УДК 004.432.2

Проектирование и разработка компилятора С-подобного языка программирования общего назначения с поддержкой исполнения кода на этапе компиляции

Н.М. Ткачёв, А.И. Андрюхин
Донецкий национальный технический университет
m4dbrat@gmail.com, alexandruckin@rambler.ru

Ткачёв Н.М., Андрюхин А.И. Проектирование и разработка компилятора С-подобного языка программирования общего назначения с поддержкой исполнения кода на этапе компиляции. *В статье представлен краткий обзор современных принципов проектирования языков программирования и сформированы требования к проектируемому языку. Спроектирована начальная итерация языка программирования общего назначения и кратко описана структура разрабатываемого компилятора. Перспективами развития языка является расширение системы типов и внедрение в компилятор комплексных средств анализа кода для предотвращения ошибок и исполнения кода на этапе компиляции.*

Введение

Задача проектирования новых языков программирования в наше время является как никогда актуальной, поскольку широкое распространение компьютерных технологий ставит задачи максимальной оптимизации процесса разработки и написания кода в различных предметных областях. Именно из-за широкого использования тех или иных технологий становятся более заметными их недостатки, потому новые средства призваны их исправлять [1].

Высокоуровневые средства программирования значительно ускоряют решение типичных задач, но зачастую их гибкости может не хватать, поскольку обстоятельства решаемой задачи требуют более точного контроля над ситуацией. В частности, наиболее распространённой является проблема производительности, когда управление абстракциями приводит к вычислениям, не влияющим на конечный результат. Другой проблемой являются «протекающие абстракции» - явление, при котором программисту приходится отвлекаться от решения задачи на попытки понять внутреннее устройство используемых им средств. Как правило, такая проблема возникает при недостаточном внимании при проектировании средств разработки или несовместимости парадигм среды разработки, языка и библиотек. Например, Win32 API пытается сочетать объектно-ориентированную модель на основе обмена сообщениями вместе с низкоуровневым процедурным стилем языка С, из-за чего программисту приходиться выполнять множество неочевидных действий и увеличиваются временные затраты на отладку.

Язык программирования является низшим слоем, на котором держится процесс разработки, потому важно убедиться в том, чтобы разработчики могли чётко выражать в нём свои намерения (абстракции и алгоритмы, оперирующие ими), по минимуму отвлекаясь на языковые особенности («бороться» с языком).

Цель работы: исследование современных принципов проектирования языков программирования и их проблем, проектирование языка на их основе полученных сведений, а также разработка и развитие компилятора этого языка.

1 Цели при проектировании языков программирования

Ввиду развития и распространения множества различных парадигм программирования были выделены следующие требования к языкам, выполнение которых позволяет оптимизировать процесс разработки:

1. Язык программирования должен иметь синтаксис и модель работы, которые позволяют максимально просто записывать некоторые манипуляции и в то же время позволяют чётко их видеть при чтении кода. Такие языки, как Java, могут требовать написания сотен строк повторяющегося кода для решения задач, которые языки наподобие Haskell могут решить в одну строку. Однако необходимо помнить, что более краткий код может использовать более сложные понятия, потому на его понимание может уйти гораздо больше времени, чем на чтение аналогичного, но более длинного кода. Баланс между этими сторонами должен соблюдаться как

программистами, так и проектировщиками языка. Наиболее распространённой формой языка в этом плане являются языки с C-подобным синтаксисом и объектно-ориентированной моделью с некоторыми элементами функционального программирования (например, lambda-выражениями и замыканиями).

2. Язык, спроектированный для определённой предметной области, должен позволять прозрачно манипулировать понятиями данной области. Язык общего назначения должен быть достаточно гибким для формирования абстракций различных предметных областей, а также давать возможность построения связей между этими областями.

3. Язык должен предотвращать типичные для языков более низкого уровня логические ошибки, превращая их в семантические. Обычно к этой цели стремятся языки с гибкой системой типов, позволяя программистам описывать допустимые и недопустимые связи между объектами в определённых общих понятиях.

4. Абстракции, встроенные в язык, должны нести минимальную нагрузку, необходимую для их применения. Язык C++ работает на принципе «платишь только за то, чем пользуешься», позволяя, например, отказаться от таблицы виртуальных функций при использовании классов [2], если программисту не нужен полиморфизм, а только наследование и инкапсуляция. Язык Rust создаёт новую модель управления памятью, которая предотвращает типичные ошибки в языках с ручной моделью (такие как попытки доступа к преждевременно уничтоженным объектам), при этом не используя сборки мусора или автоматического подсчёта ссылок. Используемая в этих целях абстракция lifetime [3] (время жизни объекта) является zero-cost abstraction (абстракция с нулевой стоимостью), поскольку с её помощью такие моменты, как оптимальное время удаления объекта, полностью рассчитываются на этапе компиляции без влияния на производительность конечной программы, в то же время увеличивая производительность разработки за счёт сокращения этапа отладки.

Как видно по примеру с Rust, последние два пункта тесно связаны с вычислениями на этапе компиляции. Перенос решений как можно большего количества задач с конечной программы на компилятор улучшает производительность как самой программы, так и процесса отладки, потому является перспективным направлением исследований.

2 Структура разрабатываемого компилятора

Общая структура программы описана на рис. 1. Каждый компилятор имеет 3 основные фазы [4]:

1. Разбор исходного кода. Данная фаза состоит из двух процессов: лексический и синтаксический анализ. Лексический анализатор разбивает исходную программу на поток токенов (последовательность идентификаторов, ключевых слов, операторов и констант), после чего синтаксический анализатор проверяет соответствие образуемых конструкций грамматике языка. При отсутствии ошибок на этом этапе возможно формирование однозначной информации об описанных объектах и осуществляемых над ними действиях. Разрабатываемый компилятор использует связку генераторов Flex + GNU Bison для автоматизации разработки этих анализаторов, поскольку они позволяют описать грамматику языка в декларативной форме.

2. Семантический анализ. На этой стадии осуществляются такие операции, как построение промежуточного представления, сверка с таблицей символов, проверка типов, анализ потока управления, предварительная оптимизация и прочее. Основные цели: убедиться, что описанные действия верны в понятиях описанных абстракций, и определить корректные действия в понятиях генерируемой программы. Эти операции производятся с помощью таблицы символов путём анализа деревьев выражений. Каждый описанный тип данных в анализируемой программе сохраняется как объект подклассов Type (рис. 2). Аналогично хранятся инструкции (Statement), выражения (Expression) и прочие элементы.

3. Генерация кода программы. Действия в понятиях программы переводятся в понятия платформы, на которой будет работать целевая программа. На этой фазе могут осуществляться оптимизации, специфичные для данной ОС и/или архитектуры процессора. Традиционные компиляторы генерируют программы в ассемблерных инструкциях целевой платформы, затем используют программу-ассемблер для сборки рабочей программы. Перспективным направлением является компиляция кода в ANSI C, поскольку компиляторы языка C доступны на большом количестве платформ, а их встроенные оптимизаторы позволяют получить производительность, близкую к написанному вручную ассемблерному коду.

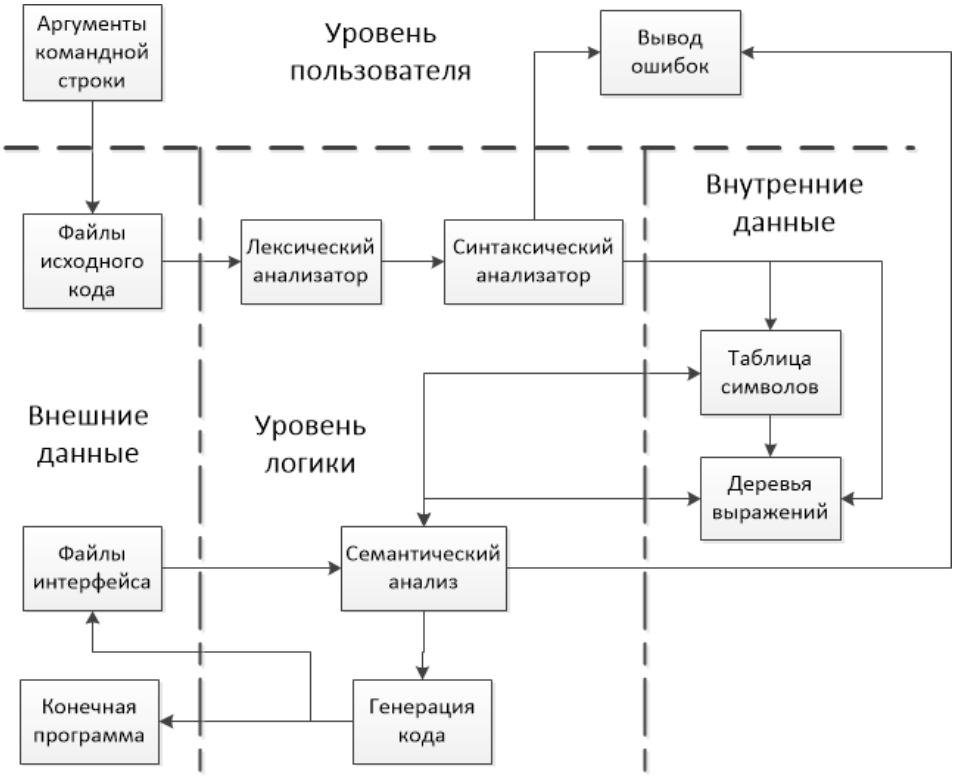

Рисунок 1 – Схема архитектуры компилятора

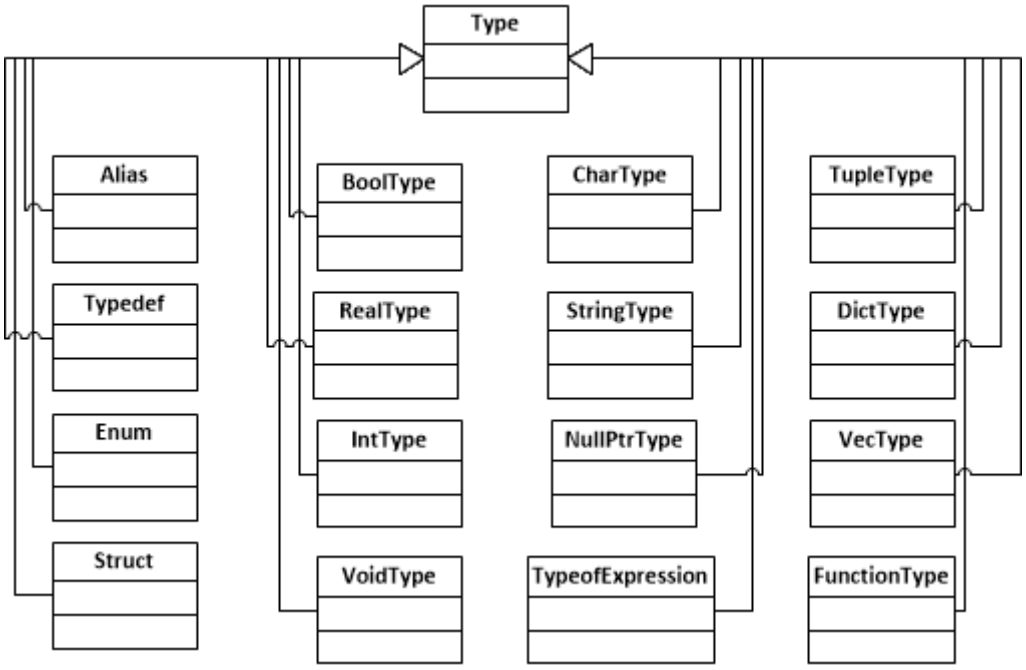

Рисунок 2 – Потомки класса Type, представляющие собой различные стандартные типы и способы построения пользовательских типов

3 Примеры конструкций проектируемого языка

На рис. 3 рассмотрен пример исходного кода. Синтаксис и многие конструкции языка являются производными от С-подобных языков, но с некоторыми структурными отличиями. Из данных примеров видны следующие:

1. Директива import указывает компилятору путь к модулям, символы которых необходимо добавить в глобальное пространство имён. Директива позволяет перечислять символы в иерархическом порядке, таким образом разрешая указать несколько символов в единожды указанном модуле. При импорте компилятор проходит соответствующий путь в файловой системе, пытаясь найти файл интерфейса указанного модуля. Файлы интерфейса генерируются при компиляции вместе с исходным кодом на С.

2. Типы-энумерации (enum) являются алгебраическими типами данных, или помеченными объединениями (type-tagged union). Каждый вариант является не только некоторым значением, но ещё и сопровождающей это значение структурой данных. Все варианты могут иметь параметры разных типов. При использовании объектов таких типов в кратком синтаксисе скрывается множество операций по проверке истинного типа объекта и компилятор гарантирует, что все операции над этими объектами типобезопасны.

3. Условные переходы, циклы и блоки кода вместо инструкций являются выражениями. Т. е. они имеют некоторое значение, вычисляемое кодом внутри них. Это позволяет: записывать некоторые операции более лаконично, сохранять чистоту внешней области видимости от временных переменных и обеспечивать константность переменных, требующих сложный процесс инициализации.

4. Цикл for имеет более простую форму записи, которая указывает имена индексной переменной и итератора, а также диапазон обхода без излишней информации о внутренних условиях работы цикла.

Кроме того, имеются ссылочные типы (в качестве абстракции над указателями) и чёткое разделение между типами nullable (которые могут иметь нулевой указатель в качестве значения) и не-nullable. Компилятор будет требовать проверку на null в случае попыток операций над объектами nullable-типов.

```
// Исходный текст программы начинается с директив import, добавляющих
в таблицу символов определения из других модулей.
import std:(io, stdlib.malloc);

// Определить новый тип на основе одного предыдущего
typedef Integer = Int;

// Определение структуры
struct Point3D {
	x, y, z : Real; // Поля
}

// Энумерация
enum Color {
	Red, Green, Blue, // Обычные варианты
	RGB(red : Int, green : Int, blue : Int) // Параметризованный
вариант
}
// Определение функции
func main(args : Vec[String]) : Int {
	var eight = 3 + 5; // Автоматически принимает тип Int
	var nine : Real = eight + 1; // Преобразуется к Real из Int

	// Конструкция if может использоваться как выражение
	// Доступ к элементам массива происходит через "()"
	// Оператор "." для доступа к свойствам
	var condInt = if (args(1).isEmpty) eight else nine;

	for (i,v in 1..10) { // аналог for (int i=0, v=1; v <= 10; i++,
v++)...
		print(i, " : ", v, "\n");
	};

	var point : Point3D = (3, 4, 0.2);
	var gray = RGB(128, 128, 128); // Тип - Color

	return 0;
}
```

Рисунок 3 – Пример исходного кода на проектируемом языке

4 Перспективы развития

Язык создан с намерением его расширения функциональными и объектно-ориентированными возможностями с целью исследования возможностей вычислений на этапе компиляции.

Среди планируемых возможностей являются такие распространённые возможности современных языков, как интерфейсы, полноценная поддержка шаблонов (в примере кода виден стандартный шаблон Vec, который является абстракцией над статическими и динамическими массивами), lambda-выражения (как минимум в форме inline-операций), контрактное программирование и выполнение произвольного кода на этапе компиляции.

Объектная модель, в которой в основе будут лежать интерфейсы, сходна с моделью языка Rust [5]. В языке Rust отдельно описываются структуры данных (struct), интерфейсы (trait) и реализации (impl), что отличается от традиционного объектно-ориентированного подхода, в котором классы объединяют в себе все три элемента. Возможность описывать реализации различных интерфейсов для различных структур данных открывает возможности для полиморфизма без явных признаков наследования («приоритет композиции вместо наследования» как один из принципов ООП подсвечивает одну из проблем этой парадигмы). При этом возможен больший фокус на статическом полиморфизме вместо динамического (компилятор может «развернуть» абстракцию и подобрать правильные операции без использования таблиц виртуальных функций, и таким образом предотвратить лишние ошибки динамического приведения типов).

Выполнение произвольного кода на этапе компиляции (compile-time function execution, CTFE [6]) также позволит повысить производительность конечной программы и обнаружить многие ошибки до её запуска. Задачи, которые можно выполнить единожды на машине разработчика, можно убрать с каждого запуска программы пользователем. Часть кода, которая требует вызова функций операционной системы для вычисления необходимых для компиляции данных, может быть обозначена специальным атрибутом. В остальных случаях компилятор может использовать анализ потока управления [7], чтобы определить возможность исполнения при компиляции конкретных участков кода. То есть ставится вопрос: требуются ли для исполнения данного кода данные от пользователя или его среды. Если нет, то в конечную программу попадает только результат его вычисления. На деле эта задача сводится к выделению в исходном коде комплекса метапрограмм, иерархически зависящих от выходных данных друг друга (рис. 5). При этом входные данные верхнего уровня прописаны в коде в качестве констант-аргументов для некоторых используемых библиотек и некоторые результаты работы среды разработчика, а выходные данные нижнего уровня напрямую попадают в конечную программу и взаимодействуют с системой пользователя без предварительных инициализаций. Такой подход также позволит очистить исполняемый файл от «мёртвого» кода и даже от «мёртвых» полей данных, которые существуют для поддержки обычно полезной функциональности, но не влияющей на работу данной конкретной программы.

Рисунок 4 – Общая схема работы будущего компилятора с возможностями исполнения кода

Модель состояний типов (typestate) [8] может обеспечить корректность применения последовательностей операций в каждой точке программы на основе описанных условий.

Анализируя различные инструкции и условные переходы, компилятор может предсказать определённые наборы значений, которые переменные могут принять в отдельных ветвях программы, и предотвратить использование ошибочных значений в отдельных операциях либо неверной последовательности действий над некоторым объектом. В сочетании с контрактным программированием она даёт возможность предотвратить многие исключительные ситуации на этапе компиляции, поскольку создатели библиотек смогут установить особые ограничения на входные данные, и компилятор будет вынуждать программистов, использующих их библиотеки, выполнить соответствующие проверки или гарантировать корректность каким-либо другим образом (например, последовательностью преобразований).

Выводы

В ходе выполнения данной работы были изучены основные современные подходы к проектированию языков программирования и рассмотрена модель работы типичного компилятора. С учётом данных принципов был спроектирован С-подобный язык программирования общего назначения и компилятор для него.

Дальнейшее развитие проекта состоит в расширении системы типов и внедрении комплексных средств анализа кода для обеспечения улучшения качества процесса разработки ПО.

Литература

1. Исполнение кода и диагностика ошибок программ на этапе компиляции. Н.М. Ткачёв, А.И. Андрюхин, Компьютерная и программная инженерия - 2015 год: - Донецк: ДонНТУ, 2015. – С.105-107.

2. Interview With Bjarne Stroustrup [электронный ресурс]. – Режим доступа: http://www.stroustrup.com/devXinterview.html.

3. Rust Programming Language. Lifetimes [электронный ресурс]. – Режим доступа: https://doc.rust-lang.org/book/lifetimes.html.

4. В.Э. Карпов "Классическая теория компиляторов", ISBN 5–230–16344–5, 2011. - С. 7-10.

5. Rust Programming Language. Traits [электронный ресурс]. – Режим доступа: https://doc.rust-lang.org/book/traits.html.

6. Wikipedia. Compile-time function execution [электронный ресурс]. – Режим доступа: https://en.wikipedia.org/wiki/Compile_time_function_execution.

7. Википедия. Анализ потока управления [электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/Анализ_потока_управления

8. Wikipedia. Typestate analysis [электронный ресурс]. – Режим доступа: https://en.wikipedia.org/wiki/Typestate_analysis.

Ткачёв Н.М., Андрюхин А.И. Проектирование и разработка компилятора С-подобного языка программирования общего назначения с поддержкой исполнения кода на этапе компиляции. *В статье представлен краткий обзор современных принципов проектирования языков программирования и сформированы требования к проектируемому языку. Спроектирована начальная итерация языка программирования общего назначения и кратко описана структура разрабатываемого компилятора. Перспективами развития языка является расширение системы типов и внедрение в компилятор комплексных средств анализа кода для предотвращения ошибок и исполнения кода на этапе компиляции.*

Ключевые слова: *компилятор, C, Flex, GNU Bison, генерация кода, вычисления на этапе компиляции, система типов.*

Tkachev Nickolay, Andryukhin Alexander Designing a C-like general purpose programming language with CTFE capabilities and building a compiler. *The article provides a brief overview of modern principles of programming language design and forms a set of requirements for a designed language. It showcases initial iteration of designed general purpose language and briefly describes architecture of its compiler. In the future language will have its type system expanded, the compiler will be extended with complex code analysis measures for error prevention and compile-time code execution.*

Key words: *compiler, C, Flex, GNU Bison, code generation, compile-time function execution, type system.*