РАЗРАБОТКА МЕТАЯЗЫКА И ГЕНЕРАТОРА КОМПИЛЯТОРА ДЛЯ СПЕЦИФИКАЦИИ ИНТЕГРИРУЕМЫХ ФУНКЦИЙ

О. А. Павловская

ВВЕДЕНИЕ

Среда исполнения платформы Microsoft .NET предполагает, что компиляторы входных языков генерируют сборки на машинно-независимом объектном языке (т.н. CIL-коде). Библиотека объектов общего назначения предоставляет необходимые средства для динамической генерации объектного CIL-кода. Такая функциональность открывает возможность эффективной разработки компиляторов на платформе .NET.

В статье рассматривается проблема разработки компиляторов специального семейства входных языков в машинно-независимый объектный код. Предлагаемая методика реализована в специальном приложении с графическим интерфейсом, позволяющем разработчику языка и компилятора эффективно взаимодействовать на различных фазах разработки компилятора.

МЕТАЯЗЫК СПЕЦИФИКАЦИИ ЛЕКСИКИ И СИНТАКСИСА

Лексика входного языка задается системой регулярных выражений [1], записанной в текстовом файле. Каждая его строка – это описание класса лексем, например, вида:

digit=0|1|2|3|4|5|6|7|8|9|0

literal=[letter]([digit]|[letter]*)

Здесь в квадратных скобках пишется название ранее определенного класса лексем для его повторного использования. Символы | (или), * (повторение 0 или более раз) и круглые скобки являются метасимволами спецификации (зарезервированы).

Синтаксис языка специфицируется контекстно-свободной грамматикой [1]. Ее правила также задаются в текстовом файле и имеют, например, следующий вид:

[plusexpr]->[plusexpr][PLUSOP][multexpr]`ADD{1}{0}{2}

Названия, написанные прописными буквами, определяют имена классов лексем, полученных сканером. После знака ` (апостроф) идет комментарий к правилу, который в дальнейшем будет использоваться при генерации кода.

Текстовый файл с правилами грамматики загружается в приложение, входная грамматика анализируется и, если выполнены LR(1)-условия,

генерируется LR(1)-анализатор [1]. В противном случае генерируется специальное сообщение, и управление возвращается пользователю для принятия решения по изменению входной грамматики.

ИНСТРУМЕНТЫ ПОСТРОЕНИЯ СКАНЕРА И СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА

По каждому классу лексем строится дерево регулярных выражений. Листья дерева помечены символами входной строки, внутренние вершины – операциями. В результате получим множество деревьев регулярных выражений.

По каждому дереву конструируется детерминированный конечный автомат(ДКА) [1],[2]. Он будет определять, принадлежит ли входная строка данному классу лексем. В коде программы ДКА представлен классом `Machine`. На вход его основного метода `MoveForward` передается текущее состояние и просматриваемый символ. По этим параметрам определяется следующее состояние, которое возвращается в качестве результата. Поле `IsFinal` экземпляра класса определяет, является ли данное состояние финальным. Если состояние финальное, то анализируемая строка принадлежит данному классу лексем.

Чтобы определить, к какому классу принадлежит лексема, создан класс `ComplexMachine.` Он содержит в себе коллекцию объектов класса `Machine`, и представляет собой некоторый ДКА, в нем тоже есть метод `MoveForward`, но текущее состояние задается вектором состояний инкапсулированных автоматов. Текущее состояние является финальным, если ровно одно состояние в векторе является финальным, так как лексема не может одновременно принадлежать различным классам. Автомат, который содержит это финальное состояние, допускает проверяемое слово, и, значит, строка принадлежит классу лексем, задаваемых этим автоматом. Единственное исключение – классы literal и keyword. Так как ключевое слово является литералом, оба автомата, соответствующие этим классам, могут одновременно быть в финальном состоянии. В этом случае лексема считается ключевым словом.

Кроме определения класса лексемы, `ComplexMachine` осуществляет разбиение входного потока символов на токены, пытаясь выделить из него максимально длинные лексемы. Именно эта способность и используется во внешних классах. Метод `GetNextToken` возвращает следующий токен и переводит итератор в положение после него. Тип `Token` содержит два поля – класс токена и его значение.

Очередной токен сканируется в момент, когда он потребуется в программе синтаксическим анализатором, полем вывода текста или другим

инструментом. Существует и другой подход, при котором вначале входной поток преобразовывается в последовательность токенов, и синтаксический анализатор работает уже с этой последовательностью.

Результатом построения LR(1)-анализатора по КС-грамматике мы получаем класс `MagazineMachine`. Главным его методом является `AnalyzeString`. Он принимает на вход код на созданном языке и выдает последовательность продукций грамматики, которые необходимо применить для интерпретации этого кода. Внутри этот метод использует созданный сканер, класс `ComplexMachine`, который и разбивает входную строку на токены. `AnalyzeString` получает эти токены по одному, и, в зависимости от текущего состояния и полученного значения, продолжает анализ (переходит в новое состояние) или завершает работу с кодом ошибки или успеха.

По последовательности правил, полученной в результате работы метода, строится синтаксическое дерево, представленное классом `ExpressionTree`. Узлы этого дерева – объекты класса `ExpressionTreeItem`, содержащие информацию о примененной продукции и текущем анализируемом токене. Узлам дерева соответствуют нетерминалы и правила, листьям – терминалы. Для генерации кода преобразуем дерево в т.н. абстрактное синтаксическое дерево (АСД).

В описании грамматики правилам соответствуют комментарии, которые определяют процедуры генерации объектного кода. По существу, спецификация грамматики содержит правила генерации объектного кода.

АЛГОРИТМЫ ГЕНЕРАЦИИ И ИСПОЛНЕНИЯ ОБЪЕКТНОГО КОДА

Для работы с созданием CIL-кода, а также созданием и упаковкой сборок .NET используется компонент библиотеки базовых классов, именуемый `System.Reflection.Emit`.

Класс `ILGenerator` применяется для генерации CIL-кода. Он использует класс `OpCodes`, являющийся большим перечислением, содержащим все допустимые инструкции CIL-языка [3].

Функция исполнения сгенерированного объектного кода называется `count`. Она получает на вход тип `double` и возвращает значение типа `double`. Чтобы записать CIL-код этой функции, используем объект класса `ILGenerator`:

```
ILGenerator il = count.GetILGenerator();
```

В процессе обхода дерева будем добавлять в него инструкции: если мы анализируем лист, то его значение помещается в стек, иначе выполняем инструкции для правила, соответствующего узлу. Порядок обхода дерева с корнем в данном узле зависит от продукции узла. Обход производится с помощью рекурсивной процедуры `EmitNode.`

Для работы с переменными используется словарь `Dictionary<string, LocalBuilder> SymbolTable.` В начале работы в него помещается переменная `x` , переданная в качестве параметра функции `count`. При использовании новой переменной в словарь заносится ее идентификатор и адрес в памяти. При операциях чтения значение по этому адресу помещается на вершину стека, а при операциях записи значение из вершины стека помещается в соответствующую ячейку памяти. На данном этапе производится контроль типов.

ЗАКЛЮЧЕНИЕ

Методика разработки компилятора по спецификациям входного языка реализуется поэтапно в приложении, которое предоставляет пользователю удобный интерфейс для контроля каждого этапа разработки. В качестве теста корректности разработки используется класс входных языков для спецификации интегрируемых функций.

Литература

1. *Ахо А., Сети Р., Ульман Д.* Компиляторы: принципы, технологии и инструментарий / М., 2003.
2. *Серебряков В.А., Галочкин М.П.* Основы конструирования компиляторов / М., 1999.
3. *Побар Дж*. Создание компилятора языка для .NET Framework // MSDN Magazine. 2008. № 2.С.17-19.