An Efficient Parser Generator for Natural Language

Masayuki ISHII*
Fujitsu Inc.
masayuki@nak.math.keio.ac.jp

Kazuhisa OHTA
Apple Technology, Inc.
k-ohta@kobo.apple.com

Hiroaki SAITO
Keio University
hxs@nak.math.keio.ac.jp

Abstract

We have developed a parser generator for natural language processing. The generator named "NLyacc" accepts grammar rules written in the Yacc format. NLyacc, unlike Yacc, can handle arbitrary context-free grammars using the generalized LR parsing algorithm. The parser produced by NLyacc efficiently parses given sentences and executes semantic actions. NLyacc, which is a free and sharable software, runs on UNIX workstations and personal computers.

1 Parser Generator for NLP

Yacc[4] was designed for unambiguous programming languages. Thus, Yacc can not elegantly handle a script language with a natural language flavor, i.e. Yacc forces a grammar writer to use tricks for handling ambiguities. To remedy this situation we have developed NLyacc which can handle arbitrary context-free grammars[1] and allows a grammar writer to write natural rules and semantic actions. Although there are several parsing algorithms for a general context-free language, such as ATN, CYK, and Earley, "the generalized LR parsing algorithm [2]" would be the best in terms of its compatibility with Yacc and its efficiency.

An ambiguous grammar causes a conflict in the parsing table, a state which has more than one action in an entry. The generalized LR parsing proceeds exactly the same way as the standard one except when it encounters a conflict. The standard deterministic LR parser chooses only one action in this situation. The generalized LR parser, on the other hand, performs all the actions in the multiple entry by splitting the parse stack for each action. The parser merges the divided stack branches, only when they have the same top state. This merger operation is important for efficiency. As a result, the stack becomes a graph instead of a simple linear state sequence.

There is already a generalized LR parser for natural language processing developed at Carnegie Mellon University [3]. NLyacc differs from CMU's system in the following points.

- NLyacc is written in C, while CMU's in Lisp.
- CMU's cannot handle ϵ rules, while NLyacc does. ϵ rules are handful for writing natural rules.
- The way to execute semantic actions differs. CMU's evaluates an LFG-like semantic action attached to each rule when reduce action is performed on that rule. NLyacc executes a semantic action in two levels; one is performed during parsing for syntactic control and the other is performed onto each successful final parse. We will describe the details of NLyacc's approach in the next section.

NLyacc is upper-compatible to Yacc. NLyacc consists of three modules; a reader, a parsing table constructor, and a drive routine for the generalized LR parsing. The reader accepts grammar rules in the Yacc format. The table constructor produces a generalized LR parsing table instead of the standard LR parsing table. We describe the details of the parser in the next section.

*This work was done while Ishii stayed at Dept. of Computer Science, Keio University, Japan.

[1]To be exact, NLyacc can not handle a circular rule like "A → A".

2 Execution of Semantic Actions

NLyacc differs from Yacc mainly in the execution process of semantic actions attached to each grammar rule. Namely, Yacc evaluates a semantic action as it parses the input. We examine if this evaluation mechanism is suitable for the generalized LR parsing here. If we can assume that there is only one final parse, the parser can evaluate semantic actions when only one branch exists on top of the stack. Although having only one final parse is often the case in practical applications, the constraint of being unambiguous is too strong in general.

2.1 Handling Side Effects

Next, we examine what would happen if semantic actions are executed during parsing. When a reduce action is performed, the parser evaluates the action attached to the current rule. As described in the previous section, the parse stack grows in a graph form. Thus, when the action contains side effects like an assignment operation to a variable shared by different actions, that side effect must not propagate to the other paths in the graph.

If an environment, which is a set of value of variables, is prepared to each path of the parse branches, such side effect can be encapsulated. When a stack splits, a copy of the environment should be created for each branch. When the parse branches are merged, however, each environment can not be merged. Instead, the merged state must have all the environments. Thus, the number of environments grows exponentially as parsing proceeds. Therefore this approach decreases the parsing efficiency drastically. Also this high cost operation would be vain when the parse fails in the middle. To sum it up, although this approach retains compatibility with Yacc, it sacrifices efficiency too much.

2.2 Two Kinds of Semantic Actions

We, therefore, take another approach to handling semantic actions in NLyacc. Namely, the parser just keeps a list of actions to be executed, and performs all the actions after parsing is done. This method can avoid the problem above during parsing. After parsing is done, the semantic action evaluator performs the task as it traces all the history paths one by one. This approach retains parsing efficiency and can avoid the execution of useless semantic actions. A drawback of this approach is that semantic actions can not control the syntactic parsing, because actions are not evaluated until the parsing is done. To compensate the cons above, we have introduced a new semantic action enclosed with [ ] to enable a user to discard semantically incorrect parses in the middle of parsing.

Namely, there are two types of semantic actions:

- An action enclosed with [ ] is executed during parsing just as done in Yacc. If '*return* 0;' is executed in the action, the partial parse having invoked this action fails and is discarded.
- An action enclosed with { } is executed after the syntactic parsing.

In the example below, the bracketed action checks if the subtraction result is negative, and, if true, discards its partial parse.

```
number : number '-' number
   [ $$ = $1-$3; if($$ < 0) return 0; ]
   { $$ = $1-$3; print("-", $1, $3, $$); }
```

2.3 Keeping Parse History

Our generalized LR parsing algorithm is different from the original one [2] in that our algorithm keeps a history of parse actions to execute semantic actions after the syntactic parsing. The original algorithm uses a packed forest representation for the stack, whereas our algorithm uses a list representation.

The algorithm of keeping the parse history is shown as follows.

1) If the next action is "*shift* s", then make < s > as the history, where < s > is a list of only one element s.

2) If the next action is "*reduce* r : $A \rightarrow B_1 B_2 \cdots B_n$", then make append($H_1, H_2, ..., H_n, [-\mathrm{r}]$) as the history, where H_i is a history of B_i, r is the rule number of production "$A \rightarrow B_1 B_2 \cdots B_n$", and the function 'append' concatenates multiple lists and returns the result.

Now we describe how to execute semantic actions using the parse history. First, before starting to parse, the parser calls "yyinit" function to initialize variables in the semantic actions. Our system requires the user to define "yyinit" to set initial values to the variables. Next, the parser starts parsing and performs a shift action or a reduce action according to the parse history and evaluates the appropriate semantic actions.

2.4 Efficient Memory Management

We use a list structure to implement the parse stack, because the stack becomes a complex graph structure as described previously. Because the parser discards failed branches of the stack, the system reclaims the memory allocated for the discarded parses using the "mark and sweep garbage collection algorithm [1]" to use memory efficiently. This garbage collection is triggered only when the memory is exhausted in our current implementation.

3 Distribution

Portability

Currently, NLyacc runs on UNIX workstations and DOS personal computers.

Debugging Grammars

For grammar debugging, NLyacc provides parse trace information such as a history of shift/reduce actions, execution information of '[ ] actions.'

When NLyacc encounters an error state, "yyerror" function is called just as in Yacc.

Distribution

NLyacc is distributed through e-mail (please contact `nlyacc@nak.math.keio.ac.jp`). Distribution package includes all the source codes, a manual, and some sample grammars.

References

[1] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part 1. *Communications of the ACM*, 3(4), April 1960.

[2] M. Tomita. *Efficient Parsing for Natural Language.* Kluwer Academic Publishers, Boston, MA, 1985.

[3] M. Tomita and J. G. Carbonell. The universal parser architecture for knowledge-based machine translation. In *Proceedings, 10th International Joint Conference on Artificial Intelligence (IJCAI)*, Milan, August 1987.

[4] yacc - yet another compiler-compiler: parsing program generator. *in UNIX manual.*

Appendix – Sample Runs –

A sample grammar below covers a small set of English sentences. The parser produces syntactic trees of a given sentence. Agreement check is done by the semantic actions.

```
/* grammar.y */
%{
#include <stdio.h>
#include <stdlib.h>
#include "grammar.h"
#include "proto.h"
%}

%token NOUN VERB DET PREP

%%
SS : S          { pr_tree($1); }

S : NP VP       [ return check1($1, $2); ]
                { $$ = mk_tree2("S", $1, $2); }

S : S PP        { $$ = mk_tree2("S", $1, $2); }

NP : NOUN       [ $$ = $1; ]
                { $$ = mk_tree1("NP", $1); }

NP : DET NOUN [ $$ = $2; return check2($1, $2);]
                { $$ = mk_tree2("NP", $1, $2); }

NP : NP PP      [ $$ = $1; ]
                { $$ = mk_tree2("NP", $1, $2); }

PP : PREP NP    { $$ = mk_tree2("PP", $1, $2); }

VP : VERB NP    [ $$ = $1; ]
                { $$ = mk_tree2("VP", $1, $2); }
%%
FILE* yyin;
extern int yydebug;

int main(argc, argv)
     int argc;
     char *argv[];
{
    int result;

    yydebug = 1;
```

```
    yyin = stdin;
    read_dictionary("dict");
    yyinitialize_heap();
    result = yyparse();
    printf("Result = %d\n", result);
    yyfree_heap();
    return 0;
}

void yyinit()
{}

int yyerror(message)
     char* message;
{
    fprintf(stderr, "%s\n", message);
    exit(1);
}

int check1(sem1, sem2)
     SEMPTR sem1, sem2;
{
    return (sem1->seigen & sem2->seigen);
}

int check2(sem1, sem2)
     SEMPTR sem1, sem2;
{
    return (sem1->seigen & sem2->seigen);
}

/* grammar.h */
#define SPELLING_SIZE   32
#define HINSHI_SIZE     32
#define BUFFER_SIZE     64

typedef struct word
{
    struct word *next;
    char *spelling;
    int hinshi;  /* parts of speech */
    int seigen;  /* constraints */
} WORD;

typedef enum tag
{
    TLEAF, TNODE
} TAG;

typedef struct node
{
    TAG tag;
    union {
   WORD* _leaf;
      struct {
         char *_pos;
         struct node *_left;
         struct node *_right;
      } _pair;
   } contents;
} NODE, *NODEPTR;

#define leaf contents._leaf
#define pos contents._pair._pos
#define left contents._pair._left
#define right contents._pair._right

typedef WORD SEM, *SEMPTR;

#define YYSTYPE NODEPTR

#define YYSEMTYPE SEMPTR

/* dict */
I:NOUN:01
You:NOUN:22
you:NOUN:22
He:NOUN:04
he:NOUN:04
She:NOUN:04
she:NOUN:04
It:NOUN:04
it:NOUN:04
We:NOUN:10
we:NOUN:10
They:NOUN:40
they:NOUN:40
see:VERB:73
sees:VERB:04
a:DET:07
the:DET:77
with:PREP:00
telescope:NOUN:07
man:NOUN:07
```

Sample Runs

```
# sentence no.1
He sees a man with a telescope ^D
# parse 1
S:(S:(NP:(NOUN:He)
      VP:(VERB:sees
          NP:(DET:a NOUN:man)))
   PP:(PREP:with NP:(DET:a
                     NOUN:telescope)))
# parse 2
S:(NP:(NOUN:He)
   VP:(VERB:sees
       NP:(NP:(DET:a NOUN:man)
           PP:(PREP:with
               NP:(DET:a NOUN:telescope)))))

# sentence no.2
He see a man ^D
# The semantic actions prune syntactically-
# sound but semantically-incorrect parses.
```