Пример реализации синтаксического анализатора

Гаврилов Владимир Сергеевич

5 ноября 2017 г.

Содержание

1. Введение

В данном тексте приводится пример реализации синтаксического анализатора, а именно реализации анализатора для варианта №5 по построению лексического анализатора. Студентам, у которых другой номер варианта, нужно видоизменить для своего случая то, что приводится ниже.

2. Синтаксический анализатор для варианта №5

Условимся, что все заголовочные файлы (а именно, файлы с расширением `.h`) располагаются в каталоге `include`, а файлы реализации (файлы с расширением `.cpp`) — в каталоге `src`. При этом условимся также, что написанный синтаксический анализатор будет порождать расположенное в памяти синтаксическое дерево обрабатываемой программы.

2.1. Абстрактное синтаксическое дерево

Прежде всего опишем тип абстрактного синтаксического дерева. Это описание будет располагаться в файле `ast.h`.

```
// Файл ast.h.
#ifndef AST_H
#define AST_H
#include <vector>
#include <cstddef>
namespace AST{
    struct Node{
        Node()            = default;
        Node(const Node&) = default;
        virtual ~Node()   = 0;
    };

    struct Program : public Node{
        std::vector<Node*> children;

        Program()               = default;
        Program(const Program&) = default;
        virtual ~Program();
```

```
};

enum class Simple_type_kind : size_t{
    Integer, Float, Logic, Char, String, Void
};

struct Type : public Node{
    Type()            = default;
    Type(const Type&) = default;
    virtual ~Type()   = 0;
};

struct Expr : public Node{
    Expr()            = default;
    Expr(const Expr&) = default;
    virtual ~Expr()   = 0;
};

struct Simple_type: public Type{
    Simple_type_kind kind;

    Simple_type()                    = default;
    Simple_type(const Simple_type&) = default;
    virtual ~Simple_type();
};

struct Array_type : public Type{
    std::vector<Expr*> dimensions;
    Type*              elem_type;

    Array_type()                   = default;
    Array_type(const Array_type&) = default;
    virtual ~Array_type();
};

struct Variable : public Node{
    size_t name_idx;
    Type*  type;

    Variable()                = default;
    Variable(const Variable&) = default;
    virtual ~Variable();
};

enum class Parameter_kind : size_t{
    Value, Reference, Const_reference
};

struct Parameter : public Node{
    Parameter_kind kind;
    size_t         name_idx;
    Type*          type;

    Parameter()                 = default;
    Parameter(const Parameter&) = default;
    virtual ~Parameter();
};
```

```
struct Prototype : public Node{
    size_t                  name_idx;
    std::vector<Parameter*> params;
    Type*                   ret_val;

    Prototype()                 = default;
    Prototype(const Prototype&) = default;
    virtual ~Prototype();
};

struct Function : public Node{
    Prototype*         proto;
    std::vector<Node*> body;

    Function()                = default;
    Function(const Function&) = default;
    virtual ~Function();
};

enum class Unary_kind : size_t{
    Logical_not, Bitwise_not, Plus, Minus
};

struct Unary : public Expr{
    Unary_kind kind;
    Expr*      operand;

    Unary()             = default;
    Unary(const Unary&) = default;
    virtual ~Unary();
};

enum class Binary_kind : size_t{
    Logical_or,    Logical_orn,      Logical_xor,  Logical_xorn,  Logical_and,
    Logical_andn,  Less,             Greater,      Equal,         Not_equal,
    Less_or_equal, Greater_or_equal, Bitwise_or,   Bitwise_orn,   Bitwise_xor,
    Bitwise_xorn,  Bitwise_and,      Bitwise_andn, Left_shift,    Right_shift,
    Add,           Sub,              Mul,          Div,           Mod,
    Float_div,     Power,            Float_power,  Dimension_size
};

struct Binary : public Expr{
    Binary_kind kind;
    Expr*       left_op;
    Expr*       right_op;

    Binary()              = default;
    Binary(const Binary&) = default;
    virtual ~Binary();
};

struct Ternary : public Expr{
    Expr*       condition;
    Expr*       true_branch_op;
    Expr*       false_branch_op;
```

```
        Ternary()               = default;
        Ternary(const Ternary&) = default;
        virtual ~Ternary();
    };
}
#endif
```

Список литературы

[1] Ахо А., Лам М., Сети Р., Ульман Дж. Компиляторы: принципы, технологии и инструментарий. 2–е изд.: Пер. с англ. — М.: Вильямс, 2008.

[2] Хопкрофт Дж. Э., Мотвани Р., Ульман Дж. Введение в теорию автоматов, языков и вычислений. 2–е изд.: Пер. с англ. — М.: Вильямс, 2002.

[3] Фридл Дж. Регулярные выражения. 3–е изд: Пер. с англ. — СПб.: Символ-Плюс, 2008.

[4] Мозговой М.В. Классика программирования: алгоритмы, языки, автоматы, компиляторы. Практический подход. — СПб.: Наука и техника, 2006.

[5] Свердлов С.З. Языки программирования и методы трансляции. — СПб.: Питер, 2007.

[6] Вирт Н. Построение компиляторов. — М.: ДМК Пресс, 2014.

[7] Креншоу Дж. Давайте создадим компилятор! — М.: 1995.

[8] Кнут Д.Э. Искусство программирования для ЭВМ. Т.3. Сортировка и поиск.: Пер. с англ. — М.: Мир, 1978.

[9] Кнут Д.Э. Искусство программирования. Т.3. Сортировка и поиск. 2–е изд.: Пер. с англ. — М.: Вильямс, 2008.

[10] Ахо А., Хопкрофт Дж., Ульман Дж. Структуры данных и алгоритмы.: Пер с англ. — М.: Вильямс, 2000.