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

Институт космических и информационных технологий
институт
Кафедра «Информатика»
кафедра

УТВЕРЖДАЮ
Заведующий кафедрой
_________ А.И. Рубан
подпись инициалы, фамилия
«_____» _________ 2016 г.

ДИПЛОМНЫЙ ПРОЕКТ

230105.65 Программное обеспечение вычислительной техники и автоматизированных систем
код и наименование специальности

Визуальный редактор-генератор синтаксических анализаторов Visual Bison
тема

Пояснительная записка

Руководитель	___________	доцент, к.т.н.	А.С. Кузнецов
	подпись, дата	должность, ученая степень	инициалы, фамилия
Нормоконтролер	___________	доцент, к.т.н.	О.А. Антамошкин
	подпись, дата	должность, ученая степень	инициалы, фамилия
Выпускник	___________		Р.В. Карпов
	подпись, дата		инициалы, фамилия

Красноярск 2016

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

Институт космических и информационных технологий
институт
Кафедра «Информатика»
кафедра

УТВЕРЖДАЮ
Заведующий кафедрой
________ А.И. Рубан
подпись инициалы, фамилия
«_____ _ _» _________ 2016 г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ
в форме дипломного проекта

Студенту Карпову Роману Владимировичу

Группа ЗКИ10-05 Направление (специальность) 230105.65, Программное обеспечение вычислительной техники и автоматизированных систем.

Тема выпускной квалификационной работы: «Визуальный редактор-генератор синтаксических анализаторов Visual Bison».

Утверждена приказом по университету №4202/с от 28.03.16.

Руководитель ВКР А.С. Кузнецов, доцент кафедры «Информатика», канд. техн. наук.

Исходные данные для ВКР: спроектировать и разработать визуальный редактор-генератор синтаксических анализаторов Visual Bison.

Перечень разделов ВКР:

- введение;
- общие сведения;
- описание приложения;
- техническая реализация;
- заключение;

Перечень графического материала: презентационные слайды PowerPoint.

Руководитель ВКР ______________ А.С. Кузнецов

подпись — инициалы и фамилия

Задание принял к исполнению ______________ Р.В. Карпов

подпись, инициалы и фамилия студента

« ___ » __________ 2016 г.

АННОТАЦИЯ

Выпускная квалификационная работа по теме «Визуальный редактор-генератор синтаксических анализаторов Visual Bison» содержит 47 страниц текстового документа, 9 рисунков, 4 библиографических источника.

Целью работы являлась разработка визуального редактора-генератора синтаксических анализаторов Visual Bison.

В дипломный проект входит введение, три главы и заключение.

Во введении определяется необходимость реализации и основные задачи проекта.

В первой главе более подробно описана доступная теория, связанная с разработкой приложения.

Во второй главе идет описание приложения, основных понятий и возможностей.

В третьей главе дается описание системы, архитектуры приложения, обоснование выбора языка реализации и функциональный состав.

Заключение посвящено подведению итогов по всей проделанной работе.

<table>
<tr><td></td><td></td><td></td><td></td><td></td><td colspan="3" rowspan="3">ДП-230105.65ПЗ</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>Изм.</td><td>Лист</td><td>№ докум.</td><td>Подп.</td><td>Дата</td></tr>
<tr><td colspan="2">Разраб.</td><td>Р.В. Карпов</td><td></td><td></td><td rowspan="5">Визуальный редактор-генератор синтаксических анализаторов Visual Bison</td><td>Лит.</td><td>Лист</td><td>Листов</td></tr>
<tr><td colspan="2">Пров.</td><td>А.С. Кузнецов</td><td></td><td></td><td></td><td>4</td><td>47</td></tr>
<tr><td colspan="2"></td><td></td><td></td><td></td><td colspan="3" rowspan="3">Кафедра «Информатика»</td></tr>
<tr><td colspan="2">Н. контр.</td><td>О.А.Антамошкин</td><td></td><td></td></tr>
<tr><td colspan="2">Утв.</td><td>А.И. Рубан</td><td></td><td></td></tr>
</table>

СОДЕРЖАНИЕ

ВВЕДЕНИЕ ... 7
1 Глава. Общие сведения ... 8
1.1 GNU Bison ... 8
1.2 Контекстно-свободные грамматики ... 9
1.3 Формальная грамматика ... 12
1.4 Входная спецификация Bison ... 13
1.5 Библиотека wxWidgets ... 14
2 Глава. Описание приложения ... 17
2.1 Интерфейс ... 17
2.1.1 Общий вид ... 17
2.1.2 Главное меню ... 18
2.1.3 Панель инструментов ... 20
2.1.4 Окно грамматики ... 20
2.1.5 Окно лексического анализатора ... 21
2.1.6 Окно отладки ... 23
2.1.7 Окно вывода сообщений ... 24
2.1.8 Окно тестового файла ... 25
2.1.9 Окно вывода ошибок ... 26
2.1.10 Горячие клавиши ... 27
2.2 Перечень выводимых ошибок ... 27
2.3 Перечень ошибок в спецификации ... 28
3 Глава. Техническая реализация ... 33

3.1 Язык и библиотеки 33
3.2 Архитектура 33
3.3 Редактор грамматики 34
3.4 Редактор тестового файла 36
3.5 Проверка ошибок спецификации 36
3.6 Отладчик 38
3.7 Лексический анализатор 41
ЗАКЛЮЧЕНИЕ 43
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 44
ПРИЛОЖЕНИЕ А 45

ВВЕДЕНИЕ

Есть необходимость для визуального «прототипирования» синтаксически-управляемых трансляторов на основе GNU Bison (далее — bison). Bison умеет генерировать исходные тексты синтаксических анализаторов на C/C++/Java по заданной спецификации в формате, похожем на контекстно-свободные грамматики Хомского. На сегодня имеются несколько подобных инструментов, и они либо коммерческие, либо не бесплатные (например, Visual BNF). Возникает необходимость в бесплатном инструменте с GPL- или BSD-подобной лицензией.

Приложение должно выполнять функции:

- Написание bison-грамматики с подсветкой синтаксиса и генерацией по ней кода синтаксического анализатора;
- Выделение конфликтов по мере написания грамматики (в стиле Eclipse или MSVS);
- Пошаговое выполнение на тестовом входном файле и визуализация переносов и сверток в стеке;
- Использование точек останова в тестовом файле или грамматике;
- Проверка символов предпросмотра в каждом состоянии и изменение ввода перед каждым последующим шагом синтаксического анализа;

Данная работа пока представляет академический интерес, но может использоваться в учебном процессе при обучении студентов по «комипляторным курсам», и в отдаленной перспективе — при промышленной разработке компиляторов.

1 Глава. Общие сведения

1.1 GNU Bison

Bison – это GNU-выпуск известной программы YACC, предназначенной для порождения компиляторов по описанной пользователем КС-грамматике.

Одним из следствий расширения возможностей и доступности использования вычислительной техники стало производство прикладных программных систем со всё более разнообразным и сложным поведением. При этом краеугольный принцип кибернетики - принцип необходимого и достаточного разнообразия, сформулированный Н. Виннером и Р. Эшби ещё до 1960 г., гласит, что степень разнообразия управления подобной сущностью должна соответствовать степени разнообразия её поведения.

С какого-то уровня разнообразия (сложности) управления такой системой оказывается проще и надёжнее предложить пользователю строго определённый формальный язык, с помощью которого пользователь сообщает системе задание на выполнение тех или иных её основных задач, в т.ч. определяет состав и периодичность выдачи учётных и уведомительных сообщений о ходе этого выполнения и т.п.

К примеру, для управления воздушным движением, в т.ч. в Российской Федерации, соответствующими международными организациями разработан и используется формальный язык из нескольких десятков предложений, однозначно сообщающих в автоматизированном и ручном режимах о каждом достаточно значимом событии в использовании самолёта и его характеристиках: взлёте, изменении курса, прохождении контрольной точки полёта, посадке, возникновении различных иных вероятных обстоятельств, которые можно и нужно предвидеть. Обычным является дополнение и уточнение время от времени этого списка команд и/или их параметров по решению соответствующих комитетов. Эти изменения необходимо быстро и надёжно вносить в

соответствующие системы управления.

Для проверки правильности и разбора предложений подобных формальных языков, как известно, используют программу, именуемую компилятор.

Если мы такой компилятор научимся получать автоматически на основе описывающей этот язык грамматики, то сможем все изменения (например, дополнения) к пользовательскому языку, вносить очень быстро и при этом, что очень важно, вполне надёжно.

Важность и всё большая распространённость этой задачи была осознана в программировании довольно быстро и поэтому появился ряд разнообразных программных инструментов для автоматизированной разработки компиляторов для языков, определяемых контекстно-свободными грамматиками. Так, само название известного более двух десятилетий компилятора компиляторов YACC переводится как «ещё один компилятор компиляторов» (Yet another compiler compiler). В качестве пробы сил и оттачивания мастерства производство подобных систем продолжается и сейчас, особенно в технических университетах.

1.2 Контекстно-свободные грамматики

Для того, чтобы Bison мог разобрать программу на каком-то языке, этот язык должен быть описан контекстно-свободной грамматикой [1]. Это означает, что вы определяете одну или более синтаксических групп и задаёте правила их сборки из составных частей. Например, в языке C одна из групп называется `выражение'. Правило для составления выражения может выглядеть так: "Выражение может состоять из знака `минус' и другого выражения". Другое правило: "Выражением может быть целое число". Как вы может видеть, правила часто бывают рекурсивными, но должно быть по крайней мере одно правило, выводящее из рекурсии.

Наиболее распространённой формальной системой для представления таких правил в удобном для человека виде является форма Бэкуса-Наура (БНФ, Backus-Naur Form, BNF), которая была разработана для описания языка Algol 60. Любая грамматика, выраженная в форме Бэкуса-Наура является контекстно-свободной грамматикой. Bison принимает на вход, в сущности, особый вид БНФ, адаптированный для машинной обработки.

Bison может работать не со всеми контекстно-свободными грамматиками, а только с грамматиками класса LALR(1). Коротко, это означает, что должно быть возможно определить, как разобрать любую часть входа, заглядывая вперёд не более, чем на одну лексему. Строго говоря, это описание LR(1)-грамматики, класс LALR(1) имеет дополнительные ограничения, которые не так просто объяснить. Но в обычной практике редко встречаются LR(1)-грамматики, которые не являются LALR(1).

В правилах формальной грамматики языка каждый вид синтаксических единиц или групп называется символом. Те из них, которые формируются группировкой меньших конструкций в соответствии с правилами грамматики, называются нетерминальными символами, а те, что не могут разбиты -- терминальными символами или типами лексем. Мы называем часть входного текста, соответствующую одному терминальному символу лексемой, а соответствующую нетерминальному символу – группой.

Для примера терминальных и нетерминальных символов можно использовать язык C. Лексемами C являются идентификаторы, константы (числовые и строковые), и различные ключевые слова, знаки арифметических операций и пунктуации. Таким образом, терминальные символы грамматики C это: `идентификатор', `число', `строка' и по одному символу на каждое ключевое слово, знак операции или пунктуации: `if', 'return', `const', `static', `int', `char', `знак плюс', `открывающая скобка', `закрывающая скобка', `запятая' и многие другие (эти лексемы могут быть разбиты на литеры, но это уже вопрос составления

словарей, а не грамматики).

Синтаксические группы C это: выражение, оператор, объявление и определение функции. Они представлены в грамматике C нетерминальными символами `выражение', `оператор', `объявление' и `определение функции'. Полная грамматика, для того, чтобы выразить смысл этих четырёх, использует десятки дополнительных языковых конструкций, каждой из которых соответствует свой нетерминальный символ. Пример выше является определением функции, он содержит одно объявление и один оператор. В операторе каждое `x', так же, как и `x * x' являются выражениями.

Каждому нетерминальному символу должны быть сопоставлены правила грамматики, показывающие, как он собирается из более простых конструкций. Например, одним из операторов C является оператор return, это может быть описано правилом грамматики, неформально читающимся так:

`Оператор' может состоять из ключевого слова `return', `выражения' и `точки с запятой'.

Должно существовать множество других правил для `оператор', по одному на каждый вид оператора C.

Один нетерминальный символ должен быть отмечен как специальный, определяющий завершённое высказывание на языке. Он называется начальным символом. В компиляторе это означает полную программу на входе. В языке C эту роль играет нетерминальный символ `последовательность определений и объявлений'.

Например, 1 + 2 является правильным выражением C – правильной частью программы на C – но не является правильной целой программой на C. В контекстно-свободной грамматике C это следует из того, что `выражение' не является начальным символом.

Анализатор Bison читает на входе последовательность лексем и группирует их, используя правила грамматики. Если вход правилен, конечным результатом

будет свёртка всей последовательности лексем в одну группу, которой соответствует начальный символ грамматики. Если мы используем грамматику C, весь входной текст в целом должен быть `последовательностью определений и объявлений'. Если это не так, анализатор сообщит о синтаксической ошибке.

1.3 Формальная грамматика

Формальная грамматика – это математическая конструкция. Чтобы определить язык для Bison, вы должны написать файл, описывающий грамматику в синтаксисе Bison – файл грамматики Bison.

Нетерминальный символ формальной грамматики на входе Bison представляется идентификатором, таким же как идентификатор C. По соглашению их нужно записывать в нижнем регистре, например: expr, stmt или declaration.

Представление в Bison нетерминальных символов также называется типом лексем. Типы лексем также могут быть представлены идентификаторами в стиле C. По соглашению эти идентификаторы следует записывать в верхнем регистре, чтобы отличить их от нетерминалов, например, INTEGER, IDENTIFIER, IF, или RETURN. Терминальный символ, соответствующий конкретному ключевому слову языка следует называть так же, как это ключевое слово выглядит в верхнем регистре. Терминальный символ error зарезервирован для восстановления после ошибок.

Терминальный символ также может быть представлен как однолитерная константа, как однолитерная константа C. Вам стоит делать так всегда, когда лексема представляет собой просто одиночную литеру (скобку, знак плюс и т.д.) – используйте ту же литеру в качестве терминального символа для этой лексемы.

Третий способ представления терминального символа – представление строковой константой C из нескольких литер.

Правила грамматики также содержат выражение в синтаксисе Bison. Например, вот правило Bison для оператора C return. Точка с запятой в кавычках является однолитерной лексемой, представляющей часть синтаксиса оператора C, а отдельная точка с запятой и двоеточие являются знаками пунктуации Bison, используемыми во всех правилах.

1.4 Входная спецификация Bison

Входной файл утилиты Bison – это файл грамматики Bison. Общий вид файла грамматики Bison следующий:

```
%{
Объявления С
%}
Объявления Bison
%%
Правила грамматики
%%
Дополнительный код на С
```

`%%', `%{' и `%}' – это знаки пунктуации, присутствующие в любом файле грамматики Bison для разделения его секций.

Объявления C могут определять типы и переменные, используемые в действиях. Вы также можете использовать команды препроцессора для определения используемых там макросов и #include для включения файлов заголовков, делающих всё вышеперечисленное.

Объявления Bison задают имена терминальных и нетерминальных символов и могут также описывать приоритет операций и типы данных семантических значений различных символов.

Правила грамматики определяют, как каждый нетерминальный символ собирается из своих частей.

Дополнительный код на С может содержать любой код на С, который вы хотите использовать. Часто здесь находится определение лексического анализатора yylex и подпрограммы, вызываемые действиями правил грамматики. В простых программах здесь может находится и вся остальная часть программы.

1.5 Библиотека wxWidgets

wxWidgets (ранее известная как wxWindows) — кроссплатформенная библиотека инструментов с открытым исходным кодом для разработки кроссплатформенных на уровне исходного кода приложений [2]. Основным применением wxWidgets является построение графического интерфейса пользователя (GUI), однако библиотека включает большое количество других функций и используется для создания весьма разнообразного ПО. wxWidgets выпущена под лицензией, базирующейся на LGPL. Проект был начат в 1992 Джулианом Смартом (Julian Smart), который до сих пор является членом основной группы разработчиков.

wxWidgets — это инструмент разработчика для написания настольных или мобильных приложений с графическим интерфейсом (GUI), который экономит много времени на написание кроссплатформенных приложений и обеспечивает их стандартное поведение.

Приложения обычно показывают пользователю окна со стандартными элементами управления, изображениями и графиками; реагируют на события от мыши, клавиатуры и других источников — эти стандартные интерфейсные функции легко реализуются при использовании wxWidgets и отходят на второй план, позволяя программисту сосредоточить свои усилия на функциональности приложения. Более того. Поскольку на разных операционных системах и

устройствах по-разному могут быть реализованы и другие функции, wxWidgets включает высокоуровневые средства (наборы классов) для работы с графическими изображениями, документами в форматах XML и HTML, архивами, файловыми системами, процессами, подсистемами печати, мультимедиа, сетями, классы для организации многопоточности, конфигурирования приложений, межпроцессного взаимодействия, доступа к базам данных, отладки, отправки дампов и множество других инструментов.

Такие развитые средства библиотеки wxWidgets позволяют писать программы на базе единого API и компилировать на множестве компьютерных платформ с минимальными изменениями в исходном коде либо вообще без них. Она поддерживает системы Microsoft Windows, Apple Macintosh, UNIX-подобные (для X11, Motif и GTK+), OpenVMS и OS/2. Встраиваемая (Embedded) версия находится в разработке.

Библиотека написана на C++, но может подключаться ко множеству других распространённых языков, таких, как Ruby (wxRuby, Anvil), Python (wxPython), Smalltalk (wxSqueak), Perl, Erlang, Haskell, Lua [2].

Важная особенность wxWidgets: в отличие от некоторых других библиотек (Swing, Qt и др.), она максимально использует «родные» графические элементы интерфейса операционной системы всюду, где это возможно. Это существенное преимущество для многих пользователей, поскольку они привыкают работать в конкретной среде, и изменения интерфейса программ часто вызывают затруднения в их работе.

Крайне «либеральная» лицензия wxWidgets допускает линковку с несвободными фрагментами кода, что позволяет использовать её и в закрытых коммерческих проектах.

Все вышеперечисленные особенности способствовали популярности библиотеки у самых разных разработчиков — от программистов-энтузиастов до

крупных корпораций и государственных учреждений (в числе которых Xerox, AMD, NASA и многие другие).

2 Глава. Описание приложения

2.1 Интерфейс

2.1.1 Общий вид

Интерфейс (рисунок 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;
- Output – окно вывода сообщений от Visual Bison;
- Debug – окно состояния отладчика;
- Errors – окно вывода ошибок грамматики;
- New File – окно тестового файла. Если файл существует на диске, заголовок содержит имя файла вместо New File;

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

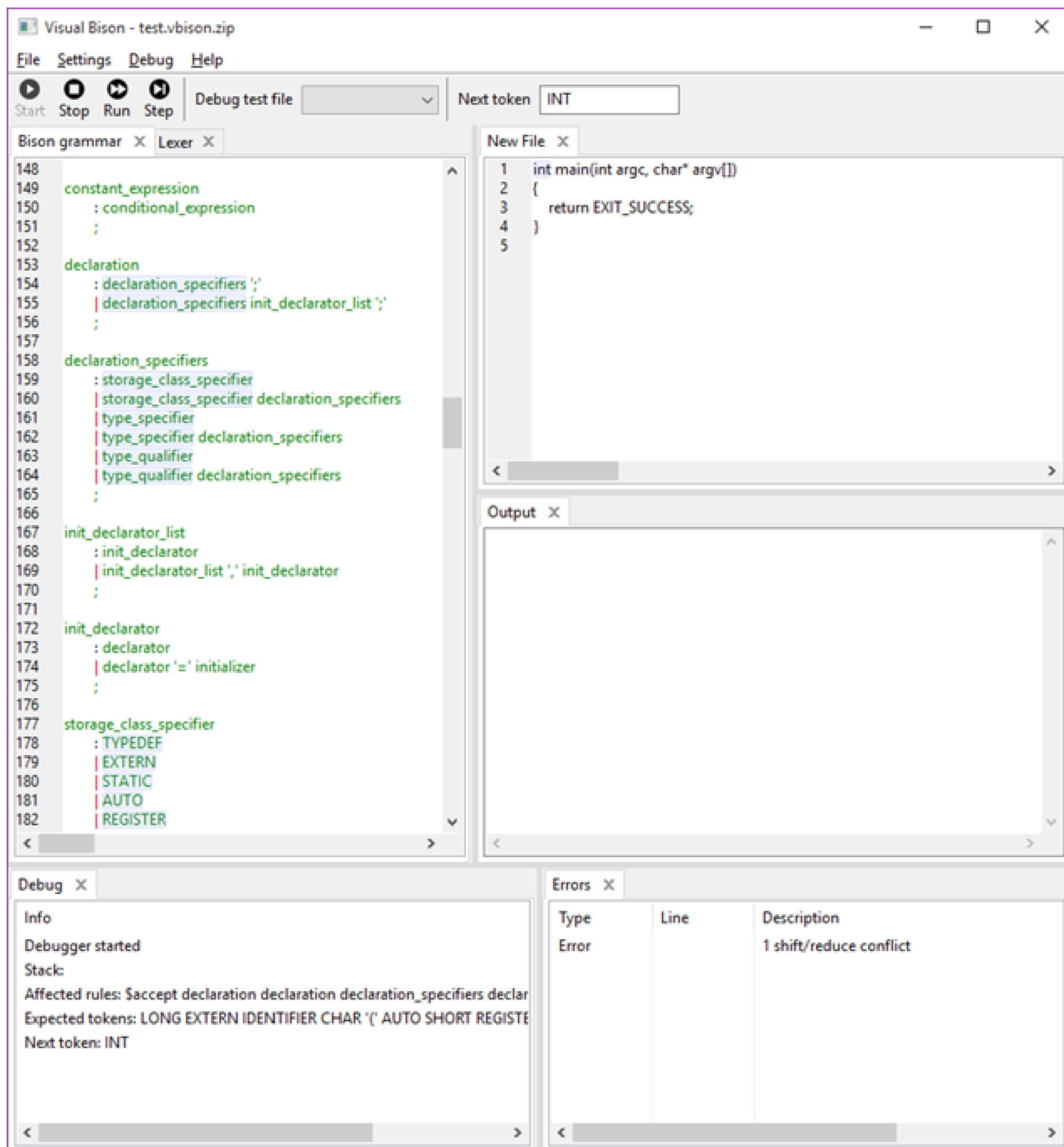


Рисунок 1 – вид главного окна

2.1.2 Главное меню

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;

2.1.3 Панель инструментов

Панель инструментов (рисунок 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.

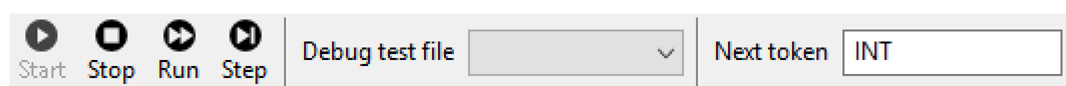


Рисунок 2 – панель инструментов

2.1.4 Окно грамматики

Окно грамматики (рисунок 3) содержит спецификацию для Bison, доступную для редактирования. Обладает подсветкой синтаксиса. Поддерживает типичные операции правки текста:

- выделение
- копирование
- вставка
- удаление

Bison grammar Lexer

```
%token IDENTIFIER CONSTANT STRING_LITERAL SIZEOF
%token PTR_OP INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
%token AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
%token SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
%token XOR_ASSIGN OR_ASSIGN TYPE_NAME

%token TYPEDEF EXTERN STATIC AUTO REGISTER
%token CHAR SHORT INT LONG SIGNED UNSIGNED FLOAT DOUBLE CONST VOLATILE VOID
%token STRUCT UNION ENUM ELLIPSIS

%token CASE DEFAULT IF ELSE SWITCH WHILE DO FOR GOTO CONTINUE BREAK RETURN

%start translation_unit
%%

primary_expression:
    IDENTIFIER
    | CONSTANT
    | STRING_LITERAL
```

Рисунок 3 – 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.

2.1.5 Окно лексического анализатора

Окно лексического анализатора (рисунок 4) 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.

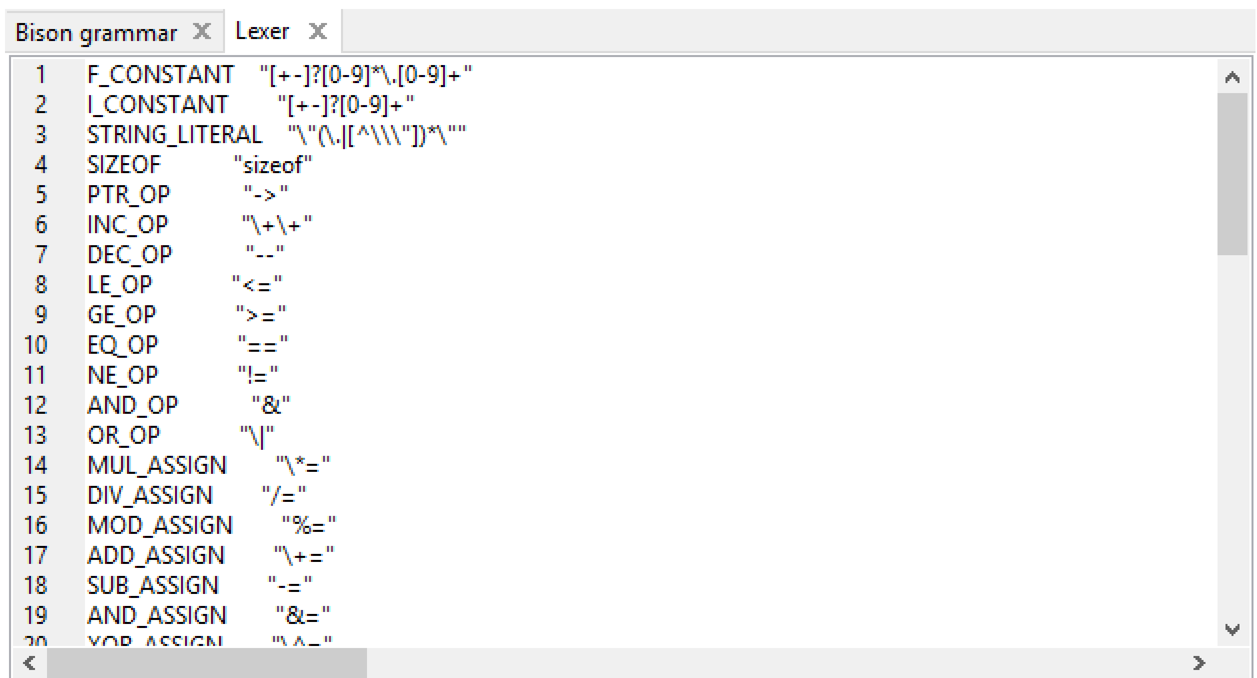


Рисунок 4 – окно лексического анализатора

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.

2.1.6 Окно отладки

Окно отладки (рисунок 5) отображает текущее состояние отладчика, а также историю сдвигов и сверток.

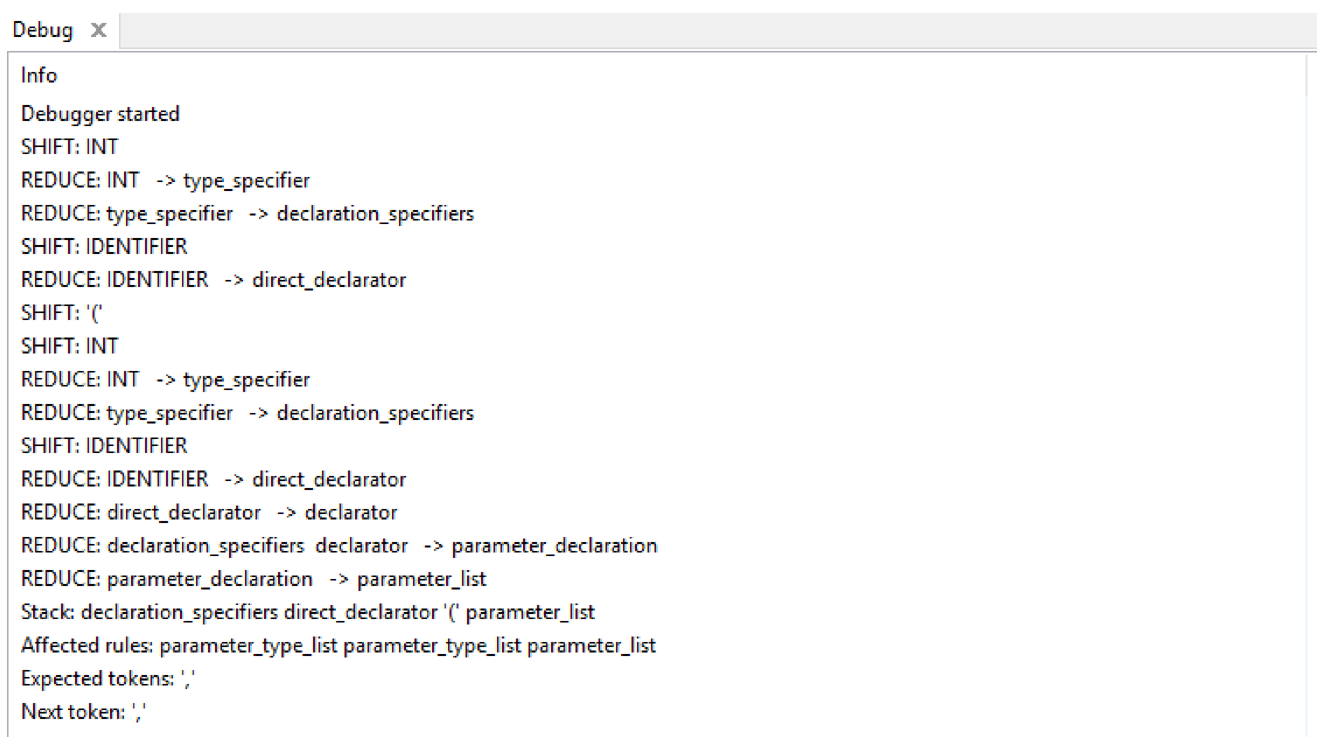


Рисунок 5 – окно отладки

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.

2.1.7 Окно вывода сообщений

В окне вывода сообщений (рисунок 6) 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;

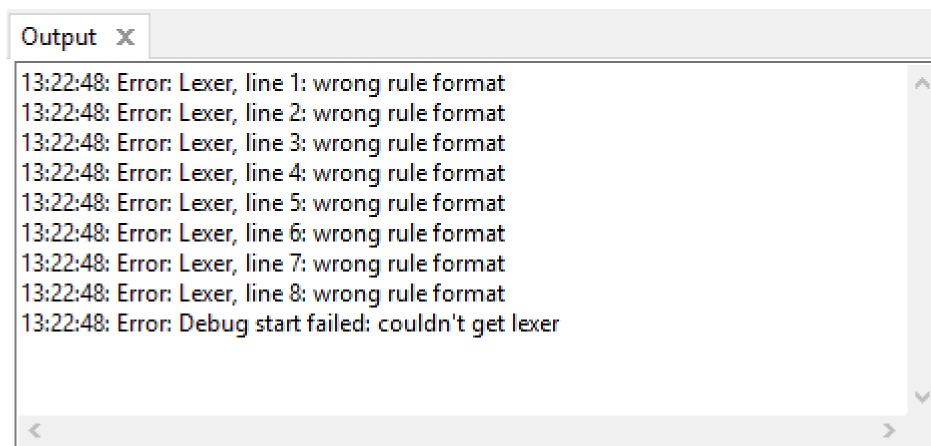


Рисунок 6 – окно вывода сообщений

2.1.8 Окно тестового файла

В окне тестового файла (рисунок 7) находится текст проверочного файла, который будет разбираться анализатором грамматики.

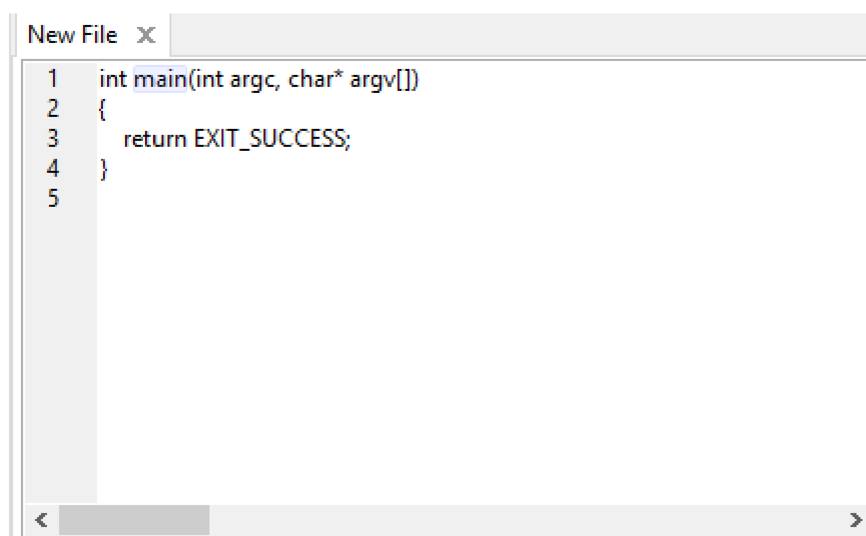


Рисунок 7 – окно тестового файла

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.

2.1.9 Окно вывода ошибок

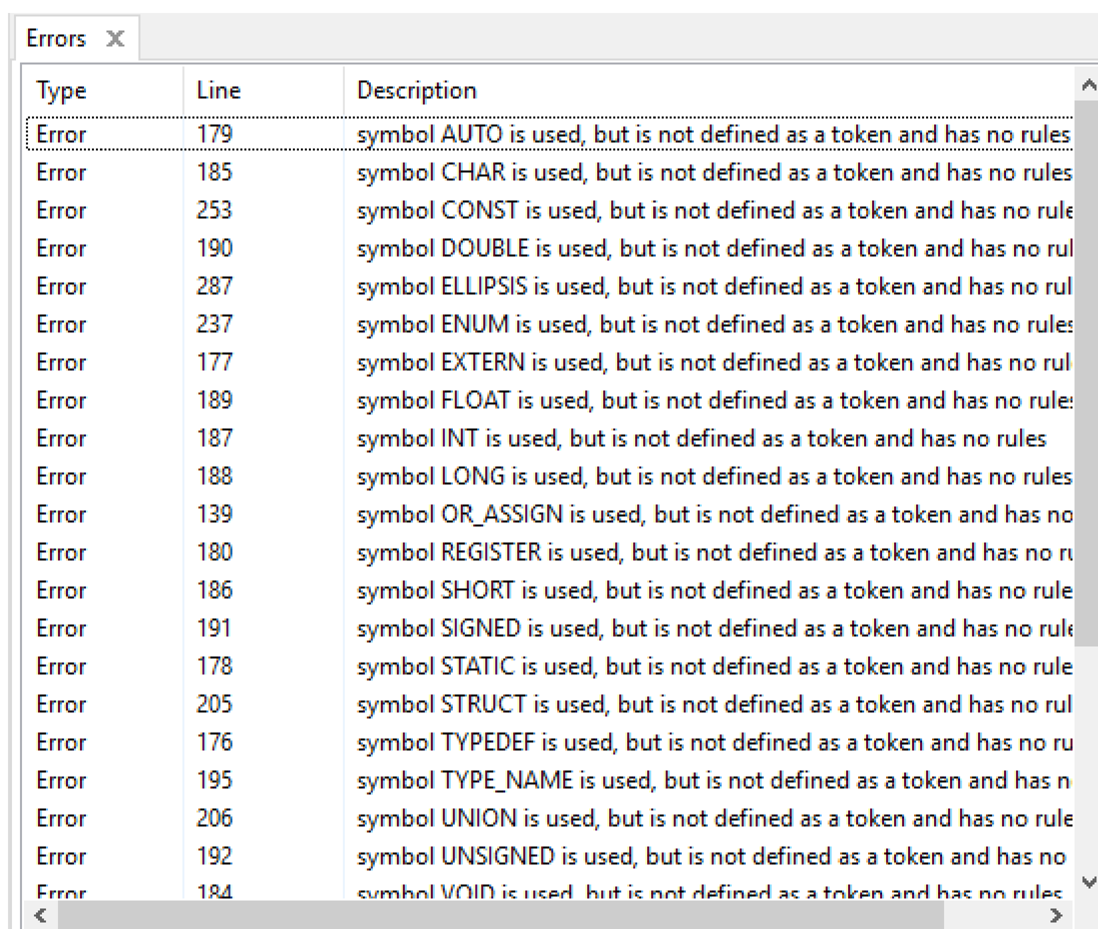
В окне вывода ошибок (рисунок 8) находится список ошибок в спецификации Bison. Список разделен на три поля:

➢ 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.


Errors

Type	Line	Description
Error	179	symbol AUTO is used, but is not defined as a token and has no rules
Error	185	symbol CHAR is used, but is not defined as a token and has no rules
Error	253	symbol CONST is used, but is not defined as a token and has no rule
Error	190	symbol DOUBLE is used, but is not defined as a token and has no rul
Error	287	symbol ELLIPSIS is used, but is not defined as a token and has no rul
Error	237	symbol ENUM is used, but is not defined as a token and has no rules
Error	177	symbol EXTERN is used, but is not defined as a token and has no rul
Error	189	symbol FLOAT is used, but is not defined as a token and has no rule
Error	187	symbol INT is used, but is not defined as a token and has no rules
Error	188	symbol LONG is used, but is not defined as a token and has no rules
Error	139	symbol OR_ASSIGN is used, but is not defined as a token and has no
Error	180	symbol REGISTER is used, but is not defined as a token and has no r
Error	186	symbol SHORT is used, but is not defined as a token and has no rule
Error	191	symbol SIGNED is used, but is not defined as a token and has no rul
Error	178	symbol STATIC is used, but is not defined as a token and has no rule
Error	205	symbol STRUCT is used, but is not defined as a token and has no rul
Error	176	symbol TYPEDEF is used, but is not defined as a token and has no ru
Error	195	symbol TYPE_NAME is used, but is not defined as a token and has n
Error	206	symbol UNION is used, but is not defined as a token and has no rule
Error	192	symbol UNSIGNED is used, but is not defined as a token and has no
Error	184	symbol VOID is used, but is not defined as a token and has no rules

Рисунок 8 – окно вывода ошибок

2.1.10 Горячие клавиши

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;

2.2 Перечень выводимых ошибок

Полный перечень выводимых ошибок указан в таблице 1.

Таблица 1 – 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

Окончание таблицы 1

Формат сообщения	Описание
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

2.3 Перечень ошибок в спецификации

Полный перечень ошибок спецификации от Bison указан в таблице 2.

Таблица 2 – 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 2

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

Продолжение таблицы 2

Формат сообщения	Описание
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
%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

Продолжение таблицы 2

Формат сообщения	Описание
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
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 граматике (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

Окончание таблицы 2

Формат сообщения	Описание
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}

3 Глава. Техническая реализация

3.1 Язык и библиотеки

Для реализации приложения был выбран язык C++, т.к. в проекте используется часть кодовой базы Bison, которая целиком написана на языке C. Также выбор обусловлен тем, C++ является кроссплатформенным высокоуровневым языком, с поддержкой парадигмы объектно-ориентированного программирования, и имеет множество бесплатных реализаций (сред для разработки, компиляторов) [3].

Графический интерфейс разработан с использованием бесплатной библиотеки wxWidgets [2].

Преимущества:

- использует стандартные элементы управления операционной системы, вместо отрисовки своих;
- является платформо-независимой библиотекой;
- есть удобный компонент для создания текстового редактора с подсветкой синтаксиса и сворачиванием структур [4];
- множество других вспомогательных классов, облегчающих разработку кроссплатформенного приложения;

3.2 Архитектура

В соответствии с архитектурой wxWidgets [2], 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, он создает все необходимые для проекта окна, и 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 просто «слушает» 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.

3.3 Редактор грамматики

Редактор выполнен с использование класс wxStyledTextCtrl [2], который является оберткой мощного текстового компонента Scintilla [4]. 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 интервалов 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. Все работу с точками организует класс CHasBreakpoints, от которого наследует редактор. CHasBreakpoints объявляет два чистых 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 «идет» 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.

3.4 Редактор тестового файла

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.

3.5 Проверка ошибок спецификации

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

Bison изначально проектировался как приложение с интерфейсом командной строки, которое выходит после обработки действия. Код написан без предположения что он будет повторно вызван. Переписывание кода вызвало бы множество правок, что затруднило бы перенос правок на новые версии, т.к. пришлось бы перепроверять их все. Поэтому, было решено использовать Bison как отдельный исполняемый файл, передавая ему имена файлов для вывода нужной информации. На момент написания приложения, Bison уже имеет вывод состояний конечного автомата в формате XML, необходимых для отладки. Но

вывод ошибок в подобном формате отсутствует, и поэтому было решено 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. Ниже приведен пример такого файла:


```xml
<?xml version="1.0"?>
<complains>
  <complaint sline="133" scolumn="11" eline="133" ecolumn="21">symbol ADD_ASSIGN is used, but is not defined as a token and has no rules</complaint>
  <complaint sline="110" scolumn="34" eline="110" ecolumn="40">symbol AND_OP is used, but is not defined as a token and has no rules</complaint>
  <complaint sline="185" scolumn="11" eline="185" ecolumn="18">symbol CHasdAR is used, but is not defined as a token and has no rules</complaint>
</complains>
```


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. Он 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, было решено вызывать Bison через интервал в 1,5 секунды.

3.6 Отладчик

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 запакованы все остальные таблицы. Содержимое таблиц defgoto, pgoto, defact, и pact является индексом начала данных в table. К нему надо 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 очень похожи. 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;

3.7 Лексический анализатор

Правила 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;

Строка с регулярным выражением проверяется также, как и строка лексемы.

Анализатор реализует для отладчика интерфейс ITokenSupply, через который передает ему лексемы. Через него же 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 бизона, чтобы выполнить заключительное правило $accept. Через метод Reset, можно вернуть анализатор в начальное состояние, к первой лексеме.

ЗАКЛЮЧЕНИЕ

В ходе выполнения дипломного проекта была подробно изучена работа генератора синтаксических анализаторов GNU Bison, в т.ч. его внутренняя организация путем изучения исходных кодов. Также рассмотрена библиотека wxWidgets, изучены ее компоненты, необходимые для реализации проекта, проведен анализ ее работы на различных платформах.

В соответствии с полученными знаниями и заданием на выпускную работу, реализовано приложение Visual Bison. Visual Bison является кроссплатформенным приложением, и позволяет пошагово выполнять разбор синтаксических конструкций разрабатываемой грамматики, ставить точки останова, для быстрого перехода к интересующему месту, имеет удобный графический интерфейс пользователя с подсветкой синтаксиса и выделением ошибок по мере написания грамматики. Процесс отладки проходит с выводом подробной информации о текущем шаге, об истории шагов, и печатью уточняющих сообщений от отладчика. Примеры работы отладчика представлены в приложении А.

Учитывая все возможности визуального редактора-генератора синтаксических анализаторов Visual Bison, можно отметить что он хорошо подходит для студентов, начинающих осваивать компиляторные курсы, т.к. наглядно показывает работу синтаксического анализатора.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. GNU Bison [Электронный ресурс] // Документация по GNU Bison. – Режим доступа: http://www.gnu.org/software/bison/ (дата обращения: 10.06.2016)
2. wxWidgets [Электронный ресурс] // Документация по wxWidgets 3.0. – Режим доступа: http://docs.wxwidgets.org/3.0/ (дата обращения: 10.06.2016)
3. Бьерн Страуструп.Язык программирования C++, 3-е изд. : учебное пособие / пер. с англ. – Москва: издательство «Бином», 1999. – 991 с.
4. Scintilla Documentation [Электронный ресурс] // Документация по компоненту Scintilla. – Режим доступа: http://www.scintilla.org/ScintillaDoc.html (дата обращения: 10.06.2016)

ПРИЛОЖЕНИЕ А

Примеры работы программы

А1. Отладка грамматики языка С

Код лексического анализатора:

```
F_CONSTANT    "[+-]?[0-9]*\.[0-9]+"
I_CONSTANT       "[+-]?[0-9]+"
STRING_LITERAL    "\"(\.|[^\\\"])*\""
SIZEOF           "sizeof"
PTR_OP           "->"
INC_OP           "\+\+"
DEC_OP           "--"
LE_OP           "<="
GE_OP           ">="
EQ_OP           "=="
NE_OP           "!="
AND_OP           "&"
OR_OP           "\|"
MUL_ASSIGN       "\*="
DIV_ASSIGN       "/="
MOD_ASSIGN       "%="
ADD_ASSIGN       "\+="
SUB_ASSIGN       "-="
AND_ASSIGN       "&="
XOR_ASSIGN       "\^="
OR_ASSIGN       "\|="
TYPEDEF         "typedef"
EXTERN          "extern"
STATIC          "static"
AUTO            "auto"
REGISTER        "register"
```

```
INLINE          "inline"
CONST           "const"
RESTRICT        "restrict"
VOLATILE        "volatile"
BOOL            "bool"
CHAR            "char"
SHORT           "short"
INT             "int"
LONG            "long"
SIGNED          "signed"
UNSIGNED        "unsigned"
FLOAT           "float"
DOUBLE          "double"
VOID            "void"
STRUCT          "struct"
UNION           "union"
CASE            "case"
DEFAULT         "default"
IF              "if"
ELSE            "else"
SWITCH          "switch"
WHILE           "while"
DO              "do"
FOR             "for"
GOTO            "goto"
CONTINUE        "continue"
BREAK           "break"
RETURN          "return"
ALIGNAS         "alignas"
ALIGNOF         "alignof"
IDENTIFIER      "[a-zA-Z_][a-zA-Z_0-9]*"
.               "\?|\||\^|>|<|%|/|\*|\+|-|~|!|&|\.|\[|\]|\(|\)|:|,|;|\{|\}"
```

На рисунке А.1 показано главное окно приложение с процессом отладки грамматики.

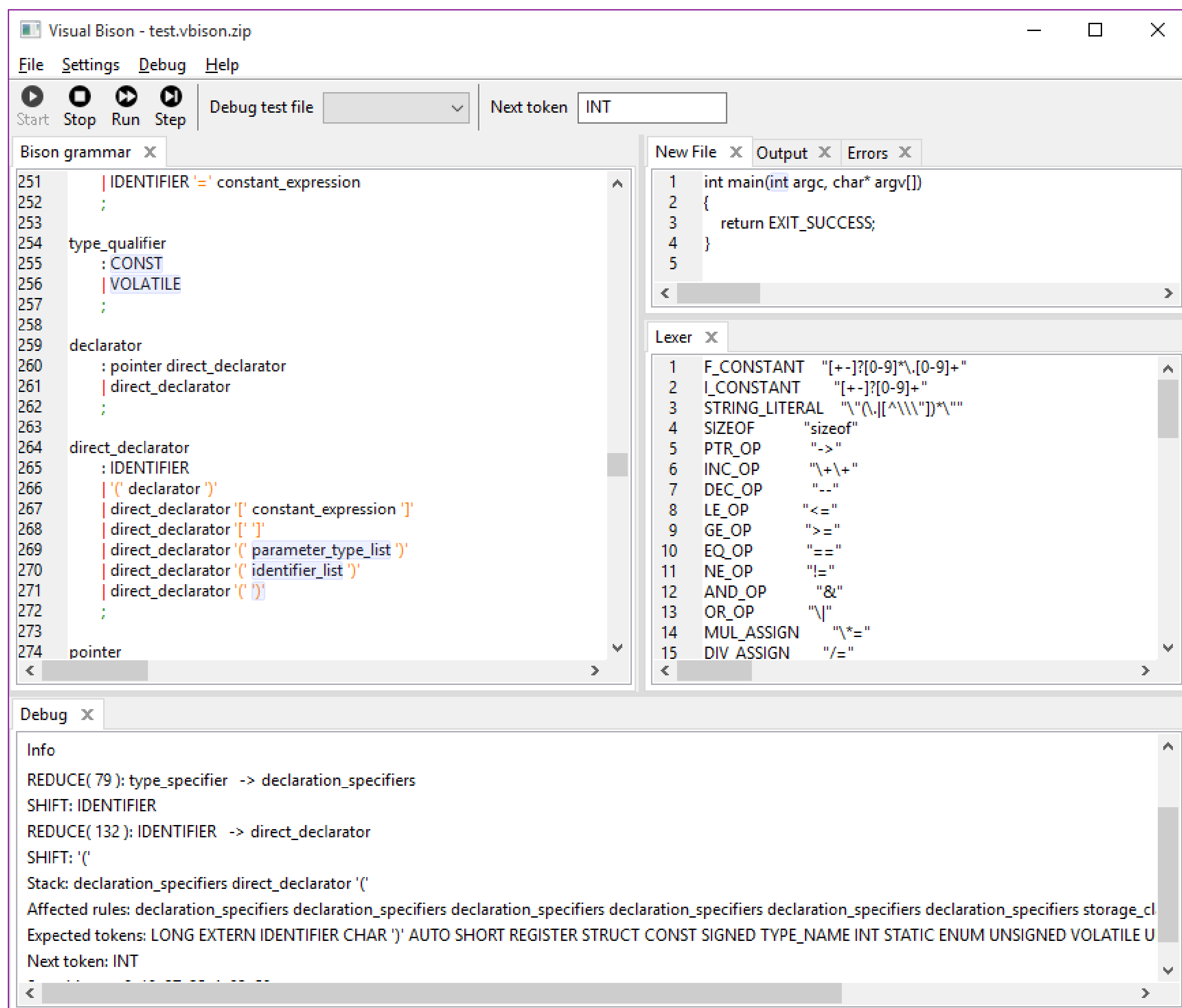


Рисунок А.1 – отладка грамматики языка С