23

Parsing with ATNs

This chapter shows how to write a nondeterministic parser as an embedded language. The first part explains what ATN parsers are, and how they represent grammar rules. The second part presents an ATN compiler which uses the nondeterministic operators defined in the previous chapter. The final sections present a small ATN grammar, and show it in action parsing sample input.

23.1 Background

Augmented Transition Networks, or ATNs, are a form of parser described by Bill Woods in 1970. Since then they have become a widely used formalism for ○ parsing natural language. In an hour you can write an ATN grammar which parses interesting English sentences. For this reason, people are often held in a sort of spell when they first encounter them.

In the 1970s, some people thought that ATNs might one day be components of truly intelligent-seeming programs. Though few hold this position today, ATNs have found a niche. They aren't as good as you are at parsing English, but they can still parse an impressive variety of sentences.

ATNs are useful if you observe the following four restrictions:

1. Use them in a semantically limited domain—in a front-end to a particular database, for example.

2. Don't feed them very difficult input. Among other things, don't expect them to understand wildly ungrammatical sentences the way people can.

3. Only use them for English, or other languages in which word order determines grammatical structure. ATNs would not be useful in parsing inflected languages like Latin.

4. Don't expect them to work all the time. Use them in applications where it's helpful if they work ninety percent of the time, not those where it's critical that they work a hundred percent of the time.

Within these limits there are plenty of useful applications. The canonical example is as the front-end of a database. If you attach an ATN-driven interface to such a system, then instead of making a formal query, users can ask questions in a constrained form of English.

23.2 The Formalism

To understand what ATNs do, we should recall their full name: augmented transition networks. A transition network is a set of nodes joined together by directed arcs—essentially, a flow-chart. One node is designated the start node, and some other nodes are designated terminal nodes. Conditions are attached to each arc, which have to be met before the arc can be followed. There will be an input sentence, with a pointer to the current word. Following some arcs will cause the pointer to be advanced. To parse a sentence on a transition network is to find a path from the start node to some terminal node, along which all the conditions can be met.

ATNs add two features to this model:

1. ATNs have registers—named slots for storing away information as the parse proceeds. As well as performing tests, arcs can modify the contents of the registers.

2. ATNs are recursive. Arcs may require that, in order to follow them, the parse must successfully make it through some sub-network.

Terminal nodes use the information which has accumulated in the registers to build list structures, which they return in much the same way that functions return values. In fact, with the exception of being nondeterministic, ATNs behave a lot like a functional programming language.

The ATN defined in Figure 23.1 is nearly the simplest possible. It parses noun-verb sentences of the form "Spot runs." The network representation of this ATN is shown in Figure 23.2.

What does this ATN do when given the input `(spot runs)`? The first node has one outgoing arc, a cat, or category arc, leading to node `s2`. It says, effectively,

```
(defnode s
  (cat noun s2
    (setr subj *)))

(defnode s2
  (cat verb s3
    (setr v *)))

(defnode s3
  (up `(sentence
         (subject ,(getr subj))
         (verb ,(getr v)))))
```

Figure 23.1: A very small ATN.

Figure 23.2: Graph of a small ATN.

you can follow me if the current word is a noun, and if you do, you must store the current word (indicated by `*`) in the `subj` register. So we leave this node with `spot` stored in the `subj` register.

There is always a pointer to the current word. Initially it points to the first word in the sentence. When cat arcs are followed, this pointer is moved forward one. So when we get to node `s2`, the current word is the second, `runs`. The second arc is just like the first, except that it is looking for a verb. It finds `runs`, stores it in register `v`, and proceeds to `s3`.

The final node, `s3`, has only a pop, or terminal, arc. (Nodes with pop arcs have dashed borders.) Because we arrive at the pop arc just as we run out of input, we have a successful parse. The pop arc returns the backquoted expression within it:

```
(sentence (subject spot)
          (verb runs))
```

An ATN corresponds to the grammar of the language it is designed to parse. A decent-sized ATN for parsing English will have a main network for parsing sentences, and sub-networks for parsing noun-phrases, prepositional phrases, modifier groups, and so on. The need for recursion is obvious when we consider that noun-phrases may contain prepositional phrases which may contain noun-phrases, *ad infinitum*, as in

"the key on the table in the hall of the house on the hill"

23.3 Nondeterminism

Although we didn't see it in this small example, ATNs are nondeterministic. A node can have several outgoing arcs, more than one of which could be followed with a given input. For example, a reasonably good ATN should be able to parse both imperative and declarative sentences. Thus the first node could have outgoing cat arcs for both nouns (in statements) and verbs (in commands).

What if the first word of the sentence is "time," which is both a noun and a verb? How does the parser know which arc to follow? When ATNs are described as nondeterministic, it means that users can assume that the parser will *correctly guess* which arc to follow. If some arcs lead only to failed parses, they won't be followed.

In reality the parser cannot look into the future. It simulates correct guessing by backtracking when it runs out of arcs, or input. But all the machinery of backtracking is inserted automatically into the code generated by the ATN compiler. We can write ATNs as if the parser really could guess which arcs to follow.

Like many (perhaps most) programs which use nondeterminism, ATNs use the depth-first implementation. Experience parsing English quickly teaches one that any given sentence has a slew of legal parsings, most of them junk. On a conventional single-processor machine, one is better off trying to get good parses quickly. Instead of getting all the parses at once, we get just the most likely. If it has a reasonable interpretation, then we have saved the effort of finding other parses; if not, we can call *fail* to get more.

To control the order in which parses are generated, the programmer needs to have some way of controlling the order in which *choose* tries alternatives. The depth-first implementation isn't the only way of controlling the order of the search. Any implementation except a randomizing one imposes some kind of order. However, ATNs, like Prolog, have the depth-first implementation conceptually built-in. In an ATN, the arcs leaving a node are tried in the order in which they were defined. This convention allows the programmer to order arcs by priority.

23.4 An ATN Compiler

Ordinarily, an ATN-based parser needs three components: the ATN itself, an interpreter for traversing it, and a dictionary which can tell it, for example, that "runs" is a verb. Dictionaries are a separate topic—here we will use a rudimentary hand-made one. Nor will we need to deal with a network interpreter, because we will translate the ATN directly into Lisp code. The program described here is called an ATN compiler because it transforms a whole ATN into code. Nodes are transformed into functions, and arcs become blocks of code within them.

Chapter 6 introduced the use of functions as a form of representation. This practice usually makes programs faster. Here it means that there will be no overhead of interpreting the network at runtime. The disadvantage is that there is less to inspect when something goes wrong, especially if you're using a Common Lisp implementation which doesn't provide `function-lambda-expression`.

Figure 23.3 contains all the code for transforming ATN nodes into Lisp code. The macro `defnode` is used to define nodes. It generates little code itself, just a `choose` over the expressions generated for each of the arcs. The two parameters of a node-function get the following values: `pos` is the current input pointer (an integer), and `regs` is the current set of registers (a list of assoc-lists).

The macro `defnode` defines a macro with the same name as the corresponding node. Node `s` will be defined as macro `s`. This convention enables arcs to know how to refer to their destination nodes—they just call the macro with that name. It also means that you shouldn't give nodes the names of existing functions or macros, or these will be redefined.

Debugging ATNs requires some sort of trace facility. Because nodes become functions, we don't have to write our own. We can use the built-in Lisp function `trace`. As mentioned on page 266, using `=defun` to define nodes means that we can trace parses going through node `mods` by saying `(trace =mods)`.

The arcs within the body of a node are simply macro calls, returning code which gets embedded in the node function being made by `defnode`. The parser uses nondeterminism at each node by executing a `choose` over the code representing each of the arcs leaving that node. A node with several outgoing arcs, say

```
(defnode foo
  <arc 1>
  <arc 2>)
```

gets translated into a function definition of the following form:

```
(=defun foo (pos regs)
  (choose
    <translation of arc 1>
    <translation of arc 2>))
```

```
(defmacro defnode (name &rest arcs)
  `(=defun ,name (pos regs) (choose ,@arcs)))

(defmacro down (sub next &rest cmds)
  `(=bind (* pos regs) (,sub pos (cons nil regs))
     (,next pos ,(compile-cmds cmds))))

(defmacro cat (cat next &rest cmds)
  `(if (= (length *sent*) pos)
       (fail)
       (let ((* (nth pos *sent*)))
         (if (member ',cat (types *))
             (,next (1+ pos) ,(compile-cmds cmds))
             (fail)))))

(defmacro jump (next &rest cmds)
  `(,next pos ,(compile-cmds cmds)))

(defun compile-cmds (cmds)
  (if (null cmds)
      'regs
      `(,@(car cmds) ,(compile-cmds (cdr cmds)))))

(defmacro up (expr)
  `(let ((* (nth pos *sent*)))
     (=values ,expr pos (cdr regs))))

(defmacro getr (key &optional (regs 'regs))
  `(let ((result (cdr (assoc ',key (car ,regs)))))
     (if (cdr result) result (car result))))

(defmacro set-register (key val regs)
  `(cons (cons (cons ,key ,val) (car ,regs))
         (cdr ,regs)))

(defmacro setr (key val regs)
  `(set-register ',key (list ,val) ,regs))

(defmacro pushr (key val regs)
  `(set-register ',key
                 (cons ,val (cdr (assoc ',key (car ,regs))))
                 ,regs))
```

Figure 23.3: Compilation of nodes and arcs.

```
(defnode s
  (down np s/subj
    (setr mood 'decl)
    (setr subj *))
  (cat v v
    (setr mood 'imp)
    (setr subj '(np (pron you)))
    (setr aux nil)
    (setr v *)))
```

is macroexpanded into:

```
(=defun s (pos regs)
  (choose
    (=bind (* pos regs) (np pos (cons nil regs))
      (s/subj pos
              (setr mood 'decl
                    (setr subj * regs))))
    (if (= (length *sent*) pos)
        (fail)
        (let ((* (nth pos *sent*)))
          (if (member 'v (types *))
              (v (1+ pos)
                 (setr mood 'imp
                       (setr subj '(np (pron you))
                             (setr aux nil
                                   (setr v * regs)))))
              (fail))))))
```

Figure 23.4: Macroexpansion of a node function.

Figure 23.4 shows the macroexpansion of the first node in the sample ATN of Figure 23.11. When called at runtime, node functions like s nondeterministically choose an arc to follow. The parameter pos will be the current position in the input sentence, and regs the current registers.

Cat arcs, as we saw in our original example, insist that the current word of input belong to a certain grammatical category. Within the body of a cat arc, the symbol * will be bound to the current word of input.

Push arcs, defined with down, require successful calls to sub-networks. They take two destination nodes, the sub-network destination sub, and the next node in the current network, next. Notice that whereas the code generated for a cat arc simply calls the next node in the network, the code generated for a push arc uses `=bind`. The push arc must successfully return from the sub-network before continuing on to the node which follows it. A clean set of registers (`nil`) gets consed onto the front of `regs` before they are passed to the sub-network. In the bodies of other types of arcs, the symbol `*` will be bound to the current word of input, but in push arcs it will be bound to the expression returned by the sub-network.

Jump arcs are like short-circuits. The parser skips right across to the destination node—no tests are required, and the input pointer isn't advanced.

The final type of arc is the pop arc, defined with `up`. Pop arcs are unusual in that they don't have a destination. Just as a Lisp `return` leads not to a subroutine but the calling function, a pop arc leads not to a new node but back to the "calling" push arc. The `=values` in a pop arc "returns" a value to the `=bind` in the most recent push arc. But, as Section 20.2 explained, what's happening is not a normal Lisp `return`: the body of the `=bind` has been wrapped up into a continuation and passed down as a parameter through any number of arcs, until the `=values` of the pop arc finally calls it on the "return" values.

Chapter 22 described two versions of nondeterministic *choose*: a fast `choose` (page 293) that wasn't guaranteed to terminate when there were loops in the search space, and a slower `true-choose` (page 304) which was safe from such loops. There can be cycles in an ATN, of course, but as long as at least one arc in each cycle advances the input pointer, the parser will eventually run off the end of the sentence. The problem arises with cycles which don't advance the input pointer. Here we have two alternatives:

1. Use the slower, correct nondeterministic choice operator (the depth-first version given on page 396).

2. Use the fast `choose`, and specify that it is an error to define networks containing cycles which could be traversed by following just jump arcs.

The code defined in Figure 23.3 takes the second approach.

The last four definitions in Figure 23.3 define the macros used to read and set registers within arc bodies. In this program, register sets are represented as assoc-lists. An ATN deals not with sets of registers, but sets of sets of registers. When the parser moves down to a sub-network, it gets a clean set of registers pushed on top of the existing ones. Thus the whole collection of registers, at any given time, is a list of assoc-lists.

The predefined register operators work on the current, or topmost, set of regis-
○ ters: `getr` reads a register; `setr` sets one; and `pushr` pushes a value into one. Both `getr` and `pushr` use the primitive register manipulation macro `set-register`.

Note that registers don't have to be declared. If `set-register` is sent a certain name, it will create a register with that name.

The register operators are all completely nondestructive. Cons, cons, cons, says `set-register`. This makes them slow and generates a lot of garbage, but, as explained on page 261, objects used in a part of a program where continuations are made should not be destructively modified. An object in one thread of control may be shared by another thread which is currently suspended. In this case, the registers found in one parse will share structure with the registers in many of the other parses. If speed became an issue, we could store registers in vectors instead of assoc-lists, and recycle used vectors into a common pool.

Push, cat, and jump arcs can all contain bodies of expressions. Ordinarily these will be just `setr`s. By calling `compile-cmds` on their bodies, the expansion functions of these arc types string a series of `setr`s into a single expression:

```
> (compile-cmds '((setr a b) (setr c d)))
(SETR A B (SETR C D REGS))
```

Each expression has the next expression inserted as its last argument, except the last, which gets `regs`. So a series of expressions in the body of an arc will be transformed into a single expression returning the new registers.

This approach allows users to insert arbitrary Lisp code into the bodies of arcs by wrapping it in a `progn`. For example:

```
> (compile-cmds '((setr a b)
                  (progn (princ "ek!"))
                  (setr c d)))
(SETR A B (PROGN (PRINC "ek!") (SETR C D REGS)))
```

Certain variables are left visible to code occurring in arc bodies. The sentence will be in the global `*sent*`. Two lexical variables will also be visible: `pos`, containing the current input pointer, and `regs`, containing the current registers. This is another example of intentional variable capture. If it were desirable to prevent the user from referring to these variables, they could be replaced with gensyms.

The macro `with-parses`, defined in Figure 23.5, gives us a way of invoking an ATN. It should be called with the name of a start node, an expression to be parsed, and a body of code describing what to do with the returned parses. The body of code within a `with-parses` expression will be evaluated once for each successful parse. Within the body, the symbol `parse` will be bound to the current parse. Superficially `with-parses` resembles operators like `dolist`, but underneath it uses backtracking search instead of simple iteration. A `with-parses` expression will return `@`, because that's what `fail` returns when it runs out of choices.

```
(defmacro with-parses (node sent &body body)
  (with-gensyms (pos regs)
    `(progn
       (setq *sent* ,sent)
       (setq *paths* nil)
       (=bind (parse ,pos ,regs) (,node 0 '(nil))
         (if (= ,pos (length *sent*))
             (progn ,@body (fail))
             (fail))))))
```

Figure 23.5: Toplevel macro.

Before going on to look at a more representative ATN, let's look at a parsing generated from the tiny ATN defined earlier. The ATN compiler (Figure 23.3) generates code which calls `types` to determine the grammatical roles of a word, so first we have to give it some definition:

```
(defun types (w)
  (cdr (assoc w '((spot noun) (runs verb)))))
```

Now we just call `with-parses` with the name of the start node as the first argument:

```
> (with-parses s '(spot runs)
    (format t "Parsing: ~A~%" parse))
Parsing: (SENTENCE (SUBJECT SPOT) (VERB RUNS))
@
```

23.5 A Sample ATN

Now that the whole ATN compiler has been described, we can go on to try out some parses using a sample network. In order to make an ATN parser handle a richer variety of sentences, you make the ATNs themselves more complicated, not the ATN compiler. The compiler presented here is a toy mainly in the sense that it's slow, not in the sense of having limited power.

The power (as distinct from speed) of a parser is in the grammar, and here limited space really will force us to use a toy version. Figures 23.8 through 23.11 define the ATN (or set of ATNs) represented in Figure 23.6. This network is just big enough to yield several parsings for the classic parser fodder "Time flies like an arrow."

Figure 23.6: Graph of a larger ATN.

```
(defun types (word)
  (case word
    ((do does did) '(aux v))
    ((time times) '(n v))
    ((fly flies) '(n v))
    ((like) '(v prep))
    ((liked likes) '(v))
    ((a an the) '(det))
    ((arrow arrows) '(n))
    ((i you he she him her it) '(pron))))
```

Figure 23.7: Nominal dictionary.

We need a slightly larger dictionary to parse more complex input. The function `types` (Figure 23.7) provides a dictionary of the most primitive sort. It defines a 22-word vocabulary, and associates each word with a list of one or more simple grammatical roles.

```
(defnode mods
  (cat n mods/n
    (setr mods *)))

(defnode mods/n
  (cat n mods/n
    (pushr mods *))
  (up `(n-group ,(getr mods))))
```

Figure 23.8: Sub-network for strings of modifiers.

The components of an ATN are themselves ATNS. The smallest ATN in our set is the one in Figure 23.8. It parses strings of modifiers, which in this case means just strings of nouns. The first node, `mods`, accepts a noun. The second node, `mods/n`, can either look for more nouns, or return a parsing.

Section 3.4 explained how writing programs in a functional style makes them easier to test:

1. In a functional program, components can be tested individually.
2. In Lisp, functions can be tested interactively, in the toplevel loop.

Together these two principles allow *interactive development:* when we write functional programs in Lisp, we can test each piece as we write it.

ATNS are so like functional programs—in this implementation, they macroexpand *into* functional programs—that the possibility of interactive development applies to them as well. We can test an ATN starting from any node, simply by giving its name as the first argument to `with-parses`:

```
> (with-parses mods '(time arrow)
    (format t "Parsing: ~A~%" parse))
Parsing: (N-GROUP (ARROW TIME))
@
```

The next two networks have to be discussed together, because they are mutually recursive. The network defined in Figure 23.9, which begins with the node `np`, is used to parse noun phrases. The network defined in Figure 23.10 parses prepositional phrases. Noun phrases may contain prepositional phrases and vice versa, so the two networks each contain a push arc which calls the other.

The noun phrase network contains six nodes. The first node, `np` has three choices. If it reads a pronoun, then it can move to node `pron`, which pops out of the network:

```
(defnode np
  (cat det np/det
    (setr det *))
  (jump np/det
    (setr det nil))
  (cat pron  pron
    (setr n *)))

(defnode pron
  (up `(np (pronoun ,(getr n)))))

(defnode np/det
  (down mods np/mods
    (setr mods *))
  (jump np/mods
    (setr mods nil)))

(defnode np/mods
  (cat n np/n
    (setr n *)))

(defnode np/n
  (up `(np (det ,(getr det))
           (modifiers ,(getr mods))
           (noun ,(getr n))))
  (down pp np/pp
    (setr pp *)))

(defnode np/pp
  (up `(np (det ,(getr det))
           (modifiers ,(getr mods))
           (noun ,(getr n))
           ,(getr pp))))
```

Figure 23.9: Noun phrase sub-network.

```
> (with-parses np '(it)
    (format t "Parsing: ~A~%" parse))
Parsing: (NP (PRONOUN IT))
@
```

```
(defnode pp
  (cat prep pp/prep
    (setr prep *)))

(defnode pp/prep
  (down np pp/np
    (setr op *)))

(defnode pp/np
  (up `(pp (prep ,(getr prep))
           (obj ,(getr op)))))
```

Figure 23.10: Prepositional phrase sub-network.

Both the other arcs lead to node `np/det`: one arc reads a determiner (e.g. "the"), and the other arc simply jumps, reading no input. At node `np/det`, both arcs lead to `np/mods`; `np/det` has the option of pushing to sub-network `mods` to pick up a string of modifiers, or jumping. Node `np-mods` reads a noun and continues to `np/n`. This node can either pop a result, or push to the prepositional phrase network to try to pick up a prepositional phrase. The final node, `np/pp`, pops a result.

Different types of noun phrases will have different parse paths. Here are two parsings on the noun phrase network:

```
> (with-parses np '(arrows)
    (pprint parse))
(NP (DET NIL)
    (MODIFIERS NIL)
    (NOUN ARROWS))
@
> (with-parses np '(a time fly like him)
    (pprint parse))
(NP (DET A)
    (MODIFIERS (N-GROUP TIME))
    (NOUN FLY)
    (PP (PREP LIKE)
        (OBJ (NP (PRONOUN HIM)))))
@
```

The first parse succeeds by jumping to `np/det`, jumping again to `np/mods`, reading a noun, then popping at `np/n`. The second never jumps, pushing first for

```
(defnode s
  (down np s/subj
    (setr mood 'decl)
    (setr subj *))
  (cat v v
    (setr mood 'imp)
    (setr subj '(np (pron you)))
    (setr aux nil)
    (setr v *)))

(defnode s/subj
  (cat v v
    (setr aux nil)
    (setr v *)))

(defnode v
  (up `(s (mood ,(getr mood))
          (subj ,(getr subj))
          (vcl (aux ,(getr aux))
               (v ,(getr v)))))
  (down np s/obj
    (setr obj *)))

(defnode s/obj
  (up `(s (mood ,(getr mood))
          (subj ,(getr subj))
          (vcl (aux ,(getr aux))
               (v ,(getr v)))
          (obj ,(getr obj)))))
```

Figure 23.11: Sentence network.

a string of modifiers, and again for a prepositional phrase. As is often the case with parsers, expressions which are syntactically well-formed are such nonsense semantically that it's difficult for humans even to detect the syntactic structure. Here the noun phrase "a time fly like him" has the same form as "a Lisp hacker like him."

Now all we need is a network for recognizing sentence structure. The network shown in Figure 23.11 parses both commands and statements. The start node is conventionally called `s`. The first node leaving it pushes for a noun phrase,

```
> (with-parses s '(time flies like an arrow)
    (pprint parse))

(S (MOOD DECL)
   (SUBJ (NP (DET NIL)
             (MODIFIERS (N-GROUP TIME))
             (NOUN FLIES)))
   (VCL (AUX NIL)
        (V LIKE))
   (OBJ (NP (DET AN)
            (MODIFIERS NIL)
            (NOUN ARROW))))

(S (MOOD IMP)
   (SUBJ (NP (PRON YOU)))
   (VCL (AUX NIL)
        (V TIME))
   (OBJ (NP (DET NIL)
            (MODIFIERS NIL)
            (NOUN FLIES)
            (PP (PREP LIKE)
                (OBJ (NP (DET AN)
                         (MODIFIERS NIL)
                         (NOUN ARROW)))))))
@
```

Figure 23.12: Two parsings for a sentence.

which will be the subject of the sentence. The second outgoing arc reads a verb. When a sentence is syntactically ambiguous, both arcs could succeed, ultimately yielding two or more parsings, as in Figure 23.12. The first parsing is analogous to “Island nations like a navy,” and the second is analogous to “Find someone like a policeman.” More complex ATNs are able to find six or more parsings for “Time flies like an arrow.”

The ATN compiler in this chapter is presented more as a distillation of the idea of an ATN than as production software. A few obvious changes would make this code much more efficient. When speed is important, the whole idea of simulating nondeterminism with closures may be too slow. But when it isn’t essential, the programming techniques described here lead to very concise programs.