Compilers

PROJECT REPORT: AN INTERPRETER FOR AUGMENTED TRANSITION NETWORKS

May 31, 2015

Daniel Otero and Guido Arnau

Universitat Politècnica de Catalunya
Facultat d'Informàtica de Barcelona

Contents

1 Introduction

This project consisted on developing of an interpreter for ATNs (Augmented Transition Networks). In this section, we will explain what an ATN is and proceed to explain the project goals.

1.1 Augmented Transition Networks

Augmented transition networks are a type of parsers that are able to recognize regular languages simulating indeterministic finite state machines. ATNs have been augmented with the capacity to call recursively other ATNs, including themselves.

The parse proceeds following a depth-first search of the ATN, it succeeds if a final node is reached. During the execution, the ATN also executes custom code in case they follow an arc.

ATN parsers have the power of a Turing machine since they have memory and can decide which action perform in every node depending on the input read and the data that they have generated. Taking into account the ATN features, they can handle any type of grammar which could possibly be parsed by any machine.

1.2 The ATN Interpreter

In this project it has been developed an ATN Interpreter called ATNLang. In other words, a program has been created in order to parse ATN's described with a custom language and execute them automatically.

The ATN interpreter has a built-in natural language analyser (the open source program FreeLing, a language analysis tool suite [1]) in order to pre-analyse the characteristics of the sentences fed into the program. The ATN interpreter receives from the Freeling module, for each word, a set of values. This set typically will include: the scanned word, the base word from which it derives from and a code describing the characteristics of it. FreeLing offers a wide array of natural language analysis features, like text tokenization and morphological analysis of sentences, which will be used in the parsing of the sentences.

Since a fully functioning language has been developed, the interpreter can also execute normal programs that do not contain ATN definitions.

The language developed parts form the ASL language, which has been extended to include the automatic execution of ATNs. On the following sections it will be precisely stated which features have been added into the language.

1.3 Usage

A small script has been developed in order to execute the FreeLing module in conjunction with the ATN interpreter. An example of usage would be the following:

```
$ ./run-atn.sh test/test-numbers.atn es "Ciento cuarenta y cinco"
```

This command would execute the program `test-numbers.atn` with the sentence *"Ciento cuarenta y cinco"* as input (in Spanish).

If we want to debug our program, we must execute it without the help of the script. The binary of our interpreter is launched with the following command, form the root of the project:

```
$ bin/ATN test/test-numbers.atn -parse freeling.dat -ast ast -trace trace
```

The `*.atn` file indicates the source code of the program. The `freeling.dat` file contains the Freeling output of the sentence we want to parse. Finally, the `ast` and `trace` files will contain the AST Tree of the source code and the execution trace of the program, respectively. The last two flags can be used to debug the program. More details about this parameters can be found in 3.2.

Other flags and options are available in order to obtain information about the program:

```
usage: ATN [options] file
 -ast <file>     write the AST
 -dot            dump the AST in dot format
 -help           print this message
 -noexec         do not execute the program
 -parse <file>   file to read the FreeLing generated output
 -trace <file>   write a trace of function calls during the execution of the program
```

2 Language syntax and semantic

The language syntax is designed as simple and intuitive as possible. Conceptually the language has two distinct aspects: ATN programming and general programming. The language used by the interpreter is a non-specific purpose language which has been extended in order to include the syntax to describe ATNs. Thus, code without ATNs descriptions could be run by the interpreter.

The main feature of the language is to be able to execute automatically the ATNs described in the code, with the user needing to only indicate the text to be parsed by the ATN, although the language can be used like any interpreted language as the developer wishes.

2.1 The ATN description language

The ATN description syntax will be centred in creating nodes and describing their arcs. The syntax follows always this scheme:

```
atn Atn_name {

    atn_var = ...

    node Node_name {
        arc (boolean_expression) goto Node_name2 {
            instruction_1;
            instruction_2;
            instruction_3;
            ...
        }
        arc (...) goto Node_name1 {...}
        ...
    }

    node Node_name2 {...}
    ...
}
```

The structure used to describe an ATN is the following one: the keyword `atn` followed by the ATN's name, a list of nodes declared with the keyword `node` followed by the node's name and finally a list of the node's arcs, declared with the keyword `arc`, followed by the boolean condition that must be satisfied in order to follow that, the node to which the arc points to (expressed with the keyword `goto`) and the instructions to be performed in case the arc is followed.

The first node declared in the ATN method will be considered the initial node in case the ATN is called.

The boolean condition (defined inside **parenthesis**) of the arcs allow the use of the instructions described in section 2.3.1 or any other sort of boolean. Note that a call to an ATN will return a boolean, so calling other ATNs is possible, although that will consume input tokens.

The code `atn`atn_name will execute the indicated ATN with the text loaded from the file indicated with the `-parse` flag when the program was called.

The actions -the code inside the arc's method- will be able to execute any of the described instructions in section 2.2, except for the `return` keyword, since ATNs only accept or reject an input. However, writing to global or local variables is still supported in order to pass parameters or accumulate values between ATN and functions calls.

The actions and boolean conditions tested to follow an arc can also call to regular functions, as explained in section 2.3.

The order in which the arc's tests will be checked is from top to bottom. That means the arc condition appearing first will be the one that is checked first; in case it fails, the second one will be checked and so on. If all the arc condition fail, the parsing of the text with this ATN will fail and, thus, the ATN call will return a false boolean.

Another important feature is that inside an ATN it is possible to declare variables whose scope will be restricted to the ATN scope.

2.1.1 The `accept` keyword

In order to label a node as an accepting one, we will use the following syntax:

```
atn Atn_name {

   ...
    node node_name accept;
    ...
}
```

This will cause the ATN to accept the input consumed so far and return a true boolean to the instruction that called it.

2.1.2 The variable

The ATN iterates through the words of the sentence that it is parsing. In order to access the current word, an iterator is needed. Since the value will be accessed frequently, a simple symbol will be used as a placeholder, in which case # was chosen.

This iterator advances automatically after following an arc of an ATN. Each # position is associated with the node currently being executed. Thus, if # is accessed in the execution of an arc's code, it will have the value corresponding to that node.

As described in section 1.2, the FreeLing module will return a set containing a series of values: the current word, the base word from which the current word derives, a code with information about the word and more depending on the indicated analysis done by the module. Thus, the symbol # can be seen as an array that stores the values of this fields. To access, for example, the string of the word being parsed, we can write `#.0`, which will access the first position of the array. Having that in mind, the programmer could modify the Freeling module to output bigger tuples or even write a custom input where the tuple size could be bigger than the standard one. The access to those extra positions would be possible by just placing the index position after #.

If an index bigger than the amount of values stored in the # array, an *Index out of bounds* exception will be raised.

In the moment that the whole input sentence is consumed and thus, the ATN runs out of words to parse, the call will also fail.

can be accessed at any point of the execution of the program, including inside non-ATN code or functions called by ATNs. The value of # in this occasions, will be the position corresponding to the last node executed. For example, if there hasn't been any call to an ATN, the value of # will be the first word of the input sentence.

2.1.3 Variable scoping in ATNs

Each ATN has multiple variable scopes in order to allow more expressivity for the language. General variable scoping of the language will be explained in more detail at section 2.2.2. The ATN's possible scopes are the following:

1. Global scope: ATNs can access and modify global variables of the program.

2. Local ATN variables: variables that can be accessed and modified by all of the arc's code of the ATN and defined outside the arc inside the ATN.

3. Local arc variables: variables created and only accessed inside an arc's code.

2.1.4 ATN features: Backtracking and recursion

By definition ATN have implicit backtracking. Backtracking happens when a one of the multiple paths produced by an input is followed and the ATN fails. Before stating the ATN parse as a fail, the ATN would try to undo the arc decision choices trying all the possible paths of execution that it can perform given the specific input. Every arc over which the ATN backtracks causes the parse index to retrocede one position over the input and recover the execution frame -values of all the variables- present when the decision occurred. More details about the algorithm can be found on section 3.1.

Recursion is also implicit on the definition of ATNs. An ATN can call another ATN in two scenarios: in an arc evaluation or in an arc instruction. In both cases the call will be performed and the input will be parsed by the called ATN from the actual position of the parse index. When the called ATN finishes it's parsing the callee will continue its execution with the parse index placed at the position the called ATN left it. This operations have to be programmed with fully understanding of the process behind because the input index **always** advances with the parsing.

2.2 Non-specific purpose language

The interpreter also implements a basic imperative, strongly and dynamically typed language. This language will be used in the arc's tests, actions and any other type of functions used by the ATNs, although it could be used for any purpose.

2.2.1 Data types and variables

Variables are strongly and dynamically typed. They need to be declared or have a value assigned before using them and the type doesn't need to be specified, it will be deduced at runtime.

The language supports the following types:

- Integer numbers. For example, `a = 2;`.
- Strings, indicated with text between `""`. For example, `a = "hello";`.
- Boolean, using the keywords `true` or `false`.
- Arrays, which can have stored any of the aforementioned types. Array access can be performed indicating the position using `array[position]`. The positions are numbered from `0` to `n-1`, where `n` is the number of elements in the array. `n` can be accessed by using the keyword `array.length`.

Additionally, array access to a String is possible using the array notation, for example:

```
a = "Hello";
print a[0];

Output: H
```

Array initialization can be done in two different ways. We can initialize it with immediate values emplaced between `{}`. Note that arrays with different data types are **not** supported.

```
a = {1,2,3};
b = {"hello", "bye"};
```

Also, we can initialize them by indicating between brackets the last position of the array and it's value. An array of size `index + 1` will be created with he rest of positions initialized to `0`, `false` or `""` in the case of the array begin an integer, boolean or String array, respectively. Resizing arrays is performed dynamically. We then are able to modify lower positions of the array or resize the array by modifying a greater and nonexistent position on the array.

```
a[3] = true;
print a;

Output: [false,false,true]
```

2.2.2 Variable scope

The scope of the variables in the program are local. However, global variables are also supported. If variables are declared inside of a function, ATN or arc, they will be local to that function, ATN or arc. If they are declared outside of any of the aforementioned environments, they will be considered global.

If there are multiple initializations of the same variable in a global scope, the last initialization will be considered the current value

Essentially, it can be stated that there are three levels of scope: global, local to ATN or function and local to ATN arc.

Local variables can have the same name as global variables and different values associated to them. However, the global variable will not be able to be accessed since the local definitions have preference over the global ones.

2.2.3 Output

The keyword `print` will be used to print on screen the content of variables or immediate values. Printing of multiple Strings is supported by separating them with commas:

```
a = {"hello", "bye"};
print a[0], a[1];

Output: hello bye
```

All the special characters (endline, tabs, etc.) have to be used with the preceded with `%`. For example: `%n` would indicate an endline.

2.2.4 Input

The keyword `read` will be able to read user input from the `stdin`. It has the ability to read integer values and String values. To use it, we just need to indicate after the read keyword the name of the variable we want to store the input at:

```
read a;
print a;

Input: hello
Output: hello
```

2.2.5 Control flow statements

The control flow statements used are the following ones:

- `if` and `else` statements, with support of `else if` statements. In case of multiple `if` clauses, the `else` statement will be associated with the nearest `if`. Thus, the evaluation is greedy.

  ```
  if (condition) instruction;
  else if (condition) instruction;
  else instruction;
  ```

- `while` statements:

  ```
  while (condition)
      instruction;
  ```

- `for` statements:

  ```
  for (var_declaration; condition; increment)
      instruction;
  ```

- `return` statements **only** for functions (declared with `def`), not ATNs. They do not need to return a value in case the function is a `void` one:

```
return value;
```

It is also worth noting that in the case the `while`, `for` or `if` statements have more than one instruction, then the multiple instructions will be enclosed in a block using `{}`.

Additionally, all individual statements must be separated with `;` at the end of the line.

2.3 Functions and parameter passing

The declaration of functions is only possible on a global context, that is, outside an ATN method. They cannot be nested. The functions will be declared using the keyword `def` followed by the name of the function and the parameters needed between parenthesis. The parameter data type does not need to be indicated. They will return values using the keyword `return`. This keyword can be also used to return the execution frame to the callee without necessarily passing any value. They must also be enclosed using the symbols `{}`.

```
def func_name (par1, &par2){
    return par1;
}
```

The parameters received by a function are typeless and they can be passed either by value or by reference: if they are passed by **value**, a copy of the variable will be passed to the function. If they are passed by **reference**, the memory address of the variable will be passed. Expressions can be passed by value, whereas only variables can be passed by reference.

2.3.1 Expressions

ATNLang has a rich set of operators. The type of the variables intervening in an expression will be evaluated at runtime and, in case the types are not compatible, an error will be raised. The operators are the following, ordered by priority (top one has the highest):

- Unary operators `not`, `+` and `-` (negation).
- Multiplication (`*`), division (`/`) and integer remainder (`%`).
- Addition (`+`) and subtraction (`-`).
- Relational operators: `>=`, `<=`, `==`, `!=`, `>`, `<`.
- Logical `and`.
- Logical `or`.

The literals `true` and `false` are used to denote true or false values correspondingly. All the unary (except the `not`) and arithmetic expressions need to be used with integers and the logical ones with booleans. The only operation for strings available is concatenation using `+`.

String lexicographical comparison is also performed when comparing strings with relational operators.

```
print "aaa" < "aba";

Output: true
```

Comparison of booleans with `>=`, `<=`, `>`, `<` is **not** supported and an exception will be raised in case this evaluation is performed.

Array comparison with any of the relational operators is **not** supported.

3 Description of the Execution Phases

Since the program being developed for this project is an interpreter, only four distinct phases are done in each execution:

1. AST generation: For this phase, a grammar was designed for the ANTLR module (found in section 7). ANTLR parses the code and generates a traversable tree which will be explored by the interpreter.

2. AST pre-processing: the interpreter traverses the whole tree and: calculates all literal values found and maps all the global variables, function definitions and ATN definitions to their respective Hashmaps.

3. Program execution: the program is interpreted instruction by instruction, starting it by the function named `"main"`. That means that if a function is never called, it will never be executed and possible errors on that function will never be found.

4. ATN execution (Optional): if an ATN is defined in the program, the ATN interpreter module will be launched. This means that, automatically, the program will jump from node to node executing the code found in the arcs. A more detailed explanation of this process can be found at section 3.1

3.1 Backtracking algorithm

An algorithm was produced in order to execute the ATNs automatically, including exploring all the possible parsing paths available for the current input. That means, a chronological backtracking algorithm was implemented. The pseudocode below illustrates the algorithm used in the interpreter.

```
function Run () {
    Stack.pushAtnVars(atnVars);
    Data d = executeNode(startingNode));
    Stack().popAtnVars();
    return d;
}

function executeNode (name_node) {
    arc_list = nodeName2Tree.get(name_node);
    if (arc_list.getType() == ACCEPT) return true;

    for each arc in arc_list.getArcs() and not inputFullyParsed() {
        pushGlobalVariables();
        pushATNVars();
        curr_index_pos = getParsePosition();
        if (evaluateExpression(arc.condition())) {
```

```
            executeAfterCode(arc.code());

            forwardParseIndex();
            if (executeNode(arc.goto())) return true;
        }

        popATNVars();
        popGlobalVariables();
        setParseIndex(curr_index_pos);
    }
    return false;
}
```

To execute an ATN, the Run function must be called. The starting node will be set to the first node definition found in the ATN method. This will trigger a series of recursive calls to the `executeNode()` instruction each time an arc is followed. However, before each arc jump, the execution frame of the current code will be saved to later be recovered in case following the current selected arc does not return a successful parsing, that is, the value of the global variables, the value of the local ATN variables, and the position of the # array in the input sentence will be backed up.

By using the already existent recursion stack in Java, we are able to implement a recursive backtracking algorithm with no auxiliary data structures. The for loop will try to execute all the possible paths by making recursive calls to itself. An example of backtracking can be found in **??**.

3.2 Debugging options of the interpreter

In order to facilitate the debugging of program written in ATNLang, two features have been developed: the `-ast` flag, which outputs in a file the AST of the source of the program, and the `-trace` flag, which saves in a file all the function calls performed during the execution of the program. This will include all the calls to ATNs and the path of calls the ATN followed. This means all the traversed nodes and all followed arcs will be shown in the trace.

For example, the trace and AST tree of the following code can be found below:

```
atn a1 {
    node n1 {
        arc (#.0 != ".") goto n1 {
            print #.0, "%n";
        }
        arc (true) goto n2 {
```

```
            print #.0, "%n";
        }
    }
    node n2 accept;
}

def main() {
   atn(a1);
}
```

AST:

```
(PROGRAM (atn a1 (NODELIST (node n1 (ARC_LIST (arc (!= (# 0) ".") (goto n1) (LIST_INSTR
(print (# 0) "%n"))) (arc true (goto n2) (LIST_INSTR (print (# 0) "%n")))))
(node n2 accept))) (def main PARAMS (LIST_INSTR (ATNCALL a1))))
```

Execution Trace:

```
%% TODO: put execution trace if finally better trace showing full execution path is impleme
```

Our language also will print the Stacktrace when a program crashes, usually because an exception was raised. The Stacktrace will show as many information as possible about the current point where the program crashed. This includes, the depth of the Stack -the number of nested calls- and all the nested calls between functions and ATNs and their line where they were called. If an ATN was being executed during the crash, the node, arc and last word of the input that was parsed will be shown, as it can be seen below:

```
Runtime error (test/test-numbers.atn, line 5): Variable b not defined.
---------------
| Stack trace |
---------------
** Depth = 3
|> last token consumed: [dos]
|> node B1 at arc n°0: line 5
|> numbers: line 1
|> main: line 231
```

4 Examples

4.1 Example 1: Variable types

This example shows the types that variables from this language can be and also it is shown the dynamic typing.

```
def main() {
    a = 23;
    print a * 2, "%n";
    a = true;
    print a or false, "%n";
    a = "hello";
    print a + " world!", "%n";
    a[5] = 9;
    print a, " position 5 -> ", a[5], "%n";
    a[3] = "bye";
    print a, "%n";
    a[2] = true;
    a[5] = true;
    a[6] = a[5];
    a[5] = a[4] and a[5];
    print a, "%n";
    a = {1,2,3,4,5,6,7};
    print a, "%n";
}
```

Input:
Output:

```
    46
    true
    hello world!
    [0,0,0,0,0,9] position 5 -> 9
    [,,,bye]
    [false,false,true,false,false,false,true]
    [1,2,3,4,5,6,7]
```

4.2 Example 2: Input and output

This example shows how the input and out can be handled by the language.

```
def main() {
    print "test the input: ";
    read a;
```

```
    print "you have entered the value ", a, "%n";
    print "test input array: ";
    read a[3];
    print "you have created the array ", a, "%n";
    print a, "%n";
}
```

Input:

```
    1
    hola
```

Output:

```
    test the input: 1
    you have entered the value 1
    test input array: hola
    you have created the array [,,,hola]
    [,,,hola]
```

4.3 Example 3: Control flow

This example shows the control flow statements available on the language.

```
def if_example() {
    print "insert a number: ";
    read a;
    if (a > 2) print "hola", "%n";
    else print "adeu", "%n";
}

def while_example() {
    a = 1;
    while (a < 10) {
        print a, " ";
        a = a+1;
    }
    print "%n";
}

def for_example() {
    b = {1,2,3,4,5,6,7,8,9,10};
    for (a = 0; a < b.length; a = a+1) print b[a], " ";
    print "%n";
}
```

```
def return_example() { return "bye bye boy!" + "%n"; }

def main() {
    if_example();
    while_example();
    for_example();
    print return_example();
}
```

Input:
Output:

```
    insert a number: 3
    hola
    1 2 3 4 5 6 7 8 9
    1 2 3 4 5 6 7 8 9 10
    bye bye boy!
```

4.4 Example 4: Variable scope

This example shows the different variable scopes available on the language.

```
a = "hola!";

def local_func() {
    print a, "%n";
    a = "local hola!";
    print a, "%n";
}

atn example {
    ab = "local_atn";

    node noLocal {
        arc (#.0 == "1") goto noLocal {
            print a, "%n";
        }
        arc (#.0 == "2") goto END {
            a = "arc_local";
            print a, "%n";
        }
    }

    node END accept;
```

```
}

def main() {
    print a, "%n";
    local_func();
    print a, "%n";
    atn(example);
    print a, "%n";
}
```

Input:
ATN input:

```
    1
    2
```

Output:

```
    hola!
    hola!
    local hola!
    local hola!
    local hola!
    arc_local
    arc_local
```

4.5 Example 5: Fibbonacci

This example shows the potential that has as general purpose language ATNLang.

```
def fib(n) {
    fib = 0;
    next = 1;
    i = 0;
    while (i < n) {
        print fib, " ";
        aux = fib;
        fib = next;
        next = aux + next;
        i = i+1;
    }
}


def main() {
```

```
    fib(23);
    print "%n";
}
```

Input:
Output:

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711
```

4.6 Example 6: ATN calls from ATN

Example about ATN calls between ATNs and its recursivity.

```
atn exemple1 {
    node recurs {
        arc(atn(exemple2)) goto recurs {
            print "-yes-";
        }
    }
    node END accept;
}

atn exemple2 {
    node recurs {
        arc (true) goto END{
            print "-no-";
        }
    }
    node END accept;
}


def main() {
    atn(exemple1);
    print "%n";
}
```

Input:
ATN input:

```
    1
    2
    3
    4
    5
```

```
    6
    7
    8
    9
    10
```

Output:

```
    -no--yes--no--yes--no--yes--no--yes--no--yes-
```

4.7 Example 7: ATN sales recognizer

This example is an ATN that looks for value information in large texts about financial sales.

```
atn findMoney {
    found = 0;
    buy = 0;
    name = "";

    node sale {

        arc (#.shit == company and not buy) goto sale {
            found = 1;
            name = #.shit;
        }

        arc (#.shit == buy and found) goto sale {
            buy = 1;
        }

        arc (#.shit == company and buy) goto sale {
            print name, "has bought ", #.shit, "%n";
            buy = 0;
            found = 0;
        }

        arc (true) goto sale;

    }
}


def main() {
    atn(findMoney);
```

```
}
```

Input:
ATN input:
Output:

4.8 Example 8: ATN names

This example is an ATN name recognizer that implements the following ATN from the FreeLing library.

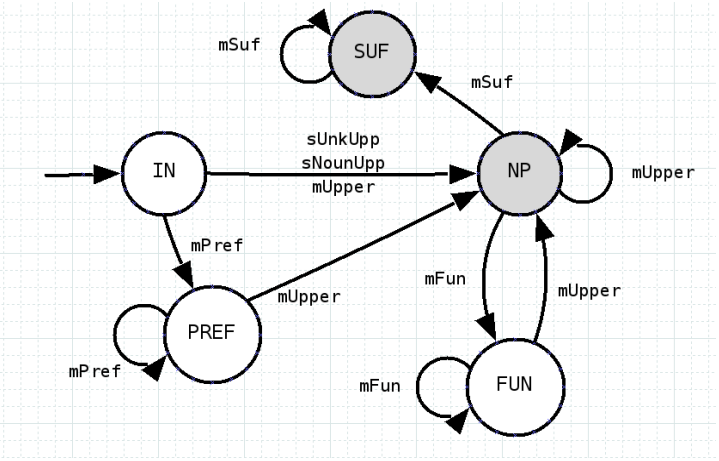

Figure 1: ATN names

```
def isUpper (s) {
    return s[0] >= "A" and s[0] <= "Z";
}

def member (s, l) {
    for (i = 0; i < l.length; i = i +1) {
        if (s == l[i]) return true;
    }
    return false;
}

nombre = "";

atn np {
    pre = {"dr", "sr", "de"};
    fun = {"de", "la", "el"};
    suf = {"junior", "senior"};
```

```
node IN {
    arc (isUpper(#.0)) goto NP {
        nombre = nombre + #.0 + " ";
    }
    arc (member(#.0, pre)) goto PREF {
        nombre = nombre + #.0 + " ";
    }
    arc(true) goto IN;
}
node PREF {
    arc (member(#.0, pre)) goto PREF {
        nombre = nombre + #.0 + " ";
    }
    arc (isUpper(#.0)) goto NP {
        nombre = nombre + #.0 + " ";
    }
    arc(true) goto IN {
        nombre = "";
    }
}
node NP {
    arc (isUpper(#.0)) goto NP {
        nombre = nombre + #.0 + " ";
    }
    arc (member(#.0, suf)) goto SUF {
        nombre = nombre + #.0 + " ";
    }
    arc (member(#.0, fun)) goto FUN {
        nombre = nombre + #.0 + " ";
    }
    arc (true) goto END;
}
node SUF {
    arc (member(#.0, suf)) goto SUF {
        nombre = nombre + #.0 + " ";
    }
    arc (true) goto END;
}
node FUN {
    arc (member(#.0, fun)) goto FUN {
        nombre = nombre + #.0 + " ";
    }
```

```
        arc (isUpper(#.0)) goto NP {
            nombre = nombre + #.0 + " ";
        }
    }
    node END accept;
}

def main() {
    a = atn(np);
    if (a) print "Nombre encontrado: ",nombre, "%n";
    else print "ningún nombre ha sido encontrado", "%n";
}
```

Input:
ATN input:
Output:

4.9 Example 9: ATN quantities

This example is an ATN quantities recognizer that implements the following ATN from the FreeLing library.

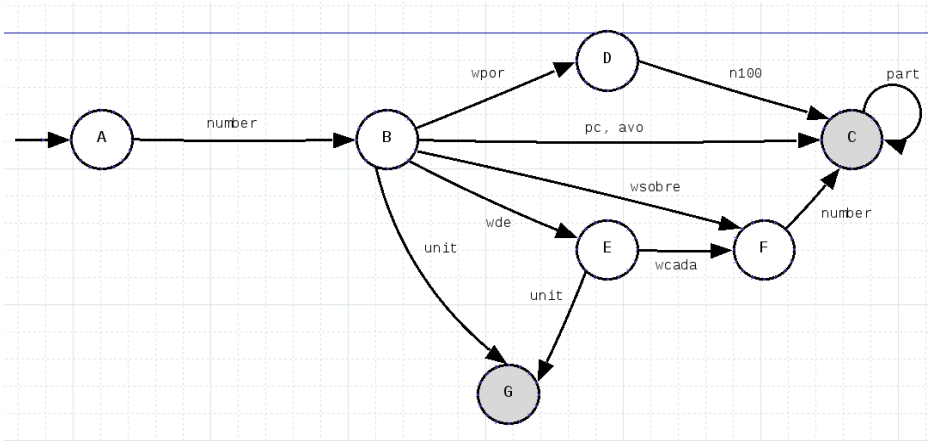

Figure 2: ATN quantities

```
% TODO: update the example
```

Input:
ATN input:
Output:

4.10 Example 10: ATN numbers

This example is an ATN numbers recognizer that implements the following ATN from the FreeLing library.

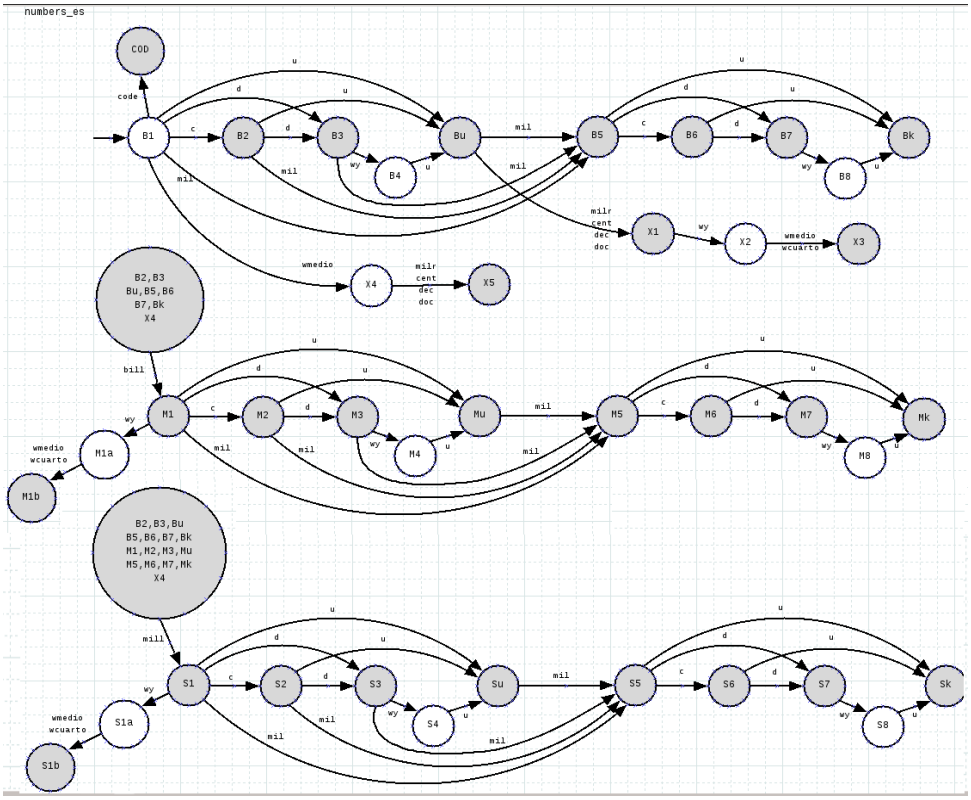

Figure 3: ATN numbers

```
% TODO: update the example
```

Input:
ATN input:
Output:

4.11 Example 11: ATN dates

This example is an ATN dates recognizer that implements the following ATN from the FreeLing library.

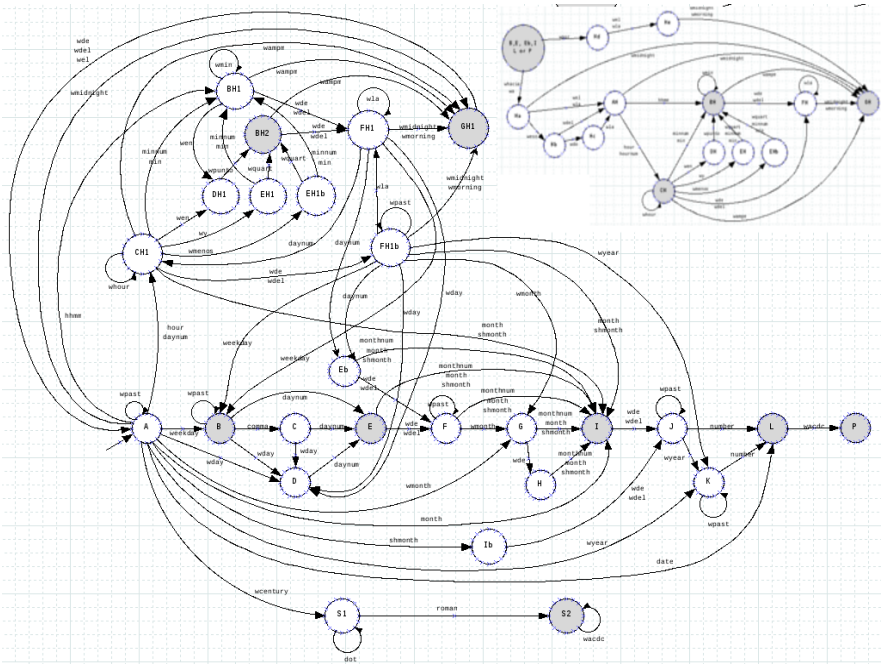

Figure 4: ATN dates

```
% TODO: code the example
```

Input:
ATN input:
Output:

4.12 Example 12: Backtracking

This example shows the capability of backtracking of the ATN. The ATN will test all the possible paths for the first node, when the input ends, it will backtrack and test the second arc, which will print the value it had the variable `a` before entering the first arc.

```
a = 1;

atn exemple1 {
    node recurs {
        arc(true) goto recurs {
            print a, ",";
            a = a+1;
        }
        arc(#.0 == "1") goto END {
```

```
            print "%n final value ", a;
        }
    }
    node END accept;
}

def main() {
    atn(exemple1);
}
```

ATN input: Text sentence with 10 words, with the first word being "1" (it does not matter the content of them).

Output:

```
1,2,3,4,5,6,7,8,9
final value 1
```

5 Workload distribution

As the two members of this project lived nearby and had similar schedules all the phases of development where performed by the both members at working in parallel in similar things together in meet ups weekly. That was decided because as that we would avoid problems on work synchronization and in a given moment where an specific problem could arise both would try to solve it sharing efforts. So all the code was pair-programmed and there are not specific parts of it that pertain to one or other member of the group.

6 Bibliography

References

[1] Xavier Carreras, Isaac Chao, Lluís Padró, and Muntsa Padró. Freeling: An open-source suite of language analyzers. In *Proceedings of the 4th International Conference on Language Resources and Evaluation (LREC'04)*, 2004.

[2] Stuart C Shapiro. Generalized augmented transition network grammars for generation from semantic networks. *Computational Linguistics*, 8(1):12–25, 1982.

[3] William A Woods. Transition network grammars for natural language analysis. *Communications of the ACM*, 13(10):591–606, 1970.

7 Annexe 1: Language Grammar

Below the grammar used to define the language in ANTLR can be found. Details about the syntax of the language can be found on section 2.

```
/**
 * ATN interpreter grammar
 */

grammar ATN;

options {
    output = AST;
    ASTLabelType = ATNTree;
}

tokens {
    PROGRAM;    // List of variables, functions and atns
    ASSIGN;     // Assignment instruction
    PARAMS;     // List of parameters in the declaration of a function
    FUNCALL;    // Function call
    ATNCALL;    // ATN function call
    ARGLIST;    // List of arguments passed in a function call
    LIST_INSTR; // Block of instructions
    ARRAY_DECL; // Array immediate declaration
    NODELIST;   // List of nodes of an ATN
    ARC_LIST;   // List of arc definitions
    BOOLEAN;    // Boolean atom (for Boolean constants "true" or "false")
    PVALUE;     // Parameter by value in the list of parameters
    PREF;       // Parameter by reference in the list of parameters
    ARRAYLENGTH;
}

@header {
package parser;
import interp.ATNTree;
}

@lexer::header {
package parser;
}

// A program is a list of utilities
prog    : utilities+ EOF -> ^(PROGRAM utilities+)
```

```
        ;

// An utility can be a function, a variable or an atn
utilities   : DEF^ ID params '{'! block_instructions '}'!
            | ATN^ ID '{'! node_list '}'!
            | assign ';'!   //Global vars
            ;

// List of nodes rooted to NODELIST
node_list   : node+ -> ^(NODELIST node+)
            ;

// A node can be a node definition or a local ATN
// variable assigment
node        : NODE^ ID arc_list
            | assign ';'!
            ;

// An arc list can also consist of only one definition
// or can be an accepting node
arc_list    : '{' arc+ '}' -> ^(ARC_LIST arc+)
            | arc -> ^(ARC_LIST arc)
            | ACCEPT^ ';'!
            ;

arc        : ARC^ '('! expr ')'! arc_jump list_instructions
           ;

arc_jump    : JUMP^ ID
            ;

// The list of parameters grouped in a subtree (it can be empty)
params  : '(' paramlist? ')' -> ^(PARAMS paramlist?)
        ;

// Parameters are separated by commas
paramlist
        : param (','! param)*
        ;

// Parameters with & as prefix are passed by reference
// Only one node with the name of the parameter is created
param   :   '&' id=ID -> ^(PREF[$id,$id.text])
```

```
        |   id=ID -> ^(PVALUE[$id,$id.text])
        ;

// A list of instructions, either enclosed in brackets or not
// Also, a list of instructions can be empty
list_instructions
        :  instruction -> ^(LIST_INSTR instruction)
        |  ';'!
        |  '{'! block_instructions '}'!
        ;

// A block of instructions, all of them gouped in a subtree
block_instructions
        :  instruction+ -> ^(LIST_INSTR instruction+)
        ;

// The different types of instructions, followed by semicolon
instruction
        :   assign ';'!         // Assignment
        |   ite_stmt            // if-then-else
        |   read ';'!           // read some input variable
        |   while_stmt          // while statement
        |   for_stmt            // for statement
        |   funcall ';'!        // Call to a procedure (no result produced)
        |   atncall ';'!        // Call to a procedure (fail if returns false)
        |   return_stmt ';'!    // Return statement
        |   print ';'!          // Write a string or an expression
        ;

// Assignment
assign  :   subatom eq=EQUAL expr -> ^(ASSIGN[$eq,":="] subatom expr)
        ;

// if-then-else (else is optional)
ite_stmt    :   IF^ '('! expr ')'! list_instructions
                (options {greedy=true;} : ELSE! list_instructions)?
            ;

// while statement
while_stmt  :   WHILE^ '('! expr ')'! list_instructions
            ;

// for statement
```

```
for_stmt    :   FOR^ '('! assign ';'! expr ';'! assign ')'! list_instructions
            ;

// Write an expression or a string
print   :   PRINT^ expr (','! expr)*
        ;

// Read a variable
read    :   READ^ subatom
        ;

//Return of a value in a function
return_stmt
        :  RETURN^ expr?
        ;

// Grammar for expressions with boolean, relational and aritmetic operators
expr    :   boolterm (OR^ boolterm)*
        ;

boolterm:   boolfact (AND^ boolfact)*
        ;

boolfact:   num_expr ((EQ_COMP^ | NOT_EQUAL^ | LT^ | LE^ | GT^ | GE^) num_expr)?
        ;

num_expr:   term ( (PLUS^ | MINUS^) term)*
        ;

term    :   factor ( (MUL^ | DIV^ | MOD^) factor)*
        ;

factor  :   (NOT^ | PLUS^ | MINUS^) factor | atom
        ;

// Atom of the expressions (variables, integer and boolean literals).
// An atom can also be a function call or another expression
// in parenthesis
atom    :   subatom
        |   INT
        |   STRING
        |   '{' expr (',' expr)* '}' -> ^(ARRAY_DECL expr*)
        |   (b=TRUE | b=FALSE)  -> ^(BOOLEAN[$b,$b.text])
```

```
        |   funcall
        |   atncall     //calls atn and saves value on var
        |   '('! expr ')'!
        |   HASH^ '.'! INT
        ;

// Array definition or array length checking
subatom :   ID (BRACKET^ expr ']'!)?
        |   ID '.length' -> ^(ARRAYLENGTH ID)
        ;

// A function call has a lits of arguments in parenthesis (possibly empty)
funcall :   ID '(' expr_list? ')' -> ^(FUNCALL ID ^(ARGLIST expr_list?))
        ;

// A function call has a lits of arguments in parenthesis (possibly empty)
atncall :   'atn' '(' ID ')' -> ^(ATNCALL ID)
        ;

// A list of expressions separated by commas
expr_list:  expr (','! expr)*
        ;

// Basic tokens
BRACKET : '[' ;
EQUAL   : '=' ;
EQ_COMP : '==' ;
NOT_EQUAL: '!=' ;
LT      : '<' ;
LE      : '<=';
GT      : '>';
GE      : '>=';
PLUS    : '+' ;
MINUS   : '-' ;
MUL     : '*';
DIV     : '/';
MOD     : '%' ;
NOT     : 'not';
AND     : 'and' ;
OR      : 'or' ;
IF      : 'if' ;
ELSE    : 'else' ;
WHILE   : 'while' ;
```

```
FOR     : 'for' ;
NODE    : 'node' ;
ARC     : 'arc' ;
ATN     : 'atn' ;
DEF     : 'def' ;
ACCEPT  : 'accept' ;
READ    : 'read' ;
RETURN  : 'return' ;
PRINT   : 'print' ;
JUMP    : 'goto' ;
TRUE    : 'true' ;
FALSE   : 'false';
HASH    : '#';
ID      :   ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')* ;
INT     :   '0'..'9'+ ;

// C-style comments
COMMENT : '//' ~('\n'|'\r')* '\r'? '\n' {$channel=HIDDEN;}
        | '/*' ( options {greedy=false;} : . )* '*/' {$channel=HIDDEN;}
        ;

// Strings (in quotes) with escape sequences
STRING  :  '"' ( ESC_SEQ | ~('\\'|'"') )* '"'
        ;

fragment
ESC_SEQ
    :   '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\')
    ;

// White spaces
WS      : ( ' '
        | '\t'
        | '\r'
        | '\n'
        ) {$channel=HIDDEN;}
        ;
```