A case study in functional programming: generating efficient functional LR(1) parsers.

Luc Duponcheel and Doaitse Swierstra
Utrecht University, Padualaan 14, 3584 CH Utrecht, the Netherlands
{luc,doaitse}@cs.ruu.nl
http://www.cs.ruu.nl/people/{luc,doaitse}

November 16, 1995

Abstract

This paper presents a case study in functional programming. The paper is based upon a chapter of the course notes of a grammars and parsing course which the authors have given to undergraduate students. The chapter presents a functional program that generates efficient functional LR(1) parsers.

1 Introduction

This paper is based upon a chapter of the course notes of a *grammars and parsing* course [6], given by the authors to undergraduate students of the Computer Science Department of the Utrecht University (the Netherlands). The students who attended the course had already attended a *functional programming* course [5] which was based upon `Gofer` (see [4]). This means that we could assume that they were familiar with list processing functions, list comprehension syntax, higher order functions, type classes and so on.

The grammars and parsing course presents (among others) some general parsing techniques: top-down and bottom-up parsing, and some specialised parsing techniques: LL(1) top-down and LR(1) bottom-up parsing. We felt uncomfortable with both a very theoretical and a very practical approach to the subject. Therefore we decided to use `Gofer` programs to illustrate the course with. On the one hand this enabled us to explain the concepts in rigorous way. On the other hand we could offer the students executable code they could experiment with (the `LaTeX` sources of the different chapters of the course notes are literate `Gofer` scripts).

This paper is based upon a chapter which deals with LR(1) parsing. LR(1) parsing is traditionally implemented in an imperative way by making use of shift-reduce tables (see e.g. [1] and [2]). Using a functional programming language makes it possible to implement LR(1) parsing in a declarative way. The main contribution of the paper is the following: the shift-reduce tables which traditionally drive the parsing process are not needed any more. By introducing a recursive data structure which corresponds to the LR(0) machine of an LR(1) grammar, we can simply use pattern matching to drive the parsing process. This technique results in time-space efficient functional LR(1) parsers. Furthermore we claim that our declarative approach gives much more insight in the problem domain of LR(1) parsing.

The paper is organised as follows. In section 3 we show some typical shift-reduce and reduce-reduce conflicts which can occur during bottom-up parsing. In section 4 we present the LR(0) machine of an LR(0) grammar for expressions. In section 5 and section 6 we present the corresponding data structure and parser code. In section 7 we show how to adapt the code of section 6 such that it can handle LR(1) grammars as well. In section 8 we show some of the techniques we have used to generate the intermediate data structures which are used when generating LR(1) parser code. Section 9 presents an application: generating a recogniser from a concrete representation of a grammar. The resulting code can be found in appendix C. In section 10 we end with drawing some conclusions. The complete code of the application can be ftp'ed from the home page of Luc Duponcheel (click on `lr1.tar.Z`).

2 Grammars

For our purposes a grammar consists of a start symbol (which is a nonterminal symbol) and a table in which we associate with every nonterminal symbol of the grammar a set of alternatives (which are sequences of terminal and nonterminal symbols). For the moment we do not want to specify yet which symbols we actually make use of: they may be characters, tokens generated by a lexical analyser, or any other type. It is convenient to make use of a class to abstract from implementation details like this one. We only require that symbols can be tested for equality and that there exists a predicate `isN` which tests if a symbol is a nonterminal one (all other symbols are terminal ones). For example: when working with characters it is convenient to use upper case characters to denote nonterminals.

```
class Eq s => Symbol s where
  isN :: s -> Bool

instance Symbol Char where
  isN = isUpper
```

As stated above, grammars are defined by making use of a table. For our purposes a table is a type in which one can look up a value which is associated with a key. We use infix operator syntax `(?)` for the lookup operator. Tables are implemented as sets of key-value pairs. Sets are implemented as (totally) ordered lists. On the one hand this enabled us to make use of standard list manipulation functions and list comprehension syntax. On the other hand it we had to arrange things in such a wy that all set manipulating functions indeed mapped sets to sets.

```
type Set x = [x]

type Table key value = Set (key,value)

(?) :: Eq key => Table key val -> key -> Set val
table ? key = [ value | (k,value) <- tab, key == k ]

type Alternative s = [s]

type Grammar s = (s,Table s (Set (Alternative s)))

grammarS :: Grammar Char
grammarS = ('S', [ ('S',["aBA"]), ('A',["ba","B","S"]), ('B',["b","ab"]) ])

grammarX :: Grammar Char
grammarX = ('X', [ ('X',["E#"]), ('E',["E-T","T"]), ('T',["1","(E)"]) ])
```

We have added two example grammars which will be used throughout the paper: `grammarS` is an LR(1) grammar which happens to be useful for illustrating LR parsing concepts with; `grammarX` is an LR(0) grammar for expressions which end with a marker `#`. We consider only a very small subset of expressions: only one arithmetic operator, `-`, and one constant, `1`, are involved.

3 Bottom-up parsing

In this section we describe how one can recognise a sentence of the language generated by a grammar by making use of a bottom-up parser. A configuration of a bottom-up parser consists of a stack of symbols, which represents the sentential form which has been recognised so far, and a sequence of terminal symbols, which represents the symbols of the input sequence which have not been read yet. A bottom-up recogniser recognises an input sequence of terminal symbols by performing two kinds of actions:

1. **Shift:** An input symbol is shifted from the sequence of terminals onto the symbol stack.
2. **Reduce:** The top of the symbol stack, being equal to the alternative of a nonterminal, is reduced to that nonterminal.

The recognising process starts with a configuration consisting of an empty symbol stack (nothing recognised so far) and the complete input sequence (no terminals read so far). The recognising process accepts an input sequence in a configuration consisting of a symbol stack which contains only the start symbol (a sentence of the language has been recognised) and an empty sequence of terminals (all input symbols have been read). The recognising process can, of course, also fail if the input sequence is not a sentence of the language of the grammar. In the following section we show some typical shift-reduce transition sequences which recognise strings which are generated by `grammarS`. We separate the symbol stack and the sequence of terminals by a dot.

3.1 Recognising `abb`, `abba`, `abab` and `ababab`

In the first example we show how the string `abb` is recognised using a shift-reduce transition sequence.

```
.abb  shift a
a.bb  shift b
ab.b  reduce b to B
aB.b  shift b
aBb.  ! reduce b to B !
aBB.  reduce B to A
aBA.  reduce aBA to S
S.
```

During the fifth transition we have reduced `b` to `B`. Sometimes it does not make sense to reduce in a similar situation. Instead it is necessary to keep on shifting input symbols. This is shown in the following example.

```
.abba  shift a
a.bba  shift b
ab.ba  reduce b to B
aB.ba  shift b
aBb.a  ! shift a !
aBba.  reduce ba to A
aBA.   reduce aBA to S
S.
```

In this example we can not reach an accepting configuration any more if we reduce `b` to `B` during the fifth transition. Shifting an `a` does lead to an accepting configuration. How can we decide whether to shift or reduce? The sentential form recognised so far does not provide us with enough information. We will see later on that, in this case, it does not make sense to reduce `b` to `B` since the whole input sequence has not been shifted yet.

In the third example we show how the string `abab` is recognised using a shift-reduce transition sequence.

```
.abab  shift a
a.bab  shift b
ab.ab  reduce b to B
aB.ab  shift a
aBa.b  shift b
aBab.  ! reduce ab to B !
aBB.   reduce B to A
aBA.   reduce aBA to S
S.
```

In this example we have reduced `ab` to `B` during the sixth transition. Sometimes it is necessary to perform another reduction in a similar situation. This is shown in the following example.

```
.ababab  reduce aBA to S
a.babab  shift b
ab.abab  reduce b to B
aB.abab  shift a
aBa.bab  shift b
aBab.ab  ! reduce b to B !
aBaB.ab  shift a
aBaBa.b  shift b
aBaBab.  reduce ab to B
aBaBB.   reduce B to A
aBaBA.   reduce aBA to S
aBS.     reduce S to A
aBA.     reduce aBA to S
S.
```

In this example we can not reach an accepting configuration any more if we reduce `ab` to `B`. Reducing `b` to `B` does lead to an accepting configuration. How can we decide how to reduce? Again, the sentential form recognised so far does not provide us with enough information. We will see later on that, in this case, it does not make sense to reduce `ab` to `B` since the whole input sequence has not been shifted yet.

4 LR(0) machine

In section 6 we present a functional parser for LR(0) grammars, which are grammars for which the shift-reduce and reduce-reduce conflicts of the foregoing section do not occur. The running example of that section is `grammarX`. Here is a shift-reduce transition sequence for recognising the sentence `1-1-1#` (recall that `#` serves as an end marker).

```
.1-1-1#  shift 1
1.-1-1#  reduce 1 to T
T.-1-1#  reduce T to E
E.-1-1#  shift -
E-.1-1#  shift 1
E-1.-1#  reduce 1 to T
E-T.-1#  reduce E-T to E
E.-1#    shift -
E-.1#    shift 1
```

```
E-1.#    reduce 1 to T
E-T.#    reduce E-T to E
E.#      shift #
E#.      reduce E# to X
X
```

One can think of a configuration like `E-.1#` as follows: we have already recognised a sentential form `E-` and we still have to read `1#` from the input sequence. Recognising a sentence of the language of `grammarX` is the same as recognising the sentential form `X`. We will construct a deterministic machine which does this job for us. The machine starts in a state where we try to recognise the start symbol (in this case `X`) as one of its alternatives (in this case only one : `E#`) and we have not read anything yet. We indicate this situation using a, so called, item `X->.E#`. A state of the machine is represented as a set of such items. For the moment the start state contains only one item. But that is not the whole story. If we want to proceed with recognising an `X`, as `E#` then we first have to recognise an `E`. This can be done in two different ways: as `E-T` or as `T`. We indicate this fact by adding the items `E->.E-T` and `E->.T` to the start state. Similarly, if we want to proceed with recognising an `E`, then we first have to recognise an `E` or a `T`. The first case has already been dealt with. Recognising a `T` can be done in two different ways: as `1` or as `(E)`. We indicate this fact by adding the items `T->.1` and `T->.(E)` to the start state. We are done now with the start state: all nonterminals which need to be recognised before we can proceed have been dealt with. We end up with the following start state (numbered as state `2`):

```
(2) { X->.E# , E->.E-T , E->.T , T->.1 , T->.(E) }
```

Next we have to specify what happens if we proceed after the dot. Note that we can both read terminals and recognise nonterminals. In the start state we can read a `(` or `1`, or recognise a `T` or `E`. If we recognise a `(`, then we are in a state which contains the item `T->(.E)`: during the process of recognising a `T` as `(E)` we have already recognised `(`. We have to add new items to the state we are in: if we want to proceed, then we first have to recognise an `E`. This observation leads to a state in which we have (just as it was the case for the start state) four additional items, ending up with:

```
(1) { T->(.E) , E->.E-T , E->.T , T->.1 , T->.(E) }
```

In a similar (but simpler) way we end up with states:

```
(9) { T->1. }
(7) { E->T. }
(4) { X->E.# , E->E.-T }
```

after reading `1`, and recognising `T` and `E` respectively. States `9` and `7` are states which contain items which have a dot at the end (an alternative of a nonterminal has been recognised). They are reduce states (the items with a dot at the end are called reduce items). We will deal with reduce states later on. Let's first continue with states `1` and `4`. In state `1` we can recognise an `E` or a `T`, or read a `(` or a `1`. If we recognise an `E`, then we end up in:

```
(3) { T->(E.) , E->E.-T }
```

if we read a `(` or a `1` or recognise a `T`, then we end up in (already visited) states `1`, `9` and `7` respectively. In state `4` we can read a `-` or a `#`. In the first case we obtain an item `E->E-.T`, which, after dealing with how the nonterminal `T` can be recognised, gives us a state:

```
(5) { E->E-.T , T->.1 , T->.(E) }
```

In the second case we obtain a reduce state:

```
(10) { X->E#. }
```

We still have to explain what happens in states `3` and `5`. If, in state `3`, we read a `-`, then we go to the (already visited) state `5`, and if we read a `)`, then we go to the reduce state:

```
(8) { T->(E). }
```

Finally, if, in state `5`, we recognise a `T`, then we arrive at the reduce state:

```
(6) { T->E-T. }
```

and if we read a `1` or a `(`, then we end up in (already visited) states `9` and `1` respectively. This finishes our LR(0) machine. A figure of the machine can be found in Figure 1 of appendix A.

We have not explained yet how we have to proceed when we are in a reduce state, where we have recognised the alternative of a nonterminal. In such a state we have to go back along the path which we used to arrive at the state. The labels of this path can be found in the second part of the reduce item, (the symbols before the dot). Afterwards we read the nonterminal. This nonterminal can be found in the first part of the item (the symbol before the arrow). For example: in state `8` we can go back to state `1`, via state `3` and state `1` and afterwards we go to state `7` (there are other possibilities as well). The netto result is that we did a transition from state `8` to `7` by reducing `(E)` to `T`.

There is an important fact to notice about this LR(0) machine: all reduce states contain only one item. This means that there are no shift-reduce or reduce-reduce conflicts (in other words **`grammarX`** is an LR(0) grammar). The LR(0) machine which is presented in this section does not differ from the ones which are presented in standard textbooks which deal with LR parsing (for example [1]). The way in which we implement the LR(0) machine differs substantially. This difference will be dealt with in section 5 and section 6.

5 LR(0) data structure

The paths which can be followed through the LR(0) machine of section 4 can be represented as follows:

```
data XPath nX nE nT
  = XP_1_1 Char (XPath nX nE nT)
  | XP_1_9 Char (XPath nX nE nT)
  | XP_1_3 nE (XPath nX nE nT)
  | XP_1_7 nT (XPath nX nE nT)
  | XP_2_1 Char (XPath nX nE nT)
  | XP_2_9 Char (XPath nX nE nT)
  | XP_2_4 nE (XPath nX nE nT)
  | XP_2_7 nT (XPath nX nE nT)
  | XP_2_0 nX (XPath nX nE nT)
  | XP_3_8 Char (XPath nX nE nT)
  | XP_3_5 Char (XPath nX nE nT)
  | XP_4_10 Char (XPath nX nE nT)
  | XP_4_5 Char (XPath nX nE nT)
  | XP_5_1 Char (XPath nX nE nT)
  | XP_5_9 Char (XPath nX nE nT)
  | XP_5_6 nT (XPath nX nE nT)
  | XEmpty
```

The type variables **`nX`**, **`nE`** and **`nT`** correspond to the nonterminals `X`, `E` and `T`. Let's look at some of the alternatives of this data structure. The transition `XP_2_1 Char` indicates that one can go from state 2 to state 1 by recognising a character (in this case a (). The transition XP_1_7 nT indicates that one can go from state 1 to state 7 by recognising a T. How this is realised is not visible in the data structure. We have seen in section 4 that this can be done by going from state 1 to state 1 (by reading a (), then going from state 1 to state 3 (by reconising an E), then going from state 3 to state 8 (by reading a)), and finally going from state 8 to state 7 (by reducing (E) to T). Note that we have added one extra state (state 0). We are in an accepting configuration if we have recognised the start symbol right from the start state and have read all the input symbols. If there is no outgoing transition from the start state which is labelled with the start symbol (for the LR(0) machine of grammarX this happens to be the case) then we have to add an extra state, called empty state, in which we can reach such a final configuration.

6 LR(0) parser code

In this section we show how the LR(0) parser code corresponding to an LR(0) machine looks like. The parsing code takes into account semantic actions which have to be performed when reducing (i.e. when having recognised something meaningful). The XPath data type of section 5 acts as a typed stack onto which we do not only remember the states which we have visited so far, but onto which we also gradually compute a final parsing result of type nX while reducing. With every grammar corresponds, in a natural way, an algebra of semantic actions which can be used to compute this final result. With grammarX corresponds an algebra which consists of three tuples of functions, one for each nonterminal of grammarX. Each tuple of functions has the same number of components as the number of alternatives for the nonterminal. It is convenient to give this algebra a name.

```
type XAlgebra nX nE nT
  = ((nE -> Char -> nX),                       -- X = E#
     (nE -> Char -> nT -> nE, nT -> nE),       -- E = E-T | T
     (Char -> nT, Char -> nE -> Char -> nT)) -- T = 1 | (E)
```

The algebra of semantic actions will be the first argument of the LR(0) parser code. The code manipulates an input sequence. Instead of using a symbol stack to represent the configuration of the LR(0) machine with (as is the case for the normal bottom-up parsing algorithm), we use a path which acts as a transition stack as described above. The idea then is to define a function stateX which expects a path and an input sequence. The function stateX makes use of local functions state_n, where n corresponds to the state the LR(0) machine is in. The function stateX is then defined as state_s where s is the number of the start state (in our case s is 2). In other words: the LR(0) parser code looks as follows:

```
stateX ((x1), (e1, e2), (t1, t2))
 = state_2
  where
      ...
   state_old oldpath oldinput = state_new newpath newinput
      ...
```

We are now ready for the local state transition functions state_n. We'll pick out four typical transitions. First we consider state_2 (which is not a reduce state).

```
state_2 path ('(':input) = state_1 (XP_2_1 '(' path) input
state_2 path ('1':input) = state_9 (XP_2_9 '1' path) input
state_2 _ _ = error "state_2"
```

An input pattern is used to distinguish between the two different configurations of the LR(0) machine (the path parameter is not important here). Notice that we remember transitions by pushing them on top of the transition stack. When shifting we do not compute an intermediate result. The last equation deals with input which does not start with an appropriate character.

Next we consider `state_8` (which is a reduce state). There are three possible ways to reduce `(E)` to `T`, depending upon the transitions which are used to reach the state. This leads to the following code:

```
state_8 (XP_3_8 _ (XP_1_3 e (XP_1_1 _ path))) input
 = state_7 (XP_1_7 (t2 '(' e ')') path) input
state_8 (XP_3_8 _ (XP_1_3 e (XP_2_1 _ path))) input
 = state_7 (XP_2_7 (t2 '(' e ')') path) input
state_8 (XP_3_8 _ (XP_1_3 e (XP_5_1 _ path))) input
 = state_6 (XP_5_6 (t2 '(' e ')') path) input
state_8 _ _ = error "state_8"
```

In this case a path pattern is used to distinguish between the three different configurations of the LR(0) machine (the input parameter is not important here). Notice that, again, we remember transitions by pushing them on top of the transition stack (after having popped appropriate transitions from it). When reducing we do compute an intermediate result.

Finally we consider states `10` and `0`. What happens if we have recognised the whole right hand side `E#` of the start symbol `X`? Just as with the other reduce states we go back (to state `2`) and recognise an `X`. Since there is no outgoing arrow labelled with an `X` we go to the empty state (in our case state `0`). This results in:

```
state_10 (XP_4_10 _ (XP_2_4 e path)) input
 = state_0 (XP_2_0 (x1 e '#') path) input
state_10 _ _ = error "state_10"
```

In state `0` we check if we have read all input and if we arrived there right from the start state by recognising an `X`. If this is the case, then we return the value on top of the transition stack as the final result of our computation. In other words:

```
state_0 (XP_2_0 x XEmpty) [] = x
state_0 _ _ = error "state_0"
```

The example grammar which we used in this section is LR(0). It is possible (for example in `state_8`) that there exist different paths along which one can reach a reduce state. The path pattern is used to determine which next state to choose. The code which we generate can be used for grammars which are more general than LR(0) grammars. Reduce-reduce conflicts are allowed as long as one can use the path pattern to determine which reduction to choose.

6.1 Semantic actions

The idea is, of course, that all the code of section 5 and section 6 is generated from the definition of the grammar. The only code which has to be written is the one for the specific algebra of semantic actions we have in mind (the code of general semantic actions, like returning the recognised sentence, can, of course, also be generated from the grammar). Here is a typical algebra which is used to evaluate an expression of `grammX`.

```
xAlgebra :: XAlgebra Int Int Int
xAlgebra = ((\x _ -> x),(\x _ z -> x - z,\x -> x),(\_ -> 1,\_ y _ -> y))

? stateX xAlgebra XEmpty "1-(1-1)-1#"
-1
```

Note that the result of evaluating the expression is indeed `-1` because the left recursive rule for `E` in `grammX` specifies that `-` is left associative. Also note that assignments, like `$$ = $1 - $3`, which are typically used as semantic actions in `Yacc` code are now replaced by anonymous functions, like `\x1 _ x3 -> x1 - x3`.

7 LR(1) parser code

In this section we return to `grammS`. A figure of the LR(0) machine can be found in Figure 2 of appendix B. The LR(0) parser code which is generated for states `4` and `10` looks like:

```
state_4 path ('a':input)
 = state_5 (SP_4_5 'a' path) input
state_4 (SP_1_4 _ (path)) input
 = state_2 (SP_1_2 (b1 'b') path) input
state_4 _ _ = error "state_4"

state_10 (SP_6_10 _ (SP_1_6 _ path)) input
 = state_2 (SP_1_2 (b2 'a' 'b') path) input
state_10 (SP_6_10 _ path) input
 = state_1 (SP_6_1 (b1 'b') path) input
state_10 _ _ = error "state_10"
```

The machine has a an overlapping path-input pattern pair in state `4`. This means that there is a shift-reduce conflict: we can either reduce `b` to `B` or shift an `a`. The machine has an overlapping path pattern in state `10`. This means that there is a reduce-reduce conflict: we can either reduce `b` to `B` or reduce `ab` to `B`. We need some more information about the LR(0) machine in order to generate patterns which are non-overlapping. So far we have only used information from the past. It turns out that we can also use information from the future. We can find out which terminal symbols (if any) we can shift after having performed a sequence of reductions starting with a conflicting reduction (the sets of such symbols are called lookahead sets). Reduce-reduce conflicts can be resolved if the lookahead sets of conflicting reductions are disjoint. Shift-reduce conflicts can be resolved if the lookahead sets of reductions do not contain any symbol which can be shifted. Indeed, we can incorporate the lookahead information in the generated code by using, for every conflicting reduction and every terminal symbol `t` of its lookahead set, an input pattern `t:input` (if the lookahead set is empty, then we use the input pattern `[]`). For our example this leads to the following LR(1) parser code:

```
state_4 path ('a':input)
  = state_5 (SP_4_5 'a' path) input
state_4 (SP_1_4 _ path) []
 = state_2 (SP_1_2 (b1 'b') path) []
state_4 _ _ = error "state_4"

state_10 (SP_6_10 _ (SP_1_6 _ path)) []
 = state_2 (SP_1_2 (b2 'a' 'b') path) []
state_10 (SP_6_10 _ path) ('a':input)
 = state_1 (SP_6_1 (b1 'b') path) ('a':input)
state_10 (SP_6_10 _ path) ('b':input)
 = state_1 (SP_6_1 (b1 'b') path) ('b':input)
state_10 _ _ = error "state_10"
```

It turns out that, in `state_4` we can only reduce `b` to `B` if the input sentence has completely been read. Similarly, in `state_10` we can only reduce `ab` to `B` if the input sentence has completely been read. It is instructive to have a look back to the examples of section 3: they deal precisely with the different situations which are described here.

The `Yacc` parser generator works for so called LALR(1) grammars. For a grammar to be LALR(1) it is required that the lookahead sets of *all* reduce items of a reduce state are disjoint. We only have to consider reduce items which have overlapping path patterns. Grammars for which this is the case are called LR(1) grammars. Our parser generator can handle all such LR(1) grammars. Also notice that using patterns avoids a state explosion. The number of states of the LR(1) parser is the same as the number of states of the LR(0) parser. We use pattern matching to resolve conflicts only in those cases where it is strictly necessary.

8 Generating LR(1) parser code

A grammar contains, implicitly, enough information to be able to generate the code which is explained in section 7. It is convenient to make this information more explicit by defining local data structures. The actual dumping of the code can then be done much more easily from those structures. We do not go into the details of how to dump the code. This can be done by making use of pretty printing library. We made use of an especially convenient one (see [3]). Instead we will concentrate on some of the techniques which we have used to generate the local data structures.

The process of computing the LR(0) machine from the grammar is an iterative one: we keep on computing new transitions until we do not find any new states. The states themselves are also computed as an iterative process: we keep on adding new items to the state until we do not find any new items. Finally, computing the lookahead sets of conflicting reduce items is also an iterative process. We keep on going back along reduce states until no new reduce states are encountered. The lookahead set of the reduce item is then the union of all symbols which can be shifted from those reduce states. An iterative process can, of course, be implemented in different ways. We do this using the function `iterate`.

```
limit :: Ord x => (Set x -> Set x) -> (Set x -> Set x)
limit more start = head [ lim | (lim,[]) <- iterate next (start,start) ]
  where next (old,todo) = (new `putAllIn` old,new)
          where new = old `getAllOut` more todo
```

The functions `putAllIn` and `getAllOut` map sets to sets. Using the function `limit` we only have to concentrate on the function `more` which generates new elements from the ones which are newly created in a previous iteration.

8.1 Generating the LR(0) machine

This subsection shows how to generate the sets of items which constitute the states of the LR(0) machine. Items are implemented as triples:

```
type Item s
  = (s    -- nonterminal
    ,[s]  -- part of an alternative before .
    ,[s]) -- part of an alternative after .

type State s = Set (Item s)
```

If a set of items contains an item which is such that its third component (that part of an alternative of a nonterminal which has not been recognised yet) starts with a nonterminal, then we have to add new items to the set which correspond to all the alternatives of the nonterminal. This idea is encode in the following function:

```
moreItems [] = []
moreItems ((nont,done,[]):items) = more items
moreItems ((nont,done,(s:_)):items)
 = newItems `putAllIn` moreItems items
  where
   newItems
    | isN s = [ (s,[],alt) | alts <- tab ? s, alt <- alts ]
    | otherwise = []
```

Using moreItems we can compute the start state and the next state function of the LR(0) machine as follows:

```
startState g@(start,tab)
  = limit moreItems
     (mkSet [ (start,[],alt) | alts <- tab?start, alt <- alts  ])

nextState s items
  = limit moreItems
     (mkSet [ (nt,done++[s],todo) | (nt,done,(s':todo)) <- items, s == s' ])
```

The function mkSet turns a list into a set. The set of all states of the LR(0) machine is also a limit. We keep on computing new states (starting with the start state) until we do not find any new ones.

```
moreStates states
  = mkSet [ nextState s state | state <- states, s <- symbols ]

allStates = limit moreStates [startState]
```

9 A simple application

We end with giving a simple application. The idea is to write a fully automated program which, given a concrete representation of a grammar, generates a the code of a recogniser for the sentences of the grammar. The recogniser simply returns the sentence which it has recognised (or prints an error message if the sequence it tries to recognise is not a sentence of the language of the grammar). The first problem we have to deal with is to turn the concrete representation of a grammar (i.e. value of type String) into an abstract one (i.e. a value of type Grammar as defined in section 2). We can use the machinery of the paper to realise this. All we need to do is to define a concrete grammar for grammars, and to generate, from the concrete representation of a grammar, an LR(1) meta-grammar[1]. The LR(1) parser code of the meta-grammar can then, together with a meta-algebra (which can also be generated from the concrete representation of the grammar), be used to generate the abstract grammar corresponding to the concrete grammar with. This process (which always is a bit confusing at first sight) is sometimes referred to as bootstrapping or self application. Once we have done all this we can generate, from this generated (and, hopefully, LR(1)) grammar, LR(1) parser code and an algebra which can be used to recognise the sentences of the concrete grammar with. We refer the interested reader to the appendix, where he can find out how all this works for grammS.

[1] The meta terminology is only used to distinguish between the grammar we are trying to generate and the grammar for grammars which is used to do this job for us.

10 Conclusion

We have shown how to implement an LR(0) machine as a recursive data structure. This structure enables us to make use of simple pattern matching techniques to avoid the usage of traditional shift-reduce tables altogether. All this results in time-space efficient LR(1) parser code. We have showed how to make use of fixed point techniques to generate both the LR(0) machine and the lookahead sets of reduce items with. Our parser generator makes use of functional semantic actions which are in one to one correspondence with the imperative semantic actions of traditional parser generators like `Yacc`. In contrast with `Yacc`, which can only handle LALR(1) grammars, our grammar can handle all LR(1) grammars. We presented a typical application: generating a recogniser from a concrete representation of a grammar in a fully automatic way. We would like to end with the following message: using a declarative approach to LR(1) parsing has given us more insight in the very nature of LR(1) parsing. One of the authors (Luc Duponcheel) had only a limited amount of knowledge of LR(1) parsing. He felt uncomfortable with the classic imperative approach to the subject. Either one first builds a very large LR(1) shift-reduce state transition table and afterwards tries to reduce its size in some way or another, or one starts with an LR(0) shift-reduce state transition table and tries to split some of its states in one way or another. Both approaches are to explicit compared with the simple implicit pattern matching argument which is used in our declarative approach. In our opinion modern functional programming languages are ideal tools to bridge the gap between the theory and practice of grammars and parsing with in an elegant way. On the one hand they make it possible to approach the subject in a rigourous declarative way. On the other hand they offer executable code to experiment with.

References

[1] Aho, A.V. and Ullman J.D. *The Theory of Parsing, Translation, and Compiling: Vol. I. Parsing*, Englewood Cliffs, NJ: Prentice Hall, 1972.

[2] Aho, A.V. and Ullman J.D. *The Theory of Parsing, Translation, and Compiling: Vol. II. Compiling*, Englewood Cliffs, NJ: Prentice Hall, 1973.

[3] John Hughes, *Pretty Printing Combinators*, Proceedings of the First International Spring School on Advanced Functional Programming, Göteborg 1995.

[4] Mark P. Jones. Official release notes gofer 2.30, ftp://ftp.cs.nott.ac.uk/nott-fp/languages/gofer/,

[5] Jeroen Fokker, *Functional Programming*, Course notes, University Utrecht, 1995.

[6] Johan Jeuring, Luc Duponcheel, Doaitse Swierstra, *Grammars and Parsing*, Course notes, University Utrecht, 1995.

A LR(0) machine for `grammX`

```
(1)  { T->(.E) , E->.E-T , E->.T , T->.1 , T->.(E) }
(2)  { X->.E# , E->.E-T , E->.T , T->.1 , T->.(E) }
(3)  { T->(E.) , E->E.-T }
(4)  { X->E.# , E->E.-T }
(5)  { E->E-.T , T->.1 , T->.(E) }
(6)  { T->E-T. }
(7)  { E->T. }
(8)  { T->(E). }
(9)  { T->1. }
(10) { X->E#. }
```

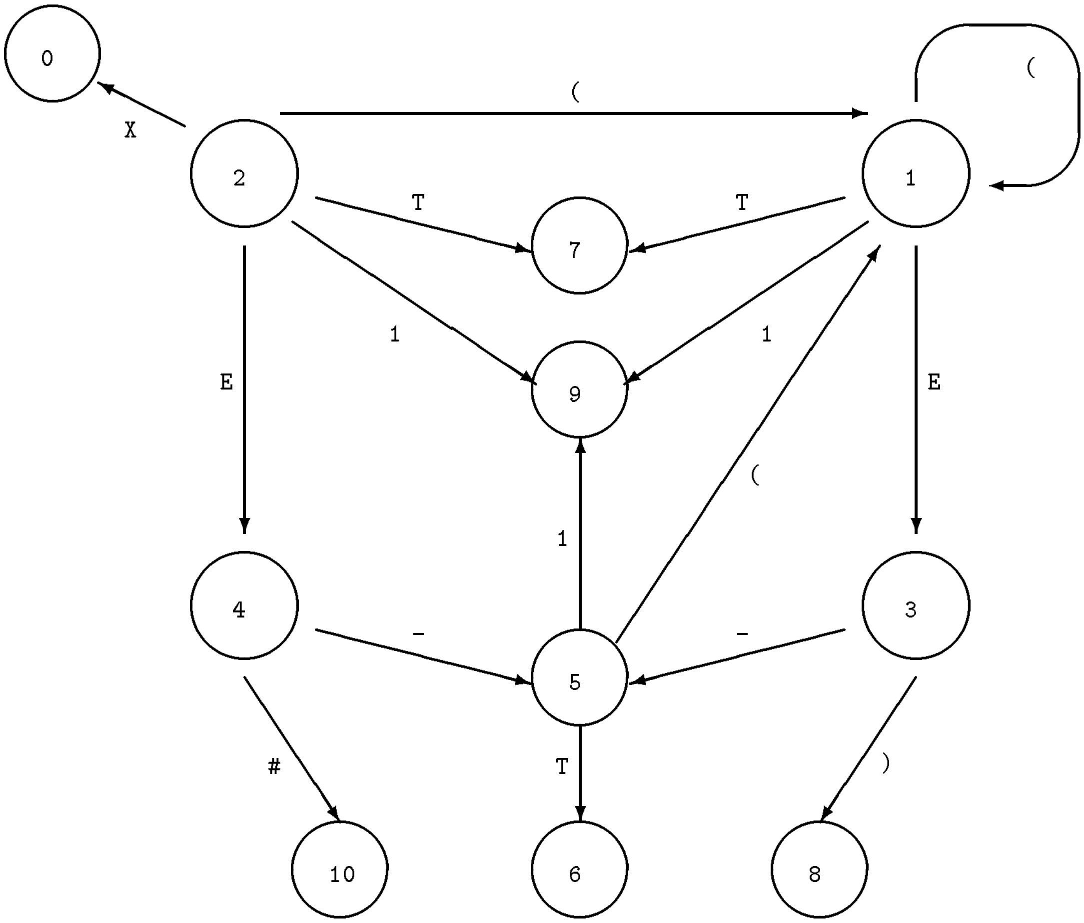

Figure 1: LR(0) machine for `grammX`

B LR(0) machine for `grammS`

```
(1)  { S->aB.A , A->.ba , A->.B , A->.S , B->.b , B->.ab , S->.aBA }
(2)  { A->B. }
(3)  { A->S. }
(4)  { A->b.a , B->b. }
(5)  { A->ba. }
(6)  { B->a.b , S->a.BA , B->.b , B->.ab }
(7)  { S->a.BA , B->.b , B->.ab }
(8)  { B->a.b }
(9)  { B->ab. }
(10) { B->ab. , B->b. }
(11) { B ->b. }
(12) { S->.aBA }
(13) { S->aBA. }
```

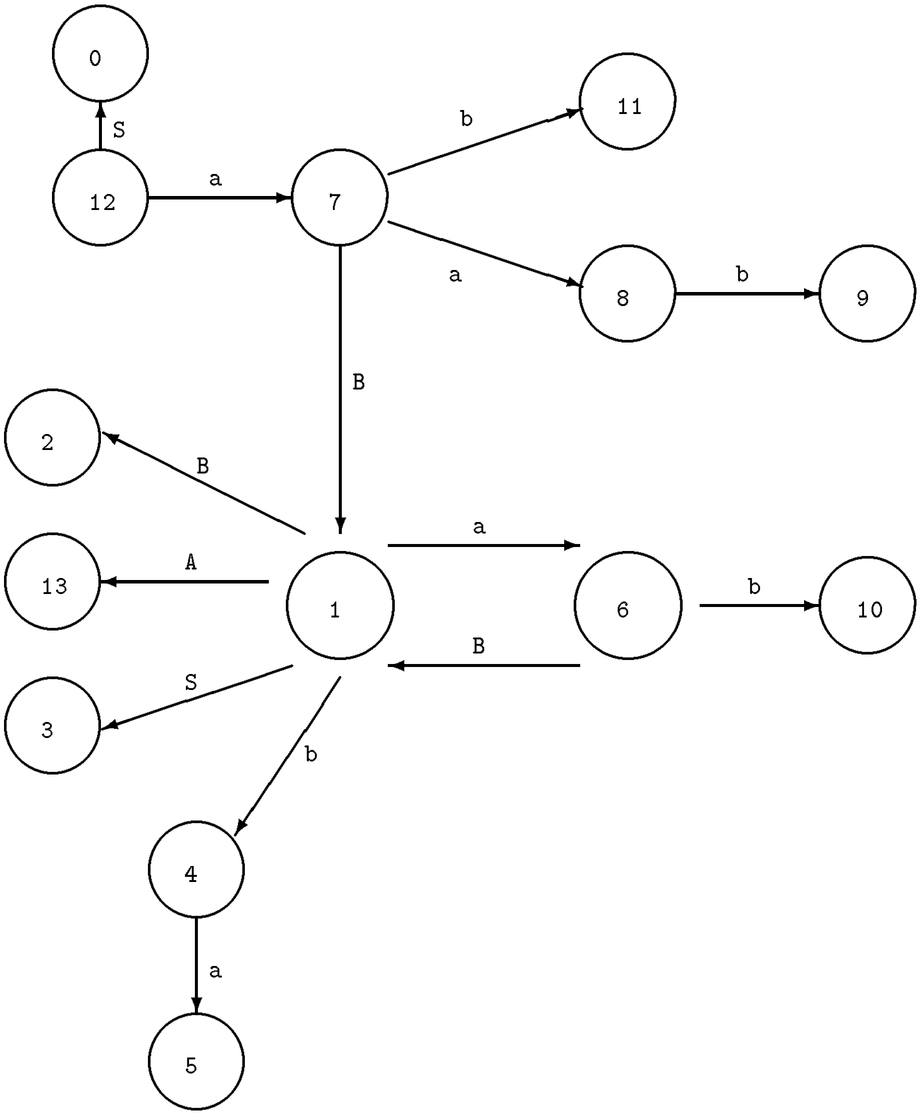

Figure 2: LR(0) machine for `grammS`

C Generated code

In this appendix we present some of the code which is generated by our program. The code is dumped using the pretty printing combinators of John Hughes (see [3]). They turn out to be an excellent rapid prototyping tool. We show how to generate a recogniser for sentences of the language generated by a concrete representation of `grammS` which looks like:

```
S,

S = aBA ;

A = ba
  | B
  | S ;

B = b
  | ab ;
```

Notice that we used the characters of the string `",=;|"` as meta-symbols. The meta-grammar and meta-algebra look like:

```
data Sym s = N String | T s

grammS
  = (N "GrammS"
    , [(N "GrammS", [[N "Nont",T ',',N "Rules"]])
      ,(N "Rules", [[N "RuleSC"], [N "RuleSC",N "Rules"]])
      ,(N "RuleSC", [[N "Rule",T ';']])
      ,(N "Rule", [[N "Nont",T '=',N "Alts"]])
      ,(N "Alts", [[N "Alt"], [N "Alt",T '|',N "Alts"]])
      ,(N "Alt", [[N "Sym"], [N "Sym",N "Alt"]])
      ,(N "Nont", [ [T u] | u <- nont ])
      ,(N "Sym", [ [T u] | u <- sym ])]
    )
   where
    nont = nub (filter isN "S,S=aBA;A=ba|B|S;B=b|ab;")
    sym = nub (filter (not . (`elem` metasym)) "S,S=aBA;A=ba|B|S;B=b|ab;")
    metasym = ",=;|"

grammSAlgebra
  = ((gramm1),
     (rules1, rules2),
     (rulesc1),
     (rule1),
     (alts1, alts2),
     (alt1, alt2),
     (nont1, nont2, nont3),
     (sym1, sym2, sym3, sym4, sym5))
   where
    gramm1 nt _ tab = (nt,tab)

    rules1 r = r:[]
    rules2 r rs = r:rs
```

```
rulesc1 r _ = r

rule1 nt _ alts = (nt,zip [1..] alts)

alts1 alt = alt:[]
alts2 alt _ alts = alt : alts

alt1 s = s:[]
alt2 s ss = s:ss

nont1 _ = 'S'
nont2 _ = 'B'
nont3 _ = 'A'

sym1 _ = 'S'
sym2 _ = 'a'
sym3 _ = 'B'
sym4 _ = 'A'
sym5 _ = 'b'
```

This code is highly independent from the actual grammar which we have used. The grammar is only used for the alternatives of `N "Nont"` and `N "Sym"` in the meta-grammar and for the semantic actions `nont1` ... `nont3` and `sym1` ... `sym5` in the meta-algebra. The LR(1) parser code for the grammar which is generated by the LR(1) code of this meta-grammar and this meta-algebra looks as follows (for the convenience of the user we have also dumped some shift-reduce and reduce-reduce information):

```
type SAlgebra nS nA nB
  = ((Char -> nB -> nA -> nS),
     (Char -> Char -> nA, nB -> nA, nS -> nA),
     (Char -> nB, Char -> Char -> nB))

data SPath nS nA nB
  = SP_1_13 nA (SPath nS nA nB)
  | SP_1_2 nB (SPath nS nA nB)
  | SP_1_3 nS (SPath nS nA nB)
  | SP_1_6 Char (SPath nS nA nB)
  | SP_1_4 Char (SPath nS nA nB)
  | SP_4_5 Char (SPath nS nA nB)
  | SP_6_1 nB (SPath nS nA nB)
  | SP_6_8 Char (SPath nS nA nB)
  | SP_6_10 Char (SPath nS nA nB)
  | SP_7_1 nB (SPath nS nA nB)
  | SP_7_8 Char (SPath nS nA nB)
  | SP_7_11 Char (SPath nS nA nB)
  | SP_8_9 Char (SPath nS nA nB)
  | SP_12_0 nS (SPath nS nA nB)
  | SP_12_7 Char (SPath nS nA nB)
  | SEmpty

{-

    LR(1) information:
```

```
    possible shift-reduce conflicts:

      state_4

      shiftsymbols = [a]
      lookaheadsymbols B -> b = []

    possible reduce-reduce conflicts:

      state_10

      lookaheadsymbols B -> b = [a, b]
      lookaheadsymbols B -> ab = []

-}

    stateS ((s1), (a1, a2, a3), (b1, b2))
     = state_12
      where

       state_1 path ('a':input)
        = state_6 (SP_1_6 'a' path) input
       state_1 path ('b':input)
        = state_4 (SP_1_4 'b' path) input
       state_1 _ _ = error "state_1"

       state_2 (SP_1_2 b path) input
        = state_13 (SP_1_13 (a2 b) path) input
       state_2 _ _ = error "state_2"

       state_3 (SP_1_3 s path) input
        = state_13 (SP_1_13 (a3 s) path) input
       state_3 _ _ = error "state_3"

       state_4 path ('a':input)
        = state_5 (SP_4_5 'a' path) input
       state_4 (SP_1_4 _ path) []
        = state_2 (SP_1_2 (b1 'b') path) []
       state_4 _ _ = error "state_4"

       state_5 (SP_4_5 _ (SP_1_4 _ path)) input
        = state_13 (SP_1_13 (a1 'b' 'a') path) input
       state_5 _ _ = error "state_5"

       state_6 path ('a':input)
        = state_8 (SP_6_8 'a' path) input
       state_6 path ('b':input)
        = state_10 (SP_6_10 'b' path) input
       state_6 _ _ = error "state_6"

       state_7 path ('a':input)
        = state_8 (SP_7_8 'a' path) input
       state_7 path ('b':input)
        = state_11 (SP_7_11 'b' path) input
```

```
state_7 _ _ = error "state_7"

state_8 path ('b':input)
 = state_9 (SP_8_9 'b' path) input
state_8 _ _ = error "state_8"

state_9 (SP_8_9 _ (SP_6_8 _ path)) input
 = state_1 (SP_6_1 (b2 'a' 'b') path) input
state_9 (SP_8_9 _ (SP_7_8 _ path)) input
 = state_1 (SP_7_1 (b2 'a' 'b') path) input
state_9 _ _ = error "state_9"

state_10 (SP_6_10 _ (SP_1_6 _ path)) []
 = state_2 (SP_1_2 (b2 'a' 'b') path) []
state_10 (SP_6_10 _ path) ('a':input)
 = state_1 (SP_6_1 (b1 'b') path) ('a':input)
state_10 (SP_6_10 _ path) ('b':input)
 = state_1 (SP_6_1 (b1 'b') path) ('b':input)
state_10 _ _ = error "state_10"

state_11 (SP_7_11 _ path) input
 = state_1 (SP_7_1 (b1 'b') path) input
state_11 _ _ = error "state_11"

state_12 path ('a':input)
 = state_7 (SP_12_7 'a' path) input
state_12 _ _ = error "state_12"

state_13 (SP_1_13 a (SP_6_1 b (SP_1_6 _ path))) input
 = state_3 (SP_1_3 (s1 'a' b a) path) input
state_13 (SP_1_13 a (SP_7_1 b (SP_12_7 _ path))) input
 = state_0 (SP_12_0 (s1 'a' b a) path) input
state_13 _ _ = error "state_13"

state_0 (SP_12_0 s SEmpty) [] = s
state_0 _ _ = error "state_0"
```

The algebra which is used to return the recognised sentences of the generated grammar is defined locally in the function `recogniserS`:

```
recogniserS :: [Char] -> [Char]
recogniserS = stateS sAlgebra SEmpty
 where
  sAlgebra
   = ((s1), (a1, a2, a3), (b1, b2))
    where
     s1 _ b a = ['a'] ++ b ++ a

     a1 _ _ = ['b'] ++ ['a']
     a2 b = b
     a3 s = s

     b1 _ = ['b']
     b2 _ _ = ['a'] ++ ['b']
```