DOI: 10.15514/ISPRAS-2022-34(5)-5

Библиотека для разработки компиляторов

С.В. Миронов, ORCID: 0000-0003-3699-5006 <mironovsv@sgu.ru>
И.А. Батраева, ORCID: 0000-0002-6539-8473 <batraevaia@info.sgu.ru>
П.Д. Дунаев, ORCID: 0000-0002-9142-0945 <herrpaulvondonau@outlook.com>

Саратовский государственный университет,
410012, Россия, г. Саратов, ул. Астраханская, 83

Аннотация. Работа посвящена разработке библиотеки, предназначенной для реализации компиляторов. Статья содержит описание возможностей библиотеки и основных принципов её функционирования. В ходе работы была изучена и реализована генерация синтаксических анализаторов с помощью LR(1)-автоматов, были спроектированы и реализованы два вспомогательных языка: язык запросов к семантической сети и язык, предназначенный для генерации исполняемого кода. Результатом работы является библиотека для платформы .NET (библиотека тестировалась, в частности, для языка C#), которая содержит классы, существенно облегчающие реализацию синтаксического анализа исходного кода, семантического анализа и генерацию исполняемого файла. Данная библиотека не имеет внешних зависимостей, кроме стандартной библиотеки .NET.

Ключевые слова: библиотека; компилятор; генерация синтаксических анализаторов; кодогенерация; .NET

Для цитирования: Миронов С.В., Батраева И.А., Дунаев П.Д. Библиотека для разработки компиляторов. Труды ИСП РАН, том 34, вып. 5, 2022 г., стр. 77-88. DOI: 10.15514/ISPRAS-2022-34(5)-5

Library for Development of Compilers

S.V. Mironov, ORCID: 0000-0003-3699-5006 <mironovsv@sgu.ru>
I.A. Batraeva, ORCID: 0000-0002-6539-8473 <batraevaia@info.sgu.ru>
P.D. Dunaev, ORCID: 0000-0002-9142-0945 <herrpaulvondonau@outlook.com>

Saratov State University,
83 Astrakhanskaya Street, Saratov, Russia, 410012

Abstract. This work is devoted to the development of a library designed to implement compilers. The article contains a description of the library's features and the main points of its functioning. In the course of the work, the generation of parsers using LR(1)-automata was studied and implemented, two auxiliary languages were designed and implemented: a semantic network query language and a language designed to generate executable code. The result of the work is a library for the platform .NET (the library was tested for the C# language), which contains classes that make easier the implementation of source code parsing, semantic analysis, and executable file generation. This library does not have external dependencies, except for the standard .NET library.

Keywords: library; compiler; parsers generation; code generation; .NET

For citation: Mironov S.V., Batraeva I.A., Dunaev P.D. Library for Development of Compilers. Trudy ISP RAN/Proc. ISP RAS, vol. 34, issue 5, 2022. pp. 77-88 (in Russian). DOI: 10.15514/ISPRAS-2022-34(5)-5

1. Введение

Потребность в новых языках программирования постоянно растёт. Разработчики могут столкнуться с необходимостью создать предметно-ориентированный язык. Некоторым командам может потребоваться разработать свой компилятор языка общего назначения или собственного его диалекта. Энтузиасты продолжают воплощать свои идеи, создавая новые языки программирования. Разработка компилятора является сложной задачей, часть решения которой может быть вынесена в качестве библиотечных функций для использования в произвольном количестве проектов. Цель данной работы – предложить новый отечественный открытый продукт, позволяющий упростить и ускорить их разработку. Библиотека призвана стать наиболее удобным инструментом для быстрой и лаконичной реализации предметно-ориентированных языков в рамках промышленной разработки и для начинающих разработчиков компиляторов.

Библиотека[1] реализована на языке C#, она не имеет иных зависимостей, кроме стандартной библиотеки .NET. Использование библиотеки возможно на любых языках для платформы .NET 6.0 и выше. Таким образом, компиляторы, использующие библиотеку, являются кроссплатформенными, однако генерировать исполняемые файлы через библиотечные функции в момент написания статьи можно пока только для 64-битных операционных систем семейства Windows.

2. Анализ существующих решений

Функциональность библиотеки можно разделить на три следующих блока.

1) Синтаксический анализ. Библиотека содержит классы для конструирования и использования парсеров. Грамматика языка описывается в пользовательских классах с помощью атрибутов – особых конструкций .NET-языков, снабжающих различные элементы метаданных дополнительной информацией, которыми помечаются, в данном случае, методы-обработчики продукций. Обработчик должен возвращать задаваемое пользователем внутреннее представление описываемой продукцией конструкции;
2) Семантическая сеть. В библиотеке предусмотрены классы для описания отношений между сущностями кода (такими как классы, методы, переменные и т. д.) и решения задачи поиска объекта, связанного более сложными отношениями с данным (например, отношение доступности между областью видимости и переменной, полем или классом). Такой поиск кодируется с помощью встроенного в библиотеку языка запросов.
3) Генерация исполняемого кода. В библиотеку встроен специальный язык кодогенерации. Это процедурный язык, который реализован “поверх” .NET-языка, на котором реализуется компилятор, т.е. его конструкции представляют собой вызовы методов определённых в библиотеке классов, включая перегруженные операторы.

Первый блок может быть реализован с помощью генерации различных типов распознавателей. В частности, существуют такие распознаватели, как LL [1], LR [2] и LALR [3], в том числе их обобщённые (generalized) модификации (GLL, GLR) [4], способные обрабатывать любые контекстно-свободные грамматики, толерантные модификации [5], предназначенные для распознавания определённых, интересных в рамках конкретных задач участков кода, а также “ленивые” модификации [6], которые позволяют исключить временные затраты на разбор неиспользуемых определений языка. Библиотека генерирует классические LR(1)-автоматы, поскольку они имеют большую распознавательную способность.

В качестве аналогов первого блока можно выделить GNU Bison [7], ANTLR [8] и Coco/R [9]. С их помощью анализатор строится следующим образом: разработчик описывает грамматику с помощью специального языка, этот код преобразуется генератором в парсер для целевого

[1] https://github.com/herrpaulvd/CompileLib

языка, который, в конечном счёте, используется в проекте компилятора. Наше решение требует определённым образом описанные классы, которые анализируются во время выполнения с помощью рефлексии. Преимуществом нашего подхода является быстрая скорость внесения изменений в описание парсера, в то время как при использовании аналогов после корректировки описания грамматики требуется повторно генерировать парсер, заменяя им результат старой генерации. В качестве недостатка выступают затраты времени исполнения на построение синтаксического анализатора при каждом запуске компилятора, что частично решается кэшированием построенного распознавателя. Описанные преимущества и недостатки делают наш генератор парсеров более подходящим для языков, от компилятора которых не требуется высокой производительности, например, некоторых предметно-ориентированных языков, и менее подходящим для компиляторов языков с богатым синтаксисом, таких как, например, C++.

Аналогом второго блока выступает продвинутый инструмент, описанный в статье [10] и позволяющий устанавливать связи между сущностями программ на языках C и C++. Наша библиотека предлагает инструментарий для решения этой задачи для любого языка и для произвольных связей, которые достаточно описать с помощью декларативного предметно-ориентированного языка, близкого к языку теории множеств.

Третий блок может быть реализован двумя способами: с помощью высокоуровневых виртуальных машин, таких как LLVM [11] или System.Reflection.Emit [12], или генерации кода на другом языке, например, на C или языке ассемблера, отправляя результат в соответствующий компилятор или ассемблер. В библиотеке для генерации кода предложен свой язык, конструкции которого представляют собой вызовы библиотечных методов. Синтаксис языка является более сложным, чем просто последовательность инструкций – язык допускает составные выражения, компонентами которых могут быть арифметические операции, вызовы функций и индексация указателей как в C. Такой подход перенимает преимущества генерации кода на другом высокоуровневым языке, исключая временные затраты на синтаксический анализ сгенерированного кода (он фактически происходит во время компиляции кода самого компилятора).

В отличие от всех вышеперечисленных аналогов, библиотека является комплексным решением, включающим в себя инструменты для всех компонентов компилятора, хотя возможно использование блоков библиотеки по-отдельности. Целостные решения являются обычно более предпочтительными при отсутствии специфических требований, поскольку позволяют сэкономить время на подбор подходящей комбинации фреймворков. Все блоки объединены в одну DLL, что значительно упрощает использование библиотеки: для её использования достаточно добавить в проект всего одну DLL, не производя дополнительных настроек. Таким образом, библиотека подходит, в первую очередь, для быстрой и простой разработки компиляторов, что делает её использование удобным в основном в рамках двух сценариев:

1) реализация предметно-ориентированных языков при отсутствии необходимости обеспечить высокую производительность компилятора или исполняемого кода;
2) первые шаги в создании собственных компиляторов, когда важно, в первую очередь, получить представление о самом процессе разработки компилятора, о его компонентах, не углубляясь при этом в детали, и закрепить изученное, разработав свой “игрушечный” компилятор.

3. Возможности библиотеки

В этом разделе описаны основные возможности библиотеки и некоторые идеи, использованные для их реализации.

3.1 Лексический и синтаксический анализ

Для конструирования парсера предназначен класс `ParsingEngineBuilder`. Он содержит метод, принимающий в качестве параметров идентификатор лексемы, регулярное выражение стандарта POSIX [13] и произвольное количество определяемых пользователем классов символов, передаваемые как замыкания вида `char -> bool`.

В качестве распознавателя лексем применяются недетерминированные конечные автоматы (НКА), модифицированные в целях экономии памяти. Использование своего варианта НКА мотивировано тем, что реализации обычных НКА, построенных на основе регулярных выражений, должны хранить множество однотипных переходов. Например, регулярному выражению `[[:print:]]*` соответствует НКА с одним состоянием, из которого существует n переходов в себя же, где n – количество печатных символов, которых, например, в Unicode более 60000. Необходим способ, который бы уменьшал количество хранимых переходов.

Основная идея модификации НКА – объединить некоторые группы символов в один, так чтобы автомат работал так, как если бы ему на вход вместо обычных символов поступали “объединённые”. Если обычные НКА представлены пятёркой $(Q, \Sigma, \delta, q_0, F)$, где Q – множество состояний автомата, Σ – входной алфавит, $\delta: Q \times \Sigma \longrightarrow 2^Q$ – функция переходов, q_0 – начальное состояние, $F \subseteq Q$ – множество конечных состояний, то в нашем представлении автомат описывается семёркой, в которую дополнительно входят U – промежуточный алфавит, $\gamma: \Sigma \longrightarrow 2^U$ – функция неоднозначного преобразования символов входного алфавита в символы промежуточного, причём изменён тип функции переходов $\delta: Q \times \mathrm{U} \longrightarrow Q$.

На каждой итерации автомат сначала считывает очередной входной символ, затем преобразует его в символ промежуточного алфавита (этот шаг недетерминирован), после этого в зависимости от полученного символа и текущего состояния автомат переводится в новое состояние (этот шаг детерминирован, в отличие от аналогичного шага в обычных НКА). Нетрудно показать, что каждый обычный НКА можно преобразовать в модифицированный, и наоборот, т. е. мощности этих типов распознавателей совпадают.

Использовать модифицированный НКА предлагается следующим образом. В качестве промежуточного алфавита используется множество всех вхождений одиночных символов и выражений в квадратных скобках для преобразуемого регулярного выражения. Например, для выражения `_[[:alnum:]]+_[01]?` промежуточный алфавит будет состоять из 4 символов: первое вхождение символа `'_'`, скобочное выражение `[[:alnum:]]`, представляющее множество всех букв и цифр, второе вхождение символа `'_'`, выражение `[01]`. Функция γ не строится – она будет определена неявно: каждый символ входного алфавита будет переводиться во все те символы промежуточного алфавита, которые представляют регулярные подвыражения, которым удовлетворяет этот входной символ.

Прочие элементы строятся согласно алгоритму построения обычного НКА по регулярному выражению [14], как если бы на вход было передано исходное регулярное выражение, в котором одиночные символы и выражения в квадратных скобках заменены на соответствующие символы промежуточного алфавита. При этом в результате применения алгоритма [14] может получиться так, что для некоторых $q \in Q$ и $u \in U$ функция переходов возвращает несколько значений, в то время как, согласно определению модифицированного НКА, значение функции должно быть определено однозначно. В качестве выхода из этой ситуации можно заменить символ u на n символов $u_1, u_2, \cdots, u_n$, где n – количество переходов, каждому переходу, таким образом, будет соответствовать свой символ промежуточного алфавита, каждый символ представляет то же регулярное подвыражение, что и u.

Каждый промежуточный символ будет представлен предикатом вида `char -> bool`, который определяет, удовлетворяет ли аргумент данному регулярному подвыражению, а

список переходов будет храниться в виде массива, в котором каждому индексу-состоянию i будет соответствовать список пар вида (p, j), где p – предикат, j – новое состояние. НКА будет работать по следующему алгоритму:

1) автомат начинает работу в множестве состояний $Q_{curr} = \{q_0\}$; функция переходов описана массивом T, в котором каждому состоянию q_i сопоставлено множество $T[q_i] = \{(p_{i,j}, q'_{i,j}) \mid j = \overline{1, k_i}\}$;
2) считывается входной символ c;
3) множество состояний заменяется на новое: $Q_{new} \coloneqq \{q' \mid (p, q') \in T[q], q \in Q_{curr}, p(q)$ истинно$\}$; $Q_{curr} \coloneqq Q_{new}$.

Для регулярного выражения `[[:print:]]*`, рассмотренного выше, будет храниться массив из одного элемента (поскольку состояние только одно), этим элементом будет список из одной пары: первым элементом пары будет предикат $p(c) = (20_{16} \le c \le 7E_{16}) \vee (c \ge 100_{16})$, который проверяет, является ли символ печатным в Unicode, вторым – единственное состояние автомата.

Синтаксис языка задаётся с помощью методов-обработчиков продукций, сами продукции задаются с помощью атрибутов, которыми помечается метод и его параметры. Отсутствует возможность задать не контекстно-свободную грамматику. Метод помечается атрибутом `SetTag`, в котором указывается идентификатор нетерминала, находящегося в левой части продукции. Каждому параметру, кроме, возможно, последнего, соответствует как минимум один символ продукции. Порядок параметров совпадает с порядком символов в продукции. Последний параметр может быть зарезервирован для обработчика ошибок.

Параметр может быть помечен одним из атрибутом повторения. С их помощью можно упростить описание грамматики. Например, если предполагается, что один элемент в некоторой конструкции может отсутствовать, можно использовать атрибут `Optional`, определив одну продукцию вместо нескольких.

Каждый параметр, кроме, возможно, последнего, должен быть помечен

ровно одним атрибутом, задающим символ грамматики, и не более чем одним

атрибутом, задающим повторение символов.

Символ грамматики могут задавать следующие атрибуты.

- `RequireTags(params string[] tags)` задаёт множество допустимых символов (лексем или нетерминалов). Указание нескольких символов позволяет избежать определения нескольких продукций для языковой конструкции, которая предполагает использование в одинаковом качестве разнотипных элементов. Например, в качестве элементов выражений могут выступать как числовые константы, так и идентификаторы.
- `Keywords(params string[] keywords)` задаёт множество допустимых ключевых слов (здесь под ними понимаются лексемы, которые задают единственную строку и не имеют собственного идентификатора. Ключевые слова передаются в качестве параметром атрибута “как есть”, т. е. они интерпретируются как обычные строки, но не регулярные выражения).

Повторение символов задаётся следующими атрибутами.

- `Single` – символ повторяется ровно один раз. Если атрибут повторения не указан, применяется по умолчанию;
- `Optional(bool greedy)` – символ является необязательным, т. е. может отсутствовать. Если параметр `greedy` установлен в `true`, то при наличии символа он обязательно будет прочитан. Если символ не прочитан, значение параметра устанавливается в `null`;
- `Many(bool canBeEmpty)` – символ может повторяться более одного раза. Если `canBeEmpty = true`, символ может ни разу не повториться. Помеченный этим

атрибутом параметр, а также следующие за ним, помеченные `TogetherWith`, должны иметь тип массива. Длина массива есть количество прочитанных повторений;

- `TogetherWith` —удлиняет цепочку символов, которая попадает под модификатор повторения. Первый параметр не может быть помечен данным атрибутом. Например, если i-й параметр и идущий перед ним помечены данным атрибутом, а $(i-2)$-й параметр помечен атрибутом `Optional`, то все три символа будут одновременно прочитаны или не прочитаны.

В качестве примера рассмотрим сигнатуру обработчика оператора `if` в C-подобных языках, код представлен на C#:

```
[SetTag("statement")]
public static Statement? ReadIfStatement(
    [Keywords("if")] Parsed<string> kw,
    [Keywords("(")] string brOpen,
    [RequireTags("expr")] Expression condition,
    [Keywords(")")] string brClose,
    [RequireTags("statement")] Statement ifBranch,
    [Optional(true)][Keywords("else")] string kwElse,
    [TogetherWith][RequireTags("statement")] Statement elseBranch)
```

Здесь `Expression` и `Statement` – пользовательские классы, определяющие конструкции языка. Ключевое слово `else` помечено атрибутом `Optional(true)`, за ним идёт ветка, помеченная `TogetherWith`. Поскольку параметр `greedy = true`, ветка `else`, следующая за остальной частью оператора, будет считываться и присоединяться к оператору. Таким образом, атрибут `Optional(true)` решает, в частности, проблему “висящего else”, описанную в [15], избавляя разработчика от необходимости разбивать описание оператора на несколько продукций.

Последний параметр обработчика может быть зарезервирован для обработки ошибок. Он помечается атрибутом `ErrorHandler` и имеет тип `ErrorHandlingDecider`. При отсутствии ошибок параметр имеет значение `null`, иначе представляет объект, с помощью которого принимается решение об обработке ошибки. При обнаружении недопустимой лексемы обработчик находит подходящую конструкцию, структура которой, возможно, была нарушена, и обработчик которой имеет соответствующий последний параметр. Обработчик, получив информацию о данной лексеме и о уже прочитанной части конструкции, принимает решение о том, как лексема должна быть обработана: например, проигнорирована, заменена, передана другому обработчику, или нужно остановить весь анализ. Принятие решения осуществляется через вызов соответствующих методов объекта `ErrorHandlingDecider`.

Класс, содержащий обработчики, передаётся объекту `ParsingEngineBuilder`. После того, как полностью были заданы лексика и синтаксис языка, вызывается метод, конструирующий объект класса `ParsingEngine`, содержащий набор автоматов для проведения лексического анализа и LR(1)-автомат, осуществляющий синтаксический анализ. В случае, если грамматика не является LR(1)-грамматикой, метод выбрасывает исключение, содержащее подробную информацию о возможных неоднозначностях.

3.2 Семантическая сеть и язык запросов

Библиотека предоставляет возможность организовывать такие элементы кода, как классы, методы, переменные и т. д. в семантическую сеть и писать к этой сети запросы. Объект семантической сети должен быть экземпляром класса-наследника `CodeObject`. Объекту должно быть задано имя (не обязательно уникальное) и тип, кроме того, он имеет две встроенные коллекции. Первая коллекция содержит атрибуты объекта, заданные в виде пар имя атрибута – значение атрибута, вторая – отношения с другими объектами, заданные в виде пар имя отношения – объект. Отношения, хранящиеся в данной коллекции, являются

простыми, как, например, отношения между классом и его членом. При решении задач, связанных с семантическим анализом, требуется устанавливать более сложные взаимосвязи, такие как, например, поиск объекта с заданным именем, видимого в заданной области. Для описания подобных сложных отношений был разработан специальный язык запросов.

Скрипт на этом языке состоит из правил – конструкций, которые описывают сложные отношения. Каждое правило имеет имя, набор явных параметров, ровно один неявный параметр и тело. Неявным параметром всегда является объект, относительно которого начинается поиск.

Правило имеет вид `@name(params) = expression;` где `name` – имя правила, `params` – имена явных параметров, указанных через запятую, `expression` – тело правила. Последнее является выражением, конструируемым по следующим правилам:

1) `name(obj_name)` – представляет собой множество объектов, связанных с данным каким-либо отношением и имеющих имя `obj_name`;
2) `type(obj_type)` – множество объектов, связанных с данным каким-либо отношением и имеющих тип `obj_type`;
3) `relation(relation_name)` – множество объектов, связанных с данным отношением `relation_name`;
4) `attribute(attr_name)` – множество объектов, связанных с данным каким-либо отношением и имеющих атрибут `attr_name` с любым значением;
5) `attribute(attr_name, attr_value)` – множество объектов, связанных с данным каким-либо отношением и имеющих атрибут `attr_name` со значением `attr_value`;
6) `@rule(params)` – множество объектов, получаемое запуском правила `rule` с заданными явными параметрами и данным объектом в качестве неявного;
7) Если `X` и `Y` – выражения языка запросов, то
   a) `(X) & (Y)` – пересечение множеств, представленных выражениями;
   b) `(X) | (Y)` – объединение множеств;
   c) `(X) ?| (Y)` возвращает `X`, если `X` не пусто, иначе `Y`;
   d) `(X) . (Y)` – вычисляется множество `X`, к каждому элементу которого как к неявному параметру применяется выражение `Y` как самостоятельное правило, результаты всех применений выражения `Y` объединяются в одно множество, возвращаемое данным выражением;
   e) `(X) +. (Y)` аналогично `(X) | ((X) . (Y))`;
   f) Круглые скобки могут быть опущены в соответствии со следующим приоритетом операций: наивысший приоритет – `&`, средний – `.` и `.+`, низший – `|` и `?|`;
8) Другие выражения недопустимы.

Правила могут быть рекурсивными. Скрипт передаётся в виде строки объекту класса `SemanticNetwork`, в котором содержатся методы для вызова правил по имени правила и объекту – неявному параметру. Ниже приведён пример правила для поиска объектов в локальной области видимости по заданному имени:

```
@local-search(@name) =
   name(@name) & type(local-var) #1
   ?| relation(parent) & type(scope) . @local-search(@name) #2
   ?| relation(parent) & type(method)
      . name(@name) & relation(parameter) #3
   ?| relation(parent) & type(method) & attribute(static)
      . relation(parent)
      . @class-search(@name, static, private) #4
   ?| relation(parent) & type(method) & attribute(instance)
      . relation(parent) . @class-search(@name, anymember,
```



```
      private) #5
   ?| relation(parent) & type(method) . @global-search(@name); #6
```

В качестве неявного параметра выступает локальная область видимости. Поиск происходит в несколько этапов (номера отмечены однострочными комментариями #), переход к следующему этапу происходит только если на предыдущем ничего не было найдено:

1) правило пытается найти переменные, находящиеся непосредственно в данной области видимости;
2) если таких нет, и эта область вложена ещё в одну такую же, поиск рекурсивно переходит в эту область;
3) если непосредственным предком области является метод, проверяются параметры метода;
4) если метод статический, передаётся управление другому правилу – `@class-search`, которое запускает поиск статических членов класса с заданным именем;
5) если метод экземплярный, передаётся управление правилу `@class-search`, которое запускает поиск любых членов класса с заданным именем;
6) происходит поиск в глобальной области видимости с помощью правила `@global-search`.

Как было показано выше, это правило использует два других пользовательских правила: `@class-search` и `@global-search`. Последнее принимает на вход только имя искомого объекта, который должен находиться в глобальной области видимости. Правило `@class-search` имеет 3 параметра: имя искомого объекта, значение атрибута, определяющего, является ли член класса статическим (`static`), экземплярным (`instance`), или должны рассматриваться оба варианта (`anymember`). Третий параметр определяет минимальную допустимую доступность члена класса (`private < protected < public`).

Предполагается, что семантическая сеть построена так, что каждый член класса обязательно имеет атрибут `anymember`, ровно один из атрибутов `static` или `instance`, а также набор атрибутов доступности, максимальным из которых является атрибут, соответствующий заданному в исходном коде модификатору, минимальным – `private`, а также представлены все атрибуты, находящиеся между ними в иерархии доступности. Так, публичные члены класса будут иметь все три атрибута доступности, protected-члены – `protected` и `private`, приватные члены – только `private`. Таким образом, избегается необходимость дублировать вызов `@class-search` для всех модификаторов доступа, достаточно лишь указать минимальный требуемый. В примере требуется поиск членов класса с любым модификатором доступа, поэтому в качестве параметра указан наименьший из них – `private`.

3.3 Язык генерации исполняемого кода

Это процедурный язык программирования, кодирование на котором осуществляется с помощью вызова соответствующих методов библиотеки. Он поддерживает различные типы, в числе которых есть:

1) целочисленные, знаковые и беззнаковые, размером в 1, 2, 4 и 8 байт;
2) вещественные, по 4 и 8 байт;
3) аналог типа `void`, используемый только для объявления функций, не предусматривающих возврата значения;
4) указатели;
5) структуры

При задании структур существует возможность указать выравнивание полей, так чтобы смещение каждого нового поля было кратно этому значению (аналогично `#pragma pack(n)` в C).

В языке можно определять собственные функции и импортировать функции из сторонних DLL, например, из системных kernel32.dll или user32.dll. Помимо функций, в языке можно объявлять глобальные и локальные переменные, константы, а также блоки инициализированных данных. Предусмотрены различные типы бинарных и унарных операций, допустима арифметика указателей. Указатели также могут быть индексированы.

Помимо функций, в качестве конструкций, управляющих потоком выполнения программы, в языке предусмотрены метки и операторы безусловного и условного переходов.

Ниже приведён участок кода, который определяет аналог функции `malloc` через системную функцию `HeapAlloc` (основной язык: C#) [16]:

```
compiler.OpenEntryPoint();
var hHeap = compiler.AddGlobalVariable(ELType.PVoid);
var GetProcessHeap = compiler.ImportFunction("kernel32.dll",
    "GetProcessHeap", ELType.PVoid);
var HeapAlloc = compiler.ImportFunction("kernel32.dll",
    "HeapAlloc", ELType.PVoid, ELType.PVoid,
    ELType.UInt32,ELType.UInt64);
hHeap.Value = GetProcessHeap.Call();
var malloc = compiler.CreateFunction(ELType.PVoid, ELType.UInt64);
var pSize = malloc.GetParameter(0);
malloc.Open();
compiler.Return(HeapAlloc.Call(hHeap,compiler.MakeConst(0U),
    pSize));
```

Объект `compiler` класса `ELCompiler` используется для определения различных объектов, таких как функции или глобальные переменные, а также для использования различных операторов или конструкций. В первой строке происходит обращение к точке входа – предопределённой функции. Во второй строке добавляется глобальная переменная `hHeap`, в которой будет сохранён дескриптор кучи, он имеет тип `void*`. Далее импортируются функции `GetProcessHeap` и `HeapAlloc` из kernel32.dll, предназначенные для получения дескриптора кучи и выделения области памяти в куче соответственно. Затем вызывается функция `GetProcessHeap`, возвращённое ей значение присваивается переменной `hHeap`. Далее определяется пользовательская функция `void* malloc(UINT64)`, и она же открывается на запись. В функцию добавляется единственное выражение – возврат результата вызова `HeapAlloc`, которой в качестве первого аргумента передаётся дескриптор кучи, в качестве второго – пустое множество флагов в виде 32-битного беззнакового нуля, в качестве последнего – размер выделяемой памяти.

Предполагается, что большая часть кода на языке кодогенерации будет расположена небольшими фрагментами между другими частями кода компилятора, также отвечающими за кодогенерацию. В качестве примера подобного случая приведём компиляцию оператора `if` (основной язык: C#):

```
var elselabel = compiler.DefineLabel(); // (1)
var endlabel = compiler.DefineLabel(); // (1)
var expr = Condition.CompileRight(compilation); // (2)
var texpr = Condition.Type;
if (!texpr.IsIntegerType())
    throw new CompilationError($"Invalid condition type
{texpr.Show(name2class)}", Line, Column);
compiler.GotoIf(!expr, elselabel); // (1)
CodeObject ifscope = new("", "scope", -1, -1);
ifscope.AddRelation("parent", compilation.Scope);
```



```
IfBranch.Compile(compilation.WithScope(ifscope)); // (2)
compiler.Goto(endlabel); // (1)
compiler.MarkLabel(elselabel); // (1)
if(ElseBranch is not null)
{
    CodeObject elsescope = new("", "scope", -1, -1);
    elsescope.AddRelation("parent", compilation.Scope);
    ElseBranch.Compile(compilation.WithScope(elsescope)); // (2)
}
compiler.MarkLabel(endlabel); // (1)
```

Здесь комментариями `(1)` отмечены строки, в которых расположены выражения языка кодогенерации, комментариями `(2)` отмечены вызовы методов, которые содержат другие выражения языка.

Сгенерированный код преобразуется в машинный код процессоров архитектуры AMD64, для написания библиотеки использовалась официальная документация [17], в качестве формата исполняемого файла был выбран Portable Executable, структура которого подробно описана в [18]. Генератор исполняемого кода не использует сторонних библиотек (кроме стандартной библиотеки .NET), ассемблеров или компоновщиков.

4. Заключение

В статье была рассмотрена функциональность библиотеки для создания компиляторов и ряд идей, использовавшихся при её написании, в частности, идея модификации определения недетерминированного конечного автомата для более компактного хранения распознавателя в оперативной памяти.

Были рассмотрены три возможности библиотеки: средства описания синтаксиса языка и синтаксического анализа, классы для построения семантической сети и язык запросов к ней, язык генерации исполняемого кода.

В качестве примера для тестирования на языке C# был разработан компилятор C-подобного языка, частично поддерживающего объектно-ориентированное программирование. Компилятор использовал все три функциональных блока библиотеки. Другие библиотеки, кроме описанной и стандартной, не были включены в проект.

Список литературы / References

[1] Lewis P., Stearns R. Syntax-Directed Transduction. Journal of the ACM, vol. 15, issue 3, 1968, pp. 465-488.

[2] Knuth D. On the Translation Languages from Left to Right. Information and Control, vol. 8, issue 6, 1965, pp. 607-639.

[3] DeRemer F. Practical Translators for LR(k) Languages. PhD Thesis, Massachusetts Institute of Technology, 1969, 215 p.

[4] Lang B. Deterministic Techniques for Efficient Non-Deterministic Parsers. Lecture Notes in Computer Science, vol. 14, 1974, pp. 255-269.

[5] Goloveshkin A.V. Tolerant parsing using modified LR(1) and LL(1) algorithms with embedded “Any” symbol. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 3, 2019, pp. 7-28. DOI: 10.15514/ISPRAS-2019-31(3)-1.

[6] Савицкий В.О., Сидоров Д.В. «Ленивый» анализ исходного кода на языках C и C++. Труды ИСП РАН, том 23, 2012 г., стр. 133-142 / Savitsky V.O., Sidorov D.V. Lazy source code analysis for C/C++ languages. Trudy ISP RAN/Proc. ISP RAS, vol. 23, 2012, pp. 133-142 (in Russian). DOI: 10.15514/ISPRAS-2012-23-8.

[7] GNU Bison. Available at: https://www.gnu.org/software/bison/.

[8] ANTLR. Available at: https://www.antlr.org/.

[9] Coco/R. Available at: https://ssw.jku.at/Research/Projects/Coco/.

[10] Белеванцев А.А., Велесевич Е.А. Анализ сущностей программ на языках Си/Си++ и связей между ними для понимания программ. Труды ИСП РАН, том 27, вып. 2, 2015 г., стр. 53-64. / A. Belevantsev,

E. Velesevich. Analyzing C/C++ Code Entities and Relations for Program Understanding Trudy ISP RAN /Proc. ISP RAS, vol. 27, issue 2, 2015, pp. 53-64 (in Russian). DOI: 10.15514/ISPRAS-2015-27(2)-4.

[11] The LLVM Compiler Infrastructure. Available at: https://llvm.org/.

[12] Microsoft Docs: System.Reflection.Emit Namespace. Available at: https://learn.microsoft.com/en-us/dotnet/api/system.reflection.emit?view=netstandard-2.0.

[13] The Open Group Base Specifications Issue 7, 2018 edition, IEEE Std 1003.1™-2017 (Revision of IEEE Std 1003.1-2008). Chapter 9. Regular Expressions. Available at: https://pubs.opengroup.org/onlinepubs/9699919799.

[14] Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. Том 1: Синтаксический анализ. М., Издательство МИР, 1978, 616 стр. / Aho A., Ullman J. The Theory of Parsing, Translation and Compiling. Volume 1: Parsing. Prentice Hall, 1972, 542 p.

[15] Ахо А., Лам М. и др. Компиляторы. Принципы, технологии и инструментарий. М., Издательский дом Вильямс, 2008, 1178 стр. / Aho A., Lam M. et al. Principles, Techniques, & Tools. M., Addison Wesley, 2006, 1040 p.

[16] Microsoft Docs: HeapAlloc function (heapapi.h). Available at: https://learn.microsoft.com/en-us/windows/win32/api/heapapi/nf-heapapi-heapalloc.

[17] AMD64 Architecture Programmer's Manual: Volumes 1-5. Available at: https://www.amd.com/system/files/TechDocs/40332_4.05.pdf.

[18] Microsoft Portable Executable and Common Object File Format Specification. Revision 11 – June 20, 2017. Available at: https://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/pecoff.docx.

Информация об авторах / Information about authors

Сергей Владимирович МИРОНОВ – кандидат физико-математических наук, доцент, декан факультета компьютерных наук и информационных технологий. Сфера научных интересов: методы сокращения диагностической информации с использованием словарей неисправностей, формальные языки и грамматики, функциональное программирование.

Sergei Vladimirovich MIRONOV – Candidate of Science in Physics and Mathematics, Associate Professor, Dean of the Faculty of Computer Science and Information Technologies. Research interests: methods of diagnostic information compression using fault dictionaries, formal languages and grammars, functional programming.

Инна Александровна БАТРАЕВА – кандидат физико-математических наук, доцент, заведующая кафедрой технологий программирования. Сфера научных интересов: дискретная математика, теория автоматов, теория формальных языков и грамматик, информационные системы в теоретической и прикладной лингвистике.

Inna Aleksandrovna BATRAEVA – Candidate of Science in Physics and Mathematics, Associate Professor, Head of the Department of Programming Technologies. Research interests: discrete mathematics, automata theory, theory of formal languages and grammars, information systems in theoretical and applied linguistics.

Павел Дмитриевич ДУНАЕВ – магистрант направления «Математическое обеспечение и администрирование информационных систем». Сфера научных интересов: компиляторы, операционные системы, дискретная математика.

Pavel Dmitrievich DUNAEV – Master's Student of Saratov State University. Research interests: compilers, operating systems, discrete mathematics.