

Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей
Кафедра информатики
Дисциплина: Методы трансляции

ОТЧЕТ
к лабораторной работе №3
на тему

СИНТАКСИЧЕСКИЙ АНАЛИЗ

Студент

Т. П. Власенко

Преподаватель

Н. Ю. Гриценко

Минск 2024

СОДЕРЖАНИЕ

1 Цель работы	3
2 Краткие теоретические сведения.....	4
3 Результаты выполнения лабораторной работы.....	5
Выводы	8
Список использованных источников	9
Приложение А (обязательное) Листинг кода	10

1 ЦЕЛЬ РАБОТЫ

Целью выполнения данной лабораторной работы являются освоение работы с существующими синтаксическими анализаторами, разработка синтаксического анализатора подмножества языка программирования, определенного в лабораторной работе 1, определение синтаксических правил, построение дерева разбора, выявление синтаксических ошибок.

2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Данный код представляет собой синтаксический анализатор для языка программирования *Golang*. Лексический анализатор – часть программы, преобразующей входные данные (как правило, текст) в некий структурированный формат, нужный для задач последующего их (данных) анализа и использования [1]. Она обрабатывает массив токенов, полученных от лексического анализатора, и строит дерево разбора.

Программа начинается с определения набора типов вершин дерева и их подтипов для корректного построения дерева разбора.

Функция *parseFile* начинает обработку файла с получения списка подключаемых пакетов. Затем она обрабатывает объявления уровня пакета, рекурсивно обрабатывая внутренние объявления, выражения, вызовы и другое.

Принцип работы синтаксического анализатора построен на рекурсивном обходе массива токенов, полученного на этапе лексического анализа.

Для построения различных узлов дерева были разработаны соответствующие функции: *parseIdent*, *parseStmt*, *parseDecl*, *parseBlockStmt* и другие.

Также в разработанный синтаксический анализатор добавлен механизм сообщения пользователю об ошибках. При получении списка ошибок начинать исправления кода стоит с самой первой ошибки, так как ошибки, обнаруженные анализатором после первой, могут быть ложными.

Остальная часть кода содержит вспомогательные функции, которые необходимы для проведения анализа.

В результате работы программы на выходе синтаксический анализатор предоставляет массив глобальных объявлений, каждое из которых рекурсивно содержит в себе все остальные вершины дерева. Также на выходе получается массив ошибок, которые содержат в себе позицию ошибки и сообщение, поясняющее суть ошибки. Эти массивы можно использовать для дальнейшего анализа и обработки программного кода на языке *Golang*.

3 РЕЗУЛЬТАТЫ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

В ходе выполнения лабораторной работы был разработан лексический анализатор подмножества языка *Golang*. На рисунке 1 представлен тестовый код, который использовался для анализа разработанной программой:

```
package main

func main() {
    a := 4
    a++
    a := 5
    b := max(a, a)

    for i := 10; i < 20; i++ {
        a++
    }

    for b < 3 {
        b = 5
    }
    a b
}
```

Рисунок 1 – Тестовый код программы

На рисунке 2 можно увидеть результат работы анализатора, который строит дерево по массиву токенов, полученному от лексического анализатора, и ищет синтаксические ошибки. Если ошибок нет, то на экран выводится дерево разбора.

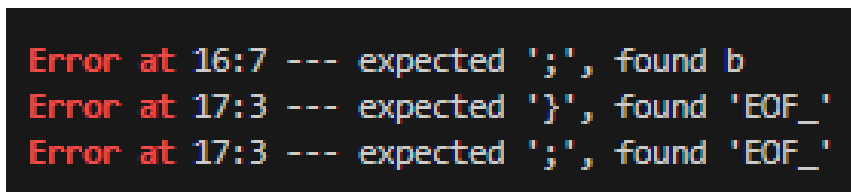
```

FuncDecl:
  IdentExpr: main
  FuncTypeExpr:
    FieldList:
  BlockStmt:
    AssignStmt:
      IdentExpr: a
      :=
      BasicLitExpr: 4
    IncDecStmt:
      ++
      IdentExpr: a
    AssignStmt:
      IdentExpr: a
      :=
      BasicLitExpr: 5
    AssignStmt:
      IdentExpr: b
      :=
      CallExpr:
        IdentExpr: max
        IdentExpr: a
        IdentExpr: a
    ForStmt:
      AssignStmt:
        IdentExpr: i
        :=
        BasicLitExpr: 10
      BinaryExpr:
        IdentExpr: i
        <
        BasicLitExpr: 20
      IncDecStmt:
        ++
        IdentExpr: i
      BlockStmt:
        IncDecStmt:
          ++
          IdentExpr: a
      ForStmt:
        BinaryExpr:
          IdentExpr: b
          <
          BasicLitExpr: 3
        BlockStmt:
          AssignStmt:
            IdentExpr: b
            =
            BasicLitExpr: 5

```

Рисунок 2 – Результат анализа тестового кода, если убрать последнюю строку (*a b*)

На рисунке 3 представлены ошибки, который выявил лексический анализатор в процессе анализа тестового кода.



```
Error at 16:7 --- expected ';', found b
Error at 17:3 --- expected '}', found 'EOF_'
Error at 17:3 --- expected ';', found 'EOF_'
```

Рисунок 3 – Синтаксические ошибки, найденные во время анализа тестового кода

Первая ошибка означает, что в коде вместо ожидаемой точки с запятой был обнаружен идентификатор *b*. Последующие ошибки можно игнорировать, так как они могут быть следствием первой ошибки.

В общем случае синтаксический анализатор может обработать множество видов ошибок: неожиданная запятая, когда ожидалась скобка, пропущенная точка с запятой в конце строки, отсутствие имен у типизированных параметров функции, пропущенный индекс в индексации среза, неверное объявление метки, объявление переменной с помощью ключевого слова *var* внутри условия *if*, отсутствие условия в *if*, неожиданная точка с запятой, неправильное имя пакета, пропуск операнда, несколько выражений вместо одного и другие.

Если после лексического анализа синтаксический анализатор обнаруживает неисправленные лексические ошибки, то дальнейший анализ проводится не будет. Некоторые ошибки не вызывают остановку анализа для более снисходительного проведения анализа.

Массив объявлений, сгенерированный синтаксически анализатором и представляющий собой дерево разбора, будет использован на этапе семантического анализа.

В данной главе была рассмотрена работа разработанного лексического анализатора языка программирования *Golang*.

ВЫВОДЫ

В ходе выполнения данной лабораторной работы был разработан синтаксический анализатор подмножества языка программирования *Golang*, который в результате своей работы получает массив специально определенного для глобальных объявлений типа данных, массив ошибок. Был продемонстрирован результат работы программы при обнаружении ошибок.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Синтаксический анализатор [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/Синтаксический_анализатор – Дата доступа: 22.02.2024

ПРИЛОЖЕНИЕ А

(обязательное)

Листинг кода

```
SP<FuncDecl> parser::parseFuncDecl() {
    pos_t pos = expect(token_type::FUNC);

    SP<FieldList> recv;
    if (tok_ == token_type::LPAREN) {
        recv = parseParameters(false).second;
    }

    auto ident = parseIdent();

    auto [tparams, params] = parseParameters(true);
    if (recv != nullptr && tparams != nullptr) {
        error(tparams->Opening, "method must have no type parameters");
        tparams = nullptr;
    }
    auto results = parseResult();

    SP<BlockStmt> body;
    switch (tok_) {
        case token_type::LBRACE:
            body = parseBody();
            expectSemi();
            break;

        case token_type::SEMICOLON:
            next();
            if (tok_ == token_type::LBRACE) {
                error(pos_, "unexpected semicolon or newline before {");
                body = parseBody();
                expectSemi();
            }
            break;

        default:
            expectSemi();
            break;
    }

    auto decl = std::make_shared<FuncDecl>(
        recv, ident, std::make_shared<FuncTypeExpr>(pos, tparams, params,
results), body);
    return decl;
}

SP<Decl> parser::parseDecl(std::unordered_map<token_type, bool> sync) {
    parseSpecFunction f;
    switch (tok_) {
        case token_type::IMPORT:
            f = std::bind(&parser::parseImportSpec, this,
std::placeholders::_1,
std::placeholders::_2);
            break;

        case token_type::CONST:
        case token_type::VAR:
```

```

        f = std::bind(&parser::parseValueSpec, this,
std::placeholders::_1,
                        std::placeholders::_2);
        break;

        case token_type::TYPE:
            f = std::bind(&parser::parseTypeSpec, this,
std::placeholders::_1,
                        std::placeholders::_2);
            break;

        case token_type::FUNC:
            return parseFuncDecl();
        default:
            pos_t pos = pos_;
            errorExpected(pos, "declaration");
            advance(sync);
            return std::make_shared<BadDecl>(pos, pos_);
    }

    return parseGenDecl(tok_, f);
}

void parser::printErrors() const {
    std::cout << '\n';
    for (const auto& e : errors_) {
        std::cout << color::error << "Error at " << color::reset <<
e.pos.line << ':' << e.pos.col
        << " --- " << e.msg << '\n';
    }
    std::cout << '\n';
}

void parser::parseFile() {
    if (!errors_.empty()) {
        return;
    }

    pos_t pos = expect(token_type::PACKAGE);
    auto ident = parseIdent();
    if (ident->Name == "_") {
        error(pos, "invalid package name _");
    }

    expectSemi();

    if (!errors_.empty()) {
        return;
    }

    while (tok_ == token_type::IMPORT) {
        decls_.push_back(parseGenDecl(
            token_type::IMPORT, std::bind(&parser::parseImportSpec, this,
std::placeholders::_1,
                                        std::placeholders::_2)));
    }

    auto prev = token_type::IMPORT;
    while (tok_ != token_type::EOF_) {
        if (tok_ == token_type::IMPORT && prev != token_type::IMPORT) {
            error(pos, "imports must appear before other declarations");
        }
        prev = tok_;
    }
}

```

```
        decls_.push_back(parseDecl(declStart));  
    }  
    printErrors();  
}
```