МИНОБРНАУКИ РОССИИ ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ
БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО
ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет прикладной математики, информатики и механники
Кафедра программного обеспечения и администрирования
информационных систем

Разработка лексического и синтаксического анализаторов с целью подсветки синтаксиса и автодополнения исходного кода для предоставленного языка

Бакалаврская работа
Направление 02.03.03. Математическое обеспечение и администрирование информационных систем

Зав. кафедрой _____ уч. ст., уч. зв. Н. А. Огаркова
Обучающийся _____ А. С. Пахомов
Руководитель _____ уч. ст., уч. зв. Н. А. Огаркова

Воронеж — 2019

Содержание

Введение

После первоначального анализа источников напишу.

Объем и структура работы. Полный объём дипломной работы составляет 24 страницы, включая 1 рисунок и 1 таблицу. Список литературы содержит 62 наименования.

Глава 1. Аналитическая часть

1.1. Основные понятия в теории компиляции

1.1.1. Компилятор

Компилятор — это программа, которая принимает текст, написанный на одном языке — *исходном*, и транслирует (переводит) его в эквивалентный текст на другом языке — *целевом*. Одна из важных ролей компилятора состоит в сообщении об ошибках в исходной программе, обнаруженных в процессе трансляции.

Процесс компиляции можно разделить на две большие части. Первая часть — *анализ* (выполняется несколькими анализаторами) разбивает исходную программу на составные части и накладывает на них грамматическую структуру. Затем эта структура используется для промежуточного представления программы. В процессе анализа пользователю сообщается об ошибках, собирается информация об исходной программе и сохраняется в *таблицу символов*. Таблица символов — структуры данных, которые используются компилятором для хранения информации о конструкциях исходной программы. Информация накапливается инкрементально в фазе анализа компилятора и используется фазой синтеза для генерации целевого кода.

Вторая часть — *синтез* (выполняется генератором кода), строит требуемую целевую программу на основе промежуточного представления и информации из таблицы символов.

Если рассмотреть процесс компиляции более детально, то можно увидеть, что он представляет собой последовательность фаз каждая из которых преобразует одно представление программы в другое:

- *Лексический анализ* или *сканирование*. Лексический анализатор читает поток символов, составляющих исходную программу, группирует их в значащие последовательности (лексемы), формирует для каждой лексемы специальные структуры данных, называемые токенами и передает их синтаксическому анализатору.
- *Синтаксический анализ* или *разбор*. Синтаксический анализатор использует информацию из предыдущей фазы и на её основе строит синтаксическое дерево, в котором каждый внутренний

узел представляет операцию, а дочерние узлы — аргументы этой операции.

- *Семантический анализ*. Эта фаза компиляции использует синтаксическое дерево и информацию из таблицы символов для проверки исходной программы на семантическую согласованность с определением языка. Важной частью семантического анализа является проверка и приведение типов.
- *Генерация промежуточного кода.* На этой фазе компилятор генерирует низкоуровневое промежуточное представление кода исходной программы, которое можно рассматривать как программу для абстрактной вычислительной машины.
- *Машинно-независимая оптимизация кода.* Фаза машинно-независимой оптимизации кода пытается улучшить промежуточный код.
- *Генерация кода.* В качестве исходных данных генератор кода получает промежуточное представление исходной программы и отображает его в целевой язык, как правило в ассемблерный код.
- *Машинно-зависимая оптимизация кода.* Фаза машинно-зависимой оптимизации кода улучшает код целевой машины, учитывая особенности архитектуры процессора.

Фазы компиляции, которым соответствуют одноименные исполнители, представлены на рис. 1.1.

Процесс анализа компиляции разделен на лексический, синтаксический и семантический анализ по ряду причин:

1. Упрощение разработки. Отделение лексического анализа от синтаксического позволяет упростить как минимум одну из фаз анализа. Например, включить в синтаксический анализатор работу с комментариями и пробельными символами существенно сложнее, чем удалить их лексическим анализатором.
2. Увеличение эффективности компилятора. Отдельный лексический анализатор позволяет применять более специализированные методики, предназначенные исключительно для решения лексических задач.

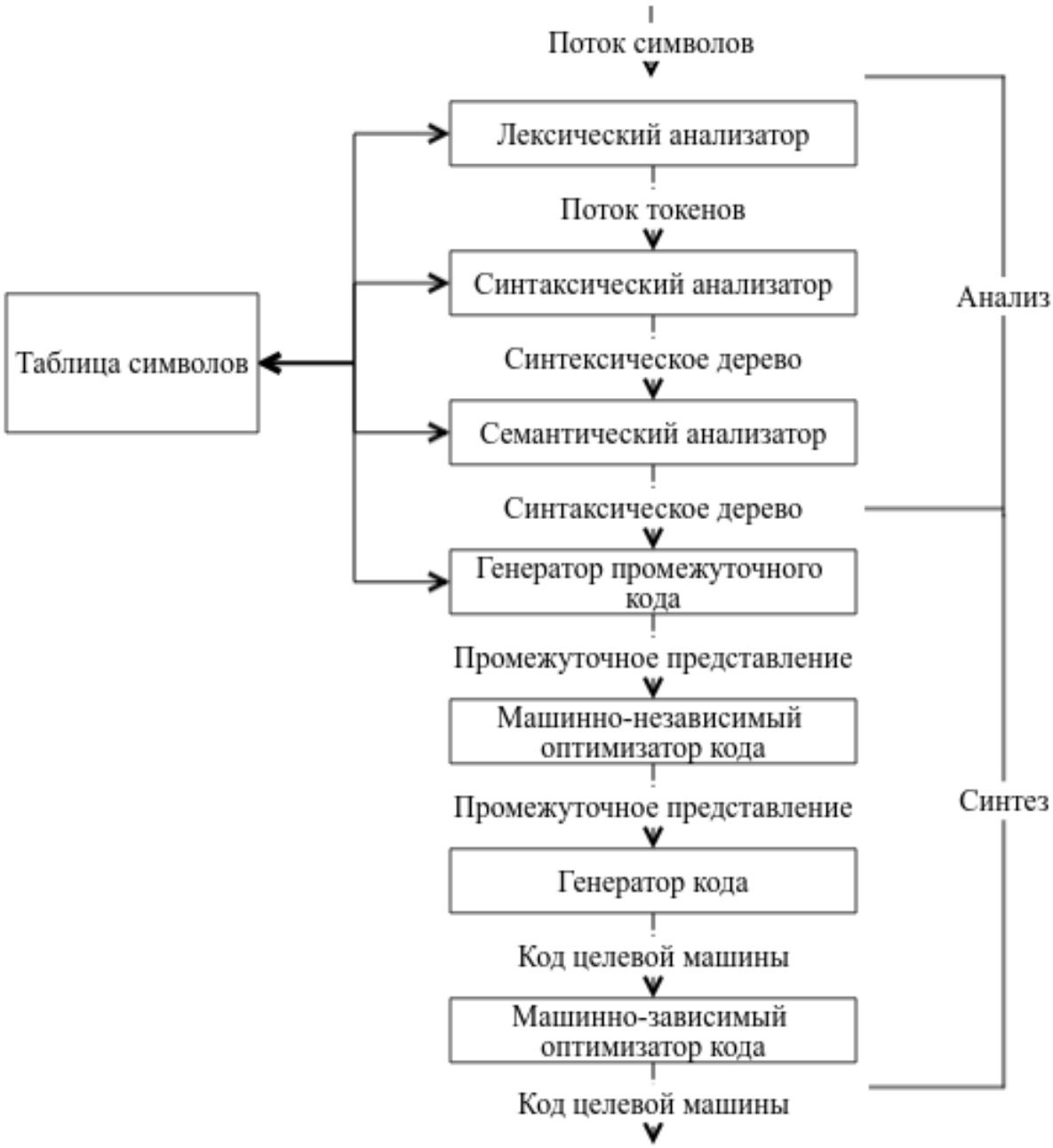

Рис. 1.1. Схема взаимодействия фаз компилятора

3. Увеличение переносимости компилятора. Особенности устройств ввода могут быть ограниченны лексическим анализатором.
4. Разделение зон ответственности. Ошибки типизации будут выявлены на стадии семантического анализа, что позволит сформулировать сообщение об ошибках типизации в более полной мере.

1.1.2. Лексический анализатор

При рассмотрении лексического анализа используются четыре связанных, но различных термина:

– *Токен* (от англ. *token* — знак, символ) представляет собой пару, состоящую из имени токена и необязательного атрибута. Имя токена — абстрактный символ, представляющий тип лексической единицы, например конкретное ключевое слово или последовательность символов, составляющих идентификатор. Имена токенов являются входными символами, обрабатываемыми синтаксическим анализатором. Примеры токенов приведены в табл. 1.1.

– *Атрибут токена* — строка или структура, объединяющая несколько блоков информации, которая содержит значение числа, в случае если токен соответствует шаблону числа, или описание лексемы (строчка кода, значение) которая представляет токен. Например, выражение на языке Fortran `E=M*2` будет представленно в виде последовательности

⟨**id**, Указатель на запись в таблице символов для E⟩
⟨**assign_op**⟩
⟨**id**, Указатель на запись в таблице символов для M⟩
⟨**mult_op**⟩
⟨**number**, Целое значение 2⟩

– *Шаблон* — это описание вида, который может принимать лексема токена. В случае ключевого слова шаблон представляет собой последовательность символов, образующая это ключевое слово. Для некоторых токенов шаблон представляет более сложную структуру.

– *Лексема* — последовательность символов исходной программы, которая соответствует шаблону токена и идентифицируется лексическим анализатором как экземпляр токена.

Основная задача лексического анализатора состоит в чтении входных символов исходной программы, их группировании в лексемы и вывод последовательностей токенов для всех лексем исходной программы. Поток токенов пересылается синтаксическому анализатору для разбора. Также лексический анализатор может удалять комментарии, пробельные символы, синхронизировать сообщения об ошибках или раскрывать макросы.

Таблица 1.1. Примеры токенов

Токен	**Неформальное описание**	**Примеры лексем**
if	Символы `i`, `f`	`if`
else	Символы `e`, `l`, `s` `e`	`else`
comparison	<, >, <=, >=, ==, !=	<=, !=
id	Буква, за которой следуют буквы и цифры	`pi`, `score`, `D2`
number	Любая числовая константа	`3.14159`, `0`
literal	Все, кроме ", заключенное в двойные кавычки	`"core dumped"`

1.1.3. Формальное определение контекстно-свободной грамматики

Существует множество грамматик (грамматики типа 3, контекстно-свободные грамматики, контекстно-зависимые грамматики и грамматики без ограничений), с помощью которых можно описать формальные языки. Для описания языков программирования используются контекстно-свободные грамматики (далее КС-грамматики).

КС-грамматика используется для определения синтаксиса языка программирования. КС-грамматика естественным образом описывает иерархическую структуру множества конструкций языка программирования. Состоит она из четырёх частей:

1. *Терминалы* — базовые символы, формирующие строки. Термин «имя токена» является синонимом слова «терминал». Пример терминала: `if`, `else`, `(`, `)`.
2. *Нетерминалы* — синтаксические переменные, которые обозначают множества строк. В примере $statement$ и $expression$ являются нетерминалами. Эти множества строк, обозначаемые нетерминалами, помогают определить язык, порождаемый КС-грамматикой. Нетерминалы также налагают на язык иерархическую структуру, облегчающую синтаксический анализ и трансляцию.
3. *Стартовый символ* — один из нетерминалов, который обозначает множество строк, являющиеся языком, определяемым КС-грамматикой. По соглашению, продукции стартового символа указываются первыми.

4. *Продукция* — способ, которым терминалы и нетерминалы объединяются в строки. Каждая продукция состоит из заголовка(левая часть), символа $\rightarrow$ и тела (правая часть), которое состоит из нуля или некоторого количества терминалов и нетерминалов.

$$
\begin{aligned}
expression &\rightarrow expression + term \\
expression &\rightarrow expression - term \\
expression &\rightarrow term \\
term &\rightarrow term * factor \\
term &\rightarrow term / factor \\
term &\rightarrow factor \\
factor &\rightarrow (expression) \\
factor &\rightarrow \mathbf{id}
\end{aligned}
$$

1.1.4. Синтаксический анализатор

В классической модели компилятора синтаксический анализатор получает строку токенов от лексического анализатора и проверяет, может ли эта строка имен токенов порождаться грамматикой исходного языка. Также от синтаксического анализатора ожидаются сообщения обо всех выявленных ошибках и умение продолжать работу с оставшейся частью программы. В случае корректной программы синтаксический анализатор строит дерево разбора и передает его следующей части компилятора для дальнейшей обработки.

Имеется три основных типа синтаксических анализаторов грамматик:

- *Универсальные методы разбора*, например, алгоритмы Кока-Янгера-Касами (Cocke-Younger-Kasami) и Эрли (Earley) могут работать с любой грамматикой. Однако эти обобщённые методы слишком неэффективны для использования в промышленных компиляторах.
- *Восходящие методы разбора* (bottom-up), построение дерева разбора происходит снизу (от листьев) вверх (к корню). Поток токенов сканируется слева направо.
- *Нисходящие методы разбора* (top-down) строят дерево разбора сверху (от корня) вниз (к листьям). Входной поток синтаксического

анализатора, как и в восходящих методах, сканируется посимвольно слева направо.

1.1.5. Семантический анализатор

Семантический анализатор использует синтаксическое дерево и информацию из таблицы символов для проверки исходной программы на семантическую согласованность с определением языка. Он также собирает информацию о типах и сохраняет ее в синтаксическом дереве или в таблице символов для последующего использования в процессе генерации промежуточного кода.

Важной частью семантического анализа является проверка типов, когда компилятор проверяет, имеет ли каждый оператор операнды соответствующего типа. Например, многие определения языков программирования требуют, чтобы индекс массива был целым числом. Компилятор должен сообщить об ошибке, если в качестве индекса массива используется число с плавающей точкой.

1.1.6. Генератор лексических анализаторов Lex

Lex (в более поздних реализациях *Flex*) — программный инструмент, который позволяет определить лексический анализатор, указывая регулярные выражения для описания шаблонов токенов. Входные обозначения для Lex обычно называют *языком Lex*, а сам инструмент — *компилятором Lex*. Компилятор Lex преобразует входные шаблоны в диаграмму переходов.

Нужна ли схема и структура программы на Lex?

1.1.7. Генератор синтаксических анализаторов Yacc

Yacc —компьютерная программа, служащая стандартным генератором синтаксических анализаторов в Unix-системах. Название является акронимом *«Yet Another Compiler Compiler»* («ещё один компилятор компиляторов»). Yacc генерирует синтаксический анализатор на основе аналитической грамматики, описанной в нотации Бэкуса-Наура или контекстно-свободной грамматики. На выходе уасс выдаётся код на языке программирования Си.

Поскольку синтаксический анализатор, генерируемый с помощью уасс, требует использования лексического анализатора, то часто он используется

совместно с генератором лексических анализаторов, в большинстве случаев это lex либо flex.

Схема? Пример программы на yacc?

Глава 2. Постановка задачи

Изучить существующие подходы к лексическому и синтаксическому анализу языков программирования с контекстно-свободной грамматикой и динамической компиляцией. Разработать лексический и синтаксический анализаторы с целью подсветки синтаксиса и автодополнения исходного кода для предоставленного языка (предметно ориентированного).

Глава 3. Анализ задачи

3.1. Анализ существующих подходов и инструментов

Тут будет обзор всех существующих инструментов и подходов. [1—62]

3.1.1. Семантика контекстно-свободных языков

В статье [31] приводятся базовые определения семантики контекстно-свободных языков, дается определение семантической и синтаксической однозначности граматик.

Статья состоит из пяти разделов, в разделе 1 описывается «значение» языков, определённых с помощью контекстно-свободных граматик. Математические основы описываются в 2 и 3 разделах. В 4 разделе описано формальное определение языка программирования с помощью контекстно-свободных граматик. Поледняя секция содержит сравнение основных подходов определения языков программирования с помощью контекство-свободных граматик.

Классика, (Кнут!), обязательно провести более глубокий анализ

3.1.2. Yacc, статья Стефана Джонсона

Статья «Еще один компилятор-компилятор» [35] описывает Yacc. Yacc — это инструмент, предоставляющий пользователю возможность описать входящие структуры, а затем связать их с некоторыми действиями. По сути, Yacc является мощным инструментом разбора входящего потока строк, который потом предпринимает опредеоленные пользователем действия в случае совпадения некоторых правил, которы так же определены пользователем.

Статья достаточно полно покрывает основные возможности Yacc, а так же приводит примеры их использования.

Yacc в любом случаю должен быть подробно разобран, статья хорошая, но пока оставлю ее на первом уровне, так как ожидаю найти более полный материал

3.1.3. Эффективный алгоритм разбора контекстно-свободных грамматик

Статья [46] является описанием алгоритма, разработанного в 1968г Джеем Орлей. В относительно небольшом объеме разобран алгоритм,

основные термины и сравнения производительности. В конце приводятся приемущества над другими алгоритмами тех лет.

Классика, нужно разобрать подробнее. Но необходимо выбрать материал между настоящим и **??**. Статья [46] мне кажется более доступной для понимания.

3.1.4. Практичный инструмент для создания портативных компиляторов

Статья [9] описывает «Амстердамский инструмент для компиляторов» который решает проблему написания $N \times M$ программ. То есть для интерпритации и компиляции N языков программирования на M разных компьютерных архитектурах, требуется $N \times M$ программ. «Амстердамский инструмент для компиляторов» решает эту проблему путем написания N программ, которые транслируют N языков в единое промежуточное представление и M программ, которые транслируют единое промежуточное представление в язык ассемблера для каждой из M архитектур. Таким образом нужно написать всего $N + M$ программ. Для поддержания нового языка на всех архитектурах требуется написать всего 1 программу, также как и для поддержания всех языков новой архитектурой.

Стоит отметить, что задача $N \times M$ программ совсем не простая, и для ее решения авторам пришлось пойти на некоторые уступки, а именно, они ограничились только алгебраическими языками и 8-ми битными архитектурами.

Очень полезная статья. Похоже, что это «дедушка» всех современных инструментов, нужно изучить.

3.1.5. Эффективный контекстно-свободный алгоритм для разбора естественных языков

Статья [6] описывает алгоритм разбора естественных языков. Автор описывает существующие на то время алгоритмы разбора языков и объясняет почему они не годятся для разбора естественных языков.Он разделяет все алгоритмы на 2 группы:

1. Алгоритмы разбора языков программирования.
2. Алгоритмы разбора общих контексно свободных языков.

Обе группы, по словам автора, не подходят для разбора естественных языков. Первые — ограниченны слишком маленьким набором грамматик,

вторые — слишком громоздкие для естественных языков, которые ближе к языкам программирования, а не к контекстно-свободным грамматикам.

Автор предлагает свой алгоритм, который находится между алгоримами разбора языков программирования и алгоритмами разбора контекстно-свободных языков. И подробно его описывает.

Будет полезно для дипломной работы, стоит изучить статью.

3.1.6. Обзор методом интерпритации конекстно-свободных языков

В статье [30] описываются основные подходы для интерпритации контекстно-свободных языков. В начале авторы описывают негативные стороны контекстно-свободных языков, а потом проводят верхнеуровневый обзор пяти методов интерпритации контекстно-свободных языков:

1. Методы, которые берут текст на вход.
2. Методы, которые берут структурную информацию на вход.
3. Методы, не основанные на контекстно-свободных языках.
4. Методы, которые берут описание сабклассов.
5. Стохастические методы.

В начале статьи описывается проблема изучения и интерпретации контекстно-свободных граматик. Затрагиваются темы интерпритации речи, изучения языка на лету, проблемы интерпритации граматик за полиномиальное время, и т.д.

Нужно изучить подробнее, так как описание самих методов довольно понятное и будет полезно для диплома.

3.1.7. Маленькие языки: Маленькая поддержка?

В статье [50] приводится определение предметно-ориентированного языка и то, как он может помочь или навредить приложениям и бизнесу.

Согластно статье [50], предметно-ориентированный язык — небольшой, чаще декларативный, язык, выражающий различные характеристики набора программ в конекретной проблеме конкретной предметной области.

Предметно-ориентированное описание — программа (спецификация, описание, запрос, процесс, задача, ...), написанная на предметно-ориентированном языке.

Предметно-ориентированный процессор — програмное обеспечение для компиляции, интрепритации или анализа предметно-ориентированного языка.

Авторы приводят плюсы предметно-ориентрованных языков, например: легкость в написании предметных программ, простота эксплуатации и поддержиня таких програм, само-документирование. А также описываются и проблемы, такие, как сложность в расширении подобных язков, при необходимости расширения предметной области.

В заключении, авторы говорят, что предметно-ориентированные языки это не панацея для решения всех прикладных задач, но хорошо спроектированные языки, с хорошим набором инструментов для работы с ними, могут значительно облегчить и ускорить процесс написания и поддержания программ.

Позитивный сценарий использования подобных языков: Вся реализация сложных процессов возложенна на компилятор, в то время как бизнес логика легко описывается с помощью языка.

Негативный сценарий: Предметная область может быть не до конца изучена, для того что бы проектировать под нее специальный язык. В такой ситуации следует обратить внимание на стандартные средства написания программ.

Альтернатива предметно-ориентированным языкам: Использование объекно-ориентированных языков вместе с библиотеками типов и функций, которые описывают предметную область.

Возможные направления для улучшения предметно-ориентированных языков: Виуализация, приближение языков к естественным, симулирование среды для выполнения программ.

Полезная статья, из нее можно взять основные определения. Оставлю, если не найду что-то лучше.

3.1.8. Delite: Архитектура компилятора для встроенных производительных предметно-ориентированных языков

Статья [11] описывает инструмент под названием Delite, разработанный Oracle Labs совместно со Стэндфорским Университетом, а так же предметно-ориентированные языки, написанные при помощи этого инструмента.

Delite позволяет упростить трудоёмкий процесс написания производительных предметно-ориентированных языков для работы с параллельным программированием. Простота написания подобных языков достигается за счет предоставления параллельных шаблонов проектирования, оптимизаций и кодо-генераций, которые направлены на использование в предметно-ориентированных языках программирования.

Инструмент интегрирован с языком Scala, а так же предоставляет возможность написания предметно-ориентированных языков, которые будут выполняться на C++, CUDA, OpenCL и MPI.

Зачем нужны подобные инструменты? Основная проблема заключается в том, что высокоуровневые языки программирования предоставляют понятные абстракции и читаемый код программ, но лишают пользователей использовать низкоуровневые оптимизации для достижения высокой производительности программ. Низкоуровневые языки, в свою очередь, предоставляют большие возможности по части оптимизации, но код программ становится очень сложным.

Авторы статьи предлагают решить эту проблему с помощью предметно-ориентированных языков, которы позволят оптимизировать программы на низком уровни и в то же время сохранят читаемость кода.

Статья предлагает использовать Delite для написания подобных предметно-ориентированных языков, а так же описывает несколько уже существующих языков (OptiML для машинного обучения, OptiQL для запросов к данным, OptiGraph для анализа графов и OptiMesh для распределённых вычислений), разработанных с помощью Delite.

Так же авторы статьи приводят анализ производительности, результаты которого сопоставимы с эквивалентными программ, написанными на C++.

3.1.9. Как компиляторы должны сообщать об ошибках?

Статья [57] описывает проблему непонятного объяснения ошибок компиляции и предлагает применить модель Тюльмина для повышения эффективности объяснения ошибок компиляции. Для полного понимания способа объяснения ошибок авторы статьи провели сравнительный анализ среди 68 разработчиков и эмпирическое изучение сообщений об ошибках компиляции среди пользователей портала Stack Overflow.

Авторы выяснили, что короткое сообщение об ошибке, которое содержит решение проблемы, оказалось более предпочтительнее, чем развернутое сообщение об ошибке без предложений как её можно решить.

Авторы утверждают, что если сообщение об ошибке оформлено как объяснение, к нему может быть применена теория объяснения Тюльмина, для того, что бы понять почему одни сообщения об ошибках эффективнее других.

В результате исследований, в статье предлагаются 3 принципа, которым нужно следовать при объяснении ошибок компиляции:

1. Предоставляйте разработчикам возможность самостоятельного выбора глубины объяснения проблемы
2. Разделяйте понятия «объяснения» и «решения» проблемы
3. Сопоставляйте сообщение об ошибке с содержимым выражения, где произошла ошибка

Заключение

Основные результаты работы заключаются в следующем.

1. На основе анализа …
2. Численные исследования показали, что …
3. Математическое моделирование показало …
4. Для выполнения поставленных задач был создан …

И какая-нибудь заключающая фраза.

Список литературы

1. *Wulf, W. A.* Compilers and computer architecture / W. A. Wulf // Computer. — 1981. — № 7. — С. 41—47.
2. *Wimmer, C.* One VM to rule them all / C. Wimmer, C. Seaton // In Proceedings of the JVM Language Summit. — 2013.
3. *Wimmer, C.* Truffle: a self-optimizing runtime system / C. Wimmer, T. Würthinger. — 2012.
4. *Vergu, V.* Specializing a meta-interpreter: JIT compilation of dynsem specifications on the graal VM / V. Vergu, E. Visser. — 2018.
5. *Van Deursen, A.* Domain-specific languages: An annotated bibliography / A. Van Deursen, P. Klint, J. Visser // ACM Sigplan Notices. — 2000. — Т. 35, № 6. — С. 26—36.
6. *Tomita, M.* An Efficient Context-Free Parsing Algorithm for Natural Languages. / M. Tomita. — 1985.
7. *Thatcher, J. W.* More on advice on structuring compilers and proving them correct / J. W. Thatcher, E. G. Wagner, J. B. Wright // Theoretical Computer Science. — 1981. — Т. 15, № 3. — С. 223—249.
8. *Temkin, J. M.* Extraction of protein interaction information from unstructured text using a context-free grammar / J. M. Temkin, M. R. Gilder // Bioinformatics. — 2003. — Т. 19, № 16. — С. 2046—2053.
9. A practical tool kit for making portable compilers / A. S. Tanenbaum [и др.] // Communications of the ACM. — 1983. — Т. 26, № 9. — С. 654—660.
10. *Takahashi, M.* Generalizations of regular sets and their application to a study of context-free languages / M. Takahashi // Information and Control. — 1975. — Т. 27, № 1. — С. 1—36.
11. Delite: A compiler architecture for performance-oriented embedded domain-specific languages / A. K. Sujeeth [и др.] // ACM Transactions on Embedded Computing Systems (TECS). — 2014. — Т. 13, 4s. — С. 134.
12. An experimental study of the influence of dynamic compiler optimizations on Scala performance / L. Stadler [и др.]. — 2013.

13. *Spinellis*, *D.* Notable design patterns for domain-specific languages / D. Spinellis // Journal of systems and software. — 2001. — Т. 56, № 1. — С. 91—99.

14. *Sikkel*, *K.* Parsing of context-free languages / K. Sikkel, A. Nijholt. — Springer, 1997. — С. 61—100.

15. Frame model of a compiler of cluster parallelism for embedded computing systems / V. Ruchkin [и др.]. — 2017.

16. Bringing low-level languages to the JVM: efficient execution of LLVM IR on Truffle / M. Rigger [и др.]. — 2016.

17. *Rigger*, *M.* Sulong-execution of llvm-based languages on the jvm / M. Rigger, M. Grimmer, H. Mössenböck // ICOOOLPS'16. — 2016.

18. A Case for Context-Free Grammar / C. Price [и др.] // Journal of Computer Science and Software Engineering. — 2018. — Т. 9, № 4.

19. *Pool*, *T.* Trufflereloader: a low-overhead language-neutral reloader / T. Pool, A. R. Gregersen, V. Vojdani. — 2016.

20. *Paulson*, *L.* A semantics-directed compiler generator / L. Paulson. — 1982.

21. binpac: A yacc for writing application protocol parsers / R. Pang [и др.]. — 2006.

22. *Niephaus*, *F.* GraalSqueak: A Fast Smalltalk Bytecode Interpreter Written in an AST Interpreter Framework / F. Niephaus, T. Felgentreff, R. Hirschfeld. — 2018.

23. *Moore*, *R. C.* Removing left recursion from context-free grammars / R. C. Moore. — 2000.

24. *Mernik*, *M.* When and how to develop domain-specific languages / M. Mernik, J. Heering, A. M. Sloane // ACM computing surveys (CSUR). — 2005. — Т. 37, № 4. — С. 316—344.

25. An evaluation of vectorizing compilers / S. Maleki [и др.]. — 2011.

26. *Lovato*, *M. E.* Parser visualizations for developing grammars with yacc / M. E. Lovato, M. F. Kleyn. — 1995.

27. *Louden*, *K. C.* Compiler construction / K. C. Louden // Cengage Learning. — 1997.

28. Lex & yacc / J. R. Levine [и др.]. — 1992.

29. *Leijen*, *D.* Domain specific embedded compilers / D. Leijen, E. Meijer. — 1999.

30. *Lee*, *L.* Learning of context-free languages: A survey of the literature / L. Lee. — 1996.

31. *Knuth*, *D. E.* Semantics of context-free languages / D. E. Knuth // Mathematical systems theory. — 1968. — Т. 2, № 2. — С. 127—145.

32. *Kitaura*, *K.* Random testing of compilers' performance based on mixed static and dynamic code comparison / K. Kitaura, N. Ishiura. — 2018.

33. *Jørring*, *U.* Compilers and staging transformations / U. Jørring, W. L. Scherlis. — 1986.

34. *Jones*, *S. L. P.* YACC in SASL—an exercise in functional programming / S. L. P. Jones // Software: Practice and Experience. — 1985. — Т. 15, № 8. — С. 807—820.

35. Yacc: Yet another compiler-compiler / S. C. Johnson [и др.]. — 1975.

36. Context-free grammar induction using genetic programming / F. Javed [и др.]. — 2004.

37. Librando: transparent code randomization for just-in-time compilers / A. Homescu [и др.]. — 2013.

38. *Hearnden*, *D.* Anti-Yacc: MOF-to-text / D. Hearnden, K. Raymond, J. Steel. — 2002.

39. *Grosch*, *J.* A tool box for compiler construction / J. Grosch, H. Emmelmann. — 1990. — С. 106—116.

40. TruffleC: dynamic execution of C on a Java virtual machine / M. Grimmer [и др.]. — 2014.

41. *Griffiths*, *T. V.* On the relative efficiencies of context-free grammar / T. V. Griffiths, S. R. Petrick // Communications of the ACM. — 1965. — Т. 8, № 5. — С. 289—300.

42. *Glanville*, *R. S.* A new method for compiler code generation / R. S. Glanville, S. L. Graham. — 1978.

43. *Gaikwad*, *S.* Performance analysis for languages hosted on the truffle framework / S. Gaikwad, A. Nisbet, M. Luján. — 2018.

44. *Ferro*, *M. V.* Efficient incremental parsing for context-free languages / M. V. Ferro, B. A. Dion. — 1994.

45. *Ekman*, *T.* The jastadd extensible java compiler / T. Ekman, G. Hedin // ACM Sigplan Notices. — 2007. — Т. 42, № 10. — С. 1—18.

46. *Earley*, *J.* An efficient context-free parsing algorithm / J. Earley // Communications of the ACM. — 1983. — Т. 26, № 1. — С. 57—61.

47. *Earley*, *J.* An efficient context-free parsing algorithm / J. Earley // Communications of the ACM. — 1970. — Т. 13, № 2. — С. 94—102.

48. *Duboscq*, *G.* Speculation without regret: reducing deoptimization meta-data in the Graal compiler / G. Duboscq, T. Würthinger, H. Mössenböck. — 2014.

49. An intermediate representation for speculative optimizations in a dynamic compiler / G. Duboscq [и др.]. — 2013.

50. *Deursen*, *A. V.* Little languages: Little maintenance? / A. V. Deursen, P. Klint // Journal of Software Maintenance: Research and Practice. — 1998. — Т. 10, № 2. — С. 75—92.

51. *Cooper*, *K. D.* Adaptive optimizing compilers for the 21st century / K. D. Cooper, D. Subramanian, L. Torczon // The Journal of Supercomputing. — 2002. — Т. 23, № 1. — С. 7—22.

52. A generative programming approach to developing DSL compilers / C. Consel [и др.]. — 2005.

53. A DSL compiler for accelerating image processing pipelines on FPGAs / N. Chugh [и др.]. — 2016.

54. Testing and Verification of Compilers (Dagstuhl Seminar 17502) / J. Chen [и др.]. — 2018.

55. *Brantner*, *M.* Modern stored procedures using GraalVM: invited talk / M. Brantner. — 2017.

56. *Bhamidipaty*, *A.* Very fast YACC-compatible parsers (for very little effort) / A. Bhamidipaty, T. A. Proebsting // Software: Practice and Experience. — 1998. — Т. 28, № 2. — С. 181—190.

57. How should compilers explain problems to developers? / T. Barik [и др.]. — 2018.

58. *Aho*, *A. V.* Compilers: principles, techniques and tools (for Anna University), 2/e / A. V. Aho. — Pearson Education India, 2003.

59. *Aho*, *A. V.* Compilers: principles, techniques and tools (for Anna University), 2/e / A. V. Aho. — 2003.

60. *Aho*, *A. V.* A minimum distance error-correcting parser for context-free languages / A. V. Aho, T. G. Peterson // SIAM Journal on Computing. — 1972. — Т. 1, № 4. — С. 305—312.

61. *Aho*, *A. V.* Translations on a context free grammar / A. V. Aho, J. D. Ullman // Information and Control. — 1971. — Т. 19, № 5. — С. 439—475.

62. OCEANS: Optimizing compilers for embedded applications / B. Aarts [и др.]. — 1997.