

Unrestricted Natural Language Implementation in Programming

Deepanjali Satu¹, A.Avinash²

¹Student, B.Tech, Dept. of Computer Science and Engineering
Centurion University of Technology and Management, Gajapati, Odisha, INDIA

²Assistant Professor, B.Tech, Dept. of Computer Science and Engineering
Centurion University of Technology and Management, Gajapati, Odisha, INDIA

Abstract - We argue it is better to program in a natural language such as English, instead of a programming language like Java. A natural language interface for programming should result in greater readability, as well as making possible a more intuitive way of writing code. In contrast to previous controlled language systems, we allow unrestricted syntax, using wide-coverage syntactic and semantic methods to extract information from the user's instructions. We also look at how people actually give programming instructions in English, collecting and annotating a corpus of such statements. We identify differences between sentences in this corpus and in typical newspaper text, and the effect they have on how we process the natural language input. Finally, we demonstrate a prototype system, that is capable of translating some English instructions into executable code. This paper describes about the implementation of the unrestricted natural language in programming. This paper contain a collection of basic description of NLP, its interconnectivity with database code, dealing with simplicity of the code and parser etc.,

Key Words: Pseudo code, English code, technical code, parse phase, NLID, Semantic, etc

1.INTRODUCTION

Any language that human beings learn from their surroundings and apply to communicate with one another is called natural language. Natural languages are employed to articulate the knowledge, acquaintance and sensations and to communicate our responses to others. In essence the natural languages are the most powerful, proper and logical way of communication. A language is not simply a system of communication, but also a form of power.

Artificial languages are created by humans to communicate with their technologies. The term artificial language implies a language specially crafted by humans. Most of the artificial languages are developed to communicate with technologies like computers. All programming languages are artificial languages. Programming languages are developed by humans for expressing algorithms in computational way and

instructing the machines. Scientific study of languages is called linguistics. The detailed studies of languages from linguistics point of view signify that among all the communicational systems, natural languages are the most powerful, effective and precise way of communication, consequently it is viable and attainable to use natural languages in other computational areas. Programming is hard. It requires a number of specialised skills and knowledge of the syntax of the particular programming language being used. Programmers need to know a number of different languages, that can vary in control structures, syntax, and standard libraries. In order to reduce these difficulties, we would like to express the steps of the algorithm we are writing in a more natural manner, without being forced into a particular syntax. Ideally, we want a plain English description. We have built an initial prototype of such a system, taking unrestricted English as input, and outputting code in the Python programming language. Also, it is often easier to write an English sentence describing what is to be done, than to figure out the equivalent code. Many programmers write in a pseudocode style that is almost English before elaborating on the details of an algorithm. There are also many tasks that can easily be described using English sentences, but are much harder to express as code, such as negation and quantification. Another advantage is that code written in English will be much easier to read and understand than in a traditional programming language. These complications are a result of the computer's implementation, rather than the algorithm we are trying to describe. We would like to abstract away these issues, using information present in the English sentences to figure out the correct action to take.

2. Descriptive Natural Language Programming

When story tellers speak fairy tales, they first describe the fantasy world—its characters, places, and situations – and then relate how events unfold in this world. Programming, resembling storytelling, can likewise be distinguished into the complementary tasks of description and proceduralization. While the basics of NLP (Natural Language Processing) for NLP (Natural Language Programming) 321 Building procedures out of steps and loops, it would be fruitful to also contextualize procedural rendition by discussing the architecture of the descriptive

world that procedures animate. Among the various paradigms for computer programming– such as logical, declarative, procedural, functional, object-oriented, and agent-oriented – the object-oriented and agent-oriented formats most closely embody human story telling intuition. Consider the task of programming a MUD2 world by natural language description, and the sentence. A theory of programmatic semantics for descriptive natural language programming is presented in here, we overview its major features, and highlight some of the differences between descriptive and procedural rendition. These features are at the core of the Meta for natural language programming system that can render code following the descriptive paradigm, starting with a natural language text.

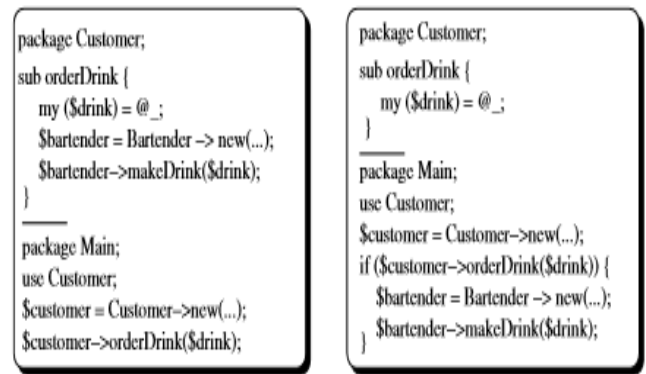
2.1 Syntactic Correspondences

There are numerous syntactic correspondences between natural language and descriptive structures. Most of today’s natural languages distinguish between various parts of speech that taggers such as Brill’s can parse – noun chunks are things, verbs are actions, adjectives are properties of things, adverbs are parameters of actions. Almost all natural languages are built a top the basic construction called independent clause, which at its heart has a who-does-what structure, or subject-verb-direct Object-indirect Object (SVO) construction. Although the ordering of subject, verb, and objects differ across verb-initial (VSO and VOS, e.g. Tagalog), verb-medial (SVO, e.g. Thai and English), and verb-final languages (SOV, e.g., Japanese), these basic three ingredients are rather invariant across languages, corresponding to an encoding of agent-method and method argument relationships. This kind of syntactic relationships can be easily recovered from the output of a syntactic parser, either supervised, if a Treebank is available, or unsupervised for those languages for which manually parsed data does not exist. Moreover, other ambiguity phenomena that are typically encountered in language, e.g. pronoun resolution, noun-modifier relationships, named entities, can be also tackled using current state-of-the-art natural language processing techniques, such as coreference tools, named entity annotators, and others. Starting with an SVO structure, we can derive agent-method and method-argument constructions that form the basis of descriptive programming. Particular attention needs to be paid to the ISA type of constructions that indicate inheritance. For instance, the statement Pacman is a character who ... indicates a super-class character for the more specific class Pacman. 2 A MUD (multi-user dungeon, dimension, or dialogue) is a multi-player computer game that combines elements of role-playing games, hack and slash style computer games, and social instant messaging chat rooms (definition from wikipedia.org). 322 R. Mihalcea, H. Liu, and H. Lieberman.

2.2 Scoping Descriptions

Scoping descriptions allow conditional if/then rules to be inferred from natural language. Conditional sentences are

explicit declarations of if/then rules, e.g. When the customer orders a drink, make it, or Pacman runs away if ghosts approach. Conditionals are also implied when uncertain voice is used, achieved through modals as in e.g. Pacman may eat ghosts, or adverbials like sometimes–although in the latter case the antecedent to their/then is under specified or omitted, as in Some times Pacman runs away.



```

package Customer;
sub orderDrink {
  my ($drink) = @_ ;
  $bartender = Bartender -> new(...);
  $bartender->makeDrink($drink);
}

package Main;
use Customer;
$customer = Customer->new(...);
$customer->orderDrink($drink);

```

```

package Customer;
sub orderDrink {
  my ($drink) = @_ ;
}

package Main;
use Customer;
$customer = Customer->new(...);
if ($customer->orderDrink($drink)) {
  $bartender = Bartender -> new(...);
  $bartender->makeDrink($drink);
}

```

Fig -1: The descriptive and procedural representations for the conditional statement When customer orders a drink, the bartender makes it.

An interesting interpretative choice must be made in the case of conditionals, as they can be rendered either descriptively as functional specifications, or procedurally as if/then constructions. For example, consider the utterance When customer orders a drink, the bartender makes it. It could be rendered descriptively as shown on the left of Figure , or it could be proceduralized as shown on the right of the same figure. Depending upon the surrounding discourse context of the utterance, or the desired representational orientation, one mode of rendering might be preferred over the other. For example, if the story teller is in a descriptive mood and the preceding utterance was there is a customer who orders drinks, then most likely the descriptive rendition is more appropriate.

2.3 Set-Based Dynamic Reference

Set-based dynamic reference suggests that one way to interpret the rich descriptive semantics of compound noun phrases is to map them into mathematical sets and set-based operations. For example, consider the compound noun phrase a random sweet drink from the menu. Here, the head noun drink is being successively modified by from the menu, sweet, and random. One strategy in unraveling the utterance’s programmatic implications is to view each modifier as a constraint filter over these to fall drink instances. Thus the object a Random Sweet Drink From The Menu implies a procedure that creates a set of all drink instances, filters for just those listed in the Menu, filters for those having the property sweet, and then applies a random choice to the remaining drinks to select a single one. Set-based dynamic reference lends great conciseness and power to NLP (Natural Language Processing) for NLP (Natural

Language Programming) 323 natural language descriptions, but a caveat is that world semantic knowledge is often needed to fully exploit their semantic potential. Still, without such additional knowledge, several descriptive facts can be inferred from just the surface semantics of a random sweet drink from the menu – there are things called drinks, there are things called menus, drinks can be contained by menus, drinks can have the property sweet, drinks can have the property random or be selected randomly. Later in this paper, we harness the power of set-based dynamic reference to discover implied repetition and loops.

Occam’s Razor would urge that code representation should be as simple as possible, and only complexified when necessary. In this spirit, we suggest that automatic programming systems should adopt the simplest code interpretation of a natural language description, and then complexify, or dynamically refactor, the code as necessary to accommodate further descriptions. For example, consider the following progression of descriptions and the simplest common denominator representation implied by all utterances up to that step.

- a) There is a bar. (atom)
- b) The bar contains two customers. (unimorphic list of type Customer)
- c) It also has a waiter. (unimorphic list of type Person)
- d) It has some tools. (polymorphic list)
- e) The bar opens and closes. (class/agent)
- f) The bar is a kind of store. (agent with inheritance)
- g) Some bars close at 6pm, others at 7pm. (forks into two subclasses)

Applying the semantic patterns of syntactic correspondence, representational equivalence, set-based dynamic reference, and scoping description to the interpretation of natural language description, object-oriented code skeletons can be produced. These description skeletons then serve as a code model which procedures can be built out of. Mixed-initiative dialog interaction between computer and storyteller can disambiguate difficult utterances, and the machine can also use dialog to help a storyteller describe particular objects or actions more thoroughly. The Metafor natural language programming system implementing the features highlighted in this section was evaluated in a user study, where 13 non-programmers and intermediate programmers estimated the usefulness of the system as a brainstorming tool. The non-programmers found that Metafor reduced their programming task time by 22%, while for intermediate programmers the figure was 11%. This result supports the initial intuition from and that natural language programming can be a useful tool, in particular for non-expert programmers.

3. Example

We can see in Table 1. two example programs that could be entered by a user. The code for the first program matches what is outputted by the current system, but the second is more complicated and does yet work correctly. Looking at

the these examples, we can see a number of difficulties that make the problem hard, as well as form some intuitions that can help to solve the task. For example, the first line of both programs involves three function calls because of variable typing.

ENGLISH	PYTHON
read in a number number =	int(sys.stdin.readline() .strip())
add 2 to the number print out the number	number += 2 print number
read in 2 numbers number1 =	int(sys.stdin.readline() .strip())
number2 =	int(sys.stdin.readline() .strip())
add them together print out the result	result = number1 + number2 print result

Table 1: Some example English sentences and their Python translations.

In Python, we must first read in a string, then strip away the newline character, and finally convert it to an integer. We can tell that integer conversion is required, firstly because of the name of the variable itself, and secondly, because a mathematical operation is applied to it later on. Of course, it is still ambiguous. The user may have expected the number to be a string, and to have the string 2 concatenated to what was read in. However, the code in Figure 1 is more likely to be correct, and if the user wants to use a string representation, then they could specify as much by saying: read in a number as a string. Another problem to deal with is the referencing of variables. In the first program, it is fairly easy to know that number is the same variable in all three sentences, but this is not as easy in the second. For the first sentence of the second program, the system needs to interpret 2 numbers correctly, and map it to multiple lines of code. Another complication is them, which references the previously mentioned variables. Finally, result, which does not appear in the second line, must still be part of the equivalent code, so that it can be used later. However, we do not want to restrict the vocabulary available to a user, or force them to construct sentences in a specific way, as is the case for existing restricted natural languages (Fuchs and Schwiter, 1996). Of course, this means that we must then deal with the inherent ambiguity and the great breadth of unrestricted natural English. For this reason, we employ wide-coverage syntactic and semantic processing, that is able to process this extensive range of inputs. In order to resolve ambiguities, we can apply the intuitions we have described above. We may not be sure that the number should be treated as an integer, but this is more likely than treating it as a string. This is the conclusion that our system should come to as well.

4. Background

Clearly, the task we are undertaking is not trivial. Though there are a number of related systems to the one we

propose, which have had success implementing a natural language interface for some task.

4.1. Natural Language Interfaces to Databases

The most popular task is a Natural Language Interface for a Database (NLIDB) (Androutsopoulos et al., 1995). This is because databases present a large amount of information, which both novice and expert users need to query. A specific query language such as SQL must be used, which requires one to understand the syntax for entering a query, and also the way to join the underlying tables to extract data that is needed. A NLIDB simplifies the task, by not requiring any knowledge of a specific query language, or of the underlying table structure of the database. We can see how this is similar to the English programming system that we are constructing. Both take a natural language as input, and map to some output that a computer can process. There are a number of problems that exist with NLIDBs. Firstly, it is not easy to understand all the ambiguity of natural language, and as such, a NLIDB can simply respond with the wrong answers. As a result of this, many NLIDBs only accept a restricted subset of natural language. For example, in the NLIDB PRE (Epstein, 1985), relative clauses must come directly after the noun phrases they are attached to. One feature of many NLIDBs, is the ability to engage the user in a dialogue, so that past events and previously mentioned objects can be referenced more easily. Two examples of this, anaphora and elliptical sentences, are shown in Figure 2. Understanding that it refers to the ship, and that the female manager's degrees are again the subject of the question, reduces the amount of effort required by the user, and makes the discourse more natural. We also intend to maintain a discourse between the user and the computer for our own system.

```

• ANAPHORA
> Is there a ship whose destination is unknown?
Yes.
> What is it?
What is [the ship whose
destination is unknown]?
Saratoga

• ELLIPTICAL SENTENCE
> Does the highest paid female manager have
any degrees from Harvard?
Yes, 1.
> How about MIT?
No, none.

```

Fig 2: An example of anaphora and an elliptical sentence
This would also allow us to resolve much of the ambiguity involved in natural language by asking the user which possibility they actually meant.

4.2. Early Systems

One of the first natural language interfaces is SHRDLU (Winograd, 1972), which allows users to interact with a number of objects in what was called Blocksworld. This system is capable of discriminating between objects, fulfilling goals, and answering questions entered by the user.

It also uses discourse in order to better interpret sentences from the user. There were also a handful of systems that attempted to build a system similar to what we describe in this paper (Heidorn, 1976; Biermann et al., 1983). Most of these used a restricted syntax, or defined a specific domain over which they could be used. Our system should have much greater coverage, and be able to interpret most instructions from the user in some way. More generally, we can look at a system that interprets natural language utterances about planetary bodies (Frost and Launchbury, 1989). This system processes queries about its knowledge base, but is restricted to sentences that are covered by its vocabulary and grammar. It deals with ambiguous questions by providing answers to each possible reading, even when those readings would be easily dismissed by humans. With our system, we will determine the most likely reading, and process the sentence accordingly.

4.3. Understanding Natural Language

One thing that we have not yet considered is how people would describe a task to be carried out, if they could use English to do so. The constructs and formalisms required by traditional programming languages do not apply when using a natural language. In fact, there are many differences between the way non-programmers describe a task, to the method that would be employed if one were using a typical programming language (Pane et al., 2001). Firstly, loops are hardly ever used explicitly, and instead, aggregate operations are applied to an entire list. These two methods for describing the same action are shown in Fig 4.

```

• AGGREGATE
sum up all the values in the list

• ITERATION
start the sum at 0
for each in value in the list
add this value to the sum

```

Fig 3: Finding the sum of the values in a list

Another point of difference comes in the way people use logical connectives such as AND and OR, which are not necessarily meant in the strictly logical way that is the case when using a programming language. There are also differences in the way that people describe conditions, remember the state of objects, and the way they reference those objects. There are actually many ways in which natural language constructions map onto programming concepts. These grammatic semantics (Liu and Lieberman, 2004) can be seen in syntactic types, where nouns map to objects or classes, verbs map to methods, and adjectives to attributes of the classes. Using these concepts could allow us to more easily understand an English sentence, and map it to a corresponding code output. Metafor (Liu and Lieberman, 2005) is a system that uses these ideas, taking a natural language description as input. As output, the system provides scaffolding code, that is, the outline for classes and methods, and only a small amount of actual content. The code is not immediately executable, but can help the programmer in getting started.

Natural Java (Price et al., 2000) is another natural language programming system that allows users to create and edit Java programs using English commands. Each sentence in the natural language input given to the system is mapped to one of 400 manually created case frames, which then extracts the triggering word and the arguments required for that frame. The frame can generate a change in the Abstract Syntax Tree (AST), an intermediate representation of the code, which is turned in Java code later. This system has a number of problems that we intend to improve on. Firstly, it can only handle one action per sentence. Our prototype can detect multiple verbs in a sentence, and generate code for each of them. Also, the AST representation Natural Java uses makes it hard to navigate around a large amount of code, since only simple movement operations are available. Another problem with Natural Java is that it maps to specific operations that are included in Java, rather than more general programming language concepts. This means that it is not adaptable to different programming languages. We intend to be more language-neutral. A user of our system should not need to look at the underlying code at all, just as a programmer writing in C does not need to look at the machine code.

5. English Code Corpus

In order to investigate the way that people would use English to describe a programming task, we licited responses from programmers, asking them to describe how they would solve sample tasks. These tasks included finding the smallest number in a list, splitting a string on a character and finding all primes less than 100. The respondents were all experienced programmers, since computer science staff were all that were easily available. As a result of this, they tended to impose typical programming constructs on what they wanted to do, rather than using a simpler English sentence. For example, one respondent wrote For each number in the list compare to min., when Compare each number in the list to the min. is more straightforward. This demonstrates quite well the way that programming languages force us to use a specific unnatural syntax, rather than the freer style that a natural language allows. The corpus is comprised of 370 sentences, from 12 different respondents. They range in style quite significantly, with some using typically procedural constructs such as loops and ifs (complete with the non-sensical English statement: end loop in some cases), while others used a more declarative style. We have semi-automatically tagged the entire corpus with CCG categories (called supertags). This process consisted of running the parser on the corpus, and then manually correcting each parse. Corrections were required in most sentences, as the way people express programming statements varies significantly from sentences found in newspaper text.

An example of this is in Fig5. This sentence uses an imperative construction, beginning with a verb, which is quite different from declarative sentences found in newspaper text, and the earlier example in Figure 4. We can

also notice that the final category for the sentence is S[b]\NP, rather than simply S. Another difference is in the vocabulary used for programming tasks, compared to Wall Street Journal (WSJ) text. We find if, loop, and variables in the former, and million, dollars, and executives in the latter.

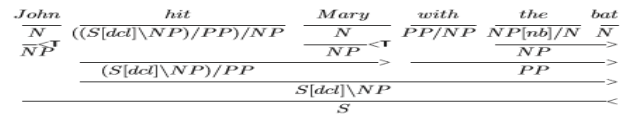


Figure 4: An example CCG derivation

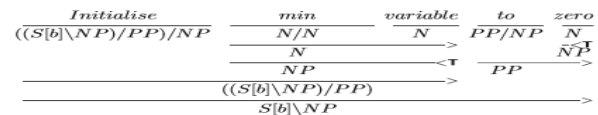


Fig 4: A CCG derivation for an English programming instruction.

Particular words can also have different grammatical functions. For example: print is usually a noun in the WSJ, but mostly a verb while programming.

6. Combinatory Categorical Grammar

Combinatory Categorical Grammar (CCG) is a typedriven, lexicalised theory of grammar (Steedman, 2000). Each word receives a syntactic category that defines its predicate-argument relationship with surrounding words. We can see a simple example of this in Fig 4. Each word is assigned a category that defines how it is involved with other words in the sentence. These relationships are carried out through a number of rules, such as forward and backward application, which can be seen in the example. Additional rules such as composition and conjunction also allow the formalism to easily capture long-range dependencies. This is particularly important for our system, as the constructions used to describe programming instructions often contain non-standard constituents such as extraction, relativization, and coordination. These possibilities result in a large number of interpretations, as a single word can be assigned a different category depending on how it is used, and the words that surround it. However, the application of statistical parsing techniques for CCG have shown that it is capable of performing wide coverage parsing at state-of-the-art levels (Clark and Curran, 2004).

7. System Architecture

The system architecture and its components are shown in Fig 5.



Fig 5: The system architecture

Firstly, the user will enter text that will be parsed by the CCG parser. We then translate the predicateargument structure

generated by the parser into a first-order logic representation of DRS predicates. This gives us a more generic representation of the sentence, rather than the specific wording chosen by the user. The final step is to generate the code itself. Throughout these three phases, we also intend to use a dialogue system that will interact with the user in order to resolve ambiguity in their input. For example, if the probability with which the parser gives its output is too low, we may ask the user to confirm the main verb or noun. This is especially important, as we do not intend for the system to be foolproof, but we do intend that the user should be able to solve the problems that they encounter, either through greater specification or rephrasing. At this current stage though, we have only dealt with basic functionality. Also, as we progress through each stage, we will follow the example previously shown in Fig 5. We will see how the processing we do manages to begin with this English input, and eventually output working Python code.

8. Parse Phase

In The Complete Lojban Language, Cowan provides both YACC and EBNF grammars for defining legal Lojban utterances, however additional post-processing is required for properly handling many constructs, particularly optional terminators. Robin Lee Powell has done additional work to produce a more powerful Parsing Expression Grammar to represent Lojban. Because PEG allows for ordered choices, rather than the unordered choices of YACC and EBNF, Powell’s grammar can handle directly those cases which required post-processing in Cowan’s solutions. For the first analysis pass, the system utilizes a parser produced by Robert Grimm’s Rats! parser generator using Powell’s PEG grammar. Using the grammar, the parser verifies the statement is legal Lojban and annotates its syntactic structure. For example, the phrase “lo cribe poixekri cuklana” would be annotated as:

“The output of the parser is then used to build an in-memory n-tree of tokens for the next analysis phase.”

9. Semantics

From the syntactic representation of the sentence, we wish to build a more semantically abstracted version of what the user wants to translate into code. The advantage of this is that we can more readily extract the particular verbs and nouns that will become functions and their arguments respectively.

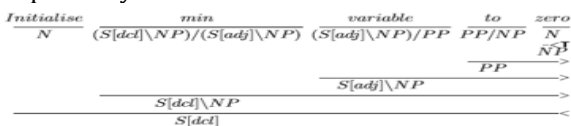


Fig 6: The original, incorrect CCG derivation.

TRAINING DATA	COVERAGE	POS WORD	SUPER WORD	POS LINE	SUPER LINE
Original model	95.6757%	0.873	0.736	0.359	0.197
1x code	93.2432%	0.975	0.848	0.829	0.510
5x code	85.9459%	0.994	0.931	0.962	0.708
10x code	82.4324%	0.996	0.962	0.975	0.821
20x code	82.7027%	0.998	0.978	0.986	0.889
20x code, 10-fold cross validation	85.40542%	0.974	0.896	0.840	0.624

Fig 7: Parse result

```

%%% Initialise 'min' variable to zero .
-----x2-----
x4 x3 x5 x1 x2
thing (x4)
'min' (x5)
nm (x5, x3)
variable (x3)
initialise (x1)
agent (x1, x4)
patient (x1, x3)
|x2|>0
to (x1, x2)
event (x1)
    
```

Fig 8: DRS for example sentence

Having a logical form also means we can apply inference tools, and thereby detect anomalies in the user’s descriptions, as well as including other sources of knowledge into the system. The ccg2sem system (Bos et al., 2004; Blackburn and Bos, 2005) performs this task, taking CCG parse trees as input, and outputting DRS logical predicates. A single unambiguous reading is always outputted for each sentence. The DRS for our example sentence is shown in Fig 8. We can see that the verb (x1) is identified by an event predicate, while the agent (x4) and patient (x3) are also found. One particular discriminating feature of the imperative sentences that we see, is that the agent has no representation in the sentence. We can also find the preposition (x2) attached to the verb, and this becomes an additional argument for the function. This logical form also extracts conditions that would be found in if statements and loops very well. Fig 9 shows the DRSs for the sentence: If num is -1, quit. We can see the proposition DRS (the middle box) and the proposition itself (x2), which entails another verb (x3) to be interpreted. That is, we should carry out the verb quit (x1), if the proposition is true. Almost all if statements in the corpus are identified in this way.

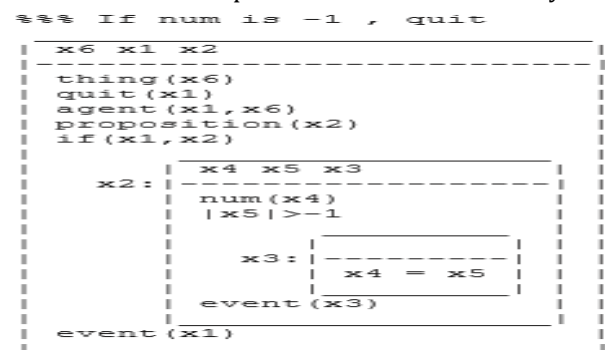


Fig 9: DRS for if statement

10. Generation

Having extracted the functional verb and its arguments, we then need to find a mapping onto an equivalent line of code. The simplest technique, which the current system uses, consists of a list of primitives, each of which describes the specific verb in question as well as a number of arguments. If the semantic information matches perfectly with a primitive, then the equivalent code is generated. At present, there exist

only a few primitives, shown in Figure 11. This system obviously has a number of weaknesses. Firstly, that if the user chooses a verb that is not listed in a primitive, then no code can be generated. Also, some primitives would be described with the same verb and arguments, but require different code, such as adding two numbers together, compared to adding one number to a list. This is similar to operator overloading, a feature present in a number of programming languages such as C++. We can also constrain the number of possibilities by using intuitive notions, such as not being able to output a previously unseen variable. Also, we can take advantage of the limited domain of programming. Rather than trying to list every sense of every verb in the English language together with its equivalent programming concept, we could create a much smaller set of programming primitives, and simply map everything onto one of those. Considering the small number of choices and the constraints mentioned above, this may be possible using a machine learning approach. Of course, we must consider what to do when a function that is not one of the primitives is referred to. In such a case, and assuming it can be detected, we believe the most sensible thing to do is to ask the user to describe how to carry out such a function, using the more basic primitive functions that already exist. Thus we would allow the creation of user-defined functions, just as a normal programming language would.

FUNCTIONAL VERB	ARGUMENTS	CODE TEMPLATE
read	input	<input> = int(sys.stdin.readline())
print	output	print <output>
add	addAmount, addTo	<addTo> += <addAmount>
initialise	variable, setting	<variable> = <setting>
set	variable, setting	<variable> = <setting>
assign	variable, setting	<variable> = <setting>
iterate	item, list	for <item> in <list>:

Fig 10: Primitives used for generation

Looking back to our example sentence once more, we proceed to extract the predicate (initialise) and argument information (min variable, 0) from the DRS. This maps to the initialise primitive in Figure 11. The matching code, stored in the primitive, then comes out as: `min.variable = 0`. This is clearly a suitable outcome, and we can say that for this case, the system has worked perfectly.

10. CONCLUSIONS

Programming is a very complicated task, and any way in which it can be simplified will be of great benefit. For this, a user may be asked to clarify or rephrase a number of points, but will not have to correct syntax errors as when using a normal programming language. Using modern parsing techniques, and a better understanding of just how programmers would write English code, we have built a prototype that is capable of translating natural language input to working code. More complicated sentences that describe typical programming structures, such as if statements and loops, are also understood. Here we have mentioned about the NLP and unrestricted NLP. This paper will help in programming all the things that are required in there, i.e., dealing with the areas all data and information are stored in the database. By using this the user can easily

access the program, they does not need to remember the whole technical code or process to write the program like in java. They just need to write the tags in English code to execute the program and if error arises then autocompiled and corrected.

REFERENCES

- [1] Muhammad Shumail Naveed, Muhammad Sarim, Kamran Ahsan Department of Computer Science, Federal Urdu University of Arts, Science & Technology, Karachi, Pakistan.
- [2] J. Bos, S. Clark, M. Steedman, J.R. Curran, and J. Hockenmaier. 2004. Wide-coverage semantic representations from a CCG parser. In Proceedings of the 20th International Conference on Computational Linguistics (COLING '04), Geneva, Switzerland.
- [3] S. Clark and J.R. Curran. 2004. Parsing the WSJ using CCG and log-linear models. In Proceedings of the 42nd Meeting of the ACL, Barcelona, Spain.
- [4] S.S. Epstein. 1985. Transportable natural language processing through simplicity – the PRE system. ACM Transactions on Office Information Systems, 3:107-120.
- [5] R. Frost and J. Launchbury. 1989. Constructing natural language interpreters in a lazy functional language. The Computer Journal. Special issue on lazy functional programming, 32(2):108-121, April.
- [6] N. E. Fuchs and R. Schwitter. 1996. Attempto controlled English (ACE). In Proceedings of the First International Workshop on Controlled Language Applications, pages 124-136. G. E. Heidorn. 1976. Automatic programming through natural language dialogue: A survey. IBM Journal of Research and Development, 20(4):302-313, July.

BIOGRAPHIES



Student of Centurion University of Technology and Management. Studing B.tech 3rd year in Dept. of Computer Science and Engineering (CSE).

E-mail Id:
140101csr035@cutm.ac.in



Assistant Professor at Centurion University of Technology and Management. M.Tech from GITAM UNIVERSITY. Research area in Image Mining. B.TECH from JNTUH.

E-mail Id: a.avinash@cutm.ac.in