МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

МАТЕРИАЛЫ

Международной научной конференции

«МАТЕМАТИЧЕСКОЕ И ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ИНФОРМАЦИОННЫХ, ТЕХНИЧЕСКИХ И ЭКОНОМИЧЕСКИХ СИСТЕМ»

Томск, 28–30 мая 2020 г.

Под общей редакцией
кандидата технических наук И.С. Шмырина

Томск
Издательство Томского государственного университета
2020

ББК 22.17–22.19
УДК 519.2, 519.7, 519.8
Т78

Т78 Труды Томского государственного университета. – Т. 305. Серия физико-математическая: Математическое и программное обеспечение информационных, технических и экономических систем : материалы Международной научной конференции. Томск, 28–30 мая 2020 г. / под общ. ред. И.С. Шмырина. – Томск : Издательство Томского государственного университета, 2020. – 322 с.

ISBN 978-5-94621-970-9

Сборник содержит материалы Международной научной конференции «Математическое и программное обеспечение информационных, технических и экономических систем», проводившейся 28–30 мая 2020 г. на базе Института прикладной математики и компьютерных наук Томского государственного университета. Материалы сгруппированы в соответствии с работавшими на конференции секциями.

Для научных работников, преподавателей, аспирантов, магистрантов и студентов.

УДК 539.3.004
ББК 22,25.22.251.22.62

ISBN 978-5-94621-970-9

3. *Ричардсон Я.* Видеокодирование. H.264 и MPEG-4 - стандарты нового поколения. - М.: ТЕХНОСФЕРА, 2005. - 368с.
4. *Сэломон Д.* Сжатие данных, изображений и звука: Пер. с англ. - М.: Техносфера, 2004. - 368с.

ПАРСЕР ДЛЯ ЯЗЫКА ПРОГРАММИРОВАНИЯ RHINESTONE

Чалых Е.П., Самохина С.И.
Томский государственный университет
egor.chalyh.98@gmail.com, sv.sam.tsk@gmail.com

Введение

На данный момент в мире существует большое число языков программирования, каждый из них создан с определённой целью. Изначальной целью проекта было создание мощного и быстрого языка программирования с поддержкой математических вычислений, таких как дифференцирование, интегрирование, работа с матрицами и другие. Но первая проблема нового языка возникла при выборе его типа: интерпретируемый или компилируемый. Очевидный минус первого типа заключается в том, что процесс выполнения кода будет довольно долгим, особенно на очень больших файлах исходного кода. Но главное достоинство – интерпретатор языка можно сделать кроссплатформенным. Компилируемый язык программирования будет работать гораздо быстрей, т.к. из исходного кода мы получим сразу исполняемый файл. А минус такого языка, естественно, платформозависимость.

1. Постановка задачи

Первый этап создания языка программирования – это создание работоспособной, желательно кроссплатформенной, виртуальной машины [1]. Второй – создание компилятора [2], который будет транслировать файлы с текстом исходного кода в код понятный для виртуальной машины.

Скорость работы компилятора можно повысить, если использовать язык C++ [3]. Кроссплатформенности можно добиться, используя систему сборки CMake [4], она сама выберет компилятор в зависимости от операционной системы, а также подготовит проект к сборке. Инструментом создания языка является мощная интегрированная среда разработки CLion [5] от компании JetBrains [6].

2. Реализация языка программирования

Язык реализован на языке C++, с использованием стандарта STDLIB 20. Исходный код реализации распространяется в рамках проекта OpenRSP (Open RhineStone Project) [7]. Это проект с открытым исходным кодом под лицензией Apache License 2.0 [8]. В нем есть 3 подпроекта: libs, vm, compiler. Первый – набор дополнительных библиотек, используемых в vm и compiler. Второй – реализация виртуальной машины RhineStone VM. Третий – реализация компилятора.

RhineStone VM – портированная на язык C++ виртуальная машина EmeraVM [9].

Компилятор разделен на 3 части. Первая – преобразование текста в структуру данных, которую понимает компьютер, в данном случае абстрактное синтаксическое дерево (далее АСД). Вторая – анализ АСД и добавление дополнительной информации (атрибутов). Третья – сборка дерева в байт-код, который будет выполнять виртуальная машина. Рассмотрим первый компонент компилятора – парсер.

2.1. Реализация парсера для языка программирования

Парсер – инструмент, который позволяет преобразовать текстовые данные в АСД. Парсинг – процесс такого преобразования. Он разделен на лексический и синтаксический анализы.

2.2. Лексический анализ

Лексический анализ – процесс выделения из текста определенных лексем (или токенов). Эти лексемы содержат информацию о типе лексемы (ключевое слово, оператор и т.п.) и значение лексемы (например, если тип – число, то значение – последовательность цифр). Также в лексеме можно хранить данные о местоположении в исходном коде. Лексический анализатор можно сгенерировать с помощью специальных утилит, например, ANTLR [10], GNU Bison [11] и других. Исходными данными для таких утилит служит описание грамматики в форме Бэкуса-Наура (БНФ) [12], иногда используется расширенная БНФ. Результатом выполнения утилиты является лексический анализатор на основе детерминированного конечного автомата [13]. Очевидным плюсом такого подхода является скорость работы автомата. А минус – очень сложно отлаживать программу, а также для работы такого анализатора требуется дополнительная библиотека из утилиты. Второй способ создать лексический анализатор – это самостоятельно описать грамматику языка программным способом, т.е. определить правила на языке C++ для каждого входного символа так, чтобы лексемы выделялись однозначно.

Принцип такого подхода следующий. Читаем символ из входного файла. Определяем его тип, например, пробел – разделитель, его мы пропускаем, цифра – является частью числа, значит применяем правило чтения числа, прочли букву – часть идентификатора или ключевого слова, и т.д.

Рассмотрим правило чтения идентификатора. У нас есть его начало, теперь продолжаем читать символы пока они удовлетворяют следующим требованиям: буква, цифра, знак нижнего подчеркивания, знак доллара. После прочтения у нас есть последовательность символов для идентификатора. Если эта последовательность принадлежит таблице зарезервированных слов, то лексема – ключевое слово, иначе идентификатор.

Для операторов принцип такой же, но для однозначности определения выбирается последовательность максимальной длины. Пример: «a >>> b». Символы a и b определятся как идентификаторы, пробелы пропустятся. Язык в своем наборе имеет операторы «>>» (сдвиг вправо) и «>» (больше чем), оператора «>>>» нет. Поэтому список лексем для этого выражения будет следующим: идентификатор «a», оператор «>>», оператор «>», идентификатор «b».

2.3. Абстрактное синтаксическое дерево

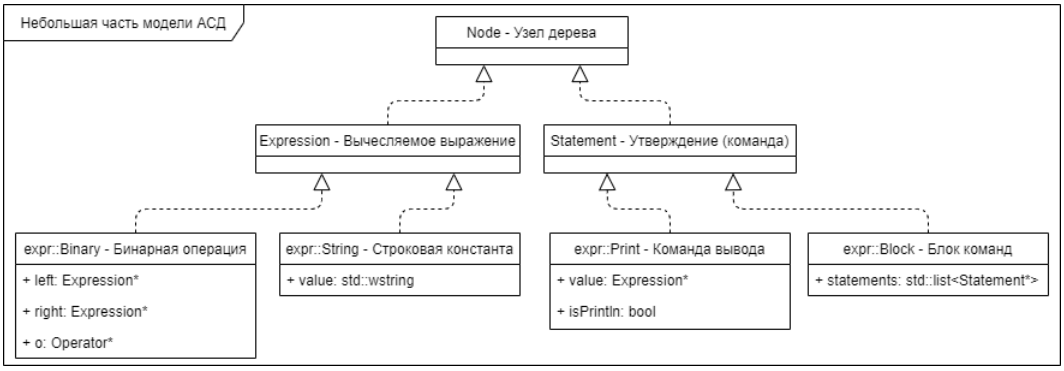
Представить исходный код программы с помощью дерева – логичный и очевидный шаг. Например, определение цикла While: условие и тело цикла. Узел будет состоять из узла для условия, и узла для тела. Подобно While можно представить условный оператор, определение функции и т.д.

Анализ дерева происходит путем его прямого обхода с помощью специального интерфейса «Посетитель» (IVisitor). С помощью этого интерфейса можно также перестраивать дерево (например, для оптимизации), преобразовывать в другие структуры данных (например, XML дерево, или string).

Это дерево является синтаксическим, т.к. было построено согласно определенным синтаксическим правилам. Пример правила для цикла While: ключевое слово «while», оператор «(«, правило «условное выражение», оператор «)», правило «последовательность команд». Правила «условное выражение» и «последовательность команд» также определены. Соответственно полученное дерево исключает в себе синтаксические ошибки.

Абстрактность понимается как отсутствие дополнительной информации в узле. Эта информация образуется после всех анализов дерева и используется в дальнейшем компилятором. Например, использование переменной до её объявления, можно выявить на этапе анализа дерева, и вывести соответствующую ошибку. Или определить общий тип

операции над числами типа integer и double. На этапе создания АСД, это выявить сложно. После всех возможных анализов дерево становится атрибутным деревом разбора.



2.4. Синтаксический анализ

На вход для анализа подается список лексем, полученных после лексического анализа, а на выходе получается АСД. Такое преобразование достигается описанием синтаксических правил методом рекурсивного спуска. Каждое правило должно определяться однозначно и порождаться одной или несколькими лексемами. Например, правило для While порождается лексемой «ключевое слово «while»» и определяется однозначно как While.

Принцип метода рекурсивного спуска следующий. Обозначим лексемы как терминальные (конечные) символы, а правила – нетерминальные символы. Берем один терминальный символ, определяем его принадлежность правилу, и спускаемся к этому правилу. В правиле берется следующий нетерминальный символ, и также идет спуск, пока правило полностью не определится как элемент АСД. Пример правила «блок или команда»: (терминал «{», нетерминал «команда» [0; N], терминал «}») или (нетерминал «команда»). При спуске в это правило мы проверим лексему, если встретилась фигурная открывающаяся скобка, то будем применять правило «команда» до тех пор, пока не встретится фигурная закрывающаяся скобка. Соответственно если не встретилась фигурная открывающаяся скобка, то выберется альтернатива – правило «команда», в которую спускаемся.

На этом этапе мы можем выделить синтаксические ошибки такие как неожиданный конец файла (когда правилу требуется больше лексем, чем есть), неожиданная лексема (например, после слова if должна стоять кругла скобка, а стоит точка с запятой) и другие.

3. Особенности парсера

1. По мере преобразования лексем актуальность прочитанных лексем теряется, поэтому парсер их удаляет, освобождая при этом память.

2. Есть возможность парсить несколько файлов асинхронно, но это нужно явно указывать в настройках компилятора.

3. Язык поддерживает арифметические выражения внутри строк, соответственно есть специальный подпарсер для этой цели. Пример "a + b = ${a + b}". Это выражение является константной строкой, но часть «${a + b}» будет посчитана при выполнении программы, и подставится в строку.

4. Внедрение новых правил требует небольших изменений в других правилах, как минимум в одном.

4. Примеры

В следующих примерах показана малая часть возможностей парсера, которых нет в других языках программирования.

4.1. Пример объявления функции

Возможности парсера позволяют объявить функцию несколькими способами:

1) func NAME(): (RESULT_VAR_NAME) {
   RESULT_VAR_NAME = SOMETHING
   }
   NAME – имя функции.
   RESULT_VAR_NAME – имя переменной для возвращаемого значения (может быть не одна).
   SOMETHING – какое-то определенное значение.

2) func NAME() = SOMETHING
   Если функция делает какое-либо простое арифметическое действие, то можно её таким образом сократить.

3) func NAME = SOMETHING
   Если у функции нет параметров, то скобки можно опустить.

4.2. Пример объявления функции с несколькими возвращаемыми значениями

```
func `111 my awesome function`(a, b): (quotient, modulo) {
  quotient = a \ b // частное ( \ - целочисленное деление)
  module = a % b // остаток от деления
}
```

Последовательность символов, заключенная в такие кавычки `` (косой апостроф), является идентификатором.

Вызвать такую функцию можно двумя способами:

1) a = 10
   q, m = `111 my awesome function`(a, 2)
   Этот способ поместит результат выполнения в переменные q и m
2) a, q, m = (1, 0, 0)
   `111 my awesome function`(a, q, m)
   В этом способе явно передаются переменные, в которые нужно сохранить результат

Заключение

Целью парсера является построение АСД на основе входных текстовых данных, выявление лексических и синтаксических ошибок и не более. Дальнейшая работа с АСД лежит уже на других компонентах компилятора.

В настоящей работе рассмотрен один из способов создания парсеров для языков программирования. Метод рекурсивного спуска очень прост в написании, но программист должен уверенно знать все аспекты языка для созданий правил.

ЛИТЕРАТУРА

1. Виртуальная машина [Электронный ресурс]: сайт / Википедия – URL: https://ru.wikipedia.org/wiki/Виртуальная_машина (дата обращения: 25.05.2020).

2. Компилятор [Электронный ресурс]: сайт / Википедия – URL: https://ru.wikipedia.org/wiki/Компилятор (дата обращения: 25.05.2020).

3. Язык программирования C++ [Электронный ресурс]: сайт / Википедия – URL: https://ru.wikipedia.org/wiki/C%2B%2B (дата обращения: 25.05.2020).

4. CMake [Электронный ресурс]: сайт / Википедия – URL: https://ru.wikipedia.org/wiki/CMake (дата обращения: 25.05.2020).

5. CLion [Электронный ресурс]: сайт / JetBrains – URL: https://www.jetbrains.com/clion (дата обращения: 25.05.2020).
6. JetBrains [Электронный ресурс]: сайт / JetBrains – URL: https://www.jetbrains.com (дата обращения: 25.05.2020).
7. OpenRSP [Электронный ресурс]: сайт / GitLab – URL: https://gitlab.com/rhinestone-project/openrsp (дата обращения: 25.05.2020).
8. Apache License 2.0 [Электронный ресурс]: сайт / Apache – URL: https://www.apache.org/licenses/LICENSE-2.0 (дата обращения: 25.05.2020).
9. *Чалых Е.П., Самохина С.И.* Виртуальная машина EmeraVM //Труды Томского государственного университета. – Т. 304. Серия физико-математическая: Математическое и программное обеспечение информационных, технических и экономических систем : материалы VII Междунар. молодежной науч. конф. Томск, 23-25 мая 2019 г. Томск: Издательский Дом Томского государственного университета, 2019. С. 210-215.
10. ANTLR [Электронный ресурс]: сайт / ANTLR – URL: https://www.antlr.org (дата обращения: 25.05.2020).
11. GNU Bison [Электронный ресурс]: сайт / GNU – URL: https://www.gnu.org/software/bison (дата обращения: 25.05.2020).
12. БНФ [Электронный ресурс]: сайт / Википедия – URL: https://ru.wikipedia.org/wiki/Форма_Бэкуса_—_Наура (дата обращения: 25.05.2020).
13. ДКА [Электронный ресурс]: сайт / Википедия – URL: https://ru.wikipedia.org/wiki/Детерминированный_конечный_автомат (дата обращения: 25.05.2020).

RESEARCH OF THE CONVERGENCE OF THE FLEXIBLE TOLERANCE METHOD DEPENDING ON THE PARAMETERS VALUES

Alimbaeva E.A., Balashova O.M., Keba A.V.
Tomsk State University
alimb97@mail.ru, balashovajkz@mail.ru, mir.na.mig7@mail.ru

Introduction

There are a considerable number of algorithms for solving the general nonlinear programming problem, which includes the optimization of an objective function subject, in the most general case, to both equality and inequality constraints.

Formally, the nonlinear programming problem can be formulated as follows [2]:

$$\text{minimize } f(\mathbf{x}),\ \mathbf{x} \in E^n, \tag{1}$$

with m linear and (or) nonlinear constraints in the form of equalities

$$h_i(\mathbf{x}) = 0,\ i = \overline{1,m}, \tag{2}$$

and $(p-m)$ linear and (or) nonlinear constraints in the form of inequalities

$$g_i(\mathbf{x}) \geq 0,\ i = \overline{m+1,p}. \tag{3}$$

Nonlinear programming techniques can be roughly divided into two broad categories: 1) direct search methods that depend upon a direct comparison of the values of the objective function; 2) gradient methods that seek the extremum by using first- and perhaps second-order derivatives [1]. Direct-search algorithms are more time consuming in their execution, but the net cost, including preparation time, for the solving the problem may be less than for gradient methods. Thus, the algorithm presented in the article was developed in accordance with the direct-search logic.

This work is focused in constrained nonlinear optimization using the Flexible Tolerance Method (FTM) proposed by Paviani and Himmelblau [5] for minimization of a functional subject to nonlinear equality and inequality constraints.

In this article, the convergence of FTM is investigated depending on the selected combination of parameter values. The article is organized as follows. In section 1, we provide statement of the problem. Section 2 presents a description of FTM and the Nelder-Mead Method (or the Flexible Polyhedron Method or FPM) and a flowchart of FTM. Section 3 illustrates the numerical results, where we provide various figures for different values of the model parameters, and we also numerically compare our results with exist in the literature.