WAKEFIELD CYBERSECURITY

Dayton Microcomputer Association
Dynamic Languages SIG
November 14, 2018

Dayton Dynamic BASIC or DDB is a non-trivial parsing example written for this SIG by Marc Abel. Whereas many parsing examples have small grammars (such as a four-function calculator) and return results in CPU registers (meaning we don't have to worry about resource allocation and cleanup), DDB is a complete parser for a BASIC-like programming language inspired by the TRS-80 days (which incidentally are not yet over).

The DDB interpreter is not yet complete, so you cannot enter and run full BASIC programs. That will happen at our December 12 meeting when Marc presents DDB as a programming language (instead of as a parser). But you can enter and execute many BASIC statements (one-liners) that don't involve branching (e.g, no FOR loops yet). DDB is 2,500 lines long now, so it's not appropriate to share on the screen or distribute in print. You can download it all from **http://wakesecure.com/basic.tar.gz** and try it out. But we *can* share some highlights for conversation. First, highlights from our grammer:

```
str_term:
    str_lit
    str_var
    ( str_exp )
    func_returning_str
        ( argument_list )

str_exp:
    str_exp + str_term
    str_term

num_term:
    num_lit
    num_var
    ( num_exp )
    func_returning_num
        ( argument_list )

unary:
    + num_term
    - num_term
    num_term

power:
    power ^ unary
    unary

prod:
    prod * power
    prod / power
    prod MOD power
    prod \ power
    prod

sum:
    sum + prod
    sum - prod
    prod

inequality:
    inequality > sum
    inequality >= sum
    inequality < sum
    inequality <= sum

equality:
    equality = inequality
    equality <> inequality

logic_not:
    NOT equality
    equality

logic_and:
    logic_and AND equality
    logic_and NAND equality
    equality

logic_xor:
    logic_xor XOR logic_and
    logic_and

logic_or:
    logic_or OR logic_xor
    logic_or NOR logic_xor
    logic_xor

logic_eqv:
    logic_eqv EQV logic_or
    logic_or
```

Marc W. Abel
(937) 848-0942
mabel@wakesecure.com

3951 W. Franklin St., Suite 3
Bellbrook, OH 45305
wakesecure.com

```
logic_imp:
    logic_imp IMP logic_eqv
    logic_eqv

num_exp:
    logic_imp

mixed_exp:
    num_exp
    str_exp

mixed_var:
    str_var
    num_var

line_list:
    [line_list ,] line_num

line_range:
    line_num - line_num
    line_num [-]
    - [line_num]

var_list:
    [var_list ,] mixed_var

exp_list:
    [exp_list ,] mixed_exp

print_list:
    print_list ; mixed_exp
    print_list , mixed_exp
    mixed_exp

trivial_st: one of
    NEW     END
    STOP    CONT
    RETURN  CLS

line_range_st:
    LIST [line_range]
    DEL [line_range]

line_num_st:
    GOSUB line_num
    GOTO line_num
    RUN [line_num]
    RESTORE [line_num]

line_list_st:
    ON num_exp GOTO line_list
    ON num_exp GOSUB line_list

rem_st:
    REM flush_input_line
    ' flush_input_line

for_st:
    FOR num_var = num_exp TO
        num_exp [STEP num_exp]

next_st:
    NEXT [num_var]

if_st:
    IF num_var THEN statements
        [ELSE statements]

read_data_st:
    READ var_list
    DATA exp_list

let_st:
    [LET] str_var = str_exp
    [LET] num_var = num_exp

print_st:
    PRINT [print_list]
    ? [print_list]

input_st:
    INPUT str_exp ; var_list
    INPUT var_list

statement: one of
    trivial_st    line_range_st
    line_num_s    line_list_st
    rem_st        for_st
    next_st       read_data_st
    print_st      input_st

statements:
    statements : statement
    statement

command:
    line_num statements
    statements
    line_num

command_line:
    command eol
    eol
```

Implementing our grammar in C is straightforward, except there is so much repetition that several helper functions for common patterns have been written. A consequence of this is that the routines for several grammar elements aren't intuitive as to what they do, because of the helper functions they call a "black boxes." A good example is how we parse sums:

```
int sum(char **ss, lego **result)
{
    char *syms[] = { "+", "-", NULL };
    int enums[] = { wADD, wSUB, 0 };

    return general_left_binary(ss, result, syms, enums, prod,
        "need number after + or -");
}
```

A more readable example of how the parser works handles IF statements:

```
int if_st(char **ss, lego **result)
{
    char *s = *ss;
    lego *a0 = NULL, *a1 = NULL, *a2 = NULL, *res = NULL;

    if (!keyword(&s, "if"))
        return 0;
    if (!num_exp(&s, &a0)) {
        warn("need numeric expression after IF");
        goto N;
    }
    if (!keyword(&s, "then")) {
        warn("need THEN after IF ...");
        goto N;
    }
    if (!statements(&s, &a1)) {
        warn("need statement after THEN");
        goto N;
    }

    if (keyword(&s, "else")) {
        if (!statements(&s, &a2)) {
            warn("need statements after ELSE");
            goto N;
        }
    }

    res = newLego(wIF);
    res -> a[0] = a0, res -> a[1] = a1, res -> a[2] = a2;
    *result = res;
    *ss = s;
    return 1;

N:  byeLego(a0); byeLego(a1); byeLego(a2);
    return 0;
}
```

Closer to the "lexing" function of the parser, we can see more specifics as to how individual characters are managed in C. We close for today with code for parsing numbers and string literals:

```
/*
 *  Two formats are supported for string literals:
 *
 *      [Most strings literals are enclosed in square brackets.]
 *
 *      ]$Here is how you can choose the delim. Start with ].$
 *      ]7Here's another example using the numeral seven.7
 */
int str_lit(char **ss, lego **result)
{
    char *s = *ss, *found;
    int ender = ']';
    lego *res;

    eatBlanks(&s);
    if (*s == ender)
        ender = *++s;
    else if (*s != '[')
        return 0;
    if (!isgraph(ender))
        return 0;
    ifnot (found = strchr(++s, ender))
        return 0;

    res = newLego(wSTRLIT);
    res -> s = copySubstring(s, found);
    res -> lit_delim = ender != ']' ? ender : 0;
    *result = res;
    *ss = 1 + found;
    return 1;
}

/*
 *  Detect numerals. Allows . for fractional and _ for grouping.
 *  There is no scientific notation support.
 *
 *  Examples:  0  .01  555  12  123.456  937_848_0942
 */
int num_lit(char **ss, lego **result)
{
    char *s = *ss;
    double r = 0, scale = 1;
    int dot = 0, dig, any = 0;
    lego *res;

    eatBlanks(&s);
    for (;; ++s) {
        if (*s == '.') {
            ++dot;
            continue;
        }
        if (*s == '_')
            continue;
        ifnot (isdigit(dig = *s))
            break;
        any = 1;
        r = 10 * r + dig - '0';
        if (dot) scale *= 10;
    }

    if (!any || dot > 1)
        return 0;

    res = newLego(wNUMLIT);
    res -> n = r / scale;
    *result = res;
    *ss = s;
    return 1;
}
```