Памяти переводчика

Семена Викторовича Минца,

просто очень хорошего человека

Эта книга – его последний перевод.

С Семеном было очень приятно работать – он был немногословен и деловит.

Отлично знал информатику и программирование и перевел для нас «Введение в логическое программирование», «Объяснимые модели искусственного интеллекта на Python», «Искусство неизменяемой архитектуры» и эту последнюю.

Он ушел слишком несправедливо рано, мог бы еще многое сделать.

Будет не хватать его. Людей, особенно талантливых, заменить невозможно.

Заместитель главного редактора
Сенченкова Елена

Оглавление

Об авторах 16

О рецензентах 16

Предисловие 17

Для кого эта книга 17
Что скрывает обложка 17
Как получить от этой книги максимальную пользу 20
Загрузка примеров 20
Видео 20
Цветные иллюстрации 20
Используемые сокращения 20
Список опечаток 21
Нарушение авторских прав 21

ЧАСТЬ I. ИНТЕРФЕЙСЫ ЯЗЫКА ПРОГРАММИРОВАНИЯ 23

Глава 1. Зачем создавать еще один язык программирования? 25

Итак, вы хотите создать свой собственный язык программирования 25
Типы реализации языков программирования 26
Организация реализации языка байт-кода 27
Языки, используемые в примерах 28
Язык и библиотека – в чем разница? 29
Применимость к другим задачам разработки программного обеспечения 30
Определение требований к вашему языку 31
Тематическое исследование – требования, которые вдохновили на создание языка Unicon 33
Требование Unicon № 1 – сохранять то, что люди любят в Icon 33
Требование Unicon № 2 – поддержка крупномасштабных программ, работающих с большими данными 34
Требование Unicon № 3 – высокоуровневый ввод/вывод для современных приложений 34
Требование Unicon № 4 – обеспечить универсально реализуемые системные интерфейсы 35
Заключение 35
Вопросы 36

Глава 2. Дизайн языка программирования 37

Определение видов слов и пунктуации в вашем языке 38
Определение потока управления 40
Решение о том, какие типы данных поддерживать 41
Атомарные типы 41
Составные типы 42
Типы, специфичные для конкретной области 44
Общая структура программы 44
Завершение определения языка Jzero 45
Тематическое исследование – проектирование графических объектов в Unicon 46
Поддержка языка для графики 2D 47
Добавление поддержки трехмерной графики 49
Заключение 50
Вопросы 50

Глава 3. Сканирование исходного кода 52

Технические требования 52
Лексемы, лексические категории и токены 53
Регулярные выражения 54
Правила регулярных выражений 54
Примеры регулярных выражений 56
Использование UFlex и JFlex 57
Раздел заголовка 58
Раздел регулярных выражений 58
Написание простого сканера исходного кода 59
Запуск сканера 62
Токены и лексические атрибуты 63
Расширение нашего примера для построения токенов 64
Написание сканера для Jzero 66
Спецификация Jzero flex 66
Код Unicon Jzero 69
Код Java Jzero 72
Запуск сканера Jzero 75
Регулярных выражений не всегда достаточно 76
Заключение 80
Вопросы 80

Глава 4. Парсинг 81

Технические требования 81
Анализ синтаксиса 82
Понимание бесконтекстных грамматик 83
Написание правил бесконтекстной грамматики 84
Написание правил для программных конструкций 85
Использование iyacc и BYACC/J 87
Объявление символов в разделе заголовка 88

Составление раздела бесконтекстной грамматики *yacc* 89
Понимание парсеров *yacc* 90
Устранение конфликтов в парсерах *yacc* 92
Исправление синтаксических ошибок 93
Создание игрушечного примера 93
Написание парсера для Jzero 98
Спецификация Jzero lex 98
Спецификация yacc в Jzero 98
Код Unicon Jzero 103
Код парсера Jzero на языке Java 105
Запуск парсера Jzero 105
Улучшение сообщений об ошибках синтаксиса 107
Добавление деталей в сообщения Unicon об ошибках синтаксиса 108
Добавление деталей в сообщения Java об ошибках синтаксиса 108
Использование Merr для создания лучших сообщений об ошибках синтаксиса 109
Заключение 110
Вопросы 110

Глава 5. Деревья синтаксиса 111

Технические требования 111
Использование GNU make 112
Изучение деревьев 115
Определение типа дерева синтаксиса 115
Деревья разбора в сравнении с деревьями синтаксиса 117
Создание листьев из терминальных символов 119
Обертывание токенов в листья 120
Работа со стеком значений YACC 120
Обертка листьев для стека значений парсера 122
Определение нужных вам листьев 123
Построение внутренних узлов из правил производства 124
Доступ к узлам дерева в стеке значений 124
Использование фабричного метода узла дерева 126
Формирование деревьев синтаксиса для языка Jzero 127
Отладка и тестирование вашего дерева синтаксиса 134
Предотвращение распространенных ошибок в дереве синтаксиса 134
Распечатка вашего дерева в текстовом формате 136
Печать дерева с помощью dot 138
Заключение 143
Вопросы 143

ЧАСТЬ II. ОБХОДЫ ДЕРЕВА СИНТАКСИСА 145

Глава 6. Таблицы символов 147

Технические требования 148
Создание основы для таблиц символов 148
Объявления и области видимости 148

Присваивание и разыменование переменных 149
Выбор подходящего обхода дерева для работы 150
Создание и заполнение таблиц символов для каждой области видимости 151
Добавление семантических атрибутов к деревьям синтаксиса 152
Определение классов для таблиц символов и записей в таблицах символов 154
Создание таблиц символов 155
Заполнение таблиц символов 157
Синтез атрибута isConst 159
Проверка наличия необъявленных переменных 160
Идентификация тел методов 160
Выявление использования переменных в теле метода 161
Поиск повторно объявленных переменных 162
Вставка символов в таблицу символов 163
Сообщение о семантических ошибках 163
Обработка пакетов и областей видимости классов в Unicon 164
Искажение имен 165
Вставка self для ссылок на переменные-члены 166
Вставка self в качестве первого параметра в вызовы методов 166
Тестирование и отладка таблиц символов 167
Заключение 169
Вопросы 170

Глава 7. Проверка базовых типов 171

Технические требования 171
Представление типов в компиляторе 171
Определение базового класса для представления типов 172
Подклассификация базового класса для сложных типов 173
Присвоение информации о типе объявленным переменным 175
Синтез типов из зарезервированных слов 177
Наследование типов в списке переменных 178
Определение типа в каждом узле дерева синтаксиса 179
Определение типа в листьях 180
Вычисление и проверка типов во внутренних узлах 182
Проверка типов во время выполнения и вывод типов в Unicon 186
Заключение 188
Вопросы 188

Глава 8. Проверка типов в массивах, вызовах методов и доступах к структурам 189

Технические требования 189
Операции проверки типов массивов 189
Управление объявлениями переменных в массивах 190
Проверка типов при создании массива 191
Проверка типов при обращении к массиву 193
Проверка вызовов методов 194

Вычисление параметров и информации о возвращаемом типе ... 194
Проверка типов в каждом месте вызова метода ... 197
Проверка типов в операторах возврата ... 200
Проверка обращений к структурированным типам ... 202
Обработка объявлений переменных экземпляра ... 202
Проверка типов при создании экземпляра ... 203
Проверка типов при обращении к экземпляру ... 205
Заключение ... 208
Вопросы ... 209

Глава 9. Генерация промежуточного кода ... 210

Технические требования ... 210
Подготовка к генерации кода ... 210
Зачем генерировать промежуточный код? ... 211
Изучение областей памяти в созданной программе ... 211
Представление типов данных для промежуточного кода ... 212
Добавление атрибутов промежуточного кода в дерево ... 214
Генерация меток и временных переменных ... 215
Набор инструкций промежуточного кода ... 218
Инструкции ... 218
Декларации ... 219
Аннотирование деревьев синтаксиса метками для потока управления ... 219
Генерация кода для выражений ... 222
Генерация кода для потока управления ... 225
Генерация целевых меток для выражений условий ... 225
Генерация кода для циклов ... 228
Генерация промежуточного кода для вызовов методов ... 229
Проверка сгенерированного промежуточного кода ... 231
Заключение ... 232

Глава 10. Раскраска синтаксиса в IDE ... 233

Загрузка примеров IDE, используемых в этой главе ... 234
Интеграция компилятора в редактор программиста ... 236
Анализ исходного кода из среды IDE ... 236
Отправка выходных данных компилятора в IDE ... 237
Предотвращение повторного разбора всего файла при каждом изменении ... 238
Использование лексической информации для раскрашивания токенов ... 242
Расширение компонента EditableTextList для поддержки цвета ... 242
Раскрашивание отдельных токенов по мере их создания ... 242
Подсветка ошибок с использованием результатов разбора ... 243
Добавление поддержки Java ... 245
Заключение ... 247

ЧАСТЬ III. ГЕНЕРАЦИЯ КОДА И СРЕДЫ ВЫПОЛНЕНИЯ 249

Глава 11. Интерпретаторы байт-кода 251

Технические требования 251
Понимание, что такое байт-код 252
Сравнение байт-кода с промежуточным кодом 253
Построение набора инструкций байт-кода для Jzero 255
Определение формата файла байт-кода Jzero 255
Понимание основ работы стековой машины 258
Реализация интерпретатора байт-кода 259
Загрузка байт-кода в память 259
Инициализация состояния интерпретатора 261
Выборка инструкций и продвижение указателя инструкции 263
Декодирование инструкций 264
Выполнение инструкций 265
Запуск интерпретатора Jzero 268
Написание среды выполнения для Jzero 269
Запуск программы Jzero 269
Изучение iconx, интерпретатора байт-кода Unicon 270
Понимание целенаправленного байт-кода 271
Сохранение информации о типе во время выполнения 271
Выборка, декодирование и выполнение инструкций 272
Создание остальной части среды выполнения 272
Заключение 273
Вопросы 273

Глава 12. Генерация байт-кода 274

Технические требования 274
Преобразование промежуточного кода в байт-код Jzero 275
Добавление класса для инструкций байт-кода 276
Соответствие адресов промежуточного кода адресам байт-кода 276
Реализация метода генератора байт-кода 278
Генерация байт-кода для простых выражений 278
Генерация кода для обработки указателей 280
Генерация байт-кода для безусловных и условных переходов 281
Генерация кода для вызовов методов и возвратов 282
Обработка меток и других псевдоинструкций промежуточного кода 284
Сравнение ассемблера байт-кода с двоичными форматами 285
Вывод байт-кода в формате ассемблера 285
Вывод байт-кода в двоичном формате 287
Линковка, загрузка и включение среды выполнения 288
Пример Unicon – генерация байт-кода в icont 288
Заключение 290
Вопросы 290

Глава 13. Генерация собственного кода 292

Технические требования 292
Принятие решения о генерации собственного кода 292
Знакомство с набором инструкций x64 293
Добавление класса для инструкций x64 294
Соответствие областей памяти регистровым режимам адресации x64 294
Использование регистров 295
Начинаем с нулевой стратегии 296
Преобразование промежуточного кода в код x64 299
Соответствие адресов промежуточного кода местоположению в x64 300
Реализация метода генератора кода x64 303
Генерация кода x64 для простых выражений 304
Генерация кода для обработки указателей 305
Генерация собственного кода для безусловных и условных переходов 306
Генерация кода для вызовов методов и возвратов 307
Обработка меток и псевдоинструкций 309
Генерация выходных данных x64 311
Запись кода x64 в формате ассемблера 311
Переход от ассемблера к объектному файлу 312
Линковка, загрузка и включение среды выполнения 313
Заключение 314
Вопросы 315

Глава 14. Реализация операторов и встроенных функций 316

Реализация операторов 316
Подразумевают ли операторы аппаратную поддержку, и наоборот 317
Добавление конкатенации строк в генерацию промежуточного кода 318
Добавление конкатенации строк в интерпретатор байт-кода 319
Добавление конкатенации строк в собственную среду выполнения 322
Написание встроенных функций 323
Добавление встроенных функций в интерпретатор байт-кода 323
Написание встроенных функций для использования в реализации собственного кода 324
Интеграция встроенных функций со структурами управления 325
Разработка операторов и функций для Unicon 326
Написание операторов в Unicon 327
Разработка встроенных функций Unicon 329
Заключение 330
Вопросы 330

Глава 15. Структуры управления доменами 331
Понимание необходимости новой структуры управления 331
Определение структуры управления 332
Устранение избыточных параметров 333
Сканирование строк в Icon и Unicon 333
Среды сканирования и их примитивные операции 334
Устранение избыточных параметров с помощью структуры управления 336
Рендеринг областей в Unicon 337
Отображение 3D-графики из списка отображения 337
Указание областей рендеринга с помощью встроенных функций 338
Изменение графических уровней детализации с помощью вложенного рендеринга областей 339
Создание структуры управления рендерингом областей 340
Добавление зарезервированного слова для рендеринга областей 340
Добавление правила грамматики 341
Проверка wsection на семантические ошибки 342
Генерация кода для структуры управления wsection 343
Заключение 345
Вопросы 345

Глава 16. Сборка мусора 347
Оценка важности сборки мусора 347
Подсчет ссылок на объекты 349
Добавление подсчета ссылок в Jzero 350
Генерация кода для распределения кучи 350
Изменение сгенерированного кода для оператора присваивания 352
Учет недостатков и ограничений, связанных с подсчетом ссылок 353
Пометка реальных данных и очистка остальных 354
Организация областей памяти кучи 355
Обход базиса для пометки живых данных 357
Восстановление живой памяти и размещение ее в непрерывных фрагментах 361
Заключение 363
Вопросы 364

Глава 17. Заключительные размышления 365
Размышления о том, что изучено при написании этой книги 365
Решение о том, куда двигаться дальше 366
Изучение дизайна языков программирования 366
Изучение реализации интерпретаторов и машин байт-кода 367
Приобретение опыта в оптимизации кода 368
Мониторинг и отладка выполнения программ 369
Проектирование и реализация IDE и построителей GUI 369
Изучение ссылок для дальнейшего чтения 370

Изучение дизайна языков программирования ... 370
Изучение реализации интерпретаторов и машин байт-кода ... 371
Приобретение опыта работы с собственным кодом и оптимизации кода ... 371
Мониторинг и отладка выполнения программ ... 372
Проектирование и реализация IDE и построителей GUI ... 372
Заключение ... 373

ЧАСТЬ IV. ПРИЛОЖЕНИЕ ... 375

Приложение. Основы Unicon ... 377

Запуск Unicon ... 377
Использование объявлений и типов данных Unicon ... 379
Объявление различных типов компонентов программы ... 379
Использование атомарных типов данных ... 381
Организация нескольких значений с помощью структурных типов ... 382
Оценка выражений ... 384
Формирование базовых выражений с помощью операторов ... 384
Вызов процедур, функций и методов ... 387
Итерации и выбор того, что и как выполнять ... 388
Генераторы ... 389
Отладка и вопросы окружения ... 390
Изучение основ отладчика UDB ... 390
Переменные окружения ... 391
Препроцессор ... 391
Мини-справочник функций ... 393
Избранные ключевые слова ... 398

Оценки ... 400

Глава 1 ... 400
Глава 2 ... 400
Глава 3 ... 401
Глава 4 ... 401
Глава 5 ... 402
Глава 6 ... 402
Глава 7 ... 403
Глава 8 ... 403
Глава 11 ... 404
Глава 12 ... 404
Глава 13 ... 405
Глава 14 ... 405
Глава 16 ... 407

Эта книга посвящается Сьюзи, Кертису, Кэри и всем, кто создает свои собственные языки программирования.

Клинтон Л. Джеффери

Об авторах

Клинтон Л. Джеффри – профессор и заведующий кафедрой компьютерных наук и инженерии Горно-технологического института Нью-Мексико. Он получил степень бакалавра в Вашингтонском университете, а также степень магистра и доктора философии в Университете Аризоны в области компьютерных наук. Проводил исследования и написал много книг и статей по языкам программирования, мониторингу программ, отладке, графике, виртуальным средам и визуализации. Вместе с коллегами изобрел язык программирования Unicon.

О рецензентах

Филлип Ли – доброволец Корпуса мира в Сараваке, Малайзия. Он получил степень бакалавра в Университете штата Орегон, магистра, докторскую степень в Университете Вашингтона, степень магистра в области малайской/индонезийской литературы в Университете Малайзии и степень магистра в области вычислительной техники в Университете Мердока в Перте. Преподавал для студентов и аспирантов в Оклендском университете и Университете Мердока. У Филиппа есть публикации по латинской, греческой, малайской и индонезийской литературе. Он является сопрограммистом библиотеки Конгресса *thomas.loc.gov*, поисковой системы Конгресса Национальной медицинской библиотеки *toxnet.nlm.nih.gov*. Кроме того, трудится разработчиком программ анализа текста для англо-иранского словаря Фонда Тун Джуга.

Стив Уамплер получил степень доктора философии в области компьютерных наук в Университете Аризоны. После чего он был адъюнкт-профессором компьютерных наук с 1981 по 1993 год. Стив работал разработчиком программного обеспечения в нескольких крупных проектах телескопов, включая проект Gemini 8m Telescopes Project и Солнечный телескоп Daniel K Inouye, в рамках Ассоциации исследований в области астрономии. Наряду с этим он был рецензентом программного обеспечения для ряда крупных телескопов, в том числе LSST, TMT, GMT, Keck, VLT ESO и GTC. Стив был техническим рецензентом первого издания книги Марка Собелла «Практическое руководство по операционной системе Linux», 1997 год.

Предисловие

После 60 лет высокоуровневой разработки языков программирование все еще остается сложным. Спрос на программное обеспечение постоянно увеличивающегося объема и сложности реализации резко возрос из-за аппаратных достижений, в то время как языки программирования совершенствуются гораздо медленнее. Создание новых языков для конкретных целей – одно из противоядий от кризиса программного обеспечения.

Эта книга посвящена созданию новых языков программирования. Вводится тема проектирования языка программирования, хотя основной акцент делается на реализации языка программирования. В рамках этой интенсивно изучаемой темы новым аспектом данной книги является слияние традиционных инструментов компиляции (Flex и Byacc) с двумя языками реализации более высокого уровня. Язык очень высокого уровня (Unicon) обрабатывает структуры данных и алгоритмы компилятора, как нож масло, в то время как основной современный язык (Java) показывает, как реализовать тот же код в более типичной производственной среде.

Для кого эта книга

Эта книга предназначена для разработчиков программного обеспечения, заинтересованных в идее создания собственного языка или разработки языка, специфичного для конкретной предметной области. Студенты, изучающие информатику на курсах построения компиляторов, также найдут эту книгу весьма полезной в качестве практического руководства по реализации языка в дополнение к другим теоретическим учебникам. Чтобы извлечь максимальную пользу из данной книги, требуются знания среднего уровня и опыт работы с языком высокого уровня, таким как Java или C++.

Что скрывает обложка

В *главе 1 «Зачем создавать другой язык программирования?»* обсуждается, когда следует создавать язык программирования, а когда вместо этого создавать библиотеку функций или библиотеку классов. Многие читатели этой книги уже знают, что они хотят создать свой собственный язык программирования. Некоторые должны вместо этого создать библиотеку.

Глава 2 «Проектирование языка программирования» описывает, как точно определить язык программирования, что важно знать, прежде чем пытаться создать язык программирования. Это включает в себя разработку лексических и синтаксических особенностей языка, а также его семантики. Хорошие языковые проекты обычно используют как можно больше знакомого синтаксиса.

Глава 3 «Сканирование исходного кода» представляет лексический анализ, включая регулярные обозначения выражений и инструменты Ulex и JFlex.

В конце вы будете открывать файлы исходного кода, читать их символ за символом и сообщать об их содержимом в виде потока токенов, состоящих из отдельных слов, операторов и знаков препинания в исходном файле.

В *главе 4 «Синтаксический анализ»* представлен синтаксический анализ, включая контекстно-свободные грамматики и инструменты iyacc и byacc/j. Вы узнаете, как отлаживать проблемы в грамматиках, которые препятствуют синтаксическому анализу, и сообщать о синтаксических ошибках, когда они возникают.

В *главе 5 «Синтаксические деревья»* рассматриваются синтаксические деревья. Основным побочным продуктом процесса синтаксического анализа является построение древовидной структуры данных, которая представляет логическую структуру исходного кода. Построение узлов дерева происходит в семантических действиях, которые выполняются для каждого правила грамматики.

В *главе 6 «Таблицы символов»* показано, как создавать таблицы символов, вставлять в них символы и использовать таблицы для выявления двух видов семантических ошибок: необъявленных и незаконно повторно объявленных переменных. Чтобы понять ссылки на переменные в исполняемом коде, необходимо отслеживать область действия и время жизни каждой переменной. Это достигается с помощью табличных структур данных, которые являются вспомогательными для синтаксического дерева.

Глава 7 «Проверка базовых типов» посвящена проверке типов, которая является основной задачей, требуемой в большинстве языков программирования. Проверка типов может выполняться во время компиляции или во время выполнения. В этой главе рассматривается общий случай статической проверки типов во время компиляции для базовых типов, также называемых атомарными, или скалярными, типами.

В *главе 8 «Проверка типов массивов, вызовов методов и доступа к структурам»* показано, как выполнять проверку типов массивов, параметров и возвращаемых типов вызовов методов в подмножестве Java Jzero. Более сложные части проверки типов – это когда должны быть проверены несколько массовов или составные массивы.

Глава 9 «Генерация промежуточного кода» показывает вам, как генерировать промежуточный код, рассматривая примеры для языка Jzero. Прежде чем сгенерировать код для выполнения, большинство компиляторов превращают синтаксическое дерево в список машинно независимых инструкций промежуточного кода. На этом этапе обрабатываются ключевые аспекты потока управления, такие как генерация меток и инструкции goto.

В *главе 10 «Раскрашивание синтаксиса в среде IDE»* рассматривается задача включения информации из синтаксического анализа в среду IDE, чтобы обеспечить раскрашивание синтаксиса и визуальную обратную связь о синтаксических ошибках. Язык программирования требует большего, чем просто компилятор или интерпретатор, – он требует экосистемы инструментов для разработчиков. Эта экосистема может включать в себя отладчики, интерактивную справку или интегрированную среду разработки. Эта глава представляет собой пример Unicon, взятый из среды разработки Unicon IDE.

Глава 11 «Интерпретаторы байт-кода» посвящена разработке набора команд и интерпретатора, который выполняет байт-код. Новый язык, специфич-

ный для конкретной предметной области, может включать в себя высокоуровневые функции программирования предметной области, которые напрямую не поддерживаются основными процессорами. Наиболее практичным способом генерации кода для многих языков является генерация байт-кода для абстрактной машины, набор команд которой напрямую поддерживает целевое назначение языка с последующим выполнением этой программы путем интерпретации команд.

Глава 12 «Генерация байт-кода» рассматривает прохождение по гигантскому связанному списку, перевод каждой инструкции промежуточного кода в одну или несколько инструкций байт-кода. Как правило, это цикл для обхода связанного списка с разным фрагментом кода для каждой промежуточной кодовой инструкции.

Глава 13 «Генерация собственного кода» содержит обзор генерации собственного кода для x86_64. Некоторые языки программирования требуют собственного кода для достижения своих требований к производительности. Генерация собственного кода похожа на генерацию байт-кода, но более сложна, включает в себя выделение регистров и режимы адресации памяти.

Глава 14 «Реализация операторов и встроенных функций» описывает, как поддерживать языковые функции очень высокого уровня и специфичные для предметной области, добавляя операторы и функции, встроенные в язык. Языковые возможности очень высокого уровня и специфичные для предметной области часто лучше всего представлены операторами и функциями, встроенными в язык, а не библиотечными функциями. Добавление встроенных модулей может упростить ваш язык, улучшить его.

Глава 15 «Структуры управления доменом» описывает, когда вам нужна новая структура управления, и предоставляет примеры структур управления, которые обрабатывают текст с помощью сканирования строк и отображают графические области. Общий код в предыдущих главах охватывал основные условные и циклические структуры управления, но языки, зависящие от предметной области, часто имеют уникальную или настраиваемую семантику, для которой они вводят новые структуры управления. Добавление новых структур управления существенно сложнее, чем добавление новой функции или оператора, но именно это делает языки, специфичные для предметной области, достойными разработки, а не просто написания библиотек классов.

В *главе 16 «Сборка мусора»* представлена пара методов, с помощью которых вы можете реализовать сборку мусора на вашем языке. Управление памятью является одним из наиболее важных аспектов современных языков программирования, и все классные языки программирования имеют функцию автоматического управления памятью с помощью сборки мусора. В этой главе приводится несколько вариантов того, как вы могли бы реализовать сборку мусора на вашем языке, включая подсчет ссылок и сборку мусора с пометкой и разверткой.

Глава 17 «Заключительные мысли» отражает основные темы, представленные в книге, и дает вам некоторую пищу для размышлений. В ней рассматривается то, что было извлечено из написания этой книги, и дается множество рекомендаций для дальнейшего чтения.

Приложение «Unicon Essentials» описывает язык программирования Unicon в достаточном количестве, чтобы понять те примеры в этой книге, которые

находятся в Unicon. Большинство примеров приведены рядом на Unicon и Java, но версии Unicon обычно короче и легче читаются.

Как получить от этой книги максимальную пользу

Чтобы понять эту книгу, вы должны быть программистом среднего уровня на Java или подобном языке; программист C, который знает объектно-ориентированный язык, подойдет.

Программное обеспечение, упомянутое в книге	Необходимая операционная система
Unicon 13.2, Uflex,	Windows, Linux
Java, Jflex, and Byacc/J	
GNU Make	

Инструкции по установке и использованию инструментов немного расширены, чтобы сократить время запуска, и приведены в главе 3 «Сканирование исходного кода» и главе 5 «Синтаксические деревья». Если вы технически одарены, вы, возможно, сможете запустить все эти инструменты на macOS, но во время написания этой книги они не использовались и не тестировались.

> **Примечание**
>
> Если вы используете цифровую версию этой книги, мы советуем вам ввести код самостоятельно или получить доступ к коду из репозитория книги на GitHub (ссылка доступна в следующем разделе). Это поможет вам избежать любых потенциальных ошибок, связанных с копированием и вставкой кода.

Загрузка примеров

Примеры для данной книге вы можете загрузить на сайте нашего издательсва по ссылке: (_____страница книги)

Видео

Видеоролики с кодом в действии для этой книги можно посмотреть по адресу *https://bit.ly/3njc15D*.

Цветные иллюстрации

Цветные иллюстрации к данной книге вы можете загрузить на сайте нашего издательсва по ссылке: (_____страница книги).

Используемые сокращения

Вот несколько текстовых условных обозначений, используемых на протяжении всей этой книги.

Код в тексте: указывает кодовые слова в тексте, имена таблиц базы данных, имена папок, имена файлов, расширения файлов, пути, фиктивные URL-адреса, вводимые пользователем, и дескрипторы Twitter. Вот пример: «Соответствующий Java main() должен быть помещен в класс».

Блок кода задается следующим образом:

```
procedure main(argv)
procedure main(argv)
 yyin := open(argv[1])
 yyin := open(argv[1]) yyin := open(argv[1])
 yyin := open(argv[1])
 yyin := open(argv[1])
```

Когда мы хотим привлечь ваше внимание к определенной части блока кода, соответствующие строки или элементы выделяются жирным шрифтом:

```
procedure main(argv)
procedure main(argv)
 yyin := open(argv[1])
 yyin := open(argv[1]) yyin := open(argv[1])
 yyin := open(argv[1])
```

Ввод или вывод из командной строки записывается так:

```
procedure main(argv)
 yyin := open(argv[1])
```

Жирный шрифт: обозначает новый термин, важное слово или слова, которые вы видите на экране. Например, слова в меню или диалоговых окнах выделены **жирным** шрифтом. Вот пример: «Выберите **Информацию о системе** на панели администрирования».

> **Советы или важные примечания**
>
> Представляют собой текст, помещенный в рамку.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в тексте или в коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии данной книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу *dmkpress@gmail.com*, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Packt очень серьезно относятся к вопросам защиты ав-

торских прав и лицензирования. Если вы столкнетесь в интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции.

Пожалуйста, свяжитесь с нами по адресу электронной почты *dmkpress@gmail.com* со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Часть I

Интерфейсы языка программирования

В этой части вы создадите базовую конструкцию языка и реализуете интерфейс компилятора для него, включая лексический анализатор и синтаксический анализатор, который строит синтаксическое дерево из входного исходного файла.

Эта часть включает в себя следующие главы:

- глава 1 «Зачем создавать еще один язык программирования»;
- глава 2 «Проектирование языка программирования»;
- глава 3 «Сканирование исходного кода»;
- глава 4 «Парсинг»;
- глава 5 «Деревья синтаксиса».

Глава 1

Зачем создавать еще один язык программирования?

Эта книга покажет вам, как создать свой собственный язык программирования, но сначала вы должны спросить себя: зачем мне это нужно? Для некоторых из вас ответ будет простым – потому что это очень весело. Однако для остальных создание языка программирования – это большой труд, и вы должны быть уверены в необходимости этого, прежде чем начинать. Хватит ли у вас терпения и настойчивости?

В этой главе приведены несколько веских причин для создания собственного языка программирования, а также некоторые ситуации, когда вам не обязательно создавать задуманный язык. В конце концов, разработка библиотеки классов для вашей области применения может быть более простой и столь же эффективной. Однако у библиотек есть свои недостатки, и иногда подходит только новый язык.

После этой главы, в остальной части этой книги, тщательно все обдумав, вы примете как должное, что решили создать язык. В этом случае вы должны определить некоторые требования к вашему языку. Но сначала мы рассмотрим следующие основные темы данной главы:

- мотивы для написания собственного языка программирования;
- различие между языками программирования и библиотеками;
- применимость инструментов языка программирования к другим программным проектам;
- определение требований к языку;
- тематическое исследование, в котором обсуждаются требования к языку Unicon.

Давайте начнем с мотивации.

Итак, вы хотите создать свой собственный язык программирования...

Конечно, некоторые изобретатели языков программирования – это рок-звезды компьютерной науки, такие как Деннис Ричи (Dennis Ritchie) или Гвидо ван Россум (Guido van Rossum)! Но стать рок-звездой компьютерной науки было проще в те времена. Давным-давно я услышал следующее сообщение от од-

ного из участников второй конференции по истории языков программирования: «*Все сошлись во мнении, что область языков программирования мертва. Все важные языки уже изобретены*». Это было признано в корне ошибочным через год или два, когда на сцену вышел Java, и, возможно, десятки раз с тех пор, как появились такие языки, как Go. Спустя всего шесть десятилетий было бы неразумно утверждать, что наша область является зрелой и что нет ничего нового, что можно было бы изобрести, чтобы стать знаменитым.

Тем не менее знаменитость – это плохая причина для создания языка программирования. Шансы приобрести славу или богатство благодаря изобретению языка программирования невелики. Любопытство и желание узнать, как все устроено, являются вескими причинами, если у вас есть время и склонность, но, возможно, лучшими причинами для создания собственного языка программирования являются необходимость и потребность.

Некоторым необходимо создать новый язык или новую реализацию существующего языка программирования для нового процессора или для соперничества с конкурирующей компанией. Если это не вы, то, возможно, вы изучили лучшие языки (и компиляторы или интерпретаторы), доступные для какой-то области, в которой вы разрабатываете программы, и для того, что вы делаете, не хватает некоторых ключевых функций, и эти недостающие функции причиняют вам боль. Время от времени кто-то придумывает совершенно новый стиль вычислений, для которого в соответствии с новой парадигмой программирования требуется новый язык.

Пока мы обсуждаем ваши мотивы для создания языка, давайте поговорим о различных видах языков, организации и примерах, которые будут использованы в этой книге. Каждая из данных тем заслуживает внимания.

Типы реализации языков программирования

Какими бы ни были ваши причины, прежде чем создавать язык программирования, вы должны выбрать лучшие инструменты и технологии, которые можно найти для выполнения этой работы. Эта книга подберет их для вас. Во-первых, возникает вопрос о языке реализации, на котором вы создаете свой язык. Академики, изучающие языки программирования, любят хвастаться тем, что пишут свой язык на этом самом языке, но обычно это лишь полуправда (или кто-то был очень непрактичным и в то же время выпендривался). Существует также вопрос о том, какую реализацию языка программирования строить:

- чистый *интерпретатор*, который сам выполняет исходный код;
- *собственный компилятор* и среду выполнения, как, например, в C;
- *транспайлер*, который переводит ваш язык на другой язык высокого уровня;
- *компилятор байт-кода* с сопутствующей машиной байт-кода, например Java.

Первый вариант веселый, но обычно слишком медленный. Второй вариант – самый лучший, но обычно он слишком трудоемкий. Хороший собственный компилятор может потребовать многих человеко-лет усилий.

Хотя третий вариант, безусловно, самый простой и, вероятно, самый веселый, и я уже с успехом использовал его, но если это не прототип, то это своего

рода обман. Конечно, первая версия C++ была транспайлером, но она уступила место компиляторам, и не только потому, что была глючной. Странно, но генерация высокоуровневого кода, кажется, делает ваш язык еще более зависимым от базового языка, чем другие варианты, а языки – это движущиеся мишени. Хорошие языки умерли, потому что их базовые зависимости исчезли или непоправимо отказали. Это может быть смерть от тысячи мелких порезов.

В этой книге выбран четвертый вариант: мы построим компилятор байт-кода с сопутствующей машиной байт-кода, потому что это обеспечивает наибольшую гибкость и притом обладает достойной производительностью. Глава о компиляции собственного кода включена для тех, кому требуется максимально быстрое выполнение.

Понятие машины байт-кода очень старое. Оно стало известным благодаря реализации Pascal в Калифорнийском университете в Сан-Диего (University of California, San Diego – UCSD) и классической реализации SmallTalk-80[1] среди прочих. Оно стало повсеместным и вошло в повседневный английский язык с появлением JVM в Java. Машины байт-кода – это абстрактные процессоры, интерпретируемые программным обеспечением, их часто называют *виртуальными машинами* (как в *Java Virtual Machine*), хотя я не буду использовать данную терминологию, потому что она применяется для обозначения программных средств, использующих реальные наборы инструкций аппаратного обеспечения, таких, например, как классические платформы IBM или более современные инструменты, допустим *Virtual Box*.

Машина байт-кода обычно имеет более высокий уровень, чем часть аппаратного обеспечения, поэтому реализация байт-кода обеспечивает большую гибкость. Давайте вкратце рассмотрим, что для этого нужно...

Организация реализации языка байт-кода

В значительной степени организация этой книги соответствует классической организации компилятора байт-кода и соответствующей виртуальной машины. Эти компоненты определены ниже, а затем приведена диаграмма для их обобщения:

- *лексический анализатор* (сканер) считывает символы исходного кода и определяет, как они сгруппированы в последовательность слов или токенов;
- *синтаксический анализатор* (парсер) считывает последовательность токенов и определяет, является ли эта последовательность допустимой в соответствии с грамматикой языка. Если токены расположены в законном порядке, то получается дерево синтаксиса;
- *семантический анализатор* проверяет, что все используемые имена являются законными для тех операций, в которых они используются. Он проверяет их типы, чтобы точно определить, какие операции выпол-

[1] Smalltalk-80 – объектно-ориентированный язык программирования, разработанный в 1970-х годах. Представляет собой интегрированную среду разработки и исполнения, программирование в которой в итоге сводится к модификации ее собственного поведения. – *Прим. перев.*

няются. Все эти проверки делают дерево синтаксиса толстым, отягощенным дополнительной информацией о том, где объявлены переменные и каковы их типы;

- *генератор промежуточного кода* определяет места в памяти для всех переменных и всех мест, где ход выполнения программы может резко измениться, например циклов и вызовов функций. Он добавляет их в дерево синтаксиса, а затем проходит по этому еще более толстому дереву, прежде чем построить список машинно независимых инструкций промежуточного кода;
- *генератор окончательного кода* превращает список инструкций промежуточного кода в фактический байт-код в формате файла, который будет эффективно загружаться и выполняться.

Для загрузки и выполнения программ независимо от шагов компилятора виртуальной машины байт-кода написан интерпретатор байт-кода. Он представляет собой гигантский цикл с оператором `switch`. Для экзотических языков программирования компилятор может не иметь большого значения, и вся магия будет происходить в интерпретаторе байт-кода. Всю организацию можно обобщить схемой, показанной на рис. 1.1.

Потребуется много кода, чтобы проиллюстрировать построение машинной реализации байт-кода языка программирования. То, как представлен этот код, важно, и много информации вы можете почерпнуть как раз из нашей книги.

Языки, используемые в примерах

В этой книге приводятся примеры кода на двух языках с использованием модели параллельных переводов. Первый язык – *Java*, потому что этот язык повсеместно распространен. Надеемся, что вы знаете его или *C++* и сможете прочитать примеры со средним уровнем владения. Второй язык примеров – это собственный язык автора, *Unicon*. Читая эту книгу, вы сможете сами оценить, какой язык лучше подходит для создания вашего собственного языка программирования. Как можно больше примеров будет приведено на обоих языках, и примеры на двух языках будут написаны максимально одинаково. Иногда предпочтение будет отдано меньшему языку.

Различия между Java и Unicon будут очевидны, но они несколько уменьшаются благодаря инструментам построения компилятора, которые мы будем использовать. А использовать мы будем современных потомков старинных инструментов Lex и YACC для генерации нашего сканера и парсера и придерживаться инструментов для Java и Unicon, которые остаются максимально совместимыми с оригинальными Lex и YACC. В результате фронтенды нашего компилятора будут практически идентичны в обоих языках. Lex и YACC – это языки декларативного программирования, которые решают некоторые из наших трудных проблем на еще более высоком уровне, чем Java или Unicon.

Пока мы применяем Java и Unicon в качестве языков реализации, нам придется поговорить еще об одном языке – о языке примеров, который мы создаем. Он является заменой для любого языка, который вы решите создать. Несколько произвольно я введу для этой цели язык под названием *Jzero*. Никлаус Вирт (Niklaus Wirth) придумал игрушечный язык под названием *PL/0* (нулевой язык

программирования – *programming language zero*, название является аналогом названия языка PL/1), который использовался на курсах по созданию компиляторов. Jzero будет крошечным подмножеством Java, которое служит для аналогичной цели. Я довольно *усердно искал* (то есть гуглил *Jzero*, а затем *Jzero compiler*), не публиковал ли кто-нибудь определение Jzero, которое мы могли бы использовать, и не обнаружил этого, так что мы просто придумаем его по ходу дела.

Примеры Java в этой книге будут протестированы с использованием *OpenJDK 14*, возможно, другие версии Java (например, OpenJDK 12 или Oracle Java JDK) будут работать так же, а возможно, и нет. Вы можете получить OpenJDK на сайте *http://openjdk.java.net*, или, если вы работаете в Linux, ваша операционная система, вероятно, имеет пакет OpenJDK, который вы можете установить. Дополнительные инструменты для построения языка программирования (Jflex и byacc/j), которые требуются для примеров Java, будут представлены в последующих главах по мере их использования. Реализации Java, которые мы будем поддерживать, могут быть более ограничены тем, в каких версиях будут работать эти средства построения языка, чем чем-либо другим.

Примеры Unicon, приведенные в этой книге, работают с Unicon версии 13.2, которую можно получить на сайте *http://unicon.org*. Чтобы установить Unicon на Windows, необходимо загрузить файл **.msi** и запустить инсталлятор. Для установки на Linux обычно делают git-клон исходных текстов и вводят `make`. Затем нужно добавить каталог *unicon/bin* в **PATH**:

```
git clone git://git.code.sf.net/p/unicon/unicon
make
```

Пройдя нашу организацию и реализацию, которая будет использоваться в этой книге, возможно, стоит еще раз взглянуть на то, когда необходим язык программирования и когда можно обойтись без него, разработав вместо него библиотеку.

Язык и библиотека – в чем разница?

Не создавайте язык программирования, если с этой задачей справится библиотека. Библиотеки являются наиболее распространенным способом расширения существующего языка программирования для выполнения новой задачи. Библиотека – это набор функций или классов, которые могут быть использованы совместно для написания приложений для некоторых аппаратных или программных технологий. Многие языки, включая C и Java, почти полностью разработаны для того, чтобы использовать богатый набор библиотек. Язык сам по себе очень простой и общий, в то время как большая часть того, что разработчик должен изучить для создания приложений, заключается в том, как использовать различные библиотеки.

Библиотеки могут делать следующее:

- представлять новые типы данных (классы) и предоставлять публичные функции (API) для управления ими;
- предоставлять уровень абстракции поверх набора вызовов аппаратного обеспечения или операционной системы.

Что не могут делать библиотеки:

- вводить новые управляющие структуры и синтаксис для поддержки новых областей применения;
- встраивать/поддерживать новую семантику в существующей среде выполнения языка.

Библиотеки делают некоторые вещи плохо, в том смысле, что в конечном счете вы можете предпочесть создать новый язык:

- библиотеки часто становятся больше и сложнее, чем нужно;
- библиотеки могут иметь еще более крутые кривые обучения и более скудную документацию, чем языки;
- время от времени библиотеки конфликтуют с другими библиотеками, а несовместимость версий часто повреждает приложения, использующие библиотеки.

Существует естественный эволюционный путь от библиотеки к языку. Разумный подход к созданию нового языка для поддержки прикладной области заключается в том, чтобы начать с создания или покупки лучшей библиотеки, доступной для этой прикладной области. Если результат не удовлетворяет вашим требованиям с точки зрения поддержки области и упрощения написания программ для данной области, то у вас есть веские аргументы в пользу нового языка.

Эта книга о создании собственного языка, а не только о создании собственной библиотеки. Оказывается, изучение этих инструментов и методов полезно и в других контекстах.

Применимость к другим задачам разработки программного обеспечения

Инструменты и технологии, о которых вы узнаете в процессе создания собственного языка программирования, могут быть применены к ряду других задач разработки программного обеспечения. Например, можно разделить практически любую задачу обработки файлов или сетевого ввода на три категории:

- чтение данных XML с помощью библиотеки XML;
- чтение данных JSON с помощью библиотеки JSON;
- чтение чего-либо еще путем написания кода для анализа его в собственном формате.

Технологии, описанные в данной книге, полезны в широком спектре задач разработки программного обеспечения, и именно здесь мы сталкиваемся с третьей из этих категорий. Часто структурированные данные должны быть прочитаны в пользовательском формате файла.

Для некоторых из вас опыт создания собственного языка программирования может стать самой большой программой, которую вы написали до сих пор. Если вы будете упорны и доведете дело до конца, то это научит вас многим практическим навыкам разработки программного обеспечения, помимо того что вы узнаете о компиляторах, интерпретаторах и т. п. Среди прочих навыков это будет включать в себя работу с большими динамическими

структурами данных, тестирование программного обеспечения, диагностику сложных проблем.

Достаточно вдохновляющей мотивации. Давайте поговорим о том, что вы должны сделать в первую очередь, а именно выяснить свои требования.

Определение требований к вашему языку

После того как вы убедились, что вам нужен новый язык программирования для того, чем вы занимаетесь, потратьте несколько минут на определение требований. Это открытый вопрос. Вы сами определяете, как будет выглядеть успех вашего проекта. Мудрые изобретатели языков не создают совершенно новый синтаксис с нуля. Вместо этого они определяют его в виде набора модификаций популярного существующего языка. Многие великие языки программирования (Lisp, Forth, SmallTalk и многие другие) были значительно ограничены в своем успехе из-за того, что их синтаксис неоправданно отличался от основных языков. Тем не менее ваши требования к языку включают в себя то, как он будет выглядеть, в том числе и синтаксис.

Что еще более важно, вы должны определить набор управляющих структур или семантику, в которых ваш язык программирования должен выйти за рамки существующих языков. Иногда это будет включать в себя специальную поддержку прикладной области, которая не очень хорошо обслуживается существующими языками и их библиотеками. Такие *языки, специфичные для конкретной области (доменно-ориентированные языки – domain-specific languages – DSLs)*, настолько распространены, что этой теме посвящены целые книги. Цель данной книги – сосредоточиться на деталях создания компилятора и среды выполнения для такого языка, независимо от област, в которой вы работаете.

В обычном процессе разработки программного обеспечения анализ требований начинается с мозгового штурма списков функциональных и нефункциональных требований. Функциональные требования к языку программирования включают в себя специфику того, как конечный пользователь – разработчик будет взаимодействовать с ним. Вы можете не предусмотреть все опции командной строки для языка, но вы, вероятно, знаете, требуется ли интерактивность или достаточно отдельного шага компиляции. При обсуждении интерпретаторов и компиляторов прошлого, а также компилятора в этой книге может показаться, что выбор за вас уже сделан. Однако Python – это пример языка, который предоставляет полностью интерактивный интерфейс, несмотря на то что исходный код, который вы набираете в нем, преобразуется в байт-код, а не интерпретируется.

Нефункциональные требования – это свойства вашего языка программирования, которые не связаны напрямую со взаимодействием с конечным пользователем – разработчиком. К ним относятся такие вещи, как то, в какой операционной системе (системах) он должен работать, насколько быстрым должно быть выполнение или какой объем памяти должны занимать программы, написанные на вашем языке.

Нефункциональное требование относительно того, насколько быстрым должно быть выполнение, обычно определяет ответ на вопрос, можно ли использо-

вать программную машину (машину байт-кода) или необходимо использовать собственный код. Собственный код не только быстрее, его также значительно сложнее генерировать, и это может сделать ваш язык значительно менее гибким с точки зрения возможностей среды выполнения. Вы можете сначала выбрать байт-код, а затем работать над генератором собственного кода.

Первым языком, на котором я научился программировать, был интерпретатор BASIC, в котором программы должны были выполняться в пределах 4 Кб оперативной памяти. В то время BASIC имел низкие требования к объему памяти. Но даже в наше время нередко можно встретиться с платформой, где Java не запускается по умолчанию! Например, на виртуальных машинах с настроенными ограничениями памяти для пользовательских процессов вам, возможно, придется изучить некоторые неудобные параметры командной строки, чтобы скомпилировать или запустить даже простые Java-программы.

Многие процессы анализа требований также определяют набор вариантов использования и просят разработчика написать описания для них. Изобретение языка программирования отличается от вашего обычного проекта по разработке программного обеспечения, но прежде чем закончите, вы, возможно, захотите этим заняться. Пример использования – это задача, которую кто-то выполняет с помощью программного приложения. Когда программным приложением является язык программирования, если вы не будете осторожны, примеры использования могут быть слишком общими, чтобы быть полезными, например «*написать мое приложение*» или «*запустить мою программу*». Хотя эти два варианта могут быть не очень полезными, вы, возможно, захотите подумать о том, должна ли ваша реализация языка программирования поддерживать разработку программ, отладку, отдельную компиляцию и компоновку, интеграцию с внешними языками и библиотеками и т. д. Большинство этих тем выходят за рамки данной книги, но мы рассмотрим некоторые из них.

Поскольку в этой книге будет представлена реализация языка под названием Jzero, ниже приведены некоторые требования к нему. Некоторые из этих требований могут показаться произвольными. Если вам не ясно, откуда взялось одно из них, то оно либо пришло из нашего исходного языка вдохновения (PL/0), либо из предыдущего опыта обучения построению компиляторов:

- *Jzero должен быть строгим подмножеством Java*. Все легальные программы Jzero должны быть легальными программами Java. Это требование позволяет проверять поведение наших тестовых программ при отладке реализации нашего языка;
- *Jzero должен предоставлять достаточно возможностей для проведения интересных вычислений*. Это включает операторы `if`, циклы `while` и множество функций с параметрами;
- *Jzero должен поддерживать несколько типов данных, включая булевы, целые числа, массивы и тип String*. Он должен поддерживать только часть их функциональности, как будет описано позже. Этих типов достаточно, чтобы обеспечить ввод и вывод интересных значений в вычислениях;
- *Jzero должен выдавать приемлемые сообщения об ошибках, показывая имя файла и номер строки, включая сообщения о попытках использования возможностей Java, не предусмотренных в Jzero*. Нам понадобятся разумные сообщения об ошибках для отладки реализации;

- *Izero должен работать достаточно быстро, чтобы быть практичным.* Это требование расплывчато, но оно подразумевает, что мы не будем делать чистый интерпретатор. Чистые интерпретаторы – это очень древняя вещь, напоминающая о 1960-х и 1970-х годах;
- *Izero должен быть как можно более простым, чтобы я мог его объяснить.* К сожалению, это не позволяет генерировать собственный код или даже байт-код JVM. Мы предоставим нашу собственную простую машину байт-кода.

Возможно, по ходу дела появятся дополнительные требования, но это только начало. Поскольку мы ограничены во времени и пространстве, возможно, этот список требований более важен тем, что в нем не сказано, а не тем, что в нем сказано. Для сравнения, вот некоторые из требований, которые привели к созданию языка программирования Unicon.

Тематическое исследование – требования, которые вдохновили на создание языка Unicon

Для запуска конкретного примера в этой книге будет использован язык программирования Unicon, расположенный по адресу *http://unicon.org*. Мы можем начать с разумных вопросов, таких как «зачем создавать Unicon?» и «каковы его требования?». Чтобы ответить на первый вопрос, мы будем работать, отталкиваясь от второго.

Unicon существует благодаря более раннему языку программирования *Icon*, разработанному в университете Аризоны (*http://www.cs.arizona.edu/icon/*). Icon имеет особенно хорошие возможности обработки строк и списков, используется для создания многих скриптов и утилит, а также в проектах по обработке языков программирования и естественного языка. Фантастические встроенные типы данных Icon, включая структурные типы, такие как списки и (хеш-) таблицы, оказали влияние на несколько языков, включая Python и Unicon. Фирменным исследовательским вкладом Icon в знакомый основной синтаксис является интегрированная целенаправленная оценка, в том числе перебор с возвратом и автоматическое возобновление генераторов. Требование № 1 к Unicon – сохранить эти лучшие части Icon.

Требование Unicon № 1 – сохранять то, что люди любят в Icon

Одна из тех вещей, которые люди любят в Icon, – это семантика выражений, включая генераторы и целенаправленную оценку. Icon также предоставляет богатый набор встроенных функций и типов данных, так что многие или большинство программ могут быть поняты непосредственно из исходного кода. Целью Unicon была стопроцентная совместимость с Icon. В конце концов мы достигли более 99%-ной совместимости.

Это своего рода скачок от *сохранения лучших фрагментов* к цели бессмертия – обеспечению того, чтобы старый исходный код работал вечно, но для Unicon мы включили это в требование № 1. Мы предъявили более жесткие тре-

бования к обратной совместимости, чем большинство современных языков. В то время как C обладает очень хорошей обратной совместимостью, C++, Java, Python и Perl являются примерами языков, которые ушли, в некоторых случаях очень далеко, от совместимости с программами, написанными на них в дни их славы. В случае с Unicon, возможно, 99 % программ Icon работают без изменений, как программы Unicon.

Icon был разработан для максимальной производительности программистов в небольших проектах. Типичная программа Icon – это менее 1000 строк кода, но Icon – это очень высокий уровень, и вы можете создать множество программ в несколько сотен строк кода! Тем не менее компьютеры продолжают становиться более мощными, и пользователи хотят писать гораздо более крупные программы, чем те, на которые рассчитан Icon. Требованием № 2 Unicon была поддержка программирования в крупномасштабных проектах.

Требование Unicon № 2 – поддержка крупномасштабных программ, работающих с большими данными

По этой причине Unicon добавляет классы и пакеты в Icon, подобно тому как C++ добавляет их в C. Unicon также улучшил формат объектного файла байткода и сделал многочисленные улучшения масштабируемости компилятора и среды выполнения. Он также улучшает существующую реализацию Icon с целью сделать его более масштабируемым во многих специфических аспектах, например используя гораздо более сложную хеш-функцию.

Icon предназначен для классической текстовой обработки локальных файлов UNIX с помощью конвейеров и фильтров. Со временем все больше и больше людей хотели писать на нем и требовали более сложных форм ввода/вывода, таких как работа с сетью или графикой. Требование Unicon № 3 заключается в поддержке повсеместных возможностей ввода/вывода на том же высоком уровне, что и встроенные типы.

Требование Unicon № 3 – высокоуровневый ввод/вывод для современных приложений

Поддержка ввода/вывода – это движущаяся цель. Сначала она включала сетевые средства, средства GDBM и базы данных ODBC для сопровождения двухмерной графики Icon. Затем она расширилась и стала включать различные популярные интернет-протоколы и трехмерную графику. Определение того, какие возможности ввода/вывода являются повсеместными, продолжает развиваться и варьироваться в зависимости от платформы. Примерами вещей, которые стали в настоящий момент достаточно распространенными, являются сенсорный ввод и жесты, а также программирование шейдеров.

Пожалуй, несмотря на миллиардное увеличение в скорости процессора и объеме памяти, самая большая разница между программированием в 1970 и в 2020 годах заключается в том, что мы ожидаем, что современные приложения будут использовать огромное количество сложных форм ввода-вывода – графику, сети, базы данных и т. д. Библиотеки могут обеспечить доступ

к такому вводу/выводу, но поддержка на уровне языка может сделать его более простым и интуитивно понятным.

Icon довольно портативен, его запускали на всех компьютерах – от Amigas до Crays и мейнфреймов IBM с набором символов EBCDIC. Хотя за прошедшие годы платформы изменились почти невероятно, Unicon по-прежнему сохраняет цель Icon – максимальную переносимость исходного кода: код, написанный на Unicon, должен продолжать работать без изменений на всех важных вычислительных платформах. Это приводит к требованию Unicon № 4.

Требование Unicon № 4 – обеспечить универсально реализуемые системные интерфейсы

В течение очень долгого времени переносимость означала работу на ПК, Mac и рабочих станциях UNIX. Но, опять же, набор важных вычислительных платформ является движущейся целью. В настоящее время в Unicon ведется работа над поддержкой Android и iOS, в случае если вы считаете их вычислительными платформами. Будут ли они считаться таковыми, зависит от того, достаточно ли они открыты и используются ли для общих вычислительных задач, но они определенно могут быть использованы в этом качестве.

Все те впечатляющие средства ввода-вывода, которые были реализованы для требования № 3, должны быть разработаны таким образом, чтобы они могли быть многоплатформенными и переносимыми на все основные платформы.

Имея некоторые из основных требований к Unicon, можно ответить на вопрос, зачем вообще создавать Unicon. Один из ответов заключается в том, что после изучения многих языков я пришел к выводу, что генераторы Icon и целенаправленная оценка (требование № 1) были теми функциями, которые я хотел бы иметь при дальнейшем написании программ. Но после того как я позволил добавить двумерную графику в свой язык, изобретатели Icon больше не хотели рассматривать дальнейшие дополнения для удовлетворения требований № 2 и № 3. Другой ответ заключается в том, что существовал общественный спрос на новые возможности, включая партнеров-добровольцев и некоторую финансовую поддержку. Таким образом, родился Unicon.

Заключение

В этой главе вы узнали о разнице между изобретением языка программирования и изобретением библиотеки API для поддержки любых видов вычислений, которыми вы хотите заниматься. Было рассмотрено несколько различных форм реализации языка программирования. Первая глава позволила вам подумать о функциональных и нефункциональных требованиях к вашему собственному языку. Эти требования могут отличаться от приведенных в примерах требований, рассмотренных для подмножества Java Jzero и языка программирования Unicon.

Требования важны, потому что они позволяют вам установить цели и определить, как будет выглядеть успех. В случае реализации языка программирования требования включают в себя то, как все будет выглядеть и ощущаться программистами, использующими ваш язык, а также то, на каких аппарат-

ных и программных платформах он должен работать. Внешний вид и ощущение языка программирования включают в себя ответы на внешние вопросы о том, как будет выглядеть реализация языка и как вызываются программы, написанные на нем, а также на внутренние вопросы, такие как многословие – сколько программист должен написать, чтобы выполнить заданную вычислительную задачу.

Возможно, вам захочется сразу перейти к кодированию. Хотя классический принцип менталитета начинающих программистов «*собери и исправь*» может работать со скриптами и короткими программами, для такой большой части программного обеспечения, как язык программирования, нужно немного больше планирования. После рассмотрения требований в этой главе *глава 2* «*Проектирование языка программирования*» подготовит вас к составлению подробного плана реализации, который займет все наше внимание до конца нашей книги.

Вопросы

1. Каковы плюсы и минусы написания транспайлера языка (генерирует код на языке C) вместо традиционного компилятора (генерирует ассемблер или собственный машинный код)?
2. Каковы основные компоненты или шаги традиционного компилятора?
3. Каковы, по вашему опыту, некоторые болевые точки, в которых программирование оказывается сложнее, чем должно быть? Какая новая функция(и) языка программирования решает эти болевые точки?
4. Напишите набор функциональных требований для нового языка программирования.