ABSTRACT

Natural Language Processing holds great promise for making computer interfaces that are easier to use for people, since people will (hopefully) be able to talk to the computer in their own language, rather than learn a specialized language of computer commands. For programming, however, the necessity of a formal programming language for communicating with a computer has always been taken for granted. We would like to challenge this assumption. We believe that modern Natural Language Processing techniques can make possible the use of natural language to (at least partially) express programming ideas, thus drastically increasing the accessibility of programming to non-expert users. To demonstrate the feasibility of Natural Language Programming, this paper tackles what are perceived to be some of the hardest cases: steps and loops. We look at a corpus of English descriptions used as programming assignments, and develop some techniques for mapping linguistic constructs onto program structures, which we refer to as programmatic semantics.

KEYWORDS : Natural language, programming, loop finder, procedural.

1. INTRODUCTION

Natural Language Processing and Programming Languages are both established areas in the field of Computer Science, each of them with a long research tradition. Although they are both centered around a common theme – “languages” – over the years, there has been only little interaction (if any) between them [1]. This paper tries to address this gap by proposing a system that attempts to convert natural language text into computer programs.
2. MATERIALS AND METHODS

2.1. Background

Early work in natural language programming was rather ambitious, targeting the generation of complete computer programs that would compile and run. For instance, the “NLC” prototype [1] aimed at creating a natural language interface for processing data stored in arrays and matrices, with the ability of handling low level operations such as the transformation of numbers into type declarations as e.g. float-constant(2.0), or turning natural language statements like add y1 to y2 into the programmatic expression y1 + y2. These first attempts triggered the criticism of the community [3], and eventually discouraged subsequent research on this topic.

3. DESCRIPTIVE NATURAL LANGUAGE PROGRAMMING

Programming, resembling storytelling, can likewise be distinguished into the complementary tasks of description and proceduralization.

3.1. 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 tags such as Brill’s [2] 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 atop the basic construction called independent clause, which at its heart has a who-does-what structure, or subject-verb-directObject-indirectObject(SVO) construction.

3.2. 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 the if/then is underspecified or omitted, as in Sometimes Pacman runs away.

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

The descriptive and procedural representation for the conditional statement When customer orders a drink, the bartender makes it

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

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

Fig 2: Scoping Descriptions

3.3. 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.

4. Procedural Natural Language Programming

In procedural programming, a computer program is typically composed of sequences of action statements that indicate the operations to be performed on various data structures. Correspondingly, procedural natural language programming is targeting the generation of computer programs following the procedural paradigm, starting with a natural language text.
Write a program to generate 1000 numbers between 0 and 99 inclusive. You should count how many times each number is generated and write these counts out to the screen.

```perl
@counts;
for($i = 0; $i < 10000; $i++) {
    &generateRandomNumber ($number);
    &count($number);
    $i = 0;
    foreach $count (@counts) {
        &writeCount($i++, $count);
    }
    sub generateRandomNumber {
        ($ref) = @_; 
    }
    sub count {
        ($number) = @_; 
    }
    sub writeCount {
        ($index, $count) = @_; 
    }
    $$ref = 1 + rand(99);
    $counts[$number]++;
    print $index, "", $count,"n";
```

Next, steps are identified as statements containing one verb in the active voice. We are therefore identifying all verbs that could be potentially turned into program functions, such as e.g. read, write, count.

Finally, the object of each action is identified, consisting of the direct object of the active voice verb previously found, if such a direct object exists.

The output of the step finder process is therefore a series of natural language statements that are likely to correspond to programming statements, each of them with their corresponding action that can be turned into a program function (as represented by the active voice verb), and the corresponding action object that can be turned into a function.

### 4.2. The Loop Finder

An important property of any program statement is the number of times the statement should be executed.

The role of the loop finder component is to identify such natural language structures that indicate repetitive statements. The input to this process consists of steps, fed one at a time, from the series of steps identified by the step finder process, together with their corresponding actions and parameters. The output is an indication of whether the current action should be repeated or not, together with information about the loop variable and/or the number of times the action should be repeated.

### 4.3. Comment Identification

The comment identification step has the role of identifying those statements in the input natural language text that have a descriptive role, i.e. they provide additional specifications on the statements that will be executed by the program.

### 4.4. A Walk – Through Example

The generation of a computer program skeleton follows the three main steps highlighted earlier: step identification, comment identification, loop finder.

First, the step finder identifies the main steps that could be potentially turned into programming statements.
Next, the comment finder does not identify any descriptive statements for this input text, and thus none of the steps found by the step finder are marked as comments. By default, all the steps are listed in the output program in a comment section.

Finally, the loop finder inspects the steps and tries to identify the presence of repetition.

For the evaluation, we randomly selected a subset of 25 programming assignments from the set of Web-mined

5. CONCLUSION

In this paper, we showed how current state-of-the-art techniques in natural language processing can allow us to devise a system for natural language programming that addresses both the descriptive and procedural programming paradigms. The output of the system consists of automatically generated program skeletons, which were shown to help non-expert programmers in their task of describing algorithms in a programmatic way. As it turns out, advances in natural language processing helped the task of natural language programming.

But we believe that natural language processing could also benefit from natural language programming. The process of deriving computer programs starting with a natural language text implies a plethora of sophisticated language processing tools—such as syntactic parsers, clause detectors, argument structure identifiers, semantic analyzers, methods for co-reference resolution, and so forth—which can be effectively put to work and evaluated within the framework of natural language programming. We thus see natural language programming as a potential large scale end-user (or rather, end computer) application of text processing tools, which puts forward challenges for the natural language processing community and could eventually trigger advances in this field.

REFERENCES

Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05) (Pittsburgh, 2005).


