Язык описания абстрактного синтаксиса TreeDL и его использование

Алексей Демаков (demakov@ispras.ru)

Аннотация. Разработка трансляторов и других инструментов обработки языков требует значительных усилий. Одним из способов снижения трудоемкости является использование явного описания абстрактного синтаксиса. Предлагается специализированный язык TreeDL для описания абстрактного синтаксиса и операций над ним. Снижение трудоемкости разработки достигается за счет генерации отдельных компонентов инструмента. Поддерживается переиспользование распознавателя расширяемого языка и структуры внутреннего представления при специализированной обработке или расширении существующих языков.

1. Введение

Задача разработки трансляторов языков программирования изучена достаточно хорошо [1,2,3]. Несмотря на то, что принципиальных проблем при разработке подобных инструментов не возникает, реализация их остается весьма трудоемкой задачей. Потребность в таких инструментах велика, поэтому требуется дополнительное изучение способов облегчения разработки.

Функциональность транслятора укладывается в обобщенную схему, которая показана на Рис. 1.

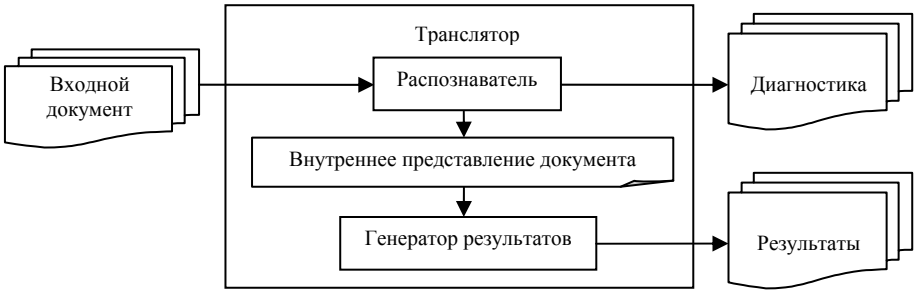

Рис. 1

Документ поступает на вход *распознавателя*, который проверяет его корректность относительно входного языка. В случае обнаружения ошибок

выдается соответствующая диагностика, и работа транслятора завершается. Корректный документ переводится во *внутреннее представление* и передается *генератору результатов* для обработки и получения требуемых результатов. Таким образом, структура внутреннего представления определяет интерфейс взаимодействия основных подсистем транслятора.

В случае трансляции языка программирования входной документ – это программа на входном языке программирования, а результаты – это исполнимый код.

Входной язык может быть не только языком программирования, но и языком запросов, языком разметки и описания данных, языком спецификации и т.п.

Кроме трансляции существуют и другие виды обработки, вот некоторые из них:

- Генерация кода на языке программирования по спецификации (описанию модели).
- Генерация тестов. Исходной информацией может быть как модель или спецификация, так и тестируемая программа.
- Дальнейший анализ или верификация документа на исходном языке.
- Трансформация документа в другой формат или представление.
- Генерация отчетов или документации.

Далее в этой статье термин *транслятор* будет использоваться в широком смысле слова – для обозначения любого инструмента обработки документов на некотором языке.[1]

Существует класс задач, для решения которых генератору результатов достаточно информации, поступающей от распознавателя в процессе разбора входного документа – то есть, обработка документа может быть осуществлена за один проход и нет необходимости строить внутреннее представление всего документа. Такой подход обеспечивает высокую скорость обработки и оправдан, если входной документ может иметь очень большой размер и его полное внутреннее представление потребует значительных ресурсов. Такую структуру имеют интерпретаторы языков программирования, однопроходные компиляторы и потоковые XML парсеры.

В общем случае, если для анализа требуется полная информация о входном документе, распознаватель строит полное внутреннее представление документа, которое затем обрабатывается генератором результатов. Далее предполагается именно такое устройство транслятора.

Внутреннее представление документа, которое представляет собой дерево, соответствующее иерархии сущностей входного документа, называют деревом

[1] Такое расширение понятия трансляции характерно для англоязычной литературы.

абстрактного синтаксиса. Этот вид внутреннего представления используется наиболее широко. Далее предполагается именно такой вид внутреннего представления. Способы представления абстрактного синтаксиса и работы с ним будут подробнее описаны в последующих разделах.

В этой статье рассматривается задача *быстрой разработки* или *прототипирования* трансляторов. Эта задача возникает в исследовательской практике при разработке новых или расширении существующих языков программирования, а также при решении практических задач, требующих использования специализированных языков [4]. От таких трансляторов в первую очередь требуется не высокая производительность, а короткие сроки реализации, а также удобство поддержки и модификации, в частности, в условиях постоянного изменения входного языка.

Рассмотрим основные способы облегчения разработки трансляторов, их использование в существующих подходах к разработке трансляторов и проблемы, которые при этом возникают.

1. Выбор подходящего языка реализации транслятора.

   Помимо процедурных языков общего назначения (в частности, объектно-ориентированных) в качестве языка реализации транслятора используются функциональные языки [5, 6] и специализированные языки для разработки трансляторов [7].

   Возможности функциональных языков [8] и специализированных языков для разработки трансляторов облегчают реализацию трансляторов по сравнению с использованием процедурных языков общего назначения.

   Использование функциональных языков и специализированных языков для разработки трансляторов затруднено тем, что они сравнительно мало распространены, поэтому вполне вероятно, что преимущества от использования этих языков будут сведены на нет необходимостью их изучения.

   При использовании процедурного языка в качестве языка реализации транслятора трудоемкость разработки зависит от наличия в языке развитых механизмов общего назначения. К таким механизмам относятся, в частности, объектно-ориентированные возможности, автоматическое управление памятью и строгий контроль типов. Наличие таких механизмов облегчает переиспользование компонентов, позволяет выявить большее количество ошибок во время компиляции, что в целом облегчает разработку.

   С этой точки зрения современные объектно-ориентированные языки, такие как Java или C# выглядят предпочтительнее C/C++.

2. Выбор подходящих вспомогательных инструментов.

Отдельные компоненты транслятора удобно описывать на специализированных языках и генерировать по этим описаниям представление компонентов на языке реализации транслятора.

Наиболее широко распространены языки описания лексической и синтаксической грамматик, обычно в виде регулярных выражений и контекстно-свободной грамматики. Для всех широко распространенных языков, которые могут быть использованы для разработки трансляторов, реализованы генераторы лексических и синтаксических анализаторов, генераторы структуры внутреннего представления и базовых компонентов обработки внутреннего представления.

Использование специализированных языков описания компонентов транслятора частично компенсирует отсутствие в языках общего назначения специализированных возможностей, удобных для разработки трансляторов.

К недостаткам генераторов компонентов транслятора следует отнести тот факт, что чаще всего не предусмотрено изменение или расширение их функциональности пользователем. То есть набор и структура компонентов, которые могут быть созданы автоматически по описанию компонентов, фиксированы.

Например, описание структуры внутреннего представления может быть использовано для генерации типов данных, представляющих структуру внутреннего представления на языке реализации транслятора. Кроме этого, по описанию структуры внутреннего представления может быть сгенерирован код компонента обходчик, осуществляющего обход внутреннего представления. Если генератор по описанию структуры внутреннего представления создает только типы данных, но не обходчик, и его функциональность не может быть расширена, обходчик придется реализовывать вручную. Если генератор создает обходчик, но порядок обхода не такой, как требуется, и функциональность генератора не может быть изменена или расширена, обходчик также придется реализовывать вручную.

Нерасширяемые генераторы компонентов затрудняют использование имеющегося описания распознавателей и внутреннего представления для генерации компонентов транслятора, необходимых пользователю.

Отметим также разработку систем метапрограммирования [9], облегчающих описание предметно-ориентированных языков и правил их обработки. Эти системы облегчают интеграцию предметно-ориентированного языка с основным языком программирования. Системы метапрограммирования обладают большим потенциалом, но пока представляют собой скорее

прототипы, не пригодные для использования в реальных проектах. Реализация языка с использованием системы метапрограммирования сильно привязана к этой системе, что накладывает ограничения на способ использования предметно-ориентированного языка.

3. Повторное использование готовых компонентов транслятора. В частности, при специализированной обработке или расширении существующих языков, крайне желательно повторно использовать распознаватель расширяемого языка.

   Хотя существуют расширяемые трансляторы [10] и каркасные библиотеки, облегчающие расширение существующих языков [11, 12], в большинстве своем трансляторы не предоставляют сторонним разработчикам программный интерфейс к результатам работы распознавателя языка. Не существует основанной на едином подходе свободно-распространяемой библиотеки готовых распознавателей для распространенных языков[2]. Отсутствие такой библиотеки существенно затрудняет разработку инструментов, осуществляющих специализированную обработку существующих языков и их расширений.

Таким образом, достаточно распространенный подход к разработке транслятора выглядит следующим образом:

1. В качестве языка реализации транслятора используется современный широко-распространенный объектно-ориентированный язык (например, Java или C#).
2. Для создания отдельных компонентов транслятора используются нерасширяемые вспомогательные инструменты, генерирующие реализацию этих компонентов по их описанию на специализированном языке. Практически всегда таким образом создаются лексические и синтаксические анализаторы текстовых языков. Используются также специализированные описания структуры внутреннего представления и операций над ним (в том числе трансформации и генерации текста на основе шаблонов).
3. Реализация транслятора использует стандартные компоненты (например, для разбора параметров командной строки, вывода диагностических сообщений и т.п.). Распознаватель языка и структура внутреннего представления чаще всего создается заново.

[2] Существует коммерческая библиотека [13] распознавателей и построенных на их основе инструментов (форматирование исходных текстов, сбор метрик, оптимизация и т.п.). Библиотека реализована на языке программирования PARLANSE, поддерживающем параллельные вычисления.

2. Использование описания абстрактного синтаксиса языка при разработке трансляторов

Предлагаются следующие дополнительные требования к описанному подходу разработки трансляторов, связанные с абстрактным синтаксисом.

- Использование описания абстрактного синтаксиса, которое не зависит от распознавателя языка. Это позволяет вести независимую разработку основных подсистем транслятора – распознавателя языка и генератора результатов.
- Использование языка описания абстрактного синтаксиса, который обеспечивает модульность описания абстрактного синтаксиса. Это позволяет при создании транслятора расширения языка переиспользовать модуль описания абстрактного синтаксиса расширяемого языка.
- Использование расширяемого пользователем инструмента, который обрабатывает описание абстрактного синтаксиса. Расиряемый инструмент дает пользователю возможность пополнять набор компонентов транслятора, генерируемых по описанию абстрактного синтаксиса, а также изменять имеющиеся схемы генерации компонентов.

Таким образом, использование подходящего языка описания абстрактного синтаксиса и соответствующей инструментальной поддержки облегчает разработку трансляторов в рамках описанного подхода. В следующих разделах рассмотрены преимущества и недостатки существующих способов описания абстрактного синтаксиса, на этой основе предлагается язык описания абстрактного синтаксиса и его инструментальная поддержка, удовлетворяющие сформулированным требованиям.

2.1. Абстрактный синтаксис

В качестве внутреннего представления текстового документа, язык которого задан грамматикой, может использоваться дерево синтаксического разбора документа. Однако внутреннее представление документа отражает его логическую структуру и может скрывать несущественные для анализа и дальнейшей обработки детали конкретного внешнего представления. Например, внутреннее представление программы может абстрагироваться от способа форматирования текста и конкретного синтаксиса конструкций.

Такой вид внутреннего представления называют *абстрактным синтаксисом*. Различают абстрактный синтаксис первого и более высоких порядков. Абстрактный синтаксис первого порядка представляет собой дерево, соответствующее иерархии сущностей документа (классов, методов, выражений и т.п.). Если это дерево дополнить связями между использованиями и определениями сущностей (переменных, типов и т.п.), получится

ориентированный граф, представляющий абстрактный синтаксис более высокого порядка [14].

Дерево абстрактного синтаксиса первого порядка строится распознавателем при анализе синтаксической структуры входного документа. При анализе контекстных ограничений это дерево может быть достроено до ориентированного графа абстрактного синтаксиса более высокого порядка.

Использование абстрактного синтаксиса в качестве внутреннего представления позволяет использовать один и тот же генератор результатов для обработки нескольких представлений входного языка. Например, для языка UML [15] основным является графическое представление, но существует эквивалентное XML представление [16] и текстовое представление [17], которые могут быть преобразованы распознавателем в одно и то же абстрактное внутреннее представление и обработаны одним и тем же генератором результатов. Аналогично, для языка RELAX NG [18] описания схем XML документов существуют эквивалентные XML представление и компактный синтаксис, которые могут быть представлены с использованием общего абстрактного синтаксиса для дальнейшей обработки.

Для представления абстрактного синтаксиса может быть использовано *гомогенное* или *гетерогенное* дерево.

Реализация *гомогенного* дерева использует один и тот же тип данных для всех вершин дерева. Этот тип данных содержит два или более полей для хранения ссылок на дочерние вершины. Тип вершины при этом также хранится в одном из полей вершины. Тип данных, представляющий вершину дерева, не зависит от структуры абстрактного синтаксиса. При изменении структуры абстрактного синтаксиса может измениться только набор констант, определяющих типы вершин. На первый взгляд это облегчает изменение структуры абстрактного синтаксиса. Однако при изменении структуры дерева требуется не только изменить набор типов вершин, но и определить, на какие дочерние вершины ссылаются поля вершины. Эта информация не хранится в структуре гомогенного дерева, но необходима для работы с деревом абстрактного синтаксиса. Поэтому при изменении структуры абстрактного синтаксиса дополнительно к изменению набора констант для типов вершин требуется документировать содержимое полей вершины для каждого нового или измененного типа вершины.

Поскольку информация о допустимых типах дочерних вершин отсутствует в структуре гомогенного дерева, невозможна статическая проверка корректности построения и обработки дерева средствами языка реализации транслятора. Это может увеличить трудоемкость реализации и увеличить время, необходимое для отладки транслятора. Частично проблема может быть решена использованием специализированных средств для построения и обработки дерева. Такой подход используется в инструменте ANTLR [19], который осуществляет генерацию лексических и синтаксических анализаторов, построение гомогенного дерева внутреннего представления и обработку этого

дерева с помощью анализаторов деревьев (tree parsers). Также возможно расширение языка реализации транслятора конструкциями, облегчающими создание и обработку гомогенных деревьев [20].

В заключение отметим, что использование гомогенного дерева облегчает распределение памяти для хранения дерева, что является существенным, если язык реализации транслятора не обладает развитыми средствами управления памятью.

Реализация *гетерогенного* дерева определяет множество типов вершин, допустимых в дереве. Тип вершины задает набор типов дочерних вершин и атрибутов. Фактически, структура гетерогенного дерева хранит информацию, которая содержалась в неформальных комментариях или документации к структуре гомогенного дерева. Следовательно, изменение абстрактного синтаксиса при использовании гетерогенного дерева требует примерно столько же усилий, как и при использовании хорошо документированного гомогенного дерева.

Гетерогенное дерево наиболее точно отражает логическую структуру абстрактного синтаксиса и позволяет использовать средства языка реализации транслятора для статической проверки корректности использования типов вершин. При этом сокращается время, необходимое для отладки транслятора, следовательно, снижается и трудоемкость создания транслятора в целом. Представление абстрактного синтаксиса в виде гетерогенного дерева используется инструментами SableCC [21], Cocktail/AST [22], TreeCC [23].

Если типы вершин гетерогенного дерева представляются классами языка программирования, операции обработки дерева могут быть реализованы как виртуальные методы типов вершин. Такой подход требует модификации типов вершин при добавлении каждой новой операции. Избежать этого позволяет использование шаблона проектирования *Посетитель* [24], при котором методы обработки всех типов вершин собираются в отдельный класс-посетитель, а выбор метода, соответствующего фактическому типу обрабатываемой вершины, осуществляется во время работы транслятора с помощью механизма *двойной диспетчеризации* [24].

Но и это решение имеет свои недостатки:

- Фиксированную сигнатуру методов, что затрудняет передачу параметров и возврат результатов.
- Необходимость определения методов для всех типов вершин, даже если логически операция определена только на наследниках определенного типа. Применение операции к вершинам других типов недопустимо и является ошибкой в трансляторе, но средствами языка реализации транслятора невозможно обнаружить эту ошибку на этапе компиляции.
- Невозможность предотвратить средствами языка ошибки, связанные с изменением набора типов вершин дерева – при появлении нового

типа вершин для него будет использована реализация операции по умолчанию, вместо сообщения о том, что требуется определить операцию на этом типе вершин.

Подробный разбор недостатков распространенных видов реализации операций над деревьями имеется в статье [23]. Там же описан способ реализации операций над деревьями при помощи *мультиметодов* [25].

Мультиметоды являются обобщением виртуальных методов. Для каждого наследника типа, в котором объявлен виртуальный метод, может быть определена своя реализация метода, которая будет использована в зависимости от фактического типа объекта, у которого метод вызван. Назовем *виртуальным* параметр, для которого реализация метода выбирается по фактическому типу. Виртуальный метод имеет один виртуальный параметр – объект, у которого вызывается метод. Мультиметод может иметь произвольное количество виртуальных параметров. Для каждой комбинации фактических типов виртуальных параметров может быть определена своя реализация мультиметода.

Использование мультиметодов для операций над деревом требует инструментальной поддержки, поскольку эта конструкция не реализована во многих языках программирования[3]. Для обработки гетерогенного дерева существует эффективная реализация мультиметодов, которая позволяет осуществлять выбор реализации мультиметода за время, пропорциональное количеству виртуальных параметров, независимо от количества типов вершин в описании структуры дерева. Поскольку всё множество типов известно заранее, то известно и множество наследников каждого типа. Каждому типу может быть назначен уникальный числовой идентификатор, который является индексом в многомерном массиве, содержащем реализации мультиметода. Время доступа к элементу многомерного массива зависит только от размерности, то есть от количества виртуальных параметров.

Реализация операций над деревом с помощью мультиметодов не имеет таких недостатков, как при использовании шаблона Посетитель:

- Мультиметод имеет произвольную сигнатуру, что позволяет легко передавать дополнительные параметры и возвращать значения.
- Мультиметод может быть задан не для всех типов вершин, а только для наследников определенного типа.
- Можно проверить, что для каждой комбинации фактических типов виртуальных параметров определена реализация мультиметода. При изменении набора типов вершин несоответствие определения мультиметода структуре дерева выявляется статически.

[3] Для языка Java существует расширение MultiJava [25], в котором поддержка мультиметодов реализована.

Типы вершин гетерогенного дерева могут определять универсальный интерфейс, позволяющий использовать для обработки такого дерева те же приемы, что и для гомогенных деревьев.

Таким образом, преимуществом гетерогенных деревьев является возможность использования различных способов реализации операций, что позволяет выбрать способ, приемлемый для каждого конкретного случая.

Описание дерева внутреннего представления может быть частью описания синтаксического распознавателя языка. Некоторый способ построения дерева по заданной грамматике определен по умолчанию. Могут быть указаны дополнительные пометки элементов конкретной синтаксической грамматики, используемой распознавателем, которые уточняют способ построения дерева абстрактного синтаксиса (SableCC v.2, ANTLR). В любом случае описание внутреннего представления, которое является интерфейсом между распознавателем языка и генератором результатов, оказывается зависимым от реализации распознавателя языка. Это не позволяет ускорить разработку транслятора путем независимой разработки распознавателя языка и генератора результатов. К тому же, разработчикам транслятора для изучения структуры внутреннего представления необходимо владеть достаточно сложным языком описания синтаксического распознавателя. То есть, такое описание интерфейса взаимодействия подсистем транслятора неудобно использовать в качестве документации.

Более удобным представляется явное описание абстрактного синтаксиса, не привязанное к синтаксическому анализатору. Структура гетерогенного дерева при этом может описываться с помощью языков, основанных на контекстно-свободных грамматиках (Cocktail/Ast, SableCC v.3) или на классах с одиночным наследованием (TreeCC). Одним из основных отличий между этими способами является группировка альтернатив. Контекстно-свободная грамматика явно перечисляет все возможные альтернативы в одном месте. Наследование классов позволяет использовать любого наследника заданного класса в качестве альтернативы. Следующий пример иллюстрирует использование этих способов для представления декларации типов в языках Java 2 [26] и Java 3 [27].

Пример:

В языке Java 2 декларация типа определяет класс, либо интерфейс. С помощью BNF контексно-свободной грамматики эта альтернатива описывается следующим образом:

```
JavaTypeDeclaration ::=   ClassDeclaration
                        | InterfaceDeclaration
                    ;
```

В правиле `JavaTypeDeclaration` явно перечислены все возможные варианты декларации типа. Эта же декларация типа описывается при помощи классов с одиночным наследованием так:

```
abstract class JavaTypeDeclaration { … }

class ClassDeclaration
    extends JavaTypeDeclaration { … }

class InterfaceDeclaration
    extends JavaTypeDeclaration { … }
```

Возможные варианты декларации типа описываются не самим классом `JavaTypeDeclaration`, а его наследниками. Рассмотрим изменения в приведенном фрагменте описания абстрактного синтаксиса, которые необходимы для представления дополнительных возможностей языка Java 3: деклараций перечислимых типов и типов аннотаций.

В грамматику добавляются новые альтернативы:

```
JavaTypeDeclaration ::=   ClassDeclaration
                        | InterfaceDeclaration
                        | EnumDeclaration
                        | AnnotationTypeDeclaration
                    ;
```

Добавление альтернатив языка Java 3 не требует модификации существующих классов.

```
class EnumDeclaration
    extends JavaTypeDeclaration { … }

class AnnotationTypeDeclaration
    extends JavaTypeDeclaration { … }
```

Таким образом, расширение описания структуры дерева, основанного на контекстно-свободной грамматике, требует переопределения существующих правил, тогда как описание, основанное на классах с одиночным наследованием, в тех же случаях позволяет обойтись лишь добавлением классов.

Приведенный пример показывает, что модульность описания абстрактного синтаксиса достигается при использовании языка, основанного на классах с одиночным наследованием.

3. Разработка трансляторов с использованием языка TreeDL

Для описания структуры абстрактного синтаксиса и операций над ним предлагается язык TreeDL, который будет описан далее. Обработка описания абстрактного синтаксиса и генерация компонентов транслятора осуществляется инструментом `treedl`, который также будет описан далее.

Схема реализации транслятора с использованием языка TreeDL показана на Рис. 2.

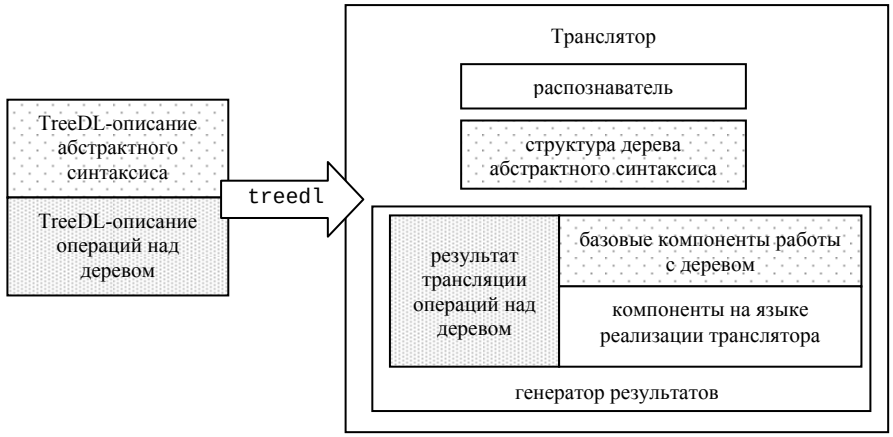

Рис. 2

Проектные решения, использованные при создании языка TreeDL, соответствуют сделанным в предыдущем разделе выводам о необходимых возможностях языка описания абстрактного синтаксиса, которые облегчают разработку транслятора:

- Язык позволяет описывать абстрактный синтаксис как структуру гетерогенных деревьев.
- Описание абстрактного синтаксиса состоит из модулей, которые могут быть использованы другими модулями. При расширении языка может быть переиспользован модуль описания абстрактного синтаксиса расширяемого языка.
- Операции над деревьями абстрактного синтаксиса определяются как мультиметоды. При переиспользовании модулей описания абстрактного синтаксиса также могут быть переиспользованы определения операций.

Инструмент `treedl` реализован в соответствии со сделанными в предыдущем разделе выводами о необходимых возможностях инструмента обработки описания абстрактного синтаксиса, облегчающих разработку трансляторов:

- Трансляция описания абстрактного синтаксиса и операций над ним осуществляется в современные объектно-ориентированные языки Java и C#.
- Описание абстрактного синтаксиса также используется для генерации компонентов транслятора, которые определяются описанием абстратного синтаксиса.
- Функциональность инструмента может быть расширена пользователем с помощью подключаемых модулей. Это позволяет расширить набор генерируемых компонентов транслятора или изменить имеющиеся схемы генерации компонентов.

Описание языка TreeDL и исходные тексты и документация инструмента обработки распространяются свободно [28].

3.1. Язык TreeDL описания структуры деревьев

Для описания структуры деревьев в Институте Системного Программирования РАН разработан и используется специализированный язык TreeDL [28]. Далее приведено краткое описание основных возможностей языка TreeDL, иллюстрированное фрагментами описания абстрактного синтаксиса языка Java.

3.2. Структура дерева

Описание структуры дерева задает набор типов вершин, которые могут быть использованы в дереве, и оформляется в виде *модуля* на языке TreeDL. *Тип вершины* определяет набор именованных дочерних вершин и атрибутов, которыми обладают все вершины этого типа. *Дочерние вершины* определяют структуру дерева, а *атрибуты* задают дополнительную информацию, связанную с вершиной. Для атрибутов и дочерних вершин определены операции получения значения и установки нового значения.

Рассмотрим пример использования перечисленных констркций языка:

Пример:

```
tree com.unitesk.java5.Java5;
    ...
node ClassDeclaration
{
    attribute string name;
    attribute string? optSuperClassName;
    child ClassMemberDeclaration* optMemberList;
        ...
}
```


В модуле `com.unitesk.java5.Java5` определен тип вершин для деклараций классов. Декларация класса имеет обязательный строковый атрибут `name`, хранящий имя класса, необязательный строковый атрибут `optSuperClassName`, хранящий имя суперкласса и список деклараций членов класса `optMemberList`, содержащий ноль или более элементов типа `ClassMemberDeclaration`. `ClassMemberDeclaration` – это тип вершин, определенный в этом же модуле.

Возможное количество значений атрибутов или дочерних вершин задается модификаторами ‘?’, ‘*’, ‘+’ типа, которые разрешают «ноль или один», «ноль или более», «один или более» элементов соответственно. В отличие от языков программирования (Java, C#) `null` не является допустимым значением типа, если тип не имеет модификатора ‘?’. При трансляции описания дерева в язык программирования в операции установки значения вставляются соответствующие проверки[4]: на `null` для атрибутов без модификатора ‘?’, на количество элементов для атрибутов с модификатором ‘+’.

В операции доступа может быть вставлен дополнительный пользовательский код, например, осуществляющий дополнительные проверки при установке значения атрибута, или выполняющий дополнительные вычисления при запросе значения атрибута.

Пример:

```
attribute string name
set {
  if( !isID( name ) )
  {
    throw new IllegalArgumentException();
  }
};
```

В метод установки значения атрибута `name` будет вставлен дополнительный код, проверяющий, что устанавливаемое значение является корректным идентификатором в языке Java.

---

[4] При трансляции описания дерева в язык программирования, который поддерживает типы, не допускающие `null` в качестве значения [29], проверки могут быть заменены на использование таких типов, что обеспечивает проверку на этапе компиляции, а не во время выполнения программы. Поддержка таких типов может быть введена и не на уровне языка, а на уровне среды разработки [30].

Ограничения могут накладываться не только на отдельные атрибуты типа вершины, но и на совокупность значений атрибутов вершины. Проверка этих ограничений производится при создании вершины.

Пример:

Рассмотрим конструкцию обработки исключений языка Java:

```
try { … }
catch( … ) { … }
finally { … }
```

Согласно описанию языка Java finally-блок не обязателен, catch-блоков может быть 0 или более, но если catch-блоки отсутствуют, то finally-блок присутствовать обязан. Такое требование не может быть выражено ограничениями для отдельных атрибутов, необходимо ограничение на совокупность значений атрибутов и дочерних вершин.

```
node TryStatement : Statement
{
  child Block block;
  child CatchClause* optCatchClauseList;
  child Block? optFinallyBlock;

  constructor
  {
    if(    sizeOptCatchClauseList() == 0
        && getOptFinallyBlock() == null
      )
    {
      throw new IllegalArgumentException();
    }
  }
}
```

При трансляции описания дерева в язык программирования в конструктор класса, представляющего тип вершин `TryStatement` будет вставлена соответствующая проверка.

Возможность использования кода на языке программирования в описании абстрактного синтаксиса введена из практической необходимости. Это мощное средство описания дополнительных ограничений на структуру дерева, но при чрезмерном использовании оно может привести к увеличению размера описания и ухудшению его наглядности.

При построении дерева абстрактного синтаксиса более высокого порядка имена используемых сущностей заменяются или дополняются ссылками на их

определения. Дерево абстрактного синтаксиса более высокого порядка может быть реализовано как снабженное дополнительными атрибутами дерево абстрактного синтаксиса первого порядка.

Пример:

Вернемся к рассмотрению типа вершин `ClassDeclaration`, представляющего декларацию класса в языке Java.

```
    ...
node ClassDeclaration
{
    attribute string? optSuperClassName;
        ...
    attribute late ClassDeclaration? optSuperClass;
}
```

Теперь для вершин `ClassDeclaration` задан дополнительный атрибут `optSuperClass`, содержащий ссылку на декларацию суперкласса. Модификатор `late` определяет, что значение атрибута будет определено не при создании вершины, а позже – в данном случае при анализе контекстных ограничений входного языка.

Для типов вершин определено одиночное наследование. Как было показано ранее, оно используется для представления альтернатив грамматики.

Пример:

Членами класса в языке Java являются конструкторы, декларации методов и полей.

```
    ...
abstract node ClassMemberDeclaration
{
    attribute string name;
}

node ConstructorDeclaration : ClassMemberDeclaration
{ ... }

node MethodDeclaration : ClassMemberDeclaration
{ ... }

node FieldDeclaration : ClassMemberDeclaration
{ ... }
```

Тип вершин `ClassMemberDeclaration` объявлен абстрактным, от него унаследовано три неабстрактных типа: `ConstructorDeclaration`, `MethodDeclaration` и `FieldDeclaration`, которые наследуют атрибут `name`.

3.3. Операции над деревом

Набор типов вершин, допустимых в дереве, является сущностью, которая не имеет аналогов в языках программирования общего назначения, так как определяет всех возможных наследников каждого типа вершины. Это дает возможность эффективно реализовать операции над деревьями при помощи мультиметодов, то есть методов с несколькими виртуальными параметрами. Для каждого набора типов виртуальных параметров должна быть задана отдельная реализация мультиметода.

Пример:

```
operation
bool check( virtual ClassMemberDeclaration member )
{
case( ConstructorDeclaration member ):
{
    if( !member.getName().equals
         ( member.getClassDeclaration().getName() )
      ) return false;
    ...
}
case( FieldDeclaration member ): { ... }
/* Ошибка: нет реализации для типа MethodDeclaration */
}
```

В приведенном примере определена операция `check` с параметром типа `ClassMemberDeclaration`. Для каждого неабстрактного типа, унаследованного от `ClassMemberDeclaration`, должна быть определена реализация операции.

3.4. Модульность описания абстрактного синтаксиса

При описании абстрактного синтаксиса расширения языка может быть использовано описание абстрактного синтаксиса расширяемого языка. Определения операций над расширением языка также могут использовать определения операций над расширяемым языком. Рассмотрим использование этой возможности на примере уже упоминавшегося правила декларации типов в языке Java:

```
JavaTypeDeclaration ::=   ClassDeclaration
                        | InterfaceDeclaration
                      ;
```

Фрагмент соответствующего модуля описания абстрактного синтаксиса на языке TreeDL выглядит следующим образом:


```
tree com.unitesk.java.Java2;

abstract node JavaTypeDeclaration { … }

node ClassDeclaration : JavaTypeDeclaration { … }

node InterfaceDeclaration : JavaTypeDeclaration { … }

operation string toString
  ( virtual JavaTypeDeclaration decl )
{
  case( ClassDeclaration decl ):
    { return "class " + …; }
  case( InterfaceDeclaration decl ):
    { return "interface " + …; }
}
```

Следующая версия языка добавляет новые альтернативы в это правило:

```
JavaTypeDeclaration ::=   ClassDeclaration
                        | InterfaceDeclaration
                        | EnumDeclaration
                        | AnnotationTypeDeclaration
                      ;
```

Модуль описания абстрактного синтаксиса для языка Java3 использует описание абстрактного синтаксиса предыдущей версии языка. Определения операций также используют ранее определенные операции.

```
tree com.unitesk.java.Java3 : com.unitesk.java.Java2;

node EnumDeclaration : Java2.JavaTypeDeclaration { … }

node AnnotationTypeDeclaration
  : Java2.JavaTypeDeclaration { … }

operation string toString
  ( virtual Java2.JavaTypeDeclaration decl )
  : Java2.toString
{
  case( EnumDeclaration decl ):
    { return "enum " + …; }
  case( AnnotationTypeDeclaration decl ):
    { return "@interface " + …; }
}
```

Как видно из приведенного примера, модуль описания абстрактного синтаксиса расширения языка содержит описания типов вершин только для

новых альтернатив. Определение операции `Java3.toString` содержит ветви только для новых типов вершин, для ранее определенных типов вершин используется реализация операции `Java2.toString`.

3.5. Генерация компонентов транслятора по описанию абстрактного синтаксиса

Обработка языка TreeDL осуществляется инструментом `treedl`. Этот инструмент выполняет следующие действия над описанием абстрактного синтаксиса:

- Проверку корректности.
- Генерацию гипертекстовой версии для использования в качестве документации.
- Трансляцию в язык программирования (в настоящее время поддерживаются языки Java и C#).
- Генерацию интерфейса и реализации *фабричных классов* [24], создающих вершины дерева.
- Генерацию интерфейса и базовых реализаций посетителей дерева:
    - Посетителя, который выполняет одно и то же действие над всеми вершинами дерева (в вырожденном случае – пустой посетитель). Этот посетитель является базовым классом для посетителей, обрабатывающих дерево. Для разработки посетителя, выполняющего одно и то же действие над всеми вершинами, достаточно переопределить один метод.
    - Посетителя, который осуществляет глубокое копирование поддерева. Наследники этого посетителя осуществляют трансформации деревьев.
- Генерацию интерфейса и некоторых реализаций обходчика дерева, в частности, обходчика, который производит обход дерева «в глубину» и выполняет при этом заданные действия.

Набор и схема генерации компонентов не фиксированы, инструмент `treedl` обладает открытой архитектурой, которая позволяет пользователю определять подключаемые модули для решения специфических задач анализа и генерации кода. Подключаемый модуль представляет собой Java класс, реализующий интерфейс подключаемого модуля. Инструмент предоставляет подключаемым модулям средства получения конфигурационных параметров и передачи диагностических сообщений. Для решения широко распространенной задачи генерации текста по дереву абстрактного синтаксиса предоставляется библиотека генерации текста на основе шаблонов.

3.6. Опыт использования описания абстрактного синтаксиса при разработке трансляторов

Различные версии описания абстрактного синтаксиса, послужившие основой для языка TreeDL, были использованы в Институте Системного Программирования РАН в 1995-2005 годах в проектах по созданию:

- Транслятора исполнимого подмножества языка спецификаций RSL в процедурный язык PROTEL, использующийся в компании Nortel [31].
- Трансляторов спецификационного расширения языков Java, C, C# в эти языки - проекты J@T, CTesK, Ch@se [32, 33].
- Инструмента генерации сложных структур данных (OTK) [34].
- Инструмента генерации тестов для парсеров формальных языков [35].
- Инструмента генерации тестов для языков программирования [36].
- Инструмента обработки языка TreeDL [25].

Практика показала, что язык TreeDL возможно использовать не только для описания абстрактного синтаксиса языков, но и для описания других моделей данных, имеющих иерархическую или графовую структуру:

- В проектах J@T и Ch@se язык TreeDL был использован для описания дескрипторов семантических сущностей языка.
- В проекте Ch@se язык TreeDL был использован для описания модели данных, создаваемых с помощью графического интерфейса пользователя, с последующей генерацией кода по этой модели.
- В самом проекте TreeDL язык был использован для описания структуры классов, представляющих сообщения об ошибках. Это позволило отделить модель сообщений об ошибках от их представления и реализовать локализацию текста сообщений об ошибках с использованием шаблонов, хранящихся в ресурсах приложения.
- В проекте OTK язык TreeDL был использован для описания моделей для генератора сложных структур данных.

4. Заключение

В статье описан способ описания структуры дерева абстрактного синтаксиса и операций над ним, который может быть использован при разработке трансляторов. Для описания абстрактного синтаксиса используется специализированный язык TreeDL.

В сводной таблице приведены характеристики способов использования абстрактного синтаксиса при разработке трансляторов.

	Cocktail	SableCC	TreeCC	TreeDL
Способ описания абстрактного синтаксиса	На основе контекстно-свободных грамматик		Аналогично классам с одиночным наследованием	
Модульность описания абстрактного синтаксиса	+[5]	–	+[1]	+
Генерация структуры абстрактного синтаксиса	C, C++, Module-2, Java	Java	C, C++, Java, C#	Java, C#[6]
Поддержка шаблона Посетитель	–	+	+	+
Генерация обходчиков дерева	+	+	–	+
Генерация фабричных классов	–	–	–	+
Поддержка сериализации/десериализации	+	–	–	–[7]
Операции над деревьями с помощью мультиметодов	–	–	+	+
Поддержка трансформации деревьев	+[8]	–	–	–[9]
Пользовательский код	–	–	+	+
Расширяемость	–	–	+[10]	+
Лицензия	Коммерческая	LGPL	GPL	BSD

[5] С помощью включения файлов
[6] Трансляция в другие языки может быть реализована подключаемыми модулями.
[7] При необходимости может быть реализовано подключаемым модулем.
[8] Для определения трансформаци деревьев на основе шаблонов используется отдельный инструмент Puma.
[9] Есть поддержка копирования деревьев, на основе которой можно реализовать трансформации.
[10] Процесс расширения, описанный разработчиками инструмента, требует модификации исходных текстов инструмента.

Как следует из таблицы, описание абстрактного синтаксиса с использованием TreeDL предоставляет большинство необходимых возможностей, как и другие способы описания абстрактного синтаксиса. Открытая архитектура инструмента `treedl` позволяет реализовать дополнительные возможности необходимые пользователю, в частности, поддержку дополнительных целевых языков.

В перечисленных проектах описание абстрактного синтаксиса на языке TreeDL в 5-10 раз меньше соответствующего кода на языке Java. Эта оценка не включает дополнительные компоненты, сгенерированные по описанию абстрактного синтаксиса: интерфейсы и стандартные реализации посетителей, обходчиков и т.п. Использование других способов описания абстрактного синтаксиса (Cocktail/AST, TreeCC) приводит к аналогичным показателям.

Описание абстрактного синтаксиса с использованием TreeDL обладает следующими достоинствами:

- Явное описание абстрактного синтаксиса позволяет четко определить интерфейс взаимодействия основных подсистем транслятора, вести работы по реализации этих подсистем одновременно и сократить время, необходимое для их интеграции.
- Все компоненты, реализация которых определяется представлением абстрактного синтаксиса, могут быть сгенерированы автоматически. При этом пользователь может изменить схему генерации существующих видов компонентов и задать способ генерации необходимых ему дополнительных компонентов.
- Для определения операций над деревьями используются специализированные средства, позволяющие повысить надежность транслятора за счет выявления несоответствий во время трансляции определения операции, а не во время её выполнения.
- Модульность описания абстрактного синтаксиса позволяет при реализации расширения языка повторно использовать компоненты транслятора расширяемого языка.

В настоящее время ведется создание библиотеки готовых распознавателей для распространенных языков с использованием TreeDL для описания абстрактного синтаксиса. Ставится задача исследовать, какая информация должна присутствовать во внутреннем представлении, чтобы распознаватель языка можно было повторно использовать для решения широкого круга задач. Опыт практического использования подхода позволяет утверждать, что такая библиотека будет полезна разработчикам трансляторов.

Литература

1. Ахо А., Сети Р., Ульман Д., *Компиляторы: принципы, технологии и инструменты*. М.: Издательский дом "Вильямс", 2001, ISBN-8459-0189-8.
2. Kenneth Slonneger, Barry L. Kurtz, *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach,* Addison-Wesley, 1995, ISBN 0-201-65697-3.

3. В. А. Серебряков, М. П. Галочкин, *Основы конструирования компиляторов*, Едиториал УРСС, 2001, ISBN 5-8360-0242-8. http://www.citforum.ru/programming/theory/serebryakov/
4. Domain-Specific Languages: An Annotated Bibliography http://homepages.cwi.nl/~arie/papers/dslbib/
5. Содружество РЕФАЛ/Суперкомпиляция http://refal.net
6. The Caml Language http://caml.inria.fr
7. The TXL Programming Language http://www.txl.ca
8. Dwight VandenBerghe. *Why ML/OCaml are good for writing compilers* http://flint.cs.yale.edu/cs421/case-for-ml.html
9. Martin Fowler. *Language Workbenches: The Killer-App for Domain Specific Languages?* http://martinfowler.com/articles/languageWorkbench.html
10. Polyglot: A compiler front end framework for building Java language extensions, http://www.cs.cornell.edu/Projects/polyglot/
11. The extensible Java pre-processor EPP, http://staff.aist.go.jp/y-ichisugi/epp/
12. RECODER: Java framework for source code metaprogramming, http://recoder.sourceforge.net/
13. http://www.semanticdesigns.com
14. F. Pfenning, C. Elliot, *Higher-order abstract syntax*, Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation, 1988. ISSN:0362-1340.
15. UML™ Resource Page http://www.uml.org
16. XML Metadata Interchange http://www.omg.org/technology/documents/formal/xmi.htm
17. Human-Usable Textual Notation http://www.omg.org/technology/documents/formal/hutn.htm
18. Relax NG home http://relaxng.org
19. ANother Tool for Language Recognition http://www.antlr.org
20. Leonidas Fegaras, *Gen: A Java Package for Constructing and Manipulating Abstract Syntax Trees* http://lambda.uta.edu/cse5317/notes/node26.html in *CSE 5317/4305: Design and Construction of Compilers, University of Texas at Arlington, CSE* http://lambda.uta.edu/cse5317/notes/nodes.html
21. Etienne M.Gagnon, Laurie J.Hendren, *SableCC, an Object –Oriented Compiler Framework*, McGill University, Quebec, Canada, 1998, http://sablecc.org
22. Cocktail toolbox http://www.cocolab.com/en/cocktail.html
23. Rhys Weatherley, *TreeCC: An Aspect-Oriented Approach to Writing Compilers,* 2001, http://www.southern-storm.com.au/treecc.html
24. Джон Влиссидес, Ральф Джонсон, Ричард Хелм, Эрих Гамма, *Приемы объектно-ориентированного проектирования. Паттерны проектирования*. Питер, 2003, ISBN 5-272-00355-1.
25. The MultiJava Project http://multijava.sourceforge.net
26. Java™ Language Specification (2nd Edition) http://java.sun.com/docs/books/jls/index.html
27. Java™ Language Specification, The (3rd Edition) http://java.sun.com/docs/books/jls/index.html

28. TreeDL project home http://treedl.sourceforge.net
29. Manuel Fähndrich K. Rustan M. Leino, *Declaring and Checking Non-null Types in an Object-Oriented Language*, OOPSLA'03, October 26–30, 2003, Anaheim, California, USA.
30. IntelliJ® IDEA http://www.jetbrains.com/idea/
31. Демаков А.В. *Исполнимое подмножество языка спецификации и его трансляция*. Приложения системного программирования: Вопросы кибернетики, Научный совет по комплексной проблеме «Кибернетика» РАН, Москва, 1998, вып.4, 17-28.
32. В. В. Кулямин, А. К. Петренко, А. С. Косачев, И. Б. Бурдонов. *Подход UniTesK к разработке тестов*. Программирование, 29(6):25-43, 2003.
33. I. B. Bourdonov, A. V. Demakov, A. A. Jarov, A. S. Kossatchev, V. V. Kuliamin, A. K. Petrenko, and S. V. Zelenov. *Java Specification Extension for Automated Test Development*. Proceedings of PSI'01. LNCS 2244, pp. 301-307. Springer-Verlag, 2001
34. С.В. Зеленов, С.А. Зеленова, А.С. Косачев, А.К. Петренко. *Генерация тестов для компиляторов и других текстовых процессоров*, Программирование, Москва, 2003, т.29, №2.
35. А.В.Демаков, С.А.Зеленова, С.В.Зеленов*, Тестирование парсеров текстов на формальных языках*. Программные системы и инструменты: Тематический сборник факультета ВМиК МГУ, Москва, 2001, вып. 2, 150-156.
36. Архипова М.В. *Генерация тестов для семантических анализаторов*. Препринт ИСП РАН, 2005