Earley Parsing Explained

Earley parsers are among the most general parsers out there. They can parse any context free language without restriction, and can even be extended towards context sensitivity. They are reasonably fast on most practical grammars, and are easy to implement (the core algorithms take less than 200 lines of code).

On the other hand, most of the information I found about them is encoded in cryptic academese. Deciphering it is hard for non-experts (it was certainly hard for *me*).

This tutorial is mostly aimed at the curious and the implementer. It tries to convey an intuitive understanding of Earley parsing. Hard core theorists seeking deep math should read the papers referenced in this tutorial. Here, I just want to help you write your own Earley parsing framework.

Prerequisites

Unfortunately, I can't assume zero knowledge. To understand this tutorial, you will need a good grasp of the vocabulary around formal grammars and the Chomksky hierarchy. Notions of automata theory can also be useful.

At the very least, you should be able to implement a recursive descent parser for a simple language, such as arithmetic expressions. If you have never written one, do so now.

Table of contents

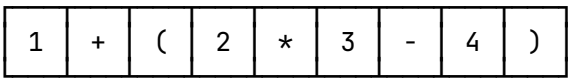
What is Earley parsing, and why you should care?

What

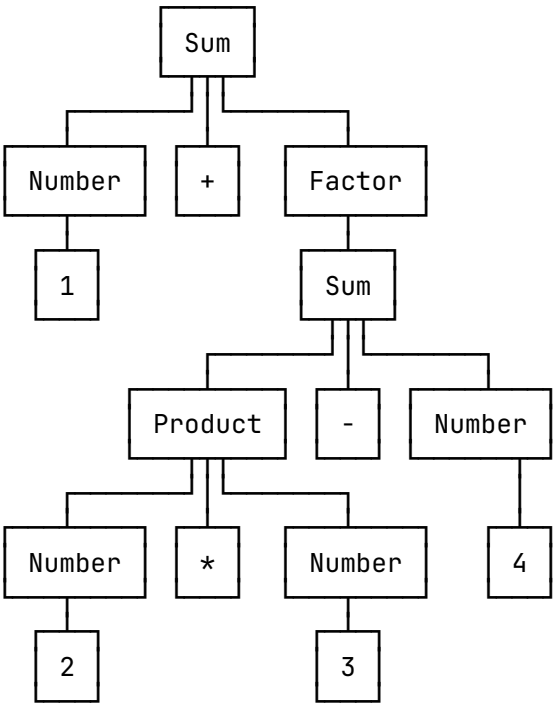
Like most parsers, Earley parsers work with a specification…

```
Sum     = Sum     [+-] Product
        | Product
Product = Product [*/] Factor
        | Factor
Factor  = '(' Sum ')'
        | Number
Number  = [0-9]+
```

…to turn some input…



…into a nicely structured tree.



So far, nothing special about them.

Why

The biggest advantage of Earley Parsing is its *accessibility*. Most other tools such as parser generators, parsing expression grammars, or combinator libraries feature restrictions that often make them hard to use. Use the wrong kind of grammar, and your PEG will enter an infinite loop. Use *another* wrong kind of grammar, and most *parser generators* will fail. To a beginner, these restrictions feel most arbitrary: it looks like it should work, but it doesn't. There are workarounds of course, but they make these tools more complex.

Earley parsing Just Works™.

On the flip side, to get this generality we must sacrifice some speed. Earley parsing cannot compete with speed demons such as Flex/Bison in terms of raw speed. It's not that bad, however:

- Earley parsing is cubic in the worst cases, which is the state of the art (and possibly the best we can do). The speed demons often don't work *at all* for those worst cases. Other parsers are prone to exponential combinatorial explosion.
- Most simple grammars can be parsed in linear time.
- Even the worst unambiguous grammars can be parsed in quadratic time.

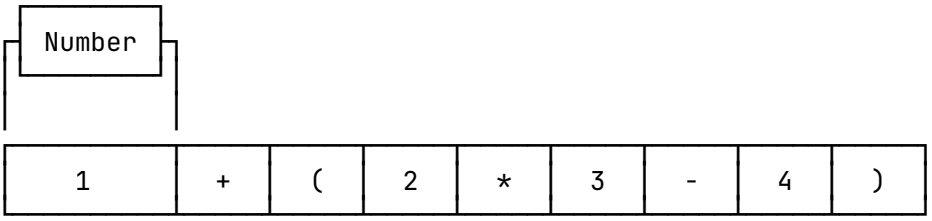
My advice would be to use Earley parsing by default, and only revert to more specific methods if performance is an issue…

…which is what I would like to say. For now we lack the tools. My goal is to get you to write them.

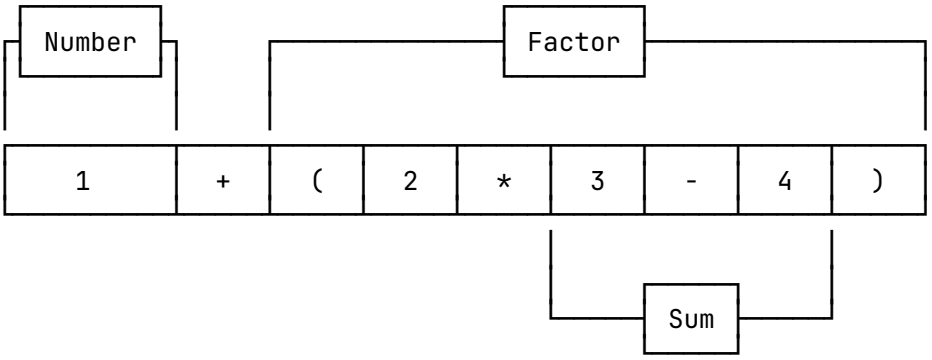
Earley Parsing Explained — Chart Parsing

We can’t parse the whole input all at once and magically get a full parse tree from the input. We have to work it out bit by bit somehow. One way to do it is to rely on induction to construct the tree directly. That’s how recursive descent parsers work.

Another way to do it is to construct a list of **partial parses**. Here’s one for instance:



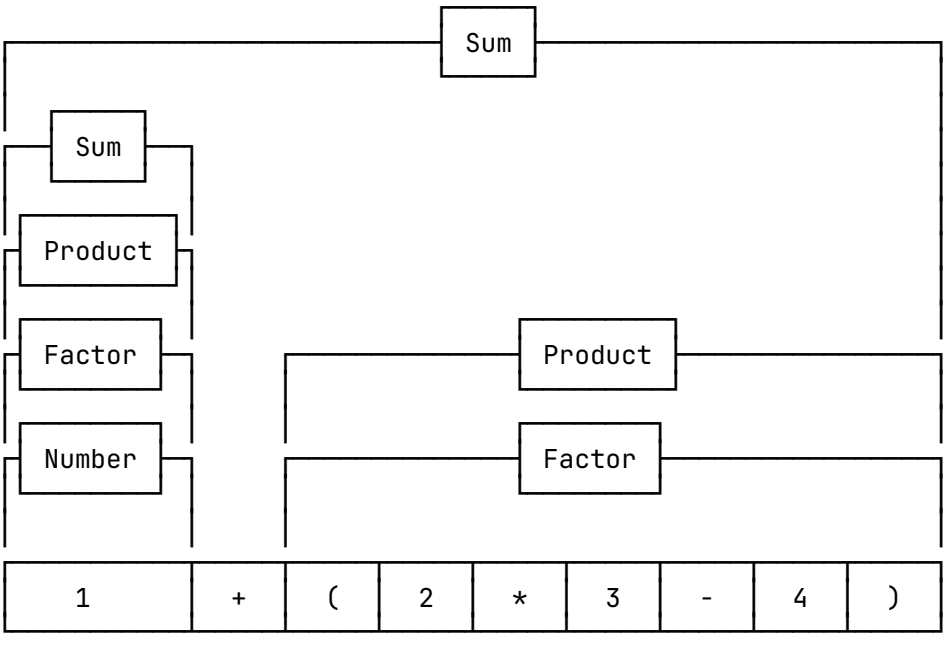
Here we just parsed the first character, a number. Here’s more:



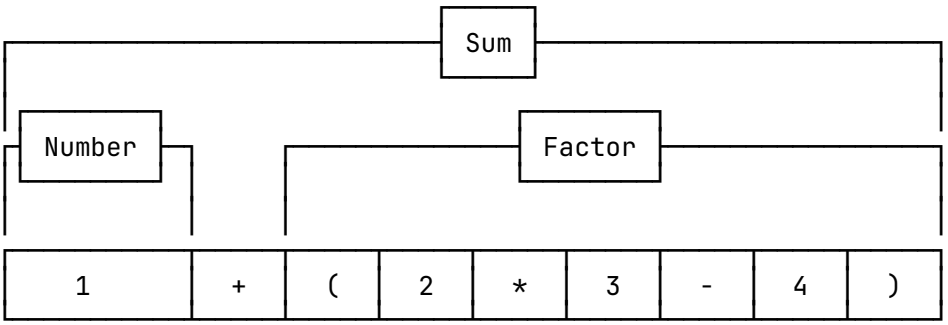
Now with a bit more work, we can construct a complete parse. Recall the following grammar:

```
Sum     = Sum     [+-] Product
        | Product
Product = Product [*/] Factor
        | Factor
Factor  = '(' Sum ')'
        | Number
Number  = [0-9]+
```

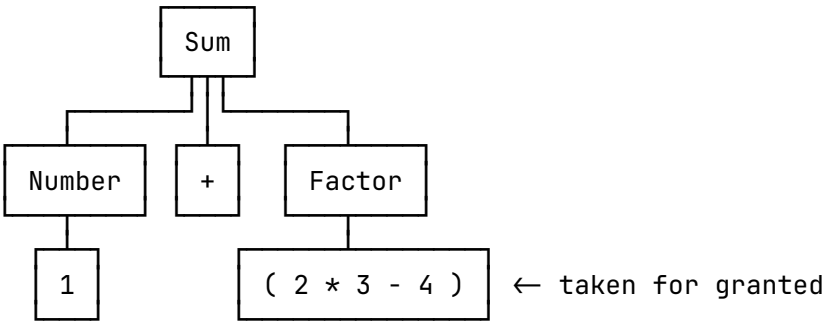
We can use it to find all the partial parses we need, including one that spans the whole input. Here are some:



Note that I take the `Factor` at the lower right for granted. A real parser would of course dig deeper before determining that this sub-string is indeed a “factor”. Now let’s remove some useless fluff for ease of reading:

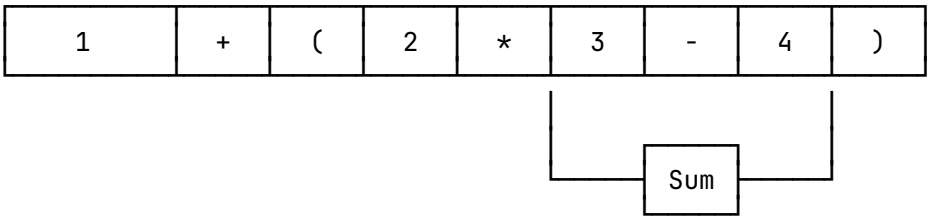


When you think about it, this is just another way of representing the following parse tree:



Useless work

All partial parses are not useful. Remember this one I showed earlier?



Taken in isolation, the sub-string `3-4` is indeed a `Sum` (a difference counts as a `Sum` in my grammar). But when you add a bit of context you can see this is a dead end: `2*3-4` reads (2×3)−4, not 2×(3−4). So, as correct as it is, this partial parse is useless. In practice many are, and some parsers waste too much time on them.

When you think about it, it doesn’t make sense to even try to parse `3-4` as a `Sum`: it follows a star (`*`), and the grammar says that stars are followed by factors, not by sums.

Here lies Earley’s genius: his parsers avoid most unnecessary work by not even trying a whole slew of hopeless partial parses like this one. This makes it faster than many alternatives.

Earley Parsing Explained — The Recogniser

(The impatient may download the source code.)

We will use this grammar:

```
Sum     = Sum     [+-] Product
        | Product
Product = Product [*/] Factor
        | Factor
Factor  = '(' Sum ')'
        | Number
Number  = [0-9]+
```

However, Earley parsers don't recognise EBNF grammars directly. We need to use this restricted syntax:

```
Sum     → Sum     [+-] Product
Sum     → Product
Product → Product [*/] Factor
Product → Factor
Factor  → '(' Sum ')'
Factor  → Number
Number  → [0-9] Number
Number  → [0-9]
```

Don't worry, this notation is just as powerful as the full EBNF. We will automate the translation process later.

Vocabulary

Grammars, rules, and symbols

For the purposes of Earley parsing, a *grammar* is a set of *rules*. Here is a rule:

```
Product → Product [*/] Factor
```

The left hand side of a rule (`Product` in this example) is called a *non-terminal symbol*. The right hand side (`Product [*/] Factor` in this example) is a list of symbols: non-terminal symbols (`Product` and `Factor` in this example), and *terminal symbols* (`[*/]` in this example).

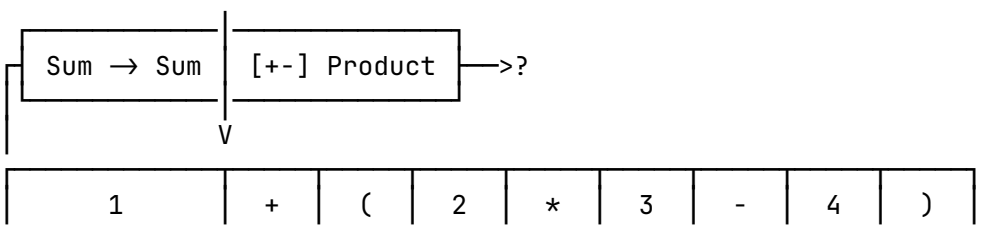
A terminal symbol is a function from characters to booleans. It tells you if a given character matches. Popular short hands for such functions look like `'x'` (matches the character 'x' only), or `[+-*/]` (matches '+', '-', '*', and'/'), or `[0-9]` (matches all numbers).

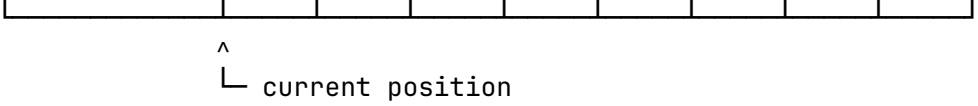
Earley items

I said previously that chart parsers construct lists of partial parses. Earley parsers are no exception. They just call those partial parses *Earley items*. Here is an Earley item:

```
Sum → Sum • [+-] Product  (0)
```

It is just like a grammar rule, with a couple more things: a fat dot somewhere in the right hand side, and a number at the far right. And of course, it represents a partial parse. Here is another way to view this item:





- The number between the parentheses represents where the item *starts*. `(0)` means it starts at the beginning of the input.
- The fat dot represent *how much* has been parsed so far. Whatever lies on its left is *done*. Whatever lies on its right has yet to be tested. In this example, the very next thing to test is the terminal token `[+-]`.

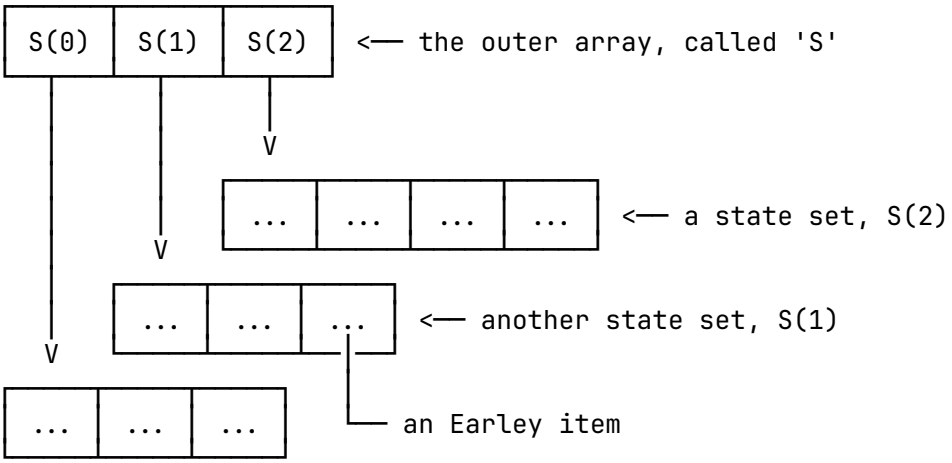
The astute reader may have noticed a key piece of information that I have represented in the chart, yet is conspicuously absent from the item itself: the current position. This information is crucial, but Earley's algorithm doesn't need the items to store their own current position. That's because all the items with the same current position are stored together in a *state set*. (I don't know why nobody calls them "item sets" instead. Whatever.) That state set is itself stored in an array. Its position in that array denotes the current position of the Earley items.

If you are able to look at a particular item, you know its current position, because that's how you found it in the first place.

State sets

State sets are kind of weird. They are sets all right, in the sense that they don't contain multiple copies of the same item. On the other hand, for the purpose of Earley's algorithm, it is easier to think of them as dynamic arrays.

Those state sets are stored in a big dynamic array, generally simply called 'S'. Here is a possible representation.

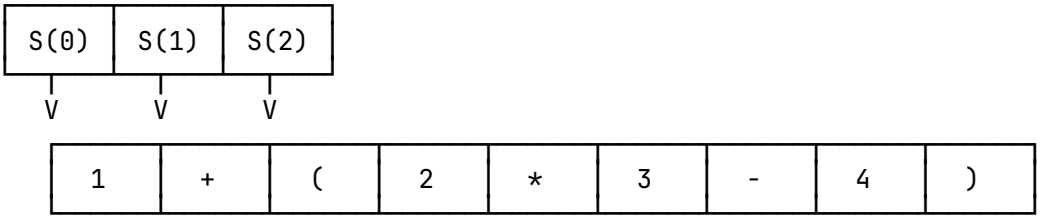


(We can do nice optimisations about storing the state sets contiguously in memory, but let's ignore that for now. New state set, new array.)

The big array and the input

Earley parsers are incremental parsers: as soon as they have read a token, they gather as much information as they can about the partial parses that make sense. Concretely, this means the big array, S, is constructed piece by piece, with one element per input token (plus one initial element).

Here is what S looks like when we have read the first two tokens.

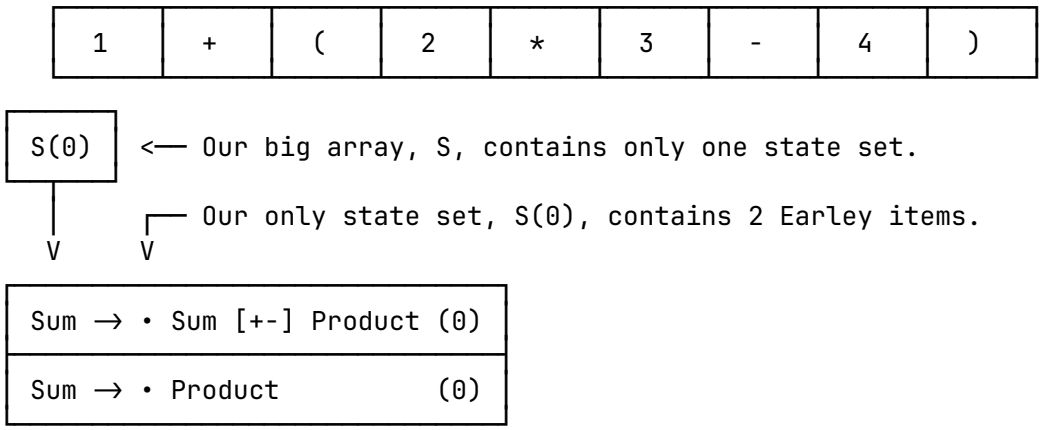


The arrows here indicate the current position of each state set in S. Those positions are *between* tokens. Which makes sense, considering a character is either read, or not read. When `S(2)` is created, it means we have read the first 2 characters.

`S(0)` is created before we read anything. That's the initial state. When we will be finished with this input, if all goes well, we will have created 10 state sets, up to `S(9)`. This last state set is created when we have read everything.

Start up

The parsing starts by initialising the first state set. We typically chose the rules by name. In our "hello world", that would be every rule named `Sum`.



Note how I produced the items:

- We start at the beginning, and have read no input yet. The current position is therefore `(0)`. I put the item in `S(0)`.
- The items are brand new So, their start position is the same as the current position: `(0)`. Which you can see at the right of both items.
- As brand new items, nothing has been parsed yet. Everything is yet to be done (or tested). We put the fat dot at the very beginning of the item.

Main loop

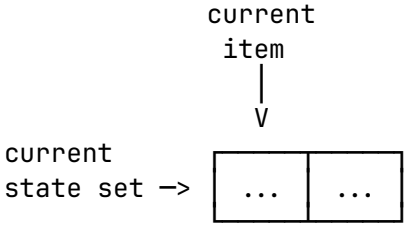
Now that we have initialised our data, we can loop over it. There are two loops to consider: an inner loop, which loops over a single state set, and an outer loop, which loops over the main array of state set.

Inner loop

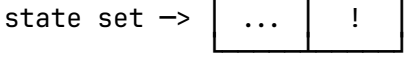
Basically, we go over each Earley item in the current state set. When we begin the algorithm, the current state set is `S(0)` (we'll do the rest later).

While going over the items, we may add even *more* items to either the current state, or the next one. (For the first inner loop, this means `S(0)` and `S(1)`). We continue until there are no more items in the current state set to process. This can go on for a while: whenever we add an item to the current state set, it will need to be processed just like the others.

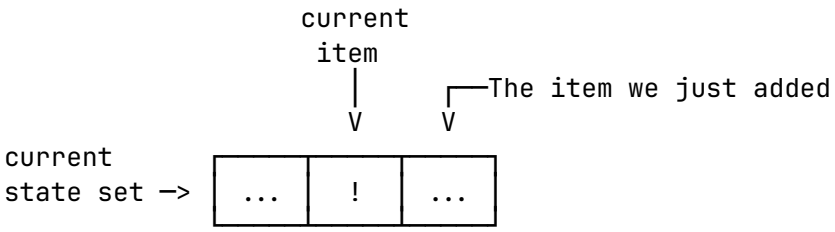
This looks like this:



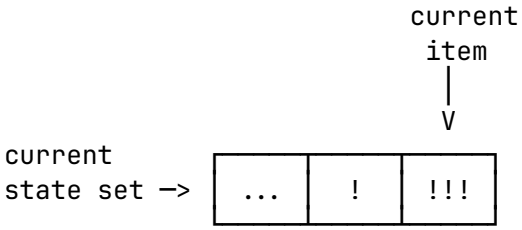
Nothing to add, let's try the next Earley item:



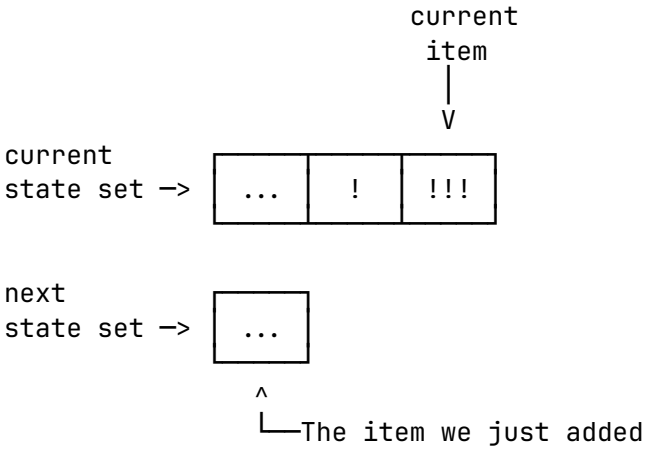
Ah, apparently, we need to add an item:



Done, now let’s go to the next item (there wasn’t one, but now there is).



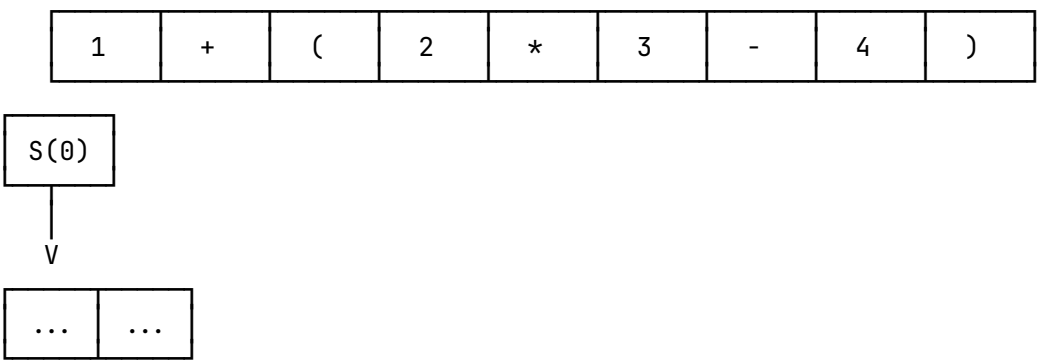
Ah, this one’s different. We need to add an item to the *next* state set! We create this state set if necessary, then add an Earley item to it.



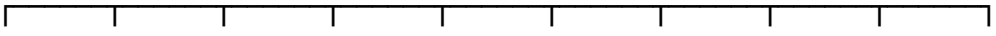
We have finished processing the current state, we can now process the next one (there wasn’t one originally, but now there is), and so on.

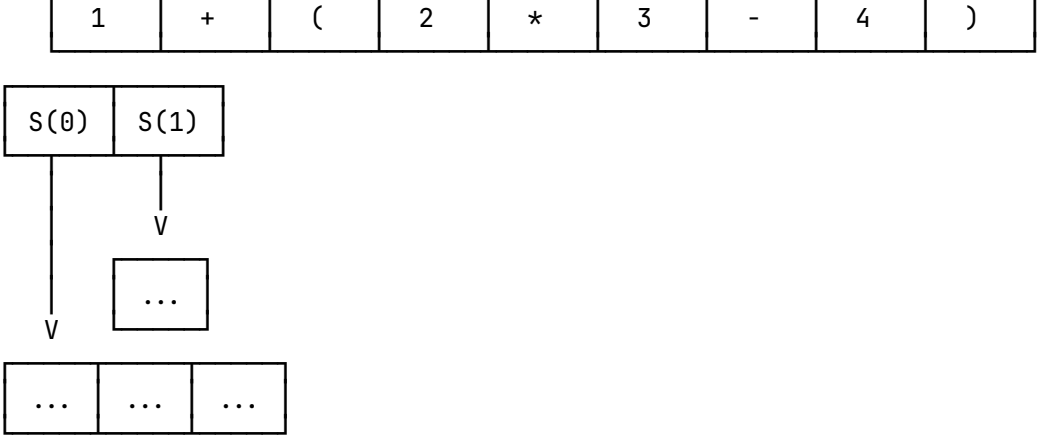
Outer loop

The outer loop works similarly: Run the inner loop over `S(0)`. Repeat the process with the next state set (which hopefully has just been created), and so on until there are no more state sets to process. Given a grammar and an input, we will start with something like this:

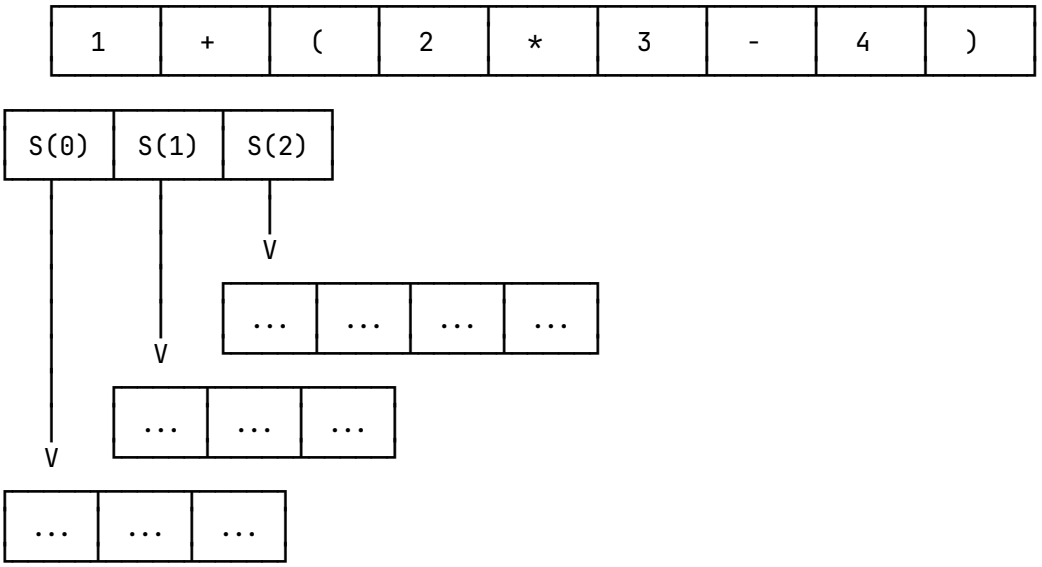


Then we run the inner loop over `S(0)`, generating some more Earley items and one state set in the process:

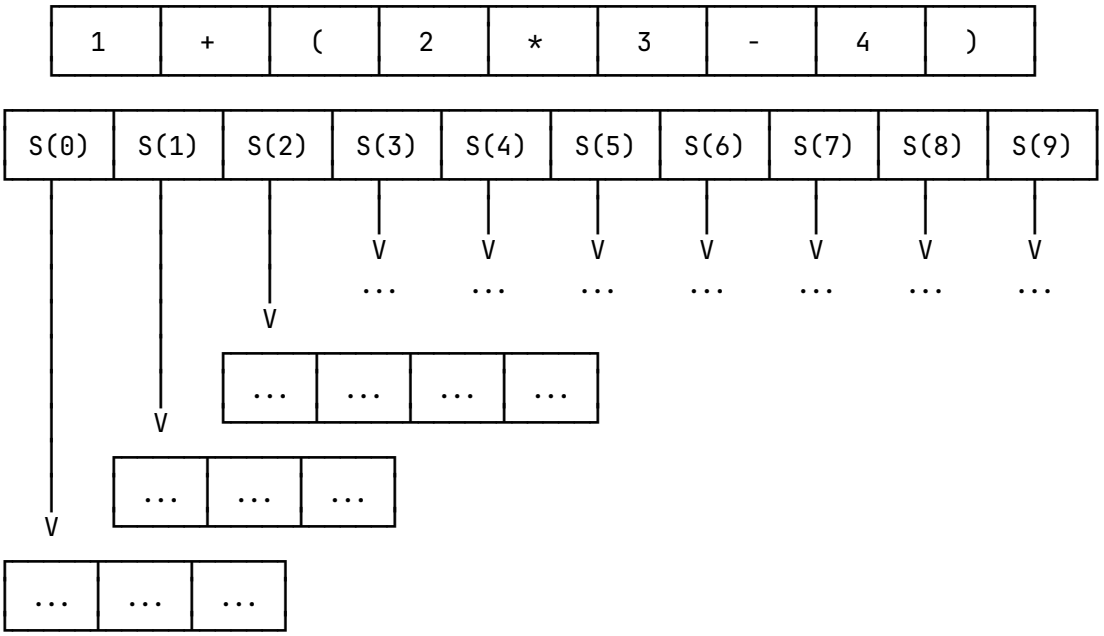




Now we run the inner loop over `S(1)`, and generate some more data:



And so on until we can't generate more state sets (either because we reached the end of the input, or because the input "stopped making sense"). I didn't represent all state sets, but you get the idea:



Prediction, Scan, and Completion

There are three specific ways of adding new Earley items to our data. Which way we choose depends on the item we are currently examining.

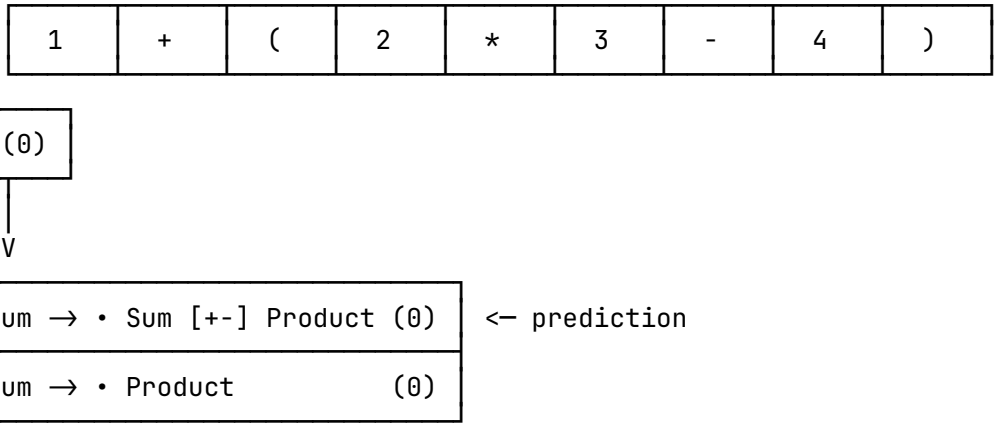
- **Prediction**. The symbol at the right of the fat dot is non-terminal. We add the the corresponding rules to the

current state set.

- **Scan**. The symbol at the right of the fat dot is terminal. We check if the input matches this symbol. If it does, we add this item (advanced one step) to the *next* state set.
- **Completion**. There is nothing at the right of the fat dot. This means we have a successful partial parse. We look for the parent items, and add them (advanced one step) to this state set.

Prediction

Let's go back to our initialised data. We must now examine the first Earley Item.



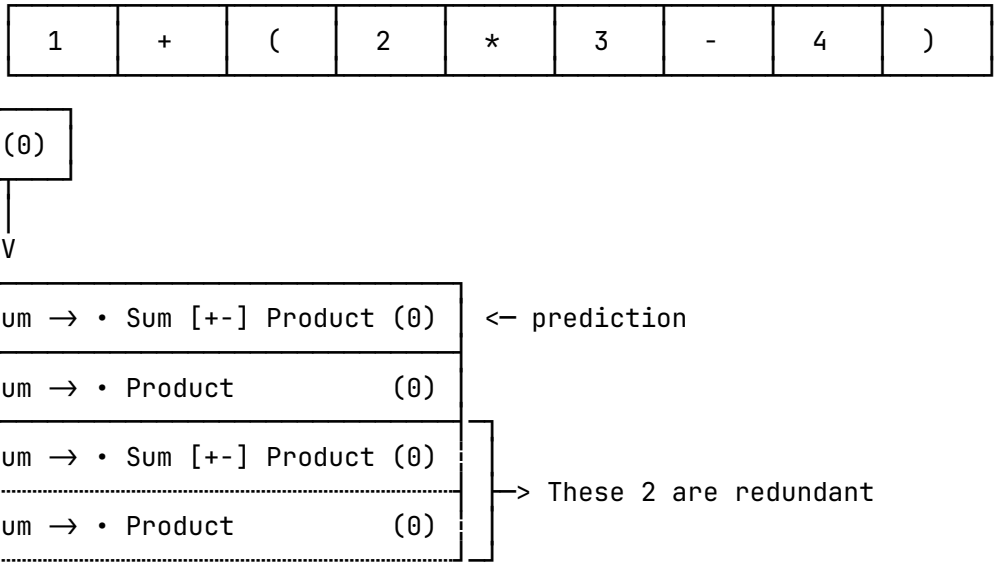
At the right of the fat dot, there is `Sum`, a non-terminal symbol. We must perform a prediction. Why? Because we're trying to parse the grammar rule embedded in this item. What's just after the fat dot? A `Sum`. Does this `Sum` actually begins by a `Sum`? Well, we have to try if we want to know. So we look up the Grammar, and see 2 rules named "Sum":

```
Sum → Sum [+-] Product
Sum → Product
```

This means we must add the following two items to `S(0)`:

```
Sum → • Sum [+-] Product (0)
Sum → • Product          (0)
       │                  │
       │                  └─ The item is predicted in S(0), so
       │                     we set the starting point at (0).
       │
       └─ We put the dot at the beginning of the rule.
```

Now we can add those items at the end of `S(0)`.



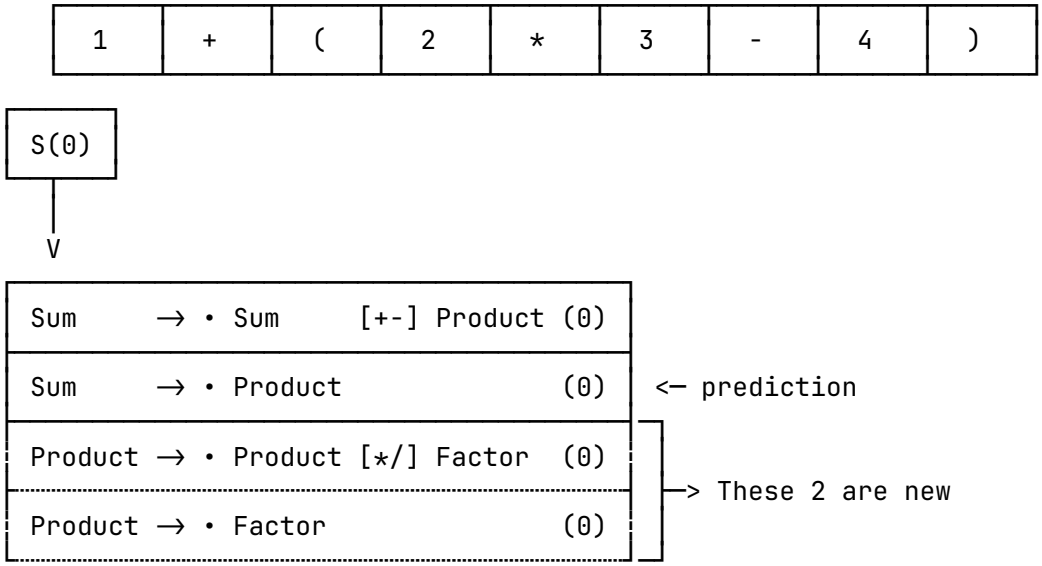
Now we have a problem: this redundancy will drop us in an infinite loop: since the first symbol of a rule named "Sum" is

also “Sum”, we will keep predicting sums until we run out of memory. This is a classic problem with most top-down parsers, and the reason why they can’t handle left-recursive grammars.

So we take one little precaution: *no duplicate*. The same items give the same results anyway, so why waste work? That way we avoid infinite loops —all of them. So don’t worry about left-recursive grammars, right-recursive grammars, magical grammars… they Just Work™.

(Okay, that was a lie. Some grammars still won’t work. We need another little precaution, which cannot be explained at this point.)

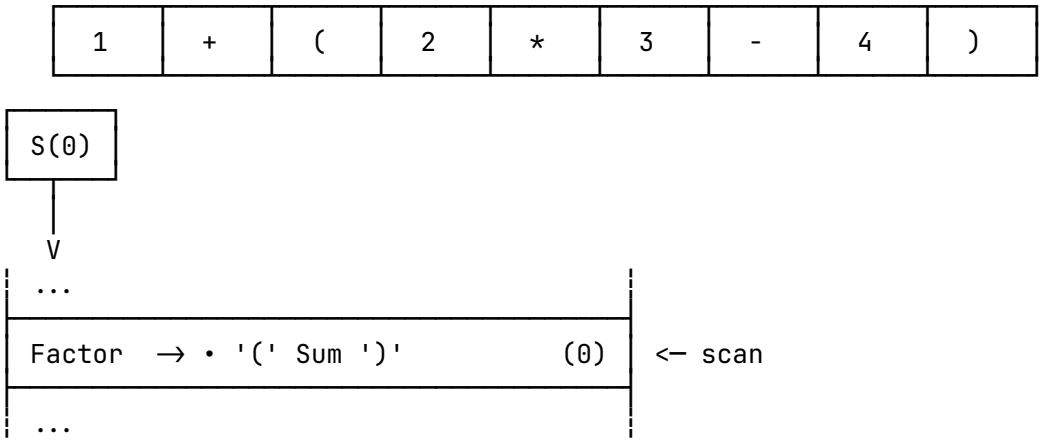
With this in mind, we go examine the next Earley item. We must still perform a prediction, this time over `Product`. (Basically, we’re trying to see if this `Sum` begins with a `Product`.)



Since those are not duplicates we actually add them to `S(0)`.

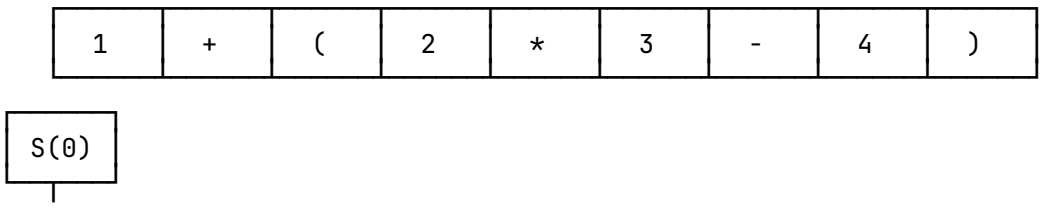
Scan

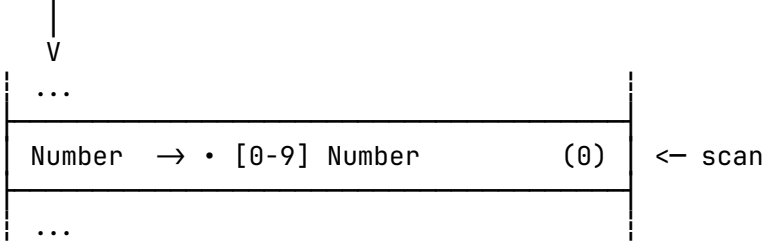
We keep looping over the items, predicting everything until we reach this Earley item:



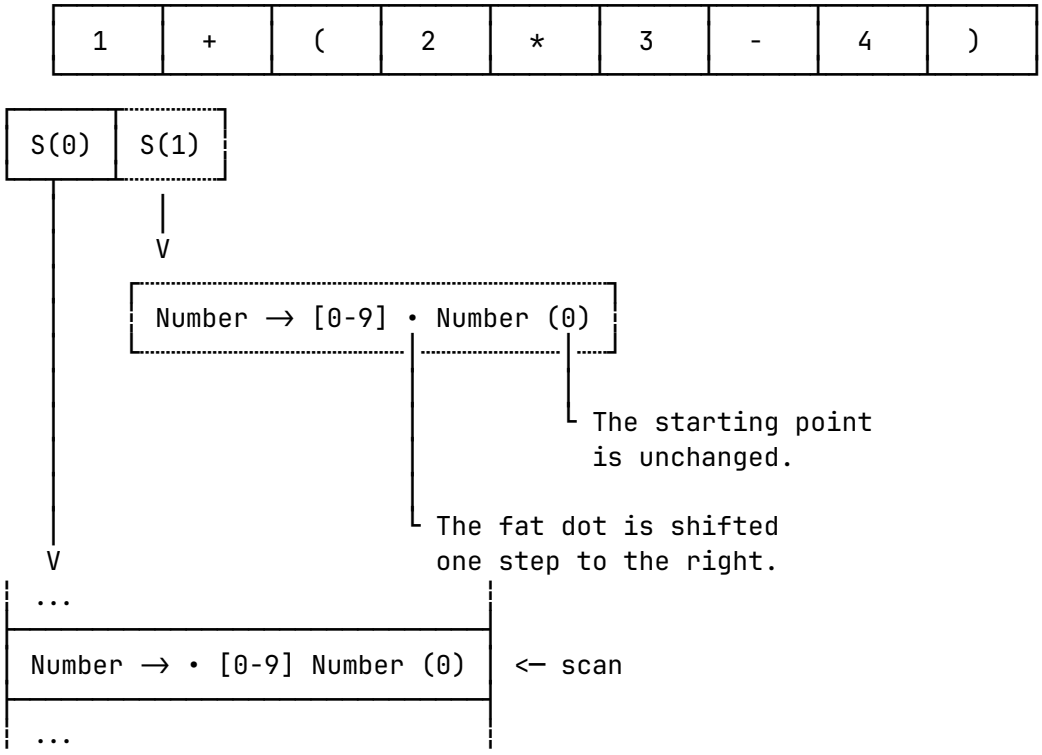
Now we can’t predict any more: at the right of the fat dot, there is a *terminal* symbol. How do we *try* a terminal symbol? We scan it. First we see if the terminal symbol (an open parens) applies to the current character (the digit ‘1’). Since ‘1’ is not an open parens, the scan fails. We continue the loop without doing anything.

The next scan looks like this:





Here, the non-terminal symbol at the right of the fat dot is `[0-9]`, and the digit ‘1’ does match that terminal symbol. Test successful! So, we copy the Earley item we’re examining over to the next state set, with a slight modification:

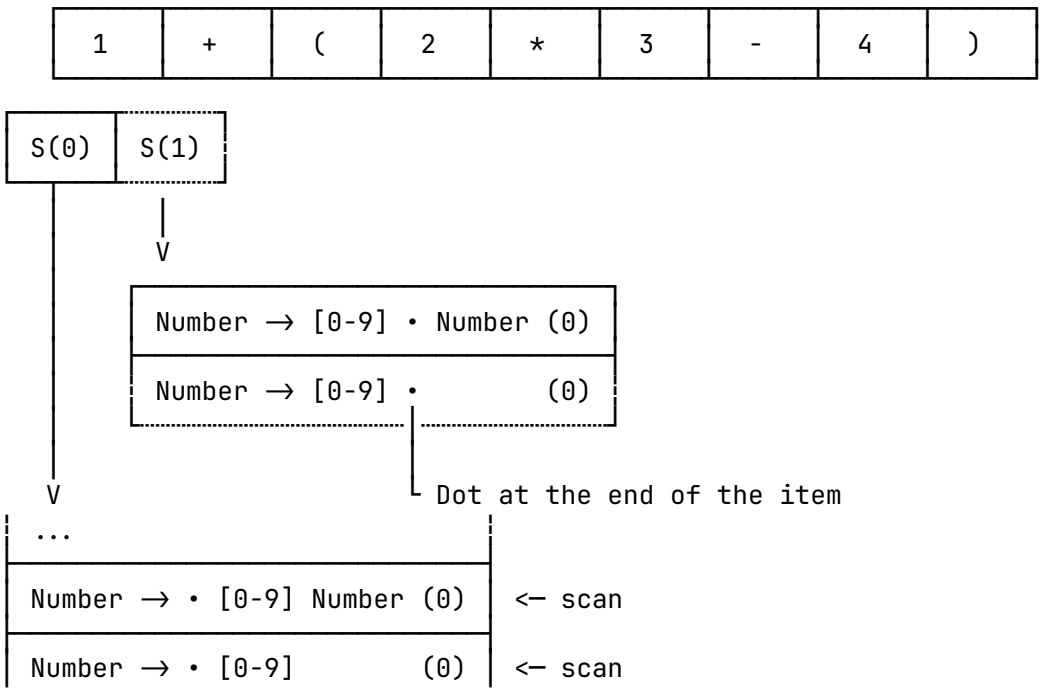


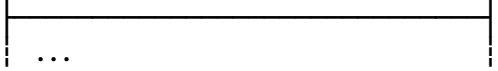
This time, we’ve done some significant work: we have tried many things, and this is our first success! At last, we managed to move a fat dot one step to the right. Now this is not a *confirmed* success, since the fat dot is not completely off to the right yet. But that’s a start.

A confirmed success will come soon though: there’s another item that can be scanned:

```
Number → • [0-9] Number (0)
```

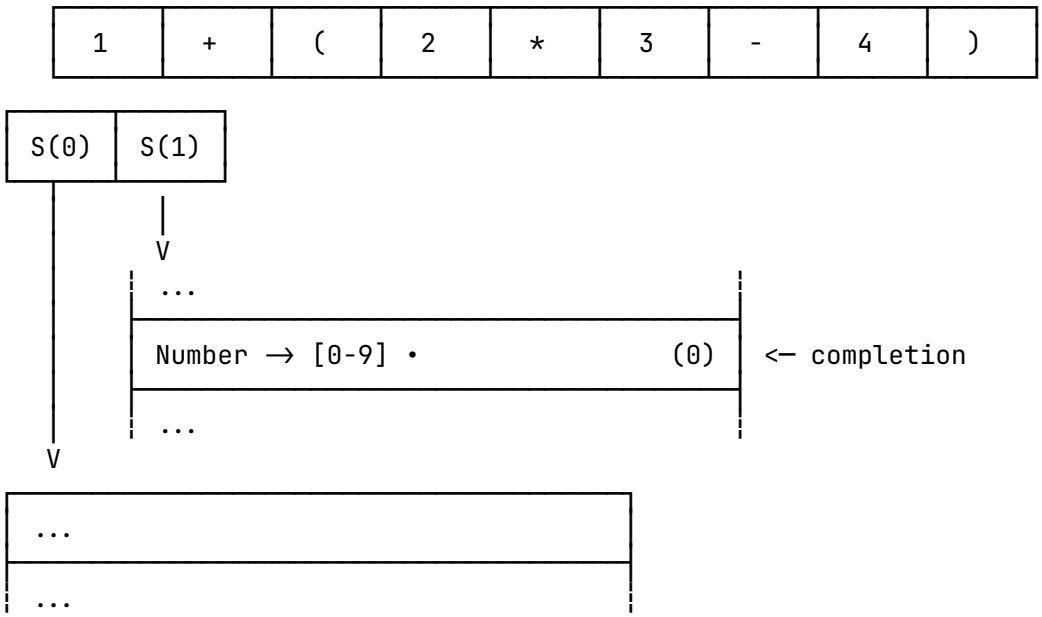
Which ultimately gives us:





Completion

We continue our main loop, finish `S(0)`, and process `S(1)`. At some point, we get to this:



Here there is no symbol at the right side of the fat dot. At this point, things get *real* interesting: we have a successful partial parse. An actual, confirmed success: between the starting point of the item (0), and its current position (1), the string `"1"` is a `Number`.

We do not stop there however: This `Number` may be a fluke, but it may also be part of something bigger. Actually, Earley's algorithm pretty much *guarantees* it will be part of something bigger. This item doesn't come from nowhere: there's a whole chain of predictions, scans, and completions to get there.

Imagine we have an item like this ('a', 'b', and 'c' are symbols, and 'i' is an integer):

```
Rule → a b c •  (i)
```

The fact that this item even *exist* means the following items also exist somewhere:

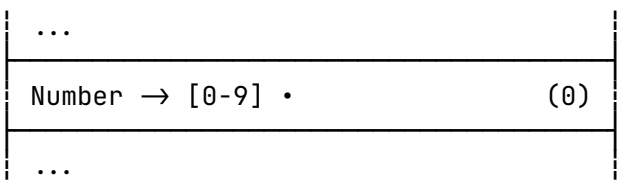
```
Foo →   a   b • c  (i)
Foo →   a • b   c  (i)
Foo → • a   b   c  (i)
```

And the last of them can only exist in `S(i)`: the state set where it was predicted. We're not interested in those items however. What we *really* want is the items that might have triggered the original prediction. *Might* have. Only one item actually triggered the original prediction, but there's often more than one path, and we want to capture them all. Anyway, those rules are the rules in `S(i)` that look like this:

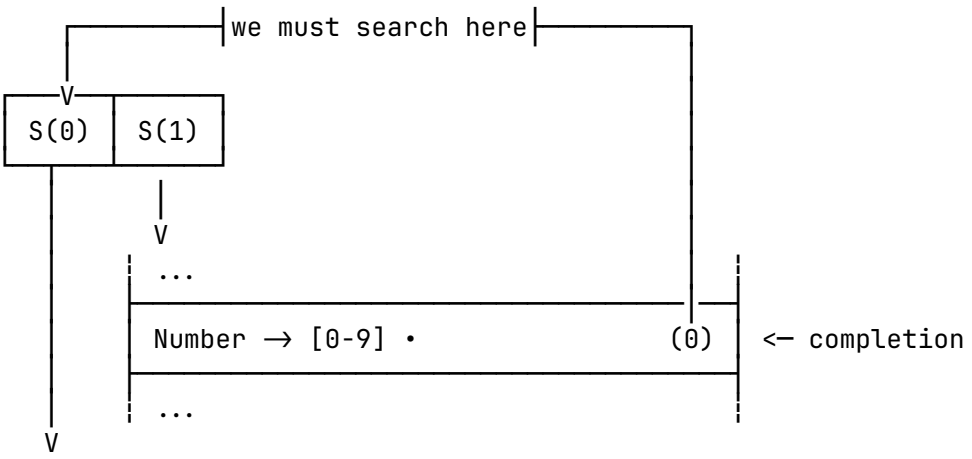
```
Bar → a b • Foo c d  (j)
```

That is, any rule in `S(i)` that has the rule `Foo` right next to the fat dot. *Those* are the "something" bigger `Foo` is a part of. Now the purpose of starting points should be obvious. With them, we can jump straight to the interesting state set, and retrieve the sets we want.

Enough with the side quest. We were triggering a completion with this item:

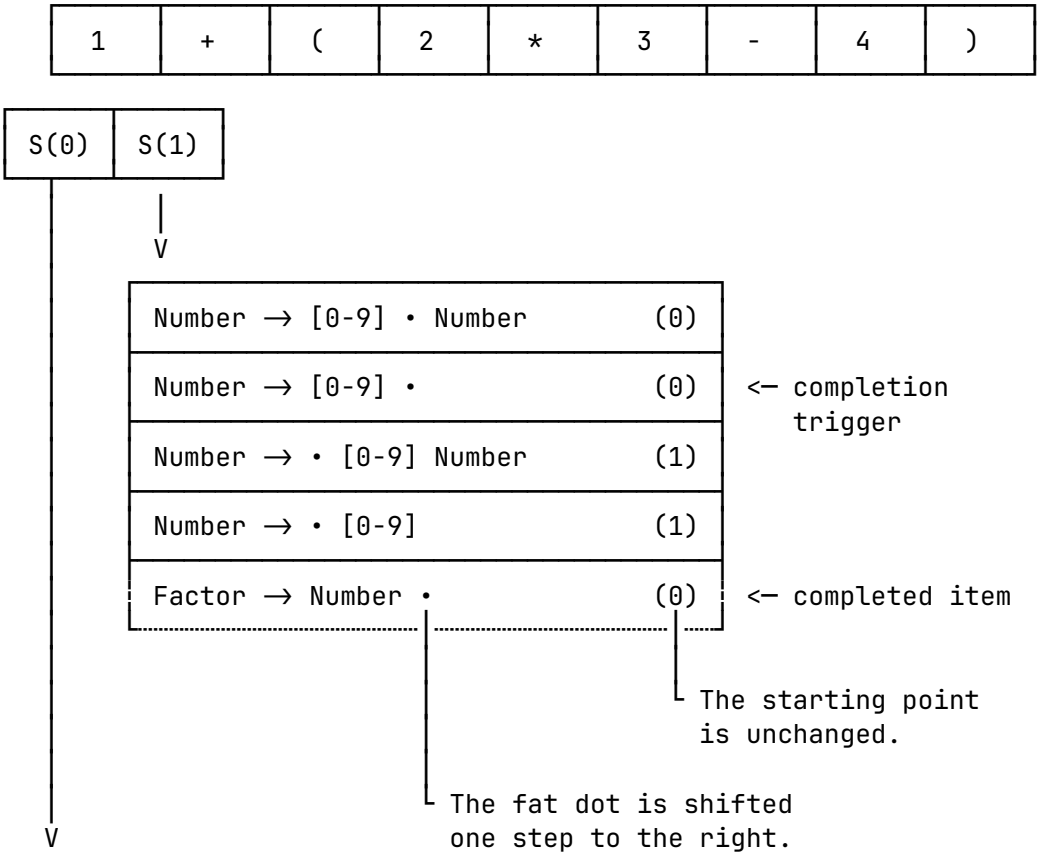


So, we search `S(0)` for any item that has `Number` right next to the dot:



Sum → • Sum [+-] Product (0)	Nope
Sum → • Product (0)	Nope
Product → • Product [*/] Factor (0)	Nope
Product → • Factor (0)	Nope
Factor → • '(' Sum ')' (0)	Nope
Factor → • Number (0)	FOUND ONE!
Number → • [0-9] Number (0)	Nope
Number → • [0-9] (0)	Nope

Now we just have to complete the items we have found. This is very similar to a successful scan: we copy all of the matching rules to the current state. The only difference is, we moved the fat dot over a *non-terminal* symbol:



Sum → • Sum [+-] Product (0)

Sum	→ • Product	(0)
Product	→ • Product [*/] Factor	(0)
Product	→ • Factor	(0)
Factor	→ • '(' Sum ')'	(0)
Factor	→ • Number	(0)
Number	→ • [0-9] Number	(0)
Number	→ • [0-9]	(0)

<- Item to complete

Note that there may be more than one item to complete (or even none at all, see the starting state). Here for instance, the item we just completed is *also* a completion trigger:

1	+	(	2	*	3	-	4	)

S(0)	S(1)

V

Number	→ [0-9] • Number	(0)
Number	→ [0-9] •	(0)
Number	→ • [0-9] Number	(1)
Number	→ • [0-9]	(1)
Factor	→ Number •	(0)
Product	→ Factor •	(0)

<- completion trigger

<- completed item

V

Sum	→ • Sum [+-] Product	(0)
Sum	→ • Product	(0)
Product	→ • Product [*/] Factor	(0)
Product	→ • Factor	(0)
Factor	→ • '(' Sum ')'	(0)
Factor	→ • Number	(0)
Number	→ • [0-9] Number	(0)
Number	→ • [0-9]	(0)

<- Item to complete

(And yes, `Product → Factor • (0)` is a completion trigger too.)

After the end

Eventually, the algorithm will stop, and you will end up with a whole slew of Earley items. for the input `1+(2*3-4)`, it

will be those:

```
═══ 0 ═══
Sum     → • Sum [+-] Product     (0)
Sum     → • Product              (0)
Product → • Product [*/] Factor  (0)
Product → • Factor               (0)
Factor  → • '(' Sum ')'          (0)
Factor  → • Number               (0)
Number  → • [0-9] Number         (0)
Number  → • [0-9]                (0)

═══ 1 ═══
Number  → [0-9] • Number         (0)
Number  → [0-9] •                (0)
Number  → • [0-9] Number         (1)
Number  → • [0-9]                (1)
Factor  → Number •               (0)
Product → Factor •               (0)
Sum     → Product •              (0)
Product → Product • [*/] Factor  (0)
Sum     → Sum • [+-] Product     (0)

═══ 2 ═══
Sum     → Sum [+-] • Product     (0)
Product → • Product [*/] Factor  (2)
Product → • Factor               (2)
Factor  → • '(' Sum ')'          (2)
Factor  → • Number               (2)
Number  → • [0-9] Number         (2)
Number  → • [0-9]                (2)

═══ 3 ═══
Factor  → '(' • Sum ')'          (2)
Sum     → • Sum [+-] Product     (3)
Sum     → • Product              (3)
Product → • Product [*/] Factor  (3)
Product → • Factor               (3)
Factor  → • '(' Sum ')'          (3)
Factor  → • Number               (3)
Number  → • [0-9] Number         (3)
Number  → • [0-9]                (3)

═══ 4 ═══
Number  → [0-9] • Number         (3)
Number  → [0-9] •                (3)
Number  → • [0-9] Number         (4)
Number  → • [0-9]                (4)
Factor  → Number •               (3)
Product → Factor •               (3)
Sum     → Product •              (3)
Product → Product • [*/] Factor  (3)
Factor  → '(' Sum • ')'          (2)
Sum     → Sum • [+-] Product     (3)

═══ 5 ═══
Product → Product [*/] • Factor  (3)
Factor  → • '(' Sum ')'          (5)
Factor  → • Number               (5)
Number  → • [0-9] Number         (5)
Number  → • [0-9]                (5)

═══ 6 ═══
Number  → [0-9] • Number         (5)
```

```
Number  → [0-9] •                (5)
Number  → • [0-9] Number         (6)
Number  → • [0-9]                (6)
Factor  → Number •               (5)
Product → Product [*/] Factor •  (3)
Sum     → Product •              (3)
Product → Product • [*/] Factor  (3)
Factor  → '(' Sum • ')'          (2)
Sum     → Sum • [+-] Product     (3)

=== 7 ===
Sum     → Sum [+-] • Product     (3)
Product → • Product [*/] Factor  (7)
Product → • Factor               (7)
Factor  → • '(' Sum ')'          (7)
Factor  → • Number               (7)
Number  → • [0-9] Number         (7)
Number  → • [0-9]                (7)

=== 8 ===
Number  → [0-9] • Number         (7)
Number  → [0-9] •                (7)
Number  → • [0-9] Number         (8)
Number  → • [0-9]                (8)
Factor  → Number •               (7)
Product → Factor •               (7)
Sum     → Sum [+-] Product •     (3)
Product → Product • [*/] Factor  (7)
Factor  → '(' Sum • ')'          (2)
Sum     → Sum • [+-] Product     (3)

=== 9 ===
Factor  → '(' Sum ')' •          (2)
Product → Factor •               (2)
Sum     → Sum [+-] Product •     (0)
Product → Product • [*/] Factor  (2)
Sum     → Sum • [+-] Product     (0)
```

Quite a mess. Now the big question is, have we made it? To know that, we must look at the last state set:

```
=== 9 ===
Factor  → '(' Sum ')' •          (2)
Product → Factor •               (2)
Sum     → Sum [+-] Product •     (0)
Product → Product • [*/] Factor  (2)
Sum     → Sum • [+-] Product     (0)
```

First this is the set number 9 (or the 10th set). The input has precisely 9 characters. So, first good news, the whole input made sense.

This is not enough however. We are looking for items that:

- are complete (the fat dot is at the end),
- have started at the beginning (state set 0),
- have the same name that has been chosen at the beginning (“Sum”).

There is one such item in S(9):

```
Sum → Sum [+-] Product •  (0)
```

We Win!

Failure modes

There are 2 ways to fail:

- We may fail to parse the whole input. This is because an invalid token caused it to stop making sense at this point. This is what happens when you try "`1+%`": the parse stops at the second character, fails at the third, then stops.
- We may parse the whole input, but fail to make a complete parse out of it. This is what happens when you try "`1+`". This is a valid beginning of a `Sum`, but the last part is missing, and we can't finish the parse.

In both cases, you get the following Earley items:

```
=== 0 ===
Sum     → • Sum [+-] Product     (0)
Sum     → • Product              (0)
Product → • Product [*/] Factor  (0)
Product → • Factor               (0)
Factor  → • '(' Sum ')'          (0)
Factor  → • Number               (0)
Number  → • [0-9] Number         (0)
Number  → • [0-9]                (0)

=== 1 ===
Number  → [0-9] • Number         (0)
Number  → [0-9] •                (0)
Number  → • [0-9] Number         (1)
Number  → • [0-9]                (1)
Factor  → Number •               (0)
Product → Factor •               (0)
Sum     → Product •              (0)
Product → Product • [*/] Factor  (0)
Sum     → Sum • [+-] Product     (0)

=== 2 ===
Sum     → Sum [+-] • Product     (0)
Product → • Product [*/] Factor  (2)
Product → • Factor               (2)
Factor  → • '(' Sum ')'          (2)
Factor  → • Number               (2)
Number  → • [0-9] Number         (2)
Number  → • [0-9]                (2)
```

(We get the same Earley items because this grammar makes no difference between an input that stops making sense, and an input that just stops. To make that difference, we need a rule that uses an "end of input" token explicitly.)

Note that in both examples, there *is* a valid parse: the number `1`, which is a `Sum` by itself. This is evidenced by the presence of this Earley item in `S(1)`:

```
Sum → Product •  (0)
```

I say this because in some cases, you just want to parse a header. Of course the input will stop making sense at some point.

Let's do this!

You should now be able to implement a Lua recogniser. It will work for any grammar, except for *one important limitation*: It handles empty rules badly. We'll correct this bug next time.

In the mean time, here is a recogniser in Lua. To use it, just run the lua interpreter on it in the command line, like this:

```
$ lua recogniser.lua
```

It has been tested with lua 5.2.3 on a Debian machine. It should work on 5.1 as well, on any machine.

If you don't know Lua, don't worry, it mostly looks like pseudocode. Use it at your own peril though: there is still that

bug about empty rules.

Source code

You should be able to read it even if you don't know Lua (I did my best to avoid Lua specific idioms). The only tricky bit lie in the tables.

First, an example grammar:

```
local Grammar = {
   start_rule_name = 'Sum',
   { name = 'Sum'    , 'Sum'      , class('+-'), 'Product'},
   { name = 'Sum'    , 'Product'  ,                        },
   { name = 'Product', 'Product'  , class('*/'), 'Factor' },
   { name = 'Product', 'Factor'   ,                        },
   { name = 'Factor' , char('(')  , 'Sum', char(')')      },
   { name = 'Factor' , 'Number'                            },
   { name = 'Number' , range('09'), 'Number'               },
   { name = 'Number' , range('09'),                        },
}
```

then how to use it:

```
local input = "1+(2*3+4)"
local S     = build_items(Grammar, input)
print_S(S, Grammar)              -- print all the internal state
diagnose(S, Grammar, input)      -- tell if the input is OK or not
```

Now the main loop:

```
local function build_items(grammar, input)
   -- Earley sets
   local S = {{}}
   -- put start item(s) in S[1]
   for i = 1, #grammar do
      if grammar[i].name == grammar.start_rule_name then
         unsafe_append(S[1], { rule  = i,
                               start = 1,
                               next  = 1 })
      end
   end
   -- populate the rest of S[i]
   local i = 1
   while i ⩽ #S do
      local j = 1
      while j ⩽ #S[i] do
         local symbol = next_symbol(grammar, S[i][j])
         if     type(symbol) == "nil"      then complete(S, i, j, grammar)
         elseif type(symbol) == "function" then scan    (S, i, j, symbol, input)
         elseif type(symbol) == "string"   then predict (S, i,    symbol, grammar)
         else error("illegal rule")
         end
         j = j + 1
      end
      i = i + 1
   end
   return S
end
```

The prediction step:

```
local function predict(S, i, symbol, grammar)
   for rule_index, rule in ipairs(grammar) do
```

```
      if rule.name == symbol then
         append(S[i], { rule  = rule_index,
                        next  = 1 ,
                        start = i })
      end
   end
end
```

The scan step:

```
local function scan(S, i, j, symbol, input)
   local item = S[i][j]
   if symbol(input, i) then -- terminal symbols are predicates
      if S[i+1] == nil then S[i+1]  = {} end
      unsafe_append(S[i+1], { rule  = item.rule,
                              next  = item.next + 1,
                              start = item.start })
   end
end
```

The completion step:

```
local function complete(S, i, j, grammar)
   local item = S[i][j]
   for old_item_index, old_item in ipairs(S[item.start]) do
      if next_symbol(grammar, old_item) == name(grammar, item) then
         append(S[i], { rule  = old_item.rule,
                        next  = old_item.next + 1,
                        start = old_item.start })
      end
   end
end
```

Some utility functions:

```
-- next element in the rule of this item
local function next_symbol(grammar, item)
   return grammar[item.rule][item.next]
end

-- gets the name of the rule pointed by the item
local function name(grammar, item)
   return grammar[item.rule].name
end

-- compares two items for equality (needed for safe append)
local function equal(item1, item2)
   return item1.rule  == item2.rule
      and item1.start == item2.start
      and item1.next  == item2.next
end

-- Adds an item at the end of the Earley set
local function unsafe_append(set, item)
   set[#set+1] = item
end

-- Adds an item at the end of the Earley set, **unless already present**
local function append(set, item)
   for i = 1, #set do
      if equal(item, set[i]) then
         return
      end
   end
   unsafe_append(set, item) -- the item wasn't already there, we add it
```

```
end
```

That should be enough. In any case, the full source code is still available.

Earley Parsing Explained — Empty Rules

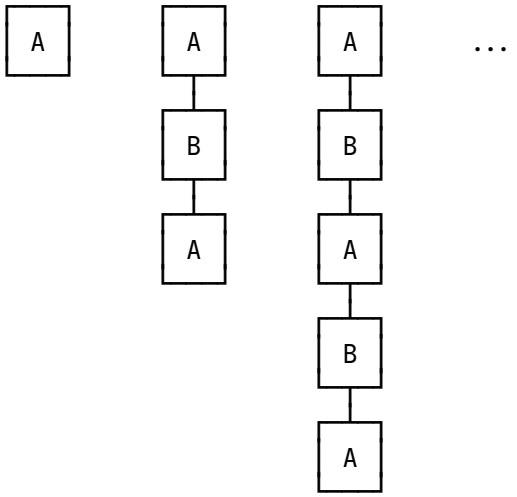
As before, the impatient and the expert can read the source code before the manual.

The bug

I told you last time there was a bug. Look at the following grammar (let us assume the start symbol is `A`):

```
A →
A → B
B → A
```

It's an uninteresting, trivial grammar that parses the empty string, but that's enough for our purposes. You will notice that this grammar is ambiguous: there are several syntax trees we could derive from an empty input. Well, a countable infinity of them, in fact:



But, when you use our Earley recogniser on this grammar, you get the following items:

```
═══ 0 ═══
A → •    (0)
A → • B  (0)
B → • A  (0)
```

Here, we can see only one completed item:

```
A → • (0)
```

Which means, the subsequent parser will only see *one* possible parse:

So we have a bug.

The problem

The problem comes from the poor timing around the handling of empty rules. Let's take a look at the items one more time:

```
═══ 0 ═══
A → •    (0)
A → • B  (0)
```

```
B → • A  (0)
```

See the last item? It calls for completion: `A` has been completed earlier, with item `A → • (0)`. So, `B → • A (0)` should be advanced one step. We should add this item:

```
B → A • (0)
```

And this one as well, while we're at it:

```
A → B • (0)
```

But as I said, there's a timing problem. When the completion step is triggered, the state set looks like this:

```
═══ 0 ═══
A → •    (0)  ← completion to perform
A → • B  (0)
```

Nice, a completion! Let's look at all the items of the form:

```
R → α • A β (j)
```

When you think about it, there is one:

```
B → • A  (0)
```

But guess what, it doesn't exist yet. It will be predicted *later*, by the second item of the set. So we can't advance it, and miss another completion in the process.

Ordinarily, it is not possible for a completed item to be created before the items it is supposed to advance one step: they belong to a later Earley set: by the time a later Earley set i being processed, the previous ones are done. Unfortunately, empty rules break that assumption: now we look for items in a set that is not fully built yet! Of course we could miss something.

The solution

One obvious solution was given by Grune and Jacob (1990):

> The easiest way to handle this mare's nest is to stay calm and keep running the Predictor and Completer in turn until neither has anything more to add

Ah, but I don't like this *at all*. One thing I liked about our algorithm, was the ability to process the Earley sets in one pass. With this method, we have to loop at least twice, add another nesting to our loops… *Yuck.*

The *real* solution came from Aycock and Horspool (2002). They noticed we could advance some items without waiting for a completion to trigger that advancement. More precisely, some symbols are *nullable*: they can parse the empty string. Hence the trick:

When performing a prediction, if the predicted symbol is nullable, then advance the predictor one step.

Let's say for instance we're processing the second item:

```
═══ 0 ═══
A → •    (0)
A → • B  (0)  ← needs a prediction.
```

So we need to predict the symbol `B`. `B` happens to be nullable. So, we can advance `A → • B (0)` without waiting for something like `B → α • (0)` to appear. It *will* come up eventually, so we might as well anticipate the completion. So there's *two* things happening in the predictor: the actual prediction, and a magical completion:

```
═══ 0 ═══
A → •    (0)
A → • B  (0)  ← predictor
B → • A  (0)  ← predicted from the predictor
```

```
A → B •  (0)  ← magical completion of the predictor
```

Now, we can keep processing the Earley sets in one pass. Don't worry, Aycock and Horspool have proven this trick works. There's just one last hurdle…

Detecting nullable symbols

A nullable symbols is a symbol that name at least one nullable rule. And a nullable rule is only composed of nullable symbols. (Possibly zero, which makes the rule empty.) Terminal symbols aren't nullable.

There are several ways to do this. I personally have chosen to just scan every rule, and maintain a list of nullable symbols in the process. If a rule is empty, or contains only symbols from the nullable set, then I add its name to the nullable set. And I keep scanning all the rules over and over, until the nullable set no longer grows.

This is simple, but slow. Quadratic with respect to the size of the grammar, in fact. We can beat this. In his Marpa parser, Jeffrey Kegler has devised an alternative which should work in linear time. If you intend to process large grammars (eventually you will), you should take a look.

Anyway, now that I have a nullable set, I can just use it to test if a symbol is nullable when I make a prediction.

Lua code

The code is very simple. I only have made 2 additions to the recogniser. First, the detection of of the nullable set:

```
---------------------------
-- Detecting nullable rules -- Nullable symbols sets are named "nss".
---------------------------
local function nullable_nss()
   return { size = 0 }
end

local function add_nullable_rule(rule_name, nss)
   if nss[rule_name] then return end  -- Do nothing for known nullable rules.
   nss[rule_name] = true              -- The others are added,
   nss.size = nss.size + 1            -- and the size is adjusted.
end

-- Returns true if it can say for sure the rule is nullable.
-- Returns false otherwise
local function is_nullable(rule, nss)
   for i = 1, #rule do
      if not nss[rule[i]] then
         return false
      end
   end
   return true
end

-- Adds nullable rules to the nss, by examining them in one pass.
local function update_nss(nss, grammar)
   for i = 1, #grammar do                     -- For each rule,
      if is_nullable(grammar[i], nss) then       -- if the rule is nullable for sure,
         add_nullable_rule(grammar[i].name, nss) -- add it to the nss.
      end
   end
end

local function nullable_rules(grammar)
   local nss = nullable_nss()
   repeat                        -- Keep...
      local old_size = nss.size
      update_nss(nss, grammar) --  ...updating the nss,
```

```
   until old_size == nss.size  -- as long as it keeps growing.
   return nss                  -- An nss that stopped growing is complete.
end
```

Then, the addition of the magic completion in the prediction step:

```
local function predict(S, i, j, symbol, grammar, nss)
   for rule_index, rule in ipairs(grammar) do
      if rule.name == symbol then
         append(S[i], { rule  = rule_index,
                        next  = 1 ,
                        start = i })
         if nss[rule.name] then -- magical completion
            append(S[i], { rule  = S[i][j].rule,
                           next  = S[i][j].next + 1,
                           start = S[i][j].start})
         end
      end
   end
end
```

And that’s basically it. As before, the full source code is available.

Optimising Right Recursion

Update, 14/12/2014: Jeffrey Kegler, the author of the Marpa parser, posted a [response](#) to this post. You probably should take a look.

Note: I have decided not to include this optimisation in the rest of the series. It complicates the core algorithm, has further repercussions in the parsing phase, and its performance benefits are unclear —they need to be tested. (Leo didn't lie about the complexity bounds, but he omitted less interesting considerations such as the ever growing memory gap). Simply put, more research is needed. I may perform it. Someday.

The problem

Consider the following grammar:

```
A → A 'a'
A →
```

Let it parse this input: `"aaaaa"`. For now, there's no problem:

```
═══ 0 ═══
A → • A 'a'  (0)
A → •        (0)
A → A • 'a'  (0)

═══ 1 ═══
A → A 'a' •  (0)
A → A • 'a'  (0)

═══ 2 ═══
A → A 'a' •  (0)
A → A • 'a'  (0)

═══ 3 ═══
A → A 'a' •  (0)
A → A • 'a'  (0)

═══ 4 ═══
A → A 'a' •  (0)
A → A • 'a'  (0)

═══ 5 ═══
A → A 'a' •  (0)
A → A • 'a'  (0)
```

But then, consider this slightly different grammar (the symbols of the first rule have been inverted):

```
A → 'a' A
A →
```

Upon the same input, it yields *this*:

```
═══ 0 ═══
A → • 'a' A  (0)
A → •        (0)

═══ 1 ═══
A → 'a' • A  (0)
A → • 'a' A  (1)
A → 'a' A •  (0)
A → •        (1)
```

```
═══ 2 ═══
A → 'a' • A  (1)
A → • 'a' A  (2)
A → 'a' A •  (1)
A → •        (2)
A → 'a' A •  (0)

═══ 3 ═══
A → 'a' • A  (2)
A → • 'a' A  (3)
A → 'a' A •  (2)
A → •        (3)
A → 'a' A •  (1)
A → 'a' A •  (0)

═══ 4 ═══
A → 'a' • A  (3)
A → • 'a' A  (4)
A → 'a' A •  (3)
A → •        (4)
A → 'a' A •  (2)
A → 'a' A •  (1)
A → 'a' A •  (0)

═══ 5 ═══
A → 'a' • A  (4)
A → • 'a' A  (5)
A → 'a' A •  (4)
A → •        (5)
A → 'a' A •  (3)
A → 'a' A •  (2)
A → 'a' A •  (1)
A → 'a' A •  (0)
```

Now we have a problem: the previous grammar yielded a reasonable number of Earley items, but this one… grows quadratically. Such problems are expected of Earley's algorithm, but come on! This grammar could be parsed by a simple LR parser in linear time! We should not need that many Earley items.

The solution

The trick was published by Joop Leo in 1991. The idea is to avoid completing certain key Earley items.

Deterministic reduction path

Picture this Earley item (Greek letters denote sequences of symbols):

`A → α • (i)`

The deterministic path, if any, is a chain of Earley items. The first item of this chain, if there is such a thing, satisfies 2 conditions:

- It is of the form: `X → β A • (k)`
- There is *exactly one* item of the form `X → β • A (k)` in the Earley set `S(i)`,

Now we can picture *this* Earley item: `X → β A • (k)` (yes, it's the one we just discovered), and find what's just above it. We can then go up the chain until we can't. The last Earley item we find is called the *topmost* complete item on the deterministic reduction path.

By the way, you may notice the eerie similarity with the regular completion step. See, when performing a completion, we check for all items of the form `X → β • A δ (k)` in `S(i)`, so we can add `X → β A • δ (k)` to the current Earley set. The reduction path business is only a special case, where `δ` happens to be empty.

The algorithm

We modify the completion step. Instead of performing a regular completion right away, we first examine our Earley item, and check if there is a deterministic reduction path.

- If there is, we just add the topmost item on this path to the current Earley set.
- Otherwise, we perform a regular completion.

The effect of these shenanigans is fairly simple: if we were doing the regular completion, we would end up adding the whole reduction path to the current Earley set. Instead, we just add the topmost item, and skip all the others.

Let's take the construction of the final Earley set in our last example. With the regular algorithm, we get this:

```
═══ 5 ═══
A → 'a' • A  (4)  -- scanned previously
A → • 'a' A  (5)  -- Predicted
A → 'a' A •  (4)  -- Special completion (because A is nullable)
A → •        (5)  -- Predicted
A → 'a' A •  (3)  -- Completed
A → 'a' A •  (2)  -- Completed
A → 'a' A •  (1)  -- Completed
A → 'a' A •  (0)  -- Completed
```

Now see the last four completions? They're the deterministic reduction path of `A → • (5)`. With our new and improved algorithm, we would only add the last one. Here is how it would go. First, we would proceed as usual:

```
═══ 5 ═══
A → 'a' • A  (4)  -- scanned previously
A → • 'a' A  (5)  -- Predicted
A → 'a' A •  (4)  -- Special completion (because A is nullable)
A → •        (5)  -- Predicted
```

Then, things change as we complete `A → • (5)`. First, we check for the presence of an item of the form `X → α • (k)` in `S(5)`. There is one, *and only one*:

```
A → 'a' • A  (4)
```

Now, we know that

```
A → 'a' A •  (4)
```

is in the deterministic path. So, instead of putting it in `S(5)` right away (which by the way wouldn't change a thing, since it is already present), we go up the chain, and look for an item of the form `X → α • (k)`, but this time, in `S(4)`. And lo and behold, we find…

```
A → 'a' • A  (3)
```

So the next item up the chain is

```
A → 'a' A •  (3)
```

And so on until we find the topmost item:

```
A → 'a' A •  (0)
```

And we just add that. We therefore get this:

```
═══ 5 ═══
A → 'a' • A  (4)  -- scanned previously
A → • 'a' A  (5)  -- Predicted
A → 'a' A •  (4)  -- Special completion (because A is nullable)
A → •        (5)  -- Predicted
A → 'a' A •  (0)  -- Topmost item in the deterministic reduction path
```

Now the number of items can stop growing at each step. Here what it would do to the whole thing:

```
═══ 0 ═══
A → • 'a' A  (0)
A → •        (0)

═══ 1 ═══
A → 'a' • A  (0)
A → • 'a' A  (1)
A → 'a' A •  (0)
A → •        (1)

═══ 2 ═══
A → 'a' • A  (1)
A → • 'a' A  (2)
A → 'a' A •  (1)
A → •        (2)
A → 'a' A •  (0)

═══ 3 ═══
A → 'a' • A  (2)
A → • 'a' A  (3)
A → 'a' A •  (2)
A → •        (3)
A → 'a' A •  (0)

═══ 4 ═══
A → 'a' • A  (3)
A → • 'a' A  (4)
A → 'a' A •  (3)
A → •        (4)
A → 'a' A •  (0)

═══ 5 ═══
A → 'a' • A  (4)
A → • 'a' A  (5)
A → 'a' A •  (4)
A → •        (5)
A → 'a' A •  (0)
```

Okay, that's still more than the left recursive version. Still, the number of Earley items per set is now bounded. We got our linear parsing back…

The *real* algorithm

…mostly. In case you didn't notice, checking the reduction path takes time. And the further we go into the parse, the longer the reduction path is, the longer it takes to go to the top. Even worse, we are checking and re-checking the *same* reduction path over and over! So, while we got spatial complexity back to normal, the temporal complexity is still quadratic.

The obvious solution: cache the topmost item in the reduction path. Next time we check we won't have to go up the whole chain all over again. To perform the cache, we use what Leo calls *transitive items*. A transitive item is something of the form:

```
X → α •  (i), Y
```

where `X` and `Y` are non-terminal symbols, and `α` is a sequence of symbols (terminal or non-terminal). Basically, it is a completed Earley item, with a symbol attached. The Earley item itself is the topmost item on some reduction path. The symbol tells us *which* reduction path is involved.

Oh, I didn't tell you: at any given point in the parse, you can't find more deterministic paths than non-terminal symbols in the grammar. Looking back at the definition of deterministic paths, it should have been obvious. (I actually missed it, and got the tip from Jeffrey Kegler, in his Marpa parser paper.) In our example above, for instance, there is only one non-

terminal symbol (`A`). So, there is only one deterministic reduction path.

Now, when we find the topmost item in the reduction path, we add the corresponding transitive item to the current Earley set. And all the Earley sets we just examined, while we're at it.

(That last one bothers me. We used to only touch two Earley sets: the current one (predictions and completions), and the next one (scans). This could enable the use of a neat dynamic array for the whole thing. But with this new "optimisation", we now have to deal with arbitrary insertions. This could be a performance killer: arrays aren't exactly insertion friendly, and anything else likely involve an uncanny amount of pointer chasing. Maybe we can mitigate the damage with some kind of cache aware data structure, but I shudder at the sheer complexity.)

In our example it would go like this:

```
≡≡ 0 ≡≡
A → • 'a' A  (0) -- as usual
A → •        (0) -- as usual

≡≡ 1 ≡≡
A → 'a' • A  (0) -- as usual
A → • 'a' A  (1) -- as usual
A → 'a' A •  (0) -- Search for the topmost item -there is none.
A → •        (1) -- as usual
A : A → 'a' A • (1) <──────────this transitive item──────┐
                                                        │
≡≡ 2 ≡≡                                                 │
A → 'a' • A  (1) -- as usual                            │
A → • 'a' A  (2) -- as usual                            │
A → 'a' A •  (1) -- Search for the topmost item          │
                    Finds A → 'a' • A  (0), creates ────┘
A : A → 'a' A • (1) <── And this one one as well
                          (Yes, it is the same)
A → •        (2) -- Search for the topmost item,
                    finds the transitive item above
A → 'a' A •  (0) -- Search for the same topmost item,
                    there is none.
```

This little dances then goes on in the same fashion:

```
≡≡ 3 ≡≡
A → 'a' • A  (2)
A → • 'a' A  (3)
A → 'a' A •  (2)
A : A → 'a' A • (1) -- copy of the transitive item above
A → •        (3)
A → 'a' A •  (0)

≡≡ 4 ≡≡
A → 'a' • A  (3)
A → • 'a' A  (4)
A → 'a' A •  (3)
A : A → 'a' A • (1) -- copy of the transitive item above
A → •        (4)
A → 'a' A •  (0)

≡≡ 5 ≡≡
A → 'a' • A  (4)
A → • 'a' A  (5)
A → 'a' A •  (4)
A : A → 'a' A • (1) -- copy of the transitive item above
A → •        (5)
A → 'a' A •  (0)
```

More or less. Sorry for the lack of details (especially with respect to empty rules), I'm not very motivated to flesh it out right now: I'm not sure this optimisation is worth the trouble, and the rest of the series is more important anyway.

Earley Parsing Explained — The Parser

We got ourselves a nice recogniser, but most of the time we want more than a binary answer. We need a parse tree.

So, how do we turn our recogniser into a parser? Let's see… Earley himself devised a method, but he was proven wrong by Tomita. Tomita's method doesn't terminate on some grammars. Elizabeth Scott described a method that works, but I totally failed to comprehend it.

Turns out, I don't need to. It all boils down to a simple insight: the Earley items produced by the recogniser are enough. *They* are the parse forest we need. And it is quite simple to extract a parse tree from that forest. (Again, this insight is not mine. I believe I picked it up from Ian Piumarta.)

The solution

We need to perform nested depth-first searches over the completed items. When you look at those items the right way, you can see the search graphs. It is not easy to see if your mind has been polluted by talks of back pointers or some such. It certainly took *me* a long time.

Now the question is, *how the hell* are we supposed to look at our Earley items?

Analysing the Earley states

This is best shown by example. Let's conjure up our trusty arithmetic expression grammar:

```
Sum     → Sum     [+-] Product
Sum     → Product
Product → Product [*/] Factor
Product → Factor
Factor  → '(' Sum ')'
Factor  → Number
Number  → [0-9] Number
Number  → [0-9]
```

When we use it on this input:

```
1+(2*3-4)
```

We get those Earley items.

```
═══ 0 ═══
Sum     → • Sum [+-] Product     (0)
Sum     → • Product              (0)
Product → • Product [*/] Factor  (0)
Product → • Factor               (0)
Factor  → • '(' Sum ')'          (0)
Factor  → • Number               (0)
Number  → • [0-9] Number         (0)
Number  → • [0-9]                (0)

═══ 1 ═══
Number  → [0-9] • Number         (0)
Number  → [0-9] •                (0)
Number  → • [0-9] Number         (1)
Number  → • [0-9]                (1)
Factor  → Number •               (0)
Product → Factor •               (0)
Sum     → Product •              (0)
Product → Product • [*/] Factor  (0)
Sum     → Sum • [+-] Product     (0)
```

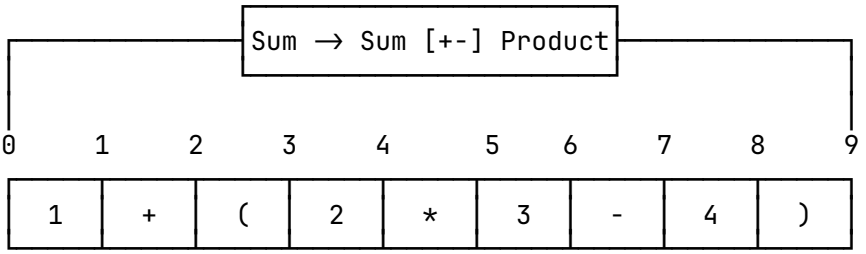
```
═══ 2 ═══
Sum     → Sum [+-] • Product     (0)
Product → • Product [*/] Factor  (2)
Product → • Factor               (2)
Factor  → • '(' Sum ')'          (2)
Factor  → • Number               (2)
Number  → • [0-9] Number         (2)
Number  → • [0-9]                (2)

═══ 3 ═══
Factor  → '(' • Sum ')'          (2)
Sum     → • Sum [+-] Product     (3)
Sum     → • Product              (3)
Product → • Product [*/] Factor  (3)
Product → • Factor               (3)
Factor  → • '(' Sum ')'          (3)
Factor  → • Number               (3)
Number  → • [0-9] Number         (3)
Number  → • [0-9]                (3)

═══ 4 ═══
Number  → [0-9] • Number         (3)
Number  → [0-9] •                (3)
Number  → • [0-9] Number         (4)
Number  → • [0-9]                (4)
Factor  → Number •               (3)
Product → Factor •               (3)
Sum     → Product •              (3)
Product → Product • [*/] Factor  (3)
Factor  → '(' Sum • ')'          (2)
Sum     → Sum • [+-] Product     (3)

═══ 5 ═══
Product → Product [*/] • Factor  (3)
Factor  → • '(' Sum ')'          (5)
Factor  → • Number               (5)
Number  → • [0-9] Number         (5)
Number  → • [0-9]                (5)

═══ 6 ═══
Number  → [0-9] • Number         (5)
Number  → [0-9] •                (5)
Number  → • [0-9] Number         (6)
Number  → • [0-9]                (6)
Factor  → Number •               (5)
Product → Product [*/] Factor •  (3)
Sum     → Product •              (3)
Product → Product • [*/] Factor  (3)
Factor  → '(' Sum • ')'          (2)
Sum     → Sum • [+-] Product     (3)

═══ 7 ═══
Sum     → Sum [+-] • Product     (3)
Product → • Product [*/] Factor  (7)
Product → • Factor               (7)
Factor  → • '(' Sum ')'          (7)
Factor  → • Number               (7)
Number  → • [0-9] Number         (7)
Number  → • [0-9]                (7)

═══ 8 ═══
Number  → [0-9] • Number         (7)
Number  → [0-9] •                (7)
```

```
Number  → • [0-9] Number         (8)
Number  → • [0-9]                (8)
Factor  → Number •               (7)
Product → Factor •               (7)
Sum     → Sum [+-] Product •     (3)
Product → Product • [*/] Factor  (7)
Factor  → '(' Sum • ')'          (2)
Sum     → Sum • [+-] Product     (3)

=== 9 ===
Factor  → '(' Sum ')' •          (2)
Product → Factor •               (2)
Sum     → Sum [+-] Product •     (0)
Product → Product • [*/] Factor  (2)
Sum     → Sum • [+-] Product     (0)
```

Okay, let's start from the beginning. We know the parse was successful: the recogniser said so, by showing us this item:

```
=== 9 ===
Sum → Sum [+-] Product • (0)
```

There is a dot at the end, so this is a completed item. It starts at `(0)` (the beginning), and stops at `(9)` (the very end). There's only one way Earley's algorithm could possibly produce such an item: the whole input is a `Sum`. We can visualise this item in a chart:



Indeed, this is a `Sum`, composed of 3 parts: the left operand, `"1"`, the operator, `"+"`, and the right operand, `"(2*3-4)"`. Anyway, that's how we know the parse succeeded as a whole.

So, Earley's algorithm works. Duh. But that's not what I'm after. I need you to recall how this state was created. More specifically, *what from*. Let me pull back what I said in the recogniser section:

> Imagine we have an item like this ('a', 'b', and 'c' are symbols, and 'i' is an integer):
>
> ```
> Rule → a b c •  (i)
> ```
>
> The fact that this item even *exist* means the following items also exist somewhere:
>
> ```
> Foo →   a   b • c  (i)
> Foo →   a • b   c  (i)
> Foo → • a   b   c  (i)
> ```

In our current example this means we can find those items:

```
Sum →   Sum   [+-] • Product (0)
Sum →   Sum • [+-]   Product (0)
Sum → • Sum   [+-]   Product (0)
```

But that's not the end of it. To advance an item one step, you need *two* things: an un-advanced version of the item (which we have here), and a completed *something*: either a completed state, or a successful scan. This has several implications:

- There is another completed `Sum` somewhere. It starts at `(0)`, and finishes at… well… let's say `(x)`.
- There is a successful scan between `(x)` and `(x+1)`. Meaning, the input at `x` matches `[+-]`.
- There is a completed `Product` somewhere. It starts at `(x+1)`, and finishes at… wait a minute this is the last one! it's got to finish wherever the overall `Sum` finishes! That would be the end of the input, or `(9)`.

Performing the search

The problem now is to search for those states, and determine the value of `(x)`. Given how Earley items are stored in the state sets, we need to start at the end: let's look for the `Product` there:

```
=== 9 ===
Factor  → '(' Sum ')' •          (2)
Product → Factor •               (2)
Sum     → Sum [+-] Product •     (0)
Product → Product • [*/] Factor  (2)
Sum     → Sum • [+-] Product     (0)
```

Okay, so there is only one that's completed:

```
=== 9 ===
Product → Factor • (2)
```

And it started at `(2)`. Okay, so `(x+1)` equals `(2)`! Which means, there should be a successful scan between `(1)` and `(2)`. Let's check the second character of the input:

```
1+(2*3-4)
```

That would be `"+"`, which definitely matches `[+-]`. All is well so far. Now, we're looking for a completed `Sum` in `(x)` — that is, `(1)`. Let's look…

```
=== 1 ===
Number  → [0-9] • Number         (0)
Number  → [0-9] •                (0)
Number  → • [0-9] Number         (1)
Number  → • [0-9]                (1)
Factor  → Number •               (0)
Product → Factor •               (0)
Sum     → Product •              (0)
Product → Product • [*/] Factor  (0)
Sum     → Sum • [+-] Product     (0)
```

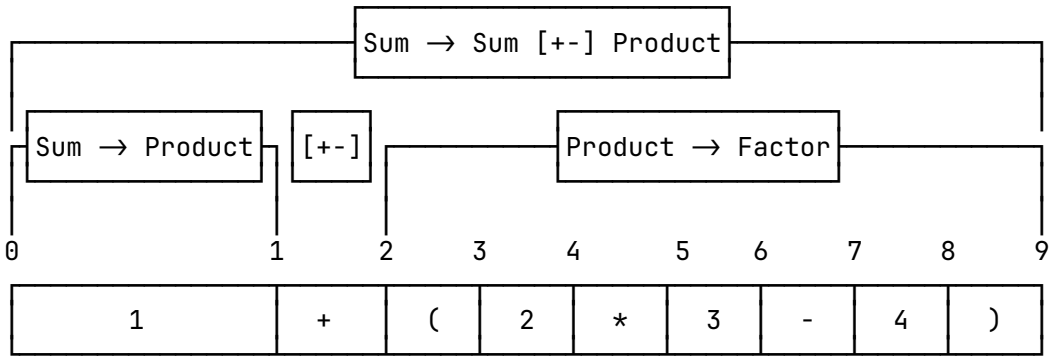
Okay, we can find only one:

```
=== 1 ===
Sum → Product • (0)
```

Now to sum it up, the whole parse is composed of 3 parts:

```
=== 1 === Sum → Product • (0)
=== 2 === [+-]
=== 9 === Product → Factor • (2)
```

Again, we can visualise this as a chart. Here, completed items become *edges* of a *graph*. The nodes of the graph are the numbers I put between each input token (0-9). In some way, those nodes represent the state sets.

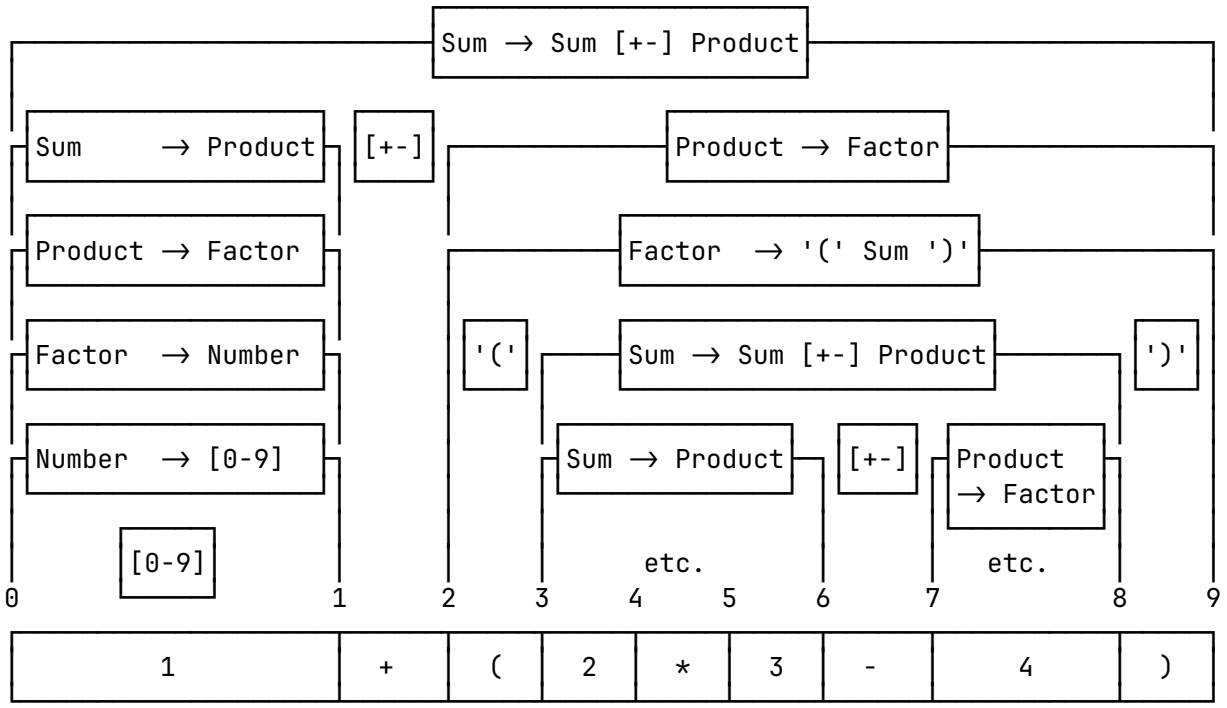


Granted, this case was easy, because there was only one solution. You can thank the grammar for not being ambiguous. In any case, there *had* to be a solution: if there wasn't one, the parse would have failed in the first place. The `Sum` that

spans the whole input wouldn't be there.

Nested searches!

So, we know exactly where the big `Sum` comes from. We know that the scan succeeded, and why. But what about the little `Sum` and `Product`? Well, we just have to perform the same work, going down one level. And then another. And another. Here is the final result, minus 2 gaps for space. Filling them should be easy.



It helps to do those things manually at least once, to get a feel of the algorithm, and the data we need.

Speaking of which (you probably have noticed this by now), we don't need all of the Earley items. Only the completed ones. This makes sense since we're after successful parses, not aborted attempts. Here is a list of all completed items: Note the correspondence with the chart above: it's basically the same data, visualised differently. Only the scans aren't represented.

```
═══ 0 ═══

═══ 1 ═══
Number  → [0-9]                (0)
Factor  → Number               (0)
Product → Factor               (0)
Sum     → Product              (0)

═══ 2 ═══

═══ 3 ═══

═══ 4 ═══
Number  → [0-9]                (3)
Factor  → Number               (3)
Product → Factor               (3)
Sum     → Product              (3)

═══ 5 ═══

═══ 6 ═══
Number  → [0-9]                (5)
Factor  → Number               (5)
Product → Product [*/] Factor  (3)
Sum     → Product              (3)
```

```
═══ 7 ═══

═══ 8 ═══
Number  → [0-9]                (7)
Factor  → Number               (7)
Product → Factor               (7)
Sum     → Sum [+-] Product     (3)

═══ 9 ═══
Factor  → '(' Sum ')'          (2)
Product → Factor               (2)
Sum     → Sum [+-] Product     (0)
```

Much more manageable.

Searching from the wrong end

We still have a problem however. Remember how we found out how the big `Sum` was parsed? We searched from the *end*, then worked up to the beginning. In this case it didn't matter, because our grammar has no ambiguity. If we found one however, we would have to resolve it from the *end*, which is not very intuitive for the user: can you imagine a longest-match rule that maximises the length of the *last* rule first? This would look like a *shortest* match in most cases.

That won't do. We need to start at the beginning. The trick is to see the completed items as edges:

- A beginning.
- An end.
- A grammar rule.

A completed item only stores its beginning and its rule. Its end is implicit: it's the Earley set it is stored on. We can reverse that. Instead of having this:

```
═══ 9 ═══
Product → Factor (2)
```

We could have the *beginning* be implicit, and store the end. Like that:

```
═══ 2 ═══
Product → Factor (9)
```

It is basically the same thing, but now we can perform searches from the beginning. Here is the whole chart, dully reversed:

```
═══ 0 ═══
Sum     → Sum [+-] Product    (9)
Sum     → Product             (1)
Product → Factor              (1)
Factor  → Number              (1)
Number  → [0-9]               (1)

═══ 1 ═══

═══ 2 ═══
Product → Factor              (9)
Factor  → '(' Sum ')'         (9)

═══ 3 ═══
Sum     → Sum [+-] Product    (8)
Sum     → Product             (6)
Sum     → Product             (4)
Product → Product [*/] Factor (6)
Product → Factor              (4)
Factor  → Number              (4)
Number  → [0-9]               (4)
```

```
═══ 4 ═══

═══ 5 ═══
Factor → Number              (6)
Number → [0-9]               (6)

═══ 6 ═══

═══ 7 ═══
Product → Factor             (8)
Factor  → Number             (8)
Number  → [0-9]              (8)

═══ 8 ═══

═══ 9 ═══
```

Now we can start again. The top rule, the one that says we parsed everything, is now at the *beginning*:

```
═══ 0 ═══
Sum → Sum [+-] Product (9)
```

It is a `Sum` that starts at `(0)` and finishes at `(9)`. Again, this confirms the input has been completely and correctly parsed. Now we can search how this `Sum` has been parsed. From this input, we know the same things we knew before:

- There is a completed `Sum` between `(0)` and `(x)`.
- There is a "+" or a "-" character between `(x)` and `(x+1)`.
- There is a `Product` between `(x+1)` and `(9)`

Now however, because the edges are stored at their beginning location, we can search this chart at `(0)`:

```
═══ 0 ═══
Sum     → Sum [+-] Product   (9)
Sum     → Product            (1)
Product → Factor             (1)
Factor  → Number             (1)
Number  → [0-9]              (1)
```

Ah, there are *Two* sums that match. Let's try the first:

```
═══ 0 ═══
Sum     → Sum [+-] Product   (9)
```

If this is the one, `(x)` should now be `(9)`. Let's see if there is a character between `(9)` and `(10)`… argh, this is the end of the input! It doesn't match. Okay, let's keep calm and backtrack…

```
═══ 0 ═══
Sum     → Sum [+-] Product   (9) !! Failed !!
Sum     → Product            (1)
Product → Factor             (1)
Factor  → Number             (1)
Number  → [0-9]              (1)
```

…so we can try the second `Sum`:

```
═══ 0 ═══
Sum     → Product            (1)
```

If this is the one, `(x)` is now `(1)`. Let's test the input between `(1)` and `(2)`… it is a "+" character, which matches `[+-]`. We're still good. Let's try the `Product` now. It should start at `(2)`:

```
═══ 2 ═══
Product → Factor             (9)
```

```
Factor  → '(' Sum ')'          (9)
```

Good, there is one, and it finishes at `(9)`. This was the last of them, and we're now at `(9)`. We win!

A real depth-first search

We have performed a search. You may now be able to implement it in code. Still, a few clarifications should help you. It may not be obvious, but what we just did was really a depth-first search through a graph.

Aside: depth first search.

This one's simple: you have a directed graph, with nodes and edges. You are at some node.

1. If the current node is a *leaf*, you win!
2. Otherwise, make a list of all edges starting from the current node. That will give you a list of *children* nodes. Go to the first child, then back to step 1. This will yield one of 2 results:
    - You eventually found a leaf. Congratulations, you win!
    - The search stopped before you found a leaf. Try the next child instead.

    If there are no more child to try, the search stops.

More details can be found in the Wikipedia.

Our search graph

We know how to search a graph, but we're not sure *what* graph exactly. More specifically, what are the nodes and the branches of this graph? Where are the leaves? Where do we start?

The *nodes* are the state sets. In our example we have 10, from `(0)` to `(9)`. One of them is the *root* (the search starts there), and one of them is the *leaf* (the search stops there). When we decompose a rule such as this:

```
═══ x ═══
A → a b c  (y)
```

the root is `(x)`, and the leaf is `(y)`.

The *edges* are the items themselves. They start from a node (indeed, they are *stored* in a node, and they point to another node (wherever they finish). There's one subtlety however. Some edges aren't represented in the state stets: the scans. But we don't really need them: first, when we search for a scan, we know it starts at the current node, and stops at the next: scans always span exactly one token. Second, we can test scans directly from the input. Third, the rule we are decomposing tell us which test we must perform.

Another important notion here is the *depth* of the current node. It tells us which symbol we should look for in the rule we are decomposing.

Finally, we should remember the whole *path* that lead us to the final node. That is, the list of the edges that lead us from the root to the leaf.

Now, I should just show you the code. It's not Lua any more, sorry. I couldn't turn my vague ideas into code without static typing to guide me. (Static typing offer a *much* tighter feedback loop than dynamic typing in many cases: failure happens sooner, and closer to the root cause. Invaluable for exploratory programming.)

Anyway:

```
let df_search (edges : int → 'node → 'edge DA.t)
              (child : int → 'edge → 'node     )
              (pred  : int → 'node → bool      )
              (root  : 'node                   )
    : ('node * 'edge) list option =
```

```
  let rec aux depth root =
    if pred depth root
    then Some []
    else opt_find_mem (child depth ⊢ aux (depth + 1))
                      (edges depth root)
         >>= (fun (edge, path) → Some ((root, edge) :: path))
  in aux 0 root
```

This is my generalised depth first search routine. It takes 4 arguments:

- `edges`: a function that given a depth and a node, will return a list of edges.
- `child`: a function that given a depth and an edge, will return a node.
- `pred`: a function that given a depth and a node, will tell you if the node is a leaf.
- `root`: A node. This will be the starting point of the search.

This search then returns a list, where each element is a node and an edge that starts from this node. I have done it this way because the edges I use later don't store their start node —only the grammar rule and their end point.

Ah, and of course, the search could fail, in which case it returns nothing (hence the `option` in the return type). I also use a couple important helper functions:

```
let (>>=)      x f = match x with None → None | Some r → f r
let opt_find     f = DA.foldl (function None → f | e → const e) None
let opt_find_mem f = opt_find (fun a → f a >>= (fun out → Some (a, out)))
```

The first operator let me use the `option` type as a monad. Less cumbersome than pattern matching. `opt_find` is a clever function that applies a function to each element of a list (here implemented as a dynamic array), and returns the first non-null result. `opt_find_mem` is the same, except it also returns the element of the list that produced the result we wanted.

Now the actual search function. It decomposes one edge into a list of sub-edges, using the depth first search above:

```
let top_list (grammar        : 'a grammar     )
             (input          : 'a input       )
             (chart          : edge DA.t DA.t )
             (start          : int            )
             ({finish; rule} : edge           )
    : (int * edge) list =
  let symbols           = rule_body grammar rule           in
  let bottom            = DA.length symbols                in
  let pred  depth start = depth = bottom && start = finish in
  let child depth edge  = edge.finish                      in
  let edges depth start =
    if depth ⩾ DA.length symbols
    then DA.make_empty ()  (* Going past the maximum depth fails the search *)
    else match symbols >: depth with
         | Terminal (t, _) → if t (input start)
                             then DA.make 1 {finish = start + 1; rule = -1}
                             else DA.make_empty () (* Non-matching token fail
                                                      the search *)
         | Non_term name   → (chart >: start) // (fun {finish; rule} →
                                                  rule_name grammar rule = name)
  in
  match df_search edges child pred start with
  | None      → failwith "there's always a solution"
  | Some path → path
```

This function takes 5 arguments. The first three form a general context:

- The grammar over which we operate. I need it to retrieve the rules referenced by the edges. (My edges only store an index to their rule.)
- The input. I need it to check the terminal symbols.
- The chart. All the completed items produced by the recogniser. Dully reversed, so the search can be done from the beginning. This is where I pick my edges from.

The last two form what I call an “edge”:

- `start`: the starting point of the edge. Not technically part of my edges, but important nonetheless (I need them for further decomposition).
- `finish`: where the edge ends.
- `rule`: an integer that denotes a grammar rule. I need the body this rule to perform the decomposition.

You will note the presence of 2 helper operators: the first (`>:`) is array indexing. The second (`//`) is a filter.

Now you can see why the depth is so important: it lets me know which symbol I am processing, and determines the valid edges that start from the current node. There are two cases when producing those edges:

- *Terminal symbols.* Those aren’t represented as edges in the chart, so I have to make one up. This edge references no rule (hence the -1), and it finishes exactly one token later.
- *Non-terminal symbols.* Those I just search for by name. If the current symbol denotes a `Product`, that’s what I will search for.

One last thing: this function *cannot fail*. It will always produce a meaningful result: every edge that is on the chart can be decomposed. Terminal symbols are different, but we don’t decompose them anyway. So, if this ever fail, you have a bug somewhere.

Building the whole parse tree

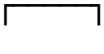
Now we can decompose *one* edge into its sub edges. The only thing left to do is apply this recursively to have the whole tree. This part is relatively obvious:

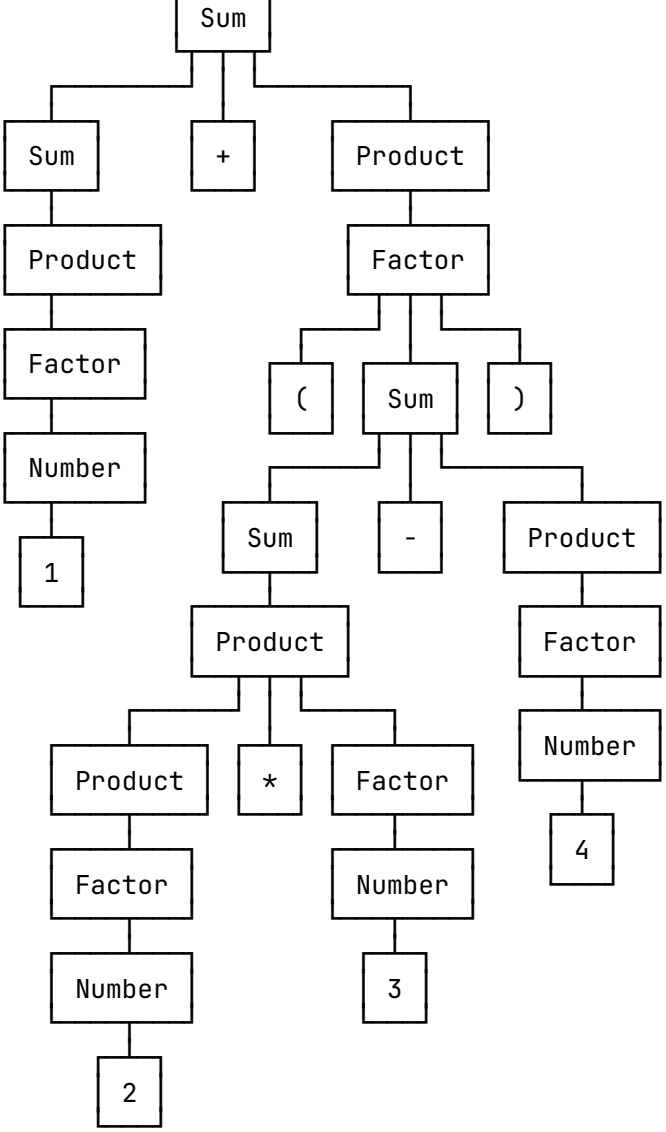
```
type 'a parse_tree = Token of 'a
                   | Node  of string * 'a parse_tree list

let parse_tree (grammar : 'a grammar    )
               (input   : 'a input      )
               (chart   : edge DA.t DA.t)
    : 'a option parse_tree =
  let start  = 0                                        in
  let finish = DA.length chart - 1                      in
  let name   = grammar.start_symbol                     in
  let rule_name {finish; rule} = rule_name grammar rule in
  let rec aux (start, edge)    =
    if edge.rule = -1
    then Token (input start)
    else Node (rule_name edge,
               List.map aux (top_list grammar input chart start edge))
  in
  match DA.find (fun edge → edge.finish = finish && rule_name edge = name)
                (chart >: start)
        >>= fun edge → Some (aux (start, edge))
  with
  | None      → failwith "Are you sure this parse succeeded?"
  | Some node → node
    type 'a parse_tree = Token of 'a
                       | Node  of string * 'a parse_tree list
```

So, a parse tree is either a token (directly taken from the input) or a node (the name of the grammar rule involved, and the list of the sub-tree that compose it). Well, a parse tree.

The function that constructs the parse tree take the same context than the function that decomposes one node: a grammar, the input, and a chart. We then use that context (start symbol of the grammar, size of the chart…) to determine where to start the search. Again, if called correctly on a parse that succeeded, this function cannot fail.

Anyway, this will get us our parse tree:





Hmm… Not as neat and clean as we might want it to be. It’s workable for now, but there are ways to make this simpler. We’ll get to that in a later post.

Resolving ambiguities

I mentioned ambiguities before, yet avoided to confront the subject. The only thing we can do right now is find a parse tree. *Any* parse tree. The problem is, if there are several possibilities, just taking one of them isn’t going to solve our problem: we want the output to be *predictable*.

You might think the problem would go away if you simply ban ambiguous grammars. Now you have *two* problems:

- The ambiguity of context free grammars is undecidable. You will either let some ambiguous grammars slip, or reject some unambiguous ones. Reliability or generality, pick one.
- Second, many ambiguous grammars are actually useful. Some context free languages are *inherently* ambiguous, and can only be parsed with ambiguous grammars. Others are simply easier to process with ambiguous grammars.

So we should accept ambiguous grammar. Let’s see an example: the famous “dangling else”:

```
if
if {}
else {}
```

There are two ways to parse this. Either the `else` branch belongs to the outer `if`:

```
if
  if {}
else {}
```

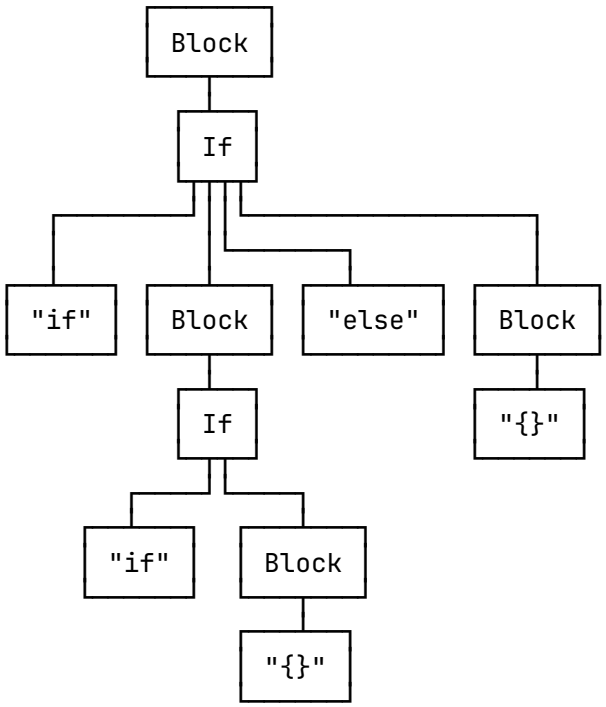
Or, it belong to the inner `if`:

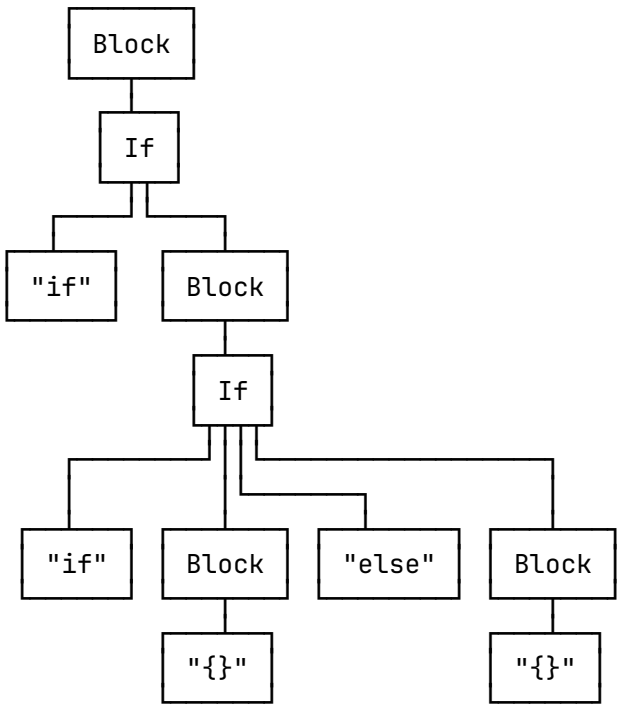
```
if
  if {}
  else {}
```

Let’s try and write a grammar for this little language:

```
Block → "{}"
Block → If
If    → "if" Block
If    → "if" Block "else" Block
```

Here are the two possible syntax trees:





What is an ambiguity?

Recall that Earley items are unique: the algorithm make sure there is no duplicate. Here, we’re only interested in completed items (or edges). So, two different edges will have at least one difference:

- In their start point.
- In their end point.
- Or in their grammar rule.

Obviously. Now recall our depth first search: for a given start point, we gather every edge whose rule name match the current symbol. Yup, there might be several. *That*, is a possible ambiguity.

I must insist on *possible*. As you have seen in our arithmetic expression example above, some branches may abort, leaving only one valid solution. Actual ambiguity only arises when the search could have yielded different results, depending on which edge was picked first.

Resolution

So, when we perform our search, we're only looking at one start point: the current node. The edges we find there can only be different in 2 ways: by their end point, or by their grammar rule. For instance, in our dangling else example, when you're searching for an `If` rule from 0 to 5 (the whole input), you will find 2 edges:

```
═══ 0 ═══
If → "if" Block               (5)
If → "if" Block "else" Block  (5)
```

Those edges end at the same point, but differ by their rule. Written as it is, our search algorithm will just pick the first one. Which is precisely what we will exploit here: when we construct the chart, we can do more than just flipping items. We can *sort* them.

My current choice is a blend of prioritised choice and longest match. The order of the rule in the grammar is significant. If there are more than one possibility, the rule that appear first will have precedence. When that does not suffice, I just put the longest edge first. This way:

```
-- Grammar --
Foo → a b
Foo → c d

-- Chart --
═══ x ═══
Foo → a b  (y)
Foo → c d  (y)
Foo → c d  (z)  -- z < y
```

I believe this gives acceptable control. Granted, there are more fine grained ways of doing this, but the algorithms involved would be significantly more complex. I think it is simpler to just decide at the top level, and let the consequences of those decisions flow downward.

That said, nothing stops you from trying other disambiguation mechanisms: by nature, they're limited to the search of edges and the construction of the parse tree. Whatever complexity lurks in there won't affect the rest of the program.

Back to our example, if we pick this rule first…

```
If → "if" Block (5)
```

…then the outer `If` will have no `else` clause. The `else` close will therefore belong to the inner `If`, just like it would in most programming languages. If you pick the other rule first, you get the other possibility.

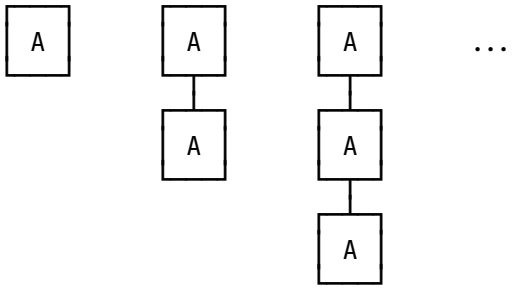
Just by flipping the order of the rules in the grammar, you can decide where the `else` clause will go.

One last trap

I forgot to tell you: some grammars will blow up in your face:

```
A → A
A →
```

The problem is, this grammar generates an infinite amount of parse trees:



So, when I'm constructing my parse tree, I could be unlucky enough to fall into one of those bottomless pits until I overflow the stack. We need to do something about it. We could try and avoid the infinite paths, but I don't like it. It would complicate the construction of the parse tree, and frankly, this is complex enough already.

My solution is more brutal: such bottomless grammars are bogus, and should be rejected. Fortunately, unlike unlike ambiguous grammars, they can be detected with a little bit of static analysis.

- First we compile the nullable rules: they are the rules which contain only nullable terminal symbols. (And we have learned to detect those earlier).

- For each nullable symbol, we compile the symbols which appear in the right hand side of the corresponding nullable rules. Imagine the following grammar:

  ```
  A → A C
  A → B
  A →
  B → A
  C → 'x'
  ```

  This grammar has 2 nullable symbols, A and B. The nullable rules are those:

  ```
  A → B
  A →
  B → A
  ```

  In this simple case, the only child of "A" is "B", and vice versa.

- Finally, we look for duplicates in the transitive closure of children. If a path includes a duplicate, then we have found a loop, and must reject the grammar. For instance, with he above grammar, When I examine the symbol "A", I find only one child, "B". So far so good. Then I examine "B" and find the grandchild "A". We have a duplicate, so this grammar is bottomless.

That's basically it. For reference, here is my code:

```
let infinite_loop : 'a grammar → bool = fun grammar →
  let null_set        = nullable_symbols grammar                          in
  let rules symbol    = grammar.rules // (fun {lhs} → lhs = symbol)
                                      // is_nullable null_set
                                      /@ (fun {rhs} → rhs)                in
  let add_rule        = DA.foldl (fun set → function
                                  | Terminal _   → failwith "impossible"
                                  | Non_term sym → String_set.add sym set) in
  let children symbol = DA.foldl add_rule
                                 String_set.empty
                                 (rules symbol)                           in
  let rec aux path symbol =
    if List.mem symbol path
    then true
    else String_set.exists (aux (symbol::path))
                           (children symbol)
  in String_set.exists (aux []) null_set
```

It is worth breaking up a bit. First, we get the null symbols. I reuse the previous nullable_symbols detector directly.

```
let null_set = nullable_symbols grammar
```

Then I need to detect the nullable rules that correspond to a given symbol. I only take the grammars rules I want: their name must be the symbol I seek (obviously), and they must be nullable. Finally, I only take the right hand side of each rule. (The `//` operator is a filter, and the `/@` operator is a map.)

```
let rules symbol = grammar.rules // (fun {lhs} → lhs = symbol)
                                 // is_nullable null_set
                                 /@ (fun {rhs} → rhs)
```

Once I have my rules for a given symbol, I need to collect the children symbols. Remember that nullable rules don't have terminal symbols.

```
let add_rule = DA.foldl (fun set → function
                         | Terminal _   → failwith "impossible"
                         | Non_term sym → String_set.add sym set)

let children symbol = DA.foldl add_rule
                               String_set.empty
                               (rules symbol)
```

Finally, I perform the search for duplicates. During the search, I keep track of the symbols of the current path. If the current symbol is in this path, we have hit a loop, and return `true` to say so. (Production code should return the path itself, for a more friendly error message.) Otherwise we search deeper, if possible.

```
let rec aux path symbol =
  if List.mem symbol path
  then true
  else String_set.exists (aux (symbol::path))
                         (children symbol)
```

Of course, We need to perform this search for every nullable symbol:

```
String_set.exists (aux []) null_set
```

Conclusion

Now you should be able to build a fully fledged Earley parser. As always, my code is available for study. Again, sorry about using Ocaml instead of sticking with Lua. Dynamic typing just required too much brainpower. If you find my code hard to read, this other tutorial may help you.

Semantic Actions

Earley Parsing Explained — Semantic Actions

(Source code for the impatient).

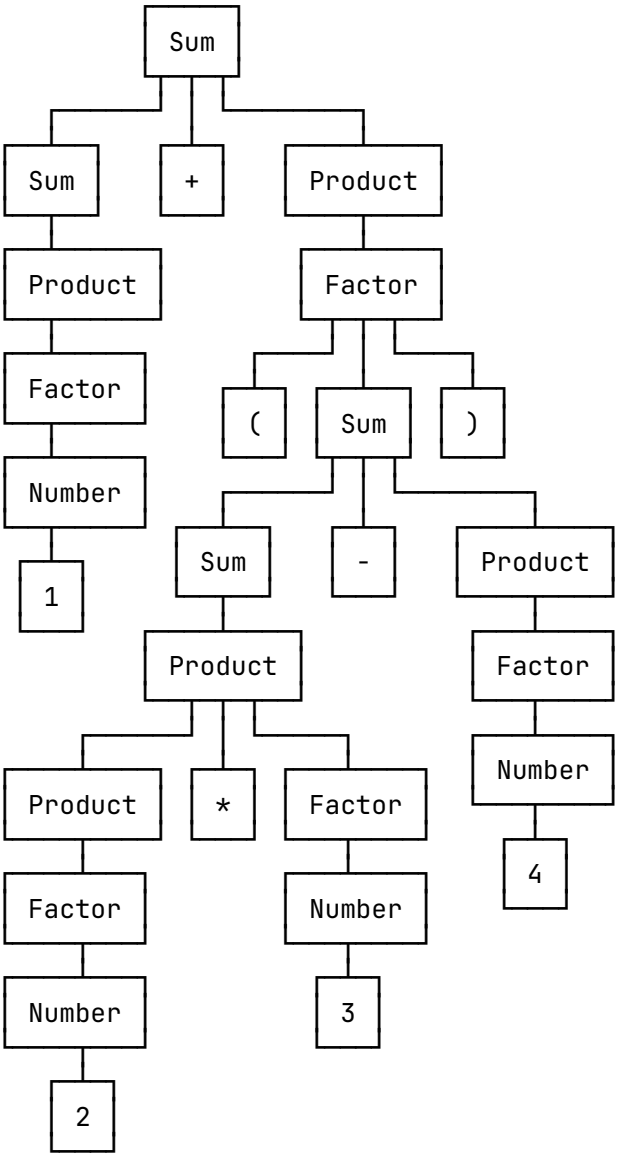
We now know how to transform unstructured input into a genuine parse tree. There's only one slight problem: that parse tree is *ugly*. More specifically, that parse tree is entirely determined by the shape of the grammar, offering us very little flexibility. Let us review our arithmetic example.

The Grammar:

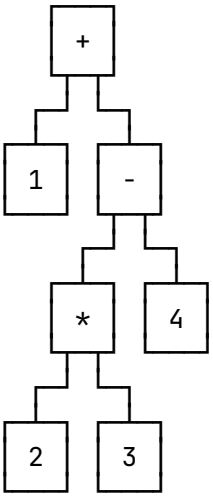
```
Sum     → Sum     [+-] Product
Sum     → Product
Product → Product [*/] Factor
Product → Factor
Factor  → '(' Sum ')'
Factor  → Number
Number  → [0-9]
```

The input we want to parse: `1+(2*3+4)`

The resulting parse tree:



Ugh. There are too many useless nodes here. I'd rather have something like this:



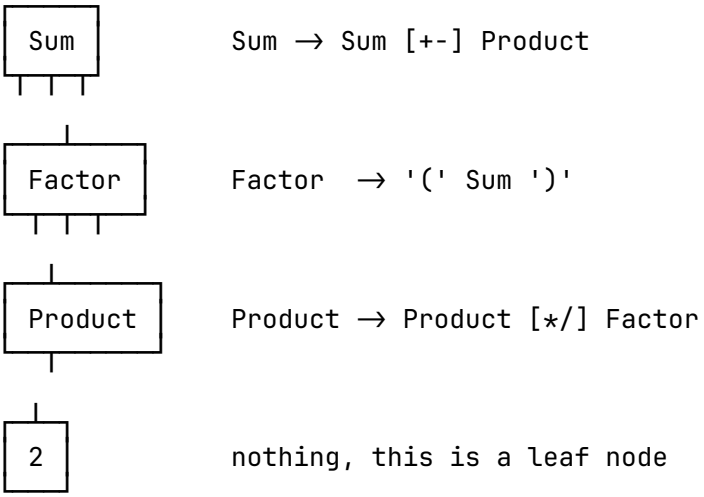
There are several ways to collapse the former into the latter. One way would be to write an ad-hoc recursive algorithm over the parse tree. Another way would be using *semantic actions*.

The structure of the parse tree

Our ugly parse tree has a very straightforward structure. Its nodes are either a token, or the aggregation of a grammar rule and a list of sub nodes. In Ocaml, we write this (the grammar rule is represented with an index.)

```
type 'a parse_tree = Token of 'a
                   | Node  of int * 'a parse_tree list
```

So, each non-leaf node is associated with exactly one grammar rule. Actually, that grammar rule is directly responsible for the creation of this node! Here are a couple examples (search for those nodes in the tree above).



This is important, because the number and nature of the sub nodes is entirely determined by the grammar rule involved. For instance, this grammar rule:

```
Sum → Sum [+-] Product
```

will always yield a node with 3 sub-nodes, one of which is either “+” or “-”. Indeed, you can see that the top node above has 3 sub nodes, one of which is “+”.

Walking down the parse tree

Those invariants are extremely convenient, because they allow potent simplifying assumptions. When we analyse a node, we can use one specialised (and *simple*) piece of code, provided we know which grammar rule was involved in the first place. So we need as many pieces of code as there are grammar rules.

Those pieces of code are the semantic actions.

Conceptually, a semantic action is a function: it takes a list of values as input, and returns a value. The input comes from the sub nodes. There is just the special case of tokens, which must have a semantic action of their own. (Possibly none, if you consider tokens to be values already.)

In a dynamic language, this would be very easy to describe. In Ocaml, we kinda have to fight the language. To keep things simple, I have decided to emulate a dynamic type system. So, values shall be s-expressions:

```
type sexpr = Nil
           | Int    of int
           | Char   of char
           | String of string
           | List   of sexpr list
```

Naturally, semantic actions are functions of this type:

```
type semantic_action = sexpr list → sexpr
```

Now we just need to walk down the parse tree:

```
let act (token_handler : 'a → sexpr         )
        (actions       : semantic_action DA.t)
        (ast           : 'a option parse_tree)
    : sexpr =
  let rec aux = function
    | Token (Some t)    → token_handler t
    | Token None        → Nil
    | Node (rule, subs) → (actions >: rule) (List.map aux subs)
  in aux ast
```

We have 3 parameters here.

- The `token_handler`, which turns a token into an s-expression (typically a char or a string). We use the same token handler for every terminal node.
- The `actions` are an array of semantic actions. There's one semantic action for each grammar rule (no more, no less).
- The `ast` is the parse tree, represented by its top node.

There are two base cases, depending on whether there is a meaningful token or not. The recursive case computes the values of each sub node (`List.map aux subs` calls `aux` recursively for each sub node), then gives that list to the relevant semantic action (`actions >: rule` denotes array indexing).

Examples of semantic actions

Now we can write our semantic actions. I'll use pseudo-code for clarity (the Ocaml code is quite ugly). I have 3 examples of semantic actions, all of which work on our trusty arithmetic expression grammar. I'll write the grammar on the left side, as a reminder. In the actual code, semantic actions are stored in a separate array.

Interpreter

```
            grammar              ||    semantic actions
---------------------------------||-----------------------------
Sum     → Sum     [+-] Product   || (l, op, r) → op(l, r)
Sum     → Product                || (p       ) → p
Product → Product [*/] Factor    || (l, op, r) → op(l, r)
Product → Factor                 || (f       ) → f
Factor  → '(' Sum ')'            || (_, s , _) → s
Factor  → Number                 || (n       ) → n
Number  → [0-9]                  || (n       ) → n
```

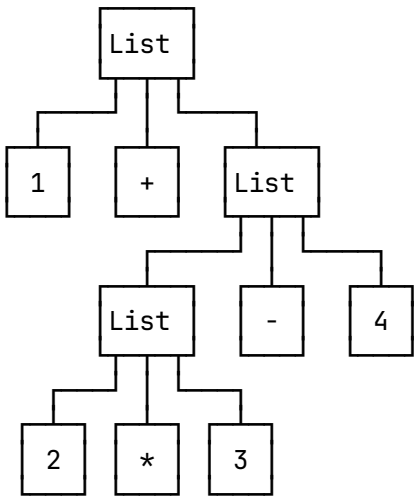
These semantic actions directly "interpret" the input. Considering the start symbol is `Sum`, they will yield a number. On the input `1+(2*3+4)`, the result is `11`.

Abstract syntax tree

We'll be using s-expressions to denote the AST.

```
            grammar              ||    semantic actions
---------------------------------||-----------------------------
Sum     → Sum     [+-] Product   || (l, op, r) → List (l, op, r)
Sum     → Product                || (p       ) → p
Product → Product [*/] Factor    || (l, op, r) → List (l, op, r)
Product → Factor                 || (f       ) → f
Factor  → '(' Sum ')'            || (_, s , _) → s
Factor  → Number                 || (n       ) → n
Number  → [0-9]                  || (n       ) → n
```

That is how you collapse the parse tree: with some nodes, you just pass the result directly to the parent node, without encapsulating it. on the input `1+(2*3+4)`, the result looks like this:



Postfix notation

This one is special: it relies on side effects. Semantic actions can be evaluated in a predictable order. If the host language uses strict evaluation, this is very easy. In our case, the recursive calls are such that the semantic actions are triggered node by node, from left to right, starting with the sub nodes. When you evaluate a given semantic action, you know the semantic actions for all the sub-nodes have been evaluated as well. From left to right, I might add.

This is a straightforward evaluation strategy, applicable to many situations. We don't have to stick to it, however. We could give control to the semantic actions themselves: instead of giving them a list of *values*, you give them a list of *closures*, which, when evaluated (if at all), will yield the value you would have had otherwise… and perform its side effects, if any.

This is the strategy used by Schorre's MetaII. The difference is, instead of parsing some input in a top down fashion, we're walking down a fully formed tree. But I digress.

```
            grammar              ||    semantic actions
---------------------------------||-----------------------------
Sum     → Sum     [+-] Product   || (_, op, _) → print op; print ' '
Sum     → Product                || (p       ) →
Product → Product [*/] Factor    || (l, op, r) → print op; print ' '
Product → Factor                 || (f       ) →
Factor  → '(' Sum ')'            || (_, s , _) →
Factor  → Number                 || (n       ) →
Number  → [0-9]                  || (d       ) → print d; print ' '
```

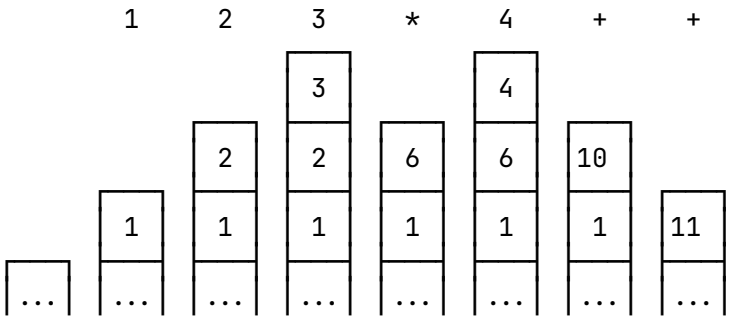
Those semantic actions don't return a meaningful result. They just print to the standard output. Thanks to their natural order of evaluation, with the input `1+(2*3+4)`, they will print this (trailing space not shown):

```
1 2 3 * 4 + +
```

This is a postfix notation, amenable to stack based evaluation:

- `[ 1 ]` Push `1` on the top of the stack.
- `[ 2 ]` Push `2` on the top of the stack.
- `[ 3 ]` Push `3` on the top of the stack.
- `[ * ]` Pop the top 2 elements on the stack, multiply them, push the result back on the top of the stack.
- `[ 4 ]` Push `4` on the top of the stack.
- `[ + ]` Pop the top 2 elements on the stack, multiply them, push the result back on the top of the stack.
- `[ + ]` Pop the top 2 elements on the stack, multiply them, push the result back on the top of the stack.

If you follow these instructions, the stack will evolve like this:



This strategy is not limited to arithmetic expressions. It can be used to generate all kinds of stacked based code.

A few words on efficiency

My implementation of semantic actions is simple and modular, but also inefficient. Semantic actions have a well defined structure, which makes them easy to optimise.

First, we don't need the parse tree to perform the semantic actions. It can be deforested away. The way to do this is simple: instead of constructing a node of that tree, just call the semantic actions directly. Second, I deferred as many decisions as I could to run time. This means an *absurd* amount of dynamic dispatch, which can be virtually eliminated with a bit of static analysis and code generation. Recall how we construct the parse tree:

1. We start from a completed Earley item. We have a start position, an end position, and a grammar rule.

2. We match each non-terminal node of this rule to a completed item (we also test the terminal nodes). Now we have a list of completed Earley items. They can be seen as the "children" of the item we had in step (1).

3. For each item from (2), we (recursively) go back to step (1). This gives us a list of sub-trees (one for each item). (If we call the semantic actions directly, we get a list of values instead)

4. We combine those sub-trees (or values) in a node, that we return to the caller.

The details of those operations are highly dependent on the particular grammar rule involved. Remember the depth first search we perform in step (2)? That search is exactly as deep as the number of symbols in the grammar rule. Moreover, the symbol involved only depends on the depth of the current node. So when the number and nature of the symbol is known in advance, our life is much simpler:

- We don't need the full power of recursion. Nested loops (one per non-terminal symbol) are enough.
- We don't have to test for the end of the search: it is implicit in the code. A success in the inner loop means we're done.
- We don't have to look up the symbols in the grammar rules: we can just "hard-code" them instead.

Specialised depth-first searches can be generated for each grammar rule. From there, the only significant dispatch lies in step (3): the recursive call to the relevant rule. We can just use an indirect call, or we can be clever and switch over the possible cases to help the branch predictor of the CPU: not every rule matches any given symbol.

And so, we have optimised step (2). Now let's take a look at (3) and (4).

In addition to the previous optimisations, code generation also enables static typing for the semantic actions themselves. Originally, I needed the semantic actions to all have the same type, effectively reverting back to dynamic typing, and all

the inefficiencies it entails. (There are other possibilities, but I won't go there.)

The values from step (3) have a type that depends on the symbol involved. I mean, it wouldn't make sense for 2 rules with the same left hand side to return values of different types. Since those types are known in advance, we don't have to go through generic semantic actions. For instance, if a semantic actions needs an integer, we can guarantee it will have just that —no need for any run time test.

But there's more. Sometimes, a semantic action doesn't need all the values it could get from below. With a generic approach, short of using lazy evaluation, we still dig to the bottom no matter what. The specialised approach can instead *omit* the parts of step (3) that are not needed. Depending on the particular grammar and semantic actions involved this can be huge: these are recursive calls. Behind them lie an entire sub-trees worth of computation.

Just one word of caution: if you're counting on side effects performed by the very semantic actions you could omit that way, it might want to give some explicit control to the semantic action writers.

And so, we have optimised steps (2), (3), and (4).

The current source code doesn't perform those optimisations. That would obscure the essence of semantic actions. Just keep them in mind, in case you end up writing a production-quality parsing framework (the whole point of this series, really).