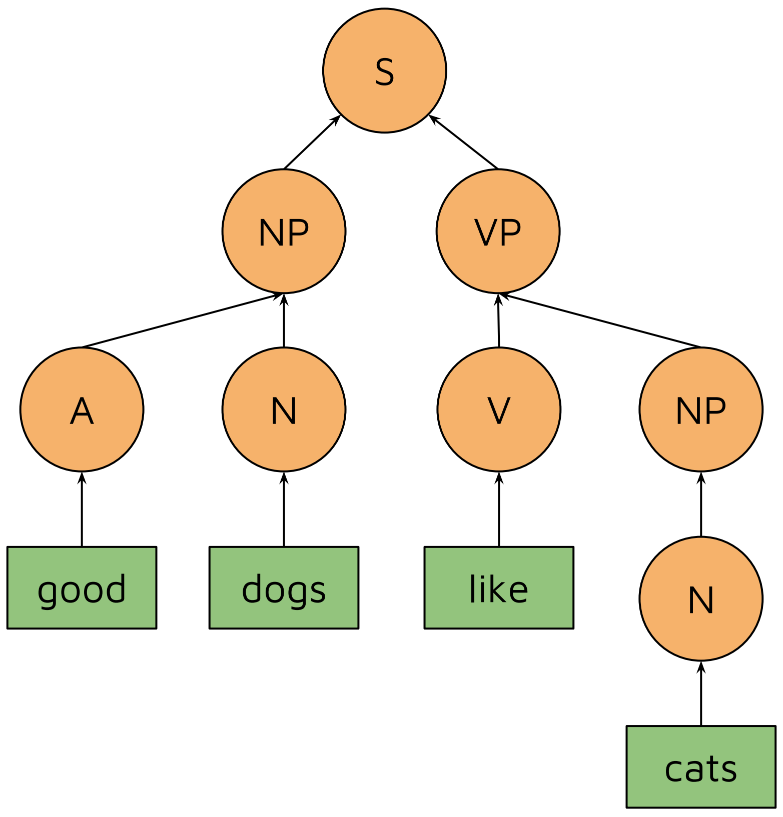

FIGURE 3.2: Example of syntax tree.

3.2.2 Combinatory-categorial grammar

Phrase-structure grammars analyze the sentence recursively applying rewriting rules, starting from identification of the parts of speech or lexical categories of individual words. This rule governs how words can be combined into phrases and sentences. Compared with phrase-structure grammars, *combinatory-categorial grammars* do not contain a separate collection of category-combining rules. Lexical categories of words such as adjectives and nouns describe the functions that determine how these words can combine with other categories. Each node in CCG tree can be translated into lambda calculus representation (Artzi, FitzGerald, and Zettlemoyer, 2013), thus CCG is a transparent interface between syntax and semantics of a sentence.

Consider an example. Adjective "good" is in the category corresponding to a function that maps from the nouns into the noun phrases. The association of this item with a function looks like that:

```
good = NP/N
```

$\lambda x.good(x)$[2]

`NP` on the left side of the slash character denotes a function return value and `N` on the right side denotes a function argument. This function could be resolved with a forward function application operation, denoted by character >:

```
dogs => N
good dogs => NP/N>N = NP
```

$\lambda x.good(x)(DOGS) = good(DOGS)$

With the backward slash character \ the category on the right side denotes a function return and the category on the left side denotes a function argument. Example:

```
bird => N
flies => N\S
bird flies => N>N\S = S
```

$\lambda x.flies(x)$

$\lambda x.flies(x)(BIRD) = flies(BIRD)$

[2]To make connection of CCG categories with the lambda calculus clear, we will complement each example of categorial operations with the corresponding lines of lambda calculus.

A verb such as “like” is usually taking two arguments and associated as follows:

```
like => (S/NP)\N
cats => N
like cats => (S/NP)\N>N = S/NP
```

$\lambda x.\lambda y.like(x,y)$

$\lambda x.\lambda y.like(y,x)(CATS) = \lambda y.like(y,CATS)$

The function `(S/NP)\N` maps `N` to a range of functions of the form `S/NP`. Character < denotes backward function application. A final sentence example:

```
good dogs => NP
like cats => S/NP
good dogs like cats => NP<S/NP = S
```

$\lambda y.like(y,CATS)$

$\lambda y.like(y,CATS)(good(DOGS)) = like(good(DOGS),CATS)$

3.2.3 Dependency parsing

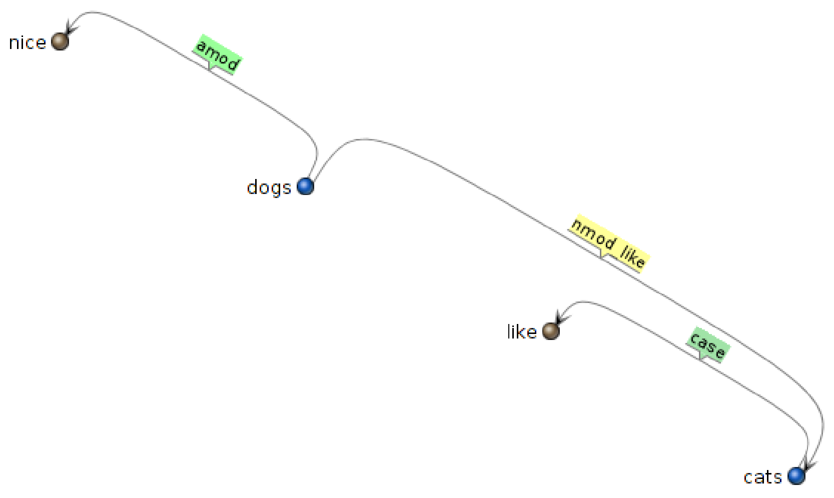

FIGURE 3.3: Dependency graph for sentence “nice dogs like cats”[3].

Another approach to represent a sentence semantical structure is the *dependency parsing*. *Dependency grammar* is different from constituency grammars like CFG or CCG, which build sentence tree by applying rewrite rules to constitute high level phrases from low level syntactic categories and lexems. The dependency grammar also defines sentence structure as a graph, but without phrasal nodes like `NP` or `VP`. Instead each node represents one word which points to it dependents. Since such representation do not rely on established phrase word order it is well suited for the analysis of languages with free word order, such as Ukrainian.

In fig. 3.3 sentence "nice dogs like cats" is parsed to dependency graph. It consists of the following dependencies:

```
amod = adjectival modifier
nmod = nominal modifier
case = case marker
```

3.3 Word embeddings

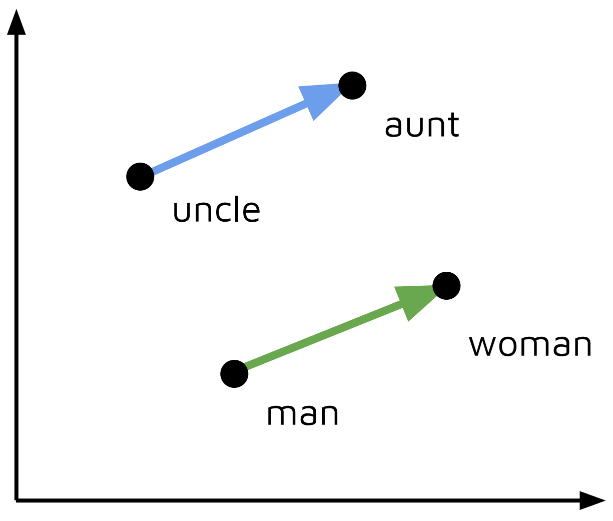

FIGURE 3.4: Example of collinearity between two word vectors.

[3]Picture is created in GrammarScope.

The task of language modeling requires the transformation of words and documents into vector representation. Simple tasks like text classification could be done using naive representations like a bag of words or one hot encoding. However, these approaches would require excessive memory usage to handle large vocabulary and usually do not infer existing semantical connections between words in a language.

Methods of *word embeddings* solve both problems, providing dense vectors of real numbers, which represent word positions in a n-dimensional space. This space represents the contextual similarity of the words, thus word embeddings support semantic relations as vector operations. For example, adding a vector woman to a vector uncle and subtracting a vector man will result in a vector approximately pointing to the same point as the vector aunt (see fig. 3.5).

3.4 Long-short term memory network

Model of LSTM network is an extension of the simple recurrent network. It can store values in the hidden layer for a short or long period of time because it uses no activation function within its recurrent components. This makes it possible to backpropagate error gradient through long sequences of data without gradient vanishing or gradient exploding. This properties allow LSTM to catch long term patterns in the input sequences and make it a perfect choice for natural language modeling.

At the time step t LSTM consumes a previous value of a hidden state $h^{(t-1)}$, a memory cell $c^{(t-1)}$ and an input vector $x^{(t)}$. The new value of memory cell uses gates to forget a part of the previous value and remember a new value. An input gate calculation:

$$i^{(t)} = \sigma(W_i \cdot [h^{(t-1)}, x^{(t)}] + b_i) \tag{3.2}$$

A forget gate:

$$f^{(t)} = \sigma(W_f \cdot [h^{(t-1)}, x^{(t)}] + b_f) \tag{3.3}$$

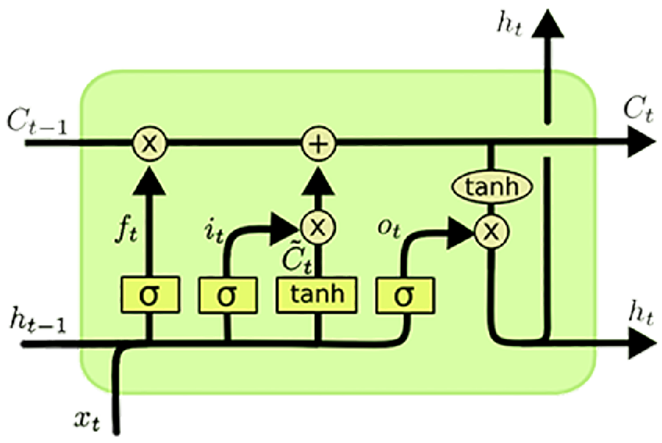

FIGURE 3.5: Long-short term memory network achitecture (Olah, 2015).

A new memory state:

$$\begin{aligned} u^{(t)} &= tanh(W_u \cdot [h^{(t-1)}, x^{(t)}] + b_u) \\ C^{(t)} &= f_t \circ C^{(t-1)} + i_t \circ u^{(t)} \end{aligned} \tag{3.4}$$

A new hidden state:

$$\begin{aligned} o^{(t)} &= \sigma(W_o[h^{(t-1)}, x^{(t)}] + b_o) \\ h^{(t)} &= o^{(t)} \circ tanh(C^{(t)}) \end{aligned} \tag{3.5}$$

An LSTM step can be presented as a function:

$$h^{(t)}, c^{(t)} = f_{lstm}(x^{(t)}, h^{(t-1)}, c^{(t-1)}) \tag{3.6}$$

The value of the memory cell often is not used in further computations and thus can be omitted:

$$h^{(t)} = f_{lstm}(x^{(t)}, h^{(t-1)}) \tag{3.7}$$

For the task of natural language processing, an important information about a current word can be stored not only in the previous words but also in the next words of the sentence. To catch this information a model of bidirectional LSTM (BiLSTM) was proposed (Graves, Fernández, and Schmidhuber, 2005; Schuster and Paliwal, 1997). It represents each word as a concatenation of a forward and a backward embedding:

$$\begin{aligned} h^{(t)}_{forward} &= f_{lstm.forward}(x^{(t)}, h^{(t-1)}_{forward}) \\ h^{(t)}_{backward} &= f_{lstm.backward}(x^{(t)}, h^{(t+1)}_{backward}) \\ \mathrm{h}^{(t)} &= [h^{(t)}_{forward}, h^{(t)}_{backward}] \end{aligned} \tag{3.8}$$

3.5 Sequence-to-sequence machine translation

The task of the machine translation can be formalized as a mapping of a sequence of words in a source language to a sequence of words in a target language (Neubig, 2017). This task can be solved with an *encoder-decoder* model, which consists of two RNNs. The first RNN consumes input sequence step by step and encodes it into so-called thought vector. After encoding, the second RNN uses thought vector as its initial hidden state and decodes an output sentence word by word. Each word from the decoder output is used as an input for the next decoding step until the model outputs the end-of-sentence token.

During the training, Seq2Seq model learns its parameters maximizing the log-likelihood $P(y|x)$ of the target sequence y given the input sequence x. The next input of the decoder can be selected in two ways:

- **Without Teacher Forcing:** the previous prediction used as the next input.
- **With Teacher Forcing:** A value from the target sequence used as the next input.

3.5.1 Attention

Theoretically, a sufficiently large encoder-decoder model should be able to perform the machine translation perfectly. However, to encode all words and their dependencies in the arbitrary-length sentences, the thought vector should have enormous length. Such model would require massive computational resources to train and to use, thus this approach is ineffective.

This problem can be solved with attention technique (Bahdanau, Cho, and Bengio, 2014). Its basic idea is to replace a single vector representation of the input sentence with references to representations of different words in it. On the encoding step, each word representation $h_e^{(t)}$ is stored as a column of matrix H_e.

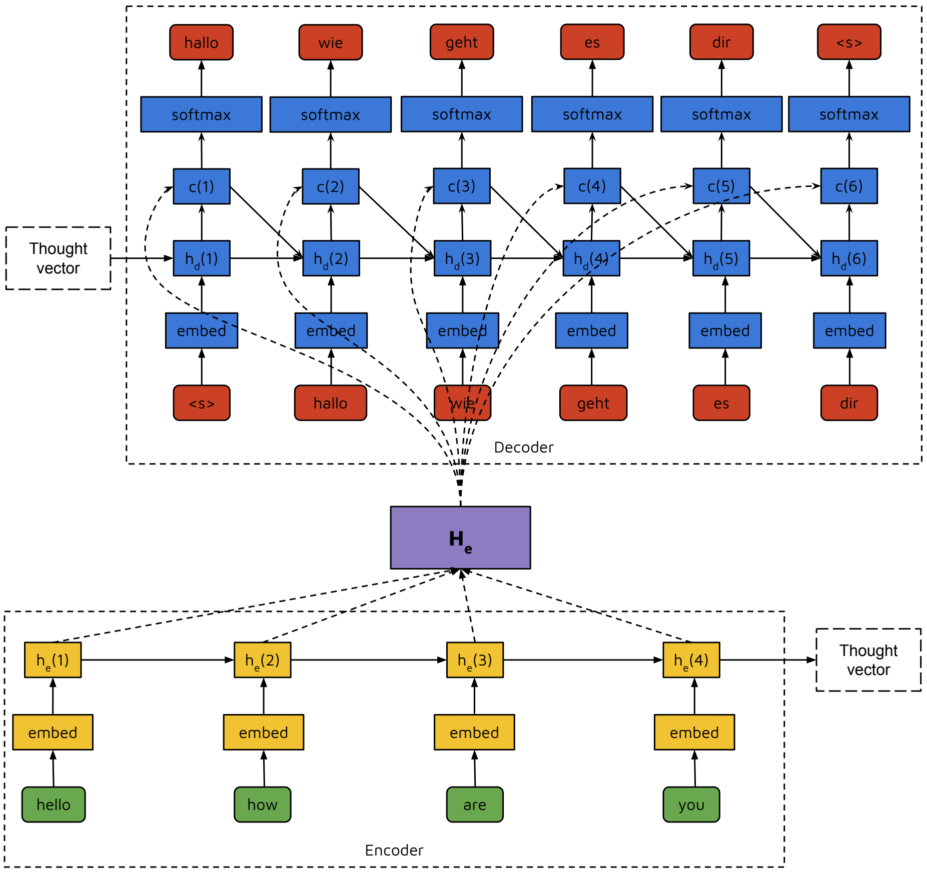

FIGURE 3.6: Seq2Seq with attention model.

During the decoding step, each decoder input is extended with context vector $\phi^{(t-1)}$:

$$h_d^{(t)} = lstm([w^{(t)}, \phi^{(t)}], h_d^{(t-1)}) \tag{3.9}$$

Context vector $\phi^{(t)}$ is calculated as a weighted sum of the encoder representations:

$$\phi^{(t)} = H_e \cdot \alpha^{(t)} \tag{3.10}$$

Weights for the attention vector $\alpha^{(t-1)}$ can be calculated with an arbitrary attention score function (for example, vector product) for each pair of the decoder vector $h_d^{(t)}$ and the encoder vector $h_e^{(i)}, \forall i \in 1..n$, where n is a length of the encoded sequence. In this work we used DNN with one hidden layer as suggested in the work of Bahdanau, Cho, and Bengio, 2014:

$$\begin{aligned}\hat{\alpha}_i^{(t)} &= W_{attn1} \cdot [h_e^{(i)}, h_d^{(t-1)}] \\ \alpha_i^{(t)} &= W_{attn2} \cdot tanh(\hat{\alpha}_i^{(t)}) \\ \alpha^{(t)} &= softmax(\alpha^{(t)})\end{aligned} \tag{3.11}$$

This way each decoder step can use information from an arbitrary part of the encoded sequence. The input representation will not be limited to the fixed length thought vector, and thus it can model natural language with any sequence and vocabulary size. An architecture of sequence-to-sequence model with attention can be seen in fig. 3.6.

3.5.2 Beam search

After the model learned the probability model on training examples, it can generate new translations. This can be done in several ways (Neubig, 2017):

- **Random Sampling:** For each time step t randomly select output words $w^{(t)}$ for y from the probability distribution $P(y|x)$.
- **Gready Search:** Find the y that maximizes $P(y|x)$, selecting each next word $w^{(i)}$ with maximum probability $\hat{w^{(t)}} = \underset{w^{(t)}}{\text{argmax}} P(w^{(i)}|x, w^{(<i)})$.
- **Beam Search:** Find the n outputs with the highest probabilities according to $P(y|x)$.

Beam search is similar to greedy search, but instead of considering only the one best hypothesis $\underset{w^{(t)}}{\text{argmax}} P(w^{(i)}|x, w^{(<i)})$, it is considering b best hypotheses at each time step, where b is the width of the beam. To find the best hypotheses, beam search explores the generation graph in the breadth-first manner. At each level of the search tree it calculates a probability for each candidate from the target space and then selects b variants with the highest probability. This way it allows to generate sequences with the higher cumulative probability, which could have been missed by the greedy search.

3.6 Recursive neural networks

The architecture of a recurrent neural network contains an inductive bias about a conditional probability of a target variable with the previous values in a sequence. Natural language can be modeled in this way, as a plain sequence of words. However, this approach ignores domain knowledge about the syntactic structure of a text. A syntactic tree contains important information about relations between individual terms and thus should not be omitted in the task of natural language modeling.

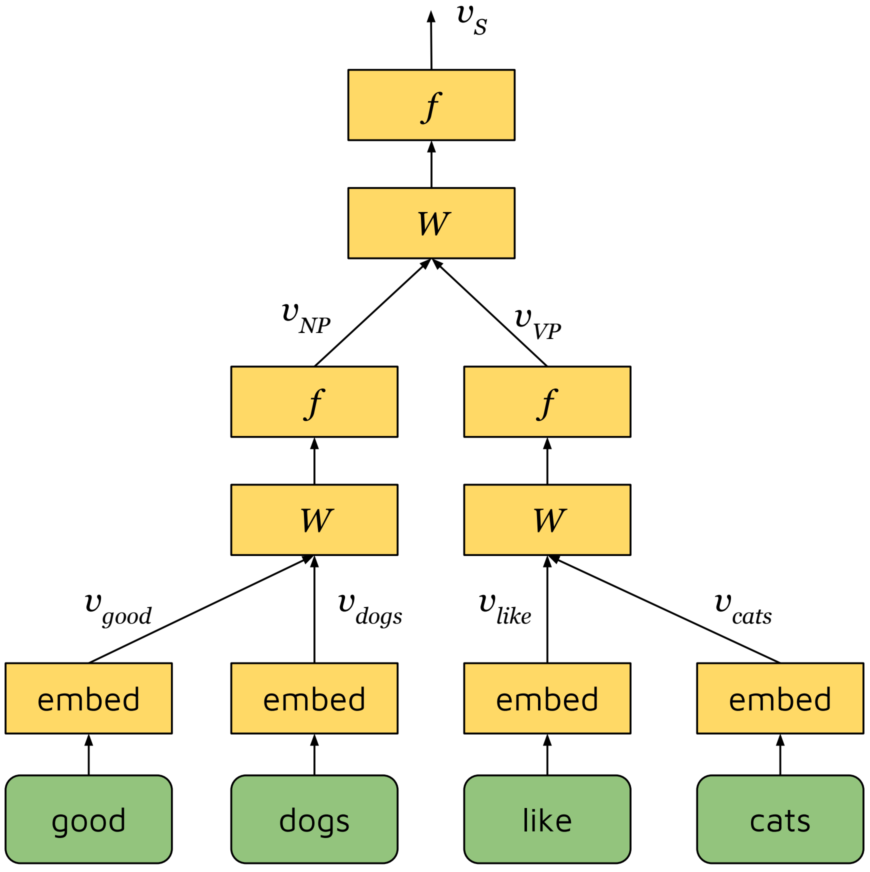

FIGURE 3.7: Examle of recursive neural network flow.

Topological structures with a variable length can be modeled by neural networks recursively applying the same set of weights to each node. To model a syntactic tree, each word is represented by a corresponding word vector and then parent vectors are computed using a bottom-up approach with composition functions. For example, representation of the syntactic tree of the sentence "good dog like cats" could be modeled in the following way:

$$\begin{aligned} h_{NP} &= f(W \cdot [w_{good}, w_{dog}] + b) \\ h_{VP} &= f(W \cdot [w_{like}, w_{cats}] + b) \\ h_S &= f(W \cdot [h_{NP}, h_{VP}] + b) \end{aligned} \tag{3.12}$$

where f is any differentiable non-linearity like $tanh$ or $ReLU$. The example of a semantic tree parsing with the recursive network is presented in fig. 3.7.

3.7 Pointer networks

A neural network operates with vector representations of words that are selected from a predefined vocabulary. This imposes the problem of unknown words that don't have a vector representation. This is especially important for the translation task where both an input and an output sequences could contain rare, special words or names. However, names of people or locations should not be translated but copied to a target sequence. Luong et al., 2015 proposed a solution of this problem with a *pointer network*. For each decoding step it calculates the probability of the next word to be copied from the input sequence. Calculation of this probability is described below step-by-step.

Let's denote $H_{encoder}$ as a matrix of encoder output vectors and $h^{(t)}_{decoder}$ - as a decoder output vector on decoding step t. First a hidden state of the pointer network is calculated:

$$\begin{aligned} \mathrm{H}_{e.pointer} &= H_e \cdot W_e \\ \mathrm{h}_{d.pointer} &= h^{(t)}_d \cdot W_d \\ \mathrm{H}_{pointer} &= tanh(H_{e.pointer} + h_{d.pointer}) \end{aligned} \tag{3.13}$$

Then each row from the matrix $H_{pointer}$ is translated to one value and a probability calculated as result of $softmax$:

$$\begin{aligned} \mathrm{H}_{prob} &= H_{pointer} \cdot W_{pointer} \\ \mathrm{P} &= \mathrm{softmax}(\mathrm{H}_{prob}) \end{aligned} \tag{3.14}$$

The vector P contains a probability for each input sequence token to be copied into the output sequence. In the following explanations, a pointer network function is denoted as $f_{pointer}$:

$$P = f_{pointer}(H_e, h_d^{(t)}) \tag{3.15}$$

Chapter 4

Model

4.1 Code generation problem

Given a natural language description x our task is to infer the Python code y based on the intent of the x. Python code y can be deterministically converted to an AST τ and vice-versa, though in this work a source code y and its abstract syntax tree τ are considered equivalent. A probabilistic grammar model of generating an abstract syntax tree τ given description x is defined as $P(\tau|x)$. The best corresponding syntax tree τ is defined as

$$\hat{\tau} = \underset{\tau}{\operatorname{argmax}}\, p(\tau|x) \tag{4.1}$$

Probability from eq. (4.1) is modeled with neural model with a set of weights θ. To learn values of θ we used a set of training examples, which consist of tuples $(\tau^{(x)}, x)$. The parameters of the model are learned by maximizing the conditional log-probabilities for the training set:

$$\theta = \underset{\theta}{\operatorname{argmax}} \sum_{\tau^{(x)},x} log\, p(\tau^{(x)}|x;\theta) \tag{4.2}$$

4.2 Abstract syntax tree generation

The output of decoder is an abstract syntax tree which consist of terminal and non-terminal nodes. As suggested in work of Yin and Neubig, 2017, we factorized the generation process of AST into sequential application of actions of two types:

- `ApplyRule[r]` corresponds to non-terminal nodes. It applies a production rule `r` to the current derivation tree.
- `GenToken[v]` corresponds to terminal nodes. It finishes the node by appending a token `v`.

Let us consider as example the AST from fig. 3.1. Its generation consist of the following actions:

Listing 1. AST production sequence.

```
ApplyRule[root => (stmt*)]
    ApplyRule[stmt => (Assign)]
        ApplyRule[Assign => (expr*{targets}), (expr{value})]
            ApplyRule[expr* => (expr)]
                ApplyRule[expr => (Name)]
                    ApplyRule[Name -> (str{id})]
                        GenToken["ls"]
                        GenToken["<eos>"]
            ApplyRule[expr => (Call)]
                ApplyRule[Call => (expr{func}), (expr*{args}), (keyword*{keywords}
                    ApplyRule[expr => (Name)]
                        ApplyRule[Name -> (str{id})]
                            GenToken["sorted"]
                            GenToken["<eos>"]
                    ApplyRule[expr* => expr]
                        ApplyRule[expr => (Name)]
                            ApplyRule[Name => (str{id})]
                                GenToken["a"]
                                GenToken["<eos>"]
                    ApplyRule[keyword* => keyword]
                        ApplyRule[keyword => (str{arg}), (expr{value})]
                            GenToken["key"]
                            GenToken["<eos>"]
                            ApplyRule[expr => (Lambda)]
...
ApplyRule[Lambda => (arguments args), (expr body)]
    ApplyRule[arguments => (arg* args)]
        ApplyRule[arg* => arg]
```

```
            ApplyRule[arg => (str{arg})]
                GenToken["x"]
                GenToken["<eos>"]
    ApplyRule[expr => Subscript]
        ApplyRule[Subscript => (expr{value}), (slice{slice})]
            ApplyRule[expr => (Name)]
                ApplyRule[Name => (str{id})]
                    GenToken["x"]
                    GenToken["<eos>"]
            ApplyRule[slice => (Index)]
                ApplyRule[Index => (expr{value})]
                    ApplyRule[expr => (Num)]
                        ApplyRule[Num => (int{n})]
                            GenToken["1"]
                            GenToken["<eos>"]
```

Under this grammar model, the probability of generating an AST τ is factorized as:

$$p(\tau|x) = \prod_{t=1}^{n} p(a^{(t)}|x, a^{(<t)}) \tag{4.3}$$

where a^t is the action taken at the time step t and $a_{<t}$ is the sequence of actions before t.

For each time step t the model selects the next action with a maximum probability — `ApplyRule` to grow the tree or `GenToken` to fill values in its terminal nodes. We must notice, that this model have one important flaw. List nodes like a `expr*` can not be expanded to an arbitrary number of children. Each number of children require a separate rule in the grammar, like `expr* => (expr), (expr)`, `expr* => (expr), (expr), (expr)` etc. Therefore this model will require huge vocabulary to cover all possible production rules in dataset with arbitrary length functions. We proposed our solution to this issue in section 6.2.

4.2.1 ApplyRule actions

At any generation moment a tree τ contains a single frontier node (for time step t it is n_{f_t}). An action `ApplyRule` expands the frontier node in depth-first, left-to-right traversal of the tree. A production rule r expands n_{f_t} by appending all child nodes specified by the selected production. For example, in listing 1 in step 10 the rule for the node `Call` extends this node with three new nodes: `expr{func}`, `expr*{args}`, `keyword*{keywords}`.

When n_{f_t} is a terminal node, which can not be expanded further, the next action must be `GenToken`.

Unary closures. Sometimes, generating an AST requires applying a chain of unary productions. For example, in listing 1 in steps 4-6 it takes three time steps to generate target for `Assign` statement:

```
ApplyRule[expr* => (expr)]
    ApplyRule[expr => (Name)]
        ApplyRule[Name -> (str{id})]
```

Such a formal redundancy allows to have a smaller production rule grammar but would increase a sequence length. Thus, they can be replaced with one action by taking the closure of the chain of unary productions:

```
ApplyRule[expr* => (str{id})]
```

Model was tested both with and without the unary closures.

4.2.2 GenToken actions

If a tree has reached a leaf and n_{f_t} is a terminal node, the `GenToken` actions is used to fill this node. Each token generation ends with a special end-of-string token `"<eos>"`. This way complex tokens like the a function name `sortBySecondIndex` can be split on parts `['sort', 'By', 'Second', 'Index']`, thus reduce the token vocabulary and allow complex rare tokens to be constructed from their constituents. After the end-of-string token generation model proceeds to the next frontier node.

The vocabulary of predefined token values can be inferred from a dataset. However, it is clear that this vocabulary will not cover all possible tokens for any environment. To cope with this problem, values can be copied directly from the input sequence. Therefore, it allows the model to use literals and names from a code description.

4.3 Action probabilities

Probabilities in eq. (4.3) are estimated by a neural attentional encoder-decoder model. Both encoder and decoder informational flow is structured by syntactic trees.

4.3.1 Encoder

The main architectural novelty in this work is a Tree-LSTM encoder. Details about its input structures and implementation described below.

Input NL description x consist of two parts. The first is a sequence of length n of word vectors $\{w^{(t)}\}_{t=1}^{n}$. The second is a syntax tree which consists of m nodes $\{\eta^{(t)}\}_{t=1}^{m}$, where $m \geq n$. Details about the description tree parsing can be found in section 5.1.1. For all $t \leq n$, each tree node $\eta^{(t)}$ has a corresponding input vector from a word sequences $w^{(t)}$. For $t > n$, an input vector for $\eta^{(t)}$ is padding-vector. Each tree node $\eta^{(t)}$ has a set of children nodes $ch(\eta^{(t)}) = \{\eta_i\}_{i=1}^{k}$. Therefore a single input element $x^{(t)}$ can be defined as a tuple $(\eta^{(t)}, w^{(t)}, ch(\eta^{(t)}))$:

$$x^{(t)} = \begin{cases} (\eta^{(t)}, w^{(t)}, ch(\eta^{(t)})) & \text{if } t \leq n \\ (\eta^{(t)}, w_{pad}, ch(\eta^{(t)})) & \text{if } t > n \end{cases} \tag{4.4}$$

To encode this structures we used a Tree-LSTM from Tai, Socher, and Manning, 2015[1]. Similary to a SRvN, described in section 3.6, this model starts from tree leaves, and recursively computes a node embedding $h^{(t)}$ for each $x^{(t)}$ using values of a memory cell $\{c_i\}_{i=1}^{k} = memory(ch(\eta^{(t)}))$ and previous embeddings $\{h_i\}_{i=1}^{k} = hidden(ch(\eta^{(t)}))$ from children nodes:

[1]We used pytorch implementation from https://github.com/dasguptar/treelstm.pytorch

$$
\begin{aligned}
\hat{h} &= \sum_{i=1}^{k} h_i \\
\mathrm{i}^{(t)} &= \sigma(W_i \cdot [\hat{h}, w^{(t)}] + b_i) \\
\mathrm{u}^{(t)} &= tanh(W_u \cdot [\hat{h}, w^{(t)}] + b_u) \\
\mathrm{f}_i^{(t)} &= \sigma(W_f \cdot [h_i, w^{(t)}] + b_f) \\
\mathrm{c}^{(t)} &= i^{(t)} \circ u^{(t)} + \textstyle\sum_{i=1}^{k} f_i^{(t)} \circ c_i \\
\mathrm{o}^{(t)} &= \sigma(W_o \cdot [\hat{h}, w^{(t)}] + b_o) \\
\mathrm{h}^{(t)} &= o^{(t)} \circ tanh(c^{(t)})
\end{aligned}
\tag{4.5}
$$

4.3.2 Decoder

The decoder is a RNN which sequentially generates an AST model as defined in eq. (4.3). Each production action naturally grounds to a step in the decoder. This way, the sequence of production rules from listing 1 can be interpreted as unrolling RNN time steps with some additional connections from parent action steps.

We used implementation of decoder from Yin and Neubig, 2017. It is a vanilla LSTM with additional connections which reflect the topological structure of the code syntax. For each decoding step, input vector is concatenation of a frontier node embedding $n^{(t)}$, a previous action embedding $a^{(t-1)}$ and a parent feeding $p^{(t)}$. The parent feeding is a concatenation of a decoder hidden state from parent step $h_{dp}^{(t)}$ and a parent rule embedding $r_p^{(t)}$. Consider as example steps 3-9 in listing 1:

```
ApplyRule[Assign => (expr*{targets}), (expr{value})]
    ...............
                GetToken["<eos>"]
    ApplyRule[expr => (Call)]
```

It has a frontier node `expr`, a previous action `GetToken["<eos>"]` and a parent rule `Assign => (expr*{targets}), (expr{value})`. Corresponding vectors for the node, rule and token stored as column vectors in matrices W_n, W_r, W_v.

Additionally, a decoder input contains attention and attention over history context vectors. The context vectors are calculated as described in section 3.5.1. Attention over history uses vectors from a previous decoder output:

$$\phi_h^{(t)} = H_d \cdot \alpha_h^{(t)} \tag{4.6}$$

Given all described above input values and a previous decoder embedding $h_d^{(t-1)}$, the next decoding step is calculated as follows:

$$h_d^{(t)} = f_{lstm}([a^{(t-1)}, n^{(t)}, p^{(t)}, \phi^{(t-1)}, \phi_h^{(t-1)}], h_d^{(t-1)}) \tag{4.7}$$

4.3.3 Calculating probabilities

In this section we explained how action probabilities from eq. (4.3) are calculated from a decoder embeddings $h^{(t)}$.

ApplyRule. The probability of applying a rule r as the current action $a^{(t)}$ is given by a softmax:

$$\begin{aligned} h_r^{(t)} &= tanh(W_{r1} \cdot h^{(t)} + b_{r1}) \\ P_r &= softmax(W_r \cdot h_r^{(t)} + b_r) \\ p(a^{(t)} = ApplyRule[r] | x, a^{(<t)}) &= P_r \cdot e(r) \end{aligned} \tag{4.8}$$

where $e(r)$ is an one-hot embedding for a rule r.

GenToken. As described in section 4.2.2, a token v can be generated from a predefined vocabulary or copied from the input. Therefore probability of an action to be $GenToken[v]$ is defined as a marginal probability:

$$\begin{gathered} p(a^{(t)} = GenToken[v] | x, a^{(<t)}) = p(gen | x, a^{(<t)}) p(v | gen, x, a^{(<t)}) + \\ p(copy | x, a^{(<t)}) p(v | copy, x, a^{(<t)}) \end{gathered} \tag{4.9}$$

$$p(gen|\cdot), p(copy|\cdot) = softmax(W_s \cdot h^{(t)} + b_s) \tag{4.10}$$

Probability of selection a token v from the vocabulary is calculated similarly to eq. (4.8). The difference is that the decoder embedding is concatenated with a context vector $\phi^{(t)}$:

$$\begin{aligned} h_v^{(t)} &= tanh(W_{v1} \cdot [h^{(t)}, \phi^{(t)}] + b_{v1}) \\ P_v &= softmax(W_v \cdot h_v^{(t)} + b_v) \\ p(v|gen, x, a^{(<t)}) &= P_v \cdot e(v) \end{aligned} \tag{4.11}$$

To model the copy probability we used the pointer network (Luong et al., 2015) described in the section 3.7. The same as in eq. (4.11), we used the decoder embedding concatenated with the context vector:

$$\begin{aligned} P_c &= f_{pointer}(H_e, [h^{(t)}, \phi^{(t)}]) \\ p(v|copy, x, a^{(<t)}) &= P_c \cdot e(v) \end{aligned} \tag{4.12}$$

Usage of the concatenated vector $[h^{(t)}, \phi^{(t)}]$ for a token generation probability calculation is reasonable since tokens are likely to occur in the input sequence, thus context vector might store some important information.

4.4 Training and Inference

Values for all weights and biases were inferred from training examples, as described in eq. (4.2). Each AST τ from a training set was transformed into a sequence of ground truth actions. The next decoder input is selected from ground truth sequence (the teacher forcing, described in section 3.5). The model is optimized by maximizing the log-likelihood of the ground truth actions sequence. Optimization was performed by the method of error back-propagation using ADAM (Kingma and Ba, 2014) algorithm. At inference time, given an NL description, we used beam search described in section 3.5.2 to approximate the best AST $\hat{y}$ in eq. (4.1).

Chapter 5

Experiments

5.1 Datasets

HearthStone (HS) dataset (Ling et al., 2016) is a collection of Python classes which implements cards from the card game HearthStone. Each card has a set of attributes which is concatenated to produce an input sequence. The dataset contains 665 Python classes with descriptions.

Django dataset (Oda et al., 2015) contains a corpus of lines of Python code with manually annotated pseudo-code from the Django web framework. Corpus contains 18,805 pairs of Python statements and corresponding English pseudo-codes.

Additional information about the datasets can be found in table 5.1.

5.1.1 Preprocessing

All input descriptions was tokenized using Stanford CoreNLP[1] Java package. Quoted text, which might be referred as values for string constants, was replaced with special markers (see table 5.2). Nested object references in queries, like `re.findall` was split by the period so the pointer network can copy each part separately (see table 5.3). For **HS** we also constructed synthetic descriptions, using structured parts of target class descriptions (see table 5.4). Then all descriptions were parsed to trees with three different approaches, described below.

[1]https://stanfordnlp.github.io/CoreNLP

Dataset	**HS**	**Django**
Train	533	16.000
Development	66	1.000
Test	66	1.805
Avg. tokens in description[‡]	47.3	13.7
Avg. nodes in constituency tree	93.5	26.3
Avg. nodes in CCG tree	109.4	28
Avg. characters in code	324.3	43.9
Avg. size of AST (# nodes)	66.4	9.5
Statistics of Grammar		
terminal vocabulary size	550	3466
w/o unary closures		
# productions[†]	100	222
# node types[†]	61	96
Avg. # of actions per example[†]	173.4	20.3
w/ unary closures		
# productions[†]	100	237
# node types[†]	57	92
Avg. # of actions per example[†]	141.7	16.4

TABLE 5.1: Statistics of datasets and associated grammars ([†]Previously reported by Yin and Neubig, 2017. [‡]Number of dependency tree nodes is equal to a number of tokens in the description.)

Input query:	while '<' is contained in value and '>' is contained in value,
Input query preprocessed:	while _STR_0_ is contained in value and _STR_1_ is contained in value ,
Target code:	while '<' in value and '>' in value:
Target code preprocessed:	while '_STR_0_' in value and '_STR_1_' in value:

TABLE 5.2: Quoted items preprocessing for item #2 from the development split of **Django**.

Input query:	from django.utils.six.moves import html_parser as _html_parse into default name space.
Input query preprocessed:	from django.utils.six.moves (django utils six moves) import html_parser as _html_parse into default name space .

TABLE 5.3: Nested bject references preprocessing for item #93 from the developer split of **Django**.

Structured input:	Deadly Poison NAME_END -1 ATK_END -1 DEF_END 1 COST_END -1 DUR_END Spell TYPE_END Rogue PLAYER_CLS_END NIL RACE_END Free RARITY_END Give your weapon +2 Attack.
Synthetic description:	Name: Deadly Poison, attack: -1, defence: -1, cost: 1, duration: -1, type: Spell, player class: Rogue, race: None, rarity: Free. Give your weapon +2 Attack.

TABLE 5.4: Synthetic description for the item #3 from the developer split of **HS**.

To create CFG sentence representation we used `LexicalizedParser` (Klein and Manning, 2003) from the CoreNLP. Dependency parsing was done by `DependencyParser` (Chen and Manning, 2014) from the CoreNLP. For CCG parsing we used package EasyCCG[2] (Lewis and Steedman, 2014).[3]

5.2 Implementation details

Dynamic computational graph. Model of Yin and Neubig, 2017 was build on the framework Theano[4]. But Theano is not able to build a dynamic computational graph to encode syntactic trees. Therefore, we have implemented our model on PyTorch[5].

Mini-batch training. The nature of a recursive tree encoding does not allows to process data in batches, since each query defines a unique computational graph. But other components of the model was able to perform batch operations on data. Therefore we used a wrapper module for encoder, which was splitting input batches on single queries, processed them with Tree-LSTM module sequentially and combined back into batch. We used batches of size 10 for **HS** and 50 for **Django**.

Model parameters. The sizes of nodes, rules and terminal embeddings were 256. Except for the word embeddings, for which it was 300. We used pre-trained Common Crawl GloVe vectors (Pennington, Socher, and Manning, 2014) for

[2]http://homepages.inf.ed.ac.uk/s1049478/easyccg.html
[3]We were not able to include figures with parsed trees examples in this thesis due to their large size, but you can find them at our GitHub repo
[4]http://deeplearning.net/software/theano/
[5]http://pytorch.org/

the word embeddings weights. We were not freezing this weights, so the pre-trained values could be additionally adjusted during the training. The dimensions of the encoder and decoder hidden states and memory cells were 256. Hidden states of the attention and the pointer networks were of size 50. Also, we used the last state of the encoder as the initial state of the decoder (thought vector). For decoding, we used the beam of size 10.

Regularization. Since our datasets were relatively small for such complex neural model, we added strong regularization using Variational Dropout suggested in work of Gal and Ghahramani, 2016. Similarly to the approach described in the work of Zimmermann, Tietz, and Grothmann, 2012 we were adding Gaussian noise with mean 0.0 and STD 0.1 to the initial states $h^{(0)}$ and $c^{(0)}$ of encoder. These methods introduced significant improvement in both training speed and evaluation scores.

5.3 Experimental setup

Evaluation metrics. For this experiment, we measured **accuracy** as a fraction of output code which fully matches target examples. Additionally, to measure the quality of examples without full match we used average token level **BLEU-4**, as suggested by Ling et al., 2016 and Yin and Neubig, 2017. However, BLEU and accuracy do not measure the actual correctness of a generated code. Therefore we defined **errors** metric as a fraction of output trees which we were not able to convert into a code.

Baseline. Along with the Tree-LSTM encoder, we build a bidirectional LSTM encoder, previously described in the work of Yin and Neubig, 2017. This was done to have a clear baseline for our tree encoding method.

5.4 Results

We compared our results with two approaches: (1) Latent Predictor Network (LPN) of Ling et al., 2016 and (2) Syntactic Neural Model of Yin and Neubig, 2017.

	HS			Django		
	ACC	BLEU	ERROR	ACC	BLEU	ERROR
Retrieval system:[†]	0.0	62.5	-	14.7	18.6	-
Phrasal statistical MT:[†]	0.0	34.1	-	31.5	47.6	-
Hierarchical statistical MT:[†]	0.0	43.2	-	9.5	35.9	-
NMT[‡]	1.5	60.4	-	45.1	63.4	-
Seq2Tree[‡]	1.5	53.4	-	28.9	44.6	-
Seq2Tree-UNK[‡]	13.6	62.8	-	39.4	58.2	-
LPN[†]	4.5	65.6	-	62.3	77.6	-
Syntactic Neural Model[‡]	**16.7**	**75.8**	-	**71.6**	**84.5**	-
w/ unary closures:						
Bi-directional LSTM encoder	9.1	71.6	0.6	68	82.5	0.8
Tree-LSTM encoder						
with dependency trees	4.5	66.7	3.0	32.9	55.5	1.1
with constituency trees	4.5	63.9	13.2	42.0	61.8	0.9
with CCG trees	3.0	66.1	5.1	48.7	68.7	1.5
w/o unary closures:						
Bi-directional LSTM encoder	16.2	74.2	0.3	71.0	84.5	0.4
Tree-LSTM encoder						
with dependency trees	6	71.5	4.0	32.0	55.0	0.9
with constituency trees	4.5	64.9	13.2	42.3	61.0	1.1
with CCG trees	3.0	65.3	11.0	49.4	69.6	1.8

TABLE 5.5: Evaluation results for both datasets. [†]Results previously reported by Ling et al., 2016. [‡]Results previously reported by Yin and Neubig, 2017.

As presented in table 5.5, the performance of the models with the Tree-LSTM encoder have results comparable with LPN. Yet no model was able to improve current state-of-the-art results of Syntactic Neural Model. Also, tree encoders have shown a much higher rate of errors than the BiLSTM encoder.

Results of the model with BiLSTM encoder match the previously reported results of Syntactic Neural Model. This makes us certain that our implementation is valid and difference in performance with previous results caused actually by our tree encoders.

The dependency trees encoding has shown better results for **HS**. Dependency trees do not contain phrasal nodes and therefore they are shorter. This can be a proper justification for the difference in results for **HS** which contains long and homogeneous descriptions. However, results for the **Django** have a contrary bias. The performance of dependency trees is much lower than the performance of CFG and CCG trees. It can be explained by a fact, that **Django** has shorter and more divergent queries and due to this properties its dependency trees have lower generalization ability. CCG and CFG trees have less divergent structures where each node have no more than two children. Therefore informational flow in such trees might have higher approximation abilities. Yet this is only an assumption and requires additional research.

5.5 Case studies

To provide a clear understanding of the model performance we decided to present and comment few code generation examples from both datasets.

As can be seen in listing 1, the model was able to translate the intention of the card to deal 3 damage on the line 17. However, it was not able to produce a long sequence of actions from the lines 28-31, required to create a list of all targets. Probably, this mistake is connected to the grammar model problem that we described in section 4.2.

Listing 1. Code generated for the item #5 from the test set of HS

```
1  # Query:
2  Name: Hellfire, attack: -1, defence: -1, cost: 4,
3  duration: -1, type: Spell, player class: Warlock,
race: None, rarity: Free.
Deal $3 damage to ALL characters.

# Reference code:
class Hellfire(SpellCard):

    def __init__(self):
        super().__init__('Hellfire', 4, CHARACTER_CLASS.WARLOCK,
            CARD_RARITY.FREE)

    def use(self, player, game):
        super().use(player, game)
        targets = copy.copy(game.other_player.minions)
        targets.extend(game.current_player.minions)
        targets.append(game.other_player.hero)
        targets.append(game.current_player.hero)
        for minion in targets:
            minion.damage(player.effective_spell_damage(3), self)

# Predicted code:
class Hellfire(SpellCard):

    def __init__(self):
        super().__init__('Hellfire', 4, CHARACTER_CLASS.WARLOCK,
            CARD_RARITY.RARE, target_func=hearthbreaker.targeting.
            find_spell_target)

    def use(self, player, game):
        super().use(player, game)
        self.target.damage(player.effective_spell_damage(3), self)

```

In listing 2, you can see that model was mo able to understand complex script for a Murloc behavior from line 5. While the model was able to learn the alignment between simple query properties and the target class, really complex instructions still is not comprehensible.

Listing 2. Code generated for the item #27 from the test set of HS

```
1  # Query:
2  Name: Siltfin Spiritwalker, attack: 2, defence: 5,
3  cost: 4, duration: -1, type: Minion,
4  player class: Shaman, race : Murloc, rarity: Epic.
5  Whenever another friendly Murloc dies, draw a card.
6  Overload: (1)
7
8  # Reference code:
9  class SiltfinSpiritwalker(MinionCard):
10
11     def __init__(self):
12         super().__init__('Siltfin Spiritwalker', 4, CHARACTER_CLASS.SHAMAN,
13             CARD_RARITY.EPIC, minion_type=MINION_TYPE.MURLOC, overload=1)
14
15     def create_minion(self, player):
16         return Minion(2, 5, effects=[Effect(MinionDied(IsType(MINION_TYPE.
17             MURLOC)), ActionTag(Draw(), PlayerSelector()))])
18
19 # Predicted code:
20 class SiltfinSpiritwalker(MinionCard):
21
22     def __init__(self):
23         super().__init__('Siltfin Spiritwalker', 4, CHARACTER_CLASS.SHAMAN,
24             CARD_RARITY.EPIC, battlecry=Battlecry(Give(Buff(ManaChange(-1))
25             ), CardSelector(condition=IsSpell())))
26
27     def create_minion(self, player):
28         return Minion(2, 2)
29
```

In listings 3 and 4 we presented a comparison of results from the models with different encoders. The result from the baseline model with BiLSTM encoder almost matched the target in both cases. However, a predicted code has not all elements from a reference code. This can be a result of inability of the grammar to produce the lists of arbitrary lengths, described in section 4.2. Still, the model with BiLSTM encoder produced the code with a much higher quality than the models with the tree encoders.

Listing 3. Code generated for the item #665 from the test set of Django

```
# Query:
call the method self._text_chars with 4 arguments: length,
truncate, text and truncate_len, return the result.

# Reference code:
return self._text_chars(length, truncate, text, truncate_len)

# Code, predicted with BiLSTM encoder:
return self._text_chars(length, truncate, text)

# Code, predicted with tree encoder and CCG trees:
return self.clear_checkbox_id(text, text)

# Code, predicted with tree encoder and dependency trees:
return self._text_chars(text, text)
```

Listing 4. Code generated for the item #924 from the test set of Django

```
# Query:
tt is a tuple with 9 elements: dt.year, dt.month,
dt.day, dt.hour, dt.minute, dt.second,
result of the method dt.weekday,

# Reference code:
tt = dt.year, dt.month, dt.day, dt.hour,
    dt.minute, dt.second, dt.weekday(), 0, 0

# Code, predicted with BiLSTM encoder:
tt = dt.year, dt.month, dt.day

```

```
# Code, predicted with tree encoder and dependency trees:
timetuple = dt.year, dt.timetuple, dt.timetuple

# Code, predicted with tree encoder and CFG trees:
date_data = dt.year, dt.year, dt.microsecond
```

Chapter 6

Conclusion

In this work, we have not reached a substantial performance improvement over the previously reported results. Conventional approach with BiLSTM encoder has shown better results with the lower error rate. What also has a significant value, it trains faster and it can be trained using batches. However, our study of the Tree2Tree models has an important value for the further exploration of machine translation and code generation. We believe that potential of Tree-LSTM is yet to be discovered in other applications and we want to continue our research in this field.

6.1 Contribution

In this work we made the following contribution:

- Implemented Tree2Tree model on PyTorch.[1]
- Evaluated performance of tree encoders in sequence-to-sequence model.
- Created online API for code generation. [2]

6.2 Points to improve

Tree encoder. The recursive encoder has not surpassed results of the conventional BiLSTM encoder. However, it still has a potential to explore. Embeddings for lexical categories can be added to an input along with the word embeddings.

[1] All code is available on GitHub.
[2] API is available here.

Additional layers (recursive or recurrent) can be added on the top of the first layer. Knowledge about a tree structure can be induced into the attention layer to make an attention coherent with a query hierarchy. But since the syntactic models require a substantial effort to maintain the query parsing and cannot be encoded in batches, we do not consider this as a priority for the code generation task.

Improve grammar model. As we mentioned in section 4.2, the grammar model that we have used in this project is not suitable for a code with a long sequences of expressions in its AST. This can be solved with another grammar model, which supports arbitrary number of children nodes for the list nodes. A special final action can be used to indicate the list end.

Code context encoding. Our model used as an input only code description. Obviously, other code around a code line can contain important information which can be used for generation. In this model, we used an unconstrained terminal vocabulary, what is the naive approach, since each code line has different names of variables and functions in its scope. Therefore, usage of a code context in the model can made significant improvement for the correctness of the result.

Datasets. Datasets used for this work has few important flaws. **HS** is homogeneous and small, therefore it can only be used for a model evaluation and experiments. **Django** actually does not contains NL descriptions since it has pseudo-codes generated from the underlying code. Therefore this model requires evaluation on more heterogeneous datasets (like created by Barone and Sennrich, 2017). And the initial goal of development of an IDE plugin capable to be a handy tool for any developer, requires dataset which was created with a special attention to the most frequent developer requests. Probably, StackOverflow can be used as a source of statistics for that.

Evaluation. As mentioned in section 5.3, BLEU is a metric specifically designed for a human languages translation evaluation, therefore it can not appropriately represent the performance of a language to code translation. Tools like unit testing or static code analysis can provide measurements more relevant to this domain field. This should be considered during a development of more appropriate language-to-code dataset.

Another applications. Developed in this project codebase can be easily reused. The idea of the structured decoding has a great potential in other problems like question answering or text generation. We are planning to continue our research of other problems on a base of this work.

Appendix A

Python 3.6 abstract syntax tree grammar

The following grammar is taken from official documentation of Python module ast. It contains the description of all AST rules. Each rule describes how specific node type expands to other nodes and what types of children it could contain. Modifier * denotes that there could be multiple child nodes of that type. Modifier ? denotes that this child node is optional. For example, `Assert(expr test, expr? msg)` mean that node `Assert` requires expression to test and optionally it could contain expression with a message.

```
-- ASDL's 7 builtin types are:
-- identifier, int, string, bytes, object, singleton, constant
--
-- singleton: None, True or False
-- constant can be None, whereas None means "no value" for object.

module Python
{
    mod = Module(stmt* body)
        | Interactive(stmt* body)
        | Expression(expr body)

        -- not really an actual node but useful in Jython's typesystem.
        | Suite(stmt* body)

    stmt = FunctionDef(identifier name, arguments args,
                       stmt* body, expr* decorator_list, expr? returns)
```

```
| AsyncFunctionDef(identifier name, arguments args,
                 stmt* body, expr* decorator_list, expr? returns)

| ClassDef(identifier name,
   expr* bases,
   keyword* keywords,
   stmt* body,
   expr* decorator_list)
| Return(expr? value)

| Delete(expr* targets)
| Assign(expr* targets, expr value)
| AugAssign(expr target, operator op, expr value)
-- 'simple' indicates that we annotate simple name without parens
| AnnAssign(expr target, expr annotation, expr? value, int simple)

-- use 'orelse' because else is a keyword in target languages
| For(expr target, expr iter, stmt* body, stmt* orelse)
| AsyncFor(expr target, expr iter, stmt* body, stmt* orelse)
| While(expr test, stmt* body, stmt* orelse)
| If(expr test, stmt* body, stmt* orelse)
| With(withitem* items, stmt* body)
| AsyncWith(withitem* items, stmt* body)

| Raise(expr? exc, expr? cause)
| Try(stmt* body, excepthandler* handlers, stmt* orelse, stmt* finalbody)
| Assert(expr test, expr? msg)

| Import(alias* names)
| ImportFrom(identifier? module, alias* names, int? level)

| Global(identifier* names)
| Nonlocal(identifier* names)
| Expr(expr value)
| Pass | Break | Continue

-- XXX Jython will be different
```

```
        -- col_offset is the byte offset in the utf8 string the parser uses
        attributes (int lineno, int col_offset)

        -- BoolOp() can use left & right?
  expr = BoolOp(boolop op, expr* values)
       | BinOp(expr left, operator op, expr right)
       | UnaryOp(unaryop op, expr operand)
       | Lambda(arguments args, expr body)
       | IfExp(expr test, expr body, expr orelse)
       | Dict(expr* keys, expr* values)
       | Set(expr* elts)
       | ListComp(expr elt, comprehension* generators)
       | SetComp(expr elt, comprehension* generators)
       | DictComp(expr key, expr value, comprehension* generators)
       | GeneratorExp(expr elt, comprehension* generators)
       -- the grammar constrains where yield expressions can occur
       | Await(expr value)
       | Yield(expr? value)
       | YieldFrom(expr value)
       -- need sequences for compare to distinguish between
       -- x < 4 < 3 and (x < 4) < 3
       | Compare(expr left, cmpop* ops, expr* comparators)
       | Call(expr func, expr* args, keyword* keywords)
       | Num(object n) -- a number as a PyObject.
       | Str(string s) -- need to specify raw, unicode, etc?
       | FormattedValue(expr value, int? conversion, expr? format_spec)
       | JoinedStr(expr* values)
       | Bytes(bytes s)
       | NameConstant(singleton value)
       | Ellipsis
       | Constant(constant value)

       -- the following expression can appear in assignment context
       | Attribute(expr value, identifier attr, expr_context ctx)
       | Subscript(expr value, slice slice, expr_context ctx)
       | Starred(expr value, expr_context ctx)
       | Name(identifier id, expr_context ctx)
```

```
         | List(expr* elts, expr_context ctx)
         | Tuple(expr* elts, expr_context ctx)

          -- col_offset is the byte offset in the utf8 string the parser uses
          attributes (int lineno, int col_offset)

    expr_context = Load | Store | Del | AugLoad | AugStore | Param

    slice = Slice(expr? lower, expr? upper, expr? step)
          | ExtSlice(slice* dims)
          | Index(expr value)

    boolop = And | Or

    operator = Add | Sub | Mult | MatMult | Div | Mod | Pow | LShift
                 | RShift | BitOr | BitXor | BitAnd | FloorDiv

    unaryop = Invert | Not | UAdd | USub

    cmpop = Eq | NotEq | Lt | LtE | Gt | GtE | Is | IsNot | In | NotIn

    comprehension = (expr target, expr iter, expr* ifs, int is_async)

    excepthandler = ExceptHandler(expr? type, identifier? name, stmt* body)
                    attributes (int lineno, int col_offset)

    arguments = (arg* args, arg? vararg, arg* kwonlyargs, expr* kw_defaults,
                 arg? kwarg, expr* defaults)

    arg = (identifier arg, expr? annotation)
           attributes (int lineno, int col_offset)

    -- keyword arguments supplied to call (NULL identifier for **kwargs)
    keyword = (identifier? arg, expr value)

    -- import name with optional 'as' alias.
    alias = (identifier name, identifier? asname)
    withitem = (expr context_expr, expr? optional_vars)
}
```

Bibliography

Allamanis, Miltos et al. (2015). “Bimodal Modelling of Source Code and Natural Language”. In: *Proceedings of the 32nd International Conference on Machine Learning*. Ed. by Francis Bach and David Blei. Vol. 37. Proceedings of Machine Learning Research. Lille, France: PMLR, pp. 2123–2132. URL: http://proceedings.mlr.press/v37/allamanis15.html.

Artzi, Yoav, Nicholas FitzGerald, and Luke S Zettlemoyer (2013). “Semantic Parsing with Combinatory Categorial Grammars.” In: *ACL (Tutorial Abstracts)* 3.

Artzi, Yoav and Luke Zettlemoyer (2013). “Weakly supervised learning of semantic parsers for mapping instructions to actions”. In: *Transactions of the Association for Computational Linguistics* 1, pp. 49–62.

Backus, John W et al. (1957). “The FORTRAN automatic coding system”. In: *Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability*. ACM, pp. 188–198.

Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio (2014). “Neural Machine Translation by Jointly Learning to Align and Translate”. In: arXiv: 1409.0473v7 [cs.CL].

Balzer, R. (1985). “A 15 Year Perspective on Automatic Programming”. In: *IEEE Transactions on Software Engineering* SE-11.11, pp. 1257–1268. DOI: 10.1109/tse.1985.231877.

Balzer, Robert, Noreen Goldman, and David Wile (1978). “Informality in program specifications”. In: *IEEE Transactions on Software Engineering* 2, pp. 94–103.

Banarescu, Laura et al. (2013). “Abstract meaning representation for sembanking”. In: *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*, pp. 178–186.

Barone, Antonio Valerio Miceli and Rico Sennrich (2017). “A parallel corpus of Python functions and documentation strings for automated code documentation and code generation”. In: arXiv: 1707.02275v1 [cs.CL].

Barstow, David R (1979). "An experiment in knowledge-based automatic programming". In: *Artificial Intelligence* 12.2, pp. 73–119.

Bengio, Y., A. Courville, and P. Vincent (2013). "Representation Learning: A Review and New Perspectives". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35.8, pp. 1798–1828. DOI: 10.1109/tpami.2013.50.

Bengio, Yoshua et al. (2003). "A neural probabilistic language model". In: *Journal of machine learning research* 3.Feb, pp. 1137–1155.

Berant, Jonathan et al. (2013). "Semantic Parsing on Freebase from Question-Answer Pairs." In: *EMNLP*. Vol. 2. 5, p. 6.

Bhoopchand, Avishkar et al. (2016). "Learning Python Code Suggestion with a Sparse Pointer Network". In: arXiv: 1611.08307v1 [cs.NE].

Brandt, Joel et al. (2009). "Two studies of opportunistic programming". In: *Proceedings of the 27th international conference on Human factors in computing systems - CHI 09*. ACM Press. DOI: 10.1145/1518701.1518944.

Brandt, Joel et al. (2010). "Example-centric programming". In: *Proceedings of the 28th international conference on Human factors in computing systems - CHI 10*. ACM Press. DOI: 10.1145/1753326.1753402.

Chen, Danqi and Christopher Manning (2014). "A fast and accurate dependency parser using neural networks". In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pp. 740–750.

Chen, Huadong et al. (2017). "Improved Neural Machine Translation with a Syntax-Aware Encoder and Decoder". In: arXiv: 1707.05436v1 [cs.CL].

Chen, Xinyun et al. (2016). "Latent Attention For If-Then Program Synthesis". In: arXiv: 1611.01867v1 [cs.CL].

Clark, Stephen and James R. Curran (2007). "Wide-Coverage Efficient Statistical Parsing with CCG and Log-Linear Models". In: *Computational Linguistics* 33.4, pp. 493–552. DOI: 10.1162/coli.2007.33.4.493.

Dong, Li and Mirella Lapata (2016). "Language to Logical Form with Neural Attention". In: arXiv: 1601.01280v2 [cs.CL].

Dreyfus, Hubert L (1994). "What computers still can't do". In: *Topics in Health Information Management* 15.1, p. 87.

Dyer, Chris et al. (2016). "Recurrent Neural Network Grammars". In: arXiv: 1602.07776v4 [cs.CL].

Gal, Yarin and Zoubin Ghahramani (2016). "A Theoretically Grounded Application of Dropout in Recurrent Neural Networks". In: *Advances in Neural Information Processing Systems 29*. Ed. by D. D. Lee et al. Curran Associates, Inc., pp. 1019–1027. URL: http://papers.nips.cc/paper/6241-

a-theoretically-grounded-application-of-dropout-in-recurrent-neural-networks.pdf.

Galenson, Joel et al. (2014). “CodeHint: dynamic and interactive synthesis of code snippets”. In: *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*. ACM Press. DOI: 10.1145/2568225.2568250.

Gers, F.A. and E. Schmidhuber (2001). “LSTM recurrent networks learn simple context-free and context-sensitive languages”. In: *IEEE Transactions on Neural Networks* 12.6, pp. 1333–1340. DOI: 10.1109/72.963769.

Goller, C. and A. Kuchler (1996). “Learning task-dependent distributed representations by backpropagation through structure”. In: *Proceedings of International Conference on Neural Networks (ICNN96)*. IEEE. DOI: 10.1109/icnn.1996.548916.

Graves, Alex, Santiago Fernández, and Jürgen Schmidhuber (2005). “Bidirectional LSTM networks for improved phoneme classification and recognition”. In: *Artificial Neural Networks: Formal Models and Their Applications–ICANN 2005*, pp. 753–753.

Green, Cordell (1969). *Application of theorem proving to problem solving*. Tech. rep. SRI INTERNATIONAL MENLO PARK CA ARTIFICIAL INTELLIGENCE CENTER.

– (1976). “The design of the PSI program synthesis system”. In: *Proceedings of the 2nd international conference on Software engineering*. IEEE Computer Society Press, pp. 4–18.

Green, Cordell et al. (1977). “A Summary of the PSI Program Synthesis System.” In: *IJCAI*. Vol. 5, pp. 380–381.

Gvero, Tihomir and Viktor Kuncak (2015). “Interactive Synthesis Using Free-form Queries”. In: *Proceedings of the 37th International Conference on Software Engineering - Volume 2*. ICSE ’15. Florence, Italy: IEEE Press, pp. 689–692. URL: http://dl.acm.org/citation.cfm?id=2819009.2819139.

Harnad, Stevan (1990). “The symbol grounding problem”. In: *Physica D: Nonlinear Phenomena* 42.1-3, pp. 335–346.

Haugeland, John (1989). *Artificial intelligence: The very idea*. MIT press.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). “Long short-term memory”. In: *Neural computation* 9.8, pp. 1735–1780.

Jean, Sébastien et al. (2014). “On Using Very Large Target Vocabulary for Neural Machine Translation”. In: arXiv: 1412.2007v2 [cs.CL].

Jha, Susmit et al. (2010). "Oracle-guided component-based program synthesis". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE 10*. ACM Press. DOI: 10.1145/1806799.1806833.

Jozefowicz, Rafal et al. (2016). "Exploring the Limits of Language Modeling". In: arXiv: 1602.02410v2 [cs.CL].

Kingma, Diederik P. and Jimmy Ba (2014). "Adam: A Method for Stochastic Optimization". In: *CoRR* abs/1412.6980. arXiv: 1412.6980. URL: http://arxiv.org/abs/1412.6980.

Klein, Dan and Christopher D Manning (2003). "Accurate unlexicalized parsing". In: *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics-Volume 1*. Association for Computational Linguistics, pp. 423–430.

Lee, R. C. T., R. J. Waldinger, and C. L. Chang (1974). "An improved program-synthesizing algorithm and its correctness". In: *Communications of the ACM* 17.4, pp. 211–217. DOI: 10.1145/360924.360967.

Lewis, Mike and Mark Steedman (2014). "A* CCG Parsing with a Supertag-factored Model." In: *EMNLP*, pp. 990–1000.

Ling, Wang et al. (2016). "Latent Predictor Networks for Code Generation". In: arXiv: 1603.06744v2 [cs.CL].

Little, Greg and Robert C Miller (2009). "Keyword programming in Java". In: *Automated Software Engineering* 16.1, p. 37.

Luong, Minh-Thang, Hieu Pham, and Christopher D. Manning (2015). "Effective Approaches to Attention-based Neural Machine Translation". In: arXiv: 1508.04025v5 [cs.CL].

Luong, Thang et al. (2015). "Addressing the Rare Word Problem in Neural Machine Translation". In: *CoRR* abs/1410.8206. arXiv: 1410.8206. URL: http://arxiv.org/abs/1410.8206.

McDermott, Drew (1987). "A critique of pure reason". In: *Computational intelligence* 3.1, pp. 151–160.

Miriyala, K. and M.T. Harandi (1991). "Automatic derivation of formal software specifications from informal descriptions". In: *IEEE Transactions on Software Engineering* 17.10, pp. 1126–1142. DOI: 10.1109/32.99198.

Neubig, Graham (2017). "Neural Machine Translation and Sequence-to-sequence Models: A Tutorial". In: *CoRR* abs/1703.01619. arXiv: 1703.01619. URL: http://arxiv.org/abs/1703.01619.

Oda, Yusuke et al. (2015). “Learning to Generate Pseudo-Code from Source Code Using Statistical Machine Translation (T)”. In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. DOI: `10.1109/ase.2015.36`.

Olah, C. (2015). “Understanding LSTM Networks”. URL: `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`.

Pennington, Jeffrey, Richard Socher, and Christopher Manning (2014). “Glove: Global vectors for word representation”. In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pp. 1532–1543.

Price, David et al. (2000). “NaturalJava”. In: *Proceedings of the 5th international conference on Intelligent user interfaces - IUI 00*. ACM Press. DOI: `10.1145/325737.325845`.

Quirk, Chris, Raymond J Mooney, and Michel Galley (2015). “Language to Code: Learning Semantic Parsers for If-This-Then-That Recipes.” In: *ACL (1)*, pp. 878–888.

Rabinovich, Maxim, Mitchell Stern, and Dan Klein (2017). “Abstract Syntax Networks for Code Generation and Semantic Parsing”. In: arXiv: `1704.07535v1 [cs.CL]`.

Raychev, Veselin, Martin Vechev, and Eran Yahav (2014). “Code completion with statistical language models”. In: *ACM SIGPLAN Notices* 49.6, pp. 419–428. DOI: `10.1145/2666356.2594321`.

Robillard, Pierre N. (1999). “The role of knowledge in software development”. In: *Communications of the ACM* 42.1, pp. 87–92. DOI: `10.1145/291469.291476`.

Schuster, Mike and Kuldip K Paliwal (1997). “Bidirectional recurrent neural networks”. In: *IEEE Transactions on Signal Processing* 45.11, pp. 2673–2681.

Socher, Richard et al. (2011). “Parsing natural scenes and natural language with recursive neural networks”. In: *Proceedings of the 28th international conference on machine learning (ICML-11)*, pp. 129–136.

Solar-Lezama, Armando et al. (2005). “Programming by sketching for bit-streaming programs”. In: *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation - PLDI 05*. ACM Press. DOI: `10.1145/1065010.1065045`.

Srivastava, Saurabh, Sumit Gulwani, and Jeffrey S. Foster (2010). “From program verification to program synthesis”. In: *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL 10*. ACM Press. DOI: `10.1145/1706299.1706337`.

Sundermeyer, Martin, Ralf Schlüter, and Hermann Ney (2012). “LSTM neural networks for language modeling”. In: *Thirteenth Annual Conference of the International Speech Communication Association*.

Sutskever, Ilya, Oriol Vinyals, and Quoc V Le (2014). “Sequence to Sequence Learning with Neural Networks”. In: *Advances in Neural Information Processing Systems 27*. Ed. by Z. Ghahramani et al. Curran Associates, Inc., pp. 3104–3112. URL: http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf.

Tai, Kai Sheng, Richard Socher, and Christopher D. Manning (2015). “Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks”. In: arXiv: 1503.00075v3 [cs.CL].

Treude, Christoph, Ohad Barzilay, and Margaret-Anne Storey (2011). “How do programmers ask and answer questions on the web?” In: *Proceeding of the 33rd international conference on Software engineering - ICSE 11*. ACM Press. DOI: 10.1145/1985793.1985907.

Vinyals, Oriol, Meire Fortunato, and Navdeep Jaitly (2015). “Pointer Networks”. In: *Advances in Neural Information Processing Systems 28*. Ed. by C. Cortes et al. Curran Associates, Inc., pp. 2692–2700. URL: http://papers.nips.cc/paper/5866-pointer-networks.pdf.

White, Halbert (1992). *Artificial neural networks: approximation and learning theory*. Blackwell Publishers, Inc.

Wu, Yonghui et al. (2016). “Google’s Neural Machine Translation System: Bridging the Gap between Human and Machine Translation”. In: arXiv: 1609.08144v2 [cs.CL].

Xie, Pengtao and Eric Xing (2017). “A Constituent-Centric Neural Architecture for Reading Comprehension”. In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vol. 1, pp. 1405–1414.

Yin, Pengcheng and Graham Neubig (2017). “A Syntactic Neural Model for General-Purpose Code Generation”. In: arXiv: 1704.01696v1 [cs.CL].

Zettlemoyer, Luke S. and Michael Collins (2012). “Learning to Map Sentences to Logical Form: Structured Classification with Probabilistic Categorial Grammars”. In: arXiv: 1207.1420v1 [cs.CL].

Zhong, Victor, Caiming Xiong, and Richard Socher (2017). “Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning”. In: arXiv: 1709.00103v4 [cs.CL].

Zhu, Chenxi et al. (2015). “A Re-ranking Model for Dependency Parser with Recursive Convolutional Neural Network”. In: arXiv: `1505.05667v1 [cs.CL]`.

Zimmermann, Hans-Georg, Christoph Tietz, and Ralph Grothmann (2012). “Forecasting with recurrent neural networks: 12 tricks”. In: *Neural Networks: Tricks of the Trade*. Springer, pp. 687–707.