Conditional recursive grammars

Sergey Strukov 2014, version 1.00

1. Formal languages.

Developing computer technologies is based on using formal languages of different kinds and purposes. “Formal” means a language, whose syntax and semantic can be algorithmically defined. The first step of the language definition is a definition of texts, which are considered as formally correct. From the mathematical perspective we have an alphabet as a given finite set of *letters* and the set of all sequences of such letters – *words*. To define a language we need to select some subset of words. The selection must be constructive, i.e. there should be given a precise algorithm to determine if the given word belongs to the language. It is important, that this algorithm would be efficient. Another important thing is: the word processing must not only determine if the word is correct, but also give some information about meaning of this word. The usual meaning is a representation of the word as an *abstract syntax tree* (**AST**). Consider the following simple example.

Let the alphabet consists of the following letters: `x ( ) + *` . There is a way of language definition, which is called *context-free or recursive grammar*. In order to define a language such grammar introduce a finite set of variables, which are commonly called *non-terminals*. Each non-terminal has a finite set of associated *production rules*. Here is a grammar of the simple language **AMP**.

```
! A
 {
  M      : castA
  A + M  : opAdd
 }


M
 {
  P      : castM
  P * M  : opMul
 }


P
 {
  x      : opVar
  ( A )  : opBra
 }
```

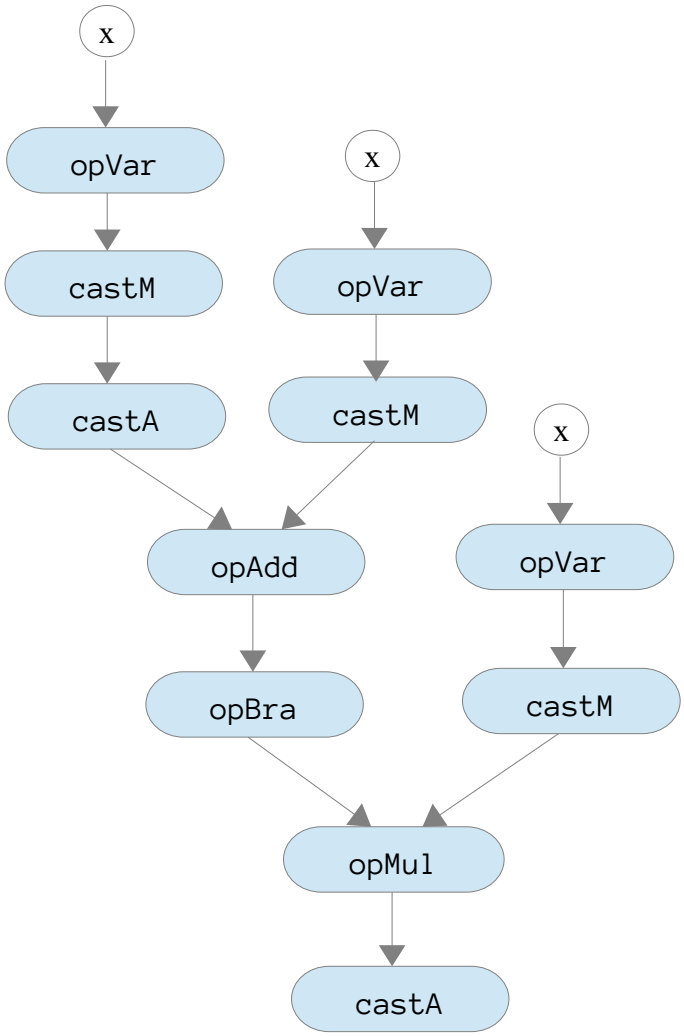
This grammar has tree non-terminals: `A M P`. Production rules are written inside figure brackets. Each rule has a name. It is given after `:` . The rule body is simply a sequence of letters or non-terminals. A grammar defines word sets per each non-terminal simultaneously and recursively. The informal meaning of a rule is: substitute each non-terminal with the word from the

correspondent non-terminal word set and you will receive a word from the non-terminal word set the rule belongs to. The word set for the non-terminal A, marked by **!**, constitutes the language.

For example, consider the following word: `( x + x ) * x`. It may be produced as following:

```
opVar() → x : P
castM( x ) → x : M
castA( x ) → x : A
opAdd( x , x ) → x + x : A
opBra( x + x ) → ( x + x ) : P
opMul( ( x + x ) , x ) → ( x + x ) * x : M
castA( ( x + x ) * x ) → ( x + x ) * x : A
```

Here words in brackets are substituted instead of non-terminals. This set of productions can be folded into the tree.



We will discuss recursive grammars and *conditional recursive grammars* and methods of *parsing* in the following sections. The language **AMP** will be used as the main example to make the narration more concrete and illustrative.

2. Recursive grammars and parsing. LR1 grammars.

Let us start with the terminology. We are using the term *atoms* instead of letters and s*yntax classes* (or simply *synts*) instead of non-terminals in the further narration. The second important thing is: don't assume that atoms are directly corresponds to the text letters. In the real examples of the formal language applications for the text processing the input text is processed first by a *tokenizer*. This process converts a text into the sequence of tokens and then these tokens are converted to atoms. Each atom belongs to some atom class and may have an associated value of some type. These values are essential for the text interpretation. For example, consider such text as:

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

It can be interpreted as the **AMP** word like this

```
( x[1] + x[2] ) * x[3]
```

Here values of each **x** is presented in square brackets. Then we can process the text above using usual arithmetic operations according the syntax tree, derived from the **AMP** word. From the programmers perspective we can imagine some calculation system. This system operates on values of the following types:

```
type_A

type_M

type_P

type_x
```

Production rules have correspondent functions:

```
type_P opVar(type_x);

type_P opBra(type_A);

type_A castA(type_M);

type_M castM(type_P);

type_A opAdd(type_A,type_M);

type_M opMul(type_P,type_M);
```

Then interpretation of the text can be given as the following formula:

```
type_A result = castA(opMul( opBra(opAdd( castA(castM(opVar(x[1]))) ,
castM(opVar(x[2])) )) , castM(opVar(x[3])) ));
```

An exact meaning of these types and functions is not important from the language perspective. They may build an **AST** representation of some kind or perform a direct arithmetic computation of the arguments.

Parsing means the restoration of the way how the given word is produced by the grammar. There is a convenient way of the plain representation of the **AST** and the definition of the parsing process. Consider the **AMP** grammar. Let's modify it by the following way:

```
! A
 {
  M      @castA : castA
  A + M @opAdd : opAdd
 }


M
 {
  P      @castM : castM
  P * M @opMul : opMul
 }


P
 {
  x      @opVar : opVar
  ( A ) @opBra : opBra
 }
```

We add a trailing special atom at the each rule. When we produce a word, using this extended grammar, these atoms allow us to restore the production process uniquely. For example,

```
( x @opVar @castM @castA + x @opVar @castM @opAdd ) @opBra * x @opVar
@castM @opMul @castA
```

Compare it with the **AST** graph above. If you erase the extended atoms, you will get the original word:

```
( x + x ) * x
```

So a *parsing* may be considered as the process of a *lifting* the given word to the extended one by the insertion of extended atoms. Once it's done, you have the **AST**, in the plane form. If this can be done uniquely, the grammar is called uniquely decoded. The main problem is to develop an algorithm to perform the parsing efficiently.

Let's back to the **AST** folding. Behold, the rule functions can be treated with extended lists of arguments.

```
type_P opVar(type_x);

type_P opBra(type_unused,type_A,type_unused);

type_A castA(type_M);

type_M castM(type_P);
```

```
type_A opAdd(type_A,type_unused,type_M);

type_M opMul(type_P,type_unused,type_M);
```

In practice we don't need some atoms to interpret a language word. So we eliminate the unused rule arguments. It can be done using the special type `type_unused` to express a value of such atom. If we have an extended word, we can calculate the correspondent function formula using the parsing stack. This stack is a polymorphic stack. It holds values of different types. The algorithm for the word processing is simple. Initially the stack is empty. We peak up atoms one-by-one and push them to the stack. But if the atom is extended, we don't push it, instead we apply the correspondent function to the top stack entries. These entries are pop from the stack, passed to the function and the result is pushed to the stack. Once we finish, the stack will contain the single element with the formula result. Consider this input:

```
( x[1] @opVar @castM @castA + x[2] @opVar @castM @opAdd ) @opBra * x[3]
@opVar @castM @opMul @castA
```

The stack evolution is:

```
(
( x[1]
( opVar(x[1])
( castM(opVar(x[1]))
( castA(castM(opVar(x[1])))
( castA(castM(opVar(x[1]))) +
( castA(castM(opVar(x[1]))) + x[2]
( castA(castM(opVar(x[1]))) + opVar(x[2])
( castA(castM(opVar(x[1]))) + castM(opVar(x[2]))
( opAdd(castA(castM(opVar(x[1]))),+,castM(opVar(x[2])))
( opAdd(...) )
opBra((,opAdd(...),))
opBra((,opAdd(...),)) *
opBra((,opAdd(...),)) * x[3]
opBra((,opAdd(...),)) * opVar(x[3])
opBra((,opAdd(...),)) * castM(opVar(x[3]))
opMul(opBra((,opAdd(...),)),*,castM(opVar(x[3])))
castA(opMul(opBra((,opAdd(...),)),*,castM(opVar(x[3]))))
```

Parsing stack can be used not only for the tree folding, but for the parsing itself also. In practice, these two tasks are performed simultaneously. Let's take the same input and split it into two parts:

```
( x[1] @opVar @castM @castA + x[2] @opVar @castM
and
@opAdd ) @opBra * x[3] @opVar @castM @opMul @castA
```

Then erase extended atoms in the second part.

```
) * x[3]
```

We can fold the first part using the stack.

```
( castA(castM(opVar(x[1]))) + castM(opVar(x[2]))
or
( A + M
```

What we need at the moment is to determine the next atom. It may be `)` or one of the extended atoms (we actually know, that in this example it is `@opAdd`). This atom is a function of two arguments:

```
Rule NextRule( Element* , Atom* )
```

Here `Atom` is the set of atoms, `Element` is the set of element. Elements are atoms and synts. The upper asterisk is a Kleene star. The stack contains a sequence of element. The first argument is the stack content and the second is the remaining atom sequence. `NextRule` returns either a rule or the special *shift rule* (`<-`). The shift rule pushes the next atom to the stack. We can define such function, if the grammar is uniquely decoded. This function is a *partial function*, i.e. it is defined only on some subset of pairs of arguments. If it is not defined for some pair, it means that this pair cannot appear from the correct input. The next task is to find an efficient algorithm to calculate this function.

There is a practically important class of grammars, called *LR1 grammars*. These grammars has efficient algorithm to calculate the function `NextRule`. By definition, the grammar is **LR1**, if `NextRule` depends only on the first atom of the second argument. I.e. there is a function

```
Rule NextRule1( Element* , Atom )
```

such as

```
NextRule( S , R ) = NextRule1( S , first_atom( R ) )
```

To handle the situation of the empty `R` we add the special atom `(End)`, such that `fist_atom(  ) = (End)`.

It is convenient to draw a table like this:

```
(End) : opAdd
*     : <-
+     : opAdd
```

This table defines the next rule for the given first argument. In our article we call such table *final*. The set of all finals is finite, so we can define `NextRule1` with means of the function

```
Final NextFinal( Element* )
```

For example, `NextFinal( ( A + M )` is

```
) : opAdd
+ : opAdd
```

If some atom is not listed in the final, it means this atom cannot appear in the correct input. In the example above only `)` and `+` may follow.

The key theorem claims that the function `NextFinal` can be calculated using some finite state machine. We will discuss this way of calculation in the next section.

Some additional comments. What if the grammar is not **LR1**? How can we determine if it is such? The answer is simple. For **any** grammar we can construct the function `NextFinal`(and the correspondent state machine). The only problem is, that some bad final entries may have multiple rules like:

```
) : opAdd <- (rule-shift conflict)
```

or

```
) : opAdd opMul (rule-rule conflict)
```

In this case the grammar is not **LR1**.

The similar definitions exist for grammars **LR2**, **LR3**, etc. An also there are state machines to calculate functions `NextFinal` for such grammars. The problem is the state machine size may grow so quickly, that it is not practically usable. That is why **LR1** is the most convenient class of grammars: such grammars are powerful enough for many situations and the same time they can be efficiently parsed.

3. State machines as calculators.

A *state machine* consists of the following parts: two sets S (states) and E (input elements), a selected state start, transition function S T(S,E). These data defines a function S $T^{ext}(E^*)$ on the sequences of input elements E^* as following:

$(e_1,e_2,...,e_n) \to T(...T(T(start,e_1),e_2)...,e_n)$.

Usually, S and E are finite and T can be defined by a finite table. So T^{ext} can be efficiently calculated. And so any function of the form $R \circ T^{ext}$, where R is any function, defined on S. For example, the function NextFinal can be calculated using some finite state machine, for this machine E is the set of elements, R is a function from states to finals.

There is a more convenient in practice variant of state machines. For this state machines the transition function T is a partial function, i.e. it is not defined for all pairs of arguments. Such function may require less space for the function representation. The extended function T^{ext} is also partial. If it is not defined for some input, we assign to this input some special resulting value. For example, if we deal with NextFinal, this special value is the empty final.

A partial state machine can be converted into the full by adding the special state error. All undefined transitions and all transitions from this state go to error.

State machine for the function NextFinal can be generated using special software.

2. Conditional recursive grammars.

Consider the following grammar:

```
! EXPR
 {
  x            : opVar
  ( EXPR )     : opBra
  EXPR + EXPR  : opAdd
  EXPR * EXPR  : opMul
 }
```

This grammar defines the same language as **AMP**, but it is not uniquely decoded. To make is such we have to elaborate the grammar, adding extra synts. Doing so we not only make the grammar uniquely decoded, but also elaborate the associativity and priority of operations `+` and `*`. `*` has a higher priority than `+`, `+` is left associative, `*` is right associative. An important note is: grammar is more informative, than the language it defines. It rules the language interpretation also.

This article is to introduce an improved variant of recursive grammars: *conditional recursive grammars*. This variant has the same "descriptive power", but more convenient in practice. The conditional recursive grammar of **AMP** is:

```
! EXPR : A , M , P
 {
  x                : opVar  = P
  ( EXPR )         : opBra  = P
  EXPR + EXPR.a    : if( a>=M ) opAdd  = A
  EXPR.a * EXPR.b  : if( a>=P & b>=M ) opMul  = M
 }
```

It looks similar to the original grammar, but contains some elaborations. First, the synt `EXPR` has three associated *kinds* : `A`, `M` and `P`. Then production rules has resulting kinds, given after `=`. Each production from a rule has the specified kind, assigned to it. Finally, some rules have conditions. This conditions allow rule to be applied only if the given synt argument's kinds satisfy the conditions. For example, the rule `opMul` requires the kind of the first expression to be `>=P` and the kind of the second to be `>=M`.

A conditional grammar can be reduced to an unconditional one. For this each kind is converted to the synt:

```
! EXPR.A
 {
  EXPR.a + EXPR.b  : opAdd.a.b   if( b>=M )
```

```
 }


! EXPR.M
 {
  EXPR.a * EXPR.b : opMul.a.b  if( a>=P & b>=M )
 }


! EXPR.P
 {
  x : opVar
  ( EXPR.a ) : opBra           if( true )
 }
```

Here lines with suffix of **if** form define a set of rules. The rule functions in this case are:

```
type_EXPR.A opAdd.A.M(type_EXPR.A,type_EXPR.M);
type_EXPR.A opAdd.A.P(type_EXPR.A,type_EXPR.P);
type_EXPR.A opAdd.M.M(type_EXPR.M,type_EXPR.M);
type_EXPR.A opAdd.M.P(type_EXPR.M,type_EXPR.P);
type_EXPR.A opAdd.P.M(type_EXPR.P,type_EXPR.M);
type_EXPR.A opAdd.P.P(type_EXPR.P,type_EXPR.P);
...
```

It is often, we can assume that:

```
type_EXPR.A == type_EXPR.M == type_EXPR.P
opAdd.A.M == opAdd.A.P == ...
```

In this case we can use only one rule function instead of six:

```
type_EXPR opAdd(type_EXPR,type_EXPR);
```

To accommodate this situation we are using the following extended language:

```
! EXPR.A
 {
  EXPR.a + EXPR.b @opAdd : opAdd.a.b   if( b>=M )
 }


! EXPR.M
```

```
 {
  EXPR.a * EXPR.b @opMul : opMul.a.b  if( a>=P & b>=M )
 }


! EXPR.P
 {
  x @opVar : opVar
  ( EXPR.a ) @opBra : opBra           if( true )
 }
```

To parse the input we are using the state machine to calculate the following function:

```
Final NextFinal( Element* )
```

The element set consists of atoms and synts EXPR.A, EXPR.M, EXPR.P. And finals have a form:

```
) : opAdd
+ : opAdd
```

The parsing stack contains expressions with kinds:

```
( EXPR[.A] + EXPR[.M]
```

When we apply a rule function we ignore kinds. The kind of the result if determined by the function, opAdd gives the kind A, opMul – the kind M and so on. The formula representation is

```
opMul( opBra(opAdd(opVar(x[1]),opVar(x[2]))) , opVar(x[3]) )
```

It is more compact, then the previous variant.

The stack evolutions for the word (x + x) * x is:

```
(
( x[1]
( opVar(x[1])[.P]
( opVar(x[1])[.P] +
( opVar(x[1])[.P] + x[2]
( opVar(x[1])[.P] + opVar(x[2])[.P]
( opAdd(opVar(x[1]),opVar(x[2]))[.A]
( opAdd(opVar(x[1]),opVar(x[2]))[.A] )
opBra(opAdd(opVar(x[1]),opVar(x[2])))[.P]
```

```
opBra(opAdd(opVar(x[1]),opVar(x[2])))[.P] *
opBra(opAdd(opVar(x[1]),opVar(x[2])))[.P] * x[3]
opBra(opAdd(opVar(x[1]),opVar(x[2])))[.P] * opVar(x[3])[.P]
opMul(opBra(opAdd(opVar(x[1]),opVar(x[2]))),opVar(x[3]))[.M]
```

It is also more compact, than the first variant.