

Интерпретатор языка Little C

Практическая важность интерпретаторов	2
Спецификации Little C	2
Ограничения Little C	2
Интерпретация структурированного языка	4
Неформальная теория С	4
С выражения	5
Вычисление выражений	6
Анализатор выражений	7
Сведение исходного кода к его компонентам	8
Парсер рекурсивного спуска Little C	18
Интерпретатор Little C	38
Предварительное сканирование программы	39
Функции main() и execute()	42
Функция interpret_block()	45
Обработка локальных переменных	48
Вызов пользовательских функций	50
Присвоение значений переменным	54
Инструкция if	56
Цикл while	57
Цикл do-while	58
Цикл for	59
Библиотечные функции в Little C	62

Практическая важность интерпретаторов

Программы на С компилируются. Основная причина этого в том, что язык С используется для создания программ, пригодных для продажи. Скомпилированный код хорошо подходит для коммерческих программных продуктов, поскольку он защищает конфиденциальность исходного кода, предотвращает изменение исходного кода пользователем и позволяет программам наиболее эффективно использовать компьютер.

Главное преимущество интерпретации: кросс-платформенная переносимость. Типичным примером этого является Java. Java был разработан как интерпретируемый язык, потому что он позволял выполнять одну и ту же программу Java в любой среде, в которой есть интерпретатор Java. Такая возможность полезна, когда вы хотите запускать одну и ту же программу на разных типах компьютеров, например, в сетевой среде, такой как Интернет. Создание Java и успех Интернета вызвали новый интерес к интерпретаторам в целом.

По мнению автора, интерпретаторы легко модифицировать, изменять или улучшать. Это означает, что если вы хотите создавать, экспериментировать и контролировать свой собственный язык, вам будет проще сделать это с помощью интерпретатора, чем с помощью компилятора. Интерпретаторы создают отличные среды для создания прототипов языка, потому что вы можете изменить способ работы языка и быстро увидеть результаты.

Спецификации Little C

Интерпретатор Little C понимает довольно узкое подмножество языка. Однако это конкретное подмножество включает в себя многие из наиболее важных аспектов С.

Возможности интерпретатора Little C:

- Параметризованные функции с локальными переменными
- Рекурсия
- Оператор if
- Циклы do-while, while и for
- Keyword break
- Локальные и глобальные переменные типа int и char
- Функциональные параметры типа int и char
- Целочисленные, символьные и строковые константы
- Оператор return
- Ограниченнное количество функций стандартной библиотеки.
- Операторы: +, -, *, /, %, <, >, <=, >=, ==, !=, унарный – и унарный +
- /* . . . */ и // - стиль комментариев

Ограничения Little C

На грамматику Little C наложено несколько грамматических ограничений. Во-первых, if, while, do и for должны быть блоки кода, окруженные открывающими и закрывающими фигурными скобками. Нельзя использовать один оператор. Например, Little C не будет интерпретировать такой код:

```
for (a = 0; a < 10; a = a + 1)
    for (b = 0; b < 10; b = b + 1)
        for (c = 0; c < 10; c = c + 1)
            puts('hi');

if (...)

if (...) x = 10;
```

Правильная версия того же кода

```
for (a = 0; a < 10; a = a + 1)
{
    for (b = 0; b < 10; b = b + 1)
    {
        for (c = 0; c < 10; c = c + 1)
        {
            puts('hi');
        }
    }
}

if (...)

{
    if (...)

    {
        x = 10;
    }
}
```

Это ограничение облегчает интерпретатору поиск конца кода, который необходимо выполнить.

Другое ограничение заключается в том, что прототипы не поддерживаются. Предполагается, что все функции возвращают целочисленный тип (разрешены возвращаемые типы char, но они конвертируются в int), и проверка типа параметра не выполняется.

Все локальные переменные должны быть объявлены в начале функции, сразу после открывающей фигурной скобки.

Локальные переменные не могут быть объявлены в каком-либо другом блоке. Таким образом, следующая функция не работает:

```
if(1) {
    int i; /* not allowed in Little C */
    /* . . . */
}
```

Здесь объявление `i` в блоке `if` недопустимо для Little C.

```
int myfunc()
{
int i; /* this is valid */
```

Интерпретация структурированного языка

Язык С структурированный, что позволяет использовать подпрограммы с локальными переменными. Он также поддерживает рекурсию. По мнению автора проще написать компилятор для структурированного языка, чем написать для него интерпретатор.

Например, когда компилятор генерирует код для вызова функции, он просто помещает вызывающие аргументы в системный стек и выполняет CALL функцию на машинном языке. Для возврата функция помещает возвращаемое значение в регистр ЦП, очищает стек и выполняет машинный язык RET. Однако, когда интерпретатор должен «вызывать» функцию, он должен остановить то, что он делает, сохранить свое текущее состояние, найти местоположение функции, выполнить функцию, сохранить возвращаемое значение и вернуться в исходную точку, восстановив прошлое окружение.

Неформальная теория С

Начнем с того, что все программы на С состоят из набора одной или нескольких функций, а также глобальных переменных (если они существуют). Функция состоит из имени функции, списка ее параметров и блока кода, связанного с функцией. Блок начинается с `{`, за ним следует один или несколько операторов и заканчивается `}`. В С оператор либо начинается с ключевого слова С, например `if`, либо является выражением.

Подводя итог, мы можем написать следующие преобразования (иногда называемые продукциями):

program	→	collection of functions (plus global variables)
function	→	function-specifier parameter-list code-block
code-block	→	{ statement sequence }
statement	→	keyword, expression, or code-block

Все программы на С начинаются с вызова `main()` и заканчиваются, когда в `main()` встречается либо последний `}`, либо `return` — при условии, что ни `exit()`, ни `abort()` больше нигде не вызывались. Любые другие функции, содержащиеся в программе, должны прямо или косвенно вызываться функцией `main()`; таким образом, чтобы выполнить программу на С, просто начните с начала функции `main()` и остановитесь, когда `main()` завершится.

Это именно то, что делает Little C.

С выражения

С расширяет роль выражений по сравнению со многими другими компьютерными языками. В общих чертах оператор представляет собой либо оператор с ключевым словом С, например, **while** или **switch**, либо выражение. В целях обсуждения давайте классифицируем все операторы, начинающиеся с ключевых слов С, как **keyword statements**. Любой оператор в С, который не является **keyword statement**, по определению является **expression statement**. Таким образом, в С все следующие операторы являются выражениями:

```
count = 100;                                /* line 1 */
sample = i / 22 * (c-10);                    /* line 2 */
printf(''This is an expression.''); /* line 3 */
```

В С знак равенства называется **assignment operator**, и значение, полученное операцией присваивания, равно значению, полученному правой частью выражения. Следовательно, оператор присваивания на самом деле является **assignment expression** в С; поскольку это выражение, оно имеет значение. Вот почему разрешено писать такие выражения, как следующее:

```
```c++
a = b = c = 100;
printf("%d", a=4+5);
```

```

Вычисление выражений

Прежде чем мы сможем разработать код, который будет правильно вычислять выражения С, вам необходимо более формально понять, как определяются выражения. Практически во всех компьютерных языках выражения определяется рекурсивно при помощи продукции. Интерпретатор Little C поддерживает следующие операции: +, -, *, /, %, =, операторы отношения (<, ==, > и т. д.) и круглые скобки. Следовательно, мы можем использовать эти продукционные правила для определения выражений Little C:

```
expression → [assignment] [rvalue]
assignment → lvalue = rvalue
lvalue → variable
rvalue → part [rel-op part]
part → term [+term] [-term]
term → factor [*factor] [/factor] [%factor]
factor → [+ or -] atom
atom → variable, constant, function, or (expression)
```

Нетерминалы lvalue и rvalue относятся к объектам, которые могут встречаться в левой и правой частях оператора присваивания соответственно. Одна вещь, о которой вы должны знать, это то, что приоритет операторов встроен в продукционные правила. Чем выше приоритет, тем дальше в списке будет оператор.

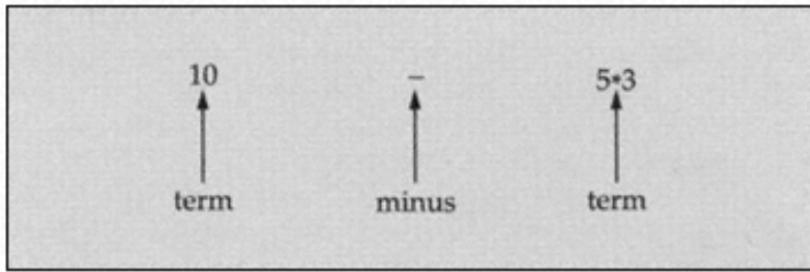
Чтобы увидеть, как работают эти правила, давайте оценим это выражение С:

```
count = 10 - 5 * 3;
```

Во-первых, мы применяем правило 1, которое разбивает выражение на следующие три части:

The diagram shows the expression `count = 10 - 5 * 3;` with three arrows pointing upwards from below to specific parts of the expression. The first arrow points to `count` and is labeled `lvalue`. The second arrow points to the assignment operator `=` and is labeled `assignment operator`. The third arrow points to the entire right-hand side of the assignment, `10 - 5 * 3`, and is labeled `rvalue`.

Поскольку в части «rvalue» подвыражения нет операторов отношения, вызывается продукция:



Второй член состоит из следующих двух множителей: 5 и 3. Эти два множителя являются константами и являются терминалами.

Анализатор выражений

Часть кода, которая читает и анализирует выражения, называется синтаксическим анализатором выражений. Синтаксический анализатор выражений является самой важной подсистемой, необходимой интерпретатору Little C. Поскольку C определяет expressions более выразительно, чем многие другие языки, поэтому большая часть кода, составляющего программу C, фактически выполняется синтаксическим анализатором выражений.

Существует несколько различных способов разработки синтаксического анализатора выражений для C. Многие коммерческие компиляторы используют ***table-driven parser***, который обычно создается при помощи ***parser-generator***. Хотя ***table-driven parsers*** обычно работают быстрее, чем другие методы, их очень сложно создать вручную. Для разработанного здесь интерпретатора Little C мы будем использовать ***recursive-descent parser***, который реализует в логике продукционные правила.

recursive-descent parser — это, по сути, набор взаимно рекурсивных функций, которые обрабатывают выражение. Если синтаксический анализатор используется в компиляторе, он генерирует правильный объектный код, соответствующий исходному коду. Однако в интерпретаторе целью синтаксического анализатора является проверка заданного выражения.

Сведение исходного кода к его компонентам

Фундаментальной для всех интерпретаторов (и компиляторов, если на то пошло) является специальная функция, которая считывает исходный код и возвращает из него следующий логический символ. По историческим причинам эти логические символы

обычно называются токенами. Компьютерные языки в целом и C в частности определяют программы в терминах токенов. Вы можете думать о токене как о неделимой программной единице. Например, оператор равенства == является токеном. Два знака равенства не могут быть разделены без изменения значения. В том же духе, если это токен. Ни «i», ни «f» сами по себе не имеют никакого значения для C.

В C токены определяются одной из представленных ниже групп:

| | | |
|----------|-------------|-------------|
| keywords | identifiers | constants |
| strings | operators | punctuation |

Ключевые слова — это те токены, которые составляют язык C, например, while. Идентификаторы — это имена переменных, функций и пользовательских типов (не реализованы в Little C). Константы и строки говорят сами за себя, как и операторы. Пунктуация включает в себя несколько элементов, таких как точки с запятой, запятые, фигурные скобки и круглые скобки. (Некоторые из них также являются операторами, в зависимости от их использования.) Учитывая выражение

```
for (x=0; x<10; x=x+1) printf("hello %d", x);
```

| Token | Category |
|-------|-------------|
| for | keyword |
| (| punctuation |
| x | identifier |
| = | operator |
| 0 | constant |
| ; | punctuation |
| x | identifier |
| < | operator |
| 10 | constant |
| ; | punctuation |
| x | identifier |

| | |
|------------|-------------|
| = | operator |
| x | identifier |
| + | operator |
| 1 | constant |
|) | punctuation |
| printf | identifier |
| (| punctuation |
| "hello %d" | string |
| , | punctuation |
| x | identifier |
|) | punctuation |
| ; | punctuation |

Однако, чтобы упростить интерпретацию С, Little C классифицирует токены, как показано здесь:

| Token Type | Includes |
|------------|-----------------------------|
| delimiter | punctuation and operators |
| keyword | keywords |
| string | quoted strings |
| identifier | variable and function names |
| number | numeric constant |
| block | { или } |

Функция, которая возвращает токены из исходного кода для интерпретатора Little C, называется `get_next_token()`, и это показано здесь:

```
char get_next_token(void)
{
    char *temp_token;

    token_type = 0;
    current_tok_datatype = 0;

    temp_token = current_token;
    *temp_token = '\0';

    // Пропуск пробелов, символов табуляции и пустой строки
    while (is_whitespace(*source_code_location) && *source_code_location)
        ++source_code_location;

    /* Обработка символов перевода на новую строку для Mac и Windows */
    /* Обработка для мака */
    if (*source_code_location == '\r')
    {
        ++source_code_location;
        /*Обработка для Windows */
        if (*source_code_location == '\n')
        {
            ++source_code_location;
        }
        /* Пропуск пробела */
        while (is_whitespace(*source_code_location) && *source_code_location)
            ++source_code_location;
    }

    /* Unix подобные символы */
    if (*source_code_location == '\n')
    {
        ++source_code_location;
        /* Пропуск перевода на новую строку */
        while (is_whitespace(*source_code_location) && *source_code_location)
            ++source_code_location;
    }
}
```

```
/// Конец файла
if (*source_code_location == '\0')
{
    *current_token = '\0';
    current_tok_datatype = FINISHED;
    return (token_type = DELIMITER);
}

/// Ограничение блока
if (strchr( Str: "{}", Val: *source_code_location))
{ /* разделители блоков */
    *temp_token = *source_code_location;
    temp_token++;
    *temp_token = '\0';
    source_code_location++;
    return (token_type = BLOCK);
}

/// Поиск комментариев
if (*source_code_location == '/')
    if (*(source_code_location + 1) == '*') /* найти конец комментария */
    {
        source_code_location += 2;
        do
        {
            while (*source_code_location != '*' && *source_code_location != '\0')
                source_code_location++;
            if (*source_code_location == '\0')
            {
                source_code_location--;
                break;
            }
            source_code_location++;
        } while (*source_code_location != '/');
        source_code_location++;
    }

/// Поиск комментариев C++ стиля
if (*source_code_location == '/')
    if (*(source_code_location + 1) == '/')
    { /* это комментарий */
        source_code_location += 2;
        /* поиск конца файла */
        while (*source_code_location != '\r' && *source_code_location != '\n' && *source_code_location != '\0')
            source_code_location++;
    }
```

```
        source_code_location++;
    if (*source_code_location == '\r' && *(source_code_location + 1) == '\n')
    {
        source_code_location++;
    }
}

/// Поиск конец файла после комментария
if (*source_code_location == '\0')
{ /* end of file */
    *current_token = '\0';
    current_tok_datatype = FINISHED;
    return (token_type = DELIMITER);
}

/// Операции отношений
if (strchr( str: "!>=", Val: *source_code_location))
{
    switch (*source_code_location)
    {
        case '=':
            if (*(source_code_location + 1) == '=')
            {
                source_code_location++;
                source_code_location++;
                *temp_token = EQUAL;
                temp_token++;
                *temp_token = EQUAL;
                temp_token++;
                *temp_token = '\0';
            }
            break;
        case '!':
            if (*(source_code_location + 1) == '=')
            {
                source_code_location++;
                source_code_location++;
                *temp_token = NOT_EQUAL;
                temp_token++;
                *temp_token = NOT_EQUAL;
                temp_token++;
                *temp_token = '\0';
            }
    }
}
```

```
    temp_token++;
    *temp_token = '\0';
}
break;
case '<':
    if (*(source_code_location + 1) == '=')
    {
        source_code_location++;
        source_code_location++;
        *temp_token = LOWER_OR_EQUAL;
        temp_token++;
        *temp_token = LOWER_OR_EQUAL;
    }
    else
    {
        source_code_location++;
        *temp_token = LOWER;
    }
    temp_token++;
    *temp_token = '\0';
    break;
case '>':
    if (*(source_code_location + 1) == '=')
    {
        source_code_location++;
        source_code_location++;
        *temp_token = GREATER_OR_EQUAL;
        temp_token++;
        *temp_token = GREATER_OR_EQUAL;
    }
    else
    {
        source_code_location++;
        *temp_token = GREATER;
    }
    temp_token++;
    *temp_token = '\0';
    break;
}
if (*current_token)
    return (token_type = DELIMITER);
```

```

/// Разделитель
if (strchr( Str: "+-*^%=-;()", ', ' , Val: *source_code_location))
{
    *temp_token = *source_code_location;
    source_code_location++; /* продвижение на следующую позицию */
    temp_token++;
    *temp_token = '\0';
    return (token_type = DELIMITER);
}

/// Стока в кавычках
if (*source_code_location == '\"')
{
    source_code_location++;
    while ((*source_code_location != '\"' &&
            *source_code_location != '\r' &&
            *source_code_location != '\n' &&
            *source_code_location != '\0') ||
           (*source_code_location == '\"' &&
            *(source_code_location - 1) == '\\'))
        *temp_token++ = *source_code_location++;

    if (*source_code_location == '\r' || *source_code_location == '\n' || *source_code_location == '\0')
        syntax_error( error_type: SYNTAX);
    source_code_location++;
    *temp_token = '\0';
    str_replace( line: current_token, search: "\\\\"a", replace: "\\a");
    str_replace( line: current_token, search: "\\\\"b", replace: "\\b");
    str_replace( line: current_token, search: "\\\\"f", replace: "\\f");
    str_replace( line: current_token, search: "\\\\"n", replace: "\\n");
    str_replace( line: current_token, search: "\\\\"r", replace: "\\r");
    str_replace( line: current_token, search: "\\\\"t", replace: "\\t");
    str_replace( line: current_token, search: "\\\\"v", replace: "\\v");
    str_replace( line: current_token, search: "\\\\"\\", replace: "\\\"");
    str_replace( line: current_token, search: "\\\\"'", replace: "\\\"");
    str_replace( line: current_token, search: "\\\\"\"", replace: "\\\"");
    return (token_type = STRING);
}

```

```

/// Число
if (isdigit( c: (int)*source_code_location))
{
    while (!is_delimiter( c: *source_code_location))
        *temp_token++ = *source_code_location++;
    *temp_token = '\0';
    return (token_type = NUMBER);
}

/// Переменная или оператор
if (isalpha( c: (int)*source_code_location))
{
    while (!is_delimiter( c: *source_code_location))
        *temp_token++ = *source_code_location++;
    token_type = TEMP;
}

*temp_token = '\0';
/// Эта строка является оператором или переменной?
if (token_type == TEMP)
{
    current_tok_datatype = look_up_token_in_table( s: current_token); /* преобразовать во внутреннее представление */
    if (current_tok_datatype) /* это зарезервированное слово */
        token_type = KEYWORD;
    else
        token_type = VARIABLE;
}

return token_type;

```

Функция `get_next_token()` использует следующие глобальные константы и типы:

```
+enum token_types
{
    DELIMITER, //Разделители по типу +-=? и т.д.
    VARIABLE, //Переменная
    NUMBER, //Число
    KEYWORD, //Ключевое слово
    TEMP, //Буфер
    STRING, //Строка
    BLOCK //Блокировка интерпретатора. Обычно нужна когда встретилась ошибка или закончили читать функцию
};

/** 
 * @brief Ключевые слова
 */
enum tokens
{
    ARG, //аргумент
    CHAR, //Вещественный тип данных
    INT, //Целочисленный тип данных
    IF, //Оператор if
    ELSE, //Оператор else
    FOR, //Оператор for
    DO, //Оператор do
    WHILE, //Оператор while
    SWITCH, //Оператор switch
    RETURN, //Оператор return
    CONTINUE, //Оператор continue
    BREAK, //Оператор break
    EOL, //Конец строки
    FINISHED,
    END
};
```

```

enum double_ops
{
    LOWER = 1, //Меньше, меньше или равно
    LOWER_OR_EQUAL,
    GREATER, //Больше, больше или равно
    GREATER_OR_EQUAL,
    EQUAL, //Эквивалентно или не эквивалентно
    NOT_EQUAL
};

/** 
 * @brief Виды ошибок
 */
enum error_msg
{
    SYNTAX, //синтаксическая ошибка
    UNBAL_PARENS, //Скобок слишком много или не хватает
    NO_EXP, //Ожидалось выражение
    EQUALS_EXPECTED, //Ожидалось приравнивание
    NOT_VAR, //не переменная
    PARAM_ERR, //неправильно задан параметр
    SEMICOLON_EXPECTED, //ожидалась ;
    UNBAL_BRACES, //операторных скобок много или не хватает
    FUNC_UNDEFINED, //функция определена неправильно
    TYPE_EXPECTED, //тип данных не указан
    NESTED_FUNCTIONS, //функция определена внутри другой функции
    RET_NOCALL, //функция ничего не возвращает
    PAREN_EXPECTED, //ожидалась скобка
    WHILE_EXPECTED, //ожидался while
    QUOTE_EXPECTED, //неправильно написан комментарий
    NOT_TEMP, //не строка
    TOO_MANY_LVARS, //слишком много локальных переменных
    DIV_BY_ZERO //на ноль делить нельзя
};

```

На текущее место в исходном коде указывает `source_code_location`. Указатель `program_start_buffer` не меняется интерпретатором и всегда указывает на начало интерпретируемой программы. Функция `get_next_token()` начинается с пропуска всех пробелов, включая возврат каретки и перевод строки. Так как в С нет токена, который указывает на пробел, пробелы должны быть игнорироваться.. Функция `get_next_token()` также пропускает комментарии. Затем строковое представление каждого токена помещается в токен; его тип (как определено перечисление `tok_types`) помещается в `token_type`; и, если токен является ключевым словом, его внутреннее представление назначается `tok` с помощью функции `look_up_token_in_table()`. Функция `get_next_token()` преобразует двухсимвольные реляционные операторы С в их соответствующее значение перечисления. Хотя это и не является технически необходимым, этот шаг делает синтаксический анализатор легче реализовать. Наконец, если

синтаксический анализатор обнаруживает синтаксическую ошибку, он вызывает функцию **syntax_error()**. с перечисляемым значением, которое соответствует типу найденной ошибки. Функция **syntax_error()** также вызывается другими подпрограммами в интерпретаторе всякий раз, когда возникает ошибка. Функция **syntax_error()** показана здесь:

```
void syntax_error(int error_type)
{
    char *program_pointer_location, *temp;
    int line_count = 0;
    int i;

    string errors_human_readable[]
    {
        [0]: "Синтаксическая ошибка",
        [1]: "Слишком много или мало скобок",
        [2]: "Нет выражения",
        [3]: "Не хватает знаков равно",
        [4]: "Не является переменной",
        [5]: "Неправильно задан параметр функции",
        [6]: "Не хватает точки с запятой",
        [7]: "Слишком много или мало операторных скобок",
        [8]: "Функция не определена",
        [9]: "Тип данных не указан",
        [10]: "Слишком много обращений к вложенным функциям",
        [11]: "Функция возвращает значения без обращения к ней",
        [12]: "Не хватает скобки",
        [13]: "Нет оператора для цикла while",
        [14]: "Не хватает закрывающих кавычек",
        [15]: "Не является строкой",
        [16]: "Слишком много локальных переменных",
        [17]: "На ноль делить НЕЛЬЗЯ"
    };

    cout << "\n"
        << errors_human_readable[error_type];

    program_pointer_location = program_start_buffer;
```

```

/// Поиск позиции ошибки
while (*program_pointer_location != source_code_location && *program_pointer_location != '\0')
{
    program_pointer_location++;
    if (*program_pointer_location == '\r')
    {
        line_count++;
        if (program_pointer_location == source_code_location)
        {
            break;
        }

        program_pointer_location++;

        if (*program_pointer_location != '\n')
        {
            program_pointer_location--;
        }
    }
    else if (*program_pointer_location == '\n')
    {
        line_count++;
    }
    else if (*program_pointer_location == '\0')
    {
        line_count++;
    }
}
cout << " на строке " << line_count << endl;

/// Поиск и указание последнего рабочего кусочка кода
temp = program_pointer_location--;
for (i = 0; i < 20 && program_pointer_location > program_start_buffer && *program_pointer_location != '\n' && *program_pointer_location != '\r'; i++, program_pointer_location--)
;
for (i = 0; i < 30 && program_pointer_location <= temp; i++, program_pointer_location++)
    printf("%c", *program_pointer_location);

longjmp(Buf, execution_buffer, Value: 1);
}

```

Обратите внимание, что **syntax_error()** также отображает номер строки, в которой была обнаружена ошибка и отображает строку, в которой она произошла. В дальнейшем работа **syntax_error()** заканчивается вызовом **longjmp()**. Поскольку синтаксические ошибки часто встречаются в глубоко вложенных или рекурсивных подпрограммах, самый простой способ справиться с ошибкой — просто перейти в безопасную область памяти. С другой стороны можно установить глобальный флаг ошибки и опрашивать этот флаг в различных точках каждой подпрограммы, однако, это увеличивает сложность программы и замедляет её.

Парсер рекурсивного спуска Little C

Здесь показан весь код анализатора рекурсивного спуска Little C вместе с некоторыми необходимыми вспомогательными функциями, глобальными данными и типами данных. Этот код, по задумке, должен находиться в отдельном файле. Назовем его `parser.cpp`.

```
/***
 * Анализатор рекурсивного спуска для целочисленных выражений, который может включать переменные и вызовы функций.
 */
#include <setjmp.h> //либа для создания переходов а-ля ассемблер
#include <ctype.h>
#include <stdlib.h>
#include <cstring>
#include <stdio.h>
#include <iostream>

#include "enum.h"
#include "const.h"
using namespace std;
/// Совместимость с защищенными функциями
#if !defined(_MSC_VER) || _MSC_VER < 1400
#define strcpy_s(dest, count, source) strncpy((dest), (source), (count))
#endif
/***
 * @brief Структура переменных
 */
struct variable_type
{
    char variable_name[ID_LEN];
    int variable_type;
    int variable_value;
};

/***
 * @brief Структура функции
 */
struct function_type
{
    char func_name[ID_LEN];
    int ret_type;
    char *loc;
};

/***
 * @brief Структура списка зарезервированных слов
 */
struct commands
{
    char command[20];
    char tok;
};

/// Структура списка функций стандартной библиотеки
struct intern_func_type
{
    char *f_name; /* имя функции */
    int (*p)(void); /* указатель на функцию */
};

extern variable_type global_vars[NUM_GLOBAL_VARS];
extern function_type func_stack[NUMBER_FUNCTIONS];
--
```

```

extern commands table_with_statements[];

// Здесь функции "стандартной библиотеки" объявлены таким образом, что их можно поместить во внутреннюю таблицу функций
int call_getche(void);
int call_putchar(void);
int call_puts(void);
int print(void);
int getnum(void);

/// Массив стандартных функций
intern_func_type intern_func[] = {
    [0] = { .f_name: "getche", .p: call_getche},
    [1] = { .f_name: "putchar", .p: call_putchar},
    [2] = { .f_name: "puts", .p: call_puts},
    [3] = { .f_name: "print", .p: print},
    [4] = { .f_name: "getnum", .p: getnum},
    [5] = { .f_name: "", .p: 0} /* этот список заканчивается нулем */
};

extern char *source_code_location; /* текущее положение в исходном тексте программы */
extern char *program_start_buffer; /* указатель на начало буфера программы */
extern jmp_buf execution_buffer; /* содержит данные для longjmp() */
extern char current_token[80]; /* строковое представление current_token */
extern char token_type; /* содержит тип current_token */
extern char current_tok_datatype; /* внутреннее представление current_token */
extern int ret_value; /* возвращаемое значение функции */

/// Функции для анализатора
void eval_expression(int *value);
void eval_assignment_expression(int *value);
void eval_comparison_expression(int *value); /* обработка операторов отношений */
void eval_sum_minus_expression(int *value); /* обработка сложения или вычитания */
void eval_multiply_division_expression(int *value); /* обработка умножения, деления, целочисленного деления */
void eval_unar_minus_expression(int *value); /* унарные плюс и минус */
void eval_exp_in_parenthesis_expression(int *value); /* обработка выражений в скобках */
void atom(int *value); /* найти значение числа, переменной или функции */
char get_next_token(void);
void shift_source_code_location_back(void);
char look_up_token_in_table(char *s);
int internal_func(char *s);
int is_delimiter(char c);
int is_whitespace(char c);
static void str_replace(char *line, const char *search, const char *replace);

#if defined(_MSC_VER) && _MSC_VER >= 1200
...
#elif __GNUC__
void syntax_error(int error) __attribute__((noreturn));
#else
void syntax_error(int error);
#endif

```

```

/// Функции из интерпретатора
void assign_var(char *var_name, int value);
int find_var(char *s);
int is_variable(char *s);
char *find_function_in_function_table(char *name);
void call_function(void);

/***
 * @brief Точка входа в синтаксический анализатор выражений
 */
void eval_expression(int *value)
{
    get_next_token();
    if (!*current_token)
    {
        syntax_error(error: NO_EXP);
        return;
    }
    if (*current_token == ';')
    {
        *value = 0; /* пустое выражение */
        return;
    }
    eval_assignment_expression(value);
    shift_source_code_location_back(); /* возврат последней лексемы во входной поток */
}

/***
 * @brief Обработка выражения в присваивании
 * @param value
 */
void eval_assignment_expression(int *value)
{
    char temp[ID_LEN]; /* Содержит имя переменной, которой присваивается значение */
    char temp_tok;

    if (token_type == VARIABLE)
    {
        /// Если встретили переменную, то проверяем, присваивается ли ей какое-либо значение */
        if (is_variable(s: current_token))
        {
            strcpy_s(Destination: temp, SizelnBytes: ID_LEN, Source: current_token);
            temp_tok = token_type;
            get_next_token();
            if (*current_token == '=') /* Если присваивается */
            {
                get_next_token();
                eval_assignment_expression(value); /* то смотрим, что надо присвоить */
                assign_var(var_name: temp, value: *value); /* присваиваем */
                return;
            }
            else
        }
    }
}

```

```
        /* Если не присваивается */
        shift_source_code_location_back(); /* То забываем про temp и копируем изначальное значение токена из temp_tok */
        strcpy_s( Destination: current_token, SizelnBytes: 80, Source: temp);
        token_type = temp_tok;
    }
}

eval_comparison_expression(value);

/***
 * @brief Обработка операций сравнения.
 * @param value
 */
void eval_comparison_expression(int *value)
{
    int partial_value;
    char op;
    char relational_operators[7] = {
        [0]: LOWER,
        [1]: LOWER_OR_EQUAL,
        [2]: GREATER,
        [3]: GREATER_OR_EQUAL,
        [4]: EQUAL,
        [5]: NOT_EQUAL,
        [6]: 0};

    eval_sum_minus_expression(value);
    op = *current_token;
    if (strchr( String: relational_operators, Ch: op))
    {
        get_next_token();
        eval_sum_minus_expression( value: &partial_value);
        switch (op)
        { /* perform the relational operation */
        case LOWER:
            *value = *value < partial_value;
            break;
        case LOWER_OR_EQUAL:
            *value = *value <= partial_value;
            break;
        case GREATER:
            *value = *value > partial_value;
            break;
        case GREATER_OR_EQUAL:
            *value = *value >= partial_value;
            break;
        case EQUAL:
            *value = *value == partial_value;
            break;
        case NOT_EQUAL:
            *value = *value != partial_value;
        }
    }
}
```

```
        case '*':
            *value = *value * partial_value;
            break;
        case '/':
            if (partial_value == 0)
                syntax_error(error: DIV_BY_ZERO);
            *value = (*value) / partial_value;
            break;
        case '%':
            t = (*value) / partial_value;
            *value = *value - (t * partial_value);
            break;
    }
}
}

/***
 * @brief Унарный + или -
 * @param value
 */
void eval_unar_minus_expression(int *value)
{
    char op;

    op = '\0';
    if (*current_token == '+' || *current_token == '-')
    {
        op = *current_token;
        get_next_token();
    }
    eval_exp_in_parenthesis_expression(value);
    if (op)
        if (op == '-')
            *value = -(*value);
}

/***
 * @brief Обработка выражения в скобках
 * @param value
 */
void eval_exp_in_parenthesis_expression(int *value)
{
    if (*current_token == '(')
    {
        get_next_token();
        eval_assignment_expression(value); // get subexpression :(
    }
}
```

```
    eval_assignment_expression(value); /* get subexpression */
    if (*current_token != ')')
        syntax_error( error: PAREN_EXPECTED );
    get_next_token();
}
else
    atom(value);
}
/***
 * @brief Получение значения числа, переменной или функции
 * @param value
 */
void atom(int *value)
{
    int i;

    switch (token_type)
    {
    case VARIABLE:
        i = internal_func( s: current_token );
        if (i != -1)
        { /* вызов стандартной функции */
            *value = (*intern_func[i].p)();
        }
        else if (find_function_in_function_table( name: current_token ))
        { /* вызов пользовательской функции */
            call_function();
            *value = ret_value;
        }
        else
            *value = find_var( s: current_token ); /* получаем значение переменной */
        get_next_token();
        return;
    case NUMBER: /* числовая константа */
        *value = atoi( String: current_token );
        get_next_token();
        return;
    case DELIMITER: /* символьная константа */
        if (*current_token == '\\')
        {
            *value = *source_code_location;
            source_code_location++;
            if (*source_code_location != '\\')
                syntax_error( error: QUOTE_EXPECTED );
            source_code_location++;
            get_next_token();
            return;
        }
        if (*current_token == ')')
            return; /* пустое выражение в скобках */
        else

```

```
    |     syntax_error( error: SYNTAX); /* синтаксическая ошибка */
    | default:
    |     syntax_error( error: SYNTAX); /* синтаксическая ошибка */
    |
    }}
/***
 * @brief Печать ошибки
 *
 * Выводит сообщение об ошибке, основываясь на полученном типе ошибки
 * @param error_type
 */
void syntax_error(int error_type)
{
    char *program_pointer_location, *temp;
    int line_count = 0;
    int i;

    string errors_human_readable[]
    {
        [0]: "Синтаксическая ошибка",
        [1]: "Слишком много или мало скобок",
        [2]: "Нет выражения",
        [3]: "Не хватает знаков равно",
        [4]: "Не является переменной",
        [5]: "Неправильно задан параметр функции",
        [6]: "Не хватает точки с запятой",
        [7]: "Слишком много или мало операторных скобок",
        [8]: "Функция не определена",
        [9]: "Тип данных не указан",
        [10]: "Слишком много обращений к вложенным функциям",
        [11]: "Функция возвращает значения без обращения к ней",
        [12]: "Не хватает скобки",
        [13]: "Нет оператора для цикла while",
        [14]: "Не хватает закрывающих кавычек",
        [15]: "Не является строкой",
        [16]: "Слишком много локальных переменных",
        [17]: "На ноль делить НЕЛЬЗЯ"
    };

    cout << "\n"
        << errors_human_readable[error_type];

    program_pointer_location = program_start_buffer;

    /// Поиск позиции ошибки
    while (program_pointer_location != source_code_location && *program_pointer_location != '\0')
    {
        program_pointer_location++;
        if (*program_pointer_location == '\r')
        {
            line_count++;
        }
    }
}
```

```
        if (program_pointer_location == source_code_location)
        {
            break;
        }

        program_pointer_location++;

        if (*program_pointer_location != '\n')
        {
            program_pointer_location--;
        }
    }
    else if (*program_pointer_location == '\n')
    {
        line_count++;
    }
    else if (*program_pointer_location == '\0')
    {
        line_count++;
    }
}

cout << " на строке " << line_count << endl;

/// Поиск и указание последнего рабочего кусочка кода
temp = program_pointer_location--;
for (i = 0; i < 20 && program_pointer_location > program_start_buffer && *program_pointer_location != '\n' && *program_pointer_location != '\r'; i++, program_pointer_location--)
{
    for (i = 0; i < 30 && program_pointer_location <= temp; i++, program_pointer_location++)
        printf("%c", *program_pointer_location);

    longjmp(buf, execution_buffer, value: 1);
}
/**/
/* @brief Считывание лексемы из входного потока
 * @return
 */
char get_next_token(void)
{
    char *temp_token;

    token_type = 0;
    current_tok_datatype = 0;

    temp_token = current_token;
    *temp_token = '\0';

    /// Пропуск пробелов, символов табуляции и пустой строки
    while (is whitespace(*source_code_location) && *source_code_location)
        ++source_code_location;

    /* Обработка символов перевода на новую строку для Mac и Windows */
}
```

```

/* Обработка для мака */
if (*source_code_location == '\r')
{
    ++source_code_location;
    /*Обработка для Windows */
    if (*source_code_location == '\n')
    {
        ++source_code_location;
    }
    /* Пропуск пробела */
    while (is_whitespace(*source_code_location) && *source_code_location)
        ++source_code_location;
}

/* Unix подобные символы */
if (*source_code_location == '\n')
{
    ++source_code_location;
    /* Пропуск перевода на новую строку */
    while (is_whitespace(*source_code_location) && *source_code_location)
        ++source_code_location;
}

/// Конец файла
if (*source_code_location == '\0')
{
    *current_token = '\0';
    current_tok_datatype = FINISHED;
    return (token_type = DELIMITER);
}

/// Ограничение блока
if (strchr(str: "{}", Val: *source_code_location))
{ /* разделители блоков */
    *temp_token = *source_code_location;
    temp_token++;
    *temp_token = '\0';
    source_code_location++;
    return (token_type = BLOCK);
}

/// Поиск комментариев
if (*source_code_location == '/')
    if (*(source_code_location + 1) == '*') /* найти конец комментария */
    {
        source_code_location += 2;
        do
        {
            while (*source_code_location != '*' && *source_code_location != '\0')
                source_code_location++;
            if (*source_code_location == '\0')
            {
                source_code_location--;
            }
        }
    }
}

```

```
        break;
    }
    source_code_location++;
} while (*source_code_location != '/');
source_code_location++;

}

/// Поиск комментариев C++ стиля
if (*source_code_location == '/')
{
    if (*(source_code_location + 1) == '/')
    { /* это комментарий */
        source_code_location += 2;
        /* поиск конца файла */
        while (*source_code_location != '\r' && *source_code_location != '\n' && *source_code_location != '\0')
            source_code_location++;
        if (*source_code_location == '\r' && *(source_code_location + 1) == '\n')
        {
            source_code_location++;
        }
    }
}

/// Поиск конец файла после комментария
if (*source_code_location == '\0')
{ /* end of file */
    *current_token = '\0';
    current_tok_datatype = FINISHED;
    return (token_type = DELIMITER);
}

/// Операции отношений
if (strchr( Str: "!<=>", Val: *source_code_location))
{
    switch (*source_code_location)
    {
        case '=':
            if (*(source_code_location + 1) == '=')
            {
                source_code_location++;
                source_code_location++;
                *temp_token = EQUAL;
                temp_token++;
                *temp_token = EQUAL;
                temp_token++;
                *temp_token = '\0';
            }
            break;
        case '!':
            if (*(source_code_location + 1) == '=')
            {
                source_code_location++;
                source_code_location++;
                *temp_token = NOT_EQUAL;
                temp_token++;
                *temp_token = NOT_EQUAL;
            }
    }
}
```

```
    temp_token++;
    *temp_token = '\0';
}
break;
case '<':
if (*(source_code_location + 1) == '=')
{
    source_code_location++;
    source_code_location++;
    *temp_token = LOWER_OR_EQUAL;
    temp_token++;
    *temp_token = LOWER_OR_EQUAL;
}
else
{
    source_code_location++;
    *temp_token = LOWER;
}
temp_token++;
*temp_token = '\0';
break;
case '>':
if (*(source_code_location + 1) == '=')
{
    source_code_location++;
    source_code_location++;
    *temp_token = GREATER_OR_EQUAL;
    temp_token++;
    *temp_token = GREATER_OR_EQUAL;
}
else
{
    source_code_location++;
    *temp_token = GREATER;
}
temp_token++;
*temp_token = '\0';
break;
}
if (*current_token)
    return (token_type = DELIMITER);
}
/// Разделитель
if (strchr( Str: "+-*^/%=;()", '", Val: *source_code_location))
{
    *temp_token = *source_code_location;
    source_code_location++; /* продвижение на следующую позицию */
    temp_token++;
    *temp_token = '\0';
    return (token_type = DELIMITER);
}
```

```

/// Стока в кавычках
if (*source_code_location == '\"')
{
    source_code_location++;
    while ((*source_code_location != '\"' &&
           *source_code_location != '\n' &&
           *source_code_location != '\r' &&
           *source_code_location != '\0') ||
           (*source_code_location == '\"' &&
            *(source_code_location - 1) == '\\'))
        *temp_token++ = *source_code_location++;

    if (*source_code_location == '\r' || *source_code_location == '\n' || *source_code_location == '\0')
        syntax_error(error_type: SYNTAX);
    source_code_location++;
    *temp_token = '\0';
    str_replace(line: current_token, search: "\\\\"a", replace: "\\a");
    str_replace(line: current_token, search: "\\\\"b", replace: "\\b");
    str_replace(line: current_token, search: "\\\\"f", replace: "\\f");
    str_replace(line: current_token, search: "\\\\"n", replace: "\\n");
    str_replace(line: current_token, search: "\\\\"r", replace: "\\r");
    str_replace(line: current_token, search: "\\\\"t", replace: "\\t");
    str_replace(line: current_token, search: "\\\\"v", replace: "\\v");
    str_replace(line: current_token, search: "\\\\"\\\", replace: "\\\"");
    str_replace(line: current_token, search: "\\\\"\\\", replace: "\\\"");
    str_replace(line: current_token, search: "\\\\"\\\", replace: "\\\"");
    return (token_type = STRING);
}

/// Число
if (isdigit(c: (int)*source_code_location))
{
    while (!is_delimiter(c: *source_code_location))
        *temp_token++ = *source_code_location++;
    *temp_token = '\0';
    return (token_type = NUMBER);
}

/// Переменная или оператор
if (isalpha(c: (int)*source_code_location))
{
    while (!is_delimiter(c: *source_code_location))
        *temp_token++ = *source_code_location++;
    token_type = TEMP;
}

*temp_token = '\0';
/// Эта строка является оператором или переменной?
if (token_type == TEMP)
{
    current_tok_datatype = look_up_token_in_table(s: current_token); /* преобразовать во внутреннее представление */
    if (current_tok_datatype) /* это зарезервированное слово */
        token_type = KEYWORD;
}

```

```

        else
            token_type = VARIABLE;
    }
    return token_type;
}
/***
 * @brief Возврат лексемы во входной поток.
 *
 * Передвигаем указатель на текущую программу на *_токен_* обратно
 */
void shift_source_code_location_back(void)
{
    char *t;

    t = current_token;
    for (; *t; t++)
        source_code_location--;
}
/***
 * @brief Поиск внутреннего представления лексемы в таблице лексем
 * @param token_string
 * @return
 */
char look_up_token_in_table(char *token_string)
{
    int i;
    char *pointer_to_token_string;

    /* переводим токен в нижний регистр */
    pointer_to_token_string = token_string;
    while (*pointer_to_token_string)
    {
        *pointer_to_token_string = (char)tolower(*pointer_to_token_string);
        pointer_to_token_string++;
    }

    /* проверяем есть ли данный токен в таблице специальных зарезервированных слов. */
    for (i = 0; *table_with_statements[i].command; i++)
    {
        if (!strcmp(table_with_statements[i].command, token_string))
            return table_with_statements[i].tok;
    }
    return 0; /* unknown command */
}
/***
 * @brief Возвращает индекс функции во внутренней библиотеке, или -1, если не найдена.
 * @param s
 * @return
 */
int internal_func(char *s)
{

```

```
int i;

for (i = 0; intern_func[i].f_name[0]; i++)
{
    if (!strcmp(intern_func[i].f_name, s))
        return i;
}
return -1;
}

/***
 * @brief Разделитель
 *
 * Проверяет является ли символ разделителем
 *
 * @param Символ
 *
 * @return 1 - разделитель
 * 0 - не разделитель
 */
int is_delimiter(char c)
{
    if (strchr(" !,->/*%^=()\", \r\n\0", c) || c == 9 || c == '\r' || c == '\n' || c == 0)
        return 1;
    return 0;
}

/***
 * @brief Пробел/таб
 * @param c
 * @return Возвращает 1, если c - пробел или табуляция
 */
int is_whitespace(char c)
{
    if (c == ' ' || c == '\t')
        return 1;
    else
        return 0;
}

/***
 * Модификация на месте находит и заменяет строку.
 * Предполагается, что буфер, на который указывает строка, достаточно велик для хранения результирующей строки
 * @param line
 * @param search
 * @param replace
 */
static void str_replace(char *line, const char *search, const char *replace)
{
    char *sp;
    while ((sp = strstr(line, search)) != NULL)
    {
        int search_len = (int)strlen(search);

        int replace_len = (int)strlen(replace);
        int tail_len = (int)strlen(sp + search_len);

        memmove(sp + replace_len, sp + search_len, tail_len + 1);
        memcpy(sp, replace, replace_len);
    }
}
```

Функции, начинающиеся с **eval_expression**, и функция **atom()** реализуют продукции для выражений Little C. Чтобы убедиться в этом, можно мысленно запустить синтаксический анализатор, используя простое выражение. Функция **atom()** находит значение целочисленной константы или переменной, функции или символьной константы. В исходном коде могут присутствовать два вида функций: определяемые пользователем или библиотечные. Если встречается определяемая пользователем функция, ее код выполняется интерпретатором, чтобы определить ее возвращаемое значение. Однако, если функция является библиотечной, сначала ее адрес ищется функцией **internal_func()**, а затем доступ к нему осуществляется через функцию интерфейса. Библиотечные функции и адреса их функций интерфейса хранятся в массиве **internal_func**, показанном здесь:

```
/***
 * @brief Структура функции
 */
struct function_type
{
    char func_name[ID_LEN];
    int ret_type;
    char *loc;
};
```

```
int call_getche(void)
{
    char ch;
#if defined(_QC)
    ch = (char)getche();
#elif defined(_MSC_VER)
    ch = (char)_getche();
#else
    ch = (char)getchar();
#endif
    while (*source_code_location != ')')
        source_code_location++;
    // Продолжаем работать, пока не достигнем конца строки
    source_code_location++;
    return ch;
}
```

```
int call_putchar(void)
{
    int value;

    eval_expression( result: &value);
    printf( Format: "%c", value);
    return value;
}
```

```
int call_puts(void)
{
    get_next_token();
    if (*current_token != '(')
        syntax_error(error: PAREN_EXPECTED);
    get_next_token();
    if (token_type != STRING)
        syntax_error(error: QUOTE_EXPECTED);
    puts(Buffer: current_token);
    get_next_token();
    if (*current_token != ')')
        syntax_error(error: PAREN_EXPECTED);

    get_next_token();
    if (*current_token != ';')
        syntax_error(error: SEMICOLON_EXPECTED);
    shift_source_code_location_back();
    return 0;
}
```

```
int print(void)
{
    int i;

    get_next_token();
    if (*current_token != '(')
        syntax_error(error: PAREN_EXPECTED);

    get_next_token();
    if (token_type == STRING)
    { /* выводим строку */
        printf(Format: "%s ", current_token);
    }
    else
    { /* выводим число */
        shift_source_code_location_back();
        eval_expression(result: &i);
        printf(Format: "%d ", i);
    }

    get_next_token();

    if (*current_token != ')')
        syntax_error(error: PAREN_EXPECTED);

    get_next_token();
    if (*current_token != ';')
        syntax_error(error: SEMICOLON_EXPECTED);
    shift_source_code_location_back();
```

```
int getnum(void)
{
    char s[80];

    if (fgets( Buffer: s, MaxCount: sizeof(s), Stream: stdin) != NULL)
    {
        while (*source_code_location != '\n')
            source_code_location++;
        source_code_location++; /* читаем до конца строки */
        return atoi( String: s);
    }
    else
    {
        return 0;
    }
}
```

И последнее замечание о подпрограммах в файле синтаксического анализатора выражений: для правильного синтаксического анализа языка С иногда требуется то, что называется **one-token lookahead**. Например, учитывая

`alpha = count();`

чтобы Little C знал, что `count` — это функция, а не переменная, он должен прочитать и `count`, и следующий токен, который в данном случае является скобкой. Однако, если бы заявление читалось

`alpha = count * 10;`

тогда следующий токен после счетчика — это `*`, который нужно будет вернуть во входной поток для последующего использования. По этой причине файл синтаксического анализатора выражений включает функцию `shift_source_code_location_back()`, которая может переместить указатель на один токен назад, в случае необходимости.

```
/**  
 * @brief Возврат лексемы во входной поток.  
 *  
 * Передвигаем указатель на текущую программу на *_токен_* обратно  
 */  
void shift_source_code_location_back(void)  
{  
    char *t;  
  
    t = current_token;  
    for (; *t; t++)  
        source_code_location--;  
}
```

Интерпретатор Little C

В этом разделе раскрывается суть интерпретатора Little C. Прежде чем перейти к коду интерпретатора, желательно понимать в общем, как работает интерпретатор. В целом код интерпретатора легче понять, чем синтаксический анализатор выражений, потому что концептуально процесс интерпретации программы на С можно описать следующим алгоритмом:

```
while(tokens_present) {  
    — get next token;  
    take appropriate action;  
}
```

Этот алгоритм может показаться невероятно простым по сравнению с парсером выражений, но это действительно именно то, что делают все интерпретаторы. Следует иметь в виду, что “action” может также включать чтение дополнительных токенов из входного потока. Чтобы понять, как алгоритм действительно работает, давайте вручную интерпретируем следующий фрагмент кода С:

```
int a;  
a = 10;  
if(a < 100) printf("%d", a);
```

Следуя алгоритму, прочитайте первый токен, который является `int`. Помимо этого токен так же выполняет функцию чтения следующего токена, чтобы узнать, как называется объявляемая переменная (в данном случае `a`) и потом хранить. Следующий токен — это точка с запятой, которой заканчивается строка. Здесь уместно проигнорировать ее. Далее возвращаемся и получаем еще один токен. Этот токен `a`. Поскольку эта строка не начинается с ключевого слова, она должна начинаться с выражения С. Таким образом, подходящим действием является вычисление выражения с помощью парсера. Этот процесс включает в себя все токены в этой строке. Наконец, мы читаем токен `if`. Этот сигнализирует о начале оператора `if`. Надлежащим действием является обработка `if`. Описанный здесь процесс выполняется для любой программы на С до тех пор, пока не будет прочитан последний токен. С этим имея в виду базовый алгоритм, давайте начнем строить интерпретатор.

Предварительное сканирование программы

Прежде чем интерпретатор сможет фактически начать выполнение программы, необходимо выполнить несколько предварительных задач. Одна из характеристик языков, которые были разработаны с интерпретацией, а не с компиляцией в том, что они начинают выполнение в начале исходного кода и заканчиваются, когда конец исходного кода достигнут. Так работает традиционный BASIC. Однако С (или любой другой структурированный язык) не поддается такому подходу по трем основным причинам.

Во-первых, все программы на С начинаются выполнение в функции **main()**. Нет требования, чтобы функция **main()** была первой в списке. programma; следовательно, необходимо, чтобы расположение функции **main()** внутри программы исходный код должен быть известен, чтобы с этого момента можно было начать выполнение. (Помните также, что глобальные переменные могут предшествовать **main()**, так что даже если это первая функция, это не обязательно первая строка код.) Должен быть разработан какой-то метод, позволяющий начать выполнение в нужном месте.

Другая проблема, которую необходимо решить, заключается в том, что все глобальные переменные должны быть известны и учтены до того, как **main()** начнет выполняться. Операторы объявления глобальных переменных никогда не выполняются интерпретатором, потому что они существуют вне всех функций. (Помните: в С весь исполняемый код существует внутри функций, поэтому у интерпретатора Little C нет причин выходить за пределы функции как только выполнение кода началось.)

Наконец, в интересах скорости выполнения важно (хотя технически и не обязательно), расположение каждой функции, определенной в программе, должно быть известно, чтобы вызов функции был как можно быстрее. Если этот шаг не выполнен, то будет выполняться длительный последовательный поиск исходного кода. Необходимо будет найти точку входа в функцию каждый раз, когда она вызывается. Решением этих проблем является интерпретатор **prescan_source_code**.

Пресканеры (или препроцессоры, как они иногда называются, хотя они мало похожи на препроцессор компилятора С) используются все коммерческие устные переводчики независимо от того, на какой язык они переводят. Предварительный сканер считывает исходный код в программу до ее выполнения и выполняет все задачи, которые могут быть выполнены до исполнение. В нашем интерпретаторе Little C он выполняет две важные задачи: во-первых, он находит и записывает расположение всех пользовательских функций, включая **main()**; во-вторых, он находит и выделяет место для всех глобальных переменных. В интерпретаторе Little C функция, выполняющая предварительное сканирование, называется предварительным сканированием.

```

void prescan_source_code()
{
    char *initial_source_code_location, *temp_source_code_location;
    char temp_token[ID_LEN + 1];
    int datatype;
    /// Если is_brace_open = 0, о текущая позиция указателя программы находится в не какой-либо функции
    int is_brace_open = 0;

    initial_source_code_location = source_code_location;
    pos_in_funcition = 0;
    do
    {
        while (is_brace_open) /* обхода кода функции внутри фигурных скобок */
        {
            get_next_token();
            if (*current_token == '{') /* когда встречаем открывающую скобку, увеличиваем is_brace_open на один */
                is_brace_open++;
            if (*current_token == '}')
                is_brace_open--; /* когда встречаем закрывающую уменьшаем на один */
        }

        temp_source_code_location = source_code_location; /* запоминаем текущую позицию */
        get_next_token();

        /// тип глобальной переменной или возвращаемого значения функции
        if (current_tok_datatype == CHAR || current_tok_datatype == INT)
        {
            datatype = current_tok_datatype; /* сохраняем тип данных */
            get_next_token();
            if (token_type == VARIABLE)
            {
                //
                strcpy_s( Destination: temp_token, SizelnBytes: ID_LEN + 1, Source: current_token);
                get_next_token();
                if (*current_token != '(' /* должно быть глобальной переменной */
                {
                    source_code_location = temp_source_code_location; /* вернуться в начало объявления */
                    declare_global_variables();
                }
                else if (*current_token == '(' /* должно быть функцией */
                {
                    function_table[pos_in_funcition].loc = source_code_location;
                    function_table[pos_in_funcition].ret_type = datatype;
                    strcpy_s( Destination: function_table[pos_in_funcition].func_name, SizelnBytes: ID_LEN, Source: temp_token);
                    pos_in_funcition++;
                    while (*source_code_location != ')')
                        source_code_location++;
                    source_code_location++;
                    /* сейчас source_code_location указывает на открывающуюся фигурную скобку функции */
                }
            }
        }
    }
}

```

```

        }
        else
            shift_source_code_location_back();
    }
}

else if (*current_token == '{')
    is_brace_open++;

} while (current_tok_datatype != FINISHED);

source_code_location = initial_source_code_location;
}

```

Функция `prescan_source_code()` работает следующим образом: каждый раз, когда встречается открывающая фигурная скобка, значение `is_brace_open` увеличивается. Всякий раз, когда читается закрывающая фигурная скобка, `is_brace_open` уменьшается.

Следовательно, всякий раз, когда `is_brace_open` больше нуля, текущий токен считывается из функции. Однако, если `is_brace_open` равна нулю, тогда переменная найдена, пресканиер знает, что это глобальная переменная. Таким же образом, если встречается имя функции, когда `is_brace_open` равна нулю, это должно быть определение этой функции.

Глобальные переменные хранятся в таблице глобальных переменных с именем `global_vars` функцией `declare_global_variables()`, как показано здесь:

```
/**  
 * @brief Структура переменных  
 */  
  
struct variable_type  
{  
    char variable_name[ID_LEN];  
    int variable_type;  
    int variable_value;  
};  
  
// Массив глобальных переменных  
variable_type global_vars[NUM_GLOBAL_VARS];  
  
void declare_global_variables()  
{  
    int variable_type;  
    get_next_token(); /* получаем тип данных */  
    variable_type = current_tok_datatype; /* запоминаем тип данных */  
  
    // Обработка списка переменных  
    do  
    {  
        global_vars[global_variable_position].variable_type = variable_type;  
        global_vars[global_variable_position].variable_value = 0; /* инициализируем нулем */  
        get_next_token(); /* определяем имя */  
        strcpy_s( Destination: global_vars[global_variable_position].variable_name, SizelnBytes: ID_LEN, Source: current_token);  
        get_next_token();  
        global_variable_position++;  
    } while (*current_token == ',');  
  
    if (*current_token != ';')  
        syntax_error( error: SEMICOLON_EXPECTED);  
}
```

Целочисленные `global_variable_position` хранят адрес следующей свободной ячейки в массиве. Расположение каждой пользовательской функции содержится в массиве `find_function_in_function_table`, показано здесь:

```

/**
 * @brief Структура функции
 */
struct function_type
{
    char func_name[ID_LEN];
    int ret_type;
    char *loc;
};

/// Массив функций
function_type function_table[NUMBER_FUNCTIONS];

char *find_function_in_function_table(char *name)
{
    int pos_in_func;

    for (pos_in_func = 0; pos_in_func < pos_in_funcition; pos_in_func++)
        if (!strcmp(name, function_table[pos_in_func].func_name))
            return function_table[pos_in_func].loc;

    return nullptr;
}

```

Переменная **pos_in_func** хранит индекс следующего свободного места в таблице

Функции **main()** и **execute()**

Показанная здесь функция **main()** интерпретатора Little C получает название и передает его в функцию **execute()**, которая в свою очередь инициализирует глобальные переменные, вызывает **prescan_source_code()**, "подготавливает" интерпретатор к вызову **main()** из интерпретируемого файла, а затем выполняет **call_function()**, который начинает выполнение программы. Работа функции **call_function()** будет коротко описана.

```
int main(int argc, char *argv[])
{
    string file;
    char *file_name;
    // если мы запустили данную программу через консоль
    // и передали в качестве параметра имя файла с программой
    if (argc == 2)
    {
        strcpy(file_name, argv[1]);
    }
    // Также мы можем ввести название программы через консоль вручную
    else
    {
        cout << "Файл: ";
        cin >> file;
        file_name = new char[file.length() + 1];
        strcpy(file_name, file.c_str());
    }

    execute(file_name);

    return 0;
}
```

```
void execute(char *fileName)
{
    /// Если названия файла нет - выход
    if (fileName[0] == ' ')
    {
        cout << "Пустое имя файла" << endl;
        exit(1);
    }
    /// Выделить память под программу PROG_SIZE - размер программы, не получилось - выход
    if ((program_start_buffer = (char *)malloc(PROG_SIZE)) == nullptr)
    {
        cout << "Сбой распределения памяти" << endl;
        exit(1);
    }
    /// Загрузить программу для выполнения
    if (!load_program(program_start_buffer, fileName))
    {
        cout << "Не удалось считать код" << endl;
        exit(1);
    }
    /// Инициализация буфера longjmp
    if (setjmp(execution_buffer))
    {
        exit(1);
    }
    /// Инициализация индекса глобальных переменных
    global_variable_position = 0;
    /// Установка указателя на начало буфера программы
    source_code_location = program_start_buffer;
    /// Определение адресов всех функций и глобальных переменных
    prescan_source_code();
    /// Инициализация индекса стека локальных переменных
    local_var_to_stack = 0;
    /// Инициализация индекса стека вызова CALL
    function_last_index_on_call_stack = 0;
    /// initialize the break occurring flag
    break_occurred = 0;
    /// Вызываем функцию main она всегда вызывается первой
    source_code_location = find_function_in_function_table("main");
    /// main написан с ошибкой или отсутствует
    if (!source_code_location)
    {
        cout << "\"main\" не найдено или написано с ошибкой" << endl;
        exit(1);
    }
    /// Возвращаемся к открывающей (
    source_code_location--;
    strcpy_s(current_token, 80, "main");
    /// Вызываем main и интерпретируем
    call_function();
}
```

Функция `interpret_block()`

Функция `interpret_block()` является сердцем интерпретатора. Это функция, которая решает, какое действие предпринять на основе следующего токена во входном потоке. Функция предназначена для интерпретации одного блока кода и последующего возврата. Если «блок» состоит из одного оператора, этот оператор интерпретируется, и функция возвращается. По умолчанию `interp_block()` интерпретирует один оператор и возвращает результат. Однако, если читается открывающая фигурная скобка, блоку флагов присваивается значение 1, и функция продолжает интерпретировать операторы до тех пор, пока не будет прочитана закрывающая фигурная скобка. Здесь показана функция `interpret_block()`:

```
/*
 * @brief Интерпретация одного оператора или блока
 *
 * Когда interpret_block() возвращает управление после первого вызова, в main() встретилась последняя закрывающаяся фигурная скобка или оператор return.
 */
void interpret_block()
{
    int value;
    char block = 0;

    do
    {
        token_type = get_next_token();
        /// При интерпретации одного оператора возврат после первой точки с запятой.

        // Определение типа лексемы
        if (token_type == VARIABLE) /* Это не зарегистрированное слово, обрабатывается выражение. */
        {
            shift_source_code_location_back(); /* возврат лексемы во входной поток для дальнейшей обработки функцией eval_exp() */
            eval_expression(&value); /* обработка выражения */
            if (*current_token != ';')
                syntax_error(SEMICOLON_EXPECTED);
        }
        else if (token_type == BLOCK) /* Это ограничитель блока */
        {
            if (*current_token == '{') /* Блок */
                block = 1; /* Интерпретация блока, а не оператора */
            else
                return; /* Это }, выход */
        }
        else /* Зарезервированное слово */
        switch (current_tok_datatype)
        {
            case CHAR:
            case INT: /* объявление локальной переменной */
                shift_source_code_location_back();
                declare_local_variables();
                break;
            case RETURN: /* возврат из вызова функции */
                function_return();
                ret_occurred = 1;
                return;
            case CONTINUE: /* обработка оператора continue */
                return;
            case BREAK: /* выход из цикла */
                break_occurred = 1;
                return;
            case IF: /* обработка оператора if */
                execute_if_statement();
                if (ret_occurred > 0 || break_occurred > 0)
                {
                    return;
                }
                break;
            case ELSE: /* обработка оператора else */
                find_eob(); /* поиск конца блока else и продолжение выполнения */
                break;
            case WHILE: /* обработка цикла while */
                exec_while();
                if (ret_occurred > 0)
                {
                    return;
                }
                break;
        }
    } while (block);
}
```

```

        case ELSE: /* обработка оператора else */
            find_eob(); /* поиск конца блока else и продолжение выполнения */
            break;
        case WHILE: /* обработка цикла while */
            exec_while();
            if (ret_occurred > 0)
            {
                return;
            }
            break;
        case DO: /* обработка цикла do-while */
            exec_do();
            if (ret_occurred > 0)
            {
                return;
            }
            break;
        case FOR: /* обработка цикла for */
            exec_for();
            if (ret_occurred > 0)
            {
                return;
            }
            break;
        case END: /* Конец иши */
            exit(0);
        }
    } while (current_tok_datatype != FINISHED && block);
}

```

За исключением вызовов таких функций, как `exit()`, программа на С завершается, когда встречается последняя фигурная скобка (или возврат) в `main()` — не обязательно в последней строке исходного кода. Это одна из причин, по которой функция `interpret_block()` выполняет только оператор или блок кода, а не всю программу. Кроме того, концептуально С состоит из блоков кода. Поэтому функция `interpret_block()` вызывается каждый раз, когда встречается новый блок кода. Сюда входят как функциональные блоки, так и блоки, начинающиеся с различных операторов С, таких как `if`. Это означает, что в процессе выполнения программы интерпретатор Little C может рекурсивно вызывать функцию `interpret_block()`.

Функция `interpret_block()` работает следующим образом: сначала она считывает из программы следующую лексему. Если токен представляет собой точку с запятой и интерпретируется только один оператор, функция возвращает значение. В противном случае он проверяет, является ли токен идентификатором; если это так, оператор должен быть выражением, поэтому вызывается синтаксическим анализатором выражений. Поскольку синтаксический анализатор выражений рассчитывает прочитать первую лексему в самом выражении, лексема возвращается во входной поток через вызов `shift_source_code_location_back()`. Когда функция `eval_expression()` возвращает значение, `token` будет содержать последнюю лексему, прочитанную синтаксическим анализатором выражений, которая должна быть точкой с запятой, если оператор синтаксически корректен. Если токен не содержит точки с запятой, сообщается об ошибке.

Если следующий токен из программы — фигурная скобка, то переменная `block` устанавливается в 1 в случае открывающей скобки, либо, если это закрывающая скобка, функция возвращает значение.

Наконец, если токен является **keyword**, выполняется оператор **switch**, вызывая соответствующую процедуру для обработки оператора. Причина, по которой ключевым словам присваиваются целочисленные эквиваленты с помощью **get_next_token()**, заключается в том, чтобы использовать оператор **switch** вместо **if**, который бы сравнивал строки, что по своей сути выполняется очень медленно.

Обработка локальных переменных

Когда интерпретатор встречает ключевое слово `int` или `char`, он вызывает `declare_local_variables()`, чтобы создать хранилище для локальной переменной. Как было сказано ранее - оператор объявление не будет задействован, если программа уже выполняется.

Следовательно, если найден оператор объявления переменной, он должен быть локальной переменной (или параметром). В структурированных языках локальные переменные хранятся в кучах. Если язык компилируемый, обычно используется системный стек; однако в интерпретируемом режиме стек для локальных переменных должен поддерживаться интерпретатором. Стек для локальных переменных хранится в массиве `local_var_stack`. Каждый раз, когда встречается локальная переменная, ее имя, тип и значение (изначально нулевое) помещаются в стек с помощью функции `local_push()`. Глобальная переменная `local_var_to_stack` индексирует стек. (По причинам, которые станут ясны, нет соответствующей «рор» функции. Вместо этого стек локальных переменных сбрасывается каждый раз, когда функция возвращает значение.) Параметры `declare_local_variables` и здесь показаны функции `local_push()`

```
/**  
 * @brief Затолкнуть локальную переменную в local_var_stack  
 * @param local_variable  
 */  
void local_push(struct variable_type local_variable)  
{  
    if (local_var_to_stack >= NUM_LOCAL_VARS)  
    {  
        syntax_error(TOO_MANY_LVARS);  
    }  
    else  
    {  
        local_var_stack[local_var_to_stack] = local_variable;  
        local_var_to_stack++;  
    }  
}  
  
// Массив локальных переменных  
variable_type local_var_stack[NUM_LOCAL_VARS];
```

```
/**  
 * @brief Объявление локальной переменной  
 *  
 * Добавляем local_var_stack через local_push  
 */  
void declare_local_variables()  
{  
    struct variable_type local_variable;  
  
    /// Получить типа  
    get_next_token();  
  
    local_variable.variable_type = current_tok_datatype;  
    local_variable.variable_value = 0; /* Инициализация нулем 0 */  
  
    /// Обработка списка переменных  
    do  
    {  
        get_next_token(); /* определение имени */  
        strcpy_s(local_variable.variable_name, ID_LEN, current_token);  
        local_push(local_variable);  
        get_next_token();  
    } while (*current_token == ',');  
  
    if (*current_token != ';')  
        syntax_error(SEMICOLON_EXPECTED);  
}
```

declare_local_variables

Функция `declare_local_variables()` сначала считывает тип объявляемой переменной или переменных и устанавливает начальное значение равным нулю. Затем он входит в цикл, который считывает список идентификаторов, разделенных запятыми. Каждый раз в цикле информация о каждой переменной помещается в стек локальных переменных. В конце окончательный токен проверяется, чтобы убедиться, что он содержит точку с запятой.

Вызов пользовательских функций

Вероятно, наиболее сложной частью реализации интерпретатора для С является выполнение определяемых пользователем функций. Интерпретатору нужно не только начать чтение исходного кода с нового места, но и затем вернуться к вызывающей процедуре после завершения функции. Кроме того необходимо решать следующие задачи: передача аргументов, выделение параметров и возвращаемые значение функции.

Все вызовы функций (кроме начального вызова `main()`) происходят через синтаксический анализатор выражений из функцию `atom()` вызовом `call_function()`. Это функция `call_function()`, который фактически обрабатывает детали вызова функции. Здесь показана функция `call_function()` вместе с двумя вспомогательными функциями.

```
void call_function()
{
    char *function_location, *temp_source_code_location;
    int local_var_temp;

    function_location = find_function_in_function_table( name: current_token); /* найти точку входа функции */
    if (function_location == NULL)
        syntax_error( error: FUNC_UNDEFINED); /* функция не определена */
    else
    {
        local_var_temp = local_var_to_stack;                                /* запоминание индекса стека локальных переменных */
        get_function_arguments();                                         /* получение аргумента функции */
        temp_source_code_location = source_code_location; /* запоминание адреса возврата */
        function_push_variables_on_call_stack( &local_var_temp); /* запоминание индекса стека локальных переменных */
        source_code_location = function_location;           /* переустановка source_code_location в начало функции */
        ret_occurred = 0;                                              /* Р возвращаемая возникающая переменная */
        get_function_parameters();                                     /* загрузка параметров функции значениями аргументов */
        interpret_block();                                            /* интерпретация функции */
        ret_occurred = 0;                                              /* обнуление возвращаемой переменной */
        source_code_location = temp_source_code_location; /* восстановление initial_source_code_location */
        local_var_to_stack = func_pop();                            /* сброс стека локальных переменных */
    }
}
```

```
/**  
 * @brief Получение значения числа, переменной или функции  
 * @param value  
 */  
void atom(int *value)  
{  
    int i;  
  
    switch (token_type)  
    {  
        case VARIABLE:  
            i = internal_func(current_token);  
            if (i != -1)  
            { /* вызов стандартной функции */  
                *value = (*intern_func[i].p)();  
            }  
            else if (find_function_in_function_table(current_token))  
            { /* вызов пользовательской функции */  
                call_function();  
                *value = ret_value;  
            }  
            else  
                *value = find_var(current_token); /* получаем значение переменной */  
            get_next_token();  
            return;  
        case NUMBER: /* числовая константа */  
            *value = atoi(current_token);  
            get_next_token();  
            return;  
        case DELIMITER: /* символьная константа */  
            if (*current_token == '\\')  
            {  
                *value = *source_code_location;  
                source_code_location++;  
                if (*source_code_location != '\\')  
                    syntax_error(QQUOTE_EXPECTED);  
                source_code_location++;  
                get_next_token();  
                return;  
            }  
            if (*current_token == ')')  
                return; /* пустое выражение в скобках */  
            else  
                syntax_error(SYNTAX); /* синтаксическая ошибка */  
        default:  
            syntax_error(SYNTAX); /* синтаксическая ошибка */  
    }  
}
```

```

void get_function_arguments()
{
    int value, count, temp[NUM_PARAMS];
    struct variable_type i;

    count = 0;
    get_next_token();
    if (*current_token != '(')
        syntax_error( error: PAREN_EXPECTED);

    /// Обработка списка значений
    do
    {
        eval_expression( value: &value);
        temp[count] = value; /* временное запоминание */
        get_next_token();
        count++;
    }
    while (*current_token == ',');
    count--;

    /// Затолкнуть в local_var_stack в обратном порядке
    for ( ; count >= 0; count--)
    {
        i.variable_value = temp[count];
        i.variable_type = ARG;
        local_push(i);
    }
}

```

Первое, что делает функция `call_function()`, — это находит в исходном коде точку входа в исходный код, указанную функцию, вызывав `find_function_in_function_table()`. Далее она сохраняет текущее значение стека локальной переменной индекс, `local_var_to_stack_index`, в `local_var_temp`; затем она вызывает `get_function_arguments()` для обработки всех аргументов функции. Функция `get_function_arguments()` считывает список выражений, разделенных запятыми, и помещает их в стек локальных переменных в обратном порядке. (Выражения помещаются в обратном порядке, чтобы их было легче сопоставить с соответствующими параметрами.). Имена параметров даются им `get_function_parameters()`, о которой мы поговорим чуть позже.

После обработки аргументов функции текущее значение **source_code_location** сохраняется в **temp_source_code_location** - это точка возврата функции. Затем значение **local_var_temp** помещается в функцию. стек вызовов. Подпрограммы **function_push_variables_on_call_stack()** и **func_pop()** поддерживают этот стек. Его назначение – хранить значение **local_var_to_stack_index** при каждом вызове функции. Это значение представляет начальную точку в локальном стеке переменных для переменных (и параметров) относительно вызываемой функции. Значение верхней части стека вызовов функций используется для предотвращения доступа функции к каким-либо локальным переменным, кроме тех, которые он декларирует.

Следующие две строки кода устанавливают указатель программы на начало функции и связывают имя его формальные параметры со значениями аргументов уже в стеке локальной переменной с вызовом для **get_function_parameters()**. Фактическое выполнение функции осуществляется через вызов **interpret_block()**. Когда функция **interpret_block()** возвращает значение, указатель программы **source_code_location** сбрасывается в точку возврата, и индекс стека локальной переменной сбрасывается до своего значения перед вызовом функции. Этот последний шаг эффективно удаляет все локальные переменные функции из стека. Если вызываемая функция содержит оператор **return**, то функция **interpret_block()** вызывает **function_return()** до возврата к **call_function()**. Эта функция обрабатывает любое возвращаемое значение. Здесь показано:

```
void function_return()
{
    int value = 0;
    // Получение возвращаемого значения, если оно есть
    eval_expression( value: &value );
    ret_value = value;
}
```

Переменная **ret_value** — это глобальное целое число, которое содержит возвращаемое значение функции. На первый взгляд, вы можете задаться вопросом, почему значение локальной переменной сначала присваивается возвращаемому значению функции, а затем присваивается **ret_value**. Причина в том, что функции могут быть рекурсивными, а функция **eval_expression()** может необходимо вызвать ту же функцию, чтобы получить ее значение.

Присвоение значений переменным

Кратко вернемся к анализатору выражений. Когда встречается оператор присваивания, значение вычисляется правая часть выражения, и это значение присваивается переменной слева используя вызов `assign_var()`. Однако, как вы знаете, язык С структурирован и поддерживает глобальные и локальные переменные. Таким образом, при наличии такой программы, как эта

```
int count;

int main()
{
    int count, i;

    count = 100;

    i = f();

    return 0;
}

int f()
{
    int count;
    count = 99;
    return count;
}
```

как функция `assign_var()` узнает, какой переменной присваивается значение в каждом назначении? Ответ прост: во-первых, в С локальные переменные имеют приоритет над глобальными переменными языка С. то же имя; во-вторых, локальные переменные неизвестны вне их собственной функции. Чтобы увидеть, как мы могут использовать эти правила для решения вышеуказанных присваиваний, проверьте функцию `assign_var()`, показанную здесь:

```
void assign_var(char *var_name, int value)
{
    int i;
    // Проверка наличия локальной переменной
    for (i = local_var_to_stack - 1; i >= call_stack[function_last_index_on_call_stack - 1]; i--)
    {
        if (!strcmp(local_var_stack[i].variable_name, var_name))
        {
            local_var_stack[i].variable_value = value;
            return;
        }
    }

    // Если переменная нелокальная, ищем ее в таблице глобальных переменных
    if (i < call_stack[function_last_index_on_call_stack - 1])
        for (i = 0; i < NUM_GLOBAL_VARS; i++)
            if (!strcmp(global_vars[i].variable_name, var_name))
            {
                global_vars[i].variable_value = value;
                return;
            }

    // Переменной не существует
    syntax_error(error: NOT_VAR);
}
```

Как объяснялось в предыдущем разделе, при каждом вызове функции текущее значение локального переменный индекс стека (`local_var_to_stack`) помещается в стек вызовов функций. Это означает, что любое местное переменные (или параметры), определенные функцией, будут помещены в стек выше

этой точки. Поэтому функция **assign_var()** сначала выполняет поиск в **local_var_stack**, начиная с текущего значения вершины стека и останавливаясь, когда индекс достигает значения, сохраненного последним вызовом функции. Этот механизм гарантирует, что проверяются только те переменные, которые являются локальными для функции. (так же помогает поддерживать рекурсивные функции, потому что текущее значение **local_var_to_stack** сохраняется каждый раз, когда функция вызывается.) Таким образом, строка **count = 100;"** в **main()** заставляет функцию **assign_var()** найти локальную количества переменных внутри функции **main()**. В **f()** функция **assign_var()** находит свой собственный счетчик и не находит тот, который находится в **main()**. Если ни одна локальная переменная не совпадает с именем переменной, выполняется поиск в списке глобальных переменных.

Инструкция if

Теперь, когда базовая структура интерпретатора Little C готова, пришло время добавить некоторые элементы управления. заявления. Каждый раз, когда оператор ключевого слова встречается внутри `interpret_block()`, соответствующий вызывается функция, которая обрабатывает этот оператор. Одним из самых простых является `if`. Оператор `if` обрабатывается функцией `execute_if_statement()`, как показано здесь:

```
void execute_if_statement()
{
    int condition;
    eval_expression( value: &condition); /* вычисление if-выражения */
    if (condition)                  /* истина - интерпретация if-предложения */
    {
        interpret_block();
    }
    else /* в противном случае пропуск if-предложения и выполнение else-предложения, если оно есть */
    {
        find_eob(); /* поиск конца блока */
        get_next_token();
        if (current_tok_datatype != ELSE)
        {
            shift_source_code_location_back(); /* восстановление лексемы, если else-предложение отсутствует */
            return;
        }
        interpret_block();
    }
}
```

Первое, что делает функция, это вычисляет значение условное выражение, вызывая `eval_expression()`. Если условие (`condition`) истинно (отлично от нуля), функция рекурсивно вызывает `interpret_block()`, позволяя выполниться блоку `if`. Если `condition` ложно, вызывается функция `find_eob()`, которая перемещает указатель программы на позицию сразу после конца блока `if`. Если присутствует `else`, то это `else` обрабатывается функцией `execute_if_statement()`, а блок `else` обрабатывается. В противном случае выполнение просто начинается со следующей строки кода. Если блок `if` выполняется и присутствует блок `else`, должен быть какой-то способ для блока `else` не исполняться. Это достигается в `interpret_block()` простым вызовом `find_eob()` чтобы когда встречается блок `else` он не выполнялся. Помните, что единственный раз, когда `else` будет обработан `interpret_block()` (в синтаксически правильной программе) выполняется после выполнения блока `if`. Когда блок `else` выполняется, `else` обрабатывается функцией `execute_if_statement()`.

Цикл while

Цикл **while**, как и цикл **if** легко интерпретировать. Функция, которая выполняет эту задачу - **exec_while**:

```
void exec_while()
{
    int condition;
    char *temp;
    break_occurred = 0; /* флаг break */
    shift_source_code_location_back();
    temp = source_code_location; /* запоминание адреса начала цикла while */
    get_next_token();
    eval_expression( value: &condition); /* вычисление управляющего выражения */
    if (condition)           /* если оно истинно, то выполнить интерпретацию */
    {
        interpret_block();
        if (break_occurred > 0)
        {
            break_occurred = 0;
            return;
        }
    }
    else /* в противном случае цикл пропускается */
    {
        find_eob();
        return;
    }
    source_code_location = temp; /* возврат к началу цикла */
```

Функция **exec_while()** работает следующим образом: во-первых, токен **while** помещается обратно во входной поток, а местоположение в то время как сохраняется в **temp**. Этот адрес будет использоваться, чтобы разрешить интерпретатору зацикливаться. вернуться к началу времени. Затем **while** перечитывается, чтобы удалить его из входного потока, и **eval_expression()** вызывается для вычисления значения условного выражения **while**. Если условно выражение истинно, то функция **interpret_block()** вызывается рекурсивно для интерпретации блока **while**. Когда **interpret_block()** возвращает значение, **source_code_location**(указатель программы) загружается с местоположением начала **while**, и управление возвращается к функции **interpret_block()**, где весь процесс повторяется. Если условное выражение ложно, найден конец блока **while** и функция возвращает значение.

Цикл do-while

Цикл **do-while** обрабатывается практически так же, как и цикл **while**. Когда **interpret_block()** встречает **do**, он вызывает **exec_do()**, показано здесь:

```
void exec_do()
{
    int condition;
    char *temp;

    shift_source_code_location_back();
    temp = source_code_location; /* запоминание адреса начала цикла */
    break_occurred = 0;         /* флаг break */

    get_next_token(); /* найти начало цикла */
    interpret_block(); /* интерпритация цикла */
    if (ret_occurred > 0)
    {
        return;
    }
    else if (break_occurred > 0)
    {
        break_occurred = 0;
        return;
    }
    get_next_token();
    if (current_tok_datatype != WHILE)
        syntax_error(error: WHILE_EXPECTED);
    eval_expression(value: &condition); /* проверка условия цикла */
    if (condition)
        source_code_location = temp; /* если условие истинно, то цикл выполняется, в противном случае происходит выход из цикла */
}
```

Основное различие между циклами **do-while** и **while** заключается в том, что цикл **do-while** выполняет свой блок кода хотя бы один раз, потому что условное выражение находится в конце цикла. Поэтому функция **exec_do()** сначала сохраняет положение начала цикла во временной памяти, а затем вызывает **interpret_block()**, рекурсивно интерпретирует блок кода, связанный с циклом. Когда **interpret_block()** возвращает значение, извлекается соответствующее время, и условное выражение оценивается. Если условие истинно, программа сбрасывается на начало цикла; в противном случае выполнение будет продолжаться.

Цикл for

Интерпретация цикла **for** представляет собой более сложную задачу, чем другие конструкции. Частично из-за того что в структуре С **for** определенно разработана с учетом компиляции. Основная проблема в том, что условное выражение **for** должно проверяться в начале цикла, но часть приращения происходит в нижней части цикла. Поэтому, хотя эти две части цикла **for** встречаются в исходном коде рядом друг с другом, их интерпретация разделена повторяющимся блоком кода. Однако, немного подумав,, **for** можно правильно интерпретировать. Когда функция **interpret_block()** встречает оператор **for**, вызывается функция **exec_for()**. Эта функция показана здесь

```
void exec_for(void)
{
    int condition;
    char *temp, *temp2;
    int brace;

    break_occurred = 0; /* флаг break */
    get_next_token();
    eval_expression( value: &condition); /* инициализирующее выражение */
    if (*current_token != ';')
        syntax_error( error: SEMICOLON_EXPECTED);

    source_code_location++; /* пропуск ; */
    temp = source_code_location;
    for (;;)
    {
        eval_expression( value: &condition); /* проверка условия */
        if (*current_token != ';')
            syntax_error( error: SEMICOLON_EXPECTED);
        source_code_location++; /* пропуск ; */
        temp2 = source_code_location;

        /// Поиск начала тела цикла
        brace = 1;
        while (brace)
        {
            get_next_token();
            if (*current_token == '(')
                brace++;
            if (*current_token == ')')
                brace--;
        }

        /// если условие выполнено, то выполнить интерпретацию */
        if (condition)
        {
            interpret_block();
            if (ret_occurred > 0)
            {
                return;
            }
            else if (break_occurred > 0)
            {
                break_occurred = 0;
                return;
            }
        }
    }
}
```

```
        return;
    }
}
else /* в противном случае обойти цикл */
{
    find_eob();
    return;
}
source_code_location = temp2;
eval_expression( value: &condition); /* выполнение инкремента */
source_code_location = temp; /* возврат в начало цикла */
}
```

Эта функция начинается с обработки выражения инициализации в **for**. Часть инициализации **for** выполняется только один раз и не является частью цикла. Далее указатель программы, находящийся сразу после знака “точка с запятой”, которая заканчивает оператор инициализации, и его значение присвоено **temp**. Затем устанавливается цикл, который проверяет условную часть **for** и присваивает **temp2** указатель на начало части приращения. Начало кода цикла найдено, и, если условное выражение истинно, блок цикла интерпретируется. (В противном случае будет найден конец блока, и выполнение продолжится после цикла **for**.) возвращается рекурсивный вызов функции **interpret_block()**, выполняется инкрементная часть цикла, и процесс повторяется.

Библиотечные функции в Little C

Поскольку программы на С, выполняемые Little C, никогда не компилируются и не компонуются, любые библиотечные процедуры должны обрабатываться непосредственно в Little C. Лучший способ сделать это — создать интерфейс, который Little C вызывает при встрече с библиотечной функцией. Этот интерфейс устанавливает вызов библиотечной функции и обрабатывает любые возвращаемые значения. Из-за нехватки места Little C содержит только пять «библиотечных» функций: **call_getche()**, **call_putch()**, **call_puts()**, **print()** и **getnum()**. Из них только **puts()** выводит строку на экран, является частью стандарта С. Функция **call_getche()** является обычным расширением языка С для интерактивные среды. Он ждет и возвращает нажатие клавиши на клавиатуре. Эта функция встречается во многих компиляторах. **call_putch()** также определяется многими компиляторами, предназначенными для использования в интерактивная среда. Он выводит на консоль аргумент из одного символа. Он не буферизирует выход. Функции **getnum()** и **print()** — мои собственные творения. Функция **getnum()** возвращает целочисленный эквивалент числа, введенного с клавиатуры. Функция **print()** очень удобна. функция, которая может выводить на экран либо строку, либо целочисленный аргумент. Библиотечные функции показаны здесь:

```
int call_getche(void)
{
    char ch;
#if defined(_QC)
    ch = (char)getche();
#elif defined(_MSC_VER)
    ch = (char)_getche();
#else
    ch = (char)getchar();
#endif
    while (*source_code_location != ')')
        source_code_location++;
    /// Продолжаем работать, пока не достигнем
    source_code_location++;
    return ch;
}
/***
 * Вывод символа на экран
 * @return
 */
int call_putchar(char ch)
```

```
int call_putch(void)
{
    int value;

    eval_expression(result: &value);
    printf(Format: "%c", value);
    return value;
}
```

```
int call_puts(void)
{
    get_next_token();
    if (*current_token != '(')
        syntax_error(error: PAREN_EXPECTED);
    get_next_token();
    if (token_type != STRING)
        syntax_error(error: QUOTE_EXPECTED);
    puts(Buffer: current_token);
    get_next_token();
    if (*current_token != ')')
        syntax_error(error: PAREN_EXPECTED);

    get_next_token();
    if (*current_token != ';')
        syntax_error(error: SEMICOLON_EXPECTED);
    shift_source_code_location_back();
    return 0;
}
```

```
int print(void)
{
    int i;

    get_next_token();
    if (*current_token != '(')
        syntax_error(error: PAREN_EXPECTED);

    get_next_token();
    if (token_type == STRING)
    { /* выводим строку */
        printf(Format: "%s ", current_token);
    }
    else
    { /* выводим число */
        shift_source_code_location_back();
        eval_expression(result: &i);
        printf(Format: "%d ", i);
    }

    get_next_token();

    if (*current_token != ')')
        syntax_error(error: PAREN_EXPECTED);

    get_next_token();
    if (*current_token != ';')
        syntax_error(error: SEMICOLON_EXPECTED);
    shift_source_code_location_back();
```

```
int getnum(void)
{
    char s[80];

    if (fgets( Buffer: s, MaxCount: sizeof(s), Stream: stdin) != NULL)
    {
        while (*source_code_location != ')')
            source_code_location++;
        source_code_location++; /* читаем до конца строки */
        return atoi( String: s);
    }
    else
    {
        return 0;
    }
}
```

Так же есть возможность добавить в библиотеку стандартных функций свои функции, для этого нужно ввести их имена и указать адреса их интерфейсов в массиве `intern_func()`. Затем по примеру, данному в коде, можно воспроизвести интерфейс самой функции.