YAEP (Yet Another Earley Parser) - C interface

Vladimir Makarov, `vmakarov@gcc.gnu.org`

Oct 10, 2015

This document describes YAEP (Yet Another Earley Parser) written in C/C++.

1 YAEP

YAEP is an abbreviation of Yet Another Earley Parser. The package 'YAEP' implements earley parser. The earley parser implementation has the following features:

- It is sufficiently fast and does not require much memory. This is the fastest implementation of Earley parser which I know. The main design goal is to achieve speed and memory requirements which are necessary to use it in prototype compilers and language processors. It parses 30K lines of C program per second on 500 MHz Pentium III and allocates about 5Mb memory for 10K line C program.
- It makes simple syntax directed translation. So an abstract tree is already the output of YAEP.
- It can parse input described by an ambiguous grammar. In this case the parse result can be an abstract tree or all possible abstract trees. Moreover it produces the compact representation of all possible parse trees by using DAG instead of real trees. This feature can be used to parse natural language sentences.
- It can parse input described by an ambiguous grammar according to the abstract node costs. In this case the parse result can be an minimal cost abstract tree or all possible minimal cost abstract trees. This feature can be used to code selection task in compilers.
- It can make syntax error recovery. Moreover its error recovery algorithms finds error recovery with minimal number of ignored tokens. It permits to implement parsers with very good error recovery and reporting.
- It has fast startup. There is no practically delay between processing grammar and start of parsing.
- It has flexible interface. The input grammar can be given by YACC-like description or providing functions returning terminals and rules.
- It has good debugging features. It can print huge amount of information about grammar, parsing, error recovery, translation. You can even output the result translation in form for a graphic visualization program.

The interface part of the parser is file 'yaep.h'. The implementation part is file 'yaep.c'. The interface contains the following external definitions and macros:

Struct 'grammar'

describes a grammar. Knowledge of structure (implementation) of this type is not visible and not needed for using the parser.

Macro 'YAEP_NIL_TRANSLATION_NUMBER'

is reserved to be designation of empty node for translation.

Macro 'YAEP_NO_MEMORY'

is error code of the parser. The parser functions return the code when parser can not allocate enough memory for its work.

Macro 'YAEP_UNDEFINED_OR_BAD_GRAMMAR'

is error code of the parser. The parser functions return the code when we call parsing without defining grammar or call parsing for bad defined grammar.

Macro 'YAEP_DESCRIPTION_SYNTAX_ERROR_CODE'

is error code of the parser. The code is returned when the grammar is defined by description and there is syntax error in the description.

Macro 'YAEP_FIXED_NAME_USAGE'

is error code of the parser. The code is returned when the grammar uses reserved names for terminals and nonterminals. There are two reserved names '$S' (for axiom) and '$eof' for end of file (input end marker). The parser adds these symbols and rules with these symbols to the grammar given by user. So user should not use these names in his grammar.

Macro 'YAEP_REPEATED_TERM_DECL'

is error code of the parser. The code is returned when the grammar contains several declarations of terminals with the same name.

Macro 'YAEP_NEGATIVE_TERM_CODE'

is error code of the parser. The code is returned when the grammar terminal is described with negative code.

Macro 'YAEP_REPEATED_TERM_CODE'

is error code of the parser. The code is returned when the two or more grammar terminals are described with the same code.

Macro 'YAEP_NO_RULES'

is error code of the parser. The code is returned when the grammar given by user has no rules.

Macro 'YAEP_TERM_IN_RULE_LHS'

is error code of the parser. The code is returned when grammar rule given by user contains terminal in left hand side of the rule.

Macro 'YAEP_INCORRECT_TRANSLATION'

is error code of the parser. The code is returned when grammar rule translation is not correct. The single reason for this is translation of the rule consists of translations of more one symbols in the right hand side of the rule without forming an abstract tree node.

Macro 'YAEP_NEGATIVE COST'

is error code of the parser. The code is returned when abstract node has a negative cost.

Macro 'YAEP_INCORRECT_SYMBOL_NUMBER'

is error code of the parser. The code is returned when grammar rule translation contains incorrect symbol number which should be nonnegative number less than rule right hand side length.

Macro 'YAEP_UNACCESSIBLE_NONTERM'

is error code of the parser. The code is returned when there is grammar nonterminal which can not be derived from axiom.

Macro 'YAEP_NONTERM_DERIVATION'

is error code of the parser. The code is returned when there is grammar nonterminal which can not derive a terminal string.

Macro 'YAEP_LOOP_NONTERM'

is error code of the parser. The code is returned when there is grammar nonterminal which can derive only itself. The parser does not work with such grammars.

Macro 'YAEP_INVALID_TOKEN_CODE'

is error code of the parser. The code is returned when the parser got input token whose code is different from all grammar terminal codes.

Enumeration 'yaep_tree_node_type'

describes all possible nodes of abstract tree representing the translation. There are the following enumeration constants:

'YAEP_NIL'

the corresponding node represents empty translations.

'YAEP_ERROR'

the corresponding node represents translation of special terminal 'error' (see error recovery).

'YAEP_TERM'

the corresponding node represents translation of a terminal.

'YAEP_ANODE'

the corresponding node represents an abstract node.

'YAEP_ALT'

the corresponding node represents an alternative of the translation. Such nodes creates only when there are two or more possible translations. It means that the grammar is ambiguous.

Structure 'yaep_tree_node'

represents node of the translation. The nodes refer for each other forming DAG (direct acyclic graph) in general case. The main reason of generating DAG is that some input fragments may have the same translation, when there are several parsings of input (which is possible only for ambiguous grammars). But DAG may be created even for unambigous grammar because some nodes (empty and error nodes) exist only in one exemplar. When such nodes are not created, the translation nodes forms a tree. This structure has the following members:

Member 'type' of type 'enum yaep_tree_node_type'

representing type of the translation node.

Union 'val'

Depending on the translation node type, one of the union members 'nil', 'error', 'term', 'anode', and 'alt' of the structure types described below is used to represent the translation node.

Structure 'yaep_nil'

represents empty node. It has no members. Actually the translation is DAG (not tree) in general case. The empty and error nodes are present only in one exemplar.

Structure 'yaep_error'

represents translation of special terminal 'error'. It has no members. The error node exists only in one exemplar.

Structure 'yaep_term'

represents translation of terminals. It has the following two members:

Integer member 'code'

representing code of the corresponding terminal.

Member 'attr' of type '* void'

is reference for the attribute of the corresponding terminal.

Structure 'yaep_anode'

represents abstract node. It has the following two members:

Member 'name' of type 'const char *'

representing name of anode as it given in the corresponding rule translation.

Member 'cost' of type 'int'

representing cost of the node plus costs of all children if the cost flag is set up. Otherwise, the value is cost of the abstract node itself.

Member 'children' of type 'struct yaep_tree_node **'

is array of nodes representing the translations of the symbols given in the rule with the abstract node.

Structure 'yaep_alt'

represents an alternative of the translation. It has the following two members:

Member 'node' of type 'struct yaep_tree_node *'

representing alternative translation.

Member 'next' of type 'struct yaep_tree_node *'

is reference for the next alternative of translation.

Function 'yaep_create_grammar'

```
'struct grammar *yaep_create_grammar (void)'
```

should be called the first. It actually creates an yaep parser with undefined grammar. You can use two or more parsers simultaneously. The function returns 'NULL' if there is no memory.

Function 'yaep_error_code'

```
'int yaep_error_code (struct grammar *g)'
```

returns the last occurred error code (see the possible error codes above) for given parser. If the function returns zero, no error was found so far.

Function 'yaep_error_message'

```
'const char *yaep_error_message (struct grammar *g)'
```

returns detail message about last occurred error. The message always corresponds to the last error code returned the previous function.

Function 'yaep_read_grammar'

```
'int yaep_read_grammar (struct grammar *g, int strict_p,
                        const char *(*read_terminal) (int *code),
                        const char *(*read_rule)
                                       (const char ***rhs,
                                        const char **abs_node,
                                        int *anode_cost,
                                        int **transl))'
```

is one of two functions which tunes the parser to given grammar. The grammar is read with the aid functions given as parameters.

'read_terminal' is function for reading terminals. This function is called before function 'read_rule'. The function should return the name and the code of the next terminal. If all terminals have been read the function returns NULL. The terminal code should be nonnegative.

'read_rule' is function called to read the next rule. This function is called after function 'read_terminal'. The function should return the name of nonterminal in the left hand side of the rule and array of names of symbols in the right hand side of the rule (the array end marker should be 'NULL'). If all rules have been read, the function returns 'NULL'. All symbol with name which was not provided function 'read_terminal' are considered to be nonterminals. The function also returns translation given by abstract node name and its fields which will be translation of symbols (with indexes given in array given by parameter 'transl') in the right hand side of the rule. All indexes in 'transl' should be different (so the translation of a symbol can not be represented twice). The end marker of the array should be a negative value. There is a reserved value of the translation symbol number denoting empty node. It is value defined by macro 'YAEP_NIL_TRANSLATION_NUMBER'. If parameter 'transl' is 'NULL' or contains only the end marker, translations of the rule will be empty node. If 'abs_node' is 'NULL', abstract node is not created. In this case 'transl' should be null or contain at most one element. This means that the translation of the rule will be correspondingly empty node or the translation of the symbol in the right hand side given by the single array element. The cost of the abstract node if given is passed through parameter 'anode_cost'. If 'abs_node' is not 'NULL', the cost should be greater or equal to zero. Otherwise the cost is ignored.

There is reserved terminal 'error' which is used to mark start point of error recovery.

Nonzero parameter 'strict_p' value means more strict checking the grammar. In this case, all nonterminals will be checked on ability to derive a terminal string instead of only checking axiom for this.

The function returns zero if it is all ok. Otherwise, the function returns the error code occured.

Function ‘yaep_parse_grammar’

```
‘int yaep_parse_grammar (struct grammar *g, int strict_p,
                         const char *description)’
```

is another function which tunes the parser to given grammar. The grammar is given by string ‘description’. The description is similiar YACC one. It has the following syntax:

```
file : file terms [’;’]
     | file rule
     | terms [’;’]
     | rule

terms : terms IDENTIFIER [’=’ NUMBER]
      | TERM

rule : IDENTIFIER ’:’ rhs [’;’]

rhs : rhs ’|’ sequence [translation]
    | sequence [translation]

sequence :
         | sequence IDENTIFIER
         | sequence C_CHARACTER_CONSTANT

translation : ’#’
            | ’#’ NUMBER
            | ’#’ ’-’
            | ’#’ IDENTIFIER [NUMBER] ’(’ numbers ’)’

numbers :
        | numbers NUMBER
        | numbers ’-’
```

So the description consists of terminal declaration and rules sections.

Terminal declaration section describes name of terminals and their codes. Terminal code is optional. If it is omitted, the terminal code will the next free code starting with 256. You can declare terminal several times (the single condition its code should be the same).

Character constant present in the rules is a terminal described by default. Its code is always code of the character constant.

Rules syntax is the same as YACC rule syntax. The single difference is an optional translation construction starting with ‘#’ right after each alternative. The translation part could be a single number which means that the translation of the alternative will be the translation of the symbol with given number (symbol numbers in alternative starts with 0). Or the translation can be empty or ‘-’ which mean empty node. Or the translation can be abstract node with given name, optional cost, and with fields whose values are the translations of the alternative symbols with numbers given in parentheses after the abstract node name. You can use ‘-’ in abstract node to show that empty node should be used in this place. If the cost is absent it is believed to be one. The cost of terminal, error node, and empty node is always zero.

There is reserved terminal 'error' which is used to mark start point of error recovery.

Function 'yaep_set_lookahead_level'

```
'int yaep_set_lookahead_level (struct grammar *grammar,
                               int level)'
```

sets up level of usage of look ahead in parser work. Value zero means no usage of lookaheads at all. Lookahead with static (independent on input tokens) context sets in parser situation (value 1) gives the best results with the point of view of space and speed, lookahead with dynamic (dependent on input tokens) context sets in parser situations (all the rest parameter values) does slightly worse, and no usage of lookaheads does the worst. The default value is 1 (lookahead with static situation context sets). The function returns the previously set up level. If the level value is negative, zero is used instead of it. If the value is greater than two, two is used in this case.

Function 'yaep_set_debug_level'

```
'int yaep_set_debug_level (struct grammar *grammar,
                           int level)'
```

sets up level of debugging information output to 'stderr'. The more level, the more information is output. The default value is 0 (no output). The debugging information includes statistics, result translation tree, grammar, parser sets, parser sets with all situations, situations with contexts. The function returns the previously set up debug level. Setting up negative debug level results in output of translation for program 'dot' of graphic visualization package 'graphviz'.

Function 'yaep_set_one_parse_flag'

```
'int yaep_set_one_parse_flag (struct grammar *grammar,
                              int flag)'
```

sets up building only one translation tree (parameter value 0) or all parse trees for ambiguous grammar for which several parsings are possible. For unambiguous grammar the flag does not affect the result. The default value is 1. The function returns the previously used flag value.

Function 'yaep_set_cost_flag'

```
'int yaep_set_cost_flag (struct grammar *grammar, int flag)'
```

sets up building only translation tree (trees if we set up one_parse_flag to 0) with minimal cost. For unambiguous grammar the flag does not affect the result. The default value is 0. The function returns the previously used flag value.

Function 'yaep_set_error_recovery_flag'

```
'int yaep_set_error_recovery_flag (struct grammar *grammar,
                                   int flag)'
```

sets up internal flag whose nonzero value means making error recovery if syntax error occurred. Otherwise, syntax error results in finishing parsing (although function 'syntax_error' in function 'yaep_parse' will be called once). The default value is 1. The function returns the previously used flag value.

Function 'yaep_set_recovery_match'

```
'int yaep_set_recovery_match (struct grammar *grammar,
                              int n_toks)'
```

sets up recovery parameter which means how much subsequent tokens should be successfully shifted to finish error recovery. The default value is 3. The function returns the previously used flag value.

Function 'yaep_parse'

```
'int yaep_parse (struct grammar *grammar,
                 int (*read_token) (void **attr),
                 void (*syntax_error)
                      (int err_tok_num, void *err_tok_attr,
                       int start_ignored_tok_num,
                       void *start_ignored_tok_attr,
                       int start_recovered_tok_num,
                       void *start_recovered_tok_attr),
                 void *(*parse_alloc) (int nmemb),
                 void (*parse_free) (void *mem),
                 struct yaep_tree_node **root,
                 int *ambiguous_p)'
```

is major parser function. It parses input according the grammar. The function returns the error code (which can be also returned by 'yaep_error_code'). If the code is zero, the function will also return root of the parse tree through parameter 'root'. The tree representing the translation. Value passed through 'root' will be 'NULL' only if syntax error was occurred and error recovery was switched off. The function sets up flag passed by parameter 'ambiguous_p' if we found that the grammar is ambiguous (it works even we asked only one parse tree without alternatives).

Function 'read_token' provides input tokens. It returns code the next input token and its attribute. If the function returns negative value we've read all tokens.

Function 'syntax_error' called when synatctic error has been found. It may print an error message about syntax error which occurred on token with number 'err_tok_num' and attribute 'err_tok_attr'. The following four parameters describes made error recovery which ignored tokens starting with token given by 3rd and 4th parameters. The first token which was not ignored is described by the last parameters. If the number of ignored tokens is zero, the all parameters describes the same token. If the error recovery is switched off (see comments for 'yaep_set_error_recovery_flag'), the third and the fifth parameters will be negative and the forth and the sixth parameters will be 'NULL'.

Function 'parse_alloc' is used by YAEP to allocate memory for parse tree representation (translation). After calling 'yaep_free_grammar' we free all memory allocated for the parser. At this point it is convenient to free all memory but parse tree. Therefore we require the following function. So the caller will be responsible to allocate and free memory for parse tree representation (translation). But the caller should not free the memory until 'yaep_free_grammar' is called for the parser. The function may be called even during reading the grammar not only during the parsing. Function 'parse_free' is used by the parser to free memory allocated by 'parse_alloc'. If it is 'NULL', the memory is not freed.

Function 'yaep_free_grammar'

```
'void yaep_free_grammar (struct grammar *grammar)'
```

frees all memory allocated for the parser. This function should be called the last for given parser.