Учебные материалы для 2 курса

О рекурсивном спуске

Предположим, что необходимо провести анализ выражения, записанного в строке, и построить дерево разбора. В выражении допускаются бинарные операции +, *, числа и скобки.

Такие выражения описываются следующей грамматикой:

```
expr0 : expr1 { '+' expr1 } ;
expr1 : expr2 { '*' expr2 } ;
expr2 : '(' expr0 ')' | NUM ;
```

Здесь символы `:`, `|`, `;`, `{`, `}` — специальные символы, используемые для записи грамматики.

Определим константы, перечисляющие возможные виды лексических единиц (лексем):

```
enum
{
    OP_EOF,    // конец строки
    OP_NUM,    // число
    OP_PLUS,   // знак '+'
    OP_STAR,   // знак '*'
    OP_LBR,    // знак '('
    OP_RBR     // знак ')'
};
```

Определим тип узла дерева, который будет использоваться как для передачи информации от лексического анализатора к синтаксическому, так и для построения дерева.

```
typedef struct Tree
{
    int opcode;          // код лексемы (см. выше)
    struct Tree *left;   // левое поддерево
    struct Tree *right;  // правое поддерево
    int num;             // значение числа
} Tree;
```

Для хранения текущего состояния анализатора будем использовать глобальные переменные.

```
static const unsigned char *parse_str; // анализируемая строка
static int parse_pos;                  // текущая позиция в строке
static Tree *parse_token;              // текущая лексема
```

Работа лексического анализатора заключается в создании узла типа `Tree` для лексемы в текущей позиции текста и продвижении текущей позиции вперед по строке.

Фрагмент лексического анализатора приведен ниже:

```
Tree *
get_token()
{
    if (!parse_str[parse_pos]) {
        // достигли конца строки
        return tree_create_token(OP_EOF);
    }

    // проверим, что в текущей позиции находится знак операции
    int opcode = OP_EOF;
    switch (parse_str[parse_pos]) {
    case '+':
```

```
        opcode = OP_PLUS;
        break;
    case '(':
        opcode = OP_LBR;
        break;
    // прочие символы операций...
    }
    if (opcode) {
        parse_pos++;
        return tree_create_token(opcode);
    }
    if (isdigit(parse_str[parse_pos])) {
        // создание лексемы для числа
    }
}
```

Работа синтаксического анализатора заключается в инициализации переменных состояния и запуска рекурсивной функции разбора. После завершения разбора нужно проверить, что вся входная строка прочитана.

```
Tree *
parse_expr(const unsigned char *str)
{
    parse_str = str;                    // задать строку для разбора
    parse_pos = 0;                      // установить позицию в строке
    parse_token = get_token();          // считать первую лексему
    Tree *t = parse_expr0();            // разобрать строку
    if (parse_token->opcode != OP_EOF) { // проверить, что дошли до конца
        tree_free(t);
        t = 0;
    }
    tree_free(parse_token);
    return t;
}
```

Функция, разбирающая выражения на слагаемые, может выглядеть примерно так:

```
Tree *
parse_expr0(void)
{
    Tree *t1 = parse_expr1();
    if (!t1) return 0;
    while (parse_token->opcode == OP_PLUS) {
        Tree *t2 = parse_token;
        parse_token = get_token();
        Tree *t3 = parse_expr1();
        if (!t3) {
            tree_free(t1);
            tree_free(t2);
            return 0;
        }
        t2->left = t1;
        t2->right = t3;
        t1 = t2;
    }
    return t1;
}
```

Функция для анализа одного множителя (то есть либо числа, либо выражения в скобках) может выглядеть следующим образом:

```
Tree *
parse_expr2(void)
{
    if (parse_token->opcode == OP_NUM) {
        Tree *t = parse_token;
        parse_token = get_token();
        return t;
    } else if (parse_token->opcode == OP_LBR) {
        tree_free(parse_token);
        parse_token = get_token();
        Tree *t = parse_expr0();
        if (t == NULL) {
            return NULL;
        }
        if (parse_token->opcode != OP_RBR) {
```

```
            tree_free(t);
            return NULL;
        }
        parse_token = get_token();
        return t;
    } else {
        return NULL;
    }
}
```

Last modified: Friday, 21-Jun-2013 16:47:49 MSK
Alexander Chernov