```
        }
}

/*
 * Procedure to close the current input stream, and return to the previous
 * input stream.
 */

close_strm()
{
        struct file_elmt *new_file_ptr ;
        fclose( file_ptr->stream ) ;
        new_file_ptr = file_ptr->link ;
        free( file_ptr ) ;
        file_ptr = new_file_ptr ;
}

/*
 * Procedure to display the ASCII equivalent of the top 16 bit value on the
 * parameter stack.
 */

void emit_op()
{
#ifdef DEBUG
        printf("emit opcode\n") ;
#endif
        putchar( pop_ps() );
}

/*
 * Procedure to scan the keyboard and push the ASCII equivalent of the key
 * pressed onto the parameter stack as a 16 bit value.
 */

void key_op()
{
#ifdef DEBUG
        printf("key opcode\n") ;
#endif
        noecho() ;
        raw() ;
        push_ps( getchar() ) ;
        noraw() ;
        echo() ;
}

/*
 * Procedure to accept a line of text from the current input stream, and
 * transfer it to the text input buffer. If an EOF is encountered then the
 * input stream is reset to accept input from the keyboard. The text
 * interpreter offset is set to point to the begining of the text input
 * buffer reguardless of any stream directed action.
 */

void inline_op()
{
        char string[MAXLINE] ;
        unsigned short buffer = get_word( TIB ) ;
        int i = 0 ;
#ifdef DEBUG
        printf("inline opcode\n") ;
#endif
        if( fgets( string , MAXLINE , file_ptr->stream ) == NULL )
        {
                if( ! terminal() )
                {
                        printf( "[End of file]\n" ) ;
                        close_strm() ;
                }
                else
                        set_exit() ;
        }
        else
                for( i = 0 ; strcmp( &string[i] , "\n" ) ; i++ )
                {
                        if( string[i] == TAB )
                                string[i] = ( char ) SPACE ;
                        put_byte( buffer + i , string[i] ) ;
```

```
                }
        put_word( buffer + i , NULL ) ;
        put_word( IN , 0 ) ;
}

/*
 * Procedure to take the string pointed to be the IN offset within the
 * text input buffer, and attempt to open it as an input stream. If this
 * generates an error then the input stream is reset to accept input from
 * the keyboard, and an error messages is displayed.
 */

void file_op()
{
#ifdef DEBUG
        printf("file opcode\n");
#endif
        push_ps( SPACE ) ;
        word_op() ;
        open_strm( dp_string() ) ;
        forth_abort() ;
}

/*
 * Procedure to check if the input is indirect via the standard input
 * stream, and return a 16 bit value to reflect the result.
 */

void stdin_op()
{
#ifdef DEBUG
        printf("stdin opcode\n");
#endif
        if( terminal() )
                push_ps( FTRUE ) ;
        else
                push_ps( FFALSE ) ;
}

/*
 * Procedure to display the counted string pointed to by the contents of
 * the instruction register.
 */

void string_op()
{
        unsigned short count ;
#ifdef DEBUG
        printf("string opcode\n");
#endif
        count = get_byte( i++ ) ;
        do
        {
                putchar( get_byte( i++ ) ) ;
wrong! -->      get_word( OUT , get_word( OUT + 1 ) ) ;   put_word(OUT, get_word(OUT)+1);
                count-- ;
        }
        while( count != 0 ) ;
}


/*
 * Procedure to configure the terminal prier to processing by
 * the forth input/output operations.
 */

void configure_term()
{
        initscr() ;

}

/*
 * Procedure to re-configure the terminal the state prier to
 * envoking the configure_term procedure.
 */

void unconfigure_term()
{
```

```
        endwin() ;
}
```

```
/*
 * Module : dictops.h
 * Author : Steven James
 * Date   : 10th February 1990
 *
 * This module implements the five dictionary constructs required by the
 * forth engine, namely WORD FIND NUMBER.
 *
 * Functions :-
 *      word_op() find_op() number_op()
 */

#define WORD_OP         800
#define FIND_OP         801
#define NUMBER_OP       802

/*
 * Procedure to copy a sequence of bytes, delimited by the value specified
 * on top of the parameter stack, from the current position in the text
 * input buffer to the top of the free dictionay as a counted string.
 * The IN offset within the text input buffer is advanced to point the
 * position directly after the delimiter specified.
 */

void word_op()
{
        unsigned short dp , tib , in , separator , count ;
#ifdef DEBUG
        printf("word opcode\n");
#endif
        dp = get_word( DP ) ;
        tib = get_word( TIB ) ;
        in  = get_word( IN ) ;
        count = 0 ;
        separator = pop_ps() ;
        if( isspace( separator ) )
                while( isspace( get_byte( tib + in ) ) )
                        in++ ;
        while( get_byte( tib + in + count ) != separator &&
               get_byte( tib + in + count ) != NULL )
                count++ ;
        put_byte( dp , count ) ;
        dp++ ;
        while( count != 0 )
        {
                put_byte( dp , get_byte( tib + in ) ) ;
                dp++ ;
                in++ ;
                count-- ;
        }
        put_word( IN , in + 1 ) ;
}

/*
 * Procedure to search the dictionary, starting from the vocabulary
 * specified by a 16 bit quantity on the top of the parameter stack, for
 * the entry that matches the counted string pointed to by the second 16 bit
 * quantity on the parameter stack. The pointer to the counted string is
 * always returned together with a 16 bit truth value of false ( 0 ) if the
 * search fails, or the name filed address of the matching dictionary word,
 * and a 16 bit truth value of true ( -1 ) if the search is successful.
 */

void find_op()
{
        unsigned short  width , count ,  cstring  , lfa , nfa , offset ;
#ifdef DEBUG
        printf("find opcode\n") ;
#endif
        nfa = pop_ps() ;
        cstring = pop_ps() ;
        width = get_word( WIDTH ) ;
        do
        {
                lfa = nfa ;
                offset = cstring ;
                count = get_byte( cstring ) ;
                while( ( get_byte( nfa )  & LENGTH_MASK  ) == count &&
                       !( get_byte( nfa ) & HIDDEN_BIT ) )
```

```
                {
                        if( count > width )
                                count = width ;
                        do
                        {
                                nfa++ ;
                                offset++ ;
                                count-- ;
                        }
                        while( get_byte( nfa ) == get_byte( offset )
                               && count != 0 ) ;
                }
                if( count == 0 && get_byte( nfa ) == get_byte( offset ) )
                        nfa = lfa ;
                else
                        nfa = get_word( lfa + width + 1 ) ;
        }
        while( nfa != 0 && nfa != lfa ) ;
        push_ps( cstring ) ;
        if( nfa == 0 )
                push_ps( FFALSE ) ;
        else
        {
                push_ps( nfa ) ;
                push_ps( FTRUE ) ;
        }
}

/*
 * Procedure to convert a lowercase character into an uppercase character.
 */

char upper( n )
        char n ;
{
        if( islower( n ) )
                return( toupper( n ) ) ;
        else
                return( n ) ;
}

/*
 * Procedure convert the counted string on the top of the free dictionary
 * to a decimal value using the user variable BASE as the current number
 * base. The 16 bit decimal aproximation together with a 16 bit value of
 * 1 is returned if the counted string was a valid 16 bit number, or the
 * 32 bit decimal aproximation together with a 16 bit value of 2 if the
 * counted string was a valid 32 bit number. Otherwise 16 bit value 0 and
 */

void number_op()
{
unsigned short dp , count , base  , number ;
int result , negative , large , error ;
#ifdef DEBUG
        printf("number opcode\n") ;
#endif
        dp = get_word( DP ) ;
        base = get_word( BASE ) ;
        large = FALSE ;
        error = FALSE ;
        negative = FALSE ;
        count = get_byte( dp++ ) ;
        result = 0 ;
        if( get_byte( dp ) == 45 )
        {
                negative = TRUE ;
                dp++ ;
                count-- ;
        }
        do
        {
                number = upper( get_byte( dp ) ) ;
                if( number == 46 )
                {
                        if( large )
                                error = TRUE ;
                        else
```

```
                        large = TRUE ;
                count-- ;
                dp++ ;
        }
        else
        {
                number -= 48 ;
                if( number > 0 )
                        if( number > 9 )
                                if( number > 16 )
                                        number = number - 7 ;
                                else
                                        error = TRUE ;
                if( ( number < base )  && !error )
                {
                        result = result * base + number ;
                        count-- ;
                        dp++ ;
                }
                else
                        error = TRUE ;
        }
    }
    while( ( count != 0 )  && ! error ) ;
    if( negative )
        result = -result ;
    push_ps( ( unsigned short ) result & 0xffff ) ;
    if( large )
    {
        if( result > 0xfff )
                push_ps( ( unsigned short ) ( result >> 16 ) ) ;
        else
        {
                if( negative )
                        push_ps( -1 ) ;
                else
                        push_ps( 0 ) ;
        }
    }
    if( ! error )
        if( large )
                push_ps( 2 ) ;
        else
                push_ps( 1 ) ;
    else
        push_ps( FFALSE ) ;
}
```

```
/*
 * Module : defineops.h
 * Author : Steven James
 * Date   : 19th February 1990
 *
 * This module implements the five defining constructs required by the
 * forth engine, namley <BUILDS DOES> CREATE : ; .
 *
 * Functions :-
 *      builds_op() does_op() create_op() define_op() end_op()
 */

#define BUILDS_OP       900
#define DOES_OP         901
#define CREATE_OP       902
#define DEFINE_OP       903
#define END_OP          904

/*
 * Procedure to push the the current content of the word address register
 * onto the top of th parameter stack as a 16 bit quantity.
 */

void builds_op()
{
#ifdef DEBUG
        printf("builds opcode\n") ;
#endif
        push_ps( wa ) ;
}

/*
 * Procedure to push the contents of the instruction register onto the top
 * of the return stack, set in instruction register to the 16 bit quantity
 * pointed to by the contents of the word address register and push the
 * nincremented contents of the word address register onto the parameter
 * stack as a 16 bit quantity.
 */

void does_op()
{
#ifdef DEBUG
        printf("does opcode\n") ;
#endif
        push_rs( i ) ;
        i = get_word( wa ) ;
        push_ps( wa + 2 ) ;
}

/*
 * Procedure to construct a header ( ie. the name and link fields ) for the
 * the next space delimited string in the text input buffer, and report any
 * attempts to redefine an existing word.
 */

void mkheader()
{
        push_ps( get_word( get_word( CURRENT ) ) ) ;
        push_ps( SPACE ) ;
        word_op() ;
        push_ps( get_word( DP ) ) ;
        push_ps( get_word( get_word( CONTEXT ) ) ) ;
        find_op() ;
        if( pop_ps() == FTRUE )
        {
                printf("'%s' has been redefined.\n" , dp_string() ) ;
                drop_op() ;
        }
        drop_op() ;
        put_word( get_word( CURRENT ) , get_word( DP ) ) ;
        put_word( DP , get_word( DP ) + get_word( WIDTH ) + 1 ) ;
        put_word( get_word( DP ) , pop_ps() ) ;
        put_word( DP , get_word( DP ) + 2 ) ;
}

/*
 * Procedure to construct a word header from the space delimited string in
 * the text input buffer, with the code field address pointing to the
 * body of the word.
```

```
 */

void create_op()
{
#ifdef DEBUG
        printf("create opcode\n") ;
#endif
        mkheader() ;
        put_word( get_word( DP ) , get_word( DP ) + 2 ) ;
        put_word( DP , get_word( DP ) + 2 ) ;
}

/*
 * Procedure to construct a colon definition from the space delimited string
 * in the text input buffer.
 */

void define_op()
{
#ifdef DEBUG
        printf("define opcode\n") ;
#endif
        put_word( CONTEXT , get_word( CURRENT ) ) ;
        mkheader() ;
        put_word( get_word( DP ) , colon_cfa ) ;
        put_word( DP , get_word( DP ) + 2 ) ;
        put_word( STATE , FTRUE ) ;
}

/*
 * Procedure to compile a semi colon and terminate compilation of a colon
 * definition by resetting the STATE user variable.
 */

void end_op()
{
#ifdef DEBUG
        printf("end opcode\n") ;
#endif
        push_ps( semi_cfa ) ;
        put_word( get_word( DP ) , pop_ps() ) ;
        put_word( DP , get_word( DP ) + 2 ) ;
        put_word( STATE , FFALSE ) ;
}
```

```
/*
 * Module : inner.h
 * Author : Steven James
 * Date   : 31th January 1990
 *
 * This module implements the operation of the forth inner interpreter.
 *
 * Functions :-
 *      inner() ;
 */

/*
 * Procedure to implements the operation of the forth inner interpreter.
 * On entry the address specified is placed in the instruction register
 * and a next is executed. The inner intrpreter will continue to operate
 * until a trap or invalid instruction is encounted, at which point the
 * interpreter will hault and return control back to the calling
 * procedure or program.
 */

void inner( addr )
unsigned short addr;
{
        i = addr ;
        next() ;
#ifdef DEBUG
        printf("registers : i %d wa %d ca %d\n",i,wa,ca) ;
#endif
        do
        {
                switch ( get_word( ca ) )
                {
                        case COLON_OP   : colon_op() ;
                                          next() ;
                                          break ;
                        case SEMI_OP    : semi_op() ;
                                          next() ;
                                          break ;
                        case EXEC_OP    : exec_op() ;
                                          run() ;
                                          break ;
                        case TRAP_OP    : trap_op() ;
                                          break ;
                        case EXIT_OP    : exit_op() ;
                                          next() ;
                                          break ;
                        case ABORT_OP   : abort_op() ;
                                          next() ;
                                          break ;
                        case DROP_OP    : drop_op() ;
                                          next() ;
                                          break ;
                        case DUP_OP     : dup_op() ;
                                          next() ;
                                          break ;
                        case SWAP_OP    : swap_op() ;
                                          next() ;
                                          break ;
                        case OVER_OP    : over_op() ;
                                          next() ;
                                          break ;
                        case ROT_OP     : rot_op() ;
                                          next() ;
                                          break ;
                        case PUSHR_OP   : pushr_op() ;
                                          next() ;
                                          break ;
                        case POPR_OP    : popr_op() ;
                                          next() ;
                                          break ;
                        case R_OP       : r_op() ;
                                          next() ;
                                          break ;
                        case J_OP       : j_op() ;
                                          next() ;
                                          break ;
                        case EMPTY_OP   : empty_op() ;
                                          next() ;
                                          break ;
```

```
case STORE_OP    : store_op() ;
                   next() ;
                   break ;
case FETCH_OP    : fetch_op() ;
                   next() ;
                   break ;
case CSTORE_OP   : cstore_op() ;
                   next() ;
                   break ;
case CFETCH_OP   : cfetch_op() ;
                   next() ;
                   break ;
case ADD_OP      : add_op() ;
                   next() ;
                   break ;
case DADD_OP     : dadd_op() ;
                   next() ;
                   break ;
case UMUL_OP     : umul_op() ;
                   next() ;
                   break ;
case UDIV_OP     : udiv_op() ;
                   next() ;
                   break ;
case MINUS_OP    : minus_op() ;
                   next() ;
                   break ;
case DMINUS_OP   : dminus_op() ;
                   next() ;
                   break ;
case AND_OP      : and_op() ;
                   next() ;
                   break ;
case OR_OP       : or_op() ;
                   next() ;
                   break ;
case XOR_OP      : xor_op() ;
                   next() ;
                   break ;
case NOT_OP      : not_op() ;
                   next() ;
                   break ;
case EQUAL_OP    : equal_op() ;
                   next() ;
                   break ;
case GREATER_OP  : greater_op() ;
                   next() ;
                   break ;
case LESS_OP     : less_op() ;
                   next() ;
                   break ;
case LIT_OP      : lit_op() ;
                   next() ;
                   break ;
case CONST_OP    : const_op() ;
                   next() ;
                   break ;
case BRANCH_OP   : branch_op() ;
                   next() ;
                   break ;
case JUMP_OP     : jump_op() ;
                   next() ;
                   break ;
case DO_OP       : do_op() ;
                   next() ;
                   break ;
case LOOP_OP     : loop_op() ;
                   next() ;
                   break ;
case ADDLOOP_OP  : addloop_op() ;
                   next() ;
                   break ;
case EMIT_OP     : emit_op() ;
                   next() ;
                   break ;
case KEY_OP      : key_op() ;
                   next() ;
                   break ;
case INLINE_OP   : inline_op() ;
```

```
                                        next() ;
                                        break ;
                        case FILE_OP    : file_op() ;
                                        next() ;
                                        break ;
                        case STDIN_OP   : stdin_op() ;
                                        next() ;
                                        break ;
                        case STRING_OP  : string_op() ;
                                        next() ;
                                        break ;
                        case WORD_OP    : word_op() ;
                                        next() ;
                                        break ;
                        case FIND_OP    : find_op() ;
                                        next() ;
                                        break ;
                        case NUMBER_OP  : number_op() ;
                                        next() ;
                                        break ;
                        case BUILDS_OP  : builds_op() ;
                                        next() ;
                                        break ;
                        case DOES_OP    : does_op() ;
                                        next() ;
                                        break ;
                        case CREATE_OP  : create_op() ;
                                        next() ;
                                        break ;
                        case DEFINE_OP  : define_op() ;
                                        next() ;
                                        break ;
                        case END_OP     : end_op() ;
                                        next() ;
                                        break ;
                        default         : printf("bad opcode\n") ;
                                        forth_abort() ;
                                        break ;
                }
#ifdef DEBUG
        printf("registers : i %d wa %d ca %d\n",i,wa,ca) ;
#endif
        }
        while ( ! trapped() ) ;
        reset_trap() ;
}
```

```
/*
 * Module : dictionary.h
 * Author : Steven James
 * Date   : 10th February 1990
 *
 * This module implements the dictionary structure required by the forth
 * engine.
 *
 * Functions :-
 *      build_dictionary()
 */

/*
 * Procedure to store a 16 bit quantity at the current dictionary address,
 * and increment the dictionary pointer.
 */

void dpword( value )
unsigned short value ;
{
        put_word( dp , value ) ;
        dp += 2 ;
}

/*
 * Procedure to store an 8 bit quabtity at the current dictionary address,
 * and increment the dictionary pointer.
 */

void dpbyte( value )
unsigned char value ;
{
        put_byte( dp , value ) ;
        dp++ ;
}

/*
 * Procedure to construct a dictionay header for the name specified,
 * and maintain the overall dictionary link structure.
 */

void header( string )
char string[INITIAL_WIDTH] ;
{
        int i = 0 ;
        dpbyte( ( unsigned char ) strlen( string ) ) ;
        do
        {
                if( i < strlen( string ) )
                        dpbyte( string[i] ) ;
                else
                        dpbyte( NULL ) ;
                i++;
        }
        while( i < INITIAL_WIDTH ) ;
        dpword( last ) ;
        last = dp - ( INITIAL_WIDTH  + 3 ) ;

}

/*
 * Procedure to set the IMMEDIATE_BIT on the name filed address of the
 * most reciently defined header.
 */

void immediate()
{
        put_byte( last , get_byte( last ) ^ IMMEDIATE_BIT ) ;
}

/*
 * Procedure to construct the initial forth dictionary.
 */

void build_dictionary()
{
        unsigned exec_cfa , trap_cfa  , word_cfa ;
        dp = INITIAL_DP ;
        last = 0 ;
```

```
colon_cfa = dp ;
dpword( COLON_OP ) ;

semi_cfa = dp ;
dpword( dp + 2 ) ;
dpword( SEMI_OP ) ;

trap_cfa = dp ;
dpword( dp + 2 ) ;
dpword( TRAP_OP ) ;

header( "execute" ) ;
exec_cfa = dp ;
dpword( dp + 2 ) ;
dpword( EXEC_OP ) ;

header( "abort" ) ;
dpword( dp + 2 ) ;
dpword( ABORT_OP ) ;

header( "bye" ) ;
dpword( dp + 2 ) ;
dpword( EXIT_OP ) ;

put_word( INNER , dp ) ;
dpword( exec_cfa ) ;
dpword( trap_cfa ) ;

header( "drop" ) ;
dpword( dp + 2 ) ;
dpword( DROP_OP ) ;

header( "dup" ) ;
dpword( dp + 2 ) ;
dpword( DUP_OP ) ;

header( "swap" ) ;
dpword( dp + 2 ) ;
dpword( SWAP_OP ) ;

header( "over" ) ;
dpword( dp + 2 ) ;
dpword( OVER_OP ) ;

header( "rot" ) ;
dpword( dp + 2 ) ;
dpword( ROT_OP ) ;

header( ">R" ) ;
dpword( dp + 2 ) ;
dpword( PUSHR_OP ) ;

header( "R>" ) ;
dpword( dp + 2 ) ;
dpword( POPR_OP ) ;

header( "R" ) ;
dpword( dp + 2 ) ;
dpword( R_OP ) ;

header( "I" ) ;
dpword( dp + 2 ) ;
dpword( R_OP ) ;

header( "J" ) ;
dpword( dp + 2 ) ;
dpword( J_OP ) ;

header( "empty?" ) ;
dpword( dp + 2 ) ;
dpword( EMPTY_OP ) ;

header( "!" ) ;
dpword( dp + 2 ) ;
dpword( STORE_OP ) ;

header( "@" ) ;
dpword( dp + 2 ) ;
```

```
dpword( FETCH_OP ) ;

header( "C!" ) ;
dpword( dp + 2 ) ;
dpword( CSTORE_OP ) ;

header( "C@" ) ;
dpword( dp + 2 ) ;
dpword( CFETCH_OP ) ;

header( "+" ) ;
dpword( dp + 2 ) ;
dpword( ADD_OP ) ;

header( "d+" ) ;
dpword( dp + 2 ) ;
dpword( DADD_OP ) ;

header( "u*" ) ;
dpword( dp + 2 ) ;
dpword( UMUL_OP ) ;

header( "u/" ) ;
dpword( dp + 2 ) ;
dpword( UDIV_OP ) ;

header( "minus" ) ;
dpword( dp + 2 ) ;
dpword( MINUS_OP ) ;

header( "dminus" ) ;
dpword( dp + 2 ) ;
dpword( DMINUS_OP ) ;

header( "and" ) ;
dpword( dp + 2 ) ;
dpword( AND_OP ) ;

header( "or" ) ;
dpword( dp + 2 ) ;
dpword( OR_OP ) ;

header( "xor" ) ;
dpword( dp + 2 ) ;
dpword( XOR_OP ) ;

header( "not" ) ;
dpword( dp + 2 ) ;
dpword( NOT_OP ) ;

header( "=" ) ;
dpword( dp + 2 ) ;
dpword( EQUAL_OP ) ;

header( ">" ) ;
dpword( dp + 2 ) ;
dpword( GREATER_OP ) ;

header( "<" ) ;
dpword( dp + 2 ) ;
dpword( LESS_OP ) ;

header( "lit" ) ;
lit_cfa = dp ;
dpword( dp + 2 ) ;
dpword( LIT_OP ) ;

header( "0branch" ) ;
dpword( dp + 2 ) ;
dpword( BRANCH_OP ) ;

header( "branch" ) ;
dpword( dp + 2) ;
dpword( JUMP_OP ) ;

header( "(do)" ) ;
dpword( dp + 2 ) ;
dpword( DO_OP ) ;
```

```
        header( "(loop)" ) ;
        dpword( dp + 2 ) ;
        dpword( LOOP_OP ) ;

        header( "(+loop)" ) ;
        dpword( dp + 2 ) ;
        dpword( ADDLOOP_OP ) ;

        header( "(emit)" ) ;
        dpword( dp + 2 ) ;
        dpword( EMIT_OP ) ;

        header( "key" ) ;
        dpword( dp + 2 ) ;
        dpword( KEY_OP ) ;

        header( "inline" ) ;
        dpword( dp + 2 ) ;
        dpword( INLINE_OP ) ;

        header( "load" ) ;
        dpword( dp + 2 ) ;
        dpword( FILE_OP ) ;

        header( "stdin?" ) ;
        dpword( dp + 2 ) ;
        dpword( STDIN_OP ) ;

        header( "(.\")" ) ;
        dpword( dp + 2 ) ;
        dpword( STRING_OP ) ;

        header( "word" ) ;
        word_cfa = dp ;
        dpword( dp + 2 ) ;
        dpword( WORD_OP ) ;

        header( "(find)" ) ;
        dpword( dp + 2 ) ;
        dpword( FIND_OP ) ;

        header( "number" ) ;
        dpword( dp + 2 ) ;
        dpword( NUMBER_OP ) ;

        builds_cfa = dp ;
        dpword( BUILDS_OP ) ;

        header( "create" ) ;
        dpword( dp + 2 ) ;
        dpword( CREATE_OP ) ;

        header( "(:)" ) ;
        dpword( dp + 2 ) ;
        dpword( DEFINE_OP ) ;

        header( "(;)" ) ;
        dpword( dp + 2 ) ;
        dpword( END_OP ) ;
        immediate() ;

        header( "\\" ) ;
        dpword( colon_cfa ) ;
        dpword( lit_cfa ) ;
        dpword( 0 ) ;
        dpword( word_cfa ) ;
        dpword( semi_cfa ) ;
        immediate() ;

        dpword( last ) ;
        put_word( CURRENT , dp ) ;
        put_word( CONTEXT , dp ) ;
        dpword( last ) ;

        put_word( DP , dp ) ;
}
```

```
(:) 0=
        0 = (;)

(:) 0<
        0 < (;)

(:) 0>
        0 > (;)

(:) <>
        = not (;)

(:) -
        minus + (;)

(:) d-
        dminus d+ (;)

(:) 1+
        1 + (;)

(:) 1-
        1 - (;)

(:) 2+
        2 + (;)

(:) 2-
        2 - (;)

(:) +!
        dup @ rot + swap ! (;)

(:) +C!
        dup C@ rot + swap C! (;)

(:) here
        0 @ (;)

(:) allot
        here + 0 ! (;)

(:) last
        12 @ @ (;)

(:) pfa
        14 @ 5 + + (;)

(:) cfa
        2- (;)

(:) lfa
        cfa 2- (;)

(:) nfa
        lfa 14 @ 1+ - (;)

(:) ,
        here ! 2 0 +! (;)

(:) C,
        here C! 1 0 +! (;)

(:) immediate
        last dup C@ 128 or swap C! (;)

(:) [
        0 2 ! (;) immediate

(:) ]
        -1 2 ! (;)

(:) -find
        32 word here last (find) dup not 0branch [ 22 , ]
        drop drop here 12 @ 2- @ @ (find) (;)

(:) '
        -find 0branch [ 14 , ] swap drop pfa cfa branch [ 8 , ] drop 0 (;)
```