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 № 4
на тему

СЕМАНТИЧЕСКИЙ АНАЛИЗАТОР

Студент П. Н. Носкович

Преподаватель Н. Ю. Гриценко

Минск 2025

СОДЕРЖАНИЕ

1 ПОСТАНОВКА ЗАДАЧИ

Цель данной лабораторной работы заключается в разработке семантического анализатора, который будет проверять корректность значений и типов данных в абстрактном синтаксическом дереве, полученном на предыдущем этапе. Особое внимание уделяется выявлению семантических ошибок, таких как несоответствие типов, использование неинициализированных переменных и неправильные операции над данными, а также формированию информативных сообщений об ошибках для облегчения отладки исходного кода.

2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Семантический анализ является важным этапом в процессе компиляции языков программирования, который следует за синтаксическим анализом. На этом этапе осуществляется проверка смысловой корректности программы, что включает в себя соответствие её контексту и правилам, установленным для данного языка. Основная задача семантического анализа – выявление ошибок, которые могут возникнуть при выполнении программы, даже если её синтаксис корректен [1].

Одной из ключевых задач семантического анализа является проверка типов. Это означает, что анализатор должен убедиться, что операции выполняются над совместимыми типами данных. Например, не допускается сложение строки и числа, так как это приведёт к ошибке выполнения. Кроме того, анализатор проверяет область видимости переменных и функций, чтобы гарантировать их корректное использование. Это включает в себя удостоверение в том, что переменные и функции инициализированы перед использованием и что они используются в пределах своей области видимости.

Семантический анализ также включает проверку семантики операторов, что предполагает оценку правильности использования различных операторов в контексте приложения. Например, оператор сравнения не может быть применён к объектам, которые не поддерживают такие операции. Анализ выражений является другой важной задачей, где проверяется, что все выражения имеют корректные значения и могут быть вычислены без ошибок.

Существует несколько методов семантического анализа, среди которых статический анализ. Этот метод выполняется на этапе компиляции и не требует выполнения программы, что позволяет выявить множество ошибок до запуска кода. Обратное связывание также является важным аспектом, который используется для связывания идентификаторов с их определениями, что помогает избежать неоднозначностей и ошибок [2].

По завершении семантического анализа формируется промежуточное представление кода, которое учитывает как синтаксическую, так и семантическую корректность. Это представление становится основой для дальнейших этапов компиляции, таких как оптимизация и генерация машинного кода, что в конечном итоге приводит к созданию работающей программы. Таким образом, семантический анализ играет ключевую роль в обеспечении корректности и надежности программного обеспечения.

3 РЕЗУЛЬТАТЫ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

Во время выполнения лабораторной работы был реализован семантический анализатор языка *PHP*. Скриншот результатов семантического анализа программы с ошибками представлен на рисунке 3.1.

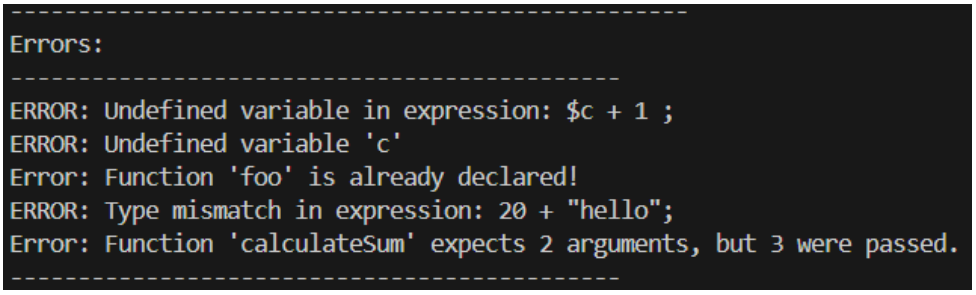

Рисунок 3.1 – Итог работы семантического анализатора

Изначальное содержание файла с анализируемым программным кодом представлено на рисунке 3.2.

```php
<?php
function calculateSum($num1, $num2) {
    $result = $num1 + $num2;
    return $result;
}

function foo(int $x) {
    $x = $c + 1 ;
}

function foo(int $x) {}

echo calculateSum(5, 10) . "\n";
$y = 20 + "hello";

$sum = calculateSum($x, $y, $Z);

$greeting = "Hello, ";
$subject = "world!";
$message = $greeting . $subject;

$a = 5;
$b = 3;
$product = $a * $b;

// Block with local scope
{
    $localVar = 100;
    $x = $localVar; // Valid (reassigns global $x inside block)
}

// Valid output (optional)
echo $message . " Sum: " . $sum . ", Product: " . $product;
?>
```

Рисунок 3.2 – Изначальное содержание текстового файла

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы по написанию семантического анализатора были изучены основные принципы семантического анализа, а также методы проверки корректности значений и типов данных в абстрактном синтаксическом дереве. Были реализованы алгоритмы для выявления семантических ошибок, таких как несоответствие типов, использование неинициализированных переменных и неправильные операции над данными. Семантический анализатор был настроен на формирование информативных сообщений об ошибках, что позволяет облегчить отладку исходного кода и повысить надежность программного обеспечения.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Синтаксический анализатор [Электронный ресурс]. – Режим доступа: https://edu.vsu.ru/mod/resource/view.php?id=25354 – Дата доступа: 04.03.2024

[2] Введение в теорию компиляторов [Электронный ресурс]. – Режим доступа: https://habr.com/ru/articles/515420/ – Дата доступа: 05.03.2024

ПРИЛОЖЕНИЕ А
(обязательное)
Листинг программного кода

```
MODULE SymbolTable
```

```
  IMPLICIT NONE
  INTEGER, PARAMETER :: max_symbols = 1000
  INTEGER, PARAMETER :: max_functions = 50
  INTEGER, PARAMETER :: max_scope_depth = 10
  INTEGER, PARAMETER :: max_errors = 100

  TYPE Symbol
    CHARACTER(LEN=32) :: name
    CHARACTER(LEN=10) :: type
    CHARACTER(LEN=10) :: scope
    INTEGER :: scope_level
  END TYPE Symbol

  TYPE Function
    CHARACTER(LEN=32) :: name
    CHARACTER(LEN=10) :: return_type
    INTEGER :: arg_count
    CHARACTER(LEN=10), DIMENSION(10) :: arg_types
  END TYPE Function

  TYPE Class
    CHARACTER(LEN=32) :: name
  END TYPE Class

  INTEGER, PARAMETER :: max_classes = 50
  TYPE(Class), DIMENSION(max_classes) :: classes
  INTEGER :: class_count = 0

  TYPE(Symbol), DIMENSION(max_symbols) :: symbols
  TYPE(Function), DIMENSION(max_functions) :: functions
  CHARACTER(LEN=256), DIMENSION(max_errors) :: errors
  INTEGER :: symbol_count = 0
  INTEGER :: function_count = 0
  INTEGER :: current_scope_level = 0
  INTEGER :: error_count = 0
  INTEGER :: current_function_index = 0
  LOGICAL :: expected_braces = .FALSE.
  LOGICAL :: inside_function = .FALSE.
  LOGICAL :: inside_class = .FALSE.

CONTAINS
  FUNCTION CleanVarName(raw_name) RESULT(clean_name)
    CHARACTER(LEN=*), INTENT(IN) :: raw_name
    CHARACTER(LEN=32) :: clean_name
    INTEGER :: i, j

    clean_name = ADJUSTL(raw_name)

    IF (clean_name(1:1) == '$') THEN
      clean_name = clean_name(2:)
      clean_name = ADJUSTL(clean_name)
    END IF

    j = LEN_TRIM(clean_name)
    DO WHILE (j > 0 .AND. (clean_name(j:j) == ';' .OR. clean_name(j:j) == '
'))
      j = j - 1
    END DO
    clean_name = clean_name(1:j)
  END FUNCTION CleanVarName

  SUBROUTINE AddSymbol(name, type, scope)
    CHARACTER(LEN=*), INTENT(IN) :: name, type, scope
    CHARACTER(LEN=32) :: cleaned_name
```

```
    cleaned_name = CleanVarName(name)

    IF (symbol_count < max_symbols) THEN
      symbol_count = symbol_count + 1
      symbols(symbol_count)%name = cleaned_name
      symbols(symbol_count)%type = type
      symbols(symbol_count)%scope = scope
      symbols(symbol_count)%scope_level = current_scope_level
      PRINT *, "Added symbol: ", TRIM(cleaned_name), " Type:", TRIM(type), "
Scope:", TRIM(scope), " Level:", current_scope_level
    ELSE
      CALL AddError("Error: Symbol table overflow!")
    END IF
  END SUBROUTINE AddSymbol

  FUNCTION GetSymbolType(name) RESULT(sym_type)
    CHARACTER(LEN=*), INTENT(IN) :: name
    CHARACTER(LEN=10) :: sym_type
    CHARACTER(LEN=32) :: cleaned_name
    INTEGER :: i

    cleaned_name = CleanVarName(name)
    sym_type = "undefined"

    DO i = 1, symbol_count
      IF (TRIM(symbols(i)%name) == TRIM(cleaned_name)) THEN
        IF (symbols(i)%scope == "global" .AND. current_scope_level == 0) THEN
          sym_type = symbols(i)%type
          RETURN
        END IF
        IF (symbols(i)%scope == "local" .AND. symbols(i)%scope_level <=
current_scope_level) THEN
          sym_type = symbols(i)%type
          RETURN
        END IF
      END IF
    END DO
  END FUNCTION GetSymbolType
  ! Добавление ошибки в список
  SUBROUTINE AddError(error_message)
    CHARACTER(LEN=*), INTENT(IN) :: error_message
    IF (error_count < max_errors) THEN
      error_count = error_count + 1
      errors(error_count) = error_message
    ELSE
      PRINT *, "Error: Too many errors! Cannot add more."
    END IF
  END SUBROUTINE AddError

  ! Вывод таблицы символов
  SUBROUTINE PrintSymbolTable()
    INTEGER :: i
    CHARACTER(LEN=50) :: fmt

    fmt = "(A10, A10, A10, I10)"

    DO i = 1, symbol_count
      WRITE(*, fmt) TRIM(symbols(i)%name), TRIM(symbols(i)%type),
TRIM(symbols(i)%scope), symbols(i)%scope_level
    END DO

    PRINT *, "--------------------------------------------------"
  END SUBROUTINE PrintSymbolTable
```

```
  ! Вывод списка ошибок
  SUBROUTINE PrintErrors()
    INTEGER :: i

    IF (error_count > 0) THEN
      PRINT *, "Errors:"
      PRINT *, "-------------------------------------------"
      DO i = 1, error_count
        PRINT *, TRIM(errors(i))
      END DO
      PRINT *, "-------------------------------------------"
    ELSE
      PRINT *, "Semantic analysis completed successfully!"
    END IF
  END SUBROUTINE PrintErrors


END MODULE SymbolTable

PROGRAM SemanticAnalyzer
  USE SymbolTable
  IMPLICIT NONE
  CHARACTER(LEN=256) :: line
  CHARACTER(LEN=256), DIMENSION(1000) :: lines
  INTEGER :: num_lines = 0, ios

  OPEN(UNIT=10, FILE='INPUT.TXT', STATUS='OLD', ACTION='READ', IOSTAT=ios)
  IF (ios /= 0) THEN
    PRINT *, "Error: Unable to open file INPUT.TXT"
    STOP
  END IF

  PRINT *, "File opened successfully!"

  ! Чтение всех строк из файла
  DO
    READ(10, '(A)', IOSTAT=ios) line
    IF (ios /= 0) EXIT
    num_lines = num_lines + 1
    lines(num_lines) = line
  END DO

  CLOSE(10)

  ! Обработка всех строк
  CALL ProcessCodeBlock(lines, num_lines)

  ! Вывод таблицы символов
  CALL PrintSymbolTable()

  ! Вывод ошибок (если есть)
  CALL PrintErrors()

END PROGRAM SemanticAnalyzer
```