Московский государственный университет имени М.В. Ломоносова
Факультет вычислительной математики и кибернетики
Кафедра алгоритмических языков

Галкина Екатерина Владимировна

Слой метапрограммирования для объектно-ориентированной модели абстрактной Лисп-машины

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

Научный руководитель:
к.ф.-м.н., доцент
А. В. Столяров

Москва, 2018

Аннотация

Работа посвящена реализации метапрограммного слоя в рамках вычислительной модели диалектов языка Lisp, представленной в библиотеке InteLib.

В ходе работы в имеющиеся диалекты InteLib Lisp и InteLib Scheme была добавлена возможность объявления пользовательских макросов, использования синтаксических правил, реализован механизм частичной блокировки вычисления выражений.

Оглавление

1 Введение

1.1 Методы многостилевого программирования

Качество программного продукта и эффективность работы программиста во многом зависят от выбранного языка программирования, а именно от того, насколько адекватны изобразительные средства выбранного языка по отношению к конкретной решаемой задаче. При решении частных подзадач может оказаться удобным использование средств языков программирования, отличных от основного.

Мультипарадигмальное программирование позволяет использовать различные выразительные средства для решения частных подзадач в рамках одного проекта. Каждая из основных парадигм имеет свою сферу применения. Например, для символьной обработки текста наиболее удобны функциональные языки, для решения переборных задач — логические, для создания интерактивных приложений — императивные. Кроме того, во многих проектах необходимо взаимодействовать с базами данных, запросы к которым декларативны. Использование разных парадигм при решении соответствующих подзадач позволяет в целом повысить качество программы и упростить процесс её создания, но может быть невозможным в рамках базового языка или повлечь за собой слишком высокие накладные расходы. Таким образом, возникает необходимость в использовании нескольких языков программирования, которым соответствуют различные парадигмы, в рамках одного проекта.

При интеграции разнородных языков неизбежно возникают технические трудности. В статье [3] приведены следующие подходы к решению проблемы многостилевого программирования и описаны их основные преимущества и недостатки:

- применение нескольких систем программирования;
- создание нового языка;
- расширение существующего языка;

- непосредственная интеграция.

Метод непосредственной интеграции является частным случаем расширения существующего языка программирования. В его основе лежит моделирование средствами заранее выбранного базового языка синтаксиса и семантики альтернативных языков.

В статье [3] в качестве базового языка предлагается использовать объектно-ориентированный язык программирования, а также приводятся требования, которым должен удовлетворять этот язык. Во-первых, для построения модели альтернативного языка необходимо наличие в базовом языке понятия класса, поддерживающего инкапсуляцию, наследование и полиморфизм. Во-вторых, для синтаксически схожего представления конструкций альтернативного языка базовый язык должен предоставлять возможность переопределения стандартных операций. Кроме того, этот язык должен быть достаточно известным, иметь библиотеки для различных областей и реализации на нескольких платформах, что делает допустимым его использование в большинстве проектов.

В библиотеке InteLib [3, 4] реализован метод непосредственной интеграции [5]. В качестве базового языка используется объектно-ориентированный язык C++ [6], как отвечающий всем перечисленным выше требованиям. На данный момент в рамках проекта InteLib реализованы библиотеки классов, моделирующие изобразительный стиль языков Lisp [7], Scheme [8], Refal [9] и Prolog [10]. Следует уточнить, что речь идёт лишь о подмножествах данных языков, включающих в себя их основные особенности, поскольку предполагается, что они будут использоваться в качестве второстепенных языков программирования для решения определённых подзадач.

1.2 Метапрограммирование в диалектах языка Lisp

Метапрограммирование представляет собой способ создания программ, которые могут манипулировать с программным кодом как с данными, т. е. имеют возможность его читать, модифицировать или генерировать. Такие программы называются метапрограммами. К метапрограммам также можно отнести самомодифицирующийся код, который изменяет собственный текст во время выполнения.

Использование шаблонов является распространённом примером генерации кода перед этапом компиляции. Шаблоны используют для сокращения количества одинаковых участков программного кода, которые незначительно отличаются друг от друга (например, только именами типов), но синтаксис

языка не позволяет вынести повторяющиеся конструкции в отдельную процедуру или функцию. Использование шаблонов позволяет повысить читабельность кода и предотвращает ошибки, которые могут возникнуть, если похожие участки кода были скопированы и вручную изменены. Ещё одним примером метапрограммирования является создание встраиваемых предметно-ориентированных языков. Такие языки позволяют создавать относительно простые конструкции для работы со специфическими для конкретной предметной области структурами данных, в то время как использование исходного языка напрямую привело бы к написанию нетривиальных и громоздких синтаксических конструкций. Эти и другие возможности использования метапрограмм приведены в статье [11].

В качестве конкретных примеров инструментов для метапрограммирования можно привести шаблоны языка C++ [6] и препроцессор языка C [12]. Во многих функциональных языках также имеется возможность использования шаблонов для генерации кода: одним из примеров являются шаблоны в языке Haskell [13], которые чаще всего применяются для автоматической генерации объявлений экземпляров классов типов для схожих структур данных. Кроме этого, компилятор GHC [14] позволяет использовать правила переписывания кода (*rewrite rules*) в целях оптимизации [15].

В приведённых выше примерах, как и во многих других языках программирования, синтаксические конструкции, предназначенные для метапрограммирования, были созданы специально для этой цели и не используются в других случаях. С другой стороны, никакие другие конструкции языка не могут быть использованы для генерации или обработки программного кода. Таким образом, метапрограммный слой является отдельной частью языка, а генерация кода представляет собой одну из стадий компиляции (обычно самую первую) или отдельный этап предобработки программного кода, предшествующий компиляции. Вообще говоря, языки, используемые при создании метапрограмм могут отличаться: входные и выходные данные могут представлять собой тексты программ на разных языках программирования, а метапрограмма может быть написана на языке, отличающимся от предыдущих двух (примером служат компиляторы и декомпиляторы).

Отличительной особенностью языка Lisp является одинаковое представление данных и программного кода в виде S-выражений [17], вследствие чего код становится объектом первого класса. Это свойство языка программирования называется гомоиконностью и впервые было описано в работе [16].

Как и любое другое S-выражение, код на диалектах языка Lisp может обрабатываться так же, как и данные, подающиеся на вход программе. Кроме того, полученные в ходе работы какой-либо программы S-выражения могут, в свою очередь, рассматриваться как код новой программы. Таким образом, нет необходимости вводить специальные синтаксические средства для мета-

программирования, достаточно использовать примитивы языка, такие как `QUOTE` (для блокировки вычисления выражения или его части) и `EVAL` (для вычисления сформированного выражения).

На основе этих примитивов в диалекты языка Lisp добавляются более удобные в использовании средства, позволяющие определить формирование и дальнейшее вычисление произвольной формы или целой программы. Так, в диалекте Common Lisp средством метапрограммирования являются макросы [7], а в Scheme — так называемые синтаксические правила (*syntax rules*), определяющие способы преобразования входного S-выражения на основе сопоставления его с образцом [21].

Задача определения языка программирования на нём самом в общем случая является нетривиальной, однако свойство гомоиконности позволяет сравнительно легко запрограммировать интерпретатор языка Lisp на нём же. Это можно сделать, определив функцию `EVAL`, вычисляющую S-выражения, используя остальные базовые функции языка, условные операторы и определения функций. Код такого интерпретатора приведён в книге [20].

1.3 Постановка задачи

Целью работы является реализация метапрограммного слоя в рамках вычислительной модели диалектов языка Lisp, реализованной в библиотеке InteLib.

Для достижения поставленной цели необходимо решить следующие задачи:

1. реализовать в рамках диалекта InteLib Lisp средства для объявления пользовательских макросов (форма `DEFMACRO`);

2. добавить в диалекты InteLib Lisp и InteLib Scheme возможность использования частичной блокировки вычислений;

3. разработать и реализовать механизм для работы с синтаксическими правилами в диалекте InteLib Scheme;

2 Представление S-выражений в библиотеке InteLib

2.1 Структура проекта

Логически структуру библиотеки InteLib можно разделить на несколько уровней, каждый из которых является «надстройкой» над предыдущим и предоставляет больший функционал, но не влияет на работу совокупности нижних уровней. Исходных код также разделяется по директориям, что даёт возможным компилировать только ту часть библиотеки, которая действительно будет использована.

В основе этой структуры лежит реализация S-выражений в виде иерархии полиморфных классов языка C++. На их основе реализованы базовые структуры данных, такие как вектор, хеш-таблица и двунаправленный список. Они также являются разновидностью S-выражений. Кроме того, библиотека предоставляет набор вспомогательных классов и функций: обобщённое представление потоков ввода и вывода в виде S-выражений, конфигурируемый лексический и синтаксический анализатор текстового представления S-выражений, функции для работы с числовыми S-выражениями.

На следующем уровне реализована вычислительная модель диалектов языка Lisp, основанная на трёхстековой виртуальной машине, которая представляет собой континуацию, в рамках которой ведутся вычисления. На этом же уровне находится реализация базовых функций и специальных форм, которые есть в обоих диалектах (`AND`, `OR`, `COND` и другие). Поскольку диалекты InteLib Lisp и InteLib Scheme существенно различаются, вычислительная модель для каждого из них реализована отдельно, с использованием базового функционала обобщённой модели. Также отдельно реализованы функции, специфические для каждого из диалектов. Кроме того, текущая версия библиотеки включает в себя вычислительные модели языков Prolog и Refal, описание последней из которых представлено в статье [22] и выходит за рамки данной работы.

Далее следует уровень, на котором реализованы трансляторы ILL и ILS и интерактивные интерпретаторы NILL и NILS.

2.2 Представление S-выражений

Основной структурой данных в диалектах языка Lisp являются S-выражения [17] (S-expressions, symbolic expressions). S-выражение представляет собой либо атомарное выражение (константа определённого типа, символ и т. п.), либо точечную пару, составленную из S-выражений. Это рекурсивное определение можно записать в виде БНФ [18]:

```
<S-выражение> ::= атом | (<S-выражение> . <S-выражение>)
```

По историческим причинам за левым и правым подвыражением в точечной паре закрепились названия CAR и CDR соответственно: так назывались регистры, предназначенные для хранения этих значения в первой реализации языка [20]. Далее будем придерживаться этой терминологии.

Библиотека InteLib предоставляет набор классов языка C++, реализующих S-выражения как гетерогенные структуры данных [5]. Общие свойства, характерные для всех возможных S-выражений, представлены в базовом абстрактном классе `SExpression`.

Атомарные S-выражения разделяются на несколько типов. Иерархия числовых атомов состоит из целых чисел, рациональных дробей, чисел с плавающей точкой, и комплексных чисел. S-выражения этих типов представлены классами `SExpressionInt`, `SExpressionRational`, `SExpressionFloat` и `SExpressionComplex`. Поскольку рациональные и комплексные числа используются реже, при сборке библиотеки есть возможность указать флаг компиляции, отключающий это расширение. Помимо числовых констант, атомарные S-выражения разделяются на строковые константы и метки. Символьные константы представляют собой строки единичной длины. Метки однозначно идентифицируются своим адресом и используются для представления уникальных объектов, т. е. S-выражений, существующих в единственном экземпляре. Например, на основе этого типа данных реализован объект пустого списка, булевские значения в диалекте InteLib Scheme и символы. S-выражения перечисленных типов представлены классами `SExpressionString`, `SExpressionLabel` и `SExpressionChar` соответственно. Все они являются наследниками базового класса `SExpression`.

Точечные пары представлены классом `SExpressionCons`, также наследуемым от `SExpression`. В соответствии с определением S-выражения, поля класса являются ссылками на два других S-выражения: CAR и CDR исходного.

Для удобства использования S-выражений был введён класс `SReference`. Объект этого класса представляет собой «умный указатель» [23] на объект класса `SExpression`, обладающий дополнительной функциональностью (подсчёт ссылок на объект, интерфейс для работы с S-выражениями и т. п.). Классы `LReference` и `SchReference` являются его наследниками и отражают различия в представлении и вычислении S-выражений в рамках диалекта InteLib Lisp и InteLib Scheme соответственно.

Перегрузка операторов в языке C++ даёт возможность конструировать объекты S-выражений, используя представление, синтаксически схожее со списочным представлением в диалектах языка Lisp. Например, списку `(2 4 6)` соответствует конструкция `(L| 2, 4, 6)`, а точечная пара `("ab" . "cd")` представляется в виде `(L| "ab" || "cd")`.

Подробное описание работы с S-выражениями, методов класса `SReference` и дополнительных структур данных в библиотеке InteLib приведено в работе [19].

Помимо внутреннего представления S-выражений и операций над ними библиотека предоставляет набор инструментов для трансляции и интерпретации кода на диалектах InteLib Lisp и InteLib Scheme. Трансляторы ILL и ILS преобразуют код на соответствующем диалекте в представление на C++, формируя при этом отдельные модули, которые могут быть использованы в дальнейшем в проектах на языке C++. Интерпретаторы NILL и NILS позволяют вычислять S-выражения в интерактивном режиме.

2.3 Вычислительная модель языка Lisp и его диалектов

Реализация вычислений S-выражений основана на работе с континуацией. Континуация представляет собой последовательность действий, которые необходимо выполнить до завершения вычислений. Каждое элементарное вычисление рассматривается как переход от одной континуации к другой. Рассмотрим вычисление простого арифметического выражения `(+ 2 (- ( * 5 6) 3))`, и пусть в текущий момент времени производится вызов функции умножения, но результат ещё не известен. Тем не менее, ясно, какие действия будут совершены с результатом вычисления выражения `(* 5 6)`, до того, как исходное выражение будет вычислено окончательно. Можно представить эти действия в виде анонимной функции (λ-выражения):

```
(LAMBDA (X) (+ 2 (- X 3)))
```

Таким образом, континуация представляет собой некоторую функциональную абстракцию, что позволяет повысить выразительную мощность языка, в котором есть возможность работать с континуацией напрямую. Например, в языке Scheme континуации являются полноправными (*first-class*) объектами, т. е. могут быть переданы как аргументы, возвращены из функции или сохранены в какой-либо структуре данных. Это свойство может использоваться в том числе при реализации сопрограмм, поиска с возвратом и механизма исключений. Примеры такой работы с континуацией в языке Scheme приведены в статье [24].

В библиотеке InteLib континуацию можно рассматривать как виртуальную вычислительную машину, а её реализация вынесена в отдельный класс `IntelibContinuation`, методами которого реализуется основной механизм вычислений S-выражений. Различия между диалектами InteLib Lisp и InteLib Scheme (см. [5], стр. 128–129), очевидно, влекут за собой и различия в реализации вычислительных моделей языков. Эти особенности учитываются в классах `LispContinuation` и `SchemeContinuation`, наследуемых от `IntelibContinuation`. Например, в этих классах добавлены разные команды вычислителя, которые отсутствуют в родительском классе, а также методы, вызываемые в ходе специфических для конкретного диалекта вычислений.

Основой реализации класса `IntelibContinuation` являются стек действий, стек результатов и стек фреймов. В стеке результатов хранятся S-выражения, полученные в ходе вычислений, в том числе промежуточных. Стек действий заполняется командами из определённого множества, которые управляют поведением вычислителя. Вычисление S-выражения считается завершённым, когда в стеке действий не остаётся команд для выполнения. Стек фреймов используется для оптимизации передачи фактических параметров при вызове функций и автоматической очистки после вычисления результата. Более подробно работа со стеками описана в статьях [5] и [1].

Рассмотрим подробнее процесс вычисления S-выражений. В классах `LReference` и `SchReference`, реализован метод `Evaluate`, в котором и создаётся континуация, соответствующая используемому диалекту. Перед началом вычислений все стеки в континуации пусты. Затем в стек действий помещается команда `just_evaluate`, а параметром является подлежащее вычислению S-выражение.

Все дальнейшие вычисления происходят пошагово. Поместив команду для вычисления выражения в стек действий, метод `Evaluate` вызывает метод континуации `Step`, производящий один шаг вычислений. На каждом таком шаге из стека действий извлекается очередная команда, а дальнейшая работа вычислительной модели зависит от её интерпретации. Например, в стек действий могут быть добавлены новые инструкции, в стек результатов — новые значения. Если очередной командой оказалась команда вызова функции от

заданного числа аргументов (далее будем обозначать её `CALL/n`, где `n` — количество аргументов), то из стека результатов будут извлечены сначала `n` значений параметров функции, а затем сама функция. Из полученных объектов формируется вызов функции, а дальнейшие действия зависят от её семантики.

Таким образом, каждый шаг вычислений в общем случае приводит к изменению стеков, т. е. происходит переход к другой континуации. После окончания вычислений на вершине стека результатов находится полученное S-выражение.

2.4 Механизм лексического связывания

При вызове функции или спецформы и применении λ-выражения вычисление происходит в два этапа. На первом из них происходит связывание формальных параметров со значениями фактических параметров. Они предварительно вычисляются в случае использования обычной функции или λ-выражения, но не при вызове спецформы. Вторым этапом является вычисление тела выражения с учётом установленных связей. Формальные параметры в InteLib Lisp по умолчанию являются лексически связанными переменными, их связи действительны только в теле той формы, в которой они определены. После выхода из этой формы значение переменной будет восстановлено, если в более внешних вызовах оно было связано с другим значением. Кроме того, изменение значений лексических переменных не влияет на значения одноимённых переменных в вызовах верхнего уровня.

Механизм лексического связывания в текущей версии библиотеки InteLib основан на использовании так называемых позиционных параметров — специальных символов, позволяющих получить прямой доступ к нужному значению параметра в стеке. Позиционный параметр является разновидностью специального атома, который имеет своё правило вычисления (все остальные атомарные S-выражения вычисляются сами в себя). Результатом вычисления позиционного параметра является значение, содержащееся в соответствующем стековом фрейме.

Замена символов на позиционные параметры происходит при создании лексического замыкания, а именно при определении функций, использовании λ-выражений и LET-форм. LET-формы предназначены для создания локальных связей переменных внутри других форм и на самом деле представляют собой синтаксически видоизменённые λ-выражения [20]. На позиционные параметры заменяются символы, находящиеся в списке формальных параметров функции или в списке инициализации LET-формы.

Для реализации указанной замены символов используется специально разработанный класс `IntelibTokenReplace`, инкапсулирующий контекст замены и предоставляющий методы для обработки различных видов спецформ, списочных структур и отдельных символов. Контекст, в котором происходит замена, реализован в виде векторов соответствий между символом и его номером в стековом фрейме. Векторы разбиты на уровни, которые учитываются при обработке вложенных форм. Эти уровни определяют, в каком именно стековом фрейме будет находиться значение фактического параметра.

Основным методом класса является метод, обрабатывающий произвольное S-выражение. Атомарный объект обрабатывается по-разному в зависимости от его типа. Позиционные параметры заменяются на объекты класса, сопоставляющие символ его значению. Символы, содержащиеся в векторе соответствий на каком-либо уровне, заменяются на позиционные параметры, остальные не изменяются. Также замена не происходит для всех динамически связываемых символов и символов, которым уже было сопоставлено некоторое вычисленное значение.

Если выражение представляет собой вызов некоторой спецформы, то для дальнейшей обработки вызывается метод, отдельно определённый для каждой из них и зависящий от семантики формы.

Лексическое замыкание представляется классом `SExpressionCompiledLambda`, содержащим информацию о количестве параметров функции и обработанное тело λ-функции с позиционными параметрами.

Чтобы проиллюстрировать использование позиционных параметров, рассмотрим пример определения и вызова пользовательской функции в диалекте InteLib Lisp (в диалекте InteLib Scheme для этого используется форма `DEFINE`, которая обрабатывается аналогичным образом). Функция `(TAKE N L)` возвращает первые `N` элементов списка `L`:

```
(DEFUN TAKE (N L)
    (COND
        ((NULL L) NIL)
        ((<= N 0) NIL)
        (T (CONS (CAR L) (TAKE (- N 1) (CDR L))))
    )
)
```

Функция принимает ровно два аргумента, в определении они обозначены символами `N` и `L`, поэтому в теле функции эти символы должны быть заменены на позиционные параметры с номерами 1 и 2 соответственно. После обработки тела получим выражение следующего вида (позиционные параметры обозначены `#1` и `#2`):

```
(COND
    ((NULL #2) NIL)
    ((<= #1 0) NIL)
    (T (CONS (CAR #2) (TAKE (- #1 1) (CDR #2))))
)
```

На основе этого выражения будет создан объект класса `SExpressionCompiledLambda`, и он будет установлен как функциональное значение символа `TAKE`.

Предположим, что на верхнем уровне функция была вызвана следующим образом: `(TAKE 3 (LIST 1 2 3 4))`. Сначала её параметры будут вычислены, и в стеке результатов будут находиться значения `3` и `(1 2 3 4)`, из которых будет сформирован стековый фрейм перед вызовом функции, который будет помещён в стек фреймов и станет его верхним элементом. В этом фрейме значения будут находиться на позициях с номерами 1 и 2 соответственно. При вычислении в теле функции позиционных параметров `#1` и `#2` произойдёт обращение к позициям фрейма с соответствующим номером, т. е. их значениями как раз будут переданные при текущем вызове аргументы.

При рекурсивном вызове функции процесс повторится: будут вычислены новые значения аргументов (`(- #1 1)` вычисляется в `2`, `(CDR #2)` — в `(2 3 4)`), из которых формируется новый стековый фрейм, который будет помещён в стек поверх предыдущего. Заметим, что предыдущий фрейм ещё не может быть удалён, поскольку значения из него используются после выхода из рекурсивного вызова. При хвостовой рекурсии стек фреймов автоматически очищается от значений, которые не будут использоваться в дальнейших вычислениях.

LET-формы предназначены для создания локальных связей переменных внутри других форм и на самом деле представляют собой синтаксически видоизменённые λ-выражения [20].

Разницу в вычислении форм `LET` и `LET*` покажем на примере:

```
>>> (LET ((X 1) (Y (* X 2))) (LIST X Y))
Error: symbol has no value: X
>>> (LET* ((X 1) (Y (* X 2))) (LIST X Y))
(1 2)
```

Как и в случае лексического замыкания, LET-формы обрабатываются одинаково в обоих диалектах. Обработанная LET-форма представляет собой объект класса `SExpressionCompiledLet`, который затем размещается в стеке для вычисления.

Создание этого объекта начинается с добавления нового уровня для вектора соответствий и прохода по списку инициализации. Отличие формы `LET`

от LET* состоит в том, что при обработке выражения из списка инициализации предыдущие символы из этого списка не учитываются. Для этого была добавлена возможность создавать отдельный «скрытый» уровень.

Поскольку в качестве нового значения символа может использоваться произвольное S-выражение, сперва обрабатывается оно (в нём некоторые символы могут быть заменены на позиционные параметры в случае вложенного вызова или использования формы LET*). Затем связываемый символ добавляется в вектор соответствий, и ему приписывается очередная позиция. После того, как все пары в списке инициализации были обработаны, т. е. вектор соответствий был целиком сформирован, в теле формы происходит замена символов на позиционные параметры.

В результате этого процесса получается набор объектов, в которых присутствуют позиционные параметры: формы инициализации и тело LET-формы. Из них создаётся новый объект, который является разновидностью специального атома. Общая схема вычисления этого атомарного объекта состоит в следующем: создаётся новый стековый фрейм, в стек действий помещается команда для вычисления тела формы, а затем — команды для вычисления выражений из списка инициализации.

В языке Lisp переменные по умолчанию считаются статическими, однако есть возможность использования динамических переменных. Эти виды связывания отличаются способом определения значения переменной и влияют на область её видимости. Статическая переменная видна только в теле той формы, в которой была объявлена (если она не является глобальной), а её значение зависит от контекста места её определения. Значение динамически связанной переменной определяется во время вычисления. Для определения динамических переменных предназначена форма DEFVAR [7]. Такая переменная, даже связанная локально (например, в LET-форме), видна во всех вложенных вызовах функций. После выхода из формы, в которой локально устанавливается новое значение динамической переменной, восстанавливается предыдущее связывание, если оно было. Таким образом, вычисления будут происходить по-разному для статических и динамических переменных, в случае, если переменная свободна в вычисляемом выражении, но при этом является формальным параметром в более внешней форме [20].

Для корректной работы в рамках имеющегося механизма связывания динамические переменные обрабатываться особым образом. Во-первых, при создании замыкания и при обработке форм LET и LET* динамически связанные символы, встречающиеся в списке формальных параметров или списке инициализации, не должны заменяться на позиционные параметры. Во-вторых, при связывании динамического символа с новым значением необходимо запланировать восстановление старого значения.

Подробное описание реализации описанного механизма лексического связывания и обработки динамических переменных в диалекте InteLib Lisp представлено в статье [1].

3 Макросы в языке Lisp

3.1 Средства макроопределения и макровызова

Использование макросов в языке Lisp основано на создании специальных форм, которые в ходе своей работы сначала формируют программный код, а затем вычисляют полученное выражение. Свойство гомоиконности делает этот процесс естественным для языка, поэтому для работы с макросами не требуется использовать специфические синтаксические средства. Для создания кода, который будет вычислен, достаточно использовать встроенные функции языка для построения и декомпозиции S-выражений (`CONS`, `LIST`, `CAR`, `CDR` и т. д.).

Например, выражение

```
(LIST 'APPEND '(LIST 1 2 3) '(LIST "foo" "bar"))
```

вычисляется в

```
(APPEND (LIST 1 2 3) (LIST "foo" "bar"))
```

Это выражение представляет собой корректный код на языке Lisp, который может быть вычислен с помощью формы `EVAL`, в результате чего будет получен ожидаемый результат: список `(1 2 3 "foo" "bar")`. Таким образом, форму

```
(EVAL (LIST 'APPEND '(LIST 1 2 3) '(LIST "foo" "bar")))
```

уже можно назвать метапрограммой на языке Lisp, поскольку в ходе вычисления этого выражения происходит формирование программного кода и дальнейшее его вычисление.

Использование метапрограммных средств подразумевает создание макросов, позволяющих генерировать похожие фрагменты кода для разных входных параметров. Составление S-выражений и дальнейшее их вычисление

вручную, как в приведённом выше примере, не предоставляет такой возможности: сгенерированный таким образом код не может быть переиспользован или частично изменён.

Для создания макросов, зависящих от параметров, в языке Lisp предназначена специальная форма `DEFMACRO`. Синтаксис объявления макроса совпадает с синтаксисом формы `DEFUN`, используемой при описании пользовательских функций:

```
(DEFMACRO NAME (<args>) <body>),
```

где `<args>` — одноуровневый список формальных параметров макроса (символы), `<body>` — тело макроса, последовательность произвольных S-выражений.

В результате работы формы символ `NAME`, обозначающий имя макроса, связывается с функциональным объектом с особым способом вычисления, отличным от вызова обычной функции. Процесс вычисления макровызова будет рассмотрен ниже.

Определение макроса, как и функции, может быть рекурсивным или содержать вложенные вызовы других макросов. Макрос может быть вариадическим, т. е. иметь несколько обязательных параметров, и произвольное число опциональных, которые передаются в виде списка (возможно, пустого). Синтаксис объявления вариадических макросов и функций может отличаться в различных диалектах языка. Например, в диалекте Common Lisp для этого в списке формальных параметров используется ключевое слово `&rest`, которое означает, что следующий за ним аргумент при вызове должен быть связан со списком опциональных параметров [7].

В диалекте InteLib Lisp ключевые слова не используются при описании функции, вместо этого формальные параметры могут быть представлены в виде точечного списка (список вида `(arg1 arg2 . rest)`), последний символ которого при вызове функции связывается со списком дополнительных параметров. В последующих примерах будем придерживаться этого синтаксиса.

Вызов макроса в диалектах языка Lisp совпадает с вызовом функции и представляется в виде списка, первый элемент которого является именем функции или макроса, а все остальные — аргументами. Несмотря на то, что синтаксически определение и вызов функции и макроса выглядят одинаково, способы их вычисления значительно отличаются.

Основное отличие функции от макроса состоит в том, что функция преобразует S-выражения, рассматривая их как данные, в то время как макросы позволяют создавать новые конструкции для преобразования синтаксиса языка, т.е. работают с S-выражениями как с представлением программного кода. Во многом именно этим объясняются особенности вычисления макровызова.

Во-первых, при вызове макроса не происходит предварительного вычисления аргументов. Они подставляются в тело макроса в том виде, в котором были переданы при вызове. Можно сказать, что аргументы макроса рассматриваются как часть синтаксической структуры, которая будет получена при его раскрытии. При этом в теле макроса аргументы могут быть как угодно преобразованы, в том числе вычислены с помощью формы `EVAL`. Напомним, что при вызове функции сначала происходит вычисление аргументов, а формальные параметры функции связываются с полученными в ходе вычисления выражениями.

В диалектах языка Lisp по умолчанию принято использовать энергичную стратегию вычислений, «слева направо, изнутри наружу» (*leftmost innermost*) [20, 25]. В некоторых диалектах, например Scheme, есть возможность использовать отложенные вычисления, но для этого необходимы специально предназначенные функции (см. § 4.2.5 и § 6.4 в [21]).

Свойство макроса не вычислять свои аргументы во время вызова может быть использовано для имитации ленивого порядка вычислений, при котором аргументы и выражения внутри формы, которые не вызываются, не будут вычислены. В качестве примера можно привести условные конструкции, такие как `IF`, `AND`, `OR`, в которых вычисления могут пойти только по одной из ветвей, в зависимости от условия.

Вторым отличием является способ вычисления тела функции и макроса. Тело функции вычисляется как обычное выражение, в котором символы, обозначающие фактические параметры, связаны с вычисленными значениями формальных параметров. Значение этого выражения и является результатом применения функции. Вычисление тела макроса происходит в несколько этапов и представляет собой более сложный процесс, хоть и схожий с вызовом функции.

На первом этапе устанавливается связь формальных параметров с фактическими аргументами в том виде, в котором они были переданы. Следующий этап называется расширением или раскрытием макроса (*macro expansion*). На этом этапе осуществляется вычисление тела макроса, подобно вычислению тела функции. Этот процесс можно рассматривать как преобразование программного кода по правилам, заданным в определении макроса. Полученная форма обычно имеет более сложную структуру, чем исходная форма вызова. В отличие от вызова функции, вычисление макровызова на этом не останавливается, поскольку на текущей стадии было произведено только формирование кода. Поэтому на следующем этапе производится вычисление полученной при расширении формы, т. е. исполнение сгенерированного кода. Её значение и возвращается в качестве результата всего макровызова.

Проиллюстрируем описанный выше механизм вычисления макровызова на примере. Реализуем вариадическую функцию `AND-FUNC`, осуществляющую

конъюнкцию своих аргументов (должен присутствовать хотя бы один аргумент). Функция по очереди проверяет значения своих аргументов. Если хотя бы один из аргументов функции вычисляется в NIL, это значение является результатом вызова, в противном случае результатом считается значение последнего аргумента:

```
(DEFUN AND-FUNC (X . XS)
    (COND
        ((NULL XS) X)
        (X (APPLY #'AND-FUNC XS))
        (T NIL)
    )
)
```

Несмотря на то, что функция может проверить значения не всех своих аргументов, все они будут вычислены во время вызова. Это легко проверить, передав в качестве аргумента выражение, содержащее побочный эффект (например, печать в стандартный поток вывода) или приводящее к аварийной остановке программы:

```
>>> (AND-FUNC (PRINT "side-effect") NIL (/ 1 0))
Error: /: division by zero
```

Значение NIL встречается в списке аргументов раньше, чем выражение, содержащее деление на ноль, и результатом вычисления согласно семантике должно быть также NIL. Однако вызов функции аварийно завершится ещё при обработке аргументов, до вычисления тела функции.

Реализация конъюнкции в виде вариадического макроса выглядит следующим образом:

```
(DEFMACRO AND-MACRO (X . XS)
    (IF (NULL XS)
        X
        (LIST 'IF X
              (CONS 'AND-MACRO XS)
              NIL
        )
    )
)
```

Заметим, что встроенная условная конструкция IF не вычисляет свои аргументы перед вызовом и может быть реализована как «синтаксический сахар» для вызова макроса COND [7]. В любом случае, из аргументов этой формы обязательно вычисляется только условие и ровно одна из ветвей (в зависимости от значения этого условия). Вторая ветвь может отсутствовать, в таком случае результат вычисления не определён и зависит от реализации.

Если в вызове макроса один аргумент, он будет получен в результате макрорасширения и в дальнейшем вычислен. В противном случае будет сгенерирован код, проверяющий первый аргумент. В зависимости от его значения либо вызов макроса рекурсивно продолжится, либо будет возвращено значение `NIL`. Каждый аргумент макроса вычисляется только в том случае, если все предыдущие аргументы вычислялись в значение, отличное от `NIL`. Применим макрос к тем же аргументам:

```
>>> (AND-MACRO (PRINT "side-effect") NIL (/ 1 0))
"side-effect"
NIL
```

В данном случае результатом макрорасширения является выражение

```
(IF (PRINT "side-effect")
    (AND-MACRO NIL (/ 1 0))
    NIL
)
```

При его вычислении будет произведена печать строки `"side-effect"` в стандартный поток вывода, а затем осуществлён ещё один вызов макроса, генерирующий следующий код:

```
(IF NIL
    (AND-MACRO (/ 1 0))
    NIL
)
```

Здесь вычисления пойдут по второй ветви `IF`, и макрос не будет вызываться повторно. В итоге выражение, содержащее побочный эффект было вычислено, но до деления на ноль вычисления так и не дошли из-за значения второго аргумента, несмотря на то, что выражение `(/ 1 0)` было передано в рекурсивный вызов макроса.

Третьим существенным отличием является контекст вычисления макроса и функции. Во время определения функции или при использовании безымянных λ-выражений и функциональной блокировки (форма `FUNCTION` или символ `#'`) происходит создание лексического замыкания, при котором фиксируются связи свободных переменных в теле функции. То же самое происходит и при определении макроса, поэтому при расширении макроса действует контекст его определения, как и при вызове функции.

Как было сказано ранее, расширение макроса представляет собой преобразование программного кода. В результате может быть получено выражение, содержащее переменные, которые не были бы связаны со значениями в

контексте определения макроса. Сам вызов макроса можно рассматривать как синтаксическую замену этого вызова на некоторый фрагмент кода. Вычисление этого фрагмента будет происходить в контексте выражения, содержащего макровызов, а связи из контекста определения макроса уже не будут действовать. Таким образом, макрорасширение и дальнейшее вычисление полученного выражения происходят в разных контекстах.

Проиллюстрируем эту особенность на примере обмена значений двух символов. Реализуем сначала функцию SWAP-FUNC, которая принимает на вход два аргумента (символы) и пытается поменять местами их значения, используя вспомогательный символ, определяемый с помощью формы LET:

```
(DEFUN SWAP-FUNC (X Y)
    (LET ((TMP X))
        (SETQ X Y)
        (SETQ Y TMP)
    )
)
```

Пусть изначально значения символов A и B были равны 0 и 1 соответственно. Попробуем поменять их значения с помощью функции:

```
>>> (SETQ A 0)
0
>>> (SETQ B 1)
1
>>> (SWAP-FUNC A B)
0
>>> A
0
>>> B
1
```

Значения исходных переменных не изменились, поскольку вычисление происходило в контексте определения функции, откуда нет доступа к самим символам A и B. Кроме того, аргументы функции были сначала вычислены, поэтому в качестве фактических параметров вызова выступали значения 0 и 1.

Для решения этих проблем реализуем обмен значений в виде макроса. Определим макрос SWAP-MACRO, который строит синтаксическую конструкцию, аналогичную телу функции SWAP-FUNC:

```
(DEFMACRO SWAP-MACRO (X Y)
    (LIST 'LET (LIST (LIST 'TMP X))
          (LIST 'SETQ X Y)
          (LIST 'SETQ Y 'TMP)
    )
)
```

В результате расширения макровызова `(SWAP-MACRO A B)` будет получено следующее выражение:

```
(LET ((TMP A))
    (SETQ A B)
    (SETQ B TMP)
)
```

Здесь вместо формальных параметров макроса были подставлены сами символы `A` и `B`, а не их значения, поскольку при вызове макроса аргументы не вычисляются.

Заметим, что форма `LET` в некоторых диалектах языка Lisp также может быть макросом, и в таком случае в результирующем выражении произойдёт дальнейшее преобразование синтаксиса. Например, `LET` как макрос может раскрываться в последовательность применений λ-функций.

Полученное выражение будет вычислено в контексте макровызова, где определены символы `A` и `B`, в результате чего их значения действительно поменяются местами:

```
>>> (SETQ A 0)
0
>>> (SETQ B 1)
1
>>> (SWAP-MACRO A B)
0
>>> A
1
>>> B
0
```

Таким образом, были обозначены следующие особенности вычисления вызова макроса в диалектах языка Lisp:

- аргументы макроса не вычисляются, а подставляются в тело макроса в исходном виде;
- тело макроса вычисляется в несколько этапов: связывание формальных параметров, макрорасширение и вычисление результирующей формы;
- при макрорасширении действует контекст его определения, а вычисление полученной формы происходит в контексте макровызова.

3.2 Внутреннее представление макроса

Для реализации объявления и вычисления макросов в имеющейся вычислительной модели необходимо, во-первых, обеспечить внутреннее представление макроса в виде объекта класса языка C++, вычисление которого эквивалентно вызову макроса с некоторыми аргументами; во-вторых, предоставить возможность пользователям объявлять собственные макросы, т. е. добавить в диалект InteLib Lisp спецформу `DEFMACRO`. Предложенная реализация была описана в работе [2].

Рассмотрим вычисление произвольной неатомарной формы в имеющейся вычислительной модели. Если выражение является списком (или точечной парой, в более общем случае), то, согласно синтаксису языка, оно представляет собой вызов функции или формы, возможно, с аргументами. При этом первый элемент списка является символом, связанным с каким-то применимым (*applicable*) объектом.

Применимые объекты представлены в библиотеке InteLib в виде двух основных абстрактных классов: `SExpressionForm` и `SExpressionFunction`, реализующих вызов специальной формы и обычной функции соответственно. Встроенные функции и формы представляются в виде наследников этих классов. Рассмотрим их отличия друг от друга более подробно.

Класс `SExpressionFunction` наследуется от класса `SExpression` и представляет S-выражение, которое можно применить к определённому числу предварительно вычисленных аргументов. Класс инкапсулирует минимальное и максимальное количество аргументов, к которому может быть применена функция. Для представления бесконечного числа аргументов используется значение -1.

Вызов функции реализован в двух методах. Первый из них, `Apply`, используется, если на текущем шаге вычислений происходит обработка команды `CALL/n`. В методе происходит проверка числа аргументов, и в случае положительного результата из стекового фрейма формируется вектор фактических параметров функции. После этого вызывается виртуальный метод, который имеет собственную реализацию в зависимости от семантики функции (все встроенные функции определяются как классы-наследники `SExpressionFunction`).

В частности, класс `SExpressionCompiledLambda` также наследуется от `SExpressionFunction`. Его применение состоит в вычислении тела анонимной функции с учётом контекста, сформированного в стековом фрейме, с помощью позиционных параметров (см. § 2.4).

Класс `SExpressionForm` представляет так называемые специальные формы языка — синтаксические конструкции с особым способом вычисления. Все спецформы наследуются от этого абстрактного класса и предоставляют ре-

ализации двух виртуальных методов. Первый из них, метод `Call`, реализует вычисление каждой отдельной формы. В качестве аргументов ему передаются параметры вызова формы в исходном виде. Второй метод, `ReplaceTokens`, предназначен для обработки тела спецформы при создании лексического замыкания (замены символов на позиционные параметры). Для некоторых спецформ, таких как `LET` или `DEFUN`, этот процесс может быть нетривиален.

При вычислении произвольной списочной структуры сначала происходит проверка первого элемента: с ним должна быть связана функция или спецформа. В случае вычисления вызова обычной функции, в стек результатов помещается функциональный объект, в стек действий — команда для вызова функции (`CALL/n`) и команды для вычисления аргументов. Во время исполнения команды `CALL/n` формируется стековый фрейм, после чего вызывается метод `Apply`.

Вызов спецформы не требует предварительных действий, поэтому происходит вызов метода `Call`, которому передаётся остаток исходной формы (хвост списка). Дальнейшее поведение вычислителя зависит от семантики конкретной формы.

Очевидно, что макрос также является применимым объектом, поэтому его внутренним представлением должен быть класс, наследуемый от одного из описанных выше абстрактных классов.

В библиотеку был добавлен класс `LExpressionCompiledMacro`, являющийся наследником класса `SExpressionForm`, поскольку при вызове макроса не вычисляются аргументы. С другой стороны, макрорасширение можно рассматривать как применение λ-выражения к аргументам в их исходном виде. Поэтому класс `LExpressionCompiledMacro` инкапсулирует объект класса `SExpressionCompiledLambda`, который представляет собой тело макроса в виде анонимной функции.

3.3 Создание и вычисление внутреннего представления

Создание объекта класса `LExpressionCompiledMacro` происходит при вызове спецформы `DEFMACRO`, которая была добавлена в диалект InteLib Lisp в ходе данной работы. Аргументами формы являются символ, обозначающий имя объявляемого макроса, список формальных параметров и тело макроса. На основании этих значений необходимо создать объект класса `LExpressionCompiledMacro`.

Обработка аргументов начинается с нумерования параметров макроса и определения их количества. Макрос, как и функция, может быть вариадическим, т. е. принимать произвольное число параметров. В этом случае спи-

сок формальных параметров является точечным. Пример определения такого макроса приведён в § 3.1. Эта ситуация также учитывается при обработке параметров.

После этого в теле макроса происходит замена символов из списка формальных параметров на позиционные параметры с соответствующим номером.

С использованием полученных выражений создаётся новое лексическое замыкание, т. е. объект класса `SExpressionCompiledLambda`, который в свою очередь служит основой для создания нового объекта класса `LExpressionCompiledMacro`. Заметим, что создание лексического замыкания происходит в контексте макроопределения.

Полученный объект, определяющий пользовательский макрос, устанавливается как функциональное значения символа, обозначающего имя нового макроса. В диалекте InteLib Lisp каждый символ может иметь два значения. Одно из них, функциональное, используется, если символ стоит на первом месте в вычисляемой списочной структуре (так происходит как раз в случае макровызова). Функциональное значение символа в диалекте InteLib Lisp также можно получить, применив к нему функцию `FUNCTION` (или её сокращённый вариант `#'`). Второе значение символа используется во всех остальных случаях.

Рассмотрим реализацию макровызова в рамках имеющегося вычислителя. Пусть макрос с именем `M` применяется к аргументам $arg_1 ... arg_n$. Будем обозначать тело макроса (λ-выражения, инкапсулированного в объекте класса `LExpressionCompiledMacro`) как $<body>$.

Для реализации расширения макроса с данными аргументами необходимо произвести вызов λ-выражения с этими аргументами, не вычисляя их. Это выполняется с помощью команд вычислителя следующим образом. Предварительно сохраняется так называемая метка стека действий, указывающая текущую позицию в стеке. Затем в стек помещается команда для вызова функции `CALL/n` (количество аргументов вызова известно), а также команды для квотирования λ-функции и аргументов в обратном порядке. Таким образом, стек действий будет иметь вид, показанный на рис. 3.1.

Затем выполняется несколько шагов вычислителя, до тех пор, пока стек действий не окажется в исходном состоянии (для того, чтобы это определить, используется заранее сохранённая метка). Инициировать очередной шаг вычисления возможно с помощью метода `Step` класса `IntelibContinuation`.

В момент выполнения команды `CALL/n` стек результатов будет подготовлен для вызова функции, т. е. для подстановки невычисленных аргументов макроса в тело и вычисления полученной формы. Конфигурация стеков в этот момент вычислений приведена на рис. 3.2. После выполнения этой ко-

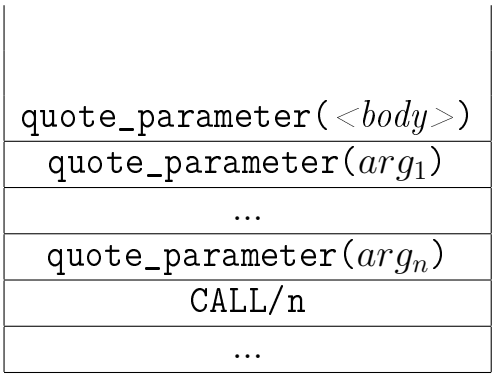

Рис. 3.1: Содержимое стека действий перед макрорасширением.

манды на вершине стека результатов окажется выражение, полученное при макрорасширении.

Рис. 3.2: Содержимое стеков во время макрорасширения.

Вызов макроса может происходить внутри выражения, для которого создаётся лексическое замыкание. Поскольку расширение тела макроса должно происходить в контексте его определения, описанный процесс макрорасширения реализован в методе `ReplaceTokens` класса `LExpressionCompiledMacro`. В противном случае, если перенести реализацию в метод `Call`, контекст определения будет утерян, поэтому как расширение макроса, так и дальнейшее вычисление полученной формы, будут происходить в контексте вызова, что в общем случае приведёт к ошибочному результату вычисления всего макровызова.

Ещё один нюанс, связанный со сменой контекста при вызове макроса во время создания лексического замыкания, заключается в том, что полученная при расширении макроса форма не связана с контекстом вычисления. Это выражение в данный момент вычислений представляет собой простую синтаксическую замену, и её дальнейшее использование может привести к неверным вычислениям. Например, если вызов макроса происходит при определении функции (внутри формы `DEFUN`), и эта форма содержит формальные параметры определяемой функции, они не будут заменены на позиционные

параметры и в результате не будут связаны с фактическими параметрами функции при её вызове. В общем случае результат макрорасширения может содержать вложенные вызовы других спецформ, в том числе рекурсивные вызовы того же макроса. Поэтому это выражение должно быть подвергнуто дальнейшей обработке.

Для решения этой проблемы из стека результатов извлекается выражение, полученное при расширении макроса, и к нему применяется метод `ReplaceTokens`. При этом устанавливается контекст вызова макроса и происходит обработка спецформ. В самом простом случае выражение не будет изменено. Таким образом, первый этап вычисления макроса будет завершён.

Для реализации второго этапа вычисления макровызова используется команда `just_evaluate`, которая вычисляет произвольное S-выражение. Она помещается в стек действий с результатом макрорасширения в качестве аргумента. Дальнейшие действия вычислителя зависят от самой формы.

Таким образом, в ходе работы была предложена реализация формы объявления макросов и вычисления макровызовов в рамках диалекта InteLib Lisp с использованием имеющегося механизма лексического связывания. Это даёт возможность пользователю объявлять собственные спецформы для нестандартного преобразования синтаксиса.

3.4 Примеры использования макросов

Стоит заметить, что в большинстве диалектах языка Lisp встроенные спецформы, такие как `AND`, `OR` и т.д., реализованы в виде макросов. Отличие диалекта InteLib Lisp состоит в том, что встроенные специальные формы также представлены в виде особых S-выражений, потомков класса `SExpressionForm`. Способ вычисления каждой из встроенных форм задан в явном виде с использованием команд вычислителя. Такая реализация обусловлена более высокой скоростью работы по сравнению с вычислением вызова пользовательского макроса.

Тем не менее, в качестве демонстрации новых возможностей диалекта, приведём реализацию некоторых встроенных спецформ диалекта InteLib Lisp в виде макросов:

```
(DEFMACRO AND (X . XS)
    (IF (NULL XS)
        X
        (LIST 'IF X
              (CONS 'AND XS)
              NIL
        )
    )
```

```
)

(DEFMACRO OR (X . XS)
    (IF (NULL XS)
        X
        (LIST 'LET
            (LIST (LIST 'TMP X))
            (LIST 'IF 'TMP
                'TMP
                (CONS 'OR XS)
            )
        )
    )
)

(DEFMACRO SWAP (X Y)
    (LIST 'LET (LIST (LIST 'TMP X))
          (LIST 'SETQ X Y)
          (LIST 'SETQ Y 'TMP)
    )
)
```

Здесь макросы `AND` и `OR` имеют стандартную семантику. В ходе тестирования эти макросы были использованы вместо имеющихся спецформ, встроенных в диалект. Макрос `SWAP` меняет местами значения символов. Его реализация и процесс вычисления уже были подробно рассмотрены выше.

4 Метапрограммирование с использованием шаблонов кода

4.1 Механизм частичной блокировки

В главе 3 приведены несколько примеров объявления макросов, которые генерируют довольно простые по своей структуре S-выражения. Тем не менее, код самого тела макроса выглядит громоздким и не позволяет программисту быстро понять, что будет сгенерировано в результате. Кроме того, при построении нового выражения почти всегда используются символы, связанные со встроенными и пользовательскими функциями и формами языка. Эти символы должны быть заквотированы, чтобы избежать их повторного вычисления в результирующем выражении.

Сравним, например, тело макроса `SWAP-MACRO` и результат макрорасширения при его применении к символам `A` и `B`:

```
(LIST 'LET (LIST (LIST 'TMP X))
      (LIST 'SETQ X Y)
      (LIST 'SETQ Y 'TMP)
)

(LET ((TMP A))
    (SETQ A B)
    (SETQ B TMP)
)
```

При объявлении макроса структура выражения, которое должно быть получено при макрорасширении, уже известна, и объявление макроса выглядело бы гораздо проще и понятнее, если бы эту структуру можно было задавать в явном виде, не вычисляя само выражение. Для блокировки вычислений в диалектах языка Lisp используется форма `QUOTE`, но в случае объявления макроса полная блокировка вычислений приведёт к тому, что фактические параметры не будут связаны с формальными, и будут присутствовать в расширенном выражении в качестве тех же символов. При вычислении этого

выражения, которое происходит в контексте вызова макроса, эти символы могут быть связаны с другими значениями или вовсе не иметь связей. Это приведёт к аварийной остановке программы или неверному поведению, источник которого найти нетривиально.

Например, если символы X и Y не связаны ни с каким значением, то возможен следующий ход вычислений при использовании полной блокировки вычислений для построения тела макроса:

```
>>> (DEFMACRO SWAP-MACRO (X Y)
  >     '(LET ((TMP X))
  >         (SETQ X Y)
  >         (SETQ Y TMP)
  >     )
  > )
SWAP-MACRO
>>> (SETQ A 0)
0
>>> (SETQ B 1)
1
>>> (SWAP-MACRO A B)
Error: symbol has no value: X
```

Здесь для корректной работы макроса было бы достаточно вычислить только значения символов, обозначающих аргументы функции, не вычисляя остальные части выражения и само выражение целиком.

Именно с этой целью в диалекты языка Lisp был добавлен механизм частичной блокировки вычислений, который позволяет значительно упростить объявление пользовательских макросов, хотя может использоваться и независимо от них.

Для частичной блокировки вычисления выражения перед ним ставится символ обратной кавычки ‘ (*backquote*), поэтому данную операцию часто называют обратной блокировкой вычислений. Перед каждым выражением, которое должно быть вычислено, ставится запятая, причём этот символ не может встречаться вне выражений, вычисление которых частично заблокировано, но может в таких выражениях отсутствовать (в этом случае частичная блокировка по семантике не отличается от обычной):

```
>>> '(1 (+ 2 3) 4)
(1 (+ 2 3) 4)
>>> `(1 (+ 2 3) 4)
(1 (+ 2 3) 4)
>>> `(1 ,(+ 2 3) 4)
(1 5 4)
```

Этот механизм также называется «квазиквотированием» (например, такой терминологии придерживались создатели языка Scheme [21]).

Выражения, содержащие частичную блокировку, могут иметь произвольную степень вложенности. В таком выражении будут вычисляться не все части, перед которыми стоит запятая, а те, к которым эта операция была применена достаточное количество раз (столько же, сколько было вложенных блокировок вычислений с учётом отмены блокировки на верхних уровнях).

Можно сказать, что у выражения есть некоторая целочисленная характеристика, которую в дальнейшем будем называть «степенью блокировки». Степень блокировки выражения и всех его подвыражений изначально равна нулю. Значение увеличивается на единицу, если к выражению применяется частичная блокировка вычислений, и уменьшается на единицу, если перед выражением стоит запятая (отмена блокировки вычислений). Тогда в выражении, вычисление которого частично заблокировано, будут вычисляться только те подвыражения, степень блокировки которых равна нулю. Заметим, что степень блокировки выражения не может быть строго отрицательной. Такой случай считается синтаксической ошибкой. Приведём несколько примеров:

```
>>> ``,(+ 1 2)
`,(+ 1 2)
>>> `,`,(+ 1 2)
3
>>> ``,,(+ 1 2)
3
>>> `,`,,(+ 1 2)
Error: illegal unquote symbol
>>> ,`(+ 1 2)
Error: illegal unquote symbol
```

Символы обратной блокировки и отмены блокировки вычислений не обязаны идти подряд, а могут встречаться в разных частях вложенных выражений:

```
>>> `(X `(Y ,(LIST FOO) ,(LIST BAR ,(+ 1 2)) A) B)
(X `(Y ,(LIST FOO) ,(LIST BAR 3) A) B)
```

Вернёмся к рассмотрению макроса SWAP-MACRO. С использованием механизма обратной блокировки можно предложить следующую реализацию:

```
(DEFMACRO SWAP-MACRO (X Y)
    `(LET ((TMP ,X))
        (SETQ ,X ,Y)
        (SETQ ,Y TMP)
    )
)
```

При вызове (SWAP-MACRO A B) будет сгенерирован тот же код, что и для предыдущей реализации, в которой это выражение составлялось «вручную»:

```
(LET ((TMP A))
    (SETQ A B)
    (SETQ B TMP)
)
```

Несомненным преимуществом последнего варианта реализации является его схожесть с результирующим кодом, что упрощает создание макросов и понимание кода.

Описанный выше механизм является достаточно минималистичным, но обладает высокой выразительной мощностью. В большинстве случаев применения частичной блокировки для написания макросов не используются вложенные блокировки степени больше двух.

В диалекте Common Lisp [7] квазиквотирование является синтаксическим сахаром, поэтому обратная блокировка любой степени вложенности раскрывается целиком и преобразовывается в новое выражение, включающее в себя вызовы функций LIST, QUOTE, CONS или APPEND. Результат такого раскрытия в общем случае может зависеть от реализации. Например, в ходе преобразования кода можно получить следующие выражения:

```
>>> ``(`x ,(y z))
(LIST ''X (Y Z))
>>> ``(`x (y z))
(CONS ''X '((Y Z)))
```

Подобные преобразования неочевидны, а исходный и полученный код больше похожи на эзотерические конструкции языка, которые, кроме того, довольно редко применяются на практике. Ещё один недостаток такого подхода к реализации квазиквотирования состоит в том, что неявно делается предположение о значении символов LIST, QUOTE, CONS и других, входящих в состав выражения, полученного при раскрытии обратной блокировки. Эти символы должны быть гарантированно связаны с функциями с соответствующей стандартной семантикой в контексте вычисления преобразованного выражения, что, вообще говоря, не всегда так: пользователь может переопределить значения этих символов.

В приведённой далее реализации квазиквотирование не рассматривается как синтаксический сахар. В выражениях, вычисление которых частично заблокировано, вычисляются только те подвыражения, степень блокировки которых равна нулю. Квазиквотирование со схожей семантикой также описано в [21].

4.2 Внутреннее представление шаблона кода

Выражение, к которому применена частичная блокировка вычислений, можно рассматривать как шаблон кода, в котором определённые подвыражения должны быть заменены на результат вычисления этих подвыражений.

С точки зрения реализации S-выражений в библиотеке InteLib, выражение-шаблон должно иметь специальное представление со своим методом вычисления. Такие выражения являются наследниками класса `SExpressionSpecialToken`. В частности, подобным образом реализован класс, представляющий подготовленную к вычислению LET-форму. Подробности реализации этого класса описаны в работе [1].

Для представления выражений-шаблонов в ходе работы был добавлен класс `SExpressionCompiledTemplate`. Класс инкапсулирует тело шаблона, список аргументов (выражений, которые должны быть заменены в шаблоне), количество аргументов и флаг, отвечающий за способ их обработки. Тело шаблона также должно быть предварительно обработано: предполагается, что вместо тех подвыражений, которые должны быть заменены, находятся позиционные параметры (объекты класса `SExpressionPositionParam`), которым соответствуют значения в списке аргументов.

Возможны два способа обработки аргументов перед подстановкой их в шаблон. В простейшем случае шаблон может быть использован для синтаксической замены, тогда аргументы подставляются в шаблон в исходном виде. Этот вариант применяется для реализации синтаксических правил в диалекте InteLib Scheme (см. § 5.3).

В случае реализации квазиквотирования аргументы должны быть вычислены, а в тело шаблона вместо позиционных параметров будут подставлены соответствующие значения, т. е. подвыражения заменяются на результат их вычисления. При этом остальные части выражения не изменяются. С точки зрения реализации, эти два случая отличаются только командой вычислителя, используемой при подготовки вектора аргументов. Выбор команды производится на основании значения флага, который передаётся в конструктор класса.

4.3 Реализация форм `BACKQUOTE` и `UNQUOTE`

Для использования класса `SExpressionCompiledTemplate` реализация квазиквотирования должна включать в себя определение вычисляемых подвыражений, а именно их замену на позиционные параметры в теле шаблона и составление списка будущих аргументов шаблона.

В предложенной реализации частичная блокировка реализована в виде спецформы `BACKQUOTE`, которая может быть применена только к одному аргументу, рассматриваемому в качестве шаблона кода. Реализация представляет собой класс `SFormBackquote`, который наследуется от класса `SExpressionForm`. Как было неоднократно упомянуто ранее, спецформы не вычисляют свои аргументы перед вызовом, и поэтому используются для реализации средств метапрограммирования.

С точки зрения синтаксиса, запись `‘X` эквивалентна `(BACKQUOTE X)` и имеет такое же внутренне представление в виде списка. Аналогично рассматриваются выражения `,X` и `(UNQUOTE X)`.

Обработка аргумента формы `BACKQUOTE` происходит рекурсивно. Перед началом этого процесса счётчик степени блокировки устанавливается равным единице. Если S-выражение является атомом, оно не изменяется. Если выражение представляет собой точечную пару, дальнейшая обработка зависит от первого символа (головы списка). Выделяются два особых случая: символ частичной блокировки и символ отмены блокировки (во внутреннем представлении соответствуют символам `BACKQUOTE` и `UNQUOTE`).

Если первым символом обрабатываемой формы является `BACKQUOTE`, происходит увеличение счётчика степени блокировки. Затем рекурсивно обрабатывается аргумент вложенного вызова. Результатом обработки всей формы является выражение вида `(BACKQUOTE` <*результат обработки аргумента*>`)`.

Если встречается символ `UNQUOTE`, счётчик степени блокировки уменьшается на единицу. Дальнейшие действия зависят от полученного значения счётчика. Если оно строго положительно, то аргумент формы `UNQUOTE` не должен вычисляться, но необходимо учесть возможные вложенные вызовы формы, в результате которых блокировка вычислений будет отменена достаточное количество раз. Поэтому в таком случае аргумент формы рекурсивно обрабатывается, а в качестве результата строится выражение `(UNQUOTE` <*результат обработки аргумента*>`)`. Если счётчик блокировки достиг нуля, то выражение должно быть заменено на результат его вычисления. Поэтому в тело шаблона вместо этого выражения добавляется позиционный параметр, а само выражение включается в список аргументов шаблона.

Все остальные списочные структуры обрабатываются одинаково: голова и хвост списка обрабатываются рекурсивно и соединяются с сохранением структуры выражения. В результате в теле шаблона происходит замена на позиционные параметры выражений, которые должны быть вычислены, а из самих выражений строится список аргументов шаблона.

В качестве примера рассмотрим следующее выражение с вложенным квазиквотированием:

```
‘(,(+ 1 2) ‘(,(LIST X ,(LIST 42))))
```

В этом случае должны быть вычислены подвыражения `(+ 1 2)` и `(LIST 42)`, а шаблон будет преобразован к следующему виду (`#1` и `#2` — обозначения позиционных параметров):

```
(#1 `(,(LIST X #2)))
```

В приведённом примере предполагается, что вычисление формы происходит на верхнем уровне, но в подавляющем большинстве случаев частичная блокировка вычислений используется в контексте, в частности, при определении пользовательских макросов.

Контекст вычислений должен быть учтён при вычислении аргументов шаблона, а создание самого шаблона должно происходить при создании лексического замыкания. Поэтому основная часть реализации вынесена в метод `ReplaceTokens` класса, представляющего спецформу `BACKQUOTE`, а также в методы класса `IntelibTokenReplace`, добавленного для реализации контекстов вычисления с использованием позиционных параметров. Если форма `BACKQUOTE` вызывается на верхнем уровне, то перед преобразованием аргумента в шаблон создаётся объект класса `IntelibTokenReplace` (точнее, одного из его наследников, в зависимости от диалекта) с пустым контекстом замены.

Степень блокировки вычислений была добавлена как поле класса `IntelibTokenReplace`. Для работы с этим значением были добавлены методы `EnterMacro` и `ExitMacro` для увеличения и уменьшения степени блокировки соответственно, а также метод `InMacro` для проверки равенства нулю этого значения. Такое решение обусловлено способом реализации синтаксических правил для диалекта InteLib Scheme, где замена на позиционные параметры также выполняется по-разному в зависимости от того, происходит ли этот процесс во время преобразования кода или во время создания лексического замыкания. Добавление этого значения непосредственно в класс `IntelibTokenReplace` обеспечивает универсальность средств и удобство реализации.

Описанный выше процесс создания шаблона реализован в методах `TemplateProcessing` и `ProcessBackquote` класса `IntelibTokenReplace`. Первый из них вызывается из метода `ReplaceTokens` класса `SFormBackquote` при обработке формы `BACKQUOTE` и устанавливает начальные значения счётчика количества параметров и накопительного параметра для списка аргументов. Перед вызовом этого метода вызывается метод `EnterMacro`.

Метод `ProcessBackquote` является вспомогательным закрытым методом класса и реализует основную логику. Для того, чтобы учесть контекст вычислений, перед добавлением очередного найденного аргумента в список к нему применяется метод `ReplaceTokens` класса `IntelibTokenReplace`, реализующий самый общий механизм замены символов на позиционные параметры.

Например, если квазиквотирование используется для объявления макроса и в аргументе шаблона присутствует символ, обозначающий один из его аргументов, то он будет заменён на позиционный параметр с соответствующим номером.

В результате работы метода формируется список аргументов и тело шаблона с позиционными параметрами. Затем на основе этих значений в методе `TemplateProcessing` создаётся объект класса `SExpressionCompiledTemplate`, который, в свою очередь является результатом работы метода `ReplaceTokens` класса `SFormBackquote`. Перед тем, как вернуть результат, вызывается метод `ExitMacro` для восстановления значения степени блокировки.

Проиллюстрируем создание шаблона и обработку аргументов на примере определения макроса `SWAP-MACRO`:

```
(DEFMACRO SWAP-MACRO (X Y)
    `(LET ((TMP ,X))
        (SETQ ,X ,Y)
        (SETQ ,Y TMP)
    )
)
```

Здесь обратная блокировка применяется один раз, поэтому должны быть вычислены все выражения, для которых осуществляется отмена блокировки. Тело макроса будет представлено в виде анонимной функции (см. § 3.2) от двух аргументов, которые должны быть заменены на позиционные параметры. Поэтому при обработке формы `BACKQUOTE` будет действовать контекст замены, в котором символы `X` и `Y` заменяются на параметры с номерами 1 и 2 соответственно.

После обработки аргумента формы `BACKQUOTE` будет создан шаблон, в котором четыре подвыражения должны быть вычислены. Он имеет следующий вид:

```
(LET ((TMP #1))
    (SETQ #2 #3)
    (SETQ #4 TMP)
)
```

В список аргументов должны войти выражения, заменённые на позиционные параметры. Без учёта контекста этот список выглядел бы следующим образом: `(X X Y Y)`. Поскольку аргументы шаблона подвергаются предварительной обработке с учётом внешнего контекста, они будет заменены на позиционные параметры, и аргументы результирующего шаблона будет иметь

следующий вид: `(#1 #1 #2 #2)`. Как в теле шаблона, так и в списке аргументов позиционные параметры обозначают номер аргумента в фрейме, находящемся на вершине стека фреймов, поэтому параметры с номерами 1 и 2 не конфликтуют друг с другом во время вычисления.

Несмотря на то, что отмена частичной блокировки не может быть использована на верхнем уровне, в диалекты InteLib Lisp и InteLib Scheme была добавлена спецформа `UNQUOTE` исключительно для того, чтобы генерировать специальное исключение при вызове. Сам по себе вызов спецформы не осуществляется, если она была использована внутри заквотированного выражения.

4.4 Вычисление обработанного шаблона кода

Ранее было сказано, что обработанный шаблон кода представляет собой объект класса, наследуемого от `SExpressionSpecialToken`, а поэтому обладающего особым способом вычисления. Далее будет рассмотрена реализация вычисления шаблона в случае квазиквотирования.

Поскольку вычисляемые подвыражения заменены на позиционные параметры, то перед раскрытием шаблона необходимо подготовить стек результатов таким образом, чтобы в результирующем стековом фрейме аргументы находились на правильных позициях. Кроме того, аргументы шаблона должны быть предварительно вычислены.

Результатом раскрытия шаблона должно быть выражение, в котором все позиционные параметры заменены на соответствующие значения, полученные из стека. В ходе работы была добавлена команда континуации `eval_posit_params`, реализующая этот процесс. Команда схожа с командой `CALL/n`, используемой для вызова функции с `n` аргументами. Параметром команды является количество аргументов шаблона (далее также обозначается как `n`). Использование команды требует предварительной подготовки стека результатов. Предполагается, что первые `n` элементов в стеке содержат выражения, на которые должны быть заменены позиционные параметры в теле шаблона, а перед ними располагается само тело шаблона.

Выполнение команды начинается с формирования стекового фрейма. Определяется позиция, на которой находится тело шаблона. Из первых `n` элементов стека результатов формируется новый фрейм, который помещается на вершину стека фреймов. Элементы именно этого фрейма будут использоваться при вычислении позиционных параметров в теле шаблона. Этот процесс реализован рекурсивно: в списке отдельно обрабатываются голова и хвост, а атомарные составляющие шаблона вычисляются в зависимости от типа. Если выражение представляет собой позиционный параметр, т. е. объект класса

SExpressionPositionParam, то вызывается метод Evaluate этого класса, что приводит к получению значения, находящемуся на соответствующей позиции в стековом фрейме. Это значение подставляется в выражение вместо позиционного параметра. Остальные атомарные выражения остаются неизменными. Результат работы команды помещается на вершину стека результатов.

При вычислении объекта класса SExpressionCompiledTemplate в стек действий континуации помещается инструкция eval_posit_params с параметром, равным количеству аргументов шаблона. Затем помещаются инструкции для подготовки аргументов. В случае квазиквотирования используется команда just_evaluate, вычисляющая свой параметр. Также может быть использована инструкция quote_parameter, которая помещает свой параметр в стек результатов в исходном виде. Наконец, в стек помещается команда для квотирования тела шаблона. На рис. 4.1 показана конфигурация стека действий после вызова метода Evaluate класса SExpressionCompiledTemplate. Для тела шаблона используется обозначение $<body>$, аргументы шаблона обозначены $arg_1 ... arg_n$.

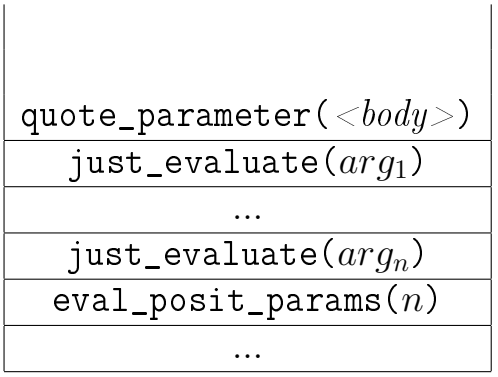

Рис. 4.1: Содержимое стека действий при подготовке к раскрытию шаблона.

В результате выполнения этих команд стек результатов будет подготовлен к корректному исполнению инструкции eval_posit_params. Необходимая конфигурация стека показана на рис. 4.2. Выражения, которые будут получены при вычислении аргументов шаблона, обозначены $val(arg_1) ... val(arg_n)$.

Описанная выше обработка команды континуации была реализована в методе EvaluatePositParams класса IntelibContinuation.

Заметим, что поскольку создание нового стекового фрейма происходит при выполнении команды eval_posit_params, позиционные параметры, которые могут присутствовать в аргументах шаблона, указывают на позиции в текущем на тот момент фрейме. Благодаря этому учитывается контекст раскрытия шаблона. Это свойство наглядно иллюстрируется при вычислении макроса SWAP-MACRO. Выше уже был указан шаблон, полученный при обработке тела макроса, и список его аргументов, в котором присутствуют

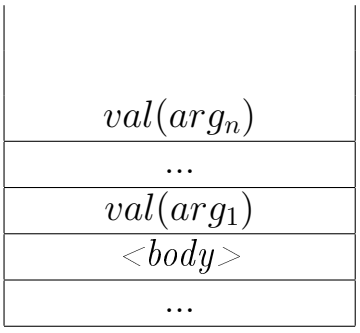

Рис. 4.2: Содержимое стека результатов непосредственно перед раскрытием шаблона.

позиционные параметры, относящиеся к аргументам макроса (но не шаблона). Рассмотрим следующий вызов макроса:

```
>>> (SETQ A 0)
0
>>> (SETQ B 1)
1
>>> (SWAP-MACRO A B)
```

При вызове макроса будет сформирован стековый фрейм из двух элементов: символов `A` и `B` (напомним, что аргументы макроса не вычисляются перед вызовом). Во время раскрытия шаблона, а именно при подготовке стека результатов, будут вычислены выражения `#1` и `#2` — позиционные параметры, указывающие на элементы последнего стекового фрейма, т. е. именно того, который был добавлен при вызове макроса. В итоге в стек результатов будут помещены по два элемента, содержащие символы `A` и `B`. Таким образом, аргументы `#1`, `#1`, `#2`, `#2` будут вычислены в `A`, `A`, `B`, `B`. Далее вычислитель перейдёт к выполнению команды `eval_posit_params`, и из этих значений будет сформирован стековый фрейм. При обработке тела шаблона позиционные параметры `#1` и `#2` будут вычисляться в символ `A` (первые два элемента фрейма), а `#3` и `#4` — в символ `B` (оставшиеся два элемента). В результате будет получено следующее выражение:

```
(LET ((TMP A))
    (SETQ A B)
    (SETQ B TMP)
)
```

Результат вычисления частично заблокированного выражения идентичен результату макрорасширения в случае реализации, не использующей квазиквотирование. Дальнейшие действия в рамках вызова макроса осуществляются аналогично.

4.5 Поддержка квазиквотирования в трансляторах

Для корректной обработки частично заблокированных выражений потребовалось внести изменения в трансляторы ILL и ILS и интерпретаторы NILL и NILS.

Во-первых, были добавлены служебные классы `LBackquoteConstructor` и `SchBackquoteConstructor` для построения выражений с частичной блокировкой вычисления и `LUnquoteConstructor` и `SchUnquoteConstructor` для отмены блокировки. Единственной целью определения этих классов служит перегрузка операции `^`, которая преобразует переданный аргумент (объект класса `LReference` или `SchReference`) таким образом, чтобы результат был идентичен добавлению символов `BACKQUOTE` и `UNQUOTE`. Например, это позволяет использовать следующие конструкции на языке C++:

```
LFunctionalSymbol<LFunctionList> LIST("LIST");
LFunctionalSymbol<LFunctionPlus> PLUS("PLUS");

LBackquoteConstructor BQ;
LUnquoteConstructor UQ;
LListConstructor L;

SReference res = (BQ^(L| LIST, 1, 2, UQ^(L| PLUS, 3, 4))).Evaluate();
```

После выполнения такого фрагмента кода в переменной `res` будет содержаться «умный указатель» на S-выражение, полученное при вычислении выражения `‘(1 2 ,(LIST 3 4))`.

Трансляторы ILL и ILS были дополнены возможностью преобразования конструкций диалектов InteLib Lisp и InteLib Scheme `‘X` или `(BACKQUOTE X)` и `,X` или `(UNQUOTE X)` в конструкции языка C++ `BQ^Y` и `UQ^Y` соответственно, где `Y` — результат трансляции выражения `X`.

Реализация интерпретаторов NILL и NILS основана на использовании классов `LispReader` и `SchemeReader`, наследуемых от класса `IntelibReader`. Этот позволяет создавать конфигурируемые синтаксические анализаторы исходного кода языков, основанных на обработке S-выражений. Интерфейс класса даёт возможность добавлять в анализатор специальные символы (например, `#T` и `#F` в диалекте InteLib Scheme), символы-ограничители, указывать синтаксис комментариев и т. д. В частности, с помощью метода `AddQuoter` в анализатор добавляются инструкции для обработки выражений, следующих за каким-либо символом. Например, так реализуется обработка сокращённых вариантов синтаксиса для форм `QUOTE` и `FUNCTION` в диалекте InteLib Lisp в конструкторе класса `LispReader`:

```
LispReader::LispReader()
{
    ...
    AddQuoter("'", QuoteExpression);
    AddQuoter("#'", FunquoteExpression);
    ...
}
```

Для поддержки квазиквотирования достаточно добавить в конструкторы классов `LispReader` и `SchemeReader` следующие инструкции:

```
AddQuoter("`", BackquoteExpression);
AddQuoter(",", UnquoteExpression);
```

Функции `BackquoteExpression` и `UnquoteExpression` принимают в качестве аргумента выражение, следующее за символами ` и , соответственно, и используют объекты описанных выше классов `[L|Sch]BackquoteConstructor`, и `[L|Sch]UnquoteConstructor` для преобразования выражения.

4.6 Примеры использования квазиквотирования

Для иллюстрации использования частичной блокировки приведём примеры объявления тех же макросов, что и в § 3.4. В данном случае получена более лаконичная реализация, с большей степенью выразительности. Кроме того, исходный код позволяет понять, какое выражение будет результатом макрорасширения, поскольку практически полностью с ним совпадает.

В реализациях макросов `AND` и `OR` генерация кода осуществляется в зависимости от количества аргументов (вызов этих макросов с одним аргументов эквивалентен вычислению значения этого аргумента). Проверка этого условия необходима для того, чтобы избежать бесконечного раскрытия вложенных вызовов макроса.

```
(DEFMACRO AND (X . XS)
    (IF (NULL XS)
        X
        `(IF ,X (AND . ,XS) NIL)
    )
)

(DEFMACRO OR (X . XS)
    (IF (NULL XS)
```

```
        X
        `(LET ((TMP ,X))
            (IF TMP TMP (OR . ,XS))
        )
    )
)

(DEFMACRO SWAP (X Y)
    `(LET ((TMP ,X))
        (SETQ ,X ,Y)
        (SETQ ,Y TMP)
    )
)
```

5 Синтаксические правила в языке Scheme

5.1 Объявление макросов на основе синтаксических правил

Язык Scheme предоставляет более выразительные средства метапрограммирования, помимо возможности объявления макросов стандартным способом и использования квазиквотирования. Более универсальным механизмом являются так называемые «синтаксические правила», работающие на основе сопоставления аргументов с образцом, в зависимости от которого выбирается шаблон для дальнейшего преобразования кода.

С точки зрения языков программирования, сопоставление с образцом — это механизм обработки данных, при котором дальнейшие действия зависят от структуры входного значения. Как правило, конструкции, использующие сопоставление с образцом представляют собой набор пар «образец – действие», где образец является записью некоторой структуры данных языка, в которой могут присутствовать константы, переменные, конструкторы значений, вложенные образцы, и специальные символы подстановки, соответствующие произвольному выражению (*wildcard*).

Язык определяет правила, по которым значение и образец считаются сопоставимыми. Правые части такого набора правил могут включать в себя имена переменных, используемых в шаблоне. Эти переменные получат значения подвыражений входного выражения, которые соответствуют им при успешном сопоставлении. Если входные данные сопоставляются с несколькими образцами, в большинстве случаев из набора возможных подходящих действий выбирается первое.

Сопоставление с образцом широко используется в декларативных языках программирования: например, в правилах логического вывода в языке Prolog или при определении функций на Haskell. Среди диалектов языка Lisp можно выделить Clojure [27], в котором сопоставление с образцом используется, например, в вариадических функциях, а также реализовано в отдельных

библиотеках. В более простом виде сопоставление с образцом используется и в императивных языках программирования: тривиальным примером могут служить конструкции вида `switch ... case`, в которых, как правило, в качестве шаблона могут использоваться только константные выражения.

Образцы в синтаксических правилах языка Scheme представляются в виде S-выражений одного из следующих видов:

- идентификатор (символ);
- константный атом (например, целое число или строка);
- символ нижнего подчёркивания (`_`), который играет роль символа подстановки;
- точечная пара, составленная из образцов.

Среди идентификаторов могут выделяться символы-литералы, играющие роль «ключевых слов», которые сопоставляются только с самими собой. Поскольку определение рекурсивно, можно сказать, что образец является произвольным S-выражением любой степени вложенности, в котором некоторые атомы имеют собственное правило сопоставления.

Само синтаксическое правило записывается как список из двух элементов (*<образец> <шаблон>*), причём в качестве шаблона может использоваться любое выражение. Приведём несколько примеров синтаксических правил:

```
(
    (_ _ _ _ X)
    (BEGIN (DISPLAY "Fifth element: ") (DISPLAY X))
)

(
    (TEST (WE _ (NEED (TO (GO (DEEPER . _))))) . _)
    (DISPLAY "So deep even Adele can't roll in it")
)

(
    (3 2 1 X . REST)
    (BEGIN (DISPLAY "After countdown: ") (DISPLAY (X . REST)))
)
```

Сопоставление выражения `S` с образцом `P` происходит по следующим правилам:

- атомарная константа сопоставляется с равной ей константой в смысле предиката `EQUAL?`;

- символы-литералы сопоставляются только с самими собой;
- символ подстановки (`_`) сопоставляется с любым выражением;
- идентификаторы сопоставляются **и связываются** с любым выражением, причём эта связь действительна только в соответствующем шаблоне;
- если `P` — точечная пара, то `S` сопоставляется с `P`, если `S` — точечная пара и `(CAR S)` сопоставляется с `(CAR P)`, а `(CDR S)` — с `(CDR P)`.

Например, с образцом `(_ _ _ _ X)` сопоставляются только списки из пяти элементов, причём в соответствующем шаблоне символ `X` будет связан с последним элементом этого списка. С образцом `(3 2 1 X . REST)` сопоставляются точечные пары, состоящие как минимум из четырёх элементов, первые три из которых должны быть равны указанным константам. Символ `X` сопоставится с четвёртым элементом списка, а с символом `REST` — список всех остальных элементов (возможно, пустой). Сопоставление вложенных образцов происходит аналогично. Например, с образцом `((X) (Y _) (Z _ _))` сопоставится список `((1) 'quoted (4 "foo" "bar"))`, но не `((1) (2 3 4) ("foo" "bar" "baz"))`, поскольку его второй элемент не сопоставляется со вторым элементом образца.

Наличие двух одинаковых идентификаторов в одном образце не означает, что сопоставленные им значения должны быть равны. Такая ситуация приводит к неопределённой связи символа в теле шаблона, и дальнейшее поведение, вообще говоря, зависит от реализации.

Объявление пользовательских макросов с использованием синтаксических правил осуществляется следующим образом:

```
(DEFINE-SYNTAX NAME
    (SYNTAX-RULES <keywords>
        (<pattern> <template>)
        ...
    )
)
```

Здесь `NAME` — имя макроса, `<keywords>` — список литералов (локальных ключевых слов), после которого следует произвольное количество синтаксических правил. В этом случае на образец налагается дополнительное ограничение: это всегда должен быть список хотя бы из одного элемента. Первый элемент списка соответствует имени макроса в форме вызова и игнорируется при сопоставлении и последующей обработке шаблона.

Синтаксис вызова макроса совпадает с вызовом функции. Так же как и в случае макросов языка Lisp, макрос на основе синтаксических правил не вычисляет свои аргументы перед вызовом.

Выражением, которое сопоставляется с образцом при вызове макроса, является вся форма вызова, включая имя макроса, т. е. (NAME arg_1 ... arg_n). Далее синтаксические правила рассматриваются по очереди в порядке описания в теле макроса. Выражение сопоставляется с каждым образцом по описанным выше правилам. Процесс останавливается либо если сопоставление произошло успешно, либо если выражение не может сопоставиться ни с одним образцом. Последняя ситуация приводит к ошибке и аварийному завершению.

При успешном сопоставлении с образцом происходит обработка соответствующего шаблона кода. Символы-идентификаторы, присутствующие в образце, были связаны со значениями в процессе сопоставления. С учётом этих связей происходит дальнейшее преобразование шаблона кода до те пор, пока в нём можно производить замены (например, в случае вложенных вызовов макросов, в том числе рекурсивных). После того, как все замены завершены, полученное выражение вычисляется.

У макросов языка Scheme есть ряд особенностей, касающихся контекстов вычисления и действия связи символов.

Во-первых, если в теле макроса какой-либо символ связывается со значением, эта связь действует только в пределах лексического контекста этого макроса. Например, связь идентификатора с сопоставленным значением действует только при преобразовании шаблона на верхнем уровне. Поясним это свойство на примере следующего макроса:

```
(DEFINE-SYNTAX BINDING-TEST
    (SYNTAX-RULES ()
        ((_ _) X)
        ((_ _ X) (TEST X))
    )
)
```

При вызове макроса с одним аргументом, возвращается символ X (первое правило), при вызове с двумя аргументами происходит вложенный вызов макроса, в который передаётся второй аргумент первого вызова (второе правило). Во втором правиле для сопоставления со вторым аргументом вызова также используется символ X. Макровызов (BINDING-TEST 1 2) в завершится ошибкой: при раскрытии шаблона во втором правиле символ X связан со значением 2, но при раскрытии первого правила эта связь уже не действует.

Во-вторых, если при преобразовании выражений в результирующей форме появляются свободные символы (для которых нет связи в контексте тела макроса), то считается, что символ должен иметь связь в контексте определения макроса, в отличие от макросов языка Lisp, где при вычислении расширенного выражения действует контекст макровызова. Например, если вызвать тот же макрос BINDING-TEST в контексте, где действует связь для символа X, то ошибка сохранится:

```
>>> (LET ((X 100)) (BINDING-TEST 1 2))
Error: symbol has no value: X
```

В-третьих, макрос может быть объявлен в контексте, где символы, являющиеся ключевыми словами этого макроса, связаны с каким-либо значением. Такая связь не учитывается при вызове макроса, сопоставлении с образцом и макрорасширении. При этом локальные связи, устанавливаемые в теле макроса могут связывать ключевые слова:

```
>>> (DEFINE-SYNTAX KEYWORD-EXAMPLE
  >     (SYNTAX-RULES (KEY)
  >         ((_ KEY) (LET ((KEY "LOCAL")) KEY))
  >         ((_ KEY "KEYWORD") KEY)
  >     )
  > )
>>> (KEYWORD-EXAMPLE KEY)
"local"
>>> (LET ((KEY "OUTER")) (KEYWORD-EXAMPLE KEY "KEYWORD"))
Error: symbol has no value: KEY
```

При макрорасширении может возникнуть ряд проблем из-за совпадающих идентификаторов и непредвиденного изменения связи символов. Во многих реализациях диалектов языка Lisp подобные проблемы решаются обфускацией идентификаторов и генерацией уникальных временных символов, которые используются при макрорасширении. К сожалению, в общем случае это не гарантирует отсутствие совпадений с идентификаторами, введёнными пользователем.

Макросы на основе синтаксических правил в языке Scheme называются гигиеническими, поскольку трансформируют выражения с учётом особенностей изменения связей символов. Этот термин был впервые представлен в работе [28], где подробно описаны проблемы, возникающие при макрорасширении, а также приводится механизм гигиенических преобразований выражений.

5.2 Внутреннее представление в диалекте InteLib Scheme

С точки зрения внутреннего представления, синтаксические правила должны представлять собой некоторый объект, который можно применить к произвольному числу аргументов. В ходе вызова такого объекта должны происходить основные процессы, необходимые для макрорасширения: сопоставление с образцом и выбор шаблона, установка связывания для сопоставленных символов, раскрытие шаблона и вычисление полученного выражения.

Как и макрос языка Lisp, синтаксические правила языка Scheme были реализованы в виде класса, наследуемого от класса `SExpressionForm`, что позволяет передавать в макровызов выражения без предварительного вычисления.

Класс `SchExpressionCompiledSyntaxRules` инкапсулирует список пар (*<образец> <шаблон>*) и список ключевых слов макроса. Предполагается, что в образце и шаблоне уже учитывается связь локальных идентификаторов, которые будут связаны с сопоставленными значениями. В классе реализованы виртуальные методы `Call` и `ReplaceTokens`, а также несколько вспомогательных методов, реализующих сопоставление с образцом. Более подробно реализация этих методов будет описана в § 5.3.

В диалекте InteLib Scheme единственным способом использования синтаксических правил является определение пользовательских макросов с помощью формы `DEFINE-SYNTAX`, которая также была реализована в виде класса, наследующегося от `SExpressionForm`.

При обработке аргументов формы сначала выделяется символ, обозначающий имя будущего макроса. С этим символом впоследствии будет связан объект класса `SchExpressionCompiledSyntaxRules`, полученный при дальнейшей обработке аргументов.

Вторым аргументом формы должно быть непосредственно описание синтаксических правил, поэтому происходит проверка корректности этого объявления: первым символом должен быть символ `SYNTAX-RULES`, затем должен идти список ключевых слов и, наконец, список пар (*<образец> <шаблон>*).

Основная часть реализации спецформы `DEFINE-SYNTAX` состоит из обработки списка образцов и шаблонов. Для каждого синтаксического правила необходимо определить, какие символы в шаблоне будут связаны с сопоставленными аргументами, и заменить эти символы на позиционные параметры. Для этого производится рекурсивный проход по выражению-образцу, из которого предварительно удаляется первый элемент (он обозначает имя макроса при вызове и игнорируется при сопоставлении). Базой рекурсии является обработка атомарного выражения. Если выражение представляет собой константу, ключевое слово или символ подстановки, оно игнорируется. Во всех остальных случаях считается, что атом является идентификатором и будет связан с сопоставленным значением, поэтому он добавляется в аккумулирующий список. В результате будет получен список символов, которые должны быть заменены на позиционные параметры.

Дальнейшие действия требуют использования объекта класса `SchemeTokenReplace`. Для сопоставления символов позициям вызывается метод `AssignPositions`, которому передаётся список символов, полученный на предыдущем этапе. Метод создаёт контекст замены, т. е. сохраняет соответствия символов и номеров позиций. Далее к выражению-шаблону

применяется метод ReplaceTokens, в результате чего эти символы будут заменены на позиционные параметры в теле шаблона. То же самое проделывается и для образца с целью упрощения сопоставления при вызове макроса.

Поскольку раскрытие шаблона представляет собой синтаксическую замену, не зависящую от семантики входящих в него выражений, вызов метода ReplaceTokens требует предварительных действий. При работе в обычном режиме метод обрабатывает спецформы специфическим образом, причём результат зависит от семантики формы (например, LET-формы в результате представляются объектами класса SExpressionCompiledLet, а при использовании анонимных функций происходит создание лексического замыкания). Для того, чтобы этого избежать, перед вызовом ReplaceTokens вызывается метод EnterMacro, увеличивающий счётчик блокировки вычислений. Его значение проверяется в методах ReplaceTokens каждой спецформы, обработка которой требует нетривиальных действий. Если этот процесс происходит внутри шаблона, форма рассматривается как обычный список. Например, метод ReplaceTokens для формы LET был преобразован следующим образом:

```
SReference SchFunctionLet::ReplaceTokens(const SReference &form,
                              IntelibContinuation &cont,
                              IntelibTokenReplace &tr) const
{
    if (tr.InMacro())
        return SReference(form.Car(),
                          tr.ReplaceTokensInList(form.Cdr(), cont));
    else
        return tr.LetProcessing(form.Cdr(), cont, false);
}
```

Рассмотрим в качестве примера обработку правил, используемых в макросе COND=>, который реализует альтернативный синтаксис существующей формы COND, в котором условие отделено от действий символом =>:

```
(DEFINE-SYNTAX COND=>
    (SYNTAX-RULES (=> ELSE)
        ((_ (ELSE => . BODY)) (BEGIN . BODY))
        ((_ (CONDITION => . BODY) . REST)
            (IF CONDITION
                (BEGIN . BODY)
                (COND=> . REST)
            )
        )
    )
)
```

Ключевыми словами макроса являются символы => и ELSE. В первом правиле на позиционный параметр необходимо заменить символ BODY, во втором — символы CONDITION, BODY и REST. После выполнения всех необходимых замен будет получено следующее выражение:

```
(
    (((ELSE => . #1)) (BEGIN . #1))
    (((#1 => . #2) . #3)
        (IF #1
            (BEGIN . #2)
            (COND=> . #3)
        )
    )
)
```

Таким образом гарантируется связь сопоставленных символов только в области видимости макроса, что является одним из перечисленных выше свойств, приводящих в гигиеничности макросов языка Scheme.

После того, как будут обработаны все правила, создаётся объект класса SchExpressionCompiledSyntaxRules. В конструктор передаются обработанные синтаксические правила и список ключевых слов. Полученный объект устанавливается как значение символа, обозначающего имя макроса. Результатом работы формы является неопределённое (*unspecified*) значение, которое возвращается в континуацию с помощью специального метода ReturnUnspecified.

Ещё несколько изменений, необходимых для реализации синтаксических правил, связаны с модификацией транслятора ILS и интерпретатора NILS. Символы _ и SYNTAX-RULES были объявлены специальными символами, которые преобразуются во внутреннее представление в виде уникального объекта. Это позволяет анализировать образцы в синтаксических правилах и проверять синтаксис объявления макросов с их использованием. При этом указанные символы могут быть использованы вне объявления макросов наравне с другими символами (в отличие от символов #T и #F).

5.3 Сопоставление с образцом и макрорасширение

Вычисление макровызова в предложенной реализации представляет собой процесс, состоящий из нескольких основных этапов:

1. сопоставление аргументов вызова с образцом и выбор соответствующего шаблона;

2. предварительная обработка аргументов для учёта контекста вызова;

3. раскрытие шаблона с учётом связей, возникших при сопоставлении с образцом;

4. вычисление выражения, полученного при макрорасширении.

Все эти процессы были реализованы в классе `SchExpressionCompiledSyntaxRules`, который представляет собой особый вид спецформы. Как и для других классов спецформ, метод `Call` реализует вызов спецформы, метод `ReplaceTokens` вызывается при создании лексического замыкания в объемлющей форме (например, если макровызов осуществляется в теле функции). Как и в случае макросов языка Lisp, в этом случае должно происходить макрорасширение, поэтому этапы 1–3 были реализованы в методе `ReplaceTokens`.

При сопоставлении с образцом, во-первых, определяется шаблон, который будет использован для макрорасширения, и, во-вторых, составляется список аргументов шаблона — выражений, которые сопоставились с идентификаторами. синтаксические правила рассматриваются по очереди, пока не произойдёт успешное сопоставление с образцом.

Реализация сопоставления выражения с образцом представлена в методе `MatchSinglePattern` и представляет собой рекурсивную процедуру обработки выражений. Поскольку образец, используемый при объявлении макросов должен быть списком, отдельно сопоставляются первый элемент и все оставшиеся. Первый элемент, в свою очередь, может быть либо списком, либо атомом.

Обработка атомарных объектов является базой рекурсии и была реализована в методе `MatchSinglePattern`. Здесь рассматривается несколько случаев:

- с позиционным параметром сопоставляется любое выражение, причём это выражение добавляется в аккумулирующий список аргументов шаблона;

- с символом подстановки (нижнее подчёркивание) сопоставляется любое выражение, которое в дальнейшем игнорируется;

- ключевые слова сопоставляются только сами с собой;

- все остальные случаи представляют собой константное атомарное выражение (например, числа, символы или строки) и могут быть сопоставлены только с равными им выражениями в смысле предиката `EQUAL?` (во внутреннем представлении используется метод `IsEqual` класса `SReference`);

- во всех остальных случаях считается, что выражение не может быть сопоставлено с образцом.

При обработке вложенных списков проверяется структура сопоставляемого выражения. Если первые элементы как образца, так и выражения, являются списками, происходит рекурсивный вызов метода `MatchSinglePattern`. При успешном сопоставлении вложенного образца продолжается обработка хвостов списков с использованием рекурсивного вызова `MatchSinglePattern`.

Метод возвращает `true` при успешном сопоставлении и `false` в противном случае. Кроме того, один из параметров метода передаётся по ссылке и используется для формирования аргументов шаблона.

При успешном сопоставлении из метода `MatchArguments` возвращаются список аргументов шаблона и сам шаблон, соответствующий сопоставленному образцу. В противном случае список синтаксических правил обрабатывается дальше. Если ни одно из правил не может сработать, генерируется исключение `IntelibX_scheme_no_matching_rule`, добавленное специально для этого случая.

Следующий этап реализован с использованием метода `ReplaceTokens`, который применяется к аргументам шаблона для установки объемлющего контекста. Это происходит после сопоставления с образцом для того, чтобы ключевые слова макроса не были заменены на позиционные параметры, если для какого-то из символов присутствует связь в контексте вызова.

Напомним, что в объекте класса `SchExpressionCompiledSyntaxRules` сохраняются обработанные шаблоны, т. е. те, в которых присутствуют позиционные параметры вместо символов-идентификаторов из соответствующего образца. Результатом раскрытия шаблона должно стать выражение, в котором вместо позиционных параметров подставлены невычисленные аргументы, полученные при сопоставлении с образцом. Такое преобразование аналогично вычислению частичной блокировки за тем исключением, что в случае шаблона синтаксических правил аргументы не должны вычисляться.

С точки зрения реализации отличие состоит лишь в команде континуации, используемой для обработки аргументов (при вычислении частичной блокировки используется команда `just_evaluate`, а в данном случае — команда `quote_parameter`). Поэтому для раскрытия шаблона создаётся объект класса `SExpressionCompiledTemplate`, а в конструктор передаётся флаг, устанавливающий режим обработки аргументов. Реализация этого класса и подробное рассмотрение процесса вычисления были представлены в § 4.2 и § 4.4.

Далее необходимо раскрыть шаблон и вернуть полученное выражение. Для раскрытия шаблона предварительно сохраняется метка стека действий, затем вызывается метод `Evaluate` класса `SExpressionCompiledTemplate`, который поместит в стек команды, необходимые для подготовки аргументов и

раскрытия шаблона. Затем вызывается метод континуации `Step` до тех пор, пока текущая метка стека действий не станет равна сохранённой. Условие остановки означает, что были выполнены все действия для макрорасширения, а на вершине стека результатов находится раскрытый шаблон. Это выражение извлекается из стека с помощью метода континуации `Get` и возвращается из метода `ReplaceTokens` класса `SchExpressionCompiledSyntaxRules`.

Заметим, что описанный выше процесс имеет сходство с реализацией макрорасширения для макросов языка Lisp. Отличием является то, что в случае макроса к результату расширения необходимо применить метод `ReplaceTokens` для установления контекста вычисления. Для синтаксических правил контекст вычисления учитывается только для аргументов вызова.

Вычисление расширенного выражения для случая макровызова на верхнем уровне реализовано в методе `Call`. Для вызова метода `ReplaceTokens` создаётся объект класса `SchemeTokenReplace`, представляющий пустой объемлющий контекст. Как было описано выше, метод возвращает результат макрорасширения, который используется для добавления в стек действий команды `just_evaluate`, при выполнении которой произойдёт вычисление расширенного выражения.

Рассмотрим пример определения функции с использованием макроса `COND=>`, реализация которого была приведена выше:

```
(DEFINE (GUESS ANSWER X)
    (COND=>
        ((= X ANSWER) => (DISPLAY "Yes!"))
        ((< X ANSWER) => (DISPLAY "Less than needed"))
        (ELSE         => (DISPLAY "More than needed"))
    )
    (NEWLINE)
    X
)
```

При определении функции происходит создание лексического замыкания, поэтому первый шаг раскрытия макроса `COND=>` произойдёт на этом этапе. Сработает первое из синтаксических правил, которое имеет следующее внутреннее представление:

```
(((#1 => . #2) . #3)
    (IF #1
        (BEGIN . #2)
        (COND=> . #3)
    )
)
```

После сопоставления с образцом и установки контекста вычисления будет получен следующий вектор аргументов:

```
(
    (= #2 #1)
    ((DISPLAY "Yes!"))
    (
        ((< #2 #1) => (DISPLAY "Less than needed"))
        (ELSE      => (DISPLAY "More than needed"))
    )
)
```

После раскрытия шаблона будет получено следующее выражение:

```
(IF (= #2 #1)
    (BEGIN (DISPLAY "Yes!"))
    (COND=>
        ((< #2 #1) => (DISPLAY "Less than needed"))
        (ELSE      => (DISPLAY "More than needed"))
    )
)
```

Здесь позиционные параметры соответствуют аргументам функции, в то время как в шаблоне кода они относятся к выражениям, которые будут подставлены в шаблон. Эти параметры ссылаются на самый верхний стековый фрейм, но не конфликтуют между собой, поскольку при вычислении на вершине стека фреймов будут находиться разные элементы.

Таким образом, предложенная реализация с использованием имеющегося механизма лексического связывания гарантирует соблюдение основных условий гигиеничности макросов.

5.4 Примеры использования синтаксических правил

В качестве иллюстрации использования синтаксических правил для объявления макросов приведём примеры реализации тех же спецформ, которые иллюстрируют применение описанных ранее средств метапрограммирования.

Макросы `AND` и `OR` отличаются от предыдущих реализаций, поскольку могут быть вызваны без аргументов. Оба макроса являются рекурсивными. Базой рекурсии является пустой список параметров. Вместо явной проверки этого условия используется первое правило преобразования кода.

Реализация макроса `SWAP` не превосходит по лаконичности реализацию с использованием частичной блокировки, но позволяет избежать разнообразных символов квотирования, что делает приведённый код чуть более удобным для чтения.

```
(DEFINE-SYNTAX AND
    (SYNTAX-RULES ()
        ((_) #T)
        ((_ X) X)
        ((_ X . XS) (IF X (AND . XS) #F))
    )
)

(DEFINE-SYNTAX OR
    (SYNTAX-RULES ()
        ((_) #F)
        ((_ X) X)
        ((_ X . XS)
            (LET ((TMP X))
                (IF TMP TMP (OR . XS))
            )
        )
    )
)

(DEFINE-SYNTAX SWAP
    (SYNTAX-RULES ()
        ((_ X Y)
            (LET ((TMP X))
                (SET! X Y)
                (SET! Y TMP)
            )
        )
    )
)
```

6 Заключение

В рамках работы были получены следующие основные результаты:

1. в диалект InteLib Lisp была добавлена возможность объявления пользовательских макросов.

2. в рамках диалекта InteLib Scheme реализована поддержка метапрограммирования на основе синтаксических правил.

3. для обоих диалектов был реализован механизм частичной блокировки вычислений.

В качестве дальнейших перспектив работы можно указать расширение реализованных механизмов метапрограммирования добавлением таких возможностей как присоединяющая отмена частичной блокировки вычислений (форма `UNQUOTE-SPLICING`) и символ многоточия в образцах синтаксических правил для более эффективной работы со списками.

Литература

[1] Е. В. Галкина. Об одной реализации лексических замыканий языка Лисп // Сборник статей молодых учёных факультета ВМК МГУ. — 2016. № 13. — С. 17–29.

[2] Галкина Е. В. Реализация метапрограммного слоя для объектно-ориентированной модели языка Лисп // Научный взгляд в будущее. — Т. 1, вып. 7. — Одесса, 2017. — С. 21–27.

[3] И. Г. Головин, А. В. Столяров. Объектно-ориентированный подход к мультипарадигмальному программированию // Вестник МГУ, сер. 15 (ВМиК). — 2002. № 1. — С. 46–50.

[4] Официальный сайт проекта InteLib [Электронный ресурс]. — Электрон. дан. — URL: http://www.intelib.org. (дата обращения 03.04.2018)

[5] А. В. Столяров. Импорт вычислительной модели языка Scheme в объектно-ориентированное окружение // Сборник статей молодых учёных факультета ВМК МГУ. — 2008. № 5. — С. 119–130.

[6] B. Stroustrup. The C++ Programming Language (Special ed.). — Reading, MA, USA: Addison-Wesley Professional, 2000. — 1030 p.

[7] G. L. Steele. Common Lisp the Language, 2nd edition. — Woburn, MA, USA: Digital Press, 1990. — 1029 p.

[8] R. K. Dybvig. The Scheme Programming Language. — New York, USA: Prentice Hall, 1996. — 272 p.

[9] V. Turchin. REFAL-5 Programming Guide and Reference Manual. — Holyoke, MA, USA: New England Publishing Co., 1989. — 186 p.

[10] I. Bratko. Prolog Programming for Artificial Intelligence, 3rd edition. — New York, USA: Pearson Education, 2001. — 678 p.

[11] Jonathan Bartlett. The art of metaprogramming, Part 1: Introduction to metaprogramming [Электронный ресурс]. — Электрон. дан. — URL: https://www.ibm.com/developerworks/library/l-metaprog1/index.html. (дата обращения 03.04.2018)

[12] Brian W. Kernighan, Dennis M. Ritchie. The C Programming Language (2^{nd} ed.). — Upper Saddle River, New Jersey: Prentice Hall PTR, 1988. — 272 p.

[13] Tim Sheard and Simon Peyton Jones. Template Meta-Programming for Haskell // ACM SIGPLAN Notices. — 2002. 37. № 12. — P. 60–75.

[14] The Glasgow Haskell Compiler [Электронный ресурс]. — Электрон. дан. — URL: https://www.haskell.org/ghc/ (дата обращения 03.04.2018)

[15] Simon Peyton Jones, Andrew Tolmach and Tony Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC // Haskell Workshop. — 2001.

[16] Douglas McIlroy. Macro Instruction Extensions of Compiler Languages // Communications of the ACM. — 1960. № 3(4). — P. 214–220.

[17] J. McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine // Communications of the ACM. — 1960. № 3(4). — P. 184–195.

[18] Cf. J. W. Backus. The syntax and semantics of the proposed international algebraic language of the Zuerich ACM-GRAMM conference // Proceedings of the International Conference on Information Processing. Preprint. — New York, USA: International Business Machines Corp., 1959. — 24 p.

[19] Andrey V. Stolyarov. A framework of heterogenous dynamic data structures for object-oriented environment: the S-expression model // Knowledge-Based Software Engineering. Proceedings of the 6^{th} JCKBSE, vol. 108 of Frontiers in Artificial Intelligence and Applications (V. Stefanuk, Kenji Kaijiri, eds.). — Protvino, Russia: IOS Press, 2004. — P. 75–82.

[20] Э. Хювёнен, И. Сеппянен. Мир Лиспа. В 2-х т. — М.: Мир, 1990.

[21] R. Kelsey, W. Clinger, J. Rees. (eds.). Revised^5 Report on the Algorithmic Language Scheme // ACM SIGPLAN Notices. — 1998. 33. № 9. — P. 26–76.

[22] И. Е. Бронштейн, А. В. Столяров. Библиотечная поддержка вычислительной модели языка Рефал // Сборник статей молодых учёных факультета ВМК МГУ. — 2009. № 6. — С. 36–46.

[23] Jeff Alger. C++ for real programmers. — Orlando, FL, USA: Academic Press, 1998. — 388 p.

[24] D. Ferguson, D. Dwight. Call with Current Continuation Patterns // Pattern Languages of Programs Conference Proceedings. — Monticello, IL, USA, 2001.

[25] Филд А., Харрисон П. Функциональное программирование. — М.: Мир, 1993. — 640 с.

[26] Simon Peyton Jones, ed. Haskell 98 Language and Libraries: The Revised Report. — Cambridge University Press, 2003. — 263 p.

[27] Chas Emerick, Brian Carper, Christophe Grand. Clojure Programming: Practical Lisp for the Java World. — Sebastopol, CA, USA: O'Reilly Media, 2012. — 630 p.

[28] E. E. Kohlbecker Jr., D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion // Proceedings of the 1986 ACM Conference on Lisp and Functional Programming. — 1986. — P. 151–161.