GLL algorithm sketch, terminology and notation

This is a brief, informal overview of the GLL algorithm, assuming a familiarity with context free grammars and parsing. It is designed as a reference for use when reading example parsers. Fill in the detail by reading the full published material, an introduction to the GLL recognition algorithm can be found here.

In a GLL parser the parse functions associated with a traditional recursive descent (RD) parser are replaced with algorithm line labels, goto statements and an explicit stack. For each nonterminal *A* there is a labelled block of code corresponding to each alternate of *A* and a return label for each position immediately after each nonterminal instance in the context free grammar. Think of a GLL parser as traversing the grammar using the input string. At each step of the traversal, (i) if the grammar pointer is immediately before a terminal we attempt to match it to the current input symbol; (ii) if it is immediately before a nonterminal *B* then the pointer moves to the start of the block of code associated with *B* (equivalent to a function call in an RD parser); (iii) if it is at the end of an alternate of *A* then it moves to the position immediately after the instance of *A* from which it came (equivalent to a function return in an RD parser). At step (ii) the return label is pushed onto the stack and at step (iii) the stack is popped to retrieve the return label.

In general there can be multiple grammar traversals, and multiple corresponding stacks. These are combined into a *graph structured stack* (GSS), the stack elements are nodes of the graph and there is a directed edge from node *u* to node *v* if *u* is immediately above *v* on a stack. For a GLL parser the edges of the GSS are each labelled with a derivation tree node. This node will be the left child of the derivation tree node constructed when the associated subparse is complete. The GSS is built using the function create() and return labels are retrieved using the function pop().

The FIRST sets of alternates are used as guards, and a travesal continues into an alternate only if the current input symbol belongs to the guard set. The function test() performs this check.

Grammar traversals may fork at the start of a block of code for nonterminal *A* if more than one alternate of *A* has the current input symbol in its FIRST set. Each of the possible traversal threads is stored in a process descriptor, a 4-tupl (L,c_U,c_I,c_N). The descriptors waiting to be continued are held in a set ***R***, and a set of all the descriptors created is also maintained to avoid repetition. The GLL algorithm works by removing descriptors from ***R*** and continuing with the associated traversals, until ***R*** is empty. Descriptors are created by the function add().

A GSS is the result of merging many stacks so a node *u* is often the top element of several stacks. If a new child is added to *u* after it has been popped, then the pop has to be applied to the new child. It is not possible, in general, to order the actions so that no pop is applied until after all the children have been added, so this situation is handled by keeping a record of the pop actions that have been performed so that they can be applied to new children if necessary. A set ***P*** is used for this, elements are added to it by *pop()* and used by *create()*.

Derivation trees are constructed by a GLL parser as it traverses the grammar. The potentially multiple trees (the underlying grammar may be ambiguous) are combined into a *shared packed parse forest* (SPPF) which is partially binarised to ensure that it always of at most cubic size in the length of the input string. The functions getNodeT() and getNodeP() build the SPPF.

Here is a glossary of the notation that we commonly use in our algorithm descriptions.

Centre for Software Language Engineering, Royal Holloway, University of London