Compilers

PROJECT REPORT: AN INTERPRETER FOR AUGMENTED TRANSITION NETWORKS

May 28, 2015

Daniel Otero and Guido Arnau

Universitat Politècnica de Catalunya
Facultat d'Informàtica de Barcelona

Contents

1 Introduction

This project consisted in the development 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 execute 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

This project consisted in the development of 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 preanalyse 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.

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).

2 Language syntax and semantic

The language syntax is designed as simple and intuitive as possible. Conceptually the language has two distinct aspects: 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.

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 moethod 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.

2.1.1 The `accept` keyword

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

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 needs to iterate 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 access 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.

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

In the event 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 the global variables of the program.
2. Local ATN variables: variables that can be accessed by all of the arc's code of the ATN.
3. Local arc variables: variables created and only accessed inside an arc's code.

2.1.4 ATN features: Backtracking and recursion

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 typed 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 initial value.

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
```

2.2.4 Input

The keyword `read` will be able to read user input from the `stdin`. It has the ability to read the word `"true"` and `"false"` as a boolean values, 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.

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

3 Description of the Execution Phases

3.1 Debugging options of the interpreter

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 | 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] & a[5]
    print a, "%n"
}
```

Input:
Output:

4.2 Exemple 2: Input and output

```
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";
}
```

5 Workload distribution

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