УДК 004.432

БИБЛИОТЕЧНАЯ РЕАЛИЗАЦИЯ ВЫЧИСЛИТЕЛЬНОЙ МОДЕЛИ ЯЗЫКА ПЛЭНЕР

© 2008 г. О. Г. Фролова

intafy@gmail.com

Кафедра Алгоритмических языков

1 Введение

В настоящее время для проектирования и разработки крупных программных проектов в большинстве случаев используются индустриальные языки программирования, которые, как правило, являются объектно-ориентированными. Такими языками, к примеру, являются C++, C#, Java.

В то же время при разработке подобных проектов могут возникать разнообразные подзадачи, для которых применение традиционных языков неэффективно с точки зрения использования труда программиста. Примерами таких задач могут служить задачи искусственного интеллекта, компьютерной лингвистики, задачи, связанные с преобразованиями деревьев, графов, математических формул и т. п. При этом они могут быть достаточно хорошо решены средствами языков функционального или логического программирования. Но эти языки, в свою очередь, либо менее удобны для задач описания графических интерфейсов, взаимодействия с операционной системой, либо оказываются недостаточно эффективными по времени работы или расходу памяти. Одним из возможных подходов к разработке сложных программных систем является одновременное использование различных языковых средств и различных парадигм программирования.

Planner является одним из наиболее известных языков программирования для искусственного интеллекта. Этот язык был разработан Карлом Хьюиттом в Лаборатории искуственного интеллекта Массачусетского Технологического Института, первые публикации о нем появились в 1969 г. [1]

Этот язык включает в себя в качестве подмножества модификацию языка Лисп, расширяя его возможностями анализа данных по образцам, поиска с возвратами, работы со встроенной базой данных, дедуктивным механизмом.

Таким образом, Planner ориентирован на задачи обработки символьной информации, решаемые методами перебора и дедуктивных выводов. Язык является удобным средством для решения задач искусственного интеллекта, таких, как планирование действий робота, доказательство теорем, вопросно-ответные системы, понимание естественного языка.

Для реализации был выбран диалект, разработанный на кафедре Алгоритмических языков факультета ВМК МГУ и получивший название Плэнер [7], [9]. Он представляет собой некоторое упрощение оригинального варианта языка. В частности, в нем отсутствуют параллельные процессы, изменен синтаксис ряда встроенных процедур, ограничено использование сопоставителей — процедур, осуществляющих расширенное сопоставление с образцом. Такие изменения упрощают язык и позволяют его эффективнее реализовывать и использовать, но практически не сужают область применения языка.

2 Метод непосредственной интеграции и библиотека InteLib

Объединение различных стилей программирования возможно, например, с помощью метода непосредственной интеграции [3]. Согласно этому методу в качестве основного средства разработки проекта выбирается объектно-ориентированный язык программирования, предоставляющий возможность переопределения символов арифметических операций. Семантики требуемых альтернативных языков рассматриваются как специфичные предметные области,

которые моделируются средствами основного языка в виде объектно-ориентированных библиотек классов. При этом определяются специальные пользовательские типы и некоторые инфиксные операции над ними, что позволяет промоделировать изобразительные средства вспомогательных языков в виде арифметических выражений базового.

Таким образом, поддержка функциональности вспомогательного языка реализуется библиотекой классов, которую можно использовать для написания частей большого программного продукта. Код, написанный с использованием такой библиотеки, получается синтаксически похожим и семантически эквивалентным коду на вспомогательном встраиваемом языке, оставаясь при этом программой на основном языке разработки проекта.

Метод непосредственной интеграции реализован в проекте InteLib [6], [4]. В качестве основного языка в библиотеке InteLib выбран язык C++. В настоящее время библиотека предоставляет поддержку языков Lisp, Scheme, Refal. Библиотека реализует S-выражения [10] как гетерогенные структуры данных [5] и развитые средства работы с ними.

Базовым классом библиотеки является абстрактный класс `SExpression`. На его основе построена иерархия наследования, моделирующая S-выражения разных типов, например, точечные пары (класс `SExpessionCons`), целочисленные константы (класс `SExpressionInt`), числа с плавающей точкой (класс `SExpressionFloat`), символы-метки (класс `SExpressionLabel`) и некоторые другие. В библиотеке реализованы «умные» указатели на S-выражения (класс `SReference`) и метод счетчика ссылок для сборки мусора.

Для реализации вычислительной модели языка Плэнер в рамках библиотеки InteLib были разработаны структуры данных для представления конструкций языка на основе S-выражений и с их помощью реализованы основные возможности языка.

3 Особенности реализации

3.1 Структуры данных

В Плэнере, как и в Лиспе, для изображения программ и данных используется единственный вид конструкций: выражения, которые делятся на атомарные и списковые. В отличие от Лиспа в Плэнере используются три типа списков. Синтаксически они отличаются по виду скобок, которыми ограничены элементы. Списки с круглыми скобками принято называть L-списками, списки с квадратными скобками — P-списками, списки с угловыми скобками — S-списками.

Поскольку синтаксис L-списков в Плэнере совпадает с синтаксисом списков языка Лисп, для представления L-списков используется имеющийся в библиотеке InteLib класс `SExpressionCons`. Для представления двух других типов списков вводятся дополнительные структуры. P-списки представляются классом `PlnExpressionCons`. Вычисление S-списков отличается от P-списков только тем, что результат вычисления должен быть при дальнейшем его использовании сегментирован. Аналогичным свойством обладают и сегментные обращения к переменным. Поэтому для представления всех сегментированных выражений было решено использовать объекты одного общего класса `PlnExpressionSegmented`. Он содержит внутри себя «умный» указатель на простую форму, соответствующую представляемой сегментной форме.

В основе библиотеки InteLib, кроме классов иерархии S-выражений, лежит класс `SReference` — «умный» указатель на S-выражения. От него был унаследован класс `PlnReference` — «умный» указатель на S-выражения, соответствующие объектам Плэнера, имеющий специальные методы для вычисления плэнерских форм и проверки списков различных типов на пустоту. В нем перегружен оператор «,», который теперь используется для создания списков: левый операнд должен быть списком одного из трех типов, к этому списку в хвост дописывается новый элемент — правый операнд. При этом действие оператора «,» различается в зависимости от типа левой части, чтобы получить корректные плэнерские списки. Таким образом, если объект `list` класса `PlnReference` является «умным» указателем на объект класса `PlnExpressionCons`, соответствущий некоторому списку в квадратных скобках, то результатом вычисления выражения языка C++ `(list, 5)` станет снова «умный» указатель на список в квадратных скобках.

Для более удобного конструирования списков были введены два новых класса: `PlnLListConstructor` и `PlnPListConstructor`. Для них перегружен оператор «|», который

возвращает список в круглых или квадратных скобках из одного элемента — своего правого операнда. Кроме того, в классе `PlnReference` перегружен оператор «!», сегментирующий его.

В Плэнере идентификаторы вычисляются сами в себя, а для обращения к локальным и глобальным переменным используются шесть типов префиксов. Для представления идентификаторов был создан класс `PlnExpressionLabel`, объекты которого хранят в себе имя идентификатора и значение соответствующей константы или функции. Эти же объекты связываются со значениями локальных переменных в лексическом контексте. Классы для представления обращений к переменным являются «обертками» `PlnExpressionLabel` и различаются только способом вычисления.

Класс `PlnSymbol` является «умным» указателем на `PlnExpressionLabel` и был введен ради перегружаемых в нем операций. Сам по себе объект этого класса соответствует идентификатору языка Плэнер, а унарные операции «-», «~» и «*», примененные к нему, возвращают объекты классов, соответствующие обращениям к переменным с префиксами «:», «.» и «*». За счет использования оператора «!» для последующего сегментирования обращения к переменной выражения языка C++ получаются внешне достаточно похожими на конструкции Плэнера.

Таким образом, если создать объекты L и P классов `PlnLListConstructor` и `PlnPListConstructor` соответственно и объекты класса `PlnSymbol` для каждого используемого в программе идентификатора, то можно будет провести следующие аналогии между выражениями на языке C++ и выражениями на языке Плэнер:

Выражение C++	Выражение Плэнера
`(L| 2, 3, 4)`	`(2 3 4)`
`(L| 2, ~X, !~Y)`	`(2 .X !.Y)`
`(P| EQ, 3, 4)`	`[EQ 3 4]`
`(P| IS, (L| 2, *Y, 4), ~X)`	`[IS (2 *Y 4) .X]`
`!(P| 2, 3, 4)`	`<2 3 4>`

3.2 Вычисление выражений и режим возвратов

В языке Плэнер существует возможность режима возвратов (бэктрекинга): в программе можно поставить *развилку*, организующую перебор. Если в дальнейшем вычисление заходит в тупик, то вырабатывается *неуспех*, программа полностью восстанавливает свое состояние на момент развилки, выбирает другой вариант вычислений и продолжает свою работу. То есть для реализации режима возвратов необходима возможность в любой момент восстановить ранее сохраненное состояние программы и продолжить вычисление с восстановленной точки. Поэтому, в отличие от Лиспа, вычисление выражений Плэнера нельзя реализовывать рекурсивно.

Для итеративного вычисления выражений было решено промоделировать вычисление с помощью модифицированного стека функций, существовавшее в реализации Плэнера для системы БЭСМ-6 [8]. В этой реализации фреймы функций, устанавливающих точку развилки, помечались особым способом, и ни они, ни фреймы, расположенные выше по стеку, не удалялись из стека после возврата из них. При установке развилки сохранялось полное состояние программы на тот момент, не исчезающее даже после завершения функции, установившей развилку. При дальшейшем возникновении в программе неуспеха происходил переход на существующий фрейм, хранящий правильный лексический контекст и точку возврата. Отдельно хранились обратные операторы, которые должны были быть выполнены при возврате для восстановления корректного состояния программы.

Для моделирования такого вычисления в рамках библиотеки классов языка C++ был создан класс `PlnEvaluator`, полностью отвечающий за вычисление выражений. Он хранит в себе указатель на двусвязный список фреймов функций, указатели на текущий и последний фреймы этого списка, список развилок, связанных с данным стеком и результат последнего вычисления (успешный либо неуспешный).

Каждый фрейм содержит в себе вычисляемое выражение, лексический контекст, массив для хранения вычисленных аргументов функций и адрес возврата, который может не совпадать с адресом предыдущего фрейма, поскольку некоторые уже завершившиеся фреймы не удаляются из стека.

При реализации перебора некоторые участки программы могут вычисляться несколько раз, то есть каждая функция должна уметь продолжать свое вычисление с конкретной точки, независимо от того, сколько аргументов уже было вычислено. Для этого фрейм функции хранит число — номер параметра, которым это выражение является. При вычислении аргументов функции или элементов списка для каждого i-го элемента создается новый стековый фрейм, и ему на хранение передается число i. Когда фрейм возвращает управление выражению, создавшему его, он, кроме результата своего вычисления, возвращает и это число. Изменяя это число при возврате к фрейму по несупеху, можно заставить функцию повторно вычислить какой-то ее аргумент.

Работа с развилками осуществляется через класс `PlnEvaluator`, позволяющий добавить развилку, соответствующую текущему вычисляемому фрейму, или удалить последнюю развилку. В последнем случае активной развилкой становится предыдущая, и при возникновении неуспеха управление будет передано ее фрейму.

При откате по неуспеху программа должна полностью восстановить свое состояние, сохраненное при установке последней развилки. При этом восстановление вычисляемого выражения, его адреса возврата и номера параметра, который оно должно будет вернуть родительскому фрейму, осуществляется за счет того, что нужный фрейм не удаляется из стека. Но значения локальных и глобальных переменных могли быть изменены на неуспешной ветви вычисления. Поэтому все действия программы на неуспешном пути запоминаются, и при откате выполняются противоположные действия. В каждом объекте класса `PlnStackFork` хранится список *обратных операторов* — действий, которые должны быть выполнены при откате к этой развилке. При возврате к развилке необходимо перебрать элементы списка и последовательно восстановить старые значения переменных, хранящиеся в них.

При такой реализации для вычисления выражения языка Плэнер, записанного в виде конструкций языка C++, в программе должен быть создан объект класса `PlnEvaluator`, соответствующий стеку фреймов функций. При вызове метода `Evaluate` этого класса с нужным выражением, представленным объектом класса `PlnReference`, в качестве параметра, в списке создается новый фрейм, соответствующий этому выражению, и начинается циклическое вычисление фреймов стека. Поскольку в процессе вычисления возможно создание новых и удаление отработавших фреймов, то цикл будет работать, пока из стека не будет удален фрейм исходного выражения, выдав конечный результат. При этом метод `Evaluate` никогда не будет вызван рекурсивно.

При вычислении выражения, находящегося в некотором стековом фрейме, используются результат последнего вычисления, количество вычисленных параметров (при нормальном ходе вычисления без развилок оно совпадает с количеством попыток вычисления данного фрейма), значения вычисленных параметров. Само вычисление осуществляется следующим образом:

- L-список: если количество вычисленных параметров меньше, чем длина списка, то в стек добавляется фрейм для вычисления очередного элемента списка, иначе из вычисленных значений формируется результирующий список с учетом возможной сегментированности элементов.

- P-список: первым элементом в нем может быть либо имя функции, являющееся идентификатором, либо обращение к переменной с префиксом «.» или «:». В последнем случае сразу же берется значение переменной из ее идентификатора или лексического контекста фрейма. Полученный таким образом идентификатор содержит значение — объект класса, представляющего функциональные объекты. Далее вычисление зависит от конкретного функционального объекта. При вычислении функций создаются новые стековые фреймы для параметров, которые должны быть вычислены, создается новый лексический контекст, в котором осуществляется связывание формальных параметров с фактическими, и либо добавляется новый фрейм для тела функции, либо вызывается метод, содержащий тело функции в виде кода на C++.

- «.»-переменная: возвращается S-выражение, связанное с соответствующим ей идентификатором в лексическом контексте, хранящемся в фрейме вычисления.
- «:»-переменная: возвращается S-выражение, хранящееся внутри идентификатора, соответствующего переменной.
- Сегментированные выражения не могут быть вычислены, попытка их вычисления вызывает ошибку.
- Для всех остальных типов S-выражений значениями являются они сами.

Если вычисление значения выражения, соответствующего текущему стековому фрейму, было успешно завершено, то возвращается результат вычисления и число — номер вычисленного параметра, которые затем могут быть использованы следующим вычисляемым фреймом. Затем выполняется проверка, можно ли удалять вычисленный фрейм из стека, и если ниже него не было фреймов с развилками, то он удаляется. Текущим становится фрейм, находившийся в адресе возврата вычислившегося.

Если же в результате вычисления выражения возник неуспех, то никакой результат не возвращается и происходит откат к последней развилке: ее фрейм становится текущим, а все, находящиеся в стеке после него, удаляются. При откате последовательно выполняются все сохраненные обратные операторы, которые затем удаляются.

3.3 Сопоставление выражения с образцом

В языке Плэнер сопоставление бывает двух типов: выражения с образцом и образца с образцом. Первое сопоставление обычно осуществляется при вызове функции `IS` или при поиске утверждения в базе данных. Второе возникает только при вызове теорем по образцу.

При сопоставлении выражения с образцом бывает необходимо вызывать функции и *сопоставители* — процедуры, осуществляющие расширенное сопоставление. При вычислении функций могут возникать неуспехи, которые должны быть отслежены, и создаваться обратные операторы, причем эти операторы должны быть сохранены даже при удачном сопоставлении. Поэтому сопоставление с образцом необходимо реализовывать тоже итеративно, с использованием стековых фреймов.

При реализации были введены новые типы стековых фреймов, отвечающие за сопоставление выражения с образцом. При сопоставлении выражения-списка с образцом, содержащим сегментированные выражения, возникает необходимость перебора элементов списка, чтобы найти сегмент, соотвествующий элементу образца. Поэтому для оптимизации сопоставления было решено использовать два типа фреймов: один, вычисление которого достаточно просто и быстро, для сопоставления атомарных выражений, и второй, в котором уже может возникать перебор, для L-списков. Сам перебор организован на основе режима возвратов: если выбранный в какой-то момент сегмент списка приводит к неудачному сопоставлению, в программе возникает неуспех, осуществляется возврат на фрейм, выбравший этот сегмент, и сопоставление продолжается с увеличенным сегментом списка.

Такое решение не вполне корректно, так как с точки зрения языка режим возвратов и механизм сопоставления являются совершенно независимыми, но оно позволяет упростить сопоставление. Перебор элементов списка должен осуществляться в рамках циклического вычисления стековых фреймов, поэтому, если не использовать имеющуюся реализацию режима возвратов, потребовалось бы создать полностью аналогичный ему механизм перебора, то есть задублировать функциональность бектрекинга, либо использовать механизм исключений языка C++, что сильно ухудшило бы производительность. Кроме того, гарантируется, что неуспех, возникший при неудачном переборе, будет обработан фреймом-сопоставителем или функцией, начавшей осуществлять сопоставление, то есть механизмы возвратов и сопоставления оказываются связанными только на уровне реализации, но не на уровне семантики программы на языке Плэнер.

Во фреймах, осуществляющих сопоставление атомарных выражений, может возникнуть обращение к функции-сопоставителю. Такое обращение обрабатывается как вычисление обычной функции. Разница заключается только в том, что для тела функции создается фрейм сопоставления, а не вычисления, и к списку фактических параметров функции, кроме значений,

соответствующих формальным параметрам, добавляется выражение, свойства которого должна проверить функция-сопоставитель. То есть функциям-сопоставителям всегда передается не меньше одного аргумента, что особенно важно при реализации встроенных сопоставителей на языке C++.

3.4 Встроенная база данных

Встроенная база данных языка Плэнер является по сути просто набором утверждений (L-списков, состоящих из атомов) с возможностью выбора тех из них, которые соответствуют конкретному образцу. Поэтому при имеющемся сопоставлении выражения с образцом реализация базы данных требует только написания трех основных встроенных функций для работы с ней: добавляющей новое выражение в массив утверждений, удаляющей выражение из массива и функции поиска. Функция поиска по базе данных должна перебирать все имеющиеся утверждения и последовательно сопоставлять их с заданным образцом (добавлять в стек фрейм для такого сопоставления). При успешном сопоставлении она должна завершаться, при неуспешном — откатывать все изменения, которые могли быть сделаны при предыдущем сопоставлении, и выбирать следующее утверждение.

Для оптимизации перебора утвержений из базы данных, которых может быть достаточно много, массив с ними хранится в классе-контейнере `DataBase`, имеющем метод `SearchSimilar(pattern)` для выбора утверждений, которые могут соответствовать образцу. Этот класс не имеет доступа к стеку вычисления функций и поэтому не может осуществить полное сопоставление выражений с образцом, но он производит частичное сопоставление: проверяет соответствие длин образца и утверждения; если в образце заданы атомы, то проверяется наличие этих атомов в утверждении. После проверки таких мягких условий список утверждений сокращается, и впоследствии требуется осуществлять полное сопоставление меньшего числа выражений.

3.5 Вызов процедур по образцу

В языке Плэнер существует еще один вид процедур — *теоремы*. Они отличаются от обычных функций тем, что вызываются не по имени, а по образцу, который играет роль и имени, и набора формальных параметров. В программе может существовать несколько процедур с одинаковыми или похожими образцами, что позволяет перебирать их по очереди, применяя ту, которая сумеет найти решение.

Любая теорема вычисляется так же, как и встроенная функция `PROG`, то есть у нее есть список локальных переменных и тело — набор последовательно вычисляющихся выражений. Поэтому вычисление уже выбранной теоремы реализовывается довольно просто на основе уже введенных конструкций.

Основная сложность при реализации теорем заключается в организации сопоставления образцов между собой. И вызывающий образец, и образец теоремы являются полноценными образцами, поскольку, например, и в одном, и в другом могут содержаться обращения, меняющие значения переменных: в одном для получения результата работы теоремы, в другом — для передачи параметров.

Сопоставление образцов между собой является в общем случае алгоритмически неразрешимой задачей [9]. Как минимум, если в каждом образце встречаются обращения к процедурам-сопоставителям, то их корректное сопоставление аналогично проверке эквивалентности алгоритмов. Если запретить использование сопоставителей, то все равно возникают проблемы из-за наличия сегментных образцов. Поэтому обычно для реализации такого сопоставления на образцы накладывают ограничения.

В описании языка Planner [2] эта проблема реализации никак не регламентируется. В диалекте Плэнер в образцах, сопоставляемых друг с другом, запрещено использовать обращения к любым процедурам (и функциям, и сопоставителям) и сегментные образцы, при этом вложенность списков и простые обращения к переменным не ограничиваются. Этого достаточно и для решения большинства задач, и для эффективной реализации.

Другой проблемой, возникающей при сопоставлении образцов, является осуществление связывания переменных. Если сопоставляются две переменные, ни одна из которых не имеет зна-

чения, между ними должна возникнуть связь: если в дальнейшем одна из этих переменных получит значение, то это значение должно быть присвоено и другой переменной. Более того, если переменная без значения сопоставляется со списком, среди элементов которого есть другая переменная, то на значение первой переменной накладываются ограничения. Например, если производится сопоставление `[MATCH *X (A *Y B)]`, то на переменную X будет наложено ограничение, что ее значение является списком, первый и третий элементы которого определены, а второй должен будет совпадать со значением переменной Y.

Эти ограничения можно рассматривать как каркас значения переменной, «полуфабрикат» [8]. Если ввести новые структуры данных для хранения этого полусформированного значения, то можно сделать его недоступным пользователю системы, но использовать в сопоставлениях наравне с другими выражениями. При этом проверка непротиворечивости ограничений на значение переменной сводится к сопоставлению имеющегося у переменной значения-«полуфабриката» с новым ограничением.

Для реализации таких «полуфабрикатов» вводится класс `PlnHalfFormed`, содержащий внутри себя «умный» указатель на какое-либо выражение. Если в каком-либо объекте этого класса этот указатель пустой, то объект считается неопределенным и может впоследствии получить любое значение. Если же указатель указывает на выражение-список, то у этого списка тоже могут быть не окончательно сформированные элементы. При сопоставлении переменных производится сопоставление таких полусформированных значений. При сопоставлении переменных, не имеющих ни настоящего, ни полусформированного значения, создается новый объект класса `PlnHalfFormed`, который становится значением обеих переменных. Таким образом осуществляется связь между ними. Если же программе необходимо получить настоящее значение переменной, например, являющейся аргументом функции, то проверяется, указывает ли указатель из объекта класса `PlnHalfFormed` на полностью сформированное выражение, и если да, то значение, связанной с переменной обновляется.

Следует отметить, что полусформированные значения могут быть только у локальных переменных, так как использование глобальных переменных в сопоставлениях полностью равносильно использованию их значений. Глобальные переменные не могут получить или изменить свое значение ни при каком сопоставлении.

4 Пример использования библиотеки

Полученная библиотека классов позволяет записать программу на языке Плэнер в виде набора выражений языка C++ и вычислить ее. Рассмотрим пример программы на Плэнере, которая вводит новый сопоставитель, проверяющий, является ли список палиндромом:

```
[DEFINE PALYNDROM (KAPPA ()
    [AUT ()                                ; либо пустой список
         [LIST 1]                          ; либо список из одного элемента
         [SAME (X) (*X <PALYNDROM> .X)]    ; либо палиндром
    ]
)]

[IS [PALYNDROM] (A 5 A)]                   ; -> T
```

Аналогичная программа на языке C++ с использованием библиотеки:

```
// объект, осуществляющий вычисления
PlnEvaluator stack;

// объекты, конструирующие списочные выражения
PlnLListConstructor L;
PlnPListConstructor P;

// используемые в программе идентификаторы
PlnSymbol PALYNDROM("PALYNDROM"), X("X"), A("A");
```

```
stack.Evaluate(
(P| DEFINE, PALYNDROM, (L| KAPPA, ~L,
     (P| AUT, ~L,
              (P| LIST, 1),
              (P| SAME, (L| X), (L| *X, !(P| PALYNDROM), ~X))
     )
)));

PlnReference result = stack.Evaluate(
    (P| IS, (P| PALYNDROM), (L| A, 5, A)));
```

После работы этой программы переменная `result` будет содержать в себе S-выражение, соответствующее символу `T`.

Объекты класса `PlnSymbol` должны создаваться для всех идентификаторов, имеющихся в программе, кроме описанных в библиотеке. К последним относятся атомы `T`, `LAMBDA`, `KAPPA`, `ANTEC`, `CONSEQ` и `ERASING`. Атом `T` появляется в библиотеке как результат работы функций, возвращающих логические значения. Остальные пять атомов используются функцией `DEFINE` для определения типа создаваемой процедуры.

Символы `DEFINE`, `IS`, `AUT`, `SAME`, `LIST` являются именами встроенных функций и сопоставителей и тоже вводятся в библиотеке.

5 Вспомогательное программное обеспечение

Созданные средства конструирования и вычисления выражений языка Плэнер позволяют реализовать интерактивный интерпретатор языка. Ядром интерпретатора является класс `IntelibPlnLoop`, осуществляющий основной цикл работы: считывание выражения из файла или потока стандартного ввода, вычисление полученного выражения с помощью объекта класса `PlnEvaluator` и вывод текстового представления результата на экран.

Считывание выражения и его лексический и синтаксический анализ осуществляются с помощью средств, предоставляемых библиотекой InteLib. В библиотеке имеется класс `IntelibGenericReader`, отвечающий за синтаксический анализ языков, основанных на S-выражениях. Он может быть перепрограммирован для конкретного языка с учетом его особенностей. В частности, для реализации чтения выражений языка Плэнер от этого класса наследуется класс `PlannerReader`, в конструкторе которого указывается, что символом комментария является «;»; списки могут быть ограничены скобками вида (), [] и <> (для каждого типа скобок указывается конкретная функция, умеющая формировать из отдельных элементов списка требуемые структуры данных); префиксы перед идентификаторами должны сформировать из них обращения к соответствующим переменным.

Благодаря перегруженным операциям и введенным классам для конструирования списков языка Плэнер, выражения на языке C++ являются синтаксически похожими на выражения Плэнера. Тем не менее, синтаксис двух языков отличается, и недостаточно просто написать подпрограмму на языке Плэнер для включения ее в большой проект. Ее надо переписать с учетом синтаксиса языка C++, что может быть неудобно из-за большого количества вспомогательных символов.

Поэтому удобным средством является транслятор программы на языке Плэнер в модуль на языке C++. При трансляции выражения Плэнера переводятся в аналогичные им конструкции, вдобавок в модуль добавляются все необходимые объявления переменных. Создаются объекты класса `PlnEvaluator` для осуществления вычисления, классов `PlnPListConstructor` и `PlnLListConstructor` для конструирования списков, класса `PlnSymbol` для всех используемых в программе идентификаторов.

Модуль на языке C++, содержащий программу языка Плэнер, генерируется автоматически этим транслятором, но тем не менее может быть прочитан и понят человеком, поскольку выражения конструируются все теми же удобными синтаксическими способами. Он может быть

подключен к основному проекту на языке C++, а остальные модули этого проекта требуют написания минимального количества кода с использованием библиотеки. В большинстве ситуаций будет достаточно написать несколько P-списков с вызовами функций, реализованных в программе Плэнера, и получить результат их работы в виде S-выражений.

Транслятор сам написан на языке Плэнер. Для его запуска можно использовать описанный выше интерпретатор. При этом из текста самого транслятора тоже можно создать модуль на языке C++. Таким образом, запуская транслятор из-под интерпретатора и применяя его к собственному коду, можно получить модуль на языке C++.

Основная программа, написанная на C++, анализирует переданные ей через командную строку параметры, среди которых, кроме имени транслируемого файла, могут быть управляющие директивы, инициализирует модуль транслятора и вызывает его основную функцию, передавая ей в качестве параметра имя файла.

Результатом компиляции основной программы вместе со сгенерированным модулем является исполняемый файл, являющийся транслятором из языка Плэнер в код на языке C++. Этот файл с одной стороны использует код на языке Плэнер для своей работы, но с другой — работает намного быстрее, чем транслятор, запущенный из-под интерпретатора.

Работа самого транслятора состоит из следующих шагов:

1. Считывание выражений программы на языке Плэнер, одновременно с этим происходит обработка директив.

2. Первый проход по программе: имена идентификаторов преобразовываются в соответствии с правилами замены символов, все используемые в программе идентификаторы заносятся в общую таблицу.

3. Генерация заголовочного файла. В заголовочном файле создается класс, инкапсулирующий объявления переменных для идентификаторов программы. Для каждого выражения в классе объявляется отдельный метод, в котором и будет происходить его вычисление. Конструктор этого класса должен поочередно вызвать эти методы, выполнив таким образом всю программу. Объявляется функция инициализации модуля

4. Второй проход: генерация файла реализации. Выражения языка Плэнер преобразуются в соответствующие им конструкции языка C++. Каждое выражение вычисляется объектом класса `PlnEvaluator` в соответствующем ему методе. Генерируется код для конструктора класса, вызывающего методы для всех выражений, и функции инициализации модуля. При вызове этой функции из основной программы будет создан объект класса, описанного выше, вызван его конструктор и вычислится подключаемая программа Плэнера.

При первом проходе по тексту программы транслятором осуществляется замена имен идентификаторов, если это необходимо. Алфавит языка Плэнер намного шире алфавита языка C++. При этом для любого идентификатора, используемого в программе, должна существовать переменная класса `PlnSymbol`, имя которой должно соответствовать правилам записи идентификатором языка C++. Так что допустимые в программе на Плэнере имена `=A+B=`, `*`, `A+-`, `/Zzz` должны быть преобразованы к другому виду, причем однозначным образом.

Для такого преобразования был использован способ, применяющийся в библиотеке InteLib для идентификаторов языков Lisp и Scheme. Он основан на том, что в этих языках не различаются заглавные и строчные символы, в отличие от языка C++. Поэтому при записи переменных для латинских букв из имен идентификаторов используются только заглавные латинские буквы, а для специальных символов — их названия, записыванные строчными буквами.

Таким образом, каждому идентификатору из программы соответствует корректный уникальный идентификатор языка C++. Например:

Идентификатор языка Плэнер	Переменная языка C++
`TheAtomName`	`THEATOMNAME`
`=A+B=`	`equalAplusBequal`
`*`	`star`
`A+-`	`Aplusminus`
`/Zzz`	`backslZZZ`

Для управления работой транслятора в программе на Плэнере могут быть использованы некоторые директивы. Они позволяют задавать правила перевода специальных символов в именах идентификаторов; идентификаторы, переменные для которых уже существуют в библиотеке; имена, используемые для генерируемых заголовочных и исходных файлов; имя функции инициализации модуля, которая должна будет быть вызвана из основной программы на C++; классы для встроенных функций и некоторые другие параметры работы транслятора.

Список литературы

[1] Hewitt C. *PLANNER: A Language for Proving Theorems in Robots.* IJCAI 1969.

[2] Hewitt C. *Description and theoretical analysis (using schemata) of PLANNER: a language for proving theoremes and manipulating models in a robot.* TR-258, AI Lab., MIT, 1973.

[3] E. Bolshakova and A. Stolyarov. *Building functional techniques into an object-oriented system.* In Knowledge-Based Software Engineering. Proceedings of the 4th JCKBSE, vol. 62 of Frontiers in Artificial Intelligence and Applications, pages 101-106, Brno, Czech Republic, September 2000. IOS Press, Amsterdam, 2000.

[4] И. Г. Головин, А. В. Столяров. *Объектно-ориентированный подход к мультипарадигмальному программированию.* Вестник МГУ, сер. 15 (ВМиК), N 1, 2002 г., стр. 46–50.

[5] Andrey V. Stolyarov. *A framework of heterogenous dynamic data structures for object-oriented environment: the S-expression model* In Knowledge-Based Software Engineering. Proceedings of the 6th JCKBSE, vol.108 of Frontiers in Artificial Intelligence and Applications, pages 75–82, Protvino, Russia, August 2004. IOS Press.

[6] Официальный сайт библиотеки InteLib http://www.intelib.org

[7] В.Н.Пильщиков. *Язык программирования ПЛЭНЕР-БЭСМ.* Издательство Московского университета, 1978.

[8] В.Н.Пильщиков. *Система программирования ПЛЭНЕР-БЭСМ.* Издательство Московского университета, 1983.

[9] В.Н.Пильщиков. *Язык Плэнер.* М.:Наука, 1983.

[10] J. McCarthy. *Recursive functions of symbolic expressions and their computation by machine.* Communications of the ACM, 3(4): p. 184–195, Apr 1960.