

Университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа №1 по дисциплине «Системное программное обеспечение» Вариант: 2

Выполнил:
Шибает Семен Сергеевич
Группа:
Р4114

Преподаватель:
Кореньков Юрий Дмитриевич

Санкт-Петербург
2023

Задание

Использовать средство синтаксического анализа по выбору, реализовать модуль для разбора текста в соответствии с языком по варианту. Реализовать построение по исходному файлу с текстом синтаксического дерева с узлами, соответствующими элементам синтаксической модели языка. Вывести полученное дерево в файл в формате, поддерживающем просмотр графического представления.

Порядок выполнения:

1. Изучить выбранное средство синтаксического анализа
 - a. Средство должно поддерживать программный интерфейс, совместимый с языком Си
 - b. Средство должно параметризоваться спецификацией, описывающей синтаксическую структуру разбираемого языка
 - c. Средство может функционировать посредством кодогенерации и/или подключения необходимых для его работы дополнительных библиотек
 - d. Средство может быть реализовано с нуля, в этом случае оно должно использовать обобщённый алгоритм, управляемый спецификацией
2. Изучить синтаксис разбираемого по варианту языка и записать спецификацию для средства синтаксического анализа, включающую следующие конструкции:
 - a. Подпрограммы со списком аргументов и возвращаемым значением
 - b. Операции контроля потока управления – простые ветвления if-else и циклы или аналоги
 - c. В зависимости от варианта – определения переменных
 - d. Целочисленные, строковые и односимвольные литералы
 - e. Выражения численной, битовой и логической арифметики
 - f. Выражения над одномерными массивами
 - g. Выражения вызова функции
3. Реализовать модуль, использующий средство синтаксического анализа для разбора языка по варианту
 - a. Программный интерфейс модуля должен принимать строку с текстом и возвращать структуру, описывающую соответствующее дерево разбора и коллекцию сообщений об ошибке

- b. Результат работы модуля – дерево разбора – должно содержать иерархическое представление для всех синтаксических конструкций, включая выражения, логически представляющие собой иерархически организованные данные, даже если на уровне средства синтаксического анализа для их разбора было использовано линейное представление
- 4. Реализовать тестовую программу для демонстрации работоспособности созданного модуля
 - a. Через аргументы командной строки программа должна принимать имя входного файла для
- 5. чтения и анализа, имя выходного файла записи для дерева, описывающего синтаксическую структуру разобранного текста
 - a. Сообщения об ошибке должны выводиться тестовой программой (не модулем, отвечающим за анализ!) в стандартный поток вывода ошибок
 - b. Результаты тестирования представить в виде отчета, в который включить:
 - c. В части 3 привести описание структур данных, представляющих результат разбора текста (3а)
 - d. В части 4 описать, какая дополнительная обработка потребовалась для результата разбора, предоставляемого средством синтаксического анализа, чтобы сформировать результат работы созданного модуля
 - e. В части 5 привести примеры исходных анализируемых текстов для всех синтаксических конструкций разбираемого языка и соответствующие результаты разбора

Грамматика по варианту

```
source: sourceItem*;

typeRef: {
    |builtin: 'bool'|'byte'|'int'|'uint'|'long'|'ulong'|'char'|'string';
    |custom: identifier;
    |array: 'array' '[' (','*) ']' 'of' typeRef;
};

funcSignature: identifier '(' list<argDef> ')' (':' typeRef)? {
    argDef: identifier (':' typeRef)?;
};

sourceItem: {
    |funcDef: 'method' funcSignature (body|';') {
        body: ('var' (list<identifier> (':' typeRef)? ';')*)? statement.block;
    };
};

statement: {
    |if: 'if' expr 'then' statement ('else' statement)?;
    |block: 'begin' statement* 'end' ';;';
    |while: 'while' expr 'do' statement;
    |do: 'repeat' statement ('while'|'until') expr ';;';
    |break: 'break' ';;';
    |expression: expr ';;';
};

expr: { // присваивание через ':='
    |binary: expr binOp expr; // где binOp - символ бинарного оператора
    |unary: unOp expr; // где unOp - символ унарного оператора
    |braces: '(' expr ')';
    |call: expr '(' list<expr> ')';
    |indexer: expr '[' list<expr> ']';
    |place: identifier;
    |literal: bool|str|char|hex|bits|dec;
};
```

Детали реализации

Определения для узлов абстрактного синтаксического дерева.

```
enum ast_node_type {
    COMMON,
    EXPR,
    SOURCE,
    FUNC_SIGN,
    BRANCH,
    BLOCK,
    LOOP,
    TYPE_NODE,
    VALUE,
    IDENTIFIER
};
```

```

static const char *ast_names[] = {
    [COMMON] = "common",
    [EXPR] = "expression",
    [SOURCE] = "source",
    [FUNC_SIGN] = "function_signature",
    [BRANCH] = "branch",
    [BLOCK] = "block",
    [LOOP] = "loop",
    [TYPE_NODE] = "type",
    [VALUE] = "value",
    [IDENTIFIER] = "identifier"
};

struct ast_node;

struct ast_expression { char
    oper_name[MAXIMUM_IDENTIFIER_LENGTH];
    struct ast_node *left; struct ast_node
    *right;
};

struct ast_common {
    char node_name[MAXIMUM_IDENTIFIER_LENGTH];
    struct ast_node *left; struct ast_node
    *right;
};

struct ast_loop {
    char loop_type[MAXIMUM_IDENTIFIER_LENGTH];
    struct ast_node *statement; struct
    ast_node *expression;
};

struct ast_block {
    struct ast_node *block_items;
};

```

```

struct ast_branch {
    struct ast_node *if_expr; struct
    ast_node *if_statement; struct
    ast_node *else_statement;
};

struct ast_function_signature {

struct ast_node *ident; struct

ast_node *args; struct ast_node

*type_ref;

};

struct ast_source { struct
    ast_node *source; struct
    ast_node *source_item;
};

struct ast_value { char
    type_name[MAXIMUM_IDENTIFIER_LENGTH]; char
    value[MAXIMUM_IDENTIFIER_LENGTH];
};

struct ast_type { char
    type_name[MAXIMUM_IDENTIFIER_LENGTH];
};

struct ast_identifier { char
    name[MAXIMUM_IDENTIFIER_LENGTH];
};

struct ast_node { enum ast_node_type type; unsigned long
    long id; union { struct ast_expression ast_expression;
    struct ast_source ast_source; struct
    ast_function_signature ast_function_signature; struct
    ast_branch ast_branch; struct ast_block ast_block; struct

```

```

    ast_loop ast_loop; struct ast_common ast_common; struct
    ast_type ast_type; struct ast_value ast_value; struct
    ast_identifier ast_identifier;
};

};

struct ast_node *make_common_node(char *, struct ast_node *,
struct ast_node *);
struct ast_node *make_expr_node(char *, struct ast_node *,
struct ast_node *);

struct ast_node *make_loop_node(char *, struct ast_node *,
struct ast_node *);

struct ast_node *make_branch_node(struct ast_node *, struct
ast_node *, struct ast_node *); struct ast_node
*make_block(struct ast_node *);

struct ast_node *make_function_signature(struct ast_node *,
struct ast_node *, struct ast_node *);

struct ast_node *make_source(struct ast_node *, struct
ast_node *); struct ast_node *make_value_node(char *,
char *); struct ast_node *make_type_node(char *);

struct ast_node *make_ident_node(char *); void
print_ast(struct ast_node *);

```

Функции для создания разных типов узлов абстрактного синтаксического дерева.

```
struct ast_node *make_node(enum ast_node_type type) { struct
    ast_node *node = (struct ast_node *) malloc(sizeof(struct
ast_node)); node->type
    = type; node->id =
    counter++; return
    node;
}
```

```
struct ast_node *make_common_node(char *name, struct ast_node
*first, struct ast_node *second) { struct
    ast_node *ident = make_node(COMMON);
    strncpy(ident->ast_common.node_name,
    name,
MAXIMUM_IDENTIFIER_LENGTH); ident->
    ast_common.left = first; ident->
    ast_common.right = second; return ident;
}
```

```
struct ast_node *make_expr_node(char *name, struct ast_node
*first, struct ast_node *second) { struct ast_node *expr =
make_node(EXPR); strncpy(expr->ast_expression.oper_name, name,
MAXIMUM_IDENTIFIER_LENGTH); expr->
    ast_expression.left = first; expr->
    ast_expression.right = second; return
    expr;
}
```

```
struct ast_node *make_loop_node(char *name, struct ast_node
*first, struct ast_node *second) { struct ast_node *loop =
make_node(LOOP); strncpy(loop->ast_loop.loop_type, name,
MAXIMUM_IDENTIFIER_LENGTH); loop->
    ast_loop.expression = first; loop->
    ast_loop.statement = second; return
    loop;
}
```



```
}
```

```
struct ast_node *make_block(struct ast_node
    *node) { struct ast_node *block =
    make_node(BLOCK); block-
    >ast_block.block_items = node; return block;
}
```

```
struct ast_node * make_branch_node(struct ast_node
*expression, struct ast_node
*statement1, struct ast_node *statement2) {
    struct ast_node *branch = make_node(BRANCH);
    branch->ast_branch.if_expr = expression;
    branch->ast_branch.if_statement =
    statement1; branch-
    >ast_branch.else_statement = statement2;
    return branch;
}
```

```
struct ast_node *make_function_signature(struct ast_node
*ident, struct ast_node *first, struct ast_node *second) {
    struct ast_node *signature = make_node(FUNC_SIGN);
    signature->ast_function_signature.ident = ident; signature-
    >ast_function_signature.args = first; signature-
    >ast_function_signature.type_ref = second; return
    signature;
}
```

```
struct ast_node *make_source(struct ast_node *source_node,
struct ast_node *source_item) { struct ast_node *source =
make_node(SOURCE); source->ast_source.source = source_node;
source->ast_source.source_item = source_item; return source;
}
```

```
struct ast_node *make_value_node(char *type, char *value) {
    struct ast_node *value_node = make_node(VALUE);
    strncpy(value_node->ast_value.value, value,
    MAXIMUM_IDENTIFIER_LENGTH); strncpy(value_node-
    >ast_identifier.name, type,
```

```

MAXIMUM_IDENTIFIER_LENGTH);
    return value_node;
}

struct ast_node *make_type_node(char *name) {
    struct ast_node *value_node =
        make_node(TYPE_NODE); strncpy(value_node-
        >ast_type.type_name, name,
        MAXIMUM_IDENTIFIER_LENGTH);
    return value_node;
}

struct ast_node *make_ident_node(char *name) {
    struct ast_node *value_node =
        make_node(IDENTIFIER); strncpy(value_node-
        >ast_identifier.name, name,
        MAXIMUM_IDENTIFIER_LENGTH);
    return value_node;
}

```

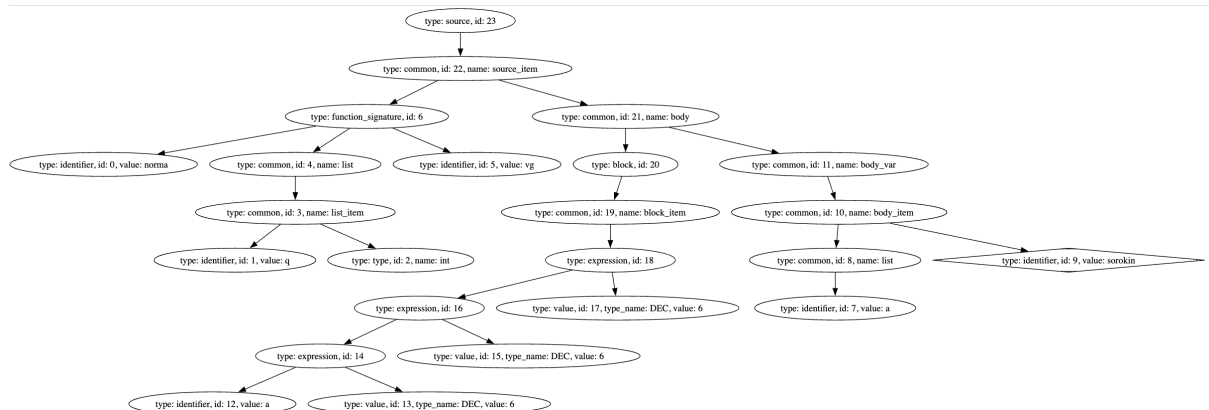
После этого дерево обходится в глубину и выводится.

Пример

```

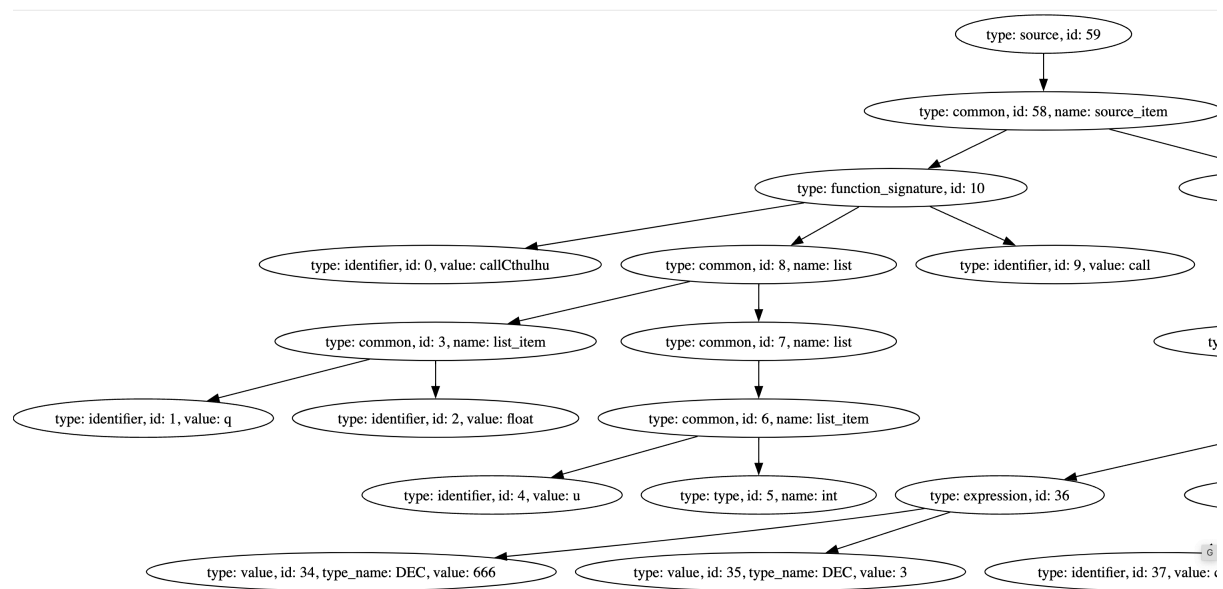
method norma(q:int):vg
    var a:sorokin;
    begin
        a:=6+6+6;
    end;

```



Чтобы граф поместился целиком в отчет был использован небольшой пример. В репозитории также находится tests/test1.txt – приведу сам пример и небольшой кусочек графа:

```
method callCthulhu(q:float,u:int):call
  var a,b,c:god; d,e:ancient; f:array[] of int;
  begin
    if 666>3 then callCthulhu(1, l[7]);
    else a:=6+6+6;
  end;
```



Вывод

Я изучил bison и lex и построил абстрактное синтаксическое дерево для алгоритма на основе грамматики по варианту.