ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ
И ИНФОРМАТИКИ»

Расчетно-графическая работа

по дисциплине
«Компиляторные технологии»
на тему

Разработка семантического анализатора для языка программирования Cool

Выполнил студент Пеалкиви Даниил Яковлевич
Ф.И.О.

Группы ИС-242

Работу принял _____ профессор д.т.н. М.Г. Курносов
подпись

Защищена _____ Оценка _____

Новосибирск – 2025

ВВЕДЕНИЕ

В данной лабораторной работе разрабатывается семантический анализатор для языка программирования Cool (Classroom Object-Oriented Language). Семантический анализ является этапом компиляции, следующим за лексическим и синтаксическим анализом. Данная работа позволяет глубже понять принципы работы компиляторов и особенности реализации семантического анализа в объектно-ориентированных языках программирования.

1 Язык программирования Cool

1.1 Грамматика языка

Грамматика языка Cool (Classroom Object-Oriented Language) определяет синтаксические правила построения программ. Cool является объектно-ориентированным языком программирования с поддержкой классов, наследования и статической типизации. Грамматика описывается расширенной формой Бэкуса-Наура (EBNF), которая является усовершенствованной версией классической формы Бэкуса-Наура. EBNF добавляет дополнительные метасимволы для более компактного и читаемого описания грамматики.

Метасимволы EBNF:

- [ ] - опциональные элементы
- { } - повторяющиеся элементы
- | - альтернативы
- ( ) - группировка
- * - любое количество повторение

Эта грамматика описывает синтаксис языка Cool и используется для построения синтаксического анализатора. Использование EBNF делает грамматику более понятной и удобной для реализации, так как она точно описывает все возможные синтаксические конструкции языка и их взаимосвязи.

Пример грамматики языка COOL в формате EBNF:

```
Program ::= { Class }

Class ::= class TYPE [inherits TYPE] { { Feature } }

Feature ::= ID( [FormalList] ) : TYPE { Expr }    -- метод

        | ID : TYPE [<- Expr]                     -- атрибут

FormalList ::= (Formal ,)* Formal

Formal ::= ID : TYPE

Expr ::= ID <- Expr                               -- присваивание

      | ID( [ExprList] )                          -- вызов метода

      | Expr@TYPE.ID( [ExprList] )                -- статический вызов

      | if Expr then Expr else Expr fi            -- условный оператор

      | while Expr loop Expr pool                 -- цикл

      | { { Expr } }                              -- блок

      | let ID : TYPE [<- Expr] in Expr           -- let-выражение

      | case Expr of { Case } esac                -- case-выражение

      | new TYPE                                  -- создание объекта

      | isvoid Expr                               -- проверка на void

      | Expr + Expr                               -- сложение

      | Expr - Expr                               -- вычитание

      | Expr * Expr                               -- умножение

      | Expr / Expr                               -- деление

      | ~ Expr                                    -- отрицание

      | Expr < Expr                               -- меньше

      | Expr <= Expr                              -- меньше или равно

      | not Expr                                  -- лог. отр.

      | (Expr)                                    -- группировка

      | ID                                        -- идентификатор

      | integer                                   -- целое число
```

Рисунок 1.1 - Грамматика языка Cool в формате EBNF

1.2 Семантика основных конструкций

Семантика основных конструкций языка Cool определяет правила и ограничения, необходимые для обеспечения корректности программ, предсказуемости поведения и обнаружения ошибок на этапе компиляции.

Классы и наследования:

- Уникальные имена классов
- Запрет наследования от встроенных типов (`Int, Bool, String`)
- Запрет циклического наследования

Методы и атрибуты:

- Уникальные имена в пределах класса
- Инициализация атрибутов при объявлении
- Переопределение методов с сохранением сигнатуры

Типы и выражения:

- Статическая типизация
- Встроенные типы: Int, Bool, String
- Арифметические операции: операнды и результат типа Int

Области видимости:

- Атрибуты класса: видимы во всех методах
- Параметры методов: видимы только в теле метода
- Локальные переменные: видимы в своей области

Семантика Cool обеспечивает строгую типизацию и объектно-ориентированную структуру программ.

2 Структура компилятора

Компилятор Cool представляет собой многофазный процессор, который преобразует исходный код на языке Cool в промежуточное представление. Основная структура компилятора включает в себя последовательную цепочку обработки, где каждая фаза выполняет свою специфическую задачу и передает результаты следующей фазе.

Процесс компиляции начинается с лексического анализа, который разбивает исходный текст на последовательность токенов. Эти токены затем обрабатываются синтаксическим анализатором, который проверяет соответствие входного текста грамматике языка и строит абстрактное синтаксическое дерево (AST).

Следующей важной фазой является семантический анализ, который проверяет корректность типов, правильность иерархии классов, валидность вызовов методов и контролирует области видимости переменных. Эта фаза использует информацию, накопленную в предыдущих фазах, и обогащает AST дополнительной информацией о типах.

Все компоненты компилятора взаимодействуют через общие структуры данных, такие как абстрактное синтаксическое дерево, таблицы символов и типов. Каждая фаза может обнаруживать ошибки, которые накапливаются и не прерывают процесс компиляции, что позволяет выявить максимальное количество ошибок за один проход.

2.1 Лексический анализатор

Лексический анализатор (lexer) является первой фазой компиляции, которая преобразует исходный текст программы в последовательность токенов. В проекте лексический анализатор реализован с помощью инструмента flex в файле `cool.flex`.

Основная задача лексического анализатора - распознавание лексем языка Cool, таких как ключевые слова (`class, if, then, else`), идентификаторы, числа, строки, операторы и разделители. При этом он игнорирует пробелы, табуляции и комментарии, которые не несут семантической нагрузки.

Пример правил из файла `cool.flex`:

```
"class"     { return (CLASS); }
"if"        { return (IF); }
"then"      { return (THEN); }
"else"      { return (ELSE); }
"fi"        { return (FI); }
"while"     { return (WHILE); }
"loop"      { return (LOOP); }
"pool"      { return (POOL); }
"let"       { return (LET); }
"in"        { return (IN); }
"case"      { return (CASE); }
[A-Z][a-zA-Z0-9_]*  { return (TYPEID); }
[a-z][a-zA-Z0-9_]*  { return (OBJECTID); }
[0-9]+              { return (INT_CONST); }
```

Рисунок 2.1 - Набор правил лексического анализатора

2.2 Синтаксический анализатор

Синтаксический анализатор (parser) является второй фазой компиляции, которая проверяет соответствие последовательности токенов грамматике языка Cool и строит абстрактное синтаксическое дерево (AST). В проекте синтаксический анализатор реализован с помощью инструмента `bison` в файле `cool.bison`.

Основная задача синтаксического анализатора - проверка правильности структуры программы и построение дерева разбора, которое отражает иерархию синтаксических конструкций. При этом он обрабатывает правила грамматики, такие как определение классов, методов, атрибутов и различных типов выражений.

Пример правил из файла cool.bison:

```
program : class_list
        ;
class_list : class
           | class_list class
           ;
class : CLASS TYPEID '{' feature_list '}'
      | CLASS TYPEID INHERITS TYPEID '{' feature_list '}'
      ;
feature_list : feature
             | feature_list feature
             ;
```

Рисунок 2.2.1 - Набор правил синтаксического анализатора

Для обработки выражений используются правила:

```
expr : OBJECTID ASSIGN expr
     | OBJECTID '(' expr_list ')'
     | expr '@' TYPEID '.' OBJECTID '(' expr_list ')'
     | IF expr THEN expr ELSE expr FI
     | WHILE expr LOOP expr POOL
     | '{' expr_list '}'
     | LET OBJECTID ':' TYPEID IN expr
     | CASE expr OF case_list ESAC
     | NEW TYPEID
     | ISVOID expr
     | expr '+' expr
     | expr '-' expr
     | expr '*' expr
     | expr '/' expr
     | '~' expr
     | expr '<' expr
     | expr '<=' expr
     | expr '=' expr
     | NOT expr
     ;
```

Рисунок 2.2.2 - Набор правил обработки выражений анализатора

При обнаружении синтаксических ошибок (например, отсутствие закрывающей скобки или неправильный порядок токенов) синтаксический анализатор генерирует сообщения об ошибках и пытается восстановиться для продолжения анализа. Построенное AST передается следующей фазе компиляции - семантическому анализатору.

2.3 Семантический анализатор

Семантический анализатор - это фаза компиляции, которая проверяет смысловую корректность программы, построенной на основе абстрактного синтаксического дерева (AST). В проекте семантический анализ реализован в файле `semantic-phase.cc`.

Основные задачи семантического анализатора:

- Проверка корректности иерархии классов (отсутствие циклов, запрет наследования от встроенных типов)
- Проверка уникальности имен классов, методов и атрибутов
- Контроль областей видимости переменных, параметров и атрибутов
- Проверка совместимости типов в выражениях, операциях и возвращаемых значениях методов
- Проверка корректности вызовов методов (существование метода, правильное количество и типы аргументов)
- Обработка специальных случаев (использование `SELF_TYPE`, ключевого слова `self`, инициализация атрибутов)

Пример кода проверок из файла `semantic-phase.cc`:

Уникальность названий классов:

```
1. for (int i = parse_results->first(); parse_results->more(i);
2.     i = parse_results->next(i)) {
3.  class__class *current_class =
4.      dynamic_cast<class__class *>(parse_results->nth(i));
5.  std::string class_name = semantic::getName(current_class);
6.
7.  auto result = classes_names.insert(class_name);
8.  if (!result.second) {
9.    semantic::error("class '" + std::string(class_name) +
10.                      "' already defined in scope");
11.    }
```

Рисунок 2.3.1 - Проверка уникальности названий классов

Существование класса с методом `main()`:

```
1. bool has_main_method = false;
2. for (int i = parse_results->first(); parse_results->more(i); i =
   parse_results->next(i)) {
3.  class__class *current_class = dynamic_cast<class__class
   *>(parse_results->nth(i));
4.  Features features = semantic::getFeatures(current_class);
5.  for (int j = features->first(); features->more(j); j = features->next(j))
   {
6.    Feature current_feature = features->nth(j);
7.    if (current_feature->get_feature_type() == "method_class" &&
8.        semantic::getName(current_feature) == "main") {
9.      has_main_method = true;
10.       break;
11.     }
```

```
12.     }
13.     if (has_main_method) break;
14.   }
15.   if (!has_main_method) {
16.     semantic::error("no class with method 'main' found");
17.   }
```

Рисунок 2.3.2 - Проверка на существование класса с методом `main()`

Уникальность названий полей, методов в классе:

```
1. for (int j = features->first(); features->more(j); j = features->next(j))
   {
2.  Feature current_feature = features->nth(j);
3.  std::string feature_name = semantic::getName(current_feature);
4.
5.  if (feature_name == "self") {
6.    semantic::error("failed to use 'self' as feature name");
7.  }
8.
9.  result = features_names.insert(feature_name);
10.    if (!result.second) {
11.      semantic::error("feature '" + std::string(feature_name) + "' in '" +
12.                      class_name + "' already defined in scope");
13.    }
14.    // ...
15.  }
```

Рисунок 2.3.3 - Проверка уникальности названий полей, методов

Уникальность названий параметров методов:

```
1. for (int k = formals->first(); formals->more(k); k = formals->next(k)) {
2.  Formal_class *current_formal = dynamic_cast<formal_class
```

```
   *>(formals->nth(k));
3.  std::string formal_name = semantic::getName(current_formal);
4.
5.  if (formal_name == "self") {
6.    semantic::error("failed to use 'self' as formal name");
7.  }
8.
9.  result = formals_names.insert(formal_name);
10.    if (!result.second) {
11.      semantic::error("formal '" + std::string(formal_name) + "' in '" +
12.                      feature_name + "' already defined in scope");
13.    }
14.    // ...
15.   }
```

Рисунок 2.3.4 - Проверка уникальности названий параметров

Существование переменных, которые используются в выражениях:

```
1. if (e->get_expr_type() == "object_class") {
2.  std::string var_name = getName(e);
3.  if (var_name != "self" &&
4.      attr_to_type.find(var_name) == attr_to_type.end() &&
5.      formal_to_type.find(var_name) == formal_to_type.end()) {
6.    error("variable '" + var_name + "' not defined in scope");
7.    return;
8.  }
9. }
10.
11.   else if (e->get_expr_type() == "object_class") {
12.    std::string var_name = getName(e);
13.    if (var_name == "self") {
```

```
14.       caller_type = "self";
15.     } else if (attr_to_type.find(var_name) != attr_to_type.end()) {
16.       caller_type = attr_to_type[var_name];
17.     } else if (formal_to_type.find(var_name) != formal_to_type.end()) {
18.       caller_type = formal_to_type[var_name];
19.     } else {
20.       error("variable '" + var_name + "' not defined in scope");
21.       return;
22.     }
23.   }
```

Рисунок 2.3.5 - Проверка существований переменных

Проверка уникальности имени переменной в области видимости:

```
1. else if (expr_type == "let_class") {
2.  std::string formal_name = semantic::getName(expr);
3.
4.  if (formal_name == "self") {
5.    semantic::error("can't use 'self' as new local variable name");
6.  }
7.
8.  auto result = formals_names.insert(formal_name);
9.  if (!result.second) {
10.       semantic::error("formal '" + formal_name + "' already defined in
   scope");
11.     }
12.     // ...
13.   }
14.
15.   for (int k = formals->first(); formals->more(k); k = formals->next(k))
   {
```

```
16.    Formal_class *current_formal = dynamic_cast<formal_class
   *>(formals->nth(k));
17.    std::string formal_name = semantic::getName(current_formal);
18.
19.    if (formal_name == "self") {
20.      semantic::error("failed to use 'self' as formal name");
21.    }
22.
23.    result = formals_names.insert(formal_name);
24.    if (!result.second) {
25.      semantic::error("formal '" + std::string(formal_name) + "' in '" +
26.                      feature_name + "' already defined in scope");
27.    }
28.    // ...
29.   }
```

Рисунок 2.3.6 - Проверка уникальности имени переменной

Проверка на существование всех классов предков:

```
1. if (non_inherited.find(parent_name) != non_inherited.end()) {
2.  semantic::error("failed to use parent class '" + parent_name + "' for
   class '" +
3.                  class_name + "' (builtin)");
4. }
5.
6. if (std::string(parent_name) != "Object") {
7.  class__class *parent = semantic::FindClass(parent_name, parse_results);
8.
9.  if (parent) {
10.      Features parent_features = semantic::getFeatures(parent);
11.      // ...
```

```
12.     } else {
13.       semantic::error("parent class '" + parent_name + "' of class '" +
14.                       class_name + "' is not defined");
15.     }
16.   }
```

Рисунок 2.3.7 - Проверка существования всех классов предков

Проверка на отсутствие циклов в графе наследования:

```
1. bool detect_cycle(STable hierarchy) {
2.  SSet visited;
3.  SSet currentlyVisiting;
4.  std::function<bool(const std::string &)> dfs =
5.      [&](const std::string &className) {
6.        if (visited.find(className) != visited.end()) {
7.          return false;
8.        }
9.        // ...
10.   if (semantic::detect_cycle(classes_hierarchy)) {
11.    semantic::error("loop detected in classes inheritance hierarchy");
12.    for (auto p : classes_hierarchy) {
13.      std::cerr << '\t' << p.first << " : " << p.second << "\n";
14.    }
15.   }
```

Рисунок 2.3.8 - Проверка отсутствия циклов в графе наследования

Проверка выражений на совместимость типов:

```
1. void checkExpression(Expression expr, STable &attr_to_type, STable
   &formal_to_type,
2.                      SSet &classes_names, SSet &formals_names, FeaturesTable
   &classes_features) {
```

```
3.  std::string expr_type = expr->get_expr_type();
4.
5.  if (expr_type == "plus_class" || expr_type == "sub_class" ||
6.      expr_type == "mul_class" || expr_type == "divide_class") {
7.    // ...
8.    if (!is_int_type(expr)) {
9.      error("non-integer value in arithmetic operation");
10.      }
11.    }
12.    else if (expr_type == "dispatch_class") {
13.      // ...
14.      if (is_builtin_type(expr)) {
15.        error("Class '" + get_type(expr) + "' has no method '" +
   get_method(expr) + "'");
16.      }
17.    }
18.    else if (expr_type == "lt_class" || expr_type == "leq_class") {
19.      // ...
20.      if (!is_int_type(expr)) {
21.        error("non-integer value in less-based compare operation");
22.      }
23.    // ...
```

Рисунок 2.3.9 - Проверка совместимости типов

3 Тестирование компилятора

Для тестирования компилятора были разработаны тестовые примеры, охватывающие различные аспекты семантического анализа. Тесты проверяют корректность обработки ошибок в следующих случаях: уникальность имен классов, полей и методов, правильность наследования классов, совместимость типов в выражениях, существование переменных в области видимости, а также наличие метода `main` в программе. Каждый тест содержит пример кода с ожидаемой ошибкой, что позволяет убедиться в правильной работе семантического анализатора.

3.1 Набор тестовых приложений

`inheritance.cl:`

```
class A inherits B {

};

class A inherits B { };
class B inherits C { };
class C inherits A { };
```

3.1.1 - Тестовое приложение корректности наследования

`type.cl:`

```
class A {
    main() : Int {
        1 + "hello"
  };

    niam() : Int {
        2.fun()
  };
};
```

3.1.2 - Тестовое приложение совместимости типов

unique.cl:

```
class A {

};
class A {

};

(*class A {
    x : Int;
      x : String;

      main() : Int { 1 };
      main() : String { "hello" };
  };*)

  (*class A {
      main(x : Int, x : String) : Int {
          1
      };
  };*)

```

3.1.3 - Тестовое приложение уникальности имен

var_scope.cl:

```
class A
{
    main(): Int {
      x + 1
    };
};

(*class A
{
      main(): Int {
      {
        let x: Int <- 1 in true;
        let x: String <- "hello" in true;
      }
      };
  };*)
```

3.1.4 - Тестовое приложение области видимости переменных

3.2 Среда тестирования

- Операционная система: Ubuntu (WSL2)
- Версия ядра: Linux 5.15.167.4-microsoft-standard-WSL2
- Компилятор: `g++`
- Сборка проекта: `./build.sh`
- Запуск тестов: `./analyzer tests/file.cl`

3.3 Результаты

`inheritance.cl`:

```
itzavangard@DESKTOP-SOMTGK4:~/CT/lab05$ ./analyzer tests/inheritance.cl
Semantic error: parent class 'B' for class 'A' is not defined
Semantic error: no class with method 'main' found
Semantic error: loop detected in classes inheritance hierarchy
        B : C
        C : A
        A : B
Semantic phase: 3 errors
```

3.3.1 - Вывод тестового приложения корректности наследования

`type.cl`:

```
itzavangard@DESKTOP-SOMTGK4:~/CT/lab05$ ./analyzer tests/type.cl
Semantic error: non-integer value string_const_class in arithmetic operation
Semantic error: Class 'Int' has no method 'fun'
Semantic phase: 2 errors
```

3.3.2 - Вывод тестового приложения совместимости типов

`unique.cl`:

```
itzavangard@DESKTOP-SOMTGK4:~/CT/lab05$ ./analyzer tests/unique.cl
Semantic error: class 'A' already defined in scope
Semantic error: feature 'x' in 'A' already defined in scope
Semantic error: feature 'main' in 'A' already defined in scope
Semantic error: formal 'x' in 'main' already defined in scope
Semantic phase: 4 errors
```

3.3.3 - Вывод тестового приложения уникальности имен

`var_scope.cl`:

```
Semantic error: variable 'x' not defined in scope
Semantic error: formal 'x' already defined in scope
Semantic phase: 2 errors
```

3.3.4 - Тестовое приложение области видимости переменных

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы был разработан семантический анализатор для языка Cool. Реализованы все требуемые проверки, включая контроль уникальности имен классов, полей и методов, проверку области видимости переменных, валидацию графа наследования классов и контроль типов в выражениях.

В процессе работы были изучены и применены на практике принципы семантического анализа, методы проверки типов и контроля области видимости переменных. Полученный опыт может быть полезен при разработке компиляторов и других инструментов статического анализа кода.