Compilers

SPECIFICATION: AN INTERPRETER FOR AUGMENTED TRANSITION NETWORKS

April 4, 2015

Daniel Otero i Guido Arnau

Universitat Politècnica de Catalunya
Facultat d'Informàtica de Barcelona

Contents

1 Project goal

In this project we want to create an interpreter for ATNs. First of all, 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 set of languages they accept is the same as the one accepted by context-free grammars.

The parse proceeds following a depth-first search of the ATN, it succeeds if a final node is reached. During this process an abstract parse tree is constructed by the actions performed during arc traversal.

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 consists in the development of an ATN interpreter called ATNLang. In other words, a program will be created in order to parse ATN code and execute it. That means an ATN will be loaded into the interpreter and queries in the form of natural language sentences will be interpreted by it. The program will output the grammatical analysis of the sentence produced by the execution of the loaded ATN with the queried sentence as input.

Other extra features will be developed and built into the interpreter: a trace feature in order to visualize the traversal through the ATN that the input sentence produced, output of various possible traversals of the ATN in case there are multiple of them (in ambiguous sentences, for example).

The ATN interpreter has a built-in natural language analyser (the open source program FreeLing, a language analysis tool suite [1]) in order to recognize the characteristics of the words given to the ATN and the connections between them. The FreeLing dictionary will return, for each word, a set of three strings: 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.

1.3 Features of the ATN description language

In order to describe ATNs, an ATN description language will be designed. This language must allow full expressivity to describe all the characteristics of an ATN. This characteristics are:

1. Recursivity - allow the call of sub-ATNs or part of the same ATN in order to check if an arc must be followed or not.
2. Internal variables - every ATN has the ability to create internal variables while parsing and use them to control the execution flow.
3. Tests - boolean functions that will be evaluated to decide to follow or not an arc.
4. Actions - set of instructions that will be executed in the case an arc is followed.

Thus, an ATN will be described as a list of nodes. The nodes are a list of arcs and each arc has a test and an action associated to it.

2 Language syntax

The language syntax is designed as simple and intuitive as possible. Conceptually we split it into two distinct syntaxes: one used to describe the ATN structure (nodes, arcs, destinations) and the second one used to describe the boolean conditions of the arcs, the actions performed in each arc and auxiliar functions included in the program (non-specific purpose language). Since the core ATN structure is always made of nodes and arcs, this design will give more expressivity to the actions and tests of the ATN and produce more elegant and intuitive programs.

2.1 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 {

    node Node_name {
        arc (boolean_expression) {
            instruction_1;
            instruction_2;
            instruction_3;
            goto Node_name; (optional)
            print text;   (optional)
            return something (optional)
            ...
        }
        arc (...) {...}
        ...
    }

    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 arc, the actions to be performed in case the arc is followed and the ATN's node we want to go to (expressed with the keyword `goto`) or `return` in case you want the ATN to return some value.

The boolean condition (defined inside **parenthesis**) of the arcs allow the use of the instructions described in section 2.3.1 plus the capability of calling other sub-ATN described

in the same program using the keyword `parse`. `parse` will return the value returned by the ATN after it's execution. In the case the ATN doesn't return any value, parse will not return anything either. This implements the **recursivity** between ATN mentioned in section 1.3.

The actions (the code inside the arc's method) will be able to execute any of the described instructions in section 2.2. They can be viewed as a normal function with a `return` or `goto` statement at the end.

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

If the arc has the keyword `return` instead of `goto`, then the ATN will pass the variable specified after return to the ATN that called him.

The order in which the 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 execution frame will be returned to the callee ATN. If the ATN is the main one, an error message will be prompted on screen.

2.1.1 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 this case `#`.

As described in section 1.2, the FreeLing module will return a triplet containing three strings: the current word, the base word from which the current word derives and a code with information about the word. Thus, the symbol `#` will have access to the triplet of the current word and will move forward to the next word using the call `#.forward()`. If the iterator reaches the end of the sentence, the call to `#.end()` will return `true`.

2.2 Non-specific purpose language

The interpreter will also implement a basic imperative, strongly and dynamically typed language. This language will be used in the arc's tests, actions and other auxiliary functions used by the ATNs. The language syntax will be similar to `C++98`.

2.2.1 Data types and variables

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`.

Variables are strongly typed and cannot change it's type dynamically. They need to be declared before using them and the type doesn't need to be specified, it will be deduced at runtime.

The assignment of variables will be the regular one found in imperative languages using the symbol `=`:

```
a = 2
```

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, node or arc, they will be local to that function, ATN, node or arc. If they are declared outside of any of the aforementioned environments, they will be considered global.

Therefore, there are two ways in order to pass variables between functions and ATNs: either by parameter passing (explained in detail in section 2.3) or by declaring global variables.

2.2.3 Output

The keyword `print` will be used to print on screen the content of variables or immediate values. For example, the following code would print on screen `bye`:

```
a = ["hello", "bye"]
print a[1]
```

2.2.4 Control flow statements

The control flow statements of the language are very similar to the ones of C++98. These are the following ones:

- `if` statements, with support of `else if` statements:

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

- `while` statements:

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

- `for` statements:

```
for (var declaration; condition; increment)
    instruction;
```

- `return` statements as previously seen for ATNs, they also work for functions:

```
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 `;`.

2.3 Functions and parameter passing

The declaration of functions is only possible on a global context, that is, outside an ATN method. The functions will be declared using the keyword `def` followed by the name of the function and the parameters needed between parenthesis. They will return values using the keyword `return` the same way that ATNs do. 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`.

Boolean expressions follow the same syntax as C++98. 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 `+`.

It is worth mentioning that the value 0 will be considered false and any value different from it will be considered true.

2.4 Examples

In this section three examples are shown in order to demonstrate the syntax and capabilities of the language.

2.5 Example 1: Detection of diminutives

An ATN that searches if there are diminutives in a text is shown below.

```
def tail (a) {
    return a[a.length-1];
}

atn Exemple1 {

    node Diminutivo {
        arc (tail(#.code) == D) {
            print true;
        }
        arc (#.end()) {
            print false;
        }
        arc (true) {
            #.forward();
            goto Diminutivo;
        }
```

```
    }

}

Exemple1(El gato es bonito) -> Output: false
Exemple1(El gatito es feo) -> Output: true
```

2.6 Example 2: Apparitions of words and their derived words

Below an ATN that count the number of apparitions of the word "animal" or derived words from it (in Spanish).

```
count = 0;

atn Example2 {

    node Apariciones {
        arc (#.base == "animal") {
            count = count + 1;
            #.forward();
            goto Apariciones;
        }
        arc (#.end()) {
            print count;
        }
        arc (true) {
            #.forward();
            goto Apariciones;
        }
    }

}

Example2(El reino de los vegetales es muy variado) -> Output: 0
Example2(Los animalillos tienen miedo de los animalotes del reino animal) -> Output: 3
```

2.7 Example 3: Noun and determinant parser

Below an ATN that count the number of apparitions of the word "animal" or derived words from it (in Spanish).

```
res = "";
tmp = "";

atn Example3 {

    node NP {
        arc (#.end()) goto PRINT;
        arc (#.code == buscarcodidet) {
            tmp = #.text;
            #.forward();
            goto NP-DET;
        }
        arc (#.code == buscarcodinom) {
            res = res + #.text + "[noun] ";
            #.forward();
            goto NP;
        }
        arc (true) {
            res = res + #.text + " ";
            #.forward();
            goto NP;
        }
    }

    node NP-DET {
        arc (#.end()) {
            res = res + tmp + "(det)";
            goto PRINT:
        }
        arc (#.code == bucarcodinom) {
            res = res + "(" + tmp + " " +
                #.text + ")[det+noun] ";
            #.forward();
            goto NP;
        }
        arc (true) {
            res = res + tmp + "(det) " + #.text + " ";
            #.forward();
            goto NP;
        }
    }

    node PRINT {
```

```
        arc (true) {
            print res;
            return;
        }
    }
}
```

```
Exemple3(El coche de papá está roto) -> Output: (El coche)[det+noun] de papá[noun] está rot
```

3 Components of the Program

In this section the different modules that make up the program will be described. These modules are: the parser, the interpreter and the external software used by the interpreter.

3.1 The parser

The parser of the program will be provided by `ANTLR 3.4`. Thus, a grammar for the language will be created in a file called `ATN.g`. `ANTLR` will generate the rest of the required `.java` files.

3.2 The interpreter

The interpreter will grab the AST tree generated by the `ANTLR` parser and load it into memory. If queries to the ATNs are found in the code, they will be executed and the result will be printed on screen. Then the user will be prompted into a command line in order to query the loaded ATNs with more sentences. Each query will execute the main ATN with the queried sentence as input.

3.3 Usage of the interpreter

The user will load a file containing the code of one or more ATNs. Once the program describing the ATN has been parsed, the interpreter can be used to query the ATN with natural language sentences. This queries can be done by specifying the ATN's name followed by the input text between parenthesis: `atns_name(input_text)`. The program then will print the output generated by the program.

The interpreter will also have the ability to trace the parse of the sentence through the ATN when the option `-trace` is used. All the performed tests, sub-ATNs called and values returned will be shown as a tree. This will be useful when debugging the program.

3.4 External software

As stated before, the interpreter will use the FreeLing [1] suite as a back-end in order to receive information about the words being interpreted. Each word of the sentence will be fed into FreeLing and the interpreter will use the returned triplet (word, word from which the original word derives from and the FreeLing code) to get information about the grammatical and syntactical significance of that word in the sentence. The programmers will be able to access to this triplet and check characteristics of the words. For example, programmers will be able to check if a noun is plural, the gender of the noun, etc. Therefore, the interpreter will benefit form the same features present in the FreeLing module (contraction splitting, suffix treatment, recognition of dates, numbers, ratios, currency, and physical magnitudes and many more).

4 Workload distribution

This project is being developed by two people so distributing well the work and performing it in parallel and efficiently is an important point to get a successful result.

The firsts steps on the design where made together because agreeing and sharing ideas was crucial. From there we did the first design, grammar, and this document together working on the same document at the same time but editing different parts of it.

The implementation will be distributed in this big blocks: interpreter, data structures, main program and extra functionalities. One of us will code the interpreter and extra functionalities and another the data structures classes and main program. even though we will work on different parts of the project we will share a repository and have all our progress in common and also we will help each other if it is needed.

If we finish all the core project and we still have some time, we will add extra functionalities to the ATN interpreter as printing the nodes and arcs in an elegant way, adding new control flow instructions or more complex data types. To do this we will assign to each of us an specific goal and try to code it.

5 Appendix: Language grammar

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

grammar ATN;

options {
    output = AST;
    ASTLabelType = AslTree;
}

tokens {
    PROGRAM;    // List of variables, functions and atns
    ASSIGN;     // Assignment instruction
    PARAMS;     // List of parameters in the declaration of a function
    FUNCALL;    // Function call
    ARGLIST;    // List of arguments passed in a function call
    LIST_INSTR; // Block of instructions
    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
    HASHCODE;   // Especial token for the atn input pointer
    HASHTEXT;   // Especial token for the atn input pointer
    HASHBASE;   // Especial token for the atn input pointer
    HASHFORWARD; // Especial token for the atn input pointer
    HASHEND;    // Especial token for the atn input pointer
}

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

@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 ';'!
| funcall ';'!
            ;

node_list   : node (node)+
            ;

node        : NODE^ ID '{'! arc_list '}'!
            ;

arc_list    : arc (arc)+
            ;

arc        : ARC^ '(' expr ')'! instructions
           ;


// 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])
        ;

instructions
        :  inst_comma -> ^(LIST_INSTR inst_comma)
        |  '{'! block_instructions '}'!
        ;

// A list of instructions, all of them gouped in a subtree
block_instructions
        :  inst_comma (inst_comma)*
            -> ^(LIST_INSTR inst_comma+)
```

```
        ;

inst_comma
        : instruction ';'!
        ;

// The different types of instructions
instruction
        :   assign          // Assignment
        |   ite_stmt        // if-then-else
        |   while_stmt      // while statement
        |   for_stmt        // for statement
        |   funcall         // Call to a procedure (no result produced)
        |   return_stmt     // Return statement
        |   print           // Write a string or an expression
        |                   // Nothing
        ;

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

// if-then-else (else is optional)
ite_stmt    :   IF^ '('! expr ')'! instructions
                (ELSE! instructions)?
            ;

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

// for statement
for_stmt    :   FOR^ '('! assign ';'! expr ';'! assign ')'! instructions
            ;

// Return statement with an expression
return_stmt :   RETURN^ expr?
            ;

// Write an expression or a string
print   :   PRINT^ (expr | STRING )
        ;
```

```
// go to an indicated atn node
goto    :   GOTO^ ID
        ;

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

boolterm:   boolfact (AND^ boolfact)*
        ;

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

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

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

factor  :   (NOT^ | PLUS^ | MINUS^)? 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
        |   (b=TRUE | b=FALSE)  -> ^(BOOLEAN[$b,$b.text])
        |   funcall
        |   '('! expr ')'!
        |   '#.code' -> HASHCODE
        |   '#.base' -> HASHBASE
        |   '#.text' -> HASHBASE
        |   '#.forward()' -> HASHFORWARD
        |   '#.end()' -> HASHEND
        ;

subatom :   ID (BRACKET^ expr ']'!)?
        ;

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

```
        ;

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

// Basic tokens
BRACKET : '[' ;
EQUAL   : '=' ;
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' ;
RETURN  : 'return' ;
PRINT   : 'print' ;
GOTO    : 'goto' ;
TRUE    : 'true' ;
FALSE   : 'false';
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;}
        ;
```

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.