Jurnal Kejuruteraan 3 (1991) 71 - 79

Compiler-Compiler for an Eight Bit Microprocessor

Kasmiran Bin Jumari
K.R. Dimond

ABSTRAK

Kertas ini memaparkan teknik yang boleh digunakan untuk merekabentuk penghimpunan-penghimpunan bagi satu mikroprosesor lapan bit. Dengan menggunakan penghantar yang biasanya sama dalam setiap aturcara, kerja merekabentuk aturcara yang bergantung kepada penggunaan dapat dipermudahkan. Apa yang penting di sini adalah perekabentuk yang biasanya merekabentuk sistem berdasarkan mikroprosesor akan dapat menghayati kemudahan yang di beri oleh penghimpun-penghimpun seperti YACC dalam rekabentuknya.

ABSTRACT

This paper shows the technique that can be used to design a compiler-compiler for an eight bit microprocessor. Using the translated parser which is the same in every application program, the task of designing application dependent programs is greatly eased. Of particular interest is the power of a Yet Another Compiler-Compiler (YACC) can now be appreciated by the microprocessor based designer.

INTRODUCTION

It is possible to use any high level language to write a compiler or an interpreter, but the process is eased if the implementation language has constructions suited to the task. The compiler-compiler is such a system which enables a compiler or interpreter to be generated semiautomatically. Historically the existence of a compiler-compiler is a result of using syntax directed compiling techniques in order to structure a compiler. It brought us as far as early 60's. Since then several compiler-compilers have been introduced including the CDL [Koster, 1974] compiler-compiler which has been used in writing Manchester ALGOL68 and YACC (Johnson 1977) (or yet another compiler-compiler) with which compilers for C, APL, Pascal, Ratfor etc have been written.

Essentially the task of a compiler-compiler is to produce a compiler from some form of specifications of a source language and the target machine. The input specification may contain a description of the lexical and syntactic structure of the source language, a description of what output for each source language, and a description of the target machine.

Context-Free-Grammar (Aho et al. 1986a; Aho et al. 1986b; Gries 1971; Nijholt 1980) has widely been accepted in describing the lexical and the syntactic structure of a language. It is similar to the normal dictionary definition in which a grammar is arranged such that the nonterminal is defined in terms of the other terminals and nonterminals, which is exactly

the same as in dictionary where a word is defined in term of the other words. This has revealed the logical structure of this grammar. It has been employed in at least in one compiler-compiler, YACC, and the derivation of this grammar leads to the innovation of a two level grammar which was embedded in the CDL compiler-compiler.

Unfortunately, the above compiler-compilers are available mainly in main frame machines. This paper demonstrates of how such a compiler writing tool for an eight bit microprocessor can be prepared under the Unix operating system in which YACC resides.

YACC COMPATIBILITY WITH A MICROPROCESSOR ASSEMBLY LANGUAGE

YACC is an automatic parser generator which converts the grammar rules in the user input specification written in Context Free Grammar into a set of parsing tables. It requires a simple but effective parser driving routine that will parse statements in the language. During the process of specifying the syntax of the source language, YACC warns of any errors and ambiguities the grammar may have. Each rule or production of the grammar can be augmented with an action which contains the decription of what output is to be generated when the rule is recognized in the input process.

The parser driving routine calls the lexical analyser whenever a token is needed in the parsing process, whilst the parser that YACC built requires a controlling routine – the main program.

The YACC input specification may take the following form:

```
% {
C statements like # define, # include,
C variable declaration, etc.
        this section is optional
%}
YACC declaration section: lexical tokens,
precedence and associavity information, etc.
        this section is also optional
%%
grammar rule section; the associated action is
written in between {and}
%%
more C statements
the main program; min( ) {...},
the lexical analyser; yylex( ) {...},
etc.
this section is optional.
```

In fact, the parser generator only considers the grammar rules of the source language and the supplied YACC declaration (if any). The content of the associated action is not touched by YACC, instead it is reproduced in a C switch statement in the parser driving routine, y. tab. c.

The layout of the YACC output file, y.tab.c, for the above input specification is as follows:

```
C statements written between % {and %}
more C statement – anything written after
the second %% such as
the main program; main ( ) {...},
the lexical analyser; yylex ( ) {...},
etc.
***the parsing table generated from the grammar.
***parser driving routine;
  yyparse ( ) {
  ...
  ...
  actions bounded in
  a switch statement
  ...
  }
```

Eventhough YACC works in the C environment, YACC itself does not understand C. Generally speaking, if the main program, the lexical analyser and the actions are written in the assembly language, only the parser driving routine and the parsing tables have to be translated into the assembly language in order to get the equivalent output program of y.tab.c. One must bear in mind that the parser driving routine is the same in all YACC output files, so that the tedious work of manually translating the parser driving routine is repaid when it is included in every output program.

The parsing tables generated from the grammar rules comprise of eight arrays in which the number of elements in each array varies depending on the grammar specification. However, the value of each element is well within the range of signed 15 bits numbers (-32768 to 32767), which can be represented by a word or two bytes. Each of them may now be reproduced in the format used by an assembler in declaring a constant word or a constant byte such as 'fdb' (form double byte) statement or the 'fcb' (form constant byte) statement. In the latter case, each element has to be represented in two digits numbers based on 256.

The conversion of the parsing tables to their equivalent tables in the assembly language is straight forward, for example , for any x:

```
if x = 0, then the equivalent digits are 0 and 0.
if x > 0, then the equivalent digits are
        (x div 256) and (x mod 256).
if x < 0, as usual find its two's complement.
        The two's complement number is z = 65565 – |x|,
          and then the equivalent digits are
          (z div 256) and (z mod 256).
```

As mentioned earlier the actions which are associated in the grammar section are reproduced in a C switch statement of y.tab.c. It may look like

```
switch (yym)
case 3:
{ACTION 1} break;
```

```
case 4:
{ACTION 2} break;
...
...
...
{LAST ACTION} break;
}
```

In translating the y.tab.c to its equivalent output in the assembly language, it is easier to compile them as a subroutine, let say 'action', which will be called by the parser driving routine.

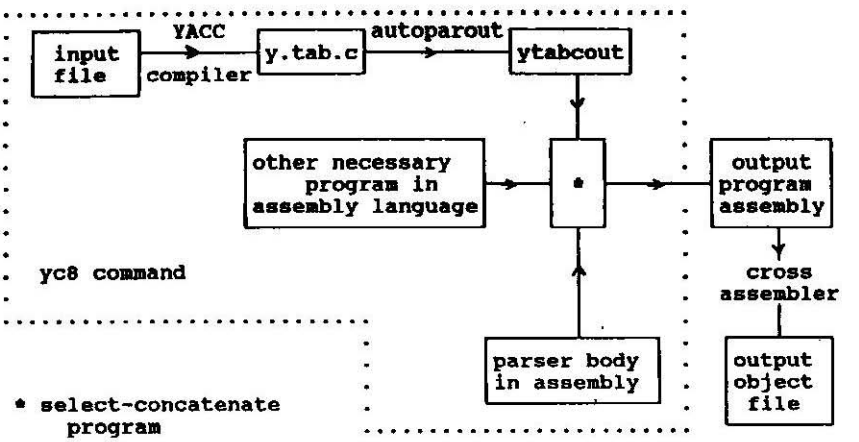
```
action:  ldx     yym      ;load a register with switch
                          ;control value.
         cpx     #3       ;compare it with the first case, say 3.
         bne     f00      ;if not equal skip forward to next case.
         ACTION 1         ;else perform ACTION 1.
         jmp  break       ;and then 'break' – as in C.
f00:     cpx     #4       ;compare the register with the second
                          ;'case', say 4.
         bne     f01      ;if not equal, skip to next 'case'.
         ACTION 2         ;the second action.
         jmp     break
f01:     ...
         ...
         ...
         LAST ACTION      ;last action in the 'switch' statement.
         jmp break
break:   rts              ;return to the parser driving routine.
```

DEVELOPING AN EIGHT BIT COMPILER-COMPILER

Figure 1 depicts the steps taken by the compiler-compiler in order to get the object program (in a microprocessor machine code) from an input file which posesses the syntactic structure of a language. The input specification file is first compiled by YACC to produce output file y.tab.c. Then, a converter program, 'autoparout' is invoked. The converter program has functions as follows:

1. to relocate and reformat the declared variables.
2. to reproduce the parsing tables in the format used in the assembly language.
3. to extract the actions routine from the C switch statement, and reproduce it in an assembly subroutine called 'action'.
4. to collect the lexical token that has been declared in the YACC declaration section so that the YACC-defined token can be freely used in other parts of the assembly program without the need to define it separately.

The file 'ytabcout' is a temporary file that contains the output of the converter program. The 'select-concatenate' program is then invoked



RAJAH 1. Preparing an assembly object program from a given grammar

which has a function of selecting and concatenating the 'ytabcout' and the other necessary assembly routines that have to be included in the output program. Incidently, the task of the select-concatenate program can easily be included in the main control program, yc8. Finally, the output program is assembled by a cross-assembler to produce the object program for execution on a target microcomputer board.

MICRO-COMPILER-COMPILER AT WORK

The compatibility realization has led to the development of at least in one microprocessor system, the Motorola 6800. The said compiler writing tool accepts an input file as follows:

YACC declaration section.
%%
Grammar rule section.
%%
The assembly program section such as the lexical analyser,
the main program etc. (optional)
%%
Assembler declaration section (optional)

Comparing the above input specification with the YACC input specification, other then the language used in preparing the input file, one obvious difference is the way in which the language variable declaration are done (a C declaration is written in between % {and %}, the assembler places them after the third %% marker).

The converter program, 'autoparout' has been written in Pascal (Grogono 1984). In addition to the function outlined in the previous section, the employed converter program is also capable of simplifying the communication task between the action and the parser driving routine. It is

done by using statements similar to the function of an assembler macro statement. These enable the user to include arithmetic operations in the action associated with the grammar in a more systematic fashion, as well as to return a value to, or to obtain a value from the value stack employed by the parser.

The syntax of the 'macro' statement is

.macro operand1 = operand2 operator operand3

.macro operand1 = − operand2

.macro array [operand1] = operand3

.macro operand1 = array [operand2]

where

operand1 = {!!, string}

operand2/3 = {!!, ! digit, digit, string, @hexa,

#digit, #string, #@hexa}

array = {string}

operator = {*,/, +, −, %, &}

The symbol ! is in fact equivalent to the symbol $ in YACC. The dollar symbol is not used here because it is a metacharacter of YACC. However, the function of the two symbols are the same. The assembler available on the author's system recognize a hexadecimal number if it is preceded by a dollar sign. The symbol @ was used to replace the YACC metacharacter symbol which will be converted back to a dollar sign in the assembler program. Symbol '#' carries its usual meaning 'with'. The arithmetic operation is for a signed or unsigned two byte number. The control program, yc8 (YACC for 8 bits micro) has been written in C shell. The syntax of the command is

yc8 Micprocessor type [−P] [−Oprogram origin]
[−Qparser origin] [−Mmain_program] [−Llexical_analyser]
[−Eerror_routine] [−Foutput_file] File_name

The microprocesser type is mandatory. It takes one of the following microprocesors name; the 6800, 6809, z80, 8085 or 6502. Different microprocesser requires different converter program, different parser driving routine which must be selected accordingly in the output program.

The options provide the user to freely configure the output file. For example, if the parser driving routine and the utility programs have been loaded to the system, the subsequent execution programs do not necessary to have them. It is done by using the option −P. By using this method the compiled program is smaller. The user may also specify a new program origin, parser routine origin, main program etc. by invoking the correspond options.

EXAMPLE PROGRAM

As a comparison, the same example program in YACC (Appendix A of Johnson 1977) is used in this paper, It has been rewritten to suit the specification required by the new compiler-compiler. It is a program for a small calculator that process integer number in the range of −32768 and 32767. Note that the macro statement has simplified the action routine that have to be written otherwise. The full listing of the example program is in the Appendix A.

CONCLUSION

Provided that there is a suitable compiler-compiler such as YACC, a comparable compiler-compiler for a microprocesser is not difficult to be developed. As the central core of the program is the same for all application programs, debugging the program is narrowed down only to the input specification containing the program rules. In some extend only the action routines which associate the grammar rule is required. The syntax of the grammar is checked by the compiler-compiler, YACC, whilst the main program, the lexical analyser and the error routine are similar in all application program. That means producing program for a variety of different application can be made more reliable. On the other hand, such a compiler-compiler open the opportunity to a hardware oriented microprocessor users to experiment the new technique of programming in their design. One good example of area where such a compiler-compiler is needed is in the development stage of designing system for a man-machine dialogue in instrumentation.

REFERENCES

Aho, A.V. & Ullman, J.D. 1986a. *Principle of Compiler Design.* Addison Wesley.

Aho, A.V., Sethi, R. & Ullman, J.D. 1986b. *Compilers-Principles, Techniques, and Tools.* Addison Wesley.

Gries, D. 1971. *Compiler Construction for Digital Computers.* John Wiley.

Grogono, P. 1984. *Programming in Pascal.* Addison Wesley.

Johnson, S.C. 1977. *Yet Another Compiler-Compiler.* Bell Laboratories, Murray Hill.

Koster, C.H.A. 1974. *Using the CDLCompiler-Compiler, Compiler Construction: An Advanced Course.* Springer-Verlag.

Nijholt, A. & 1980. *Context-Free Grammers: Covers, Normal Forms, and Parsing.* Springer-Verlag.

Appendix A

The Example program.

```
    %start list
    %token DIGIT LETTER
    %left '|'
    %left '&'
    %left '+' '-'
    %left '*' '/' '%'
    % left UMINUS /* supplies precedence for unary minus
*/
    %% /* beginning of rules section */
    list    : /* empty */
            | list stat '\n'
            | list error '\n'
    {
              jsr mmerror
    }
            ;
    stat    : expr
    {
              ldaa #1       ;
              jsr  mmgetval ;equivalent to 'print $1'.
              jsr  mmprntno ;
    }
            | LETTER '=' expr
            {.macro regs [!] = !3}
            ;
expr        :     '(' expr ')'
                    {.macro !! = !2}
            |     expr '+' expr
                    {.macro !! = !1 + !3}
            |     expr '-' expr
                    {.macro !! = !1 - !3}
            |     expr '*' expr
                    {.macro !! = !1 * !3}
            |     expr '/' expr
                    {.macro !! = !1/!3}
            |     expr '%' expr
                    {.macro !! = !1 % !3}
            |     expr '&' expr
                    {.macro !! = !1 & !3}
            |     expr '|' expr
                    {.macro !! = !1 | !3}
            |     '-' expr %prec UMINUS
                    {.macro !! = -!2}
            |     LETTER
                    {.macro !! = regs[!1]}
            |     number
            ;
number:           DIGIT
                    {.macro !! = !1}
            |     number DIGIT
```

```
{.macro !! = #10 * !1
 .macro !! = !! + !2}

%%
lex:    jsr     inch        ;input character.
        psha                ;save the character.
        cmpa    #'a'        ;**
        blt     f1f         ;
        cmpa    #'z'        ;classify for LETTER.
        bgt     f1f         ;
        ldx     #LETTER     ;LETTER is returned.
        suba    #'a'        ;**
        bra     f2f
f1f:    cmpa    #'0'        ;**
        blt     f3f         ;
        cmpa    #'9'        ;classify for DIGIT.
        bgt     f3f         ;
        ldx     #DIGIT      ;DIGIT is returned.
        suba    #'0'        ;**
f2f:    staa    mmlval+1    ;assign the value at
                            ;the value stack.
        clr     mmlval
f3f:    pula                ;retrieve the input
                            ;character.
        rts
%%
regs:   rmb     52          ;52 spaces for the
                            ;array
```

Kasmiran Bin Jumari
Jabatan Kejuruteraan Elektrik, Elektronik dan Sistem
Universiti Kebangsaan Malaysia
43600 Bangi
Selangor D.E.

K.R. Dimond
Electronics Laboratory
University of Kent
England CT2 7NT.