Missouri University of Science and Technology
Scholars' Mine

Computer Science Technical Reports | Computer Science

01 Jan 1987

LILY-A Generator for Compiler Frontends

Thomas J. Sager

Follow this and additional works at: https://scholarsmine.mst.edu/comsci_techreports

 Part of the Computer Sciences Commons

Recommended Citation

Sager, Thomas J., "LILY-A Generator for Compiler Frontends" (1987). *Computer Science Technical Reports*. 10.
https://scholarsmine.mst.edu/comsci_techreports/10

This Technical Report is brought to you for free and open access by Scholars' Mine. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Scholars' Mine. This work is protected by U. S. Copyright Law. Unauthorized use including reproduction for redistribution requires the permission of the copyright holder. For more information, please contact scholarsmine@mst.edu.

LILY-A GENERATOR FOR COMPILER FRONTENDS

Thomas J. Sager

CSc-87-1

Department of Computer Science
University of Missouri-Rolla
Rolla, MO 65401 (314) 341-4491

LILY - A Generator for Compiler Frontends

Thomas J. Sager

In this paper, LILY, a generator for compiler frontends is described. LILY uses a generator of minimal perfect hash functions, MPHF, to create small fast compilers. The saliant features of LILY are that:

1. LILY accepts multi-level grammars. That is, LILY accepts frontend specifications that contain an arbitrary number of grammars, such that the terminal symbols of the ith grammar are the goal symbols of the i-1th grammar. The output from LILY contains a sequence of parsers. Each parser gets its input by invoking the parser at the previous level. A source handler which produces terminal symbols for the lowest level grammar should be part of the frontend specification. Basically a compiler frontend might be specified by two grammars, the lowest level being a lexical analyzer and the highest a parser, although more levels are possible.

2. The grammars in LILY's input specification may be regular right part grammars, that is the rightside of each production is a finite automaton rather than a string of symbols.

3. Lily creates table driven parsers at each level. The tables are organized as minimal perfect hash tables. That is the space required for the tables is in a sense minimal and the speed of access is in a sense maximal. All transitions are either default or non-default. The non-default transitions are placed in a transition table of minimal size and accessed by state and token through the function:

```
h := (indstatetab[state] + indtokentab[token]) mod tablesize;
```

Tablesize is exactly the number of non-default transitions. If a non-default transition on a (state, token) pair exists then it is at position h in the transition table. There is exactly one default transition for each state which is placed in a default table. This feature makes LILY truly unique among compiler generators.

4. In order to create small tables, LILY expects the grammars in its input specification to be LL(1). In the event that one is not, LILY expects to be given disambiguating actions at each point where the grammar supplied is not LL(1). These actions may be anything that the user choses, but in particular may include:

   a. lookahead.

   b. lookback at the state of the parse stack.

c. information from the symbol table.

d. the state of user defined variables.

5. Semantic analysis and code generation may be performed through attribute translation although it is possible to use less formal methods also. Basically, for each grammar, we group the goal symbols together as the start non-terminal. For each nonterminal, the user may then declare variables as attributes of that nonterminal and routines for translation and manipulation of those attributes. These routines are invoked on specific (state, token) transitions in the same manner as the disambiguating actions.

6. LILY is partially interactive in that on discovering certain kinds of errors, LILY queries the user on how to fix the error. If the user requests a fix, LILY makes the requested changes to the source and continues to translate.

It is feature 3 above that makes LILY unique among compiler generators. However, the author knows of no other compiler generator that combines the other five features above in the manner that LILY does.

LILY is currently under development. The design phase has been completed and is described herein. Parts of LILY have been coded and tested and results from these portions are encouraging. A PASCAL frontend generated in part by LILY is described in [7].

In designing LILY the author drew heavily on experience with YACC [3] and LEX [5]. He was also influenced by the LL(1) attributed parser generator VATS [1] and by Jullig and DeRemer's work on attributed regular right part grammars [4].

The input to LILY is a specification for a frontend. The output is a source program in some programming language. The prototype version currently under development generates Turbo PASCAL source code. It is contemplated that subsequent versions will be developed for C and for a somewhat more robust version of PASCAL. These subsequent versions might require some minor changes in the specification language. The Syntax for the prototype version is given in BNF form in appendix 1. Appendix 2 contains a short example, a specification for an integer calculator.

Basically the input to LILY is a multilevel grammar, that is a sequence of grammars, each one taking the set of goal symbols from the previous grammar as its own terminal symbols and in its turn producing goal symbols which become terminal symbols for the next grammar. The output of LILY includes a sequence of parsers. Each one, on invocation produces one of its own goal symbols from terminal symbols obtained through invocations to the previous parser. Traditionally compilers have been built on two level, the lowest being called a lexical analyzer or scanner and

the highest a syntactic analyzer or parser. Because our method of table generation is only practical up to a certain size, approximately 512 non-default transitions, we include the possibility of breaking up a grammar into more levels in order to keep the size of the tables small for large languages. Also, we make no distinction syntactically between the different levels. If the specification for the lexical analyzer is a regular grammar, which is most often the case, LILY will recognize this and omit code for pushing and popping syntactic information.

For each parser the set of goal symbols is augmented by the special goal symbols _eofile and _error. _eofile denotes that the previous parser also produced _eofile. _error denotes that the current parser was unable to find any of its goal symbols. After the symbol _error in the specification a recovery sequence to be activited in case of error may be declared. In addition each parser may declare a special goal symbol %NULL which acts as a token separator as its value is not returned to the calling routine.

Each parser has an attribute part, a local declaration part and parts for specifying sets of terminal symbols, goal symbols and other non-terminal symbols.

The attribute part for each parser consists of declarations for variables which are used for attribute information. These attributes can be either initialized by the higher level parser for the lower level parser, (inherited) or initialized by routines within the lower level parser and returned to the higher level parser, (synthesized). These attributes are set up as global storage rather than passed parameters as it is often necessary to save them from one invocation of the higher level parser to the next. In addition, parser attributes may be saved on a system generated semantic stack for later use.

The local declaration part may be used for any constants, types, variables or routines that are needed for disambiguation rules, semantic analysis or code generation.

In addition, attributes and local declarations may be associated with each symbol declared in the nonterminal part. Unlike the parser attributes, these nonterminal attributes are coded as local to the parser. These attributes may be designated with one of three keywords, %IN (inherited), %OUT (synthesized) or %INOUT (bidirectional). On entering the start state associated with a nonterminal, these attributes are automatically initialized from their arguments. On returning from a final state associated with a nonterminal, these attributes' values are returned to their appropriate arguments. Thus, these attributes act as variable parameters, although they are coded as local variables. Like the parser attributes, these attributes may be placed on a system generated semantic stack. In order to avoid conflicts between attributes to different nonterminals, these identifiers are suffixed with the ordinal number of the nonterminal to which they are associated.

Thus, in LILY, attributes are handled slightly differently at the vertical interface between parsers than they are handled within a parser. This is because invocations of the parsers are for the most part system generated whereas invocations of semantic routines must be specified by the user. In addition, attributes are handled less formally in LILY than they might have been. The author feels, however, that the method described above allows for a good mix between formal structure and flexibility.

After the keyword %SET, identifiers may be declared to represent sets of terminal symbols, (goal symbols from the previous parser). These set identifiers can then be used within the specification of the rightside of a production to represent transitions on each member of the set.

After the keyword %GOAL, identifiers may be declared to represent goal symbols. Along with each goal symbol a finite automaton which represents the specification of the goal symbol is given. After the keyword %NTRMNL, identifiers may be declared to represent nonterminal symbols other than the start symbol. Along with each nonterminal symbol, a finite automaton representing the specification of the nonterminal is given. In addition, attributes and local declarations may be specified here as associated with a nonterminal.

A finite automaton is coded as a set of paths. At the beginning and after each token in the path a state identifier may be optionally specified. The state identifiers are unsigned integers. The prefix, $, means is a final state. The tokens may be terminals, terminal sets, non-terminals or one of the two special symbols # or @. # denotes the empty string, whereas @ denotes a default transition, that is, one that is taken on terminals for which no other transition is specified. This can be useful, for example, if one wishes to build multiple error states into the specification.

It should be noted that the specifications of a finite automaton in LILY include specifications equivalent to standard BNF as well as to most of the common extentions of BNF. In most cases BNF or a common extention will suffice for specifying the rightside of a production, however the ability to specify a general finite automaton is helpful in certain cases.

The lowest level set of goal symbols is declared after the keyword %TRMNL, as a list of identifiers. Attributes of these lowest level goal symbols and the function that produces them is specified after the keyword %SOURCE. Arbitrary source language code can be written in the first and final sections of the specification, after the %BEGIN and the %ENDPARSE keywords respectively. These sections of source code are checked for syntactic correctness in the target language and then copied verbatim to the output. Thus, LILY can create an entire compiler provided than the backend is included in these two sections.

A parser specification is translated into the short procedure in figure 1 along with the four tables: indstatetab, indtokentab, transitab and defltab. The first three tables along with the statement labeled 1: form a MPHF. With the default table and disambiguating actions they specify an entire parser.

In order to built the tables, LILY first uses standard techniques to create a deterministic finite automaton, DFA, for each nonterminal. It then uses standard LL(1) lookahead techniques to describe transitions to the start states of each nonterminal's DFA. Where there is a conflict, a transition to a global error state is created. This transition, however can be overriden by a user supplied disambiguating action. In addition, error recovery can be included in the specification through the use of special transitions for ad hoc error recovery or default transitions for more formal methods of error recovery.

The transition set is partially optimized by the following two techniques:

1. For each state, unless a default transition is specified by the user, the non-shift transition to be performed on the largest set of terminal symbols becomes the default transition for that state.

2. Where a sequence of one or more transitions are uniquely determined to follow a given transition, they are coded as a single transition.

The tables are then produced by the mincycle algorithm for generating MPHF's which is described in [6] and [7]. In [7], a parser for PASCAL containing 196 non-default transitions is generated by the mincycle algorithm. It is shown in [2] that these algorithms can be expected to produce the required tables, even when the number of non-default transitions is as large as 512. Although to the authors knowledge no specific comparison data exists, parsers generated in this manner will in general be faster and smaller than parsers generated by other known algorithms.

In order to translate properly, LILY contains a parser for the target language. Thus, sections of the specification which contain target language code are checked for syntactic correctness by LILY.

Actions can be used for disambiguating syntactic conflicts, semantic analysis, code generation or error recovery. Syntactic conflicts occur wherever the language specifications are not LL(1). The reason LL(1) was chosen rather than SLR(1), LALR(1) or some other subset of LR(1) was because of the desire to use MPHF's to generate small fast table driven parsers. Since it was desirable to keep the number of non-default transitions as small as possible, LL(1) was a natural choice. Also, experience shows that most programming languages can be put in a convenient almost LL(1) form in which existing ambiguities can be resolved through

fairly simple disambiguating actions such as a symbol table lookup or a lookahead of one symbol further.

In sum, the truly novel feature of LILY is its use of minimal perfect hash functions to create small fast table driven parsers. Other important features of LILY are:

1. The use of multi-level grammars.

2. The use of regular right part grammars.

3. The use of LL(1) parsing techniques with disambiguating actions at all levels.

4. An attributed method of semantic analysis and code generation.

5. A partial interactive capability.

Figure 1: Target Language Parser

```
     {  N and NM1 stand for character strings representing  }
     {  level number and level number - 1 respectively      }
     {  All identifiers generated by LILY begin with '_'    }

const
     _start  =  0;  {state}        _maxtransN    =  ;
     _return = -1;  {state}        _maxstateN    =  ;
     _eofile =  0;  {token}        _maxtokenN    =  ;
     _error  = -1;  {token}        _maxparstackN =  ;
                                   _maxattstackN =  ;
     { declarations for goal symbols }

type
     _transrec = record state, token, next: integer end;

var
     _parstackN:      array [0.._maxparstackN] of integer;
     _parstacktopN:   integer;
     _attstackN       array [0.._maxattstackN] of integer;
     _attstacktopN:   integer;
     _tokenN:         integer;
     _transitabN:     array [0.._maxtransN] of _transrec;
     _defltabN:       array [0.._maxstateN] of integer; {next}
     _indstatetabN:   array [0.._maxstateN] of integer;
     _indtokentabN:   array [0.._maxtokenN] of integer;
          {  parser attribute declarations  }

function _parserN:   integer;
  var _state, _h:       integer;
     {  local and nonterminal associated declarations  }
  begin
    _state  := _start; {  start state  }
    repeat
  1:  _h := (_indstatetabN[_state] + _indtokentabN[_tokenN]) mod
            (_maxtransN + 1);
      if  (_transitabN[h].state = _state) and
          (_transitabN[h].token = _tokenN) then begin
        _state := _transitabN[h].next;
        case_ h of
          { actions, both user specified and system generated }
        end {case};
        _tokenN := _parserNM1;
      end else begin
        _h := _state;
        _state := _defltabN[_state];
        case _h of
          { actions, both user specified and system generated }
        end {case};
      end {if};
    until _state = _return;
  end {_parserN};
```

References

[1] Berg, A. et al.: VATS - the visible attributed translation system. Tech. Rep. 84-19 Department of Computational Science, Univ. of Saskatchewan, 1984.

[2] Hou, P.P. and Sager, T.J.: A Monte Carlo analysis of the mincycle algorithm for generating minimal perfect hash functions. Tech. Rep. CSc-85-3 Univ. Missouri-Rolla, 1985.

[3] Johnson, S.C.: YACC: yet another compiler compiler. Computing Services Tech. Rep. 32, Bell Labs, Murray Hill, N.J., 1975.

[4] Jullig, R. and DeRemer, F.L.: Regular right part attribute grammars. ACM Sigplan Notices, 19, 6, June 1984 (171-178).

[5] Lesk, M.E. and Schmidt, E.: Lex - A lexical analyzer generator. Computing Services Tech, Rep. 39, Bell Labs, Murray Hill, N.J., 1975.

[6] Sager, T.J.: A polynomial time generator for minimal perfect hash functions. ACM Communications, 28, 5, May 1985, (523-532).

[7] Sager, T.J.: A technique for creating small fast compiler frontends. ACM Sigplan Notices, 20, 10, October 1985, (87-94).

Appendix 1: BNF Description of Input Specification for LILY.

```
     Note:   # denotes the empty string;
     Note:   -- to end of line is a comment

<specification>-> %BEGIN <source_code> %TRMNL <trmnlpart>
                  %SOURCE <attribpart> ';' <block> ';'
                  <parspart> %ENDPARSE <source_code> %END
<parspart>      -> <parser> <parspart> | #
<parser>        -> %PARSER <attribpart> <localpart> <termsets>
                  <goals> <nontrms>
<source_code>   -> { part of a target language program }
<trmnlpart>     -> <tokenid> <tokenidlist> | <tokenid>
<tokenidlist>   -> ',' <tokenid> <tokenidlist> | #
<attribpart>    -> '(' <vardecl> <vardeclist> ')' | #
<vardeclist>    -> ';' <vardecl> <vardeclist> | #
<vardecl>       -> { a target language variable declaration }
<localpart>     -> { a sequence of target language declarations }
<block>         -> { a target language block }
<termsets>      -> %SET <setdecl> <setdeclist> | #
<setdeclist>    -> <setdecl> <setdeclist> | #
<setdecl>       -> <tokenid> '=' <not> '{' <tokenid> <tokenidlist>
                  '}'
<not>           -> '-' | #
<goalpart>      -> %GOALS <goaldecl> <goaldeclist> | #
<goaldeclist>   -> <goaldecl> <goaldeclist> | #
<goaldecl>      -> <tokenid> '->' <automaton> ';'
<ntrmlpart>     -> %NTRMLS <ntrmldecl> <ntrmldeclist> | #
<ntrmldeclist>  -> <ntrmldecl> <ntrmnldeclist> | #
<ntrmldecl>     -> <tokenid> <ntattribpart> ';' <localpart> '->'
                  <automaton> ';'
<ntattribpart>  -> '(' <direction> <vardecl> <ntdeclist> ')' | #
<ntdeclist>     -> ';' <direction> <vardecl> <ntdeclist> | #
<direction>     -> %IN | %OUT | %INOUT
<automaton>     -> <path> <pathlist> | #
<pathlist>      -> '|' <path> <pathlist> | #
<path>          -> <stateid> <term> <path> | <stateid>
<term>          -> <factor> <postop>
<factor>        -> '{' <pathlist> '}' | <token> <argpart> <action>
<token>         -> <tokenid> | '#'      -- # denotes empty string
                 | '@'                  -- @ denotes default
<argpart>       -> '(' <expr> <exprlist> ')' | #
<exprlist>      -> ',' <expr> <exprlist> | #
<expr>          -> { any target language expression }
<action>        -> '[' { a target language statement } ']' | #
<stateid>       -> <final> <stateno>
<final>         -> '$' | #
<stateno>       -> unsigned_integer | #
<postop>        -> '*' | '+' | '?' | ':' lbound ':' ubound
<lbound>        -> unsigned_integer
<ubound>        -> unsigned_integer | '*' -- * means no up bound
<tokenid>       -> { a target language identifier }
```

Appendix 2: LILY Specification for an Integer Calculator.

```
%BEGIN

program calculator; { for integer values only }
  const
    starno      =   0;
    divno       =   1;
    modno       =   2;
    plusno      =   0;
    minusno     =   1;
    maxsymtab   = 255;
  type
    stringn = string[8];
  var
    symtab: array [0..maxsymtab] of
              record str: stringn; value: integer end;
  procedure initsymtab; ...;        { initialize symbol table }

%TRMNL

  letter, digit, pluschar, minchar, starchar, oparchar,
  cparchar, colonchar, eqchar, blank, eoline

%SOURCE(charval: char);      { source handler }
                      { source function will be named _parser0 }
begin
  if eof then _parser := _eofile
  else if eoln then begin _parser := eoline; readln; end
  else begin
    read(charval);
    case charval of
      'a', ... ,'z':
        begin _parser := letter; charval := upcase(charval) end;
      'A', ... ,'Z': _parser := letter;
      '0', ..., '9': _parser := digit;
      ' ':           _parser := blank;
      '+':           _parser := pluschar
      '-':           _parser := minuschar;
      '*':           _parser := starchar;
      '(':           _parser := oparchar;
      ')':           _parser := cparchar;
      ':':           _parser := colonchar;
      '=':           _parser := eqchar;
      else           _parser := _error;
    end {case};
  end {if};
end {source};
```

```
%PARSER(buff: stringn; tokenvalue: integer); { lexical analyzer }

  procedure rswdlkup;
    begin
      if buff = 'DIV' then begin
        _parser := multop;  tokenvalue := divno;
      end else if buff = 'MOD' then begin
        _parser := multop; tokenvalue := modno;
      end {if};
    end {rswdlkup};

     %SET

letdigit = {letter, digit}

     %GOAL

%NULL  -> blank ;
ident  -> letter [buff := charval]
          {letdigit[buff := buff + charval]}* -- + is concatenate
          @[rswdlkup]; -- default action to check for DIV, MOD
number -> digit [tokenvalue := ord(charval) - ord('0')] {digit
          [tokenvalue := 10*tokenvalue+ord(charval)-ord('0')]}*;
addop  -> pluschar  [tokenvalue := plusno] |
          minuschar [tokenvalue := minusno];
multop -> starchar  [tokenvalue := starno];
opar   -> oparchar;
cpar   -> cparchar;
asgnop -> colonchar eqchar;
endmrk -> eoline;

%PARSER ;

  var strng: stringn; value: integer;

  procedure symtabstore(strng: stringn; value: integer); ...;
  function  symtabretrv(strng: stringn): integer; ...;

  procedure display_store(strng: stringn; value: integer);
    begin
      writeln(strng, ' <- ',value); symtabstore(strng, value);
    end {display_store};

     %SET

others   = - {eoline, eofile}

     %GOAL

%NULL  -> endmrk;
_error -> {others}+ [writeln('*** ERROR ***')] ; -- flush line
asmt   -> ident [strng := buff] asgnop expr(value)
          [display_store(strng, value)] eoline;
gexpr  -> expr(value) [writeln(value)] eoline;
```

```
      %NTRMNL

expr(%OUT value: integer)
  var opno, tempval: integer;

  function binaryop(x,n,y: integer): integer;
    begin
      case n of
          plusno:  binaryop := x+y;
          minusno: binaryop := x-y;
      end {case}
    end {binaryop};

  function unaryop(n,x: integer): integer;
    begin
      if n = minusno then unaryop := -x else unaryop := x
    end {unaryop};

  ->  sign(opno) term(value)[value := unaryop(opno, value);]
      { addop [opno := tokenvalue] term(tempval)
      [value := binaryop(value, opno, tempval)] }*

term(%OUT value: integer);
  var opno, tempval: integer;

  function binaryop(x,n,y: integer): integer;
    begin
      case n of
          starno:  binaryop := x * y;
          divno:   binaryop := x div y;
          modno:   binaryop := x mod y;
      end {case}
    end {binaryop};

  -> fact(value) { multop[opno := tokenvalue] fact(tempval)
     [ value := binaryop(value, opno, tempval) ] }*

fact(%OUT value)
  -> oprn expr(value) cprn ! ident[ value := symtabretrv(buff) ];

sign(%OUT opno)
  -> addop [opno := tokenvalue]
   ! # [opno := plusno];  -- # means null string

%ENDPARSE

begin {calculator}                    { driver            }
  initsymtab;
  _token1 := _parser0;                { source handler    }
  _token2 := _parser1;                { lexical analyzer  }
  while _parser2 <> _eofile do ;      { parser            }
end {calculator}.

%END
```