Run-time Extensible Deterministic Top-Down Parsing

MICHAL ŽEMLIČKA and JAROSLAV KRÁL
Department of Software Engineering, Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic
E-mail: {zemlicka,kral}@ksi.mff.cuni.cz

Abstract. When reading a text or listening to speech, words are processed on-line by humans in the order they come. Humans mainly use this kind of parsing even when they process deterministic text (programs). Intuitively there are some mental actions just after the morphological analysis of any newly recognized word. This mental action helps to understand the given word (or to position the word within the frame of the – still not complete – sentence). Within parsing of formal languages the concept closest to this idea is top-down parsing that is usually used only together with different classes of LL grammars. The advantage of top-down parsing of programming languages is the possibility to implement it by a recursive descent parser – i.e. by a system of procedures that may recursively call each other. Such a system may be ‘tuned’ by handmade changes. The usage of LL grammars is not always possible, because the grammars of programming languages may have left recursive symbols. Programming language grammars are intuitively ‘close’ to LL grammars. A good model for such grammars are the kind grammars studied in this paper. Kind grammars preserve all the important features of LL grammars that are advantageous for parsing.

Key words: kind grammar, top-down parsing, extensible languages

1. Introduction

Note 1. If not specified otherwise, notation used is close to the one in Aho, Sethi and Ullman (1986).

Parsers are expected to have the following features (similarly as the other software): effectivity, robustness, simplicity of creation, maintenance, and modification. But what is behind these words? Under ‘easy to create’ we can understand both the potential usage of a constructor and the possibility to write a parser by hand easily.

Today constructors usually work for LALR(1), LR(1) and sometimes also for attributed LL(1) grammars. So the resulting parsers usually work for the above given grammar classes. In the first two cases they are implemented as a table (with some contents), in the third case they can be implemented as a set of procedures calling each other. In every case the resulting code usually does not look like the code that would be written by a programmer manually. The programmer can overcome some limitations of SLL\(^1\) grammars to obtain an enhancement of the parser.

Grammars 2: 283–293, 1999.
Table 1. The productions of a simple arithmetic expression grammar.

<table>
<thead>
<tr>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S \rightarrow E \cdot )</td>
</tr>
<tr>
<td>( E \rightarrow E + T \mid E - T \mid T \mid - T )</td>
</tr>
<tr>
<td>( T \rightarrow T \ast F \mid T \div F \mid T \mod F \mid F )</td>
</tr>
<tr>
<td>( F \rightarrow ( E ) \mid id \mid num )</td>
</tr>
</tbody>
</table>

This feature causes some trouble: it is really hard to change the already existing code because of its vastness. Under some circumstances such changes are needed – e.g. good error recovery is hard to imagine without some additional human work – see e.g. (Plátek, 1992). A similar observation holds for inserting or changing additional semantic actions.

Here we try to introduce tools for the construction of parsers that are structurally close to the parsers fully written by humans. A recursive descent parser consisting just of the procedures corresponding to the nonterminals will be constructed. A new class of grammars covering LL languages (but a bit more ‘tolerant’ than SLL grammars – i.e. left recursion and some other constructions are allowed) is presented. The grammars will be called kind. Let us note that kind grammars allow one to define syntax of most imperative programming languages, especially the syntax of arithmetic expressions and lists. Arithmetic expressions will be used as a ‘running example’.

2. Motivation

2.1. Basic Structure

Examples are the best way to visualize the principles of kind parsing. We will show how the productions may be ordered and which structures may be used for ordering kind grammar productions or for kind parsing. These structures will also be used by grammar extension mechanisms and by semantic actions.

Let us have the (kind) grammar (in Table 1) describing simple integer arithmetic expressions written in Backus-Naur form.

Productions are already divided into groups. Productions having the same nonterminal on the left-hand side form one group. Now these groups are divided further: one subgroup will collect only productions without any left recursion (NLRP), the second one will collect only direct left recursive productions (DLRP – as shown in Table 2).

In order to clarify the main idea let us concentrate only on one selected nonterminal. Let the selected nonterminal be \( E \). The \( E \)-productions can be graphically sorted e.g. like in Table 3.