Федеральное государственное автономное
образовательное учреждение
высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт космических и информационных технологий
Кафедра информатики

УТВЕРЖДАЮ
Заведующий кафедрой
_____ А.С. Кузнецов
«13» 06 2017 г.

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

Инструментальная поддержка визуального прототипирования синтаксически-управляемых трансляторов

09.04.04 «Программная инженерия»
09.04.04.00.01 «Программное обеспечение вычислительной техники и автоматизированных систем»

Научный руководитель	08.06.17	доцент, к.т.н.	А.С. Кузнецов
	подпись, дата	должность, ученая степень	инициалы, фамилия
Выпускник	08.06.17		А.Л. Богачук
	подпись, дата		инициалы, фамилия
Рецензент	08.06.17	профессор, д.т.н.	С. А. Бронов
	подпись, дата	должность, ученая степень	инициалы, фамилия
Нормоконтролер	08.06.17		О. А. Антамошкин
	подпись, дата		инициалы, фамилия

Красноярск 2017

АННОТАЦИЯ

В данной работе рассматривается создание алгоритмического и программного обеспечения инструментов разработки трансляторов.

Исследуется проблема доступности, функциональности и гибкости средств визуального прототипирования синтаксически-управляемых трансляторов. Рассматривается алгоритмическая модель использования генерации промежуточного кода LLVM по входной спецификации GNU Bison. Показывается практическая осуществимость построения предложенной модели. Дано описание программного обеспечения для моделирования визуального представления лексических и синтаксических анализаторов с возможностью взаимодействия с компиляторной инфраструктурой.

Ключевые слова: трансляторы, синтаксически-управляемые трансляторы, формальная грамматика, контекстно-свободная грамматика, восходящий разбор, абстрактное синтаксическое дерево, flex, GNU Bison, LLVM, интегрированная среда разработки.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Компиляторы составляют существенную часть программного обеспечения ЭВМ. Это связано с развитием языков программирования высокого уровня, которые являются основным средством разработки программ. Кроме того, постоянно растущая потребность в компиляторах обусловлена активным развитием архитектур ЭВМ. Возникновение новых архитектур и совершенствование старых требует создание новых компиляторов для многих языков программирования. В то же время, возрастает потребность в алгоритмических и программных средствах разработки компиляторов. Ведутся исследования в области синтаксически-управляемой трансляции, направленные на оптимизацию алгоритмов восходящего и нисходящего синтаксического разбора. Активно развиваются системы анализа, оптимизации и трансформации программ, такие как LLVM.Ведётся разработка программного обеспечения генерации лексических и синтаксических анализаторов по заданной спецификации, такого как flex и GNU Bison.

Не смотря на имеющиеся алгоритмические и программные инструменты, разработка компиляторов является весьма трудоемким процессом. Это усугубляется тем, что данные программные инструменты не имеют средств визуальной разработки. Существуют инструменты визуальной разработки синтаксических анализаторов, такие как Visual Lang Lab и Visual BNF, которые частично устраняют этот недостаток, но имеют ограниченный функционал. В данных программных средствах отсутствует независимый лексический анализ и возможность контроля оптимизации и трансформации промежуточного кода. Кроме того, подобные инструменты с закрытым исходным кодом и, как правило, коммерческие. Возникает необходимость в открытом бесплатном программном инструменте, с поддержкой генераторов программных анализаторов и компиляторной инфраструктуры, для визуального прототипирования синтаксически-управляемых трансляторов.

Цель диссертационной работы состоит в создании интегрированной среды разработки Bison- и flex-спецификаций с генерацией промежуточного кода LLVM.

В работе ставятся и решаются следующие основные задачи:

1. Обзор существующих аналогов разрабатываемой системы.

2. Применение алгоритма генерации промежуточного кода LLVM по входной спецификации GNU Bison.

3. Реализация программного обеспечения визуальной разработки синтаксически-управляемых трансляторов.

Научная новизна заключается в алгоритмическом обеспечении процесса разработки транслятора. Впервые был задействован алгоритм генерации промежуточного кода LLVM по входным спецификациям GNU Bison как средство взаимодействия среды разработки трансляторов с компиляторной инфраструктурой.

Разработанное приложение может использоваться в учебном процессе при обучении студентов по «компиляторным курсам», и в отдаленной перспективе - при промышленной разработке компиляторов.

Диссертация состоит из аннотации, введения, 3 глав, заключения, списка использованных источников и приложения. Основное содержание работы изложено на 50 страницах текста, содержит 20 рисунков. Список использованных источников включает 23 наименования.

Основное содержание работы: во введении обоснована актуальность темы диссертационной работы, дана постановка цели и задач, а также краткое описание содержания диссертации.

В первой главе проанализирована предметная область диссертационной работы. Представлено обоснование темы и задач работы. Показана актуальность исследований в области построения визуальных средств разработки синтаксически-управляемых трансляторов.

Во второй главе представлена алгоритм кодогенерации промежуточного кода LLVM по входным спецификациям GNU Bison, как метод взаимодействия среды разработки с компиляторной инфраструктурой.

В третьей главе показывается практическая осуществимость построения предложенной модели взаимодействия. Дано описание программного обеспечения для редактирования и отладки визуального представления лексических и синтаксических анализаторов, и генерации кода промежуточного представления компиляторной инфраструктуры.

В заключении перечислены результаты диссертационной работы, показаны дальнейшие направления исследования, сформулированы основные выводы.

В приложении приводится список ошибок и сообщений отладчика в интегрированной среде разработки трансляторов.

Глава 1. Исследование особенностей и процессов разработки синтаксически-управляемых трансляторов

1.1 Особенности синтаксически-управляемой трансляции

Транслятор — программа или техническое средство, выполняющее преобразование исходного кода в объектный код, понятный адресату. В случае, если адресатом является процессор, то процесс трансляции называется компиляцией. Современная компиляция делится на два этапа: генерация дерева разбора и генерация машинного кода. Такой подход позволяет обрабатывать независимый от целевой архитектуры исходный код, и наоборот, разрабатывать генераторы машинного кода для новых целевых платформ в независимости от исходного языка программирования [5].

Первым и ключевым этапом трансляции является обработка и генерация дерева разбора, в котором происходит преобразование исходного кода в промежуточное представление. Данный этап в свою очередь делится на два под-этапа: лексический анализ и синтаксический анализ.

Лексический анализ - процесс аналитического разбора входной последовательности символов специальной программой - лексическим анализатором, с целью получения на выходе последовательности символов, называемых токенами. Целью такого преобразования является подготовка входной последовательности для дальнейшего синтаксического анализа и исключения лексических подробностей, которые приводят к усложнению грамматики [3].

Синтаксический анализ – процесс преобразования входной последовательности токенов специальной программой – синтаксическим анализатором в абстрактное синтаксическое дерево, в соответствии с грамматикой языка. Таким образом получается промежуточное представление, с которым работает генератор машинного кода [4].

Абстрактное синтаксическое дерево – это конечное ориентированное дерево, в котором абстрактные атрибуты связаны с синтаксическими конструкциями. Во время обработки строки синтаксический анализатор находит последовательность применений правил и производит действия на этапе разбора. Это позволяет сократить количество операций и сэкономить память, в отличии от дерева разбора, в котором отсутствуют узлы и рёбра для тех синтаксических правил, которые не влияют на семантику программы [6]. Процесс трансляции, использующий абстрактное синтаксическое дерево называется синтаксически управляемой трансляцией. Синтаксически управляемая трансляция работает за счет добавления действий в контекстно-свободную грамматику.

Контекстно-свободная грамматика – частный случай формальной грамматики, у которой левые части всех продукций являются одиночными нетерминалами – объектами, обозначающими какую-либо сущность языка и не имеющими конкретного символьного значения. Основная задача синтаксического анализа – разбор дерева, соответствующее разбору контекстно-свободной грамматике языка. Это означает, что определяется одна или более синтаксическая группа и задаются правила их сборки из составных частей. В настоящее время чаще всего используется либо LL(1)-анализ и его вариант – метод рекурсивного спуска, либо LR(1)-анализ и его вариант LALR(1) [7].

LL(k)-анализатор – нисходящий синтаксический анализатор для контекстно-свободных грамматик. Он анализирует входной поток слева направо, и строит левый вывод грамматики. Для принятия решения используется не более k символов цепочки. LL(1)-грамматики довольно распространены, так как соответствующие LL-анализаторы просматривают поток только на один символ вперед при принятии решения о том, какое правило грамматики необходимо применить [13]. Метод рекурсивного спуска – частный случай данного алгоритма, реализуемый путём взаимного

вызова процедур, где каждая процедура соответствует одному из правил контекстно-свободной грамматики. Применения правил последовательно, слева-направо поглощают токены, полученные от лексического анализатора. Это один из простейших алгоритмов синтаксического анализа, подходящий для полностью ручной реализации [8].

LR(k)-анализатор – восходящий синтаксический анализатор для исходных кодов программ, написанных на некотором языке программирования, который читает входной поток слева направо и производит наиболее правую продукцию контекстно-свободной грамматики. Для принятия решения используется не более k символов цепочки. LR-анализатор основан на алгоритме, приводимом в действие таблицей анализа - структурой данных, которая содержит синтаксис анализируемого языка. Таким образом, термин LR-анализатор относится к классу анализаторов, которые могут разобрать почти любой язык программирования, для которого предоставлена таблица анализа [9]. Таблица анализа создаётся генератором синтаксических анализаторов. LR-анализ лучше в части сообщения об ошибках, то есть определяет синтаксические ошибки там, где вход не соответствует грамматике, как можно раньше [10]. В отличие от этого, LL(k) анализаторы могут задерживать определение ошибки до другой ветки грамматики из-за отката, часто затрудняя определение места ошибки в местах общих длинных префиксов. Синтаксис многих языков программирования может быть определён грамматикой, которая является LR(1) или близкой к этому, и по этой причине LR-анализаторы часто используются компиляторами для выполнения синтаксического анализа исходных кодов [11]. По аналогичной причине LR-анализ может быть обобщён как произвольный анализ контекстно-свободного языка без потери производительности. И наконец, восходящий разбор менее интуитивно понятный, чем нисходящий, но тем не менее позволяет разбирать больше грамматик. LALR(1) – вариант алгоритма восходящего синтаксического

анализа, расширяющий его в случае конфликтов «сдвиг-свёртка» и «свёртка-свёртка» и используется в большинстве компиляторов и системах автоматического построения синтаксических анализаторов [13].

Таким образом, синтаксически управляемая трансляция и её атрибуты применяются на практике в большинстве случаев. Абстрактное синтаксическое дерево сокращает количество переходных состояний, тем самым экономя память. Контекстно-свободные грамматики, в отличии от контекстно-зависимых, имеет эффективные способы разбора [12]. Восходящий разбор используется большинством компиляторов и генераторов синтаксических анализаторов, таких как GNU Bison [17].

1.2 Анализ инструментов разработки синтаксически-управляемых трансляторов

Самостоятельное написание лексических и синтаксических анализаторов неоправданно, так как существуют различные программные средства, способные генерировать лексические и синтаксические анализаторы на основе описанных спецификаций.

На данный момент возможны два наиболее вероятных пути порождения анализаторов исходного языка – это совместное использование проектов flex и Bison или применение программного инструмента ANTLR, остальные проекты либо не достигли подобного уровня функциональности, либо устарели и на данный момент не поддерживаются, либо не имеют подобного количества документации и поддержки со стороны сообщества.

flex и Bison – современные выпуски программ lex и YACC, предназначенных для порождения лексических и синтаксических анализаторов по описанной пользователем спецификации [16, 17]. Выбор в пользу flex и Bison обусловлен тем, что это законченные продукты, способные работать независимо друг от друга, но при этом способные быстро и эффективно работать сопряжённо. flex генерирует лексические анализаторы самостоятельно, что может быть полезным при изучении

лексического анализа. Bison работает с потоком токенов, в независимости от их источника. Раздельность данных инструментов обеспечивает взаимозаменяемость и гибкость. Имеется возможность взаимодействия собственной программной среды с системой Bison за счёт передачи консольных команд и вывода состояний конечного автомата в отдельный файл. Bison работает с контекстно-свободными грамматиками класса LALR(1), используемых в большинстве компиляторов. Кроме того, оба средства достаточно развиты, имеют подробную документацию и поддержку со стороны разработчиков [16, 17].

Программный инструментарий flex позволяет определить лексический анализатор с помощью регулярных выражений для описания шаблонов токенов. Входной файл написан на языке flex и описывает генерируемый лексический анализатор. Программа на языке flex, как правило, состоит из объявлений, которые могут включать объявления переменных, именованные константы, регулярные определения и символьный блок, содержащий определения на C++; правил трансляции, содержащих правила трансляции вида «шаблон-действие», где «шаблон» – регулярное выражение, «действие» – фрагменты кода языка C++; вспомогательные функции, включающие дополнительные функции на C++, используемые в действиях. Компилятор flex преобразует входной файл в программу на языке программирования C++. Выход компилятора C++ представляет собой работающий лексический анализатор, который на основе потока входных символов выдаёт поток токенов. Обычно полученный лексический анализатор, используется в качестве подпрограммы синтаксического анализатора [16].

```
%{
    Объявления на C++
%}

%Объявления flex

%%

Правила трансляции

%%

Вспомогательные функции на C++
```

Рисунок 1.2.1 – Структура входной спецификации flex

Программный инструмент Bison позволяет генерировать синтаксический анализатор при помощи описанной грамматики. Чтобы определить язык для Bison, необходимо написать файл, описывающий грамматику в синтаксисе Bison – файл грамматики Bison, являющийся входной спецификацией. Файл грамматики как правило состоит из объявлений на C++, определяющих типы и переменные, используемые в действиях; объявления Bison, задающие имена терминальных и нетерминальных символов, а также описывающие приоритет операций и типы данных семантических значений различных символов; правил грамматики, определяющих, как каждый нетерминальный символ собирается из своих частей; дополнительный код, содержащий любой необходимый код [17]. Нетерминальный символ формальной грамматики на входе Bison представляется идентификатором, подобному идентификатору C++ и записывается в нижнем регистре. Представление в Bison нетерминальных символов также называется типом лексем. Типы лексем также могут быть представлены идентификаторами в стиле C++ и записываются в верхнем регистре. Терминальный символ, соответствующий конкретному ключевому слову языка следует называть так же, как это ключевое слово выглядит в верхнем регистре. Терминальный символ также может быть представлен как

однолитерная константа C++. В случае, когда лексема представляет собой одиночную литеру (скобку, знак плюс и т.д.), используется та же литера в качестве терминального символа для этой лексемы. Третий способ представления терминального символа – представление строковой константой C++ из нескольких литер [17].

```
%{
    Объявления на C++
%}

%Объявления Bison

%%

Правила грамматики

%%

Вспомогательные функции на C++
```

Рисунок 1.2.2 – Структура файла грамматик Bison

Результатом работы flex и Bison является синтаксический анализатор, сгенерированный на языке C++, способный преобразовать исходную программу в промежуточное представление в виде абстрактного синтаксического дерева. На следующей стадии требуется компилятор, способный оптимизировать полученное промежуточное представление и преобразовать в машинный код.

Разработка собственного компилятора самостоятельно - это нерациональный путь, так как существующие решения с открытым кодом реализуют внешнее и внутреннее представление компилятора со множеством весьма нетривиальных алгоритмов оптимизации, которые, более того, протестированы и используются длительное время [19].

На сегодняшний день существует два реалистичных пути разработки компилятора для собственного языка: использование GCC либо

использование LLVM. Другие проекты компиляторов с открытым исходным кодом либо не достигли той степени развития, как GCC и LLVM, либо устарели и перестали развиваться, они не обладают развитыми алгоритмами оптимизации, и могут не обеспечивать полной совместимости даже со стандартом языка C, не говоря о поддержке других языков программирования.

GCC (GNU Compiler Collection) является более ранним проектом, первый выпуск которого состоялся в 1987 году. GCC написан на языке C, более поздние версии – на C++. Компилятор поддерживает множество языков программирования: C, C++, Objective C, Fortran, Java, Ada, Go. Компилятор GCC поддерживает большое количество процессорных архитектур и операционных систем, и является в настоящее время наиболее распространённым компилятором [18]. LLVM – это универсальная система анализа, трансформации и оптимизации программ. Инфраструктура компиляторов LLVM состоит из различных инструментов. LLVM в свою очередь, гораздо новее, его первый выпуск состоялся в 2003 году. LLVM написан на языке C++. Компилятор поддерживает языки программирования C, C++, Objective-C and Objective-C++, и также имеет внешние представления для множества языков программирования [19]. Относительная новизна LLVM не является недостатком, проект достаточно развит, чтобы не содержать большое количество критических ошибок, и при этом не несёт в себе набора устаревших архитектурных решений, как GCC. Модульная структура компилятора позволяет использовать внешнее представление LLVM-GCC, который обеспечивает полную поддержку стандартов GCC, при этом генерация кода целевой платформы будет осуществляться LLC (внутренне представление LLVM) [19]. LLVM достаточно документирован, и для него существует большое количество примеров кода.

Компилятор LLVM, как и некоторые другие компиляторы, состоит из внешнего представления, оптимизатора и внутреннего представления. Такая структура позволяет разделить разработку компилятора нового языка

программирования, разработку методов оптимизации и разработку кодогенератора для конкретного процессора. Связующим звеном между ними является промежуточный язык LLVM, ассемблер «виртуальной машины». Внешнее представление преобразует текст программы на языке высокого уровня в текст на промежуточном языке, оптимизатор производит над ним различные оптимизации, а внутреннее представление генерирует код целевого процессора на ассемблере или непосредственно в виде бинарного файла. Помимо этого, LLVM может работать в режиме JIT-компиляции, когда компиляция происходит непосредственно во время исполнения программы. Промежуточное представление программы может быть как в виде текстового файла на языке ассемблера LLVM, так и в виде двоичного формата, «биткода». По умолчанию clang генерирует именно «биткод», но для отладки и изучения LLVM необходимо генерировать текстовое промежуточное представление на ассемблере LLVM IR [20].

LLVM официально не содержит генератора синтаксических анализаторов и не поддерживает работу с GNU Bison [19]. Написание собственного синтаксического анализатора либо переписывание кода синтаксического анализатора Bison в терминах LLVM является нетривиальной задачей. В настоящее время открыта проблема взаимодействия систем GNU Bison и LLVM, когда как использование сильных сторон данных систем способна значительно улучшить инструментальную поддержку разработки трансляторов [1]. Кроме того, данные инструменты не имеют графической оболочки и для работы с ними требуется вручную вводить соответствующий набор команд. Это усложняет разработку трансляторов и увеличивает количество ошибок при разработке.

1.3 Обзор аналогов разрабатываемой системы визуального прототипирования синтаксически-управляемых трансляторов

С развитием интегрированных сред разработки программного обеспечения, имеющих средства отладки, рефакторинга и визуализаций,

появились способы визуального прототипирования генераторов лексических и синтаксических анализаторов. В настоящее время существует небольшое количество подобных средств, не отличающихся большой функциональностью.

Прототипирование программного обеспечения – это процесс создания прототипа программы – функциональной реализации первичных элементов системы, из которой можно создать готовый продукт, осуществляя цепочку небольших циклов разработки [2]. В случае с синтаксически-управляемым транслятором, таким элементом может выступать дерево грамматики.

Визуальное прототипирование синтаксически-управляемых трансляторов - это создание прототипа синтаксически-управляемого транслятора с применением визуальных средств разработки, такими как редактирование с графическим выделением и автодополнением конструкций языка.

Самые популярные продукты - ANTLRWorks, Visual BNF и Visual Lang Lab. Данные продукты являются визуальными редакторами спецификаций и генераторами синтаксических анализаторов. Другие аналоги не достигли подобного уровня функциональности.

Visual BNF - это генератор синтаксических анализаторов общего назначения, он преобразует описание контекстно-свободной LALR/LR грамматики в детерминированный конечный автомат, который помещается в .dll файл. Полученный файл в дальнейшем используется в разрабатываемом приложении. Программа способна делать автоматические вставки символов, терминалов, продукций и входных точек продукций. Имеется окно вывода сообщений об ошибках. Главное окно программы представлено на рисунке 1.3.1. Данный продукт получил имя Visual BNF поскольку пользователь может визуально разбирать код и видеть синтаксическое дерево грамматики [21].

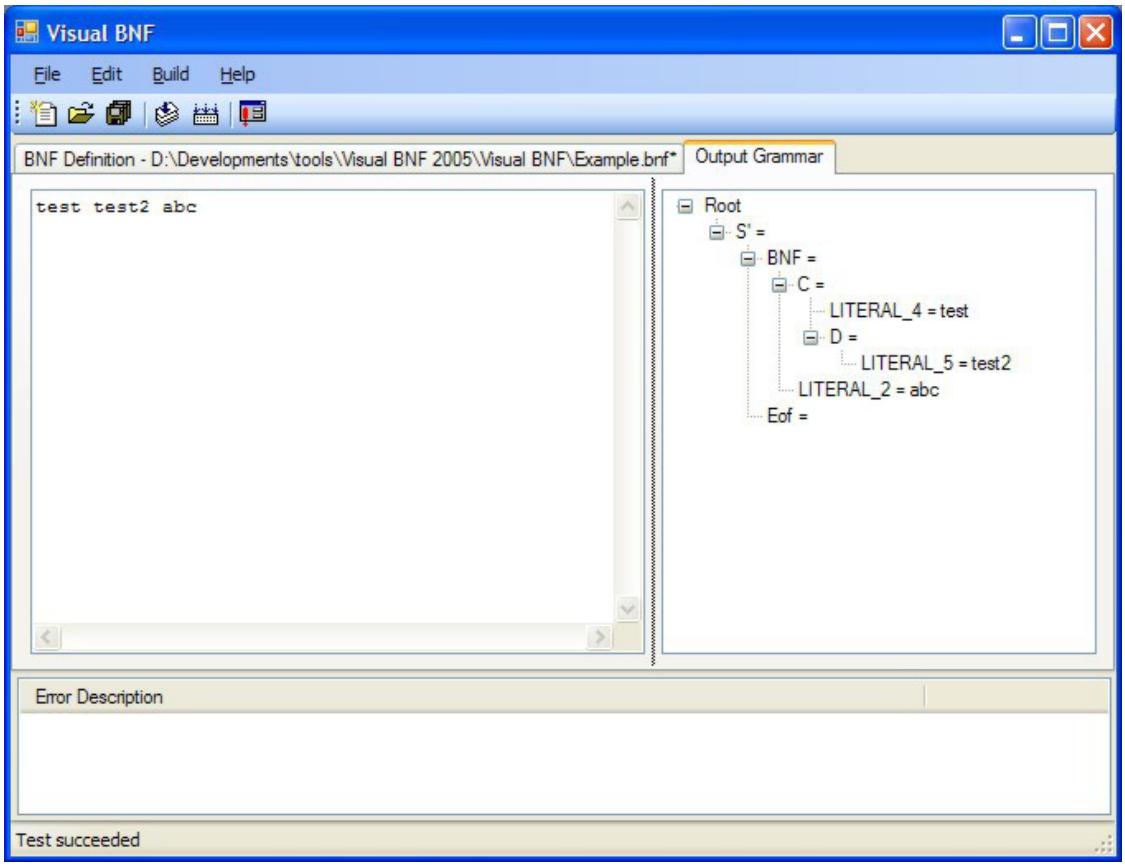

Рисунок 1.3.1 – Главное окно Visual BNF

Visual Lang Lab кроме стандартных функций редактирования и генерации использует синтаксическое дерево грамматики с иконками для лексем, нетерминалов, представления символов грамматики, грамматических правил и т.д. Кроме окна вывода сообщений присутствует окно тестового файла [22]. Главное окно программы показано на рисунке 1.3.2.

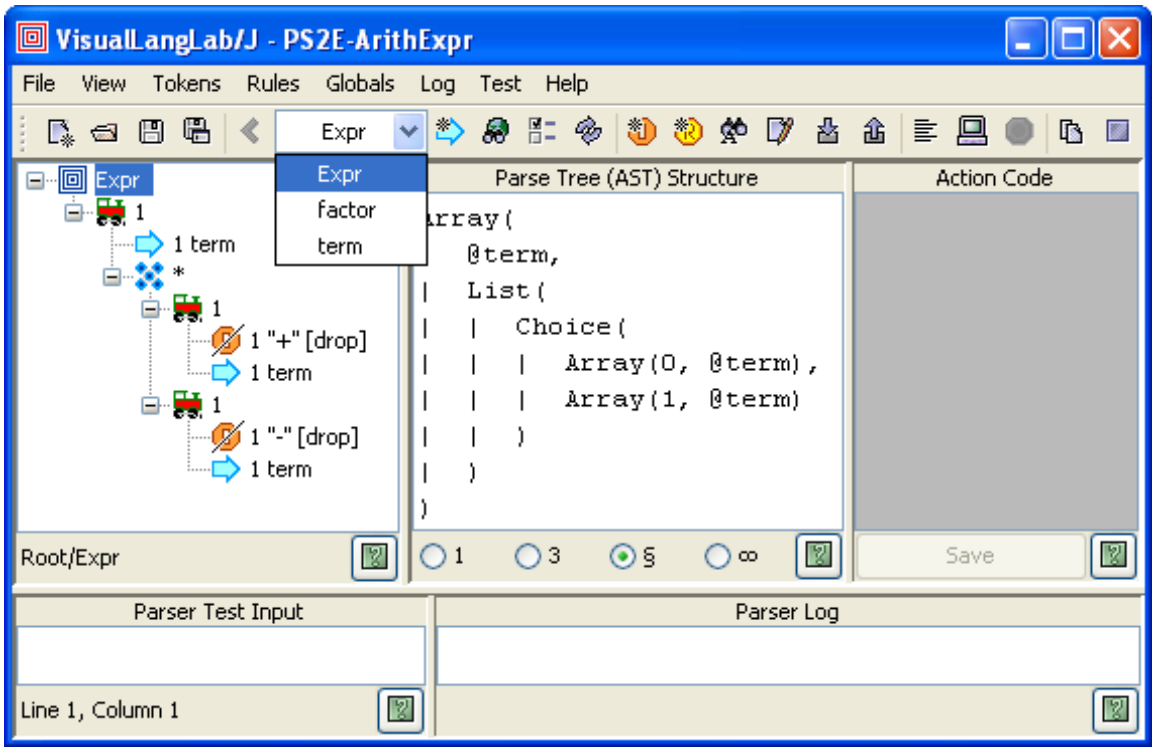

Рисунок 1.3.2 – Главное окно Visual Lang Lab

ANTLRWorks – генератор синтаксических анализаторов на основе ANTLR. Поддерживает функцию подсветки синтаксиса, отладки и визуализации дерева разбора [15].

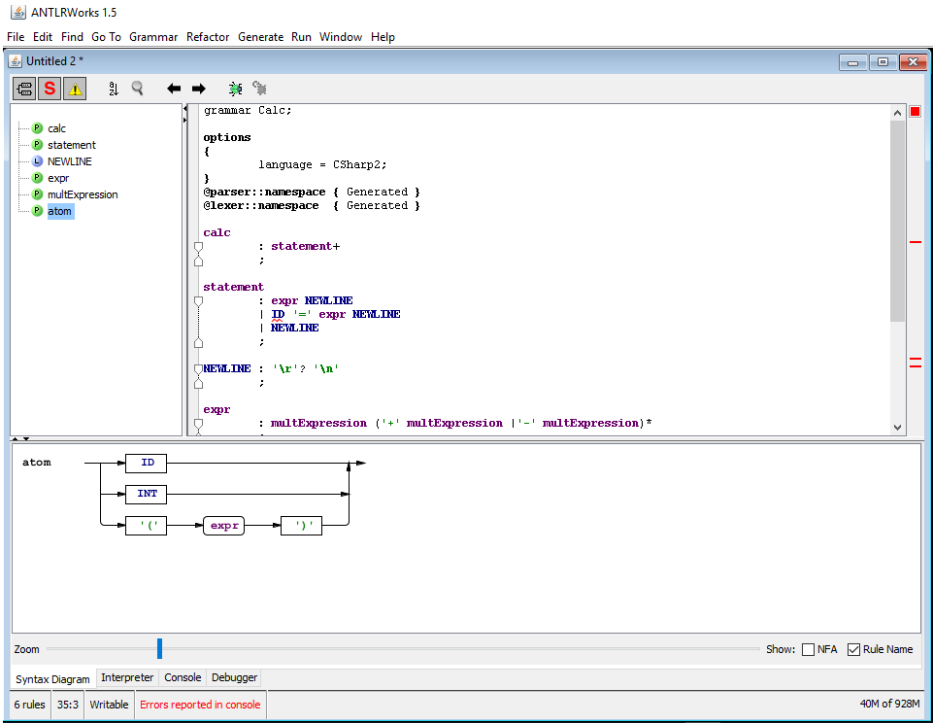

Рисунок 1.3.3 – Главное окно ANTLRWorks

Генерация независимого лексического анализатора является полезным при изучении лексического анализа и разработки отдельных модулей анализатора исходного кода. Тем не менее, лексический анализ является встроенной частью синтаксического анализа данных инструментов, генерация независимого лексического анализатора не предоставляется. Перечисленные инструменты содержат средства визуальной разработки, генерации и компиляции синтаксических анализаторов, но не предоставляют средств для оптимизации и кодогенерации под конкретную архитектуру процессора. Кроме того, некоторые из данных инструментов не являются бесплатными либо предоставляются по коммерческой лицензии, что затрудняет их широкое использование [15, 21, 22].

Выводы к главе 1. Рассмотрены основные особенности и сделан вывод о повсеместном применении синтаксически-управляемых трансляторов. Оправдано применение инструментов flex, Bison и LLVM как базовых средств разработки синтаксически-управляемых трансляторов. Сделан обзор аналогов разрабатываемой интегрированной среды разработки. Выделены основные проблемы.

Глава 2. Алгоритмическое обеспечение визуального прототипирования синтаксически управляемых трансляторов

2.1 Алгоритм генерации промежуточного представления кода LLVM по входной спецификации GNU Bison

В целях объединения функционала, улучшающего качество разработки трансляторов, такого как генерация синтаксического анализатора средствами Bison по входным спецификациям и оперирование промежуточным кодом средствами LLVM, имеется потребность в преобразовании абстрактного синтаксического дерева Bison в синтаксическое дерево LLVM на этапе анализа исходного кода.

Общий принцип алгоритма такого преобразования заключается в добавлении функции codeGen в каждый класс узла дерева. Данная функция возвращает указатель на соответствующее поддерево в терминах LLVM. Производится генерация точки входа и выполнение.

Реализуются функции для листьевых вершин, к примеру, NInteger и NDouble, которые возвращают константы в терминах LLVM. Как показано на рисунке 2.1.1, у класса ConstantInt вызывается метод get(). Все константы и типы в LLVM статические, поэтому, если требуется их применить, используется единственный конкретный экземпляр константы или типа. В менеджере констант запрашивается константа с типом Int64 и если такая константа есть, то на неё возвращается указатель.

```
Value* NInteger::codeGen(CodeGenContext& context) {
    return ConstantInt::get(Type::getInt64Ty(getGlobalContext()), value, true);
}

Value* NDouble::codeGen(CodeGenContext& context) {
    return ConstantFP::get(Type::getDoubleTy(getGlobalContext()), value);
}
```

Рисунок 2.1.1 – Функции численных листьевых вершин

В реализации для NBinaryOperator, как показано на рисунке 2.1.2, используется создание бинарной операции, вызываемой от инструкции. Тип

инструкции определяется из метаинформации, которая сохранилась для данного оператора. Левый аргумент BinaryOperator возвращает код, который сгенерировался от левого поддерева, правый аргумент, соответственно, от правого поддерева. При вызове на корневой вершине метода CodeGen программа получает указатель на корневую вершину дерева в формате LLVM.

```
Value* NBinaryOperator::codeGen(CodeGenContext& context) {
    Instruction::BinaryOps instr;
    switch (op) {
        case TPLUS:
            instr = Instruction::Add;
            goto math;
        case TMINUS:
            instr = Instruction::Sub;
            goto math;
        case TMUL:
            instr = Instruction::Mul;
            goto math;
        case TDIV:
            instr = Instruction::SDiv;
            goto math;
    }
    return NULL;
math:
    return BinaryOperator::Create(instr, leftSide.codeGen(context), rightSide.codeGen(context), "",
            context.currentBlock());
}
```

Рисунок 2.1.2 – Функция бинарных операторов

Далее необходимо сгенерировать главную функцию main, которая вызывается в первую очередь. Функция main возвращает Int64, так как производятся операции в основном с 64-битными числами. Функция принимает пустой массив аргументов. В переменной ftype, обращаясь к менеджеру функциональных типов, делается запрос на функцию, которая возвращает Int64 и принимает пустое множество аргументов. Создаётся функция такого типа, называемая main. Создаются базовые блоки внутри функции и добавляется код. В главном блоке присутствует инструкция return, которая будет возвращать то, что сгенерировалось из абстрактного синтаксического дерева. Всё что задано в данной программе, будет находиться в этом поддереве и его результат будет возвращаться. Подробная реализация метода показана на рисунке 2.1.3.

```
void CodeGenContext::generateCode(NBlock& root) {
    vector<const Type*> argTypes;
    FunctionType *ftype = FunctionType::get(Type::getVoidTy(getGlobalContext()), argTypes, false);
    mainFunction = Function::Create(ftype, GlobalValue::InternalLinkage, "main", module);
    BasicBlock *bblock = BasicBlock::Create(getGlobalContext(), "entry", mainFunction, 0);

    pushBlock(bblock);
    root.codeGen(*this);
    ReturnInst::Create(getGlobalContext(), bblock);
    popBlock();

    PassManager pm;
    pm.add(createPrintModulePass(&outs()));
    pm.run(*module);
}
```

Рисунок 2.1.3 – Генерация главной функции

Далее по функции main берётся весь код, который является кодом промежуточного представления LLVM и применяются оптимизации, генерации машинного кода и другие низкоуровневые операции.

2.2 Применение алгоритма генерации промежуточного представления кода LLVM как инструмента визуального прототипирования трансляторов

Использование изложенного алгоритма генерации промежуточного представления кода LLVM обеспечивает возможность взаимодействия среды визуального прототипирования трансляторов, работающей на основе Bison, с компиляторной инфраструктурой, располагающей инструментами дизассемблирования, оптимизации, трансляции в другие языки программирования и генерации машинного кода под различные архитектуры процессоров. Взаимодействие с компиляторной инфраструктурой происходит за счёт обработки действий пользователя над графическим интерфейсом программного инструмента и передачи консольных команд интегрированной средой разработки в LLVM, а также работы с выходными файлами компиляторной инфраструктуры. Модель продемонстрирована на рисунке 2.2.1.

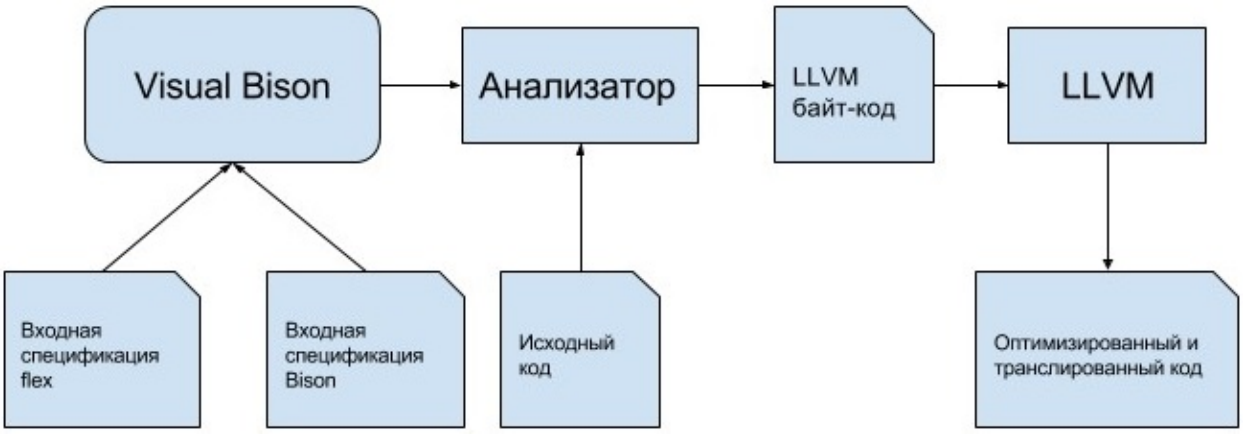

Рисунок 2.2.1 – Схема взаимодействия с LLVM

На основе данного алгоритма работает программная библиотека, встроенная в интегрированную среду разработки трансляторов. Данная библиотека подключается в корневом файле проекта синтаксического анализатора и во входном файле грамматики Bison в области объявлений на C++ указанием заголовочного файла библиотеки. В узлы дерева грамматики Bison, как показано на рисунке 2.2.2, добавляются пользователем экземпляры классов библиотеки, генерирующих соответствующие элементы синтаксического дерева LLVM. Во входном методе синтаксического анализатора необходимо вызвать метод кодогенерации. Библиотека поддерживает стандартные конструкции C-подобного языка и может расширяться пользователем, так как представлена открытым исходным кодом.

```
numeric : TINTEGER { $$ = new NInteger(atol($1->c_str())); delete $1; }
                | TDOUBLE { $$ = new NDouble(atof($1->c_str())); delete $1; }

expr : ident TEQUAL expr { $$ = new NAssignment(*$<ident>1, *$3); }
         | ident TLPAREN call_args TRPAREN { $$ = new NMethodCall(*$1, *$3); delete $3; }
         | ident { $<ident>$ = $1; }
         | numeric
         | expr TPLUS expr { $$ = new NBinaryOperator(*$1, $2, *$3); }
         | expr TMINUS expr { $$ = new NBinaryOperator(*$1, $2, *$3); }
         | expr TMUL expr { $$ = new NBinaryOperator(*$1, $2, *$3); }
         | expr TDIV expr { $$ = new NBinaryOperator(*$1, $2, *$3); }
         | expr comparison expr { $$ = new NBinaryOperator(*$1, $2, *$3); }
         | TLPAREN expr TRPAREN { $$ = $2; }
         ;
```

Рисунок 2.2.2 – Фрагмент грамматики Bison с использованием библиотеки

При помощи графического интерфейса программного инструмента имеется возможность строить дерево грамматик и производить дальнейшие этапы разработки и тестирования транслятора. Например, скомпилировать и запустить сгенерированный посредством Bison синтаксический анализатор. Компиляция анализатора происходит за счёт запуска с соответствующими параметрами средой разработки отдельного исполняемого файла компилятора, поддерживающего язык C++. При обработке исходного кода синтаксическим анализатором генерируется файл промежуточного кода с расширением «.bc». Появляется возможность выбора уровня оптимизации и запуска процесса оптимизации. Также присутствует выбор архитектуры процессора и целевого языка программирования для трансляции в машинный код или другой язык программирования соответственно. Файл промежуточного кода может быть представлен в читаемом виде, что является полезным при изучении компиляторов. Данные операции осуществляются за счёт передачи в LLVM некоторых команд:

- -c a.bc – указание входного файла, где «a» - имя входного файла;
- -O0 – уровень оптимизации, вместо «0» допустимы более высокие численные значения уровня оптимизации;
- --target – целевая платформа трансляции;
- llvm-dis – дизассемблирование входного файла;
- llvmc a.out – трансляция в бинарный файл, где «a» - имя выходного файла.

Все параметры запуска, оптимизаций и трансляций кода сохраняются в файле сборки разработанного транслятора.

С применением рассмотренного алгоритма генерации промежуточного LLVM кода в программном инструменте разработки синтаксически-управляемых трансляторов, в отличии от аналогов, появляется возможность осуществлять полный цикл разработки трансляторов посредством визуального прототипирования синтаксических деревьев с применением

графического интерфейса отладки и взаимодействия с компиляторной инфраструктурой. Модель позволит в дальнейшем обеспечивать взаимодействие подобных инструментов разработки трансляторов.

Выводы к главе 2. Произведён обзор алгоритма генерации промежуточного кода LLVM по входной спецификации GNU Bison. Предложена модель применения алгоритма, как средства взаимодействия компонентов интегрированной среды разработки трансляторов. Сделан вывод о полезности данной модели.

Глава 3. Программное обеспечение визуального прототипирования синтаксически-управляемых трансляторов

3.1 Задачи и функции программной системы

Исходя из описания актуальных средств визуальной разработки трансляторов и проблем, изложенных в разделе 1.3, разработано приложение, выполняющее функции:

- Написание flex-спецификации и генерация кода лексического анализатора.
- Написание bison-грамматики и генерация кода синтаксического анализатора.
- Написание тестового файла.
- Подсветка синтаксиса.
- Автоматическая вставка кода.
- Выделение конфликтов по мере написания грамматики.
- Использование точек останова в тестовом файле или грамматике.
- Пошаговое выполнение на тестовом входном файле и визуализация переносов и сверток в стеке.
- Проверка символов предпросмотра в каждом состоянии и изменение ввода перед каждым последующим шагом синтаксического анализа.
- Генерация промежуточного кода LLVM и выбор параметров низкоуровневой трансляции.

Написание flex-спецификаций, bison-грамматик и генерация LLVM кода обеспечивает возможность разработки полноценных синтаксически-управляемых трансляторов.

Подсветка синтаксиса важна для облегчения чтения исходного кода и позволяет избегать синтаксических ошибок. Выделение конфликтов по мере написания грамматики даёт возможность своевременно исправить ошибки в написании спецификаций и позволит ускорить разработку транслятора.

Использование точек останова в исходном коде и его пошаговое выполнение с визуализациями позволяет отслеживать состояние, с целью выявления правильности работы на этапе отладки.

Таким образом, получена интегрированная среда разработки для bison- и flex-спецификаций с возможностью генерации LLVM-кода, с графическим выделением синтаксических конструкций и ошибок, а также возможностями отладки и визуализации синтаксических анализаторов.

3.2 Структура программной системы

Для реализации приложения был выбран язык C++, так как в проекте используется часть кодовой базы Bison, которая полностью написана на языке C++. Также выбор обусловлен тем, что C++ является кроссплатформенным
высокоуровневым языком, с поддержкой парадигмы объектно-ориентированного программирования, и имеет множество бесплатных реализаций в виде сред для разработки и компиляторов [14].
Графический интерфейс разработан с использованием бесплатной библиотеки wxWidgets [23].
Преимущества:

- использует стандартные элементы управления операционной системы, вместо отрисовки собственных;
- является платформо-независимой библиотекой;
- имеется удобный компонент для создания текстового редактора с подсветкой синтаксиса и сворачиванием структур [20];
- множество других вспомогательных классов, облегчающих разработку кроссплатформенного приложения.

В соответствии с архитектурой wxWidgets [23], в программе определен статический объект класса CVisualBisonApp, наследник wxApp, представляющий собой объект приложения, в котором происходит вызов функции OnInit на старте. Функция OnInit выполняет начальную настройку

приложения, открывает и читает файл настроек, и создает главное окно приложения класса CMainFrame.

Главное окно создает меню, панель инструментов, и обрабатывает команды создания, сохранения, и открытия проектов. Сам проект представлен классом CBisonDocument, и содержит данные проекта. При создании CBisonDocument инициализируются все необходимые для проекта окна. Класс является центральной точкой взаимодействия этих окон. Взаимодействие происходит при помощи сообщений. Для реагирования на сообщение определяется функция-обработчик сообщения, которая принимает объект сообщения, и связывается с точкой выхода сообщения. Точка выхода – это объект класса, наследующего от wxEvtHandler. Окно принимает сообщения от нужного ему объекта, и выполняет соответствующие действия. Для передачи сообщения другому окну, окно создает сообщение и передает его в CBisonDocument с помощью wxPostEvent. Такое сообщение может обрабатывать как сам CBisonDocument, так и другие окна. Такая организация позволяет ослабить взаимосвязь между окном и проектом, и между окном и окном, т.к. нет необходимости хранить указатели друг на друга, и вызывать методы. Каждое окно, которое хочет сохранить свои данные в архиве проекта, наследует от класса IProjectPersistentView, и реализует его методы.

В проекте сохраняется два файла: файл грамматики, и файл лексического анализатора. Чтобы сохранить их в одном физическом файле, создается zip-архив, в который запаковываются эти файлы. Архиву присваивается расширение «.vbison.zip» – стандартное расширение файла проекта Visual Bison.

Редактор выполнен с использованием класса wxStyledTextCtrl [23], который является оберткой мощного текстового компонента Scintilla [20]. Подсветка синтаксиса достигается разбором текста грамматики на лексемы, и группировкой этих лексем по типам. Задавать можно разные цвета для следующих типов лексем:

- идентификатор;
- строковый литерал, символьный литерал;
- комментарий;
- числовой литерал;
- директива;
- код-действие правила, код-пролог, код-эпилог;
- символы-разделители, например, круглые скобки, двоеточия, и т.д.;
- метки – идентификаторы в угольных скобках;
- заголовок правила.

Файл для лексического анализатора flex позаимствован у GNU Bison, и адаптирован для работы в Visual Bison.

Редактор также ограниченно разбирает синтаксис спецификации для нахождения всех правил. Координаты правил нужны для установки точек останова на них. Разбор синтаксиса, как и лексический анализ, происходит только после правки текста пользователем, с интервалом в 1,5 секунды, вместе с разбором ошибок в тексте спецификации. Точки останова отображаются через установку маркеров на линиях компонента wxStyledTextCtrl. При установке точки вызывается метод отладчика SetBreakpoint. Работу с точками организует класс CHasBreakpoints, от которого наследует редактор. CHasBreakpoints объявляет два пустых метода для реализации дочерними классами: IsBreakpointTargetOnLine – проверяет, имеет ли смысл точка останова для данной линии; ApplyBPToDebugger – вызывается, когда надо добавить точку в отладчик.

Редактор обрабатывает события создания нового экземпляра отладчика и шага отладчика. На каждом шаге отладки, редактор проверяет набор правил текущего состояния отладчика, и подсвечивает позиции в этих правилах, по которым проходит отладчик. Для подсветки используются индикаторы, реализуемые компонентом wxStyledTextCtrl. При правке текста отладчик отключается и индикаторы сбрасываются, т.к. редактор уже не

отображает актуальную для него информацию. При получении нового отладчика, редактор отсоединяет старый, и подключает новый, передавая ему все точки останова.

Редактор тестового файла работает на том же компоненте, что и редактор грамматики. Он также наследует от CHasBreakpoints, для поддержки точек останова. При установке точки вызывает метод отладчика SetLexemBreakpoint, который принимает диапазон символов в тестовом тексте. Точка срабатывает до сдвига лексемы, находящейся в заданном диапазоне, в стек.

Редактор обрабатывает события шага отладчика и нового отладчика. На каждом шаге подсвечивается лексема предпросмотра. При правке текста, или при подключении нового отладчика, текущий отладчик отключается.

Для генерации независимого лексического анализатора разработан статический класс CFlex. Функция GenerateFlex принимает на вход файл, написанный на языке flex с расширением «.l», описывающий генерируемый лексический анализатор, обращается к функции CreateFlexFile, которая запускает генератор лексических анализаторов flex и подает ему на вход переданный файл «.l». flex преобразует входной файл в программу на языке программирования C++, и сохраняет его в проект транслятора. В результате на выходе получается готовый к использованию лексический анализатор.

Правила встроенного анализатора разбираются построчно с помощью регулярного выражения. Имя лексемы проверяется выражениями:

- '.' – любой символ в одинарных кавычках;
- "(\\.|[^\\"])*" – строка, любые символы между двойных кавычек, кроме не экранированных двойных кавычек;
- [a-zA-Z_][a-zA-Z_0-9] – идентификатор, начинается с подчеркивания или буквы, далее могут встречаться цифры;
- . – (точка) специальное имя лексемы, означает подстановку символа, совпавшего с регулярным выражением правила, в одинарные кавычки;

- - – (дефис) специальное имя лексемы, игнорирование любых, совпавших с регулярным выражением, символов.

Анализатор реализует для отладчика интерфейс ITokenSupply, через который передает ему лексемы. Через него изменяется лексема предпросмотра. Главные методы интерфейса: PopNextToken и PeekNextToken. Первый забирает лексему, а второй только просматривает ее. В классе есть переменная для хранения следующей лексемы, которую можно менять методом AlterNextToken. Когда анализатор доходит до конца анализируемого текста, он возвращает специальную лексему $end, необходимую для Bison, чтобы выполнить заключительное правило $accept. Через метод Reset, можно вернуть анализатор в начальное состояние, к первой лексеме.

Одной из целей реализации проекта было обеспечение совместимости со спецификацией GNU Bison текущей версии, и облегчение переноса на последующие версии. Исходя из этой цели, выполняется обработка ошибок путем передачи файла спецификации в Bison, и вывода возвращенных ошибок.

Bison изначально проектировался как приложение с интерфейсом командной строки, которое завершает работу после обработки действия. Код написан без предположения что он будет повторно вызван. Переписывание кода могло вызвать множество правок, что затруднило бы перенос правок на новые версии, из-за необходимости перепроверять их все. Поэтому, принято решение использовать Bison как отдельный исполняемый файл, передавая ему имена файлов для вывода нужной информации. На момент написания приложения, Bison имеет вывод состояний конечного автомата в формате XML, необходимых для отладки. Вывод ошибок в подобном формате отсутствует, поэтому принято решение написать такую возможность.

Центральным местом вывода информации об ошибках является функция complains в файле complain.c. Эта функция подготавливает

сообщение к выводу. Затем она вызывает функцию error_message для печати сообщения в стандартный поток вывода ошибок. В код Bison была добавлена функция error_message_xml, которая выводит сообщения в формате XML в файл, указанный опцией –xmlerr. Каждое сообщение об ошибке имеет узел с меткой complaint, и атрибуты начала и конца ошибки в тексте: sline – строка начала ошибки, scolumn – колонка начала ошибки, eline – строка конца ошибки, ecolumn – колонка конца ошибки. В случае, если ошибка распространяется на весь текст, это атрибуты равны -1. В содержимом узла complaint находится текстовое описание ошибки.

После правки спецификации в Visual Bison, генерируется сообщение EVT_GRAMMAR_CHANGE, которое обрабатывает CBisonDocument. Класс, в свою очередь, вызывает Bison с аргументом –xmlerr и именем временного файла в качестве значения. После окончания работы Bison, XML файл с ошибками читается, и распознается с помощью класса wxXmlDocument библиотеки wxWidgets. Список ошибок записывается в поле объекта CBisonDocument, и генерируется сообщение EVT_ERRORS_LIST_CHANGE. Его обрабатывают все окна, желающие отобразить информацию об ошибках. Например, очевидный CErrorView – список ошибок, и CGrammarView – редактор грамматики, подчеркивает ошибки красной линией. Таким образом подсветка и вывод ошибок происходит в реальном времени. Для снижения нагрузки на процессор от вызова Bison после каждого введенного символа, принято решение вызывать Bison с интервалом в 1,5 секунды.

Отладчик представлен классом CGrammarDebugger. В конструктор передается набор состояний, выведенных через аргумент --xml из Bison, таблицы конечного автомата через аргумент --xmltables, и контекст отладки. Контекст включает в себя все необходимые данные для представления текущего шага выполнения отладчика. В контекст входит указатель на лексический анализатор. Набор состояний состоит из списка всех правил грамматики и списка состояний автомата. Таблицы конечного автомата: defgoto, pgoto, defact, pact, check, и table.

Таблицы defgoto, pgoto, defact, и pact – ассоциативные массивы с двухкомпонентным ключом. Все значения и ключи – целочисленные. Чтобы осуществлять поиск в таком массиве используется таблица table. Таблица table является собирательной: в нее собраны остальные таблицы. Содержимое таблиц defgoto, pgoto, defact, и pact является индексом начала данных в table. К нему следует добавить смещение – второй компонент ключа. Например, чтобы выяснить есть ли переход из состояния с номером N после свертки нетерминала с номером M, нужно взять значение из таблицы table с нидексом pgoto[M] + N. Таким образом индексы M и N являются первым и вторым ключами соответственно. Полученный индекс не должен выходить за пределы table, иначе значение не существует. Для проверки существования значения при правильном индексе используется таблица check. Она равна по размеру table, и содержит проверочное значение. Семантика значения, и проверочного значения, определяется индексирующей таблицей. Например, для pgoto значение – это состояние, в которое необходимо перейти после свертки, а проверочное значение – номер нетерминала.

Таблицы defgoto и pgoto подобны. pgoto является ассоциативным массивом, в котором ключом являются номер состояния и номер нетерминала, полученного в результате свертки, а значением – номер состояния в которое нужно перейти после свертки. defgoto – ассоциативный массив, в котором ключ – номер состояния, значение – номер состояние в которое нужно перейти после свертки. Сначала переходное состояние ищется в pgoto, а после неудачи в defgoto.

Таблицы defact и pact похожи по тому же принципу что и defgoto с pgoto. pact – массив с ключом из пары номер состояния и номер текущего символа, и значением номера действия. Если номер отрицателен, значит действие – свертка (обратное значение номера указывает на правило свертки), иначе – сдвиг, и номер означает номер переходного состояния.

defact – номер свертки, которая выполняется при неудачном поиске в таблице pact. Номер в defact всегда положителен.

Алгоритм работы автомата:

П1 стек состояний = [0], стек символов = [ ]

П2 если действие в таблице pact отсутствует, переход в П5

П3 если значение в pact отрицательное, правило свертки = -pact, переход в П7

П4 если ошибок >= 0, ошибок уменьшается на 1

П5 выполняется сдвиг в стек текущего символа, в стек состояний добавляется значение pact, переход в П2

П6 если defact = 0, переход в П13

П7 правило свертки = defact

П8 выполняется свертка: из стека символов удаляются последние символы в количестве, равном длине правила (количество символов), в стек символов помещается символ нетерминала, производного от правила свертки

П9 если вершина стека состояний = конечному состоянию, переход П21

П10 стек состояний уменьшается на величину длины правила

П11 если есть значение в pgoto по ключу <вершина стека, номер полученного нетерминала>, в стек состояний добавляется это значение, переход в П2

П12 в стек состояний добавляется defgoto, переход в П2

П13 если ошибок != 3, переход П16

П14 если конец ввода, переход в П20

П15 если текущий символ корректен и не является концом ввода, отбросить текущий символ предпросмотра

П16 ошибок = 3

П17 если есть значение в pact по ключу <вершина стека состояний, номер терминала ошибка> и значение > 0, в стек состояний добавляется значение, переход П2

П18 если стек состояний пуст, переход в П20

П19 удалить вершину стека состояний, переход в П17

П20 возврат ошибки

П21 конец

Примечание: правила свертки индексируется с 1

Алгоритм был переделан для поддержки пошагового выполнения: места перехода в П2 заменены на возврат из функции шага. Таким образом, за 1 шаг считается успешная свертка, успешный сдвиг, откат по стеку состояний до правила ошибки, либо отброс символа по правилу ошибки. В прочих случаях шаг не выполняется и состояние отладчика не меняется.

Отладчик генерирует следующие сообщения:

- EVT_DEBUG_START – запуск отладчика;
- EVT_DEBUG_STOP – остановка отладчика;
- EVT_DEBUG_ERROR – ошибка в последнем шаге;
- EVT_DEBUG_MSG – информационное сообщение от отладчика;
- EVT_DEBUG_REDUCE – шаг закончился сверткой;
- EVT_DEBUG_SHIFT – шаг закончился сдвигом;
- EVT_DEBUG_BP – сработала точка останова;
- EVT_DEBUG_STEP – успешный шаг отладчика. Включает в себя EVT_DEBUG_REDUCE, EVT_DEBUG_SHIFT.

Управление параметрами низкоуровневой трансляции выполняется с использованием класса CLLVM. В конструктор передаются текущие значения выбора оптимизации и целевой платформы из выпадающих списков. Метод BuildConfig получает на вход текущие значения и формирует файл сборки, содержащий в себе команды компиляции синтаксического анализатора, уровня оптимизации выводимого байт-кода и целевой платформы трансляции. Метод RunTranslator автоматически загружает файл сборки, который последовательно запускает консольные команды.

3.3 Основные особенности программы

Интерфейс, показанный на рисунке 3.3.1, выполнен в виде набора окон, отображающих ту или иную деятельность программы. Вверху располагается

главное меню, а ниже панель инструментов. Все окна располагаются во вкладках, заголовок вкладки соответствует типу окна:

- Bison grammar – окно редактирования спецификации Bison;
- Lexer – окно лексического анализатора;
- New File – окно тестового файла. Если файл существует на диске, заголовок содержит имя файла вместо New File;
- Output – окно вывода сообщений от Visual Bison;
- Debug – окно состояния отладчика;
- Errors – окно вывода ошибок грамматики.

Вкладки можно свободно перетаскивать в ряде вкладок. Если перетащить вкладку в пустое пространство окна, она разделит данное окно пополам по горизонтали или вертикали, в зависимости от того, куда вкладка была помещена.

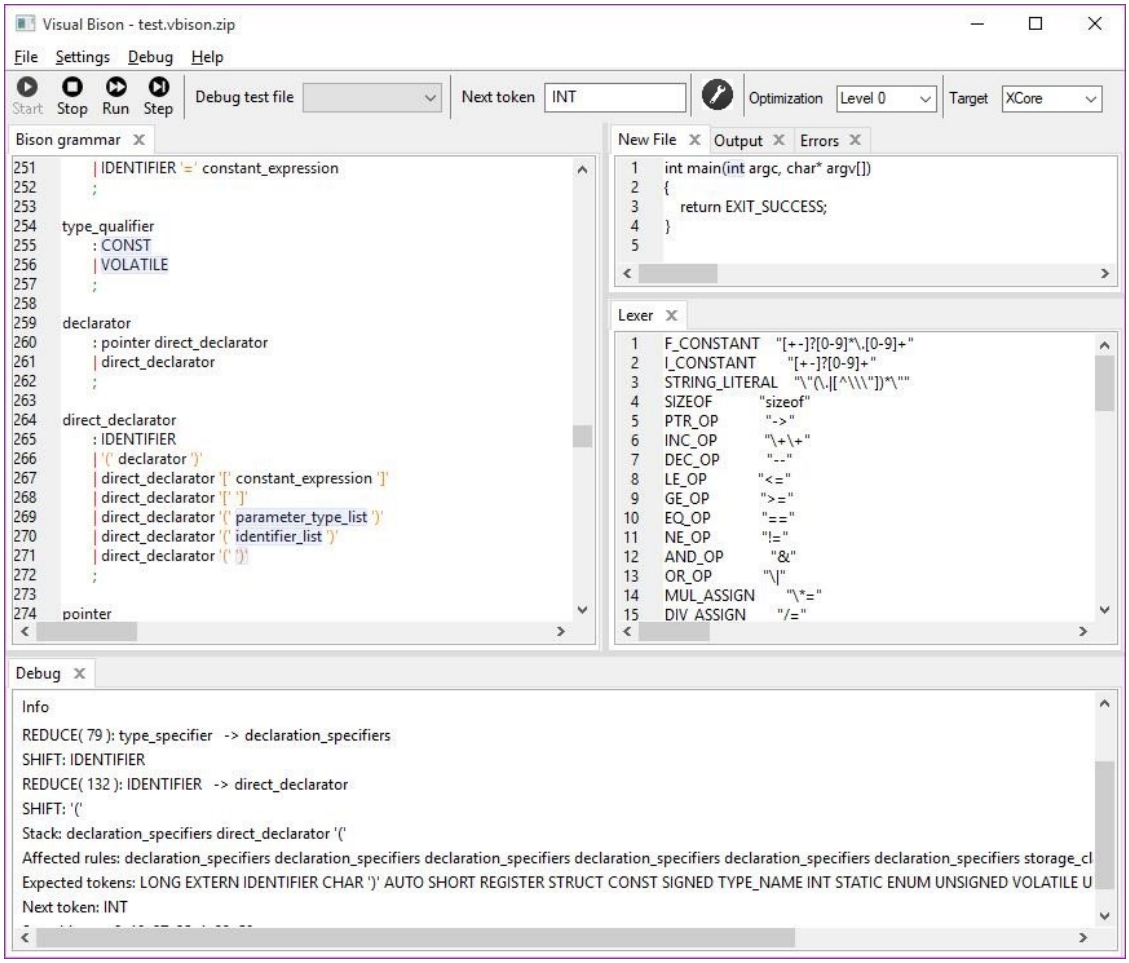

Рисунок 3.3.1 – вид главного окна

Главное меню служит инструментом передачи команд в приложение. В состав меню входят следующие пункты: File (файл), Settings (настройки), Debug (отладка), Help (помощь).

В подменю File входят команды для управления файлами проекта:

- New – создать новый проект;
- Open – открыть существующий проект;
- Save – сохранить текущий проект;
- Generate parser – создать синтаксический анализатор из спецификации грамматики;
- Open test file – открыть существующий тестовый файл;
- New test file – создать новый тестовый файл;
- Exit – выйти из программы.

В подменю Settings входят пункты:

- Preferences – настройки приложения;
- Language – выбор языка.

В подменю Debug находятся команды для отладчика грамматики:

- Start – запустить отладчик;
- Stop – остановить отладчик;
- Step – сделать шаг в отладке;
- Run – выполнять шаги до встречи ошибки или точки останова;
- Set breakpoint – установить точку останова;
- Clear breakpoint – убрать точку останова;
- Clear all breakpoints – убрать все точки останова.

В подменю Help входят пункты:

- About – отображение диалогового окна с краткой информацией о приложении.

Панель инструментов, изображенная на рисунке 3.3.2, служит для быстрого доступа к часто используемым командам, а также к полям ввода

данных, недоступных в меню. На панели инструментов располагаются кнопки:

- Start – запуск отладчика;
- Stop – остановка отладчика;
- Run – выполнять шаги до встречи ошибки или точки останова;
- Step – сделать шаг в отладке.

После метки «Debug test file» находится выпадающий список для выбора тестового файла, который будет участвовать в отладке. Далее, после метки «Next token», находится поле ввода, в котором можно изменить лексему предпросмотра. Справа от перечисленных элементов находится кнопка «Build», которая отвечает за построение и запуск проектируемого транслятора, и два выпадающих списка для выбора уровня оптимизации и целевой архитектуры.

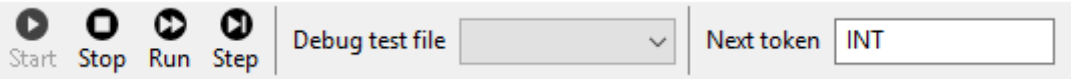

Рисунок 3.3.2 – панель инструментов

Окно лексического анализатора, показанное на рисунке 3.3.3, содержит набор правил для преобразования текста тестового файла в набор лексем для разбора грамматики. Правила имеют следующий формат:

<имя лексемы> <пробельный символ(-ы)> <регулярное выражение>

Имя лексемы может быть либо идентификатором (начинается с буквы или подчеркивания, далее может содержать цифры), либо символом (одна буква в одинарных кавычках), либо строкой (произвольный набор символов в двойных кавычках). Имя лексемы должно совпадать с именем соответствующей лексемы в файле спецификации Bison.

Регулярное выражение заключается в двойные кавычки, и имеет формат расширенного регулярного выражения POSIX.

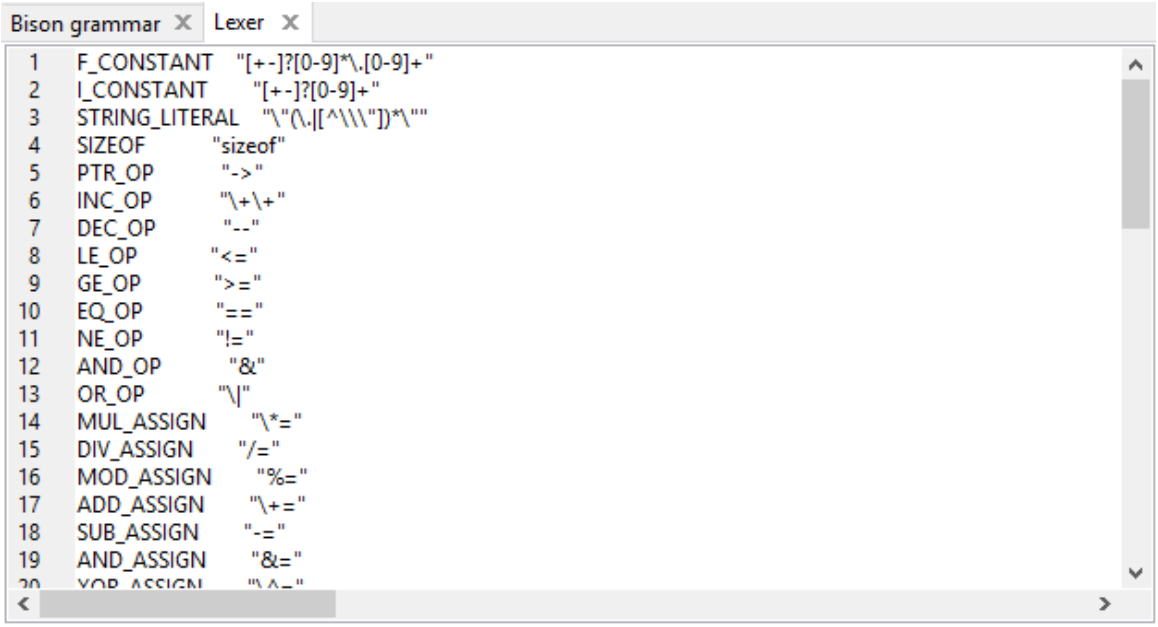

Рисунок 3.3.3 – окно лексического анализатора

В строках допустимо экранирование двойной кавычки и обратной косой черты через предварение обратной косой чертой. Перед экранируемым символом удаляется косая черта и символ считается как часть текста. Все прочие символы, идущие после обратной косой черты, не экранируются и остаются без изменений. Экранирование двойных кавычек необходимо для записи двойной кавычки в строку. Правила разделены новой строкой, и имеют приоритет в порядке от большего к меньшему. Таким образом, при разборе текста, анализатор останавливается на первом успешно проверенном регулярном выражении, и возвращает имя лексемы. При запуске отладчика, если правило составлено ошибочно, выводится сообщение об ошибке в окно вывода, и выделяется (красной чертой под текстом) ошибочная часть правила.

Окно грамматики, показанное на рисунке 3.3.4, содержит спецификацию для Bison, доступную для редактирования. Обладает подсветкой синтаксиса. Поддерживает типичные операции правки текста:

- выделение;
- копирование;
- вставка;

- удаление.

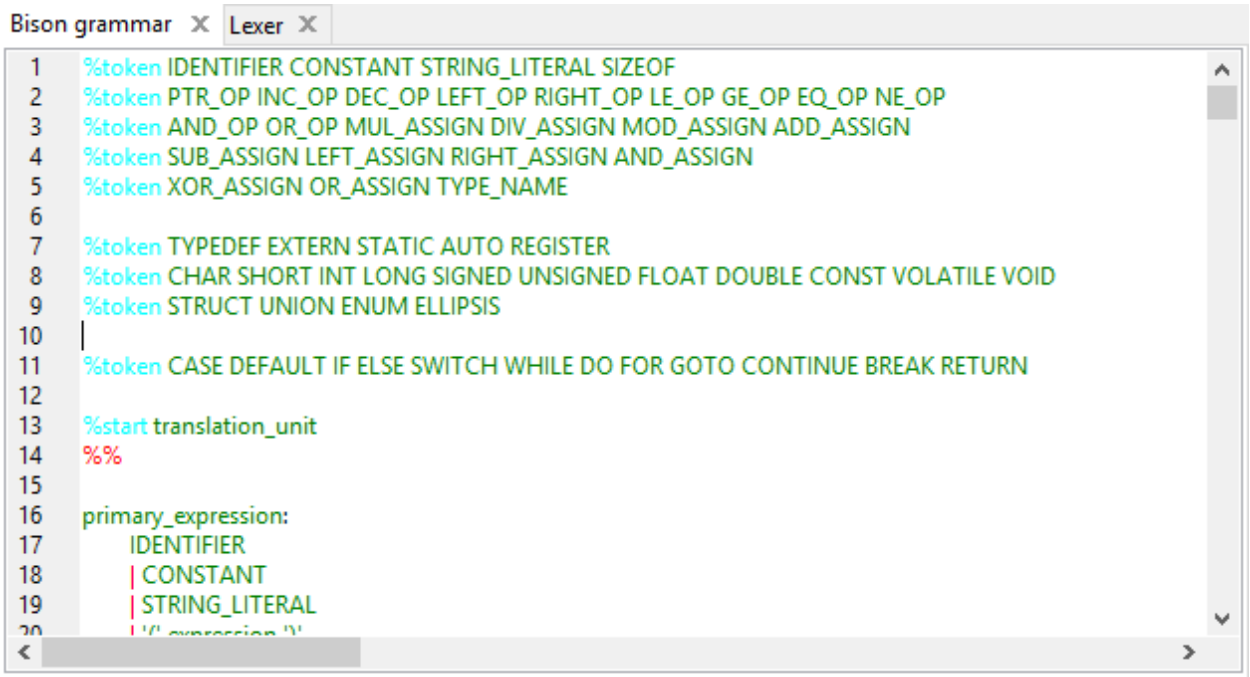

Рисунок 3.3.4 – окно грамматики

Также есть история правок, доступная через контекстное меню, либо через горячие клавиши: Ctrl-Z – для отмены, Ctrl-Y – для повтора.

Редактор поддерживает точки останова для правил. Точки ставятся на линиях, содержащих правила. Чтобы установить точку, нужно щелкнуть указателем на номере линии, либо установить курсор на эту линию, и через главное меню выбрать «Debug –> Set breakpoint». Если точка установлена на заголовке правила (на его имени), она срабатывает при свертке любого правила с этим именем, иначе только при свертке правила, находящегося на одной линии с точкой. Точки, не соответствующие ни одному из правил, подсвечиваются серым цветом, и не передаются отладчику.

В окне тестового файла, изображенное на рисунке 3.3.5, находится текст проверочного файла, который будет разбираться анализатором грамматики.

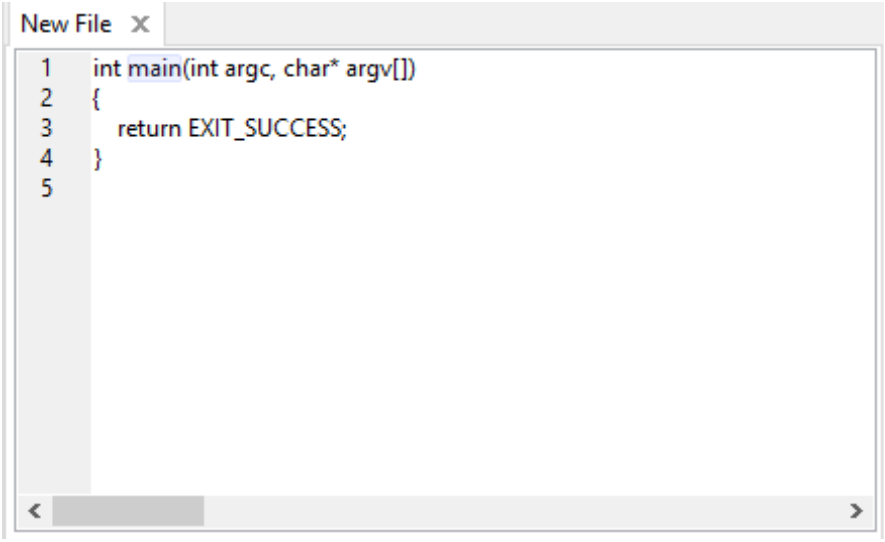

Рисунок 3.3.5 – окно тестового файла

Текст доступен для редактирования, и поддерживает те же операции с текстом что и окно грамматики. При запущенной отладке в файле выделяется следующая лексема, попадающая в стек после действия сдвиг.

Окно отладки, изображенное на рисунке 3.3.6, отображает текущее состояние отладчика, а также историю сдвигов и сверток.

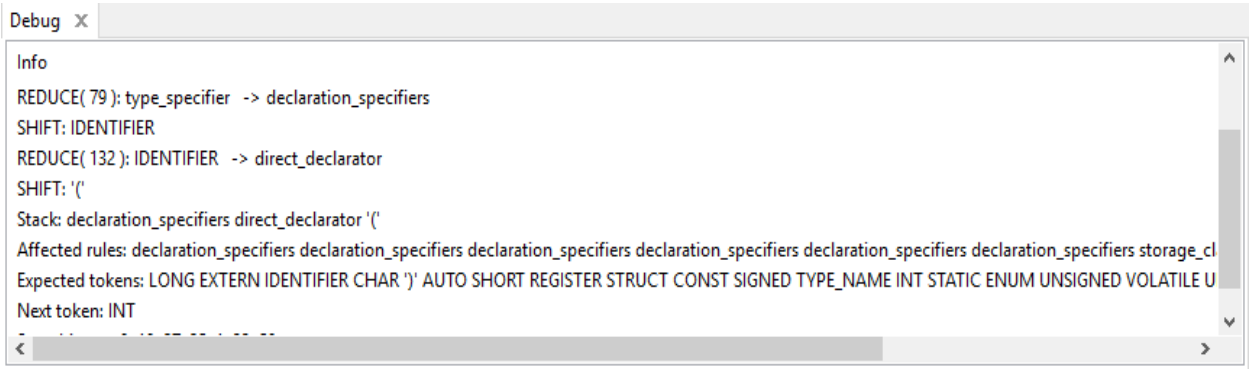

Рисунок 3.3.6 – окно отладки

Внизу располагаются 4 строки с состоянием отладчика, каждая начинается с названия состояния:

- Next token – следующая лексема, которая будет сдвинута в стек (если будет выполнено действие сдвиг);
- Expected tokens – список возможных лексем, разделенных пробелом, которые могут быть сдвинуты в стек в случае действия сдвиг;

- Affected rules – список возможных правил, разделенных пробелом, которые могут быть задействованы в случае действия свертка;
- Stack – текущее состояние стека – список лексем и нетерминалов.

Выше строки состояния стека идет история выполненных действий сдвиг/свертка. История упорядочена сверху вниз начиная с самого первого действия, заканчивая самым последним. При выделении строки с историей, в окне тестового файла выделяется лексема, или набор лексем, соответствующих данному действию. Например, при выборе свертки в тестовом файле выделятся лексемы, входящие в эту свертку.

В окне вывода сообщений, показанном на рисунке 3.3.7, находится вывод текстовых сообщений от Visual Bison. Каждое сообщение предваряется временем вывода сообщения, в формате местного времени, и типом сообщения:

- Error – ошибка;
- Warning – предупреждение;
- Info – информационное сообщение.

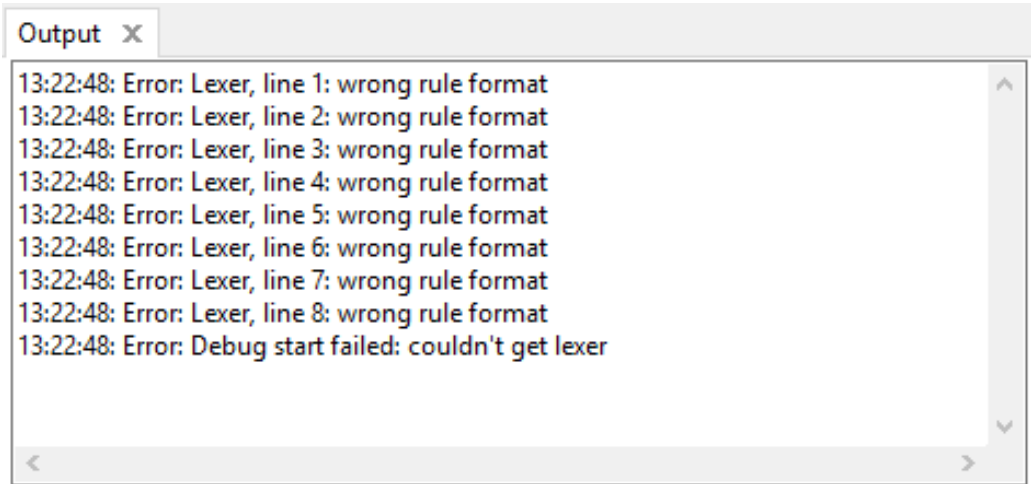

Рисунок 3.3.7 – окно вывода сообщений

В окне вывода ошибок, представленном на рисунке 3.3.8, находится список ошибок в спецификации Bison. Список разделен на три поля:

- Type – тип ошибки, может принимать значения Error (ошибка) и
- Warning (предупреждение);
- Line – номер линии, на которой обнаружена ошибка. Может быть пустым, если ошибка относится ко всему файлу;
- Description – текстовое описание ошибки.

При выделении элемента списка, в окне спецификации происходит перенос курсора на линию с этой ошибкой.

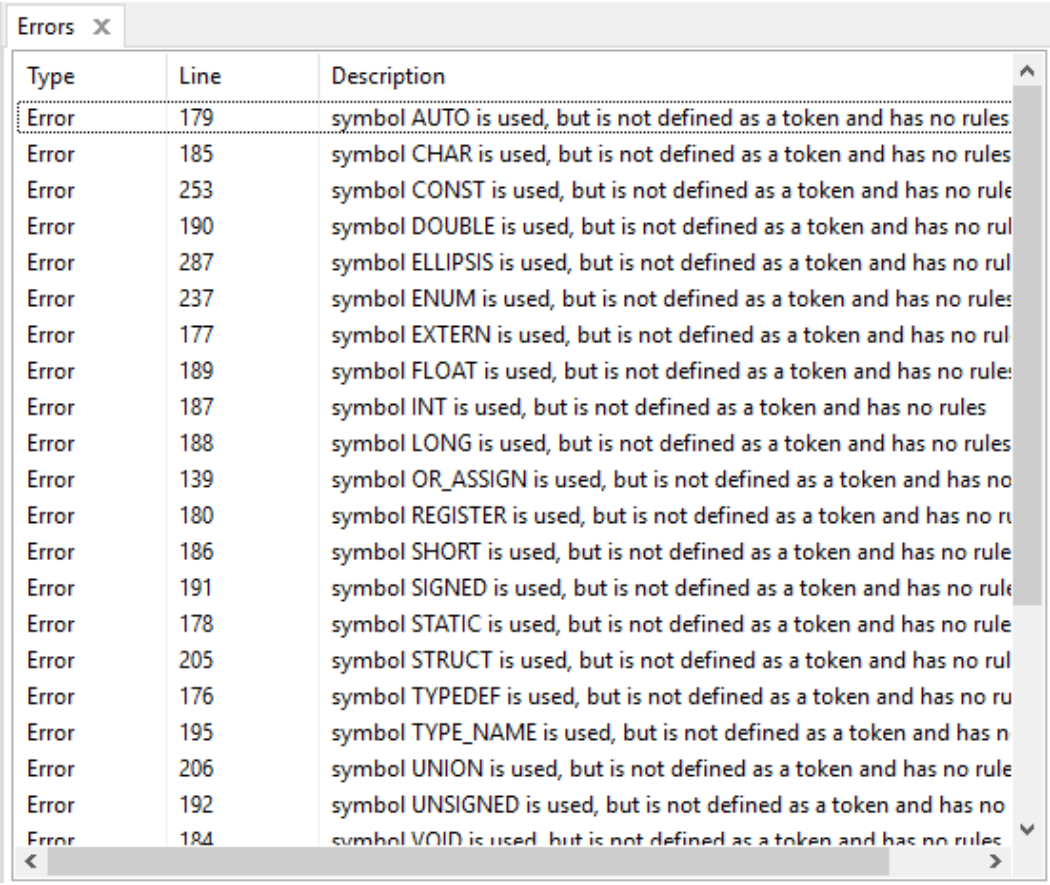

Рисунок 3.3.8 – окно вывода ошибок

Приложение поддерживает набор горячих клавиш. Для отладки:

- F1 – шаг отладчика;
- F5 – запуск отладчика;
- F8 – остановка отладчика;
- Ctrl-S – сохранение.

В редакторах:

- Ctrl+V – вставка текста из буфера обмена;
- Ctrl+C – копирование текста в буфер обмена;
- Ctrl+Z – отмена последней правки;
- Ctrl+Y – повтор отмененной правки;
- Ctrl+A – выделение всего текста.

3.4 Применение программной среды визуального прототипирования синтаксически-управляемых трансляторов

На основе современных средств визуальной разработки генераторов синтаксических анализаторов и предложенной модели взаимодействия с компиляторной инфраструктурой LLVM, реализована программная среда визуальной разработки синтаксически-управляемых трансляторов. Среда разработки направлена на покрытие всего цикла разработки трансляторов, а также полезности при изучении работы трансляторов с учётом экономической доступности данной среды. Это достигается за счёт возможности написания полноценного транслятора по его визуальному прототипу, отлаживаемой генерацией синтаксических анализаторов и работы с промежуточным кодом.

Работа пользователя начинается с написания flex спецификации. Пользователь объявляет конструкции языка слева и соответствующие им токены синтаксического дерева справа. Написание прототипа лексера сопровождается подсветкой синтаксиса языка flex. Аналогично осуществляется написание грамматик Bison. Пользователю достаточно описать конструкцию синтаксического дерева. В грамматике допускается объявление встроенной библиотеки GenerateCode. Посредством библиотеки разрабатываемый синтаксический анализатор способен генерировать промежуточный код LLVM. В ветвях дерева грамматики Bison объявляются экземпляры классов, обозначающих генерируемые дочерние элементы дерева LLVM. Редактор грамматик поддерживает подсветку синтаксиса,

автоматическую вставку кода и выделение ошибок спецификации в реальном времени. Как продемонстрировано на рисунке 3.4.1, такая визуализация грамматик позволяет упростить разработку трансляторов.

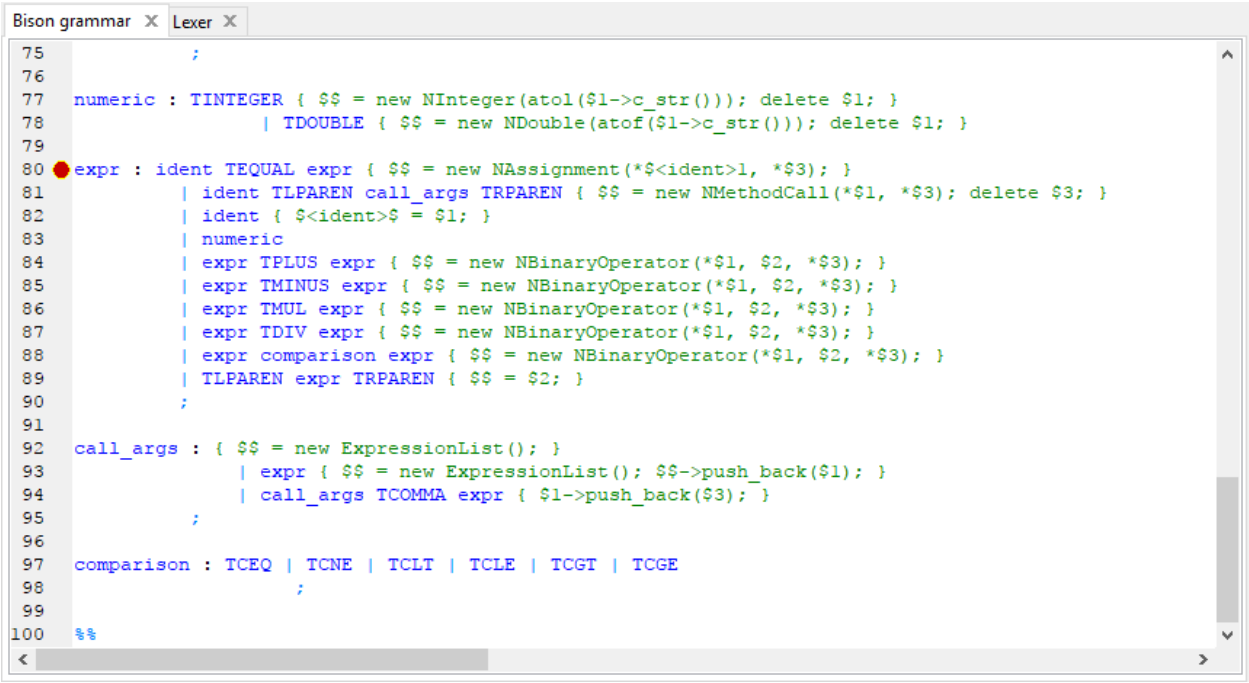

Рисунок 3.4.1 – Написание грамматики в редакторе

Пользователь также может ставить точки останова и пошагово выполнять отладку грамматики, с возможностью менять значения токенов в процессе отладки. Во время отладки, в окне отладчика выводятся сообщения о переносах и свёртках в стеке. Благодаря взаимодействию с flex и Bison среда разработки генерирует полноценный анализатор исходного кода, способный генерировать байт-код. Для создания полноценного транслятора, способного переводить полученный байт-код в другой язык программирования или ассемблер конкретной процессорной архитектуры, в среде разработки задаются параметры целевой платформы и, если это необходимо, уровень оптимизации. Полученный транслятор будет взаимодействовать с системой LLVM, передавая консольные команды и получая готовые файлы. Готовый транслятор будет содержать в себе лексический и синтаксический анализаторы и класс генерации байт-кода на

языке C++, а также файл сборки с консольными командами, как показано на рисунке 3.4.2.

Рисунок 3.4.2 – Файлы проекта транслятора

Созданная интегрированная среда разработки, использует модель, изложенную в разделе 2.2, и охватывает весь цикл разработки синтаксически-управляемого транслятора, состав которого описан в разделе 1.1. Благодаря графическому выделению синтаксических конструкций, визуализаций сдвигов и свёрток в стеке, а также независимому синтаксическому анализу, программный инструмент может быть полезен при изучении трансляторов. Кроме того, бесплатность данного инструмента позволяет использовать его без финансовых затрат, а открытый исходный код даёт возможность доработки программы пользователем.

Выводы к главе 3. Были выделены необходимые функции и поставлены задачи разрабатываемой программной системы. Рассмотрена архитектура и интерфейс программной системы. Показана полезность использованной модели применения алгоритма генерации промежуточного кода.

ЗАКЛЮЧЕНИЕ

В данной работе приведен анализ проблемы инструментов визуального прототипирования синтаксически-управляемых трансляторов. Предложена модель применения генерации промежуточного кода LLVM по входной спецификации GNU Bison. Рассматриваются возможности применения этого метода для взаимодействия с компиляторной инфраструктурой в программном обеспечении визуального построения синтаксически-управляемых трансляторов. Разработан программный продукт, позволяющий редактировать входные спецификации и генерировать синтаксические анализаторы, работающие в связке с компиляторной инфраструктурой.

В заключении необходимо отметить, что предложенные алгоритмы и реализации являются эффективным средством инструментальной поддержки визуального прототипирования синтаксически-управляемых трансляторов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Богачук, А. Л. Инструментальная поддержка визуального прототипирования синтаксически-управляемых трансляторов / А. Л. Богачук // Actualscience. – 2016. - №12. – С108.

2. Вигерс, К. Разработка требований к программному обеспечению: учебное пособие / К. Вигерс, Дж. Битти. – СПб: БХВ-Петербург, 2004. – 254с.

3. Вирт, Н. Построение компиляторов: учебное пособие / Н. Вирт. - Москва: ДМК-Пресс, 2010. — 192 с.

4. Карпов, Ю.Г. Теория и технология программирования. Основы построения трансляторов: учебное пособие / Ю.Г. Карпов. - СПб.: БХВ-Петербург, 2005. — 272 с.

5. Компиляторы: принципы, технологии и инструментарий, 2-е изд.: учебное пособие / Альфред В. Ахо, Моника С. Лам, Рави Сети, Джеффри Д. Ульфман. - Москва: издательский дом «Вильямс», 2008. – 1184 с.

6. Мартыненко, Б. К. Языки и трансляции: учебное пособие / Б.К. Мартыненко. - СПб.: Издательство С.-Петерб. ун-та, 2008. — 257 с.

7. Мозговой, М. В. Классика программирования: алгоритмы, языки, автоматы, компиляторы. Практический подход: учебное пособие / М. В. Мозговой - СПб.: Наука и Техника, 2006. — 272 с.

8. Молдованова, О. В. Языки программирования и методы трансляции.: учебное пособие / О.В. Молдованова. - Новосибирск/СибГУТИ, 2012. – 134с.

9. Опалева, Э. А., Самойленко В. П. Языки программирования и методы трансляции: учебное пособие / Э. А. Опалева, В. П. Самойленко. - СПб.:БХВ-Петербург, 2005. — 480 с.

10. Рейуорд-Смит, В. Дж. Теория формальных языков. Вводный курс: учебное пособие / В. Дж. Рейуорд-Смит. - Москва: Радио и связь, 1988. — 128 с.

11. Свердлов, С. З. Языки программирования и методы трансляции: учебное пособие / С.З. Свердлов. – СПб.: издательство «Питер», 2007. – 638с.

12. Серебряков, В. А. Основы конструирования компиляторов: учебное пособие / В.А. Серебряков, М.П. Галочкин. – Москва: издательство «Едиториал УРСС», 1999. – 193 с.

13. Серебряков, В. А. Теория и реализация языков программирования: учебное пособие / В.А. Серебряков, М.П. Галочкин. – Москва: издательство «МЗ Пресс», 2006. – 10 с.

14. Страуструп Б. Язык программирования C++, 3-е изд.: учебное пособие / пер. с англ. – Москва: издательство «Бином», 1999. – 991 с.

15. ANTLR [Электронный ресурс] // Генератор синтаксических анализа торов. – Режим доступа: http://www.antlr.org/

16. flex: The Fast Lexical Analyzer [Электронный ресурс] // Генератор лексических анализаторов. – Режим доступа: http://flex.sourceforge.net/.

17. GNU Bison [Электронный ресурс] // Документация по GNU Bison. – Режим доступа: http://www.gnu.org/software/bison

18. GCC, the GNU Compiler Collection [Электронный ресурс] // Компилятор. - Режим доступа: https://gcc.gnu.org/

19. LLVM Documentation [Электронный ресурс] // Документация по LLVM. – Режим доступа: http://llvm.org/docs/

20. Scintilla Documentation [Электронный ресурс] // Документация по компоненту Scintilla. – Режим доступа: http://www.scintilla.org/ScintillaDoc.html

21. VisualBNF [Электронный ресурс] // Генератор синтаксических анализаторов. – Режим доступа: http://www.intralogic.eu/VisualBNF

22. VisualLangLab - A Visual Parser-Generator IDE [Электронный ресурс] // Генератор синтаксических анализаторов. – Режим доступа: https://vll.java.net

23. wxWidgets [Электронный ресурс] // Документация по wxWidgets 3.0. – Режим доступа: http://docs.wxwidgets.org/3.0

ПРИЛОЖЕНИЯ

ПРИЛОЖЕНИЕ А
Перечень выводимых ошибок

Полный перечень выводимых ошибок указан в таблице А1.

Таблица А1 – Перечень выводимых ошибок

Формат сообщения	Описание
Failed to retrieve error list from bison	Не удалось получить список ошибок от Bison
Couldn't open test file	Не удалось открыть тестовый файл
Error saving file	Ошибка при сохранении файла проекта
Error opening file	Ошибка при открытии файла проекта
Error opening file: not zip archive	Ошибка при сохранении файла проекта: файл должен быть zip-архивом
Error reading file	Ошибка ввода-вывода
No test file selected	Не выбран тестовый файл для отладчика
Couldn't get automaton states from bison	Не удалось получить XML файл с состояниями от Bison
Couldn't start debugger. Couldn't get lexer	Не удалось запустить отладчик. Ошибка в лексическом анализаторе
Couldn't start debugger. Probably because of bison XML states format error.	Не удалось запустить отладчик. Возможная причина – несовместимый формат состояний
Couldn't start debugger. It's in bad state.	Не удалось запустить отладчик из-за внутренней ошибки

Полный перечень ошибок спецификации от Bison указан в таблице А2.

Таблица А2 – перечень ошибок спецификации Bison

Формат сообщения	Описание
shift/reduce conflicts: %d found, %d expected	Не удалось получить список ошибок от Bison
%expect-rr applies only to GLR parsers	Директива %expect-rr применима только к GLR парсерам
{num} shift/reduce conflict	{num} конфликтов сдвиг/свертка
%define variable {name} redefined	Переменная в директиве %define с именем {name} переопределена
%define variable '{name}' requires '{...}' values	Переменной в директиве %define с именем {name} требуется блок кода '{...}' в качестве значения

Продолжение таблицы А2

Формат сообщения	Описание
%define variable '{name}' requires keyword values	Переменной в директиве %define с именем {name} требуется ключевое слово в качестве значения
%define variable '{name}' requires '"..."' values	Переменной в директиве %define с именем {name} требуется строка '"..."' в качестве значения
invalid value for %define Boolean variable {name}	Неправильное значение булевой переменной с именем {name} в директиве %define
missing identifier in parameter declaration	Отсутствует идентификатор в объявлении параметра
require bison {version-req}, but have {version-have}	Требуется версия Bison {version-req}, а используется {version-have}
rule is too long	Длина правила слишком большая
no rules in the input grammar	Отсутствуют правила в грамматике
multiple {directive} declarations	Множественные объявления директивы {directive}
result type clash on merge function {func}: <{type1}> != <{type2}>	Конфликт типов результата функции {func}
duplicated symbol name for {sym} ignored	Дубликат символа с именем {sym} не учитывается
rule given for {sym}, which is a token	Определено правило для символа {sym}, который является терминалом
type clash on default action: <{type1}> != <{type2}>	Конфликт типов на действии по умолчанию ($$ = $1)
empty rule for typed nonterminal, and no action	Пустое правило для типизированного нетерминала, отсутствует действие
unused value: ${num}	Значение ${num} в действии не используется
unset value: $$	Значение $$ не задано
%empty on non-empty rule	Директива %empty в не пустом правиле
empty rule without %empty	Пустое правиле без директивы %empty
token for %prec is not defined: {sym}	Терминал {sym} не найден для директивы %prec
%{dir} affects only GLR parsers	Директива %{dir} имеет смысл только для GLR парсера

Продолжение таблицы А2

Формат сообщения	Описание
%dprec must be followed by positive number	После директивы %dprec должно следовать положительное число
nonterminal useless in grammar: {sym}	Нетерминал {sym} бесполезен в грамматике
{num} rule useless in grammar	Правило с номером {num} бесполезно в грамматике
start symbol {sym} does not derive any sentence	Начальный символ {sym} пуст
integer out of range: {lit}	Целое число {lit} в коде выходит за границы
invalid reference: {ref}	Неверная ссылка к {ref} в коде
syntax error after '{dollar-or-at}', expecting integer, letter, '_', '[', or '$'	Синтаксическая ошибка после знака $ или @ {dollar-or-at}
symbol not found in production before ${pos}: {sym}	Символ {sym} не найден в текущем правиле в позиции {pos}
symbol not found in production: {sym}	Символ {sym} не найден в текущем правиле
misleading reference: {sym}	Ссылка {sym} ни к чему не ведет
ambiguous reference: {sym}	Неоднозначность ссылки {sym}
explicit type given in untyped grammar	Тип указан явно в не типизированной грамматике
$$ for the midrule at ${pos} of {rule} has no declared type	Значение $$ внутри правила {rule} в позиции ${pos} правила не имеет типа
$$ of {rule} has no declared type	Значение $$ правила {rule} не имеет типа
${num} of {rule} has no declared type	Значение ${num} правила {rule} не имеет типа
stray ',' treated as white space	Литерал ‘,’ засчитан как пробельный символ
{rule} rule useless in grammar	Правило {rule} бесполезно в грамматике
invalid directive: {name}	Неизвестная директива {name}
invalid identifier: {name}	Неправильный идентификатор
invalid null character	Неправильный символ \0
unexpected identifier in bracketed name: {name}	Неожиданный идентификатор {name} в имени в скобках
an identifier expected	Неожиданный идентификатор в этом месте
invalid character in bracketed name: {sym}	Неправильный символ {sym} в имени в квадратных скобках

Продолжение таблицы А2

Формат сообщения	Описание
empty character literal	Пустой строковый литерал
extra characters in character literal	Лишние символы в символьном литерале
invalid number after \-escape: {num}	Неправильное число {num} после экранирования \
invalid character after \-escape: {sym}	Неправильный символ {sym} после экранирования \
missing {sym} at end of line	Неожиданный конец линии, ожидался символ {sym}
Missing {sym} at end of file	Неожиданный конец файла, ожидался символ {sym}
POSIX Yacc forbids dashes in symbol names: {sym}	POSIX стандарт для Yacc запрещает использование тире в именах: {sym}
too many symbols in input grammar (limit is {num})	Слишком много символов в грамматике (предел – {num})
symbol %s redefined	Повторное определение символа {sym}
symbol {sym} redeclared	Повторно объявление символа {sym}
redefining user token number of {sym}	Переопределение номера пользовательского терминала для символа {sym}
symbol {sym} is used, but is not defined as a token and has no rules	Символ {sym} используется, но не объявлен терминалом или правилом
useless {prop} for type <{type}>	Бесполезное свойство {prop} для типа {type}
type <{type}> is used, but is not associated to any symbol	Тип {type} используется, но не связан ни с каким символом
symbol {sym} used more than once as a literal string	Символ {sym} встречается более одного раза в качестве строкового литерала
symbol %s given more than one literal string	Символу {sym} назначено более одной литеральной строки
user token number {num} redeclaration for {sym}	Пользовательский терминал под номером {num} переопределен для символа {sym}
the start symbol {sym} is undefined	Начальный символ {sym} не определен
the start symbol {sym} is a token	Начальный символ {sym} является терминалом
useless precedence and associativity for {sym}	Бесполезный порядок и ассоциативность для символа {sym}
useless precedence for {sym}	Бесполезный порядок для символа {sym}
useless associativity for {sym}, use %precedence	Бесполезная ассоциативность для символа {sym}