|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ | | | | |
| ФГБОУ ВО «Пермский государственный национальный исследовательский университет» | | | | |
| ОТЧЁТ  по компилятору языка Pascal по дисциплине «Формальные грамматики и методы трансляции» | | | | |
| Работу выполнили  студенты гр. ПМИ-2  Долгирев В.Д. \_\_\_\_\_\_\_\_\_\_  (подпись) | |  | Проверил  преподаватель кафедры МОВС  Пономарёв Ф.А. \_\_\_\_\_\_\_\_\_\_  (подпись)  «\_\_\_» \_\_\_\_\_\_\_\_\_\_ 2022 | |
|  |  | | |  |
|  | Пермь 2022 | | |  |

ОГЛАВЛЕНИЕ

[ОТЧЁТ 1](#_Toc122052252)

[ОГЛАВЛЕНИЕ 2](#_Toc122052253)

[ПОСТАНОВКА ЗАДАЧИ 3](#_Toc122052254)

[ОБЩАЯ ИНФОРМАЦИЯ О КОМПИЛЯТОРЕ 4](#_Toc122052255)

[СТРУКТУРА КОМПИЛЯТОРА 4](#_Toc122052256)

[МОДУЛЬ I/O 6](#_Toc122052257)

[КОНЕЧНЫЙ АВТОМАТ МОДУЛЯ I/O 6](#_Toc122052258)

[ЛЕКСИЧЕСКИЙ АНАЛИЗАТОР 9](#_Toc122052259)

[СИНТАКСИЧЕСКИЙ АНАЛИЗАТОР 11](#_Toc122052260)

[СЕМАНТИЧЕСКИЙ АНАЛИЗАТОР 13](#_Toc122052261)

[НЕЙТРАЛИЗАЦИЯ 15](#_Toc122052262)

[НЕЙТРАЛИЗАЦИЯ ЛЕКСИЧЕСКИХ ОШИБОК 15](#_Toc122052263)

[НЕЙТРАЛИЗАЦИЯ СИНТАКСИЧЕСКИХ ОШИБОК 15](#_Toc122052264)

[НЕЙТРАЛИЗАЦИЯ СЕМАНТИЧЕСКИХ ОШАБОК 15](#_Toc122052265)

[ГЕНЕРАЦИЯ КОДА 16](#_Toc122052266)

[LLVM IR 16](#_Toc122052267)

[СБОРКА 16](#_Toc122052268)

[СТРУКТУРА 16](#_Toc122052269)

ПОСТАНОВКА ЗАДАЧИ

В рамках курса «Формальные грамматики и методы трансляции» необходимо было разработать компилятор языка Pascal.

Компилятор обязан включать в себя 4 модуля:

1. Модуль I/O
2. Лексический анализатор
3. Синтаксический анализатор
4. Семантический анализатор

Дополнительно, требовалось реализовать Генератор кода.

Согласно варианту, компилятор должен обрабатывать следующие языковые конструкции:

* Раздел описание констант, типов, переменных и операторов
* Обработку конструкций If и while
* Обработку выражений
* Обработку составных операторов
* Индексированные переменные
* Интервальные и перечислимые типы

ОБЩАЯ ИНФОРМАЦИЯ О КОМПИЛЯТОРЕ

Компилятор языка Pascal написан на языке C++20. Разработка велась при помощи современных возможностей языка C++. В программе довольно обширно используются умные указатели, итераторы, шаблоны. В отдельных случаях используются делегаты, а также функциональные операторы.

Реализованный компилятор является однопроходным, т.е. проверяет программу за одно чтение исходного кода программы.

СТРУКТУРА КОМПИЛЯТОРА

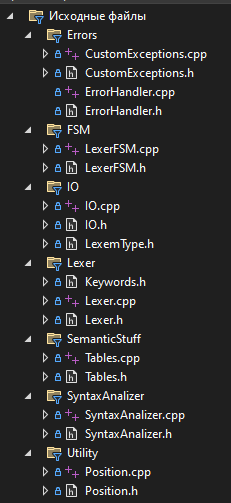
Компилятор состоит из 4 крупных модулей:

Рисунок 1. Структура программы.

1. Модуль I/O.
2. Лексический анализатор.
3. Синтаксический/семантический анализатор.
4. Обработчик ошибок.

В данной реализации семантический и синтаксический анализатор работают параллельно, поэтому их работа производится в рамках единого синтаксического модуля.

Для каждого модуля были разработаны вспомогательные модули. Так, для модуля I/O была написан конечный автомат, который помогал при парсинге исходного кода программы на языке Pascal, а для семантического анализа был разработан модуль работы с таблицами идентификаторов.

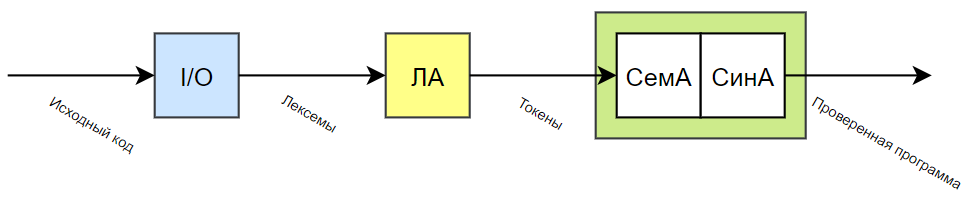
Так как компилятор является однопроходным, модули собраны в один конвейер. Полученное на входе слово преобразуется в токен, проверяется со стороны синтаксиса и семантики, после чего выдаётся вердикт: принимается данный символ или нет.

Рисунок 2. Вид конвейера анализаторов.

МОДУЛЬ I/O

Основная задача, возложенная на модуль ввода-вывода – считывание символов и формирование лексем (слов).

Считывание происходит посимвольно. Предполагалось, что достаточно будет рассматривать текущий символ, однако для определения оператора интервального типа данных, в определённых ситуациях требовалось рассматривать следующий идущий символ.

На выходе модуль выдавал лексему, которое шло далее на вход лексическому анализатору.

Для правильного считывания лексем был разработан конечный автомат.

КОНЕЧНЫЙ АВТОМАТ МОДУЛЯ I/O

Конечный автомат был разработан для помощи при считывании. В его задачи входят:

1. Определение допустимых символов
2. Обеспечение правильного считывания констант, операторов, идентификаторов, ключевых слов.
3. Обнаружение ошибок лексического характера
4. Классификация лексем

Конечный автомат состоит из 25 состояний: 19 рабочих состояний, 1 фиктивное состояние, 3 состояния ошибки, терминальное и стартовое состояние.

Каждое состояние содержит список состояний, в которое может перейти автомат. Для каждого такого перехода определён набор допустимых символов.

Состояния автомата отслеживаются Модулем I/O. После считывания определённого символа, Модуль отдаёт автомату считанный символ, а на его основе автомат переходит в следующее состояние.

После совершения перехода, Модуль смотрит на текущее состояние, и на его основе делает вывод: считывать следующий символ или нет, отдавать прочтённую лексему дальше по конвейеру или нет.

Если состояние является рабочим (Состояние с 1 по 19), то Модуль считывает символ и добавляет его к лексеме.

При достижении терминального состояния, автомат сбрасывает своё состояние до стартового, а Модуль I/O отдаёт прочтённую лексему Лексическому анализатору.

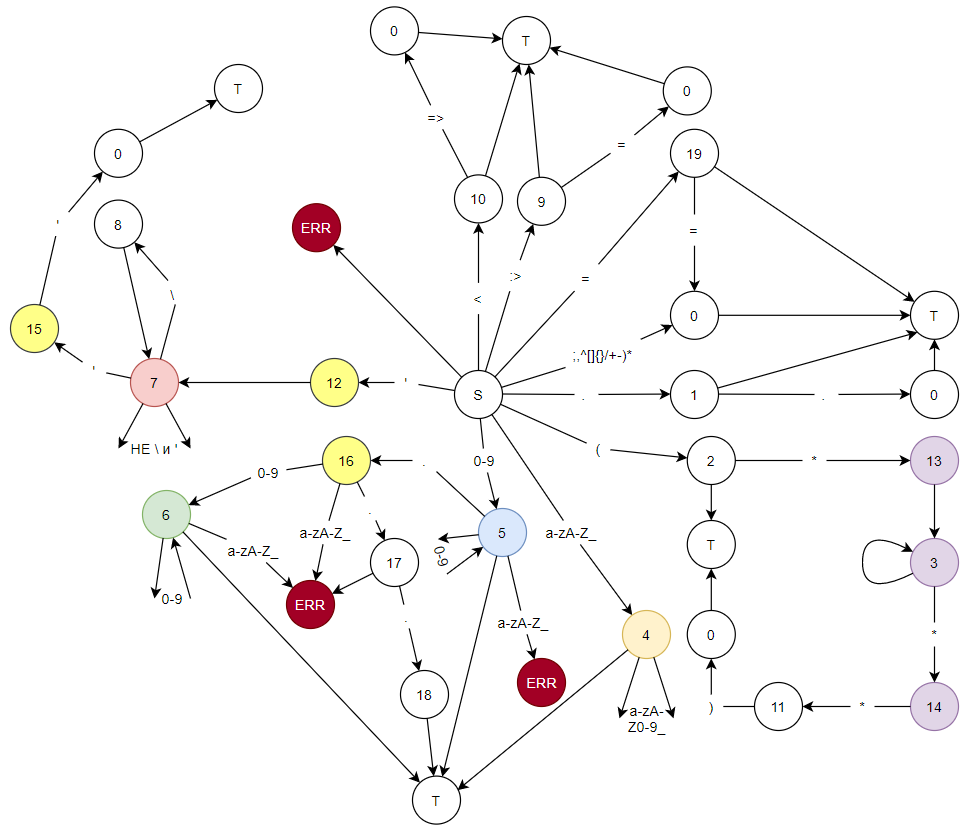
У автомата также определено 3 специальных состояния.

Рисунок 3. Структура конечного автомата.

На «Рисунке 3» можно увидеть, что существует нулевое (0) состояние. Оно сигнализирует Модулю, что следующий символ считывать не требуется. Само же по себе оно является неким буфером, между рабочим состоянием и терминальным для тех случаев, когда требуемые символы и остальное множество символов в отдельности переводят автомат в терминальное состояние. Например, если бы из 19 состояния оба перехода осуществлялись бы напрямую в терминальное состояние, то символ ‘=’ не считался, так как автомат бы находился в терминальном состоянии.

Состояния, отмеченные на «Рисунке 3» красным цветом, обозначают состояния ошибок. Таким образом, автомат распознаёт 3 вида ошибок:

* Некорректно записанное число
* Начало идентификатора с цифры
* Недопустимый символ

Если автомат переходит в данное состояние, то он кидает ошибку и сбрасывается, а Модуль I/O получает сигнал о том, что необходимо считывание до следующего пробела или конца строки.

Состояния, отмеченные на «Рисунке 3» жёлтым цветом, останавливают автомат, но не сбрасывают его. Они нужны тогда, когда требуются особые правила считывания. Например, цепочка состояний, следующая за состоянием 12, позволяет определять строковые значения. В данном случае, сначала требуется отдать символ кавычек далее по конвейеру, а только потом считать сроковую константу. Если сбросить автомат раньше времени, то начнётся считывание операторов, чисел, идентификаторов, что является неправильным.

Состояния, отмеченные другими цветами, помогают определять будущий тип лексемы. Это позволяет Лексическому анализатору быстро определять тип выдаваемого им токена.

Помимо этого, Модуль I/O хранит в себе текущую позицию считывания, которая в дальнейшем нужна для обработчика ошибок, для быстрого определения локации проблемы в коде.

ЛЕКСИЧЕСКИЙ АНАЛИЗАТОР

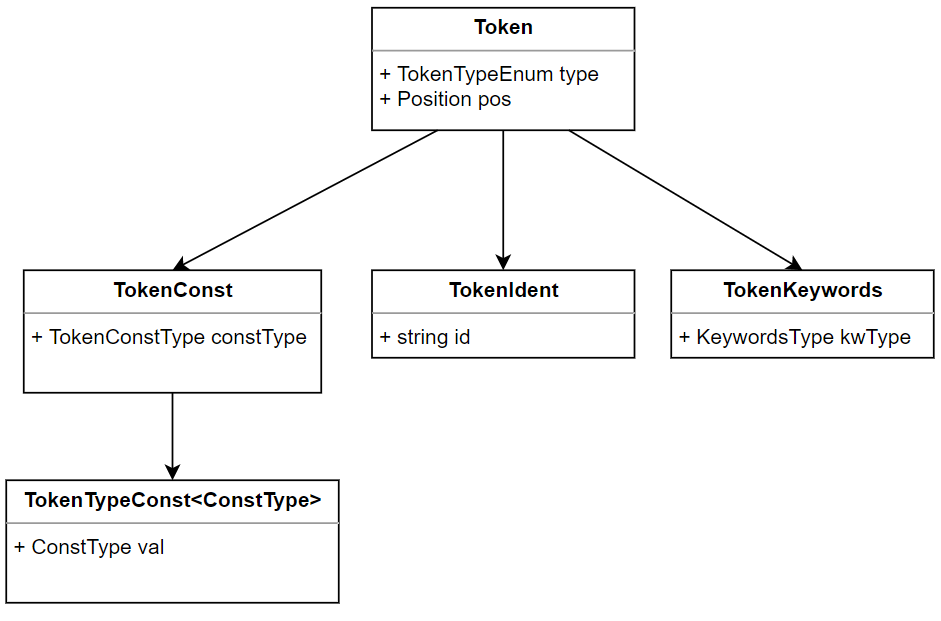
Задача лексического анализатора заключается в преобразовании лексем в токены. Так как конечный автомат, используемый в Модуле I/O, позволяет распознавать тип лексемы, анализатору остаётся только преобразовать лексему в токен, основываясь на типе лексемы.

Рисунок 4. Классы токенов.

Токены представляются несколькими классами.

* Базовый класс Token, хранящий позицию лексемы, на основе которой был создан токен.
* Класс TokenIdent, предназначенный для хранения идентификаторов.
* Класс TokenKeywords, хранящий тип ключевого слова.
* Классы TokenConst и TokenTypeComst, отвечающие за хранение значений констант.

Согласно заданию, требовалось на выходе из лексического анализатора выдавать токен, включающий преобразованную константу, если оная встретилась. Для этого, класс TokenTypeConst был написан при помощи шаблонов.

СИНТАКСИЧЕСКИЙ АНАЛИЗАТОР

В задачи синтаксического анализатора входит проверка синтаксических конструкций языка. На входе он принимает токены, который выдаёт лексический анализатор.

Синтаксический анализатор включен в класс Syntax, однако сам класс в процессе разработки больше стал классом-компилятором, в задачи которого входят синтаксический, семантический анализ, а в будущем и вызов генерации кода. Он также имеет тесную связь с Лексическим анализатором, так как запрашивает у него токены в процессе синтаксического анализа.

Компилятор поддерживает следующие БНФ конструкции:

<программа>::=**program** <имя>; <блок>.  
  
<блок>::=<раздел констант><раздел типов><раздел переменных><раздел операторов>  
  
<раздел констант>::=<пусто>|**const** <определение константы>;{<определение константы>;}  
  
<определение константы>::=<имя>=<константа>  
  
<константа>::=<число без знака>|<знак><число без знака>|<имя константы>|<знак><имя константы>|<строка>  
  
<число без знака>::=<целое без знака>|<вещественное без знака>  
  
<знак>::=**+**|**-**  
  
<имя константы>::=<имя>  
  
<строка>::=**’**<символ>{<символ>}**’**<раздел типов>::=<пусто>|type <определение типа>;{<определение типа>;}  
  
<определение типа>::=<имя>=<тип>  
  
<тип>::=<простой тип>|<составной тип>  
  
<простой тип>::=<перечислимый тип>|<ограниченный тип>|<имя типа>  
  
<перечислимый тип>::=(<имя>{,<имя>})  
  
<ограниченный тип>::=<константа>..<константа>  
  
<имя типа>::=<имя>  
  
<составной тип>::=<неупакованный составной тип>  
  
<неупакованный составной тип>::=<регулярный тип>  
  
<регулярный тип>::=array[<простой тип>{,<простой тип>}] of <тип компоненты>  
  
<тип компоненты>::=<тип>  
  
<раздел переменных>:<пусто>|var <описание однотипных переменныз>;{<описание однотипных переменныз>;}  
  
<описание однотипных переменных>::=<имя>{,<имя>}:<тип>  
  
<раздел операторов>::=<составной оператор>  
  
<оператор>::=<непомеченный оператор>  
  
<непомеченный оператор>::=<простой оператор>|<сложный оператор>  
  
<простой оператор>::=<оператор присваивания>  
  
<оператор присваивания>::=<переменная>:=<выражение>  
  
<переменная>::=<полная переменная>|<компонента переменной>  
  
<полная переменная>::=<имя переменной>  
  
<имя переменной>::=<имя>  
  
<компонента переменной>::=<индексированная переменная>  
  
<индексированная переменная>::=<переменная-массив>[<выражение>{,<выражение>}]  
  
<переменная-массив>::=<переменная>  
  
<выражение>::=<простое выражение>|<простое выражение><операция отношения><простое выражение>  
  
<операция отношения>::===|<>|<|<=|>=|>  
  
<простое выражение>::=<знак><слагаемое>{<аддитивная операция><слагаемое>}  
  
<аддитивная операция>::=+|-|or  
  
<слагаемое>::=<множитель>{<мультипликативная операция><множитель>}  
  
<мультипликативная операция>::=\*|/|div|mod|and  
  
<множитель>::=<переменная>|<константа без знака>|(<выражение>)|not <множитель>  
  
<константа без знака>::=<число без знака>|<строка>|<имя константы>  
  
<пусто>::=  
  
<сложный оператор>::=<составной оператор>|<выбирающий оператор>|<оператор цикла>  
  
<составной оператор>::=begin <оператор>{;<оператор>} end  
  
<выбирающий оператор>::=<условный оператор>  
  
<условный оператор>::=if <выражение> then <оператор>|if <выражение> then <оператор> else <оператор>  
  
<оператор цикла>::=<цикл с предусловием>  
  
<цикл с предусловием>::=while <выражение> do <оператор>

Разбор синтаксических конструкций происходит рекурсивно.

Для удобства работы с различными типами токенов было сделано три перегруженных метода, которые принимали на вход требуемый параметр токена, а также флаги для перехода к следующему токену, если текущий принят, и для генерации ошибки.

СЕМАНТИЧЕСКИЙ АНАЛИЗАТОР

В задачи семантического анализатора входит анализ совместимости типов, а также проверка идентификаторов.

Семантический анализ встроен в класс-комплилятор и выполняется параллельно с синтаксическим анализом.

Для работы семантического анализатора требовалось разработать класс Таблиц Идентификаторов. Таблица Идентификаторов хранит в себе все идентификаторы программы, а также задаёт соответствие   
(идентификатор)->(тип идентификатора, ссылка на тип). В качестве внутренней структуры была выбрана структура данных словарь, для удобства поиска идентификаторов по их названию.

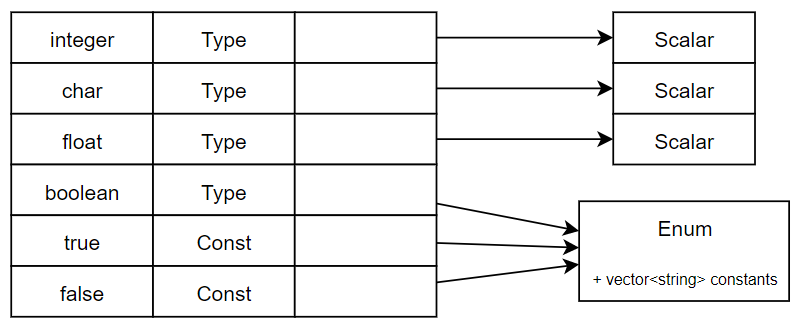
Таблица Типов, в свою очередь, не требовалось, так как для получения ссылки на тип достаточно было обратиться к идентификатору, который представляет данный тип. Для этого была выделена фиктивная область, в которую включались базовые идентификаторы ‘integer’, ‘float’, ‘boolean’, ‘char’, ‘true’, ‘false’. 

Рисунок 5. Таблица Идентификаторов фиктивной области действия.

Элемент типа, в зависимости от типа переменной, содержит в себе название данного типа (для удобства отладки), допустимые приведения типов и операции, а также необходимую информацию для данного типа.

* Для перечислимого типа требуется хранить список констант
* Для интервального типа требуется хранить ссылку на базовый тип, минимальное и максимальное значение
* Для массивов требуется хранить размерность, тип элементов, а также типы индексов в каждой размерности.

Также, таблица идентификаторов хранит в себе ссылку на объемлющую область видимости. Это позволяет реализовать вложенные области видимости, как в случае с процедурами и функциями (однако, в рамках данной задачи, программа имеет только две области видимости: фиктивную и область видимости программы).

В классе-компиляторе Таблицы Идентификаторов хранятся в структуре связанный стек. Стек позволяет однозначно определять, в какой области видимости мы находимся, так как текущая всегда находится на вершине стека.

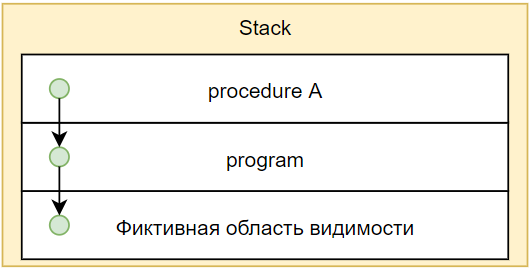


Рисунок 6. Стек Таблиц Идентификаторов.

Так как, в конечном итоге, все конструкции указывают на базовые типы (За исключением перечислимых типов), то проверка соответствия типов сводиться к простой проверке одинаковости ссылок на тип.

НЕЙТРАЛИЗАЦИЯ

Для упрощения отслеживания множества ошибок используется нейтрализация. Так, если компилятор встретит ошибку, то он не остановит проверку на текущем токене, а продолжит выполнения анализа.

Для различных типов ошибок используется различные стратегии нейтрализации.

НЕЙТРАЛИЗАЦИЯ ЛЕКСИЧЕСКИХ ОШИБОК

При возникновении лексической ошибки, Модуль I/O пропускает символы до первого оператора, конца строки или пробела. Лексический анализатор, в свою очередь, просто пропускает ошибочную лексему и запрашивает следующую у Модуля.

НЕЙТРАЛИЗАЦИЯ СИНТАКСИЧЕСКИХ ОШИБОК

Для нейтрализации ошибок используется пропуск токенов до указанных. Так, при возникновении ошибки в простом операторе, пропускаются все токены, пока на вход не придёт ‘;’, ‘.’ или begin.

НЕЙТРАЛИЗАЦИЯ СЕМАНТИЧЕСКИХ ОШАБОК

Семантические ошибки возникают из-за несоответствия типов при проведении каких-либо операций. Для нейтрализации такой ошибки, переменной, которой попытались присвоить значение недопустимого типа, присваивается ошибочный тип. Таким образом, предполагается, что в дальнейшем, при проведении различных операций, переманная будет иметь присвоенный ошибочный тип.

Каждая ошибка выводится через специальный обработчик ошибок. В его задачи входит не только вывод ошибок в консоль, но и подсчёт возникших ошибок. Таким образом, когда компиляция закончится, данный обработчик определяет успешность компиляции.

ГЕНЕРАЦИЯ КОДА

В качестве инфраструктуры для создания компилятора был выбран LLVM, так как он наиболее подходит для C++, а именно LLVM C++ API.

Однако, несмотря на существование библиотеки, создание генератора кода оказалось очень трудоёмкой работу ввиду отсутствия нормальной документации для C++ API. Сгенерированная Doxygen документация не даёт обширного представления о возможностях библиотеки.

LLVM IR

IR (Intermediate Representation) – низкоуровневое промежуточное представление, используемое инфраструктурой LLVM, платформо-независимый ассемблер с бесконечным количеством локальных регистров. Компилируется LLVM из промежуточного языка в ассемблер для конкретной платформы.

Особенностью языка является концепция SSA (Static Single Assingment) которая предполагает, что каждой переменной значение присваивается лишь единожды. Однако, если требуется, всё же, добавить модель мутабельной переменной, то её можно представить в виде переменной-ссылки на ячейку памяти, в которой можно свободно изменять значение переменной.

СБОРКА

LLVM не распространяется в виде библиотек. Их нужно собирать через CMake. На компиляцию библиотек ушло около 13-ти часов.

СТРУКТУРА

Кодогенератор представляет из себя класс, содержащий набор методов для кодогенерации. Сам генератор управляется классом-компилятором, вызывая у него нужные функции во время парсинга выражений.

В процессе интеграции базовых функций генераций стало ясно, что текущая архитектура класса-компилятора не соответствует требуемой. Функции кодогенератора, а также переменные пространства LLVM приходится вызывать прямиком из функций, обрабатывающих БНФ. В результате, функции, отвечающие за обработку БНФ, отвечают за синтаксическую, семантическую обработки и кодогенерацию.