MSTA (syntax description translator)

Vladimir Makarov, vmakarov@gcc.gnu.org May 5, 1999

This document describes MSTA (translator of syntax description of a language into code for parsing programs on the language).

Contents

1 Introduction

MSTA is syntax description translator analogous to YACC. Although MSTA can fully emulate YACC, MSTA is aimed to solve some drawbacks of YACC. Therefore MSTA has the following additional features:

- Fast LR(k) and LALR(k) grammars (with possibility resolution of conflicts). Look ahead of only necessary depth (not necessary given k). Originally LALR(k) parsers are generated by modified fast DeRemer's algorithm. Parsers generated by MSTA are up to 50% faster than ones generated by BISON and BYACC but usually have bigger size.

- Extended Backus-Naur Form (EBNF), and constructions for more convenient description of the scanners. More convenient naming attributes.
- Optimizations (extracting LALR- and regular parts of grammars and implementing parsing them by adequate methods) which permit to use MSTA for generation of effective lexical analyzers. As consequence MSTA permits to describe easily (by CFG) scanners which can not be described by regular expressions (i.e. nested comments).
- More safe error recovery and reporting (the 1st additional error recovery method besides error recovery method of YACC).
- A minimal error recovery and reporting (the 2nd additional error recovery method besides error recovery method of YACC).
- Fast generation of fast parsers.

2 MSTA description language

MSTA description languages is superset of YACC language. The major additional features are Extended Backus Naur Form (EBNF) for more convenient descriptions of languages, additional constructions in rules for more convenient description of scanners, and named attributes.

2.1 Layout of MSTA description

MSTA description structure has the following layout which is similar to one of YACC file.

```
DECLARATIONS
%%
RULES
%%
ADDITIONAL C/C++ CODE
```

The '%%' serves to separate the sections of description. All sections are optional. The first '%%' starts section of keywords and is obligatory even if the section is empty, the second '%%' may be absent if section of additional C/C++ code is absent too.

Full YACC syntax of MSTA description file is placed in Appendix 1.

2.2 Declarations

The section of declarations may contain the following construction:

```
%start  identifier
```

which determines axiom of the grammar. If such construction is absent, the axiom is believed to be nonterminal in the left hand side of the first rule. If there are several such construction, all ones except for the first are ignored.

By default, the values of attributes of the terminals (tokens) and nonterminals shall be integers. If you are going to use the values of different types, you shall use

```
<tag>
```

in constructs declaring symbols (%token, %type, %left, ...) and shall insert corresponding union member names in the following construction:

```
%union { body of union in C/C++ }
```

Alternatively, the union can be declared in interface file, and a typedef used to define the symbol YYSTYPE (see generated code) to represent this union. The effect of %union is to provide the declaration of YYSTYPE directly from the input.

There is group of the following declarators which take token (terminal) or nonterminal names as arguments.

```
%token [<tag>] name [number] [name [number]]...
%left [<tag>] name [number] [name [number]]...
%right [<tag>] name [number] [name [number]]...
%nonassoc [<tag>] name [number] [name [number]]...
%type <tag> name...
```

The names can optionally be preceded by the name of a C/C++ union member (called a tag see above) appearing within "<" and ">". The use of tag specifies that the tokens or nonterminals named in this construction are to be of the same C/C++ type as the union member referenced by the tag.

If symbol used in grammar is undefined by a %token, %left, %right, or %nonassoc declaration, the symbol will be considered as a nonterminal.

The first occurrence of a given token can be followed by a positive integer in constructions '%token', '%left', '%right', and '%nonassoc' defining tokens. In this case the value assigned to it shall be code of the corresponding token returned by scanner.

Constructions '%left', '%right', and '%nonassoc' assign precedence and to the corresponding tokens. All tokens in the same construction have the same precedence level and associativity; the constructions is suggested to be placed in order of increasing precedence. Construction '%left' denotes that the operators (tokens) in that construction are left associative, and construction '%right' similarly denotes right associative operators.

Construction '%nonassoc' means that tokens cannot be used associatively. If the parser encounters associative use of this token it will report an error.

The construction '%type' means that the attributes of the corresponding nonterminals are of type given in the tag field.

Once the type, precedence, or token number of a symbol is specified, it shall not be changed. If the first declaration of a token does not assign a token number, MSTA will assign a token number. Once this assignment is made, the token number shall not be changed by explicit assignment.

Usually real grammars can not be declared without shift/reduce conflicts. To control suggested number of shift/reduce conflicts, the following construction can be used.

```
%expect number
```

If such construction is present, MSTA will report error if the number of shift/reduce conflicts is not the same as one in the construction. Remember that it is not standard YACC construction.

The following construction in declarations means that the scanner should be generated.

```
%scanner
```

There are the following major differences in parser and scanner generated by MSTA

- In order to use a MSTA generated parser with a MSTA generated scanner, all objects in a MSTA scanner (variables, types, macro, and so on) are named by adding letter 's' or 'S' after prefixes 'yy' or 'YY'.
- Additional function 'yylex_start' is generated. The function should be used for initiation of scanner (see Generated code).
- Function 'yylex' is generated instead of function 'yyparse'. This function can be called many times for getting next token. Code of the next token is suggested to returned by statements 'return' in the actions. Input stream (look ahead characters) is saved from a call of 'yylex' to the next its call.
- Instead of function 'yylex' a function 'yyslex' is used to read the next character (token in terminology of MSTA specification file) from the input stream. -1 is used as the end of file instead of 0 because scanner must read and process zero characters.
- Macro 'YYSABORT' is -1 in order to differ token code from flag of finishing work of the scanner. Remember that analogous macro 'YYABORT' for MSTA parser is 1.
- To extract all regular parts in the scanner grammar, splitting LR-sets is fulfilled (see MSTA implementation).

You can look at a scanner specification in Appendix 2.

There may be also the following constructions in the declaration section

```
%{
   C/C++ DECLARATIONS
%}

%local {
   C/C++ DECLARATIONS
}

%import {
   C/C++ DECLARATION
}

and

%export {
   C/C++ DECLARATION
}
```

which contain any C/C++ declarations (types, variables, macros, and so on) used in sections. Remember the only first construction is standard POSIX YACC construction.

The local C/C++ declarations are inserted at the begin of generated implementation file (see section 'generated code') but after include-directive of interface file (if present – see MSTA Usage). You also can use more traditional construction of YACC %{ ... %} instead.

C/C++ declarations which start with '%import' are inserted at the begin of generated interface file. If the interface file is not generated, the code is inserted at the begin of the part of implementation file which would correspond the interface file.

C/C++ declarations which start with '%export' are inserted at the end of generated interface file. For example, such exported C/C++ code may contain definitions of external variables and functions which refer to definitions generated by MSTA. If the interface file is not generated, the code is inserted at the end of the part of implementation file which would correspond the interface file.

All C/C++ declarations are placed in the same order as in the section of declarations.

2.3 Rules

The section of declarations is followed by section of rules.

The rules section defines the context-free grammar to be accepted by the function yacc generates, and associates with those rules C language actions and additional precedence information. The grammar is described below, and a formal definition follows.

The rules section contains one or more grammar rules. A grammar rule has the following form:

```
nonterminal : pattern ;
```

The nonterminal in the left side hand of the rule describes a language construction and pattern into which the nonterminal is derivated. The semicolon at the end of the rule can be absent.

MSTA can use EBNF (Extended Backus-Naur Form) to describe the patterns. Because the pattern can be quite complex, MSTA internally transforms rules in the description into simple rules and assigns a unique number to each simple rule. Simple rule can contains only sequence of nonterminals and tokens. Simple rules and the numbers assigned to the rules appear in the description file (see MSTA usage). To achieve to the simple rules, MSTA makes the following transformations (in the same order).

1. Alternatives

   ```
   nonterminal : pattern1 | pattern2
   ```

   are transformed into

   ```
   nonterminal : pattern1
   nonterminal : pattern2
   ```

2. Lists

   ```
   nonterminal : ... pattern / s_pattern ...
   ```

   are transformed into

   ```
   nonterminal : ... N ...
   N : N s_patter pattern
   ```

   N denotes here a new nonterminal created during the transformation. This construction is very convenient for description of lists with separators, e.g. identifier separated by commas. Remember that the lists are not feature of standard POSIX YACC.

3. Naming

```
nonterminal : ... N @ identifier  ...
```

   is transformed into

```
nonterminal : ... N ...
```

   Here N denotes a nonterminal, a token, or the following constructions. Instead of number in actions, the identifier can be used for naming attributes of the nonterminal, the token, or nonterminal which is created during transformation of the following constructions. Remember that the naming is not feature of standard POSIX YACC.

4. Optional construction

```
nonterminal : ...  [ pattern ]  ...
```

   is transformed into

```
nonterminal : ... N ...
N : pattern
N :
```

   N denotes here a new nonterminal created during the transformation. This construction is very convenient for description of optional constructions. Remember that the optional construction is not feature of standard POSIX YACC.

5. Optional repetition

```
nonterminal : ... pattern * ...
```

   is transformed into

```
nonterminal : ... N ...
N : N pattern
N :
```

   N denotes here a new nonterminal created during the transformation. This construction is very convenient for description of zero or more the patterns. Remember that the optional repetition is not feature of standard POSIX YACC.

6. Repetition

```
nonterminal : ... pattern + ...
```

   is transformed into

```
nonterminal : ... N ...
N : N pattern
N : pattern
```

   N denotes here a new nonterminal created during the transformation. This construction is very convenient for description of one or more the patterns. Remember that the repetition is not feature of standard POSIX YACC.

7. Grouping

```
nonterminal : ... ( pattern ) ...
```

is transformed into

```
nonterminal : ... N ...
N : pattern
```

N denotes here a new nonterminal created during the transformation. This construction is necessary to change priority of the transformations. Remember that the grouping is not feature of standard POSIX YACC.

8. String

```
nonterminal : ... string ...
```

is transformed into

```
nonterminal : ... '1st char' '2nd char' ... 'last char' ...
```

Here the string is simply sequence of string characters as MSTA literals. Remember that the strings are not standard feature of POSIX YACC.

9. Range

```
nonterminal : ... token1  -  tokenN ...
```

is transformed into

```
nonterminal : N
N : token1
N : token2
...
N : tokenN
```

N denotes here a new nonterminal created during the transformation. The range is simply any token with code between code of token1 and code of token2 (inclusively). The code of token1 must be less or equal to the code of token2. Remember that the ranges are not feature of standard POSIX YACC.

10. Left open range

```
nonterminal : ... token1  <-  tokenN ...
```

is transformed into

```
nonterminal : N
N : token2
N : token3
...
N : tokenN
```

N denotes here a new nonterminal created during the transformation. The left open range is simply any token with code between code of token1 + 1 and code of token2 (inclusively). The code of token1 must be less to the code of token2. Remember that the ranges are not feature of standard POSIX YACC.

11. Right open range

```
nonterminal : ... token1  ->  tokenN ...
```

is transformed into

```
nonterminal : N
N : token1
N : token2
...
N : tokenN-1
```

N denotes here a new nonterminal created during the transformation. The right open range is simply any token with code between code of token1 and code of token2 - 1 (inclusively). The code of token1 must be less to the code of token2. Remember that the ranges are not feature of standard POSIX YACC.

12. Left right open range

```
nonterminal : ... token1  <->  tokenN ...
```

is transformed into

```
nonterminal : N
N : token2
N : token3
...
N : tokenN-1
```

N denotes here a new nonterminal created during the transformation. The left right open range is simply any token with code between code of token1 + 1 and code of token2 - 1 (inclusively). The code of token1 must be less to the code of token2 - 1. Remember that the ranges are not feature of standard POSIX YACC.

13. Action inside pattern

```
nonterminal : ... action  something non empty
```

is transformed into

```
nonterminal : ... N  something non empty
N : action
```

N denotes here a new nonterminal created during the transformation. The action is a C/C++ block.

After the all possible transformations mentioned above, the rules will contain sequence of only tokens (literals or token identifiers) and nonterminals finishing optional %prec or/and %la construction or/and an action.

The action is an arbitrary C/C++ block, i.e. declarations and statements enclosed in curly braces { and }. Certain pseudo-variables can be used in the action for attribute references. These are changed by data structures known internally to MSTA. The pseudo-variables have the following forms:

$$

This pseudo-variable denotes the nonterminal in the left hand side of the simple rule.

$number

This pseudo-variable refers to the attribute of sequence element (nonterminal, token, or action) specified by its number in the right side of the rule before changing actions inside pattern (see transformation above), reading from left to right. The number can be zero or negative. If it is, it refers to the attribute of the symbol (token or nonterminal) on the parser's stack preceding the leftmost symbol of the rule. (That is, $0 refers to the attribute of the symbol immediately preceding the leftmost symbol in the rule, to be found on the parser's stack, and $-1 refers to the symbol to its left.) If number refers to an element past the current point in the rule (i.e. past the action), or beyond the bottom of the stack, the result is undefined.

$identifier

These pseudo-variable is analogous to the previous one but the attribute name is used instead of its number. Of course the attribute naming must exist.

$<...>number

This pseudo-variable is used when there are attributes of different types in the grammar and the number corresponds to the nonterminal whose type is not known because the nonterminal has been generated during the transformation of rules into the simple rules. The type name of the attribute is placed into angle braces.

$<...>identifier

These pseudo-variable is analogous to the previous one but the attribute name is used instead of its number. Of course the attribute naming must exist.

$<...>$

This pseudo-variable is used when there are attributes of different types in the grammar and the type of nonterminal is not known because the nonterminal has been generated during the transformation of rules into the simple rules.

Messages about some shift/reduce conflicts are not generated if the rules in the conflict has priority and associativity. The priority and associativity of rule are simply the precedence level and associativity of the last token in the rule with declared precedence level and associativity.

The optional construction '%prec ...' can be used to change the precedence level associated with a particular simple rule. Examples of this are in cases where a unary and binary operator have the same symbolic representation, but need to be given different precedences. The reserved keyword '%prec' can be followed by a token identifier or a literal. It shall cause the precedence of the grammar rule to become that of the following token identifier or literal.

The optional construction '%la number' can be used to change the maximal look ahead associated with a particular simple rule. Example of this is when there is a classical conflict if-then-else which is to be resolved correctly with look ahead equal to 1 and there is a rule with conflict which must be resolved with look ahead equal to 3. In this case you can call MSTA with maximal look ahead equal to 1 (this is default) and place %la 3 in the rule which takes part in the conflict which must be resolved with look ahead equal to 3.

If a program section follows, the grammar rules shall be terminated by %%.

3 Generated code

A specification as described in the previous section is translated by MSTA into optional interface and implementation files having the same names as one of specification file and correspondingly suffixes '.h' and '.c' (C code) or '.cpp' (C++ code). By default the interface file is not generated.

3.1 C code

The interface and implementation files consist of the following definitions of generated macros, types, and functions (unless special information for MSTA scanner is mentioned, MSTA scanner object have the same sense and names with additional s or S after the prefix 'yy' or 'YY'):

YYSTYPE

By default this is macro. The macro value is type used for representing the parser attributes. By default this macro is defined as 'int'. You can redefine the macro if you place definition of the macro before standard definition of the macro.

If construction '%union' is present in the specification file, YYSTYPE is type definition of union with the code written inside construction '%union'.

The definition of YYSTYPE is placed in the interface file if option '-d' is on MSTA command line. Otherwise the definition will be in the implementation file. YYSTYPE is a part of YACC POSIX standard.

yychar

This variable contains code of the current token. The current token is not latest read token because MSTA can look ahead far. The codes are returned by scanner function 'yylex'. If option '-d' is present on the command line (see MSTA usage), external definition of the variable is also placed in the interface file. The variable is a part of YACC POSIX standard.

yylval

This variable is used to exchange information of the parser with a scanner. The scanner must return attribute of the latest read token in this variable. After that the variable contains attribute of the current token, i.e. whose code is in the variable 'yychar'. The variable 'yylval' is declared of type YYSTYPE. If option '-d' is present on the command line (see MSTA usage), external definition of the variable is also placed in the interface file. The variable is a part of YACC POSIX standard.

YYDEBUG

The parser generated by MSTA has code for diagnostics. The compilation of the runtime debugging code is under the control of YYDEBUG, a preprocessor symbol. If YYDEBUG has a nonzero value, the debugging code will be included. If its value is zero, the code will not be included. The macro is a part of YACC POSIX standard.

yydebug

In parser where the debugging code has been included (see macro YYDEBUG), the variable 'yydebug' can be used to turn debugging on (with a nonzero value) and off (zero value) at run time. The initial value of yydebug is zero. If option '-d' is present on the command line (see MSTA usage), external definition of the variable is also placed in the interface file. The variable is a part of YACC POSIX standard.

int yyparse ()

This function is main function of MSTA parser. The function makes parsing of the token sequence whose codes are returned by user-defined function 'yylex' and whose attributes if any are placed in variable 'yylval'. The function returns 0 if the parser successfully finished work. Nonzero returned status means that the parser found unrecoverable errors (or macro YYABORT was executed explicitly). This function is a part of YACC POSIX standard.

This function has name 'yylex' for MSTA scanner. The function makes scanning of the character (token in terminology of MSTA specification file) sequence whose codes are returned by function 'yyslex' and whose attributes if any are placed in variable 'yyslval'. The function returns 0 if the parser successfully finished work and reach end of input file stream. Negative returned status means that the parser found unrecoverable errors (or macro YYSABORT was executed explicitly). This function can be called many times for getting next token. Code of the next token is suggested to returned by statements 'return' in the actions. Input stream (look ahead characters) is saved from a call of 'yylex' to the next its call.

int yylex ()

This function is an external function to the MSTA parser. User must provide it. Each call of the function should return code of the next input token. If end of input is reached, the function should return zero (-1 for 'yyslex). Attribute of token whose code returned by the function should be returned by the function through variable 'yylval'. In the case of MSTA scanner, function 'yyparse' has name 'yylex'.

void yylex_start (int *error_code)

The function 'yylex_start' is generated only for MSTA scanner. The function should be used for initiation of the scanner. Nonzero value returned through the parameter means that ther was error in memory allocation for the scanner (this is a fatal error). The function is not a part of YACC POSIX standard.

yyprev_char

Its value is the latest shifted token (character) code. Usually the value is used for forming internal representation of tokens (e.g. identifier internal representation or number value). The variable is not a part of YACC POSIX standard.

YYACCEPT

The macro YYACCEPT will cause the parser to return with the value zero. This means normal parser work finish. The macro is a part of YACC POSIX standard.

YYABORT

The macro YYABORT will cause the parser to return with a nonzero value (1 for MSTA parser and -1 for macro YYSABORT MSTA scanner). This means abnormal parser work finish. The macro is a part of YACC POSIX standard.

yyerror

When the parser detects a syntax error in its normal state, it normally calls external function yyerror with string argument whose value is defined by macro YYERROR_MESSAGE. User must provide function 'yyerror' for building parser program. After that the parser jumps to recovery mode. The parser is considered to be recovering from a previous error until the parser has shifted over at least

YYERR_RECOVERY_MATCHES normal input tokens since the last error was detected or a semantic action has executed the macro 'yyerrok'. The function is a part of YACC POSIX standard.

Recovery mode consists of on or more steps. Each recovery step starts with searching for the uppest stack state on which the shift on special symbol 'error' is possible. This state becomes the top stack state, and shift on 'error' is made. After that the parser discards all tokens which can not be after the symbol 'error' in this state (so called stop symbols). After that any recognized syntatic error results in the new error recovery step. This is technique of standard YACC error recovery. Such technique may result in infinite looping of the parser or discarding all input tokens if the stop symbols are not met.

By default MSTA generates the standard YACC error recovery. There are two additional methods which msta can generate for error recovery.

The first one is a local error recovery which does not permit infinite parser looping and use context after several error as stop symbols. According this method look ahead set also includes look ahead tokens after token 'error' in states which have the 'error' token is acceptable and which are lower in the parser stack than the first state with acceptable token 'error'. In this case the feedback from the parser to the scanner could not work correctly because although rule actions are executed in such case the parser reads the tokens once.

The second one is a minimal cost error recovery where the cost is overall number of tokens ignored. The feedback from the parser to the scanner does not work correctly. So you shouldn't use this method when there is the feedback. Calling 'yyerrok' has no sense for such method because the parser in such recovery mode never executes the rule actions. This method is the best quality error recovery although it my be expensive method because in the worst case it might save all input tokens.

YYERROR_MESSAGE

The macro value is used as a parameter of function yyerror when a syntax error occurs. The default value of macro is "syntax error" ("lexical error" for a scanner). You can redefine its value. But in any case the value should be a string. The macro is not a part of YACC POSIX standard.

YYERR_RECOVERY_MATCHES

The parser is considered to be recovering from a previous error until the parser has shifted over at least YYSERR_RECOVERY_MATCHES normal input tokens since the last error was detected or a semantic action has executed the macro 'yyerrok'. The default value of macro is 3. You can redefine its value. But in any case the value will be positive. This macro is not a part of YACC POSIX standard.

YYERR_MAX_LOOK_AHEAD_CHARS

This macro is generated only when the local recovery mode is used. The default value is 7. This value can not be less 1. See description below.

YYERR_LOOK_AHEAD_INCREMENT

This macro is generated only when the local error recovery mode is used. The default value is 3. This value can not be less 0. See description below.

YYERR_POPPED_ERROR_STATES

This macro is generated only when the local error recovery mode is used. The default value is 2. This value can not be less 0. See description below.

YYERR_DISCARDED_CHARS

This macro is generated only when the local error recovery mode is used. The default value is 3. This value can not be less 0. See description below.

yydeeper_error_try

This and the previous macros (YYERR_MAX_LOOK_AHEAD_CHARS - YYERR_DISCARDED_CHARS) are used only when the local error recovery is generated. Before starting description of the local error recovery, let me remind how YACC error recovery works. When the parser recognizes a syntactic error, it switches into error recovery mode. Error recovery itself consists of one or more steps. Each step consists of finding the top state on the stack with possible shift on pseudo-token 'error', throwing all states upper the state with 'error', and making shift on the 'error' token. After that all token are discarded until token (so called stop symbol) which can be after the pseudo-token 'error' is read. After that any recognized error results in the local error recovery step. And finally the error recovery is switched off only when YYERR_RECOVERY_MATCHES (by default 3) tokens are shifted without occurring syntactic error.

The differences of the local error recovery from classic YACC error recovery is in the following:

- The parser saves all discarded tokens in error recovery mode and returns them back into the input stream on the local error recovery step.
- Only YYERR_LOOK_AHEAD_INCREMENT tokens can be discarded on the first step, 2 * YYERR_LOOK_AHEAD_INCREMENT on the second step and so on (but no more YYERR_MAX_LOOK_AHEAD_CHARS tokens).
- If the parser requires discarding more tokens which is possible on the step, the local error recovery step starts. Moreover if action 'yydeeper_error_try' has been fulfilled on the previous step, the new step starts searching for the error state on the stack with the state which is deeper than the error state on the previous error recovery step. Otherwise, as usually searching for the error state starts with the top of the stack.
- On each YYERR_POPPED_ERROR_STATES error recovery step (and correspondingly on each YYERR_POPPED_ERROR_STATES processing the error state), the parser discards YYERR_DISCARDED_CHARS tokens without saving them before searching for the stop symbols.

By default MSTA generates YACC error recovery which does not permit infinite parser looping and use context after several error as stop symbols. The following fragment illustrates usage of the local error recovery mode.

```
#define YYERR_END_RECOVERY() yyerr_end_recovery()
...
program :
        | program function
...
function : ...
         | error END FUNCTION
             {yyerror ("error in function");}
...
statement : ...
          | error
              {
                yyerror ("error in statement");
                ...
              }
...
expression : ...
```

```
            | error
                {
                  yyerror ("error in expression");
                  ...
                }
...
yyerror (char *s)
{
  /* save string s */
}
yyerr_end_recovery ()
{
  /* print last saved error message. */
}
```

Note that action for error rule for function does not use macro 'yydeeper_error_try', this is warranty that the all program will be processed.

YYRECOVERING()

The macro YYRECOVERING serves to determine in which state the parser works now. The macro returns 1 if a syntax error has been detected and the parser has not yet fully recovered from it. Otherwise, zero is returned. The macro is a part of YACC POSIX standard.

YYERROR

The parser detects a syntax error when it is in a state where the action associated with the lookahead symbol(s) is error. A semantic action can cause the parser to initiate error handling by executing the macro YYERROR. When YYERROR is executed, the semantic action passes control back to the parser. YYERROR can be placed only in the semantic action itself (not in a function called from the semantic action). The single difference between error detected in the parser input and error caused by macro YYERROR is that the function 'yyerror' is not called in the second case. The macro is a part of YACC POSIX standard.

yynerrs

Actually this variable contains the number of switching the parser state from normal to error recovery. This switching is performed by fixing error in the input or by executing macro YYERROR. The macro is not a part of YACC POSIX standard. In the case of MSTA scanner, the variable accumulates the number for all calls of 'yylex'.

yyerrok

This macro can be used only in a semantic action itself. The macro causes the parser to act as if it has fully recovered from any previous errors. The macro is a part of YACC POSIX standard. The macro has no sense for minimal error recovery method because the parser in such recovery mode never executes the rule actions.

YYERR_END_RECOVERY()

This macro is called when the parser switches from the recovery state into normal state. By default the macro does nothing. You can redefine this macro, e.g. to output the last error buffered by your 'yyerror' function in order to implement better error diagnostics of the parser in the local recovery mode. The macro is not a part of YACC POSIX standard and the macro is not generated when yacc error recovery is used.

YYERRCODE

The token error is reserved for error handling. The name error can be used in grammar rules. It indicates places where the parser can recover from a syntax error. The default value of error shall be 256. Its value can be changed using a %token declaration. In any case the code of token error is value of macro YYERRCODE.

yyclearin

This macro cause the parser to discard the current lookahead token. If the current lookahead token has not yet been read, yyclearin has no effect. The macro is a part of YACC POSIX standard.

YYALLOC, YYREALLOC, YYFREE

MSTA uses memory allocation for the state and attribute stacks. Moreover, stacks can be expandable (with the aid of YYREALLOC). The macro values are used for the stack memory allocation/reallocation/freeing. Default value of the macros are standard C functions malloc, realloc, free. You can redefine this value. The macros are not a part of YACC POSIX standard.

YYSTACK_SIZE

The macro value is initial size of state and attribute stacks of the parser. If a stack become overfull, macro YYABORT is executed when option -no-expand is used. Otherwise, the stacks are expanded. It is better to use left recursion in grammar rules in order to do not make overfull stacks. Default value of the macro is 500. You can redefine this value. The macro is not a part of YACC POSIX standard.

YYMAX_STACK_SIZE

The macro value is maximal size of state and attribute stacks of the parser. The macro is used when the stacks are expandable. If a stack size become bigger (may be after several stacks expansions), macro YYABORT is executed. Otherwise, the stacks are expanded. Default value of the macro is 5000. You can redefine this value. The macro is not a part of YACC POSIX standard.

YYMAX_STACK_EXPAND_SIZE

The macro value is step of state and attribute stacks expansion. The macro is used only when the stacks are expandable. Default value of the macro is 100. You can redefine this value. The macro is not a part of YACC POSIX standard.

YYMSTA

This macro defined as 1 is generated in order to differ the parser generated by YACC, BISON, or MSTA. Naturally the macro is not a part of YACC POSIX standard.

YYTOKEN_NAME(code)

This macro returns printable representation of token with given code. The macro is not a part of YACC POSIX standard.

YYLAST_TOKEN_CODE

This macro value is maximal code of tokens. the macro is not a part of YACC POSIX standard.

3.2 C++ code

The major advantage of C++ code is that it is quite easy to create many parsers of one language (and consequently reenterable parser). This is useful for implementation of module languages and languages with macro directives of type of C include directive.

Generated C++ code is different from C code in the following features:

- Abstract class 'yyparser' is generated for a parser and 'yyscanner' for a scanner. The definition of class will be present also in interface file if the interface file is generated (see MSTA usage).
- Variables 'yylval' ('yyslval'), 'yychar' ('yyschar'), and 'yydebug' ('yysdebug') are now public members of the class.
- Functions 'yylex' ('yyslex'), 'yyerror' ('yyserror') are now abstract public virtual functions of the class.
- Functions 'yyparse' for a parser and 'yylex' for a scanner are now public functions of the class.
- Function 'yylex_start' for a scanner is not generated because constructor of the class

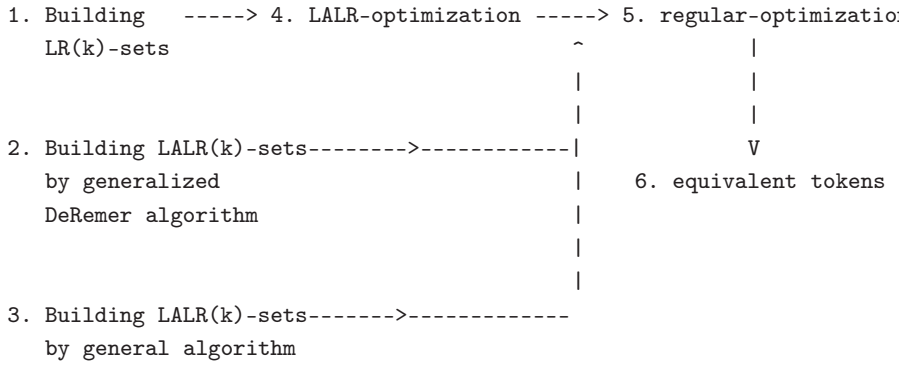
  ```
  yyscanner (int &)
  ```

  replaces the function.
- The class contains also virtual destructor.

Usually the parser (scanner) itself is implemented as sub-class of class 'yyparser' ('yyscanner'). This subclass contains definition of functions 'yylex' ('yyslex') and 'yyerror' ('yyserror').

4 Implementation

The following figure shows what major algorithms MSTA can uses to generate the parsers:



MSTA can generate LR(k)-parsers (see MSTA usage). After building canonical LR(k)-sets, MSTA usually makes LALR-optimization which significantly decreases size of the result parser. You can switch off LALR-optimization, but my advice is to use it. This optimization results in not only less size of the parser, but significantly speeds up MSTA work and decreases memory requirements (because before and during LALR-optimization, only LR-sets are represented only by their essential LR-situations).

LALR-optimization is to search for equivalent LR-sets with the same LR-core set (LR-set with LR-situations without contexts) and to merge them. In other words LALR-optimization is an extracting LALR-parts of grammars and implementing parsing them by adequate methods. If the input grammar describes LALR-language, the result LR-sets will be LALR-sets. The optimization algorithm is analogous to searching for minimal DFA (deterministic final automaton). There is my article describing the optimization in one russian journal (Programming, 1989), unfortunately only on russian. Now I have no time and place to describe it more detail here.

Usually MSTA generates LALR(k)-parsers with a generalized DeRemer algorithm. But when you want to see contexts of LR-situations in the description file (see MSTA usage), MSTA will use canonical algorithm of building LALR-sets (see for example old book of Aho and Ulman). Remember that this algorithm is slower than DeRemer ones in several times.

When k > 1, the length look ahead of the parser can be less than k. The parser generated by MSTA always look ahead only on minimal number of tokens necessary for correct resolution of the conflicts in given state and given input.

After building LR-sets, MSTA usually runs pass of so called regular-optimization. Unfortunately this algorithm is described by me nowhere. The idea of algorithm is to searching for all transitions of parser from the one state to another which are independent from the parser state (more correctly from the parser stack). In other words regular-optimization is an extracting regular-parts of grammars and implementing parsing them by adequate methods. As a result the generated parser will be faster and will use less the parser stack.

If the input grammar describes regular language, the result parser will not use stack at all. This permits to use MSTA for generation of effective scanner too. Moreover, scanner for a language with non-regular parts (e.g. nested comments) is described much more simply on MSTA and is effectively implemented by MSTA. To extract more regular parts a splitting LR-sets can be used (this is used for '%scanner' by default). Usage of splitting LR-sets is not recommended for usual programming languages grammars because this requires very much memory during optimizations.

Implementation of regular-optimization requires more number of classical LR-parser instructions (not only shift-reduce-error). This means that MSTA parser implementation is more oriented to compilation model than classical YACC or BISON. This also speeds up the parser generated by MSTA. The new instructions "(pop)-(shift)-goto" (optional parts are in parentheses here) are added. Moreover, more one actions for different rules can be executed during one of such instructions. Also, each parser state has status of necessity of pushing the result state and/or corresponding attribute into state and attribute stacks. MSTA parser generated with usage of the regular optimization may have several copies of rule actions, but usually this only slightly increases size of the parser.

MSTA also searches for equivalent tokens to decrease the generated parser size. This optimization is especially effective for scanners.

5 MSTA Usage

```
MSTA(1)                          User Manuals                          MSTA(1)



NAME
       msta - syntax description translator
```

SYNOPSIS

msta [-d -define -l -line -t -trace -v -verbose -b file_prefix -p sym_prefix -h -help --help -w -c++ -enum -no-enum -pattern-equiv -no-pattern-equiv -look-ahead number -la number -lr -no-lr -lalr-optimizaâĂŘtion -no-lalr-optimization -regular-optimization -no-regular-optimizaâĂŘtion -split-lr-sets -no-split-lr-sets -yacc-error-recovery -local-error-recovery -minimal-error-recovery -error-reduce -no-error-reduce -error-conflict -no-error-conflict -yacc-input -no-yacc-input -strict -no-strict -yacc-file-names -no-yacc-file-names -o name -y -yacc -full-lr-set -lr-situation-context -removed-lr-sets -expand -no-expand -time] specification-file

DESCRIPTION

MSTA reads input grammar which is described in specification file and generates parser for it. MSTA can process any POSIX YACC grammar and generates equivalent parser in the sence of POSIX standard. But MSTA can do many additional things: generation of more safe error recovery or much better error recovery, better output of generated parser description, generation of fast LALR(k)- and LR(k)-grammar parser with acceptable space requirements, generation of scanners described as conâĂŘtext-free grammar (CFG), and so on. The specification file must have suffix '.y'

The generated code consists of optional interface and implementation files and optional description file having the same names as one of specification file and correspondingly suffixes '.h' (if option -d or -c++ is given) and '.c' (C code) or '.cpp' (C++ code) and '.output' (description file). If MSTA runs in YACC regime, the corresponding files will be 'y.tab.h' (if option -d or -c++ is given) and 'y.tab.c' (C code) or 'y.tab.cpp' (C++ code) and 'y.output' (description file).

Full documentation of MSTA is in MSTA User's manual.

OPTIONS

The options which are known for MSTA and YACC are:

-d Create the interface file. By default the interface file is not generated.

-l Insert user's original code into generated parser without '#line' constructs. By default constructions '#line' are always generated.

-t Permit compilation of debugging code in the code file (tracing of the parser work). Runtime debugging statements are always contained in the parser, but by default conditional compilation directives prevent their compilation. This option defines macro 'YYDEBUG' as 1 (by default its value is 0) and as consequence the debugging statements are processed.

```
-v      Create a file containing description of the parser.

-b file_prefix
        Use file_prefix instead of ‘y’ as  the  prefix  for  all  output
        filenames when YACC regime of generated file names is on.

-p sym_prefix
        Use  sym_prefix  instead  of  yy  as the prefix for all external
        names (i.e.  functions yyparse , yylex , and yyerror ,  and  the
        variables yylval , yychar , and yydebug ) produced by MSTA.

Additional options which are known only for MSTA are:

-define
        This is synonym of option ‘-d’

-line   This is synonym of option ‘-l’

-trace This is synonym of option ‘-t’

-verbose
        This is synonym of option ‘-v’

-h, -help, --help
        These  options  result in that only brief description of options
        is output into stderr.  The same result can be achieved by callâĂŘ
        ing MSTA without any options and arguments.

-w      No any warning messages are generated, e.g. repeated description
        of type, value, and priority of token or terminal, repeated conâĂŘ
        struction ‘%start’ and many others.

-c++    Output of C++ code instead of C code (which is default).

-enum   Generate token constants as enumeration instead of macro.

-no-enum
        Generate token constants as macro (this is default).

-pattern-equiv
        Equivalent  patterns  are  denoted  by  the same nonterminal (by
        default).  Sometimes this results in dramatic reducing conflicts
        when complex patterns are used.

-no-pattern-equiv
        Even  equivalent  patterns are denoted by different nonterminals
        and are expanded into the different set of rules

-look-ahead number, -la number
        Determine maximal length of look-ahead when  LR(k)  and  LALR(k)
        generated  parser.   This does not mean that MSTA in each situaâĂŘ
```

tion and for each input will look ahead on given number tokens. MSTA looks ahead only on minimal number of tokens required for resolution of a conflict. The default value of maximal look ahead is 1.

-lr Input grammar is considered to be a LR(k)-grammar.

-no-lr Input grammar is considered to be a LALR(k)-grammar. This is default.

-lalr-optimization
If the input grammar is considered to be a LR(k)-grammar, MSTA will make LALR(k)-optimization after building LR(k)-parser. It means that MSTA will extract LALR(k)-parts of LR(k)-grammar and will represent them by addequate LALR-sets. If the input LR(k)-grammar is "almost" a LALR(k)-grammar, MSTA can descrease size of the generated parser in several times. If the input grammar is a LALR(k)-grammar, MSTA will create LALR(k)-parser. This option is default when option '-lr' is given.

-no-lalr-optimization
When option '-lr' is given, MSTA does not make LALR-optimizaâĂŘ tion.

-regular-optimization
After building LALR-sets or LR-sets and optional LALR-optimizaâĂŘ tion, MSTA makes extracting regular parts of input grammar and implement them by adequate methods (deterministic finite automaâĂŘ ton). This decreases number of final states in the generated parser, speeds up the parser, but sigthly increases the parser size because some actions will be present in several examplars in the parser. The more regular parts in the input grammar, the better this method works. By default this option is on.

-no-regular-optimization
Disable the regular optimizations.

-split-lr-sets
Splitting LR-sets during regular optimization in order to extract the most of regular parts (only when option '-regular-optimization' is on). By default this option is on if '%free_format' is present in the specification file.

-no-split-lr-sets
Option is opposite to the previous one. By default this option is on if '%free_format' is absent in the specification file.

-yacc-error-recovery
The generated parser will use error recover according to Posix YACC standard. This error recovery is used by default.

-local-error-recovery
: The generated parser will use sligthly modified error recovery method. This method is more safe. It is guaranteed that this method can not be cycled.

-minimal-error-recovery
: The generated parser will use a minimal cost error recovery where the cost is overall number of tokens ignored. It is guarâĂŘ anteed that this method can not be cycled. This method is expensive because it may save many input tokens, parser states and attributes.

-error-reduce
: MSTA generates parser with possible default reducing (without checking the look ahead) when the token error is in the context. It means the parser may make a few reduces before recognizing a syntax error. This option is default.

-no-error-reduce
: The option means that the generated parser makes reduces in sitâĂŘ uation where shift on error is possible only when there is corâĂŘ responding look ahead. It results in better error recovery but the parser will have bigger size.

-error-conflict
: MSTA reports conflicts on look ahead containing token error in LR-set which is a result of shifting token error. This option is default.

-no-error-conflict
: Token error is not a real token it is never read but many error recovery rules may generate conflicts on error. To avoid this you can use the option. In this case MSTA does report conflicts on look ahead containing token error in LR-set which is a result of shifting token error.

-yacc-input
: Only Posix YACC input can be used as input grammar.

-no-yacc-input
: All input grammar features of MSTA can be used. This option is default.

-yacc-file-names
: MSTA output file names are generated according to Posix YACC (y.output , y.tab.c , y.tab.h).

-strict
: Use only strict POSIX YACC input (this option is in action only when -yacc-input) not SYSTEM V one. Errors are output when nonstrict features are used. These features are usage of ';'

after definitions and C/C++ code insertions and usage of a token and a literal in %type clauses as before and after a description of the token and the literal. This option is on by default.

-no-strict
This option is in action only when -yacc-input is on. Only warnings about non strict POSIX YACC features usage are output.

-no-yacc-file-names
Output files will have the same name and suffixes (.h , .c , and .output). This option is on by default.

-o Specify names of output files (header file name.h , code file name.c , description file name.output

-y, -yacc
Emulate Posix YACC. It means that the options '-yacc-input' , '-lalr' , '-yacc-error-recovery' , and '-yacc-file-names' are on, and -c++ is off. This option is useful with other options, e.g. options '-lr' and '-y' means also '-yacc-input' , '-yacc-error-recovery' , and '-yacc-file-names'.

-full-lr-set
Write all LR-situations (besides essential ones) of LR-sets into a file containing description of the parser (the option acts only with option '-v'). Sometimes it is useful for better understanding conflicts. But in this case the description file can be significantly bigger.

-lr-situation-context
Write context of LR-situations into description of the parser (the option acts only with option '-v'). Sometimes it is useâĂŘful for better understanding conflicts. But in this case the description file can be huge espicially for LR(k)- and LALR(k)-grammar when k > 1 or when with option '-full-lr-set' parser generation speed because classic method of LR-, LALR-sets is used instead of fast DeRemer method. By default this option is off.

-removed-lr-sets
Write LR-sets removed during conflict resolution and regular optimization into description of the parser (the option acts only with option -v). By default such unchievable LR-sets are not present in the description file.

-expand
Attributes and states stacks in the parser will be expandable (this is default).

-no-expand
Attributes and states stacks in the parser will be not expandâĂŘ

```
               able.

       -time  Output detail time statistics of MSTA work on its  basic  passes
              and optimizations.

FILES
       file.y
              MSTA specification file
       file.c or y.tab.c
              generated C implementation file
       file.cpp or y.tab.cpp
              generated C++ implementation file
       file.h or y.tab.h
              generated interface file
       file.output or y.output
              generated interface file
       There are no any temporary files used by MSTA.

ENVIRONMENT
       There are no environment variables which affect MSTA behavior.

DIAGNOSTICS
       MSTA diagnostics is self-explanatory.

AUTHOR
       Vladimir N. Makarov, vmakarov@gcc.gnu.org

SEE ALSO
       oka(1), sprut(1), nona(1), shilka(1).  SHILKA manual.

BUGS
       Please, report bugs to https://github.com/dino-lang/dino/issues.




COCOM                           5 SEP 1999                          MSTA(1)
```

6 Future of MSTA development

Frequently LR(k)-grammars are not sufficient to describe modern programming languages adequately. More power grammars would be useful for this. It could be LR-regular grammars, backtracking, or something else.

7 Appendix 1 - Syntax of MSTA description file (YACC description)

```
%token IDENTIFIER_OR_LITERAL /* identifier (including '.')
```

```
                                and literal */
%token C_IDENTIFIER          /* identifier followed by a : */
%token NUMBER                /* [0-9][0-9]* */
%token STRING                /* "..." of C syntax */
%token CODE_INSERTION        /* { ... } */
%token YACC_CODE_INSERTION   /* %{ ... %} */
%token ADDITIONAL_C_CODE     /* code after second %% */

/* Reserved words : %type=>TYPE %left=>LEFT, etc.  The attributes of
   the following tokens are not defined and not used. */

%token   LEFT RIGHT NONASSOC TOKEN PREC LA TYPE START
         UNION LOCAL IMPORT EXPORT SCANNER EXPECT
%token   PERCENTS             /* the %% mark */
%token   SEMICOLON            /* ; */
%token   BAR                  /* | */
%token   SLASH                /* / */
%token   STAR                 /* * */
%token   PLUS                 /* + */
%token   LESS                 /* < */
%token   GREATER              /* > */
%token   LEFT_PARENTHESIS     /* ) */
%token   RIGHT_PARENTHESIS    /* ( */
%token   LEFT_SQUARE_BRACKET  /* [ */
%token   RIGHT_SQUARE_BRACKET /* ] */
%token   AT                   /* @ */
%token   RANGE                /* - */
%token   RANGE_NO_LEFT_BOUND  /* <- */
%token   RANGE_NO_RIGHT_BOUND /* -> */
%token   RANGE_NO_LEFT_RIGHT_BOUNDS  /* <-> */

/*       8-bit character literals stand for themselves; */
/*       tokens have to be defined for multibyte characters */

%start   description

%%

description  : definitions PERCENTS rules tail
             ;

tail  : /* empty */
      | ADDITIONAL_C_CODE
      ;

definitions  : /* empty */
             | definitions  definition  definition_tail
             ;

definition_tail :
                | definition_semicolon_list
```

```
                ;

definition_semicolon_list : SEMICOLON
                          | definition_semicolon_list SEMICOLON
                          ;

definition   : START
               IDENTIFIER_OR_LITERAL
             | UNION   CODE_INSERTION
             | YACC_CODE_INSERTION
             | LOCAL  CODE_INSERTION
             | IMPORT CODE_INSERTION
             | EXPORT CODE_INSERTION
             | SCANNER
             | EXPECT NUMBER
             | symbol_list_start tag symbol_list
             ;

symbol_list_start : TOKEN
                  | LEFT
                  | RIGHT
                  | NONASSOC
                  | TYPE
                  ;

tag : /* empty */
    | LESS  IDENTIFIER_OR_LITERAL  GREATER
    ;

symbol_list : symbol
            | symbol_list  symbol
            ;

symbol  : IDENTIFIER_OR_LITERAL
        | IDENTIFIER_OR_LITERAL  NUMBER
        ;

/* Rule section */

rules : rule  semicolons
      | rules  rule  semicolons
      ;

rule : C_IDENTIFIER pattern
     ;

pattern : alternatives
        ;

alternatives : alternatives  BAR  alternative
             | alternative
```

```
            ;

alternative : sequence prec_la
            | sequence prec_la SLASH sequence prec_la
            ;

sequence : /* empty */
         | sequence  sequence_element
         ;

sequence_element : nonamed_sequence_element
                 | nonamed_sequence_element AT IDENTIFIER_OR_LITERAL
                 ;

nonamed_sequence_element
   : LEFT_SQUARE_BRACKET
     pattern  RIGHT_SQUARE_BRACKET
   | unit  STAR
   | unit  PLUS
   | CODE_INSERTION  code_insertion_tail
   | unit
   ;

code_insertion_tail
   :
   | code_insertion_semicolon_list
   ;

code_insertion_semicolon_list
  : SEMICOLON
  | code_insertion_semicolon_list SEMICOLON
  ;

unit : LEFT_PARENTHESIS
       pattern  RIGHT_PARENTHESIS
     | IDENTIFIER_OR_LITERAL
     | STRING
     | IDENTIFIER_OR_LITERAL  RANGE  IDENTIFIER_OR_LITERAL
     | IDENTIFIER_OR_LITERAL  RANGE_NO_LEFT_BOUND  IDENTIFIER_OR_LITERAL
     | IDENTIFIER_OR_LITERAL  RANGE_NO_RIGHT_BOUND IDENTIFIER_OR_LITERAL
     | IDENTIFIER_OR_LITERAL  RANGE_NO_LEFT_RIGHT_BOUNDS
       IDENTIFIER_OR_LITERAL
     ;

prec_la : /* empty */
        | PREC  IDENTIFIER_OR_LITERAL
        | PREC  IDENTIFIER_OR_LITERAL  CODE_INSERTION  code_insertion_tail
        | LA NUMBER
        | PREC  IDENTIFIER_OR_LITERAL   LA NUMBER
        | LA NUMBER   PREC  IDENTIFIER_OR_LITERAL
        | LA NUMBER  CODE_INSERTION code_insertion_tail
```

```
        | PREC IDENTIFIER_OR_LITERAL  LA NUMBER
              CODE_INSERTION code_insertion_tail
        | LA NUMBER  PREC IDENTIFIER_OR_LITERAL
              CODE_INSERTION code_insertion_tail
       ;

semicolons : /* empty */
           | semicolons  SEMICOLON
           ;
```

8 Appendix 2 - Example of MSTA scanner description file

```
%local {

#include <stdio.h>
#include <string.h>

#define IDENTIFIER 300
#define NUMBER     301
#define STRING     302


static FILE *input;

static char lexema [100];
static int lexema_index;
static int lineno;

}

%scanner
%%
program :
        | program {lexema_index=0;} lexema
        ;
lexema : identifier {lexema [lexema_index++] = 0;return IDENTIFIER;}
       | number     {lexema [lexema_index++] = 0;return NUMBER;}
       | string     {lexema [lexema_index++] = 0;return STRING;}
       | comment
       | space
       | error
       ;

space : ' ' | '\t' | '\n' {lineno++;}

identifier : identifier (letter | digit)
           | letter
           ;

letter :  ('a'-'z' | 'A' - 'Z') {lexema[lexema_index++] = yysprev_char;}
```

```
       ;

digit :  '0' - '9'  {lexema[lexema_index++] = yysprev_char;}
      ;

number : number digit
       | digit
       ;

string : '"' (('\1' -> '"' | '"' <- '\377')
                        {lexema[lexema_index++] = yysprev_char;} ) * '"'
       ;

comment : "/*" '\0' - '\377' * "*/"  /* Conflict shift/reduce on / after * */
        ;

%%

#ifdef __cplusplus

class scanner: public yyscanner
{
public:
  inline int yyslex (void);
  void yyserror (const char *message);
};

int scanner::yyslex (void)
{
  return fgetc (input);
}

void scanner::yyserror (const char *message)
{
  fprintf (stderr, "illegal code %d on line %d\n", yyschar, lineno);
}

static scanner *scan_ptr;

#else

int yyslex (void)
{
  return fgetc (input);
}

yyserror (const char *message)
{
  fprintf (stderr, "illegal code %d on line %d\n", yyschar, lineno);
}
#endif
```

```
void
main (int argc, char **argv)
{
  int token;
  int error_flag;

  if (argc != 2)
    {
      fprintf (stderr, "Usage: lex file\n");
      exit (1);
    }
  if (strcmp (argv[1], "-") == 0)
    input = stdin;
  else
    input = fopen (argv[1], "rb");
  if (input == NULL)
    {
      perror (argv[1]);
      exit (1);
    }
#ifdef __cplusplus
  scan_ptr = new yyscanner (error_flag);
  if (error_flag)
    {
      fprintf (stderr, "no memory for object scanner");
      exit (1);
    }
  lineno = 1;
  while ((token = scan_ptr->yylex ()) > 0)
    fprintf (stderr, "%d - %s\n", token, lexema);
#else
  lineno = 1;
  yylex_start (_flag);
  if (error_flag)
    {
      fprintf (stderr, "no memory for scanner arrays");
      exit (1);
    }
  while ((token = yylex ()) > 0)
    fprintf (stderr, "%d - %s\n", token, lexema);
#endif
  exit (0);
}
```

9 Appendix 3 - Run of MSTA on the previous description file

This output is produced on Compaq Aero 486 SX/25 with 12Mb under Linux 2.0.29.

```
bash$ msta -time lex.y
```

```
  parser time            -- 0.02sec
  semantic analyzer time -- 0.11sec
  Look ahead is 1 token
      Creating LR(0)-sets -- 0.05sec
      Conflicts processing -- 0.10sec
    create LALR-sets only with needed contexts -- 0.10sec
      marking LR-sets with used attributes -- 0.01sec
      splitting LR-sets for regular optimization -- 0.01sec
      transforming LR-graph -- 0.04sec
      removal unused nonterm. LR-arcs & marking unreachable LR-sets -- 0.00sec
      marking semantically pushed LR-sets -- 0.00sec
      making concordance between pushed LR-sets -- 0.00sec
      making concordance between pushed LR-set attributes -- 0.00sec
      evaluating popped LR-sets & attributes of regular arcs -- 0.02sec
      stack displacement for used attributes evaluation -- 0.02sec
      searching for regular arcs equivalence -- 0.03sec
    all regular optimization -- 0.14sec
    making parser look-ahead trie -- 0.02sec
    making token equivalence classes -- 0.03sec
  all generation of internal parser representation -- 0.30sec
  Real look ahead is 1 token
  5 shift/reduce conflicts.
      translate vector creation & output -- 0.05sec
      action table creation & output -- 0.02sec
      nonterminal goto table creation & output -- 0.01sec
      pushed states flag table creation & output -- 0.00sec
      popped attributes number table creation & output -- 0.01sec
      token name table creation & output -- 0.03sec
    creation, compacting, output of tables -- 0.14sec
  all parser output -- 0.25sec
overall time -- 0.75sec
```