TR 74-2

MICROPROGRAMMED SUBPROCESSORS FOR HIGH LEVEL LANGUAGES

Peter G. Houlton

Abstract

A set of four microcoded subprocessors have been designed for the Microdata 3200 to provide compact efficient compilation and execution of LR(k) languages. The scanner subprocessor extracts tokens from an input string; the parser subprocessor executes a stack automaton program which parses the source program; the semantics subprocessor provides operators for data handling in tables and stacks, symbol table searching, and object code generation; the execution subprocessor is a stack machine designed for ALGOL-like languages. The design and interaction of these subprocessors is discussed.

Introduction

This paper presents a set of four microprogrammed subprocessors which have been designed for the Microdata 3200 to provide generalized (language independent) compilation and execution of LR(k) languages. Several processors have been reported for machines which will "execute" a higher level programming language [2,5,7]. However, most of these have addressed the execution time problem of finding operators which are "natural" to compile into and provide efficient execution. Although an early report of a microprogrammed compiler clearly demonstrates the advantages of such an approach, the effort is tied closely to a particular language [10]. A more recent example of microcoded compiler methods is given by Chu [1].

The approach being reported here is not to microprogram a compiler, nor to design a machine for a particular language, but to analyze the process of executing a higher level language into four subprocesses: scanning, parsing, semantic functions, and execution; and then to design subprocessors which will provide suitable operators for compact, efficient compilers and efficient execution of a wide range of programming languages.

Scanner Subprocessor

The scanner subprocessor extracts tokens from an input string in main memory returning the following token codes in a general inter-subprocessor register:

0 - literal string

1 - digit string

2 - identifier string

3 - end of line

4 ... n - delimiter or special character codes and keyword codes

The scan is controlled by character type codes in a table in main memory indexed by character codes. Character types are encoded in 5 bits with one bit each for ignore (but use as digit or identifier string delimiters), digit, identifier character, end of line, and special character. Since the bits are tested in that order, a character may be encoded as more than one type. For example, while a digit/identifier character as a leading character of a token will indicate a digit string, the same character following an identifier character will be part of an identifier string.

Literal, digit and identifier strings are also packed into a token string buffer in main memory where they are available to a semantic routine invoked by a reduction of the terminal string to some non-terminal by the parser subprocessor. Identifier strings are tested as keywords by

searching a keyword table in main memory and obtaining keyword codes from a parallel table.

The current scanner subprocessor is encoded entirely in microcode providing fast execution but less flexibility than a design incorporating a macro language. However, some flexibility is achieved by the use of character type tables and keyword tables.

The scanner subprocessor has the following seven registers:

input string: main memory location of input string

current character: input character currently being scanned

current character type: type of current character

character type table: main memory location of character type code table

keyword table: main memory location of keyword table

keyword code table: main memory location of keyword code table

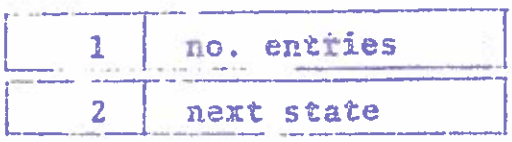
token string: main memory location of token string

Parser Subprocessor

The parser subprocessor is designed for parsing methods developed for LR(k) languages [3,6]. (Since the lookahead is currently one character, more accurately LR(1)). These methods employ a stack automaton parser derived from the LK(k) grammar of the language. The instructions implemented in the parser subprocessor are designed for a stack automaton which is described by a sequence of parsing subprocessor instructions. In general each of these instructions specifies some combination of stack manipulation, determination of a reduction, transition to a next state, and inputting the next token by a call to the scanner subprocessor. The stack of the automaton holds only state names. The transition to a next state can be specified unconditionally or dependently upon the top of the stack. The stack automaton program is generated in some other machine (currently a PDP-10) by an LR(k) parser generator. An example of a stack automaton parser for a very simple language is given in appendix A.

The specific instructions of the parser subprocessor are:

return to the control section of the semantics subprocessor indicating some error condition. (stack overflow, no transition, etc.)



pop no. entries off of stack

move unconditionally to next state

3	reduction code

return with reduction code (the next call to the parser subprocessor will enter at the next instruction in sequence)

The remaining instructions specify conditional moves determined by a test condition on the top of the stack or on the next input token. The instruction format is:

where depending upon the opc, test is a single token code, a single state, or the address of a set of token codes, or the address of a set of state names.

opc = 4: if test = top of stack, then move to next state and pop one entry from stack.

opc = 5: if test = top of stack, then move to next state.

opc = 6: if top of stack matches entry in set of names at test, then move to next state and pop stack.

opc = 7: if top of stack matches entry in set of names at test, then move to next state.

opc = 8: push current state name onto stack and if test = next token code, then move to next state and advance input one token.

opc = 9: same as opc = 8 except input is not advanced on successful test (this provides look ahead).

opc = 10: push current state name onto stack and if next token code matches entry in set at test, then move to next state and advance input.

opc = 11: same as opc = 10 except input is not advanced.

opc = 12 through opc = 15 are the same as opc = 8 through opc = 11 respectively except that the current state name is not pushed onto the stack.

In all of these cases, if the test is unsuccessful, control passes to the next instruction in sequence.

The parser subprocessor has six registers:

automaton base: main memory address of first automaton instruction

current state: offset from automaton base of current state

current instruction: offset from automaton base of current instruction

stack base: main memory address of first location of automaton stack

stack pointer: offset from stack base of current top of stack

stack limit: main memory address of last available stack location

The parser subprocessor also makes use of a general inter-subprocessor register to return condition and reduction codes.

Semantics Subprocessor

The semantics subprocessor provides instructions to facilitate the maintenance of information in tables and stacks, the generation of object code, decimal to binary conversion and input/output functions. The instructions include 16-bit integer arithmetic, partial word and character manipulation, simple pushing and popping of stacks, input/output, and symbol table searching. These instructions have been designed for the description of semantic routines associated with the productions of the grammar of the language.

In addition, the semantics subprocessor contains a control section for overall control of compilation and execution and a service section for providing input of source lines and output of listing. Both of these sections could have been designed as separate subprocessors. However, since they both make extensive use of instructions available in the semantics subprocessor, it seems most efficient to design them as sections of this subprocessor.

Since a major function of the semantic routines is the movement of data, an attractive design incorporates a number of general registers and two or three operand instructions similar to a PDP-11. The semantics subprocessor has sixteen registers all of which can be addressed, but eight of which serve the following special functions:

Program base register

current instruction location

current instruction

current reduction (returned by the parser subprocessor)

location of object program

status flags

two temporary microcode registers

The instruction operand addresses are determined by a four bit mode and a four bit register address, R. The modes are:

mode	effective address of operand
1	register R
2	contents of register R
3	increment contents of R by one, then contents of R
4	" " " " " two, " " " "
5	contents of R, then decrement R by one
6	" " " " " " " two
7	contents of R added to next 16 bits of instruction
8	contents of memory location addressed by contents of R added to next 16 bits of instruction
9	contents of R multiplied by 4 added to next 16 bits of instruction
10	contents of R added to 4 times the next 16 bits of instruction
11	contents of R multiplied by 4
12	contents of R multiplied by 4 incremented by 2

Main memory is byte addressable; thus modes 3,4,5,6 all manipulation of pointers for either byte or 16-bit word stacks. Input/output operations are performed by addressing the high end of an 18-bit addressing space. Modes 9,10,11,12 provide multiplication by 4 to produce the 2 high order bits.

In the following list of instructions, cond is a field specifying conditional execution of the instruction on the setting of a status flag for $=0$, $\neq 0$, >0, ≥ 0, <0 ≤ 0 previously set by a TST instruction. The fields source, destination, operand, string, table specify addresses as described above.

MOV cond,source, destination : move 16-bit word

MOVB cond,source,destination : move byte

MOVS cond,source,destination : move string

ARTH opr,operand,destination : arithmetic or logical operation on operand and destination with result placed into destination. opr specifies one of complement, add, subtract, multiply, divide, modulo, not, and, or, exclusive or

ARTH opr,operand1,operand2,destination: arithmetic or logical operation on two operands with result placed into destination.

SHFT opr,source,destination: shift of source with result placed into destination. opr specifies single or double word logical shift or rotate right or left and number of bits.

TST operand: tests operand against zero for = ≠ > ≥ < ≤ and sets status flags accordingly

TST operand1,operand2: test operand1 against operand2 for = ≠ > ≥ < ≤ and sets status flags

CONV string,destination: converts ASCII decimal digit string to binary and places result into destination

SRCH string,table,destination: uses a hash coding method to search table for string and places index of first matching or empty entry into destination. (entries of the table are pointers to strings)

XTR bits,source,destination: places partial word of source indicated by bits

The control section has instructions to initialize the parser subprocessor, the scanner subprocessor, and the execution subprocessor. An additional instruction provides a call to the parser subprocessor for the next reduction. The service section has an instruction for return to the scanner subprocessor.

The semantics subprocessor has been designed to facilitate describing the semantic routines in a simple higher level language in parallel with the productions of the grammar of the programming language being compiled. A compiler generator will produce the semantic routines for the semantics subprocessor as well as producing the parsing automaton. Although this semantic routine generator has not yet been implemented, it follows the approach taken by Feldman [4] and others.

Execution Subprocessor

The execution subprocessor provides instructions which will be "natural" for a compiler to generate and will provide efficient execution for a given language. Of course "natural" and "efficient" vary considerably for different languages; and several execution subprocessors might be designed. For a typical ALGOL-like language currently being implemented, the design should include typed arithmetic, a stack mechanism with display register addressing, and a procedure call mechanism.

Several designs for such stack machines have been reported for the direct execution of higher level languages [2,5,7]. Also the Microdata 32/S emulation on the 3200 is an example of a fairly sophisticated stack machine on a small computer [9]. One object of the project reported here was to design a simple, compact execution subprocessor which could share the hardware with the parser and semantics subprocessors.

A program in the execution subprocessor consists of a program area and a data or stack area. All variable storage is allocated in the stack and most data manipulation takes place on the stack. Two exceptions are constant values which may be read from the program area and input/output which is performed by absolute addressing at the high end of the memory addressing space.

Most operators implicitly address the top locations of the stack. Variables are addressed relative to the procedure in which they are declared. Each level of a nest of procedures has a display register pointing to its local storage. Variables are then addressed by two values: the number of levels out from the currently active procedure and the relative address within that level. The variable addresses are then evaluated relative to the stack base. An example is given in Figure 1.

Stack and store operations using these stack addressing methods and absolute addressing for both byte and word data move values to and from the top location of the stack. The following instructions obtain their operands from the top locations of the stack (popping the stack) and place their results back onto the stack:

integer add, subtract, multiply, divide, modulo

floating add, subtract, multiply, divide

convert integer to floating

convert floating to integer

complement

logical not, and, inclusive or, exclusive or

exchange the top two stack entries

branch

branch conditionally

shift

```
procedure x;
  begin a, b, c;
    procedure y;
      begin d, e, f;
        procedure z;
          begin g, h, i;
            f ⟵ h + a;
          end
      end
  end
```

Figure 1. Nested procedures. The assignment f ⟵ h + a would address f as (1,3), h as (0,2), a as (2,1)

extract partial word

deposit partial word

Procedure calls and returns are implemented by three instructions:

initialize procedure call: allocates heading block for activation of procedure and establishes links to outer procedure display.

call procedure: invokes the called procedure after the call has been initialized and any arguments have been placed on the stack and establishes a return address.

return: adjusts the stack to remove the current procedure heading block and returns control to the calling procedure.

Stack space for local storage is allocated with an allocate instruction.

The registers of the execution subprocessor are:

program base

current instruction (relative to program base)

stack base

stack pointer (relative to stack base)

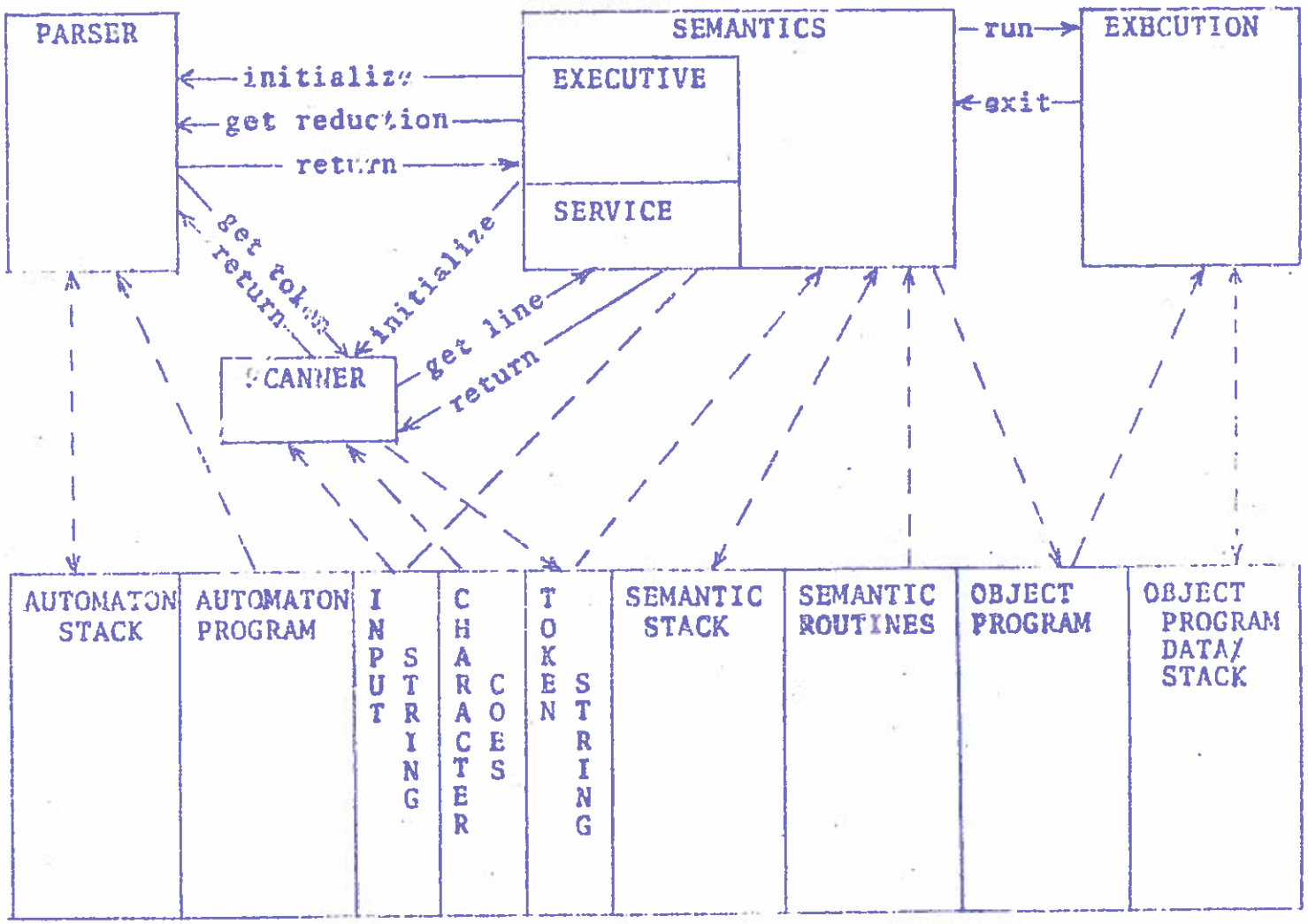
stack limit

top five entries of stack

most recently allocated procedure heading block

Interfacing

The organization of the subprocessors and their interfaces are illustrated in figure 2. The interface between the control section of the semantics subprocessor and the scanner subprocessor is an initialize instruction which initializes the scanner subprocessor character code table and keyword table registers and which causes the scanner subprocessor to request a first input line from the service section of the semantics subprocessor. The control section also has an initialize instruction to the parser subprocessor which initializes its registers and invokes a call to the scanner to initialize the first token. After initialization the control section call the parser subprocessor with a get next reduction instruction causing the parser subprocessor to execute the stack automaton program until a reduction instruction is reached. The code identifying the production associated with the reduction is returned to the control section which then uses this code to invoke the corresponding semantic routine through a branch vector. As the stack automaton requires input, the parser subprocessor makes get next token calls to the scanner subprocessor for the next token. As the scanner subprocessor exhausts input lines, get next line calls are made to the service section. The semantic routines output code to the object program area in main memory. The control section repeats this cycle until a stop bit is set in its status register. It then



SUBPROCESSOR ORGANIZATION

CONTROL FLOW ——————>

DATA FLOW - - - - ->

Figure 2.

invokes execution of the object program by a run instruction to the execution subprocessor. This instruction initializes the registers of the execution subprocessor.

As measured against some principles outlined by McKeeman [8], this interface is relatively clean in the sense that if the sections of the semantics subprocessor were separated, then all of the output of each subprocessor could be produced in some relatively simple intermediate language to be input to another subprocessor. In other words all input lines could be established in a file to be given one at a time to the scanner which could produce a file of all tokens to be given one at a time to the parser which could produce a file of reductions to be given one at a time to the control section which could call the semantic routines which would produce the file of object code.

The only breakdown of this interface is that required by the ad hoc nature of some of the error recovery techniques. In order to be useful to a programmer, a compiler must find as many errors as possible in a given program. This requires that the compiler not terminate upon encountering an error, but output an error indication and then adjust its current state in some way that will allow compilation of the remainder of the program to continue. Although work has been carried out in the area of error handling and recovery, at present no satisfactory general formal methods are available to realign the parser subprocessor after

encountering an error. When the parser encounters a syntactic error in the source language, it returns to the control section with an error condition code. The control section uses this code to invoke a "semantic" routine which outputs an error indication and then attempts to realign the parser whenever the parser cannot realign itself. These routines are usually designed for the error at hand and may involve advancing the input to some token, inserting some token into the input, adjusting the parser stack, and resetting the state of the parser automaton. As better methods for handling errors are developed, fewer of these violations of the inter-subprocessor interface will be necessary.

Hardware

The machine for which these subprocessors have been written is the Microdata 3200. The particular machine has 2K of writeable control store with a 32-bit microinstruction; it is relatively vertical in design. The data section contains three 16-bit working registers bussed into the arithmetic unit and thirty-two 16-bit general registers of local memory. The addressing of local memory makes special provision for four of these registers together with one of the working registers to be used as the top of a stack, the remainder of which extends into main memory. Main memory of this particular machine consists of 16KB of MOS memory. The particular machine used has at present no secondary file storage.

Although the machine has seemed quite versatile and easy to microprogram, a serious restriction in its design is that microcode subroutining is limited to only one level. Switches can be used for returns, but it is not possible to bring the current microinstruction address into the data section to be stacked as a return address. This restriction has seemed to limit the extent to which subprocessors could be modularized and the amount of code sharing that could be achieved.

Conclusions

The objectives of this work have centered around the problems of quickly and easily developing compact efficient compilers for small machines for research in programming language design. The need for ease in development of compilers suggests the need for a compiler generator. However, the limited memory of small machines normally will not support a sophisticated compiler generator. A compiler generator operating in a larger machine could produce compilers for a small machine with a general purpose instruction set. However, the microcoded subprocessors described here considerably simplify the task of a compiler generator and allow the resulting compiler to be considerably more compact and efficient. The compiler generator could easily be altered to generate compilers for a variety of small machines for which similar subprocessors have been implemented.

Another approach is to have the compiler generating system produce a completely microcoded compiler to be loaded into the control memory of the small machine. Although comparisons of this approach with the approach reported here have yet to be made, it appears that the completely microcoded compiler, although faster, would consume considerably more control memory; also the compiler generating system would become more complex and machine dependent. For the purposes at hand the subprocessors of this report seem to offer the most attractive method of implementing language processors.

References:

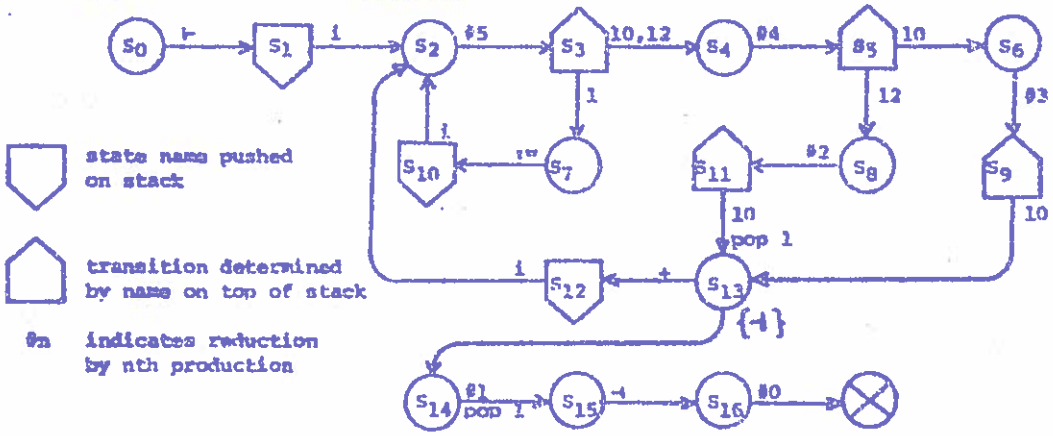
1. Chu Y., Recursive microprogramming in a syntax recognizer, Conference Record Sixth Annual Workshop on Microprogramming, September 1973
2. Broca F.R. and Merwin R.E., Direct microprogrammed execution of the intermediate text from a high-level language compiler, Proceedings of the Annual Conference of the ACM 1973
3. DeRemer F.L., Simple LR(k) Grammars, Communications of the ACM 14, 7(1971) pp 453-460
4. Feldman J.A., A formal semantics for computer languages and its application in a compiler-compiler, Communications of the ACM 9,1(1966) pp 3-9
5. Hassitt A., Lageshulte G.W.,Lyon L.E., Implementation of a high-level language machine, Report of the Fourth Annual Workshop on Microprogramming, September 1971
6. LaLonde W.R., Lee E.S., Horning J.J., An LALR(k) parser generator, Information Processing 71, 1972 pp 513-518
7. Lawson H.W., Jr., Programming-language-oriented instruction streams, IEEE Transactions on Computers C-17,5(1968),pp 476-485
8. McKeeman W., Compiler structure, Tech. Report CSRG-23 University of Toronto January 1973
9. Microdata Corp., Computer Reference Manual Micro 32/S, RM 20003250, May 1974
10. Weber H., A microprogrammed implementation of EULER on the IBM 360 model 30, Communications of the ACM 10,9 (1967), pp 547-558

APPENDIX A

Example Grammar:

0)	S →	⊢ A ⊣	3)	E →	P
1)	A →	V := E	4)	P →	V
2)	E →	E + P	5)	V →	i

GRAPHIC REPRESENTATION OF STACK AUTOMATON PARSER



Stack Automaton Program :

state	loc	one / operand or next state
S0	100	12 / ⊢
	101	103
	102	0 / 1001
S1	103	8 / i
	104	132
	105	0 / 1002
S15	106	12 / ⊣
	107	109
	108	0 / 1003
S16	109	3 / 0
S7	110	12 / :=
	111	113
	112	0 / 1004
S10	113	8 / i
	114	132
	115	0 / 1005
S13	116	13 / ⊣
	117	140
	118	12 / +
	119	121
	120	0 / 1006
S12	121	8 / i
	122	132
	123	0 / 1007

state	loc	one / operand or next state
S8	124	3 / 2
S11	125	4 / 113
	126	116
S4	127	3 / 4
S5	128	5 / 113
	129	137
	130	5 / 121
	131	124
S2	132	3 / 5
S3	133	7 / 143
	134	127
	135	5 / 1
	136	110
S6	137	3 / 3
S9	138	5 / 113
	139	116
S14	140	3 / 1
	141	1 / 1
	142	2 / 105
	143	2
	144	113
	145	121