GRM: Lisp-based syntax parser generator

USER'S MANUAL

Andrej Andrejev, Uppsala University, 2010

1. Introduction to context-free grammars

Context-free grammars are widely used in specifications of artificial languages. The formalism imposes important restrictions on the syntactic structure of a language, to enable effective automated processing, as well as certain measure of orderliness and tractability of the model. Although these restrictions are too severe for any natural language to be represented in context-free manner, most artificial languages for man/machine communication were deigned to as context-free from the very beginning.

According to [1], the *grammar* is defined as tetrad

$\langle T, N, s, R \rangle$

where T is the set of *terminal symbols*, those present in the language itself, N is the set of *non-terminal symbols*, designed as hierarchical syntactic abstractions, $s \in N$ is the *initial non-terminal*, corresponding to the largest syntactic object a correct input sequence is expected to represent, and R is the set of *rules* in the form of

$r_j : \quad n^j \to a_1^j ... a_{L(j)}^j$

where non-terminal symbol $n^j \in N$ on the *left side* is matched to sequence of symbols $a_i^j \in T \cup N$ of length $L(j)$, (possibly empty) on the *right side*. This kind of grammar is called *context-free* specifically because there is only one non-terminal on the left side of each rule.

A string of terminal symbols is considered *correct*, or *accepted*, if and only if it is possible to derive it from string consisting of one starting symbol s by successively substituting non-terminal symbols with sequences of symbols according to some rules.

Example: consider a simple grammar with terminals $T = \{C, V, +, *, (,)\}$, non-terminals $N = \{e, t, f\}$, starting symbol e and rules R

$r_1 : \quad e \to t + t$
$r_2 : \quad e \to t$
$r_3 : \quad t \to f * f$
$r_4 : \quad t \to f$
$r_5 : \quad f \to C$
$r_6 : \quad f \to V$
$r_7 : \quad f \to (e)$

This grammar defines a language of infix arithmetic expressions over C and V operands (their further meaning as constants and variables will be elaborated below), operators + and * and parentheses. A string $(V + C) * C$ can be derived like

$$e \xrightarrow{2} t \xrightarrow{3} f * f \xrightarrow{7,5} (e) * C \xrightarrow{1}$$
$$(t + t) * C \xrightarrow{4,4} (f + f) * C \xrightarrow{6,5} (V + C) * C$$

It is worth noting that these rule applications are only partially ordered, form independent branches and thus comprise a *parse tree*. Whenever more than one parse tree is possible for any given terminal string, the grammar, as syntactic model of the language, is considered *ambiguous*.

2. Usage of ascending parsers

The *syntactic parsing* is the reverse process to deriving a terminal string from initial non-terminal s. Given string as an input, right parts of the rules have to be systematically searched and substituted with their left parts, eventually converging to initial non-terminal if the input string is accepted or signaling *syntax error* otherwise. Those rule applications are called *reductions*, and the parse tree generated can be thought of as normal output.

Taken straightforwardly, the process is deterministic but computationally expensive, as it involves search with lots of backtracking, and becomes exponential if there are rules with empty right parts (so called *epsilon-rules*). However, context-free grammar restrictions allow for efficient linear-time non-backtracking parsing automata to be constructed for carefully designed unambiguous grammars.

If search is started from s, and the parse tree is constructed top-down, a *descending parser* is effectively implemented. If search is targeted towards s and the parse tree is constructed bottom-up - an *ascending parser* is at work. Theoretically, a parser may only return ACCEPT for correct strings and REJECT for others, as this involves actually constructing the parse tree.

In practice, however, each grammar rule is meaningful to the language designer, arbitrary data may be associated with symbols, and calculations over this data might be performed at the point of rule reductions. To formalize this, each rule in $r^j \in R$ has to be attributed a *reduction function*

$$\lambda^j\left(x_1,...,x_{L(j)}\right)$$

and for each symbol $a \in T \cup N$ each instance in the parse tree a_i will have *associated data* $\delta(a_i)$. The data of terminal symbols is always supplied to the parser; the data for non-terminals is to be calculated during the parsing process with λ^j functions. If the input string is accepted, let us return the value $\delta(s_0)$ of the final instance of s as the parser result.

Example: let us extend the grammar in section 1 with the following reduction functions, taking and returning sequences of symbols:

$$\lambda^1(x_1, x_2, x_3) = x_1 x_3 +$$
$$\lambda^3(x_1, x_2, x_3) = x_1 x_3 *$$
$$\lambda^7(x_1, x_2, x_3) = x_2$$

For other rules we attribute identity functions $\lambda^2, \lambda^4, \lambda^5, \lambda^6$, returning the value of their only argument. The derivation process from section 1 can be presented in backwards order as reduction process, and associated data can be shown under respective symbol instances:

$$(\underset{z}{V}+\underset{1}{C})*\underset{5}{C}\xrightarrow{6,5}(\underset{z}{f}+\underset{1}{f})*\underset{5}{C}\xrightarrow{4,4}(\underset{z}{t}+\underset{1}{t})*\underset{5}{C}$$

$$\xrightarrow{1}(\underset{z1+}{e})*\underset{5}{C}\xrightarrow{7,5}\underset{z1+}{f}*\underset{5}{f}\xrightarrow{3}\underset{z1+5*}{t}\xrightarrow{2}\underset{z1+5*}{e}$$

Note that the parsing process now effectively translates an infix arithmetic expression into postfix notation, suitable e.g. for stack-based calculators.

So, we define a parser as deterministic automaton [2] that takes a string of terminal symbol instances $t_1...t_m$ together with any associated data $\delta(t_i)$ as input, signals an error if string is not accepted, or performs all reductions in the order prescribed by syntactic parse tree and returns the data structure $\delta(s_0)$ associated with the root node of the tree. It is important that there should be at most one parse tree for any given input string, so the order of applying λ^j functions is well defined - that's why we need unambiguous grammars.

With GRM parser generator, when applied to unambiguous grammars, ascending parsers of class SLR(1) [2, p.515] are constructed. These parsers use one look-ahead input symbol as sufficient information to process any portion of input at any moment.

3. Lisp implementation

Lisp is by far the best suited language for syntax processing, as it naturally supports abstract symbols, dynamic value binding and code binding, general list and tree structures are easily created and manipulated within the basic terms of the language itself. Grammar analysis and parser code generation involves a lot of symbolic calculations, and any other programming language would essentially require a dynamic and recursive list processing infrastructure to be implemented first, which would probably converge to a light-weight lisp interpreter.

Instead, a full-scale lisp interpreter is already embedded in certain large, flexible and extendible software projects. The current version of GRM parser generator is written in ALisp dialect of Common Lisp, supported within AMOS II database mediator, being developed since 1992 at Uppsala Database Laboratory [3].

The whole parser functionality is defined in single file `ancend.lsp`, which should be loaded only for parser generation, as specified later in this chapter. No definitions are required to actually run the parser, save those used in REDUCE function as part of grammar definition.

3.1. Grammar specification format

There is lisp structure `grammar` defined

```
(defstruct grammar
  ts nts rules actions)
```

with default constructor `make-grammar`. `ts` and `nts` list disjoint sets of lisp symbols, representing terminals and non-terminals, and first symbol listed in `nts` is also treated as grammar's initial non-terminal. `rules` is the list of lists describing each grammar rule, where first symbol is the left part of the rule and the rest is the right part. `actions` list is of the same length as rules list, and each element is a function object with number of arguments equal to the length of right part of corresponding rule in `rules`. Integer values like `1` can be used as shorthand for simple Lambda-expression returning its argument with corresponding number.

Listing 1: GRM grammar specification

```
(defparameter ex1
  (make-grammar :ts '(C V + * lp rp)
                :nts '(<e> <t> <f>)
                :rules '((<e> <t> + <t>)
                         (<e> <t>)
                         (<t> <f> * <f>)
                         (<t> <f>)
                         (<f> C)
                         (<f> V)
                         (<f> lp <e> rp))
                :actions '(#'(lambda (a b c) (list '+ a c))
                           1
                           #'(lambda (a b c) (list '- a c))
                           1
                           1
                           1
                           2))
```

Listing 1 shows how the grammar in the above example can be defined as lisp structure. The only difference is that with this grammar the parser generated will effectively translate an infix expression into respective lisp S-expression.

3.2. Simplified grammar specification format

Since all non-terminals in practically useful context-free grammars occur in left parts of some rules, N set of the grammar can be inferred this way. The set A of symbols occurring in right-parts of the rules consists of all terminals and non-terminals, hence $T = A \setminus N$. Since the order rule specifications is up to language designer, one can require that the largest syntactic object is defined first, so that non-terminal that appears in left part of first rule listed will be considered the initial non-terminal.

So, the combined list of rules and their respective reduction functions is sufficient to specify a grammar - syntactic model of a language. The same grammar can be defined by a simple list in listing 2.

Listing 2: Johnson's grammar specification

```
(defparameter ex1s
  '((<e> -> <t> + <t> #'(lambda (a b c) (list '+ a c)))
    (<e> -> <t> 1)
    (<t> -> <f> * <f> #'(lambda (a b c) (list '* a c)))
    (<t> -> <f> 1)
    (<f> -> C 1)
    (<f> -> V 1)
    (<f> -> lp <e> rp 2)))
```

This format was suggested by Mark Johnson, [4]. Second element in each list is purely decorative, and the last element is always treated as function to call from inside the parser. In GRM parser generator there is a converter function `grammar-from-johnsons` to convert grammar definitions from latter format into former.

```
(grammar-from-johnsons ex1s)
```

will return the same lisp data structure as defined in listing 1.

3.3. Parser generation

There is a function `make-slr1-parser` which takes 4 arguments: the `grammar` structure defining the grammar, the name of the parser function to be defined, the string or lisp pathname pointing the file to be created, and `verbose` flag. The following call will generate parser `ex1-parser` definition in file "`ex1-parser.lsp`":

```
(make-slr1-parser
     ex1 "ex1-parser" "ex1-parser.lsp" nil)
```

Parser generated with non-nil verbose flag will print to lisp toploop all SHIFT and REDUCE operations it performs with respective rule numbers and data values. The flag is suitable for debugging purposes only and should not be used when generating production-stage parser.

The `make-slr1-parser` function returns `T` if grammar is unambiguous, and the file is successfully written. If an ambiguity found, (i.e. situation which makes possible more than one parse tree for some accepted input string), the SLR(1) parser cannot be constructed, and the error message is printed.

3.3.1. Resolving grammar ambiguities

In terms of stack-driven SLR(1) ascending parser, whenever after some input sequence $t_1...t_k$ and on observing some possible input symbol t_{k+1}, more than one rule can be applied. The stack symbol, input symbol, and conflicting actions are printed in the error message.

Though there are many possible language design faults that lead to grammar ambiguities, there are several most typical ones:

- defining the same non-terminal both as left-recursive and right-recursive

Example: the grammar in listing 3a tries to define expressions with both unary minus and binary associative subtraction, so that sequences like C, $-C$, $C-C$, $-C-C-C$ are all accepted and translated to corresponding lisp S-expressions.

Listing 3a: ambiguous grammar with combined left and right recursion

```
(defparameter ex2
  '((<s> -> - <s> #'(lambda (a b) (list '* '-1 b)))
    (<s> -> <s> - const #'(lambda (a b c) (list '- a c)))
    (<s> -> const 1)))
```

Trying to generate an ascending parser will encounter a conflict, pointed out with the following message:

```
Conflict detected at Key = (<S> (1 . 1) (2 . 3)), Input = - : SHIFT vs. REDUCE(1)
```

The `Key` is a grammar symbol annotated with list of index pairs, each pointing to an occurrence of that symbol in some rule, where first comes rule number, and second the position in that rule, counted right-to-left. Occurrences `(j . 1)` correspond to the states where parser is ready to REDUCE the contents of the stack, and push there left part n^j of rule `j`, provided the input is a terminal that may follow n^j in some valid input string. Other occurrences correspond to SHIFT-ing the input to the stack, provided the input symbol is compatible with the rule.

In this example, the parser might come to a state, (e.g. after reading $-C$ and reducing on rule 3), that observing $-$ terminal it can either:

- immediately SHIFT the input $-$ in accordance with rule 2
- further REDUCE on rule 1, and get new <s> symbol, so that $-$ may follow according to rule 2

Even though both of these may be "right" ideas from language designer standpoint, and they will produce equivalent S-expressions, the rule reduction order is different in these cases, so technically the ambiguity is present.

To resolve this problem one might introduce hierarchy among left-recursive and right-recursive syntax constructs, so that one of them will always be identified before another. This involves introducing new non-terminal <e> and new "identity" rule 2, as shown in listing 3b.

Listing 3b: resolved grammar with left and right recursions separated

```
(defparameter ex2a
  '((<e> -> - <e> #'(lambda (a b) (list '* '-1 b)))
    (<e> -> <s> 1)
    (<s> -> <s> - const #'(lambda (a b c) (list '- a c)))
    (<s> -> const 1)))
```

More ambiguity cases are discussed in Bison/Yacc manual:
http://www.gnu.org/software/bison/manual/bison.html#Reduce_002fReduce

3.4. Running the parser

After parser code is successfully generated, the specified text file will contain a pretty-printed and commented definition of lisp function `slr1-parser`, that takes one argument `input-fn`. This function of no arguments should be provided from parser invocation context. It is expected to return dotted pairs `(symbol . data)`, or `nil` as end-of-input signal.

For simple test purposes, the construct shown in listing 4 might be recommended. It mimics the input string from example in sections 2, and implies the parser is generated from grammar in section 3.1 or 3.2.

Listing 4: parser invocation

```
(let ((input '((lp) (V . z) (+) (C . 1) (rp) (*) (C . 5))))
  (slr1-parser (f/l () (pop input))))
```

References

[1] A.V. Aho, J.D. Ullman: "The Theory of Parsing, Translation and Compiling". Prentice-Hall, Vol. I 1972, Vol II 1973, http://portal.acm.org/citation.cfm?id=SERIES11430.578789

[2] P.M. Lewis, D.J. Rozenkrantz, R.E. Stearns: "Compiler Design Theory", Addison-Wesley 1976

[3] S.Flodin, M.Hansson, V.Josifovski, T.Katchaounov, T.Risch, M.Sköld, E.Zeitler:
Amos II User's Manual, http://user.it.uu.se/~udbl/amos/doc/amos_users_guide.html

[4] M.Johnson, LALR(1) parser generator in Lisp
http://compilers.iecc.com/comparch/article/91-03-044