Мета интерпретатор диалекта LISP

Георгий Миргородский, Никита Чикин (Санкт-Петербург, Россия)

В работе описывается диалект Lisp, заимствующий черты таких языков как Common Lisp, Lisp Machine Lisp и Maclisp, интерпретатор которого записан в CPS (Continuation-passing Style) с двумя "ветками": континуация-результат и континуация-ошибка. Предложена функциональная репрезентация окружения меток, имеющая некоторый преимущества над традиционной репрезентацией окружений ассоциативными списками.

Основной целью нашей работы является исследование и экспериментальное подтверждение расширения возможностей языка-субстрата в мета интерпретаторе без серьёзных изменений синтаксиса, а также попытка объединить в одном диалекте особенности нескольких диалектов Lisp.

Определения

Мета интерпретатор — это интерпретатор языка, близкородственного к языку-субстрату (язык, на котором написан сам интерпретатор). Синтаксический интерпретатор, наоборот, сильно отличается от языка-субстрата. Метациклический интерпретатор — это интерпретатор, в котором интерпретируемый язык и язык-субстрат не отличаются. Поскольку создание мета интерпретатора проще, чем создание синтаксического, то для создания экспериментального диалекта лучше использовать именно мета интерпретатор. Мы используем в качестве языка субстрата Common Lisp[1], так как он, на сегодняшний день, является стандартом. Одним из помешавших нам недостатков Common Lisp являтся то, что он не оптимизирует хвостовую рекурсию из-за сохранения в пямять места возвращения функции. Этот недостаток не позволяет в полной мере реализовать возможности нашего диалекта.

Традиционно, начиная с первых реализаций языка Lisp[2], окружения представлялись в виде ассоциативных списков. В нашей работе мы хотим показать возможность представления окружения catch-меток как функции

```
catch-env: symbol, val, error-cont -> |,
```

что позволяет расширить возможности языка-субстрата. В частности, это позволяет реализовать функцию catch-all, которой нет в Common Lisp, но присутствующую в диалекте Lisp Machine Lisp[3].

Особенностями разработанного диалекта являются:

- Разработанный диалект принадлежит классу Lisp 1, то есть окружение переменных и окружение процедур унифицированы. Класс Lisp 2 — языки, в которых окружения переменных и процедур различны.
- Созданы окружения динамических меток `catch-env`, лексических меток `block-env`, окружение лексических меток '`go-env`' и лексическое окружение '`env`'
- Основные процедуры написаны в стиле передачи продолжений (англ. Continuation-passing style, CPS), при котором передача управления осуществляется через механизм продолжений, что позволяет реализовать конструкции управления о которых будет написано далее.
- В отличие от языка-субстрата интерпретатор нашего языка вычисляет голову любой формы-списка. Благодаря чему можно делать такие конструкции как ((if x car quote) '(a b)), то есть в качестве результата у функции может быть другая функция или специальный оператор.
- Система обработки ошибок состоит в вызове континуации ошибки с помощью двух специальных операторах: '`error`' и '`errset`'. Они заимствована из диалекта Maclisp[4]. Этой континуацией можно управлять при помощи оператора '`errset`', который указывает границу, за которую ошибка не может распространяться.

Функциями в данной реализации являются объектами с стандартным порядком выполнения, слева направо. Специальными называются операторы с собственным порядком выполнения, отличающимся от стандартного.

Грамматика

Ниже представлена грамматика специальных форм в форме Бэкуса-Наура и описание специальных операторов, использующихся в этом диалекте.

(quote <form>) Блокирует вычисление своего аргумента.

(car <form>) Возвращает голову своего аргумента.

(cdr <form>) Возвращает хвост своего аргумента.

(cons <form> <form>) Возвращает cons-ячейку.

(null <form>) Возвращает `T`, если аргумент `nil`, иначе `nil`.

(print <form>) Печает аргумент в окно вывода.

(list {<form>}) Возвращает список вычисленных аргументов.

(and {<form>}) Если хотя бы один результат вычисления аргументов равен `nil`, то специальный оператор возвращает `nil`, иначе `T`.

(or {<form>}) Если хотя бы один результат вычисления аргументов равен `T`, то специальный оператор возвращает `T`, иначе `nil`.

(if <form> <form> <form>) Сначала вычисляется значение первого параметра. Если результат отличен от `nil`, то вычисляется второй параметр и специальный оператор возвращает результат вычисления. Если значение первого параметра — `nil`, то вычисляется значение третьего параметра и результат вычисления возвращается как результат специального оператора.

(cond (<form> <form>) {(<form> <form>)}) Вычисление аргументов происходят следующим образом: вычисляется первая форма первого аргумента; если получается результат, отличный от `nil`, то в качестве результата выдается значение второй формы первого аргумента, если первая форма первого аргумента результатом даёт `nil`, то вычисляется первая форма второго аргумента и её результат сравнивается с `nil` и так далее.

(progn <form> {<form>}) Вычисляет все аргументы, в качестве результата возвращает результат вычисления последнего аргумента.

(prog1 <form> {<form>}) Вычисляет все аргументы, в качестве результата возвращает результат вычисления первого аргумента.

(prog2 <form> <form> {<form>}) Вычисляет все аргументы, в качестве результата возвращает результат вычисления второго аргумента.

(when <form> <form> {<form>}) Вычисляет первый аргумент, если его результат отличен от nil, то вычисляет все аргументы, в качестве результата возвращает результат вычисления последнего аргумента. Если результат первого аргумента — nil, то специальный оператор возвращает nil.

(block <sym> <form>) Вместе с return-from используется для написания кода,который совершает выход из секции кода. Добавляет символ в лексическое окружение меток и вычисляет форму в этом окружении.

(return−from <sym> <form>) Служит для выхода из блока с тем же символом. Работает следующим образом: находит символ в лексическом окружении меток, выходит из блока и возвращает вычисленную форму.

(tagbody <sym> <form> {<sym> <form>}) Задаёт контекст, в котором определены имена, используемые go. Формы выполняются по порядку, символы (теги) игнорируются, результатом выполнения является nil. Выполняется таким образом: если проходит через символ, то создаёт его континуацию и добавляет символ в окружение лексических меток, если форму — выполняет её.

(go <sym>) Служит для перехода на любой тег и выполнения следующих за ним форм. Вычисляется следующим образом: ищет символ в окружении лексических меток и вызывает его континуацию.

(catch <form> <form>) Вместе с оператором throw являются динамическими аналогами block и return from. Catch устанавливает метку и при нахождении этой метки в первой форме оператора throw вычисление не идёт дальше, а в качестве результата идёт вторая вычисленная форма throw. При ненахождении метки результатом является последняя вычисленная форма.

(throw <form> <form>) Используется в паре с catch. Первый аргумент — заданная ранее метка, второй — вычисляется и выводиться в качестве результата для соответствующего catch.

(catch−all <form>) При нахождении любого throw с меткой действует аналогично catch.

(let ({(<sym> <form>)}) <form>) Служит для управления лексическим окружением. Вычисляется следующим образом: в начало уже существующего лексического окружения заносятся символы из аргумента, формы которых вычисляются. Далее последовательно вычисляется остальная форма. Вычисления происходят в обновлённом лексическом окружении. Результат вычисления последней формы возвращается как результат специального оператора.

(let$*$ ({(<sym> <form>)}) <form>) Так же как и предыдущий специальный оператор 'Let', служит для управления лексическим окружением. Отличается от неё тем, что при вычислении значения очередного символа из первой формы доступны значения уже ранее вычисленных форм.

((lambda (<sym> {<sym>}) <form>) <arg> {<arg>}) Вычисляется в безымянную фукцию. Форма может быть любым допустимым выражением, имеющим значение. Вычисление такого выражения выполняется так: сначала каждому из символов присваивается значение соответствующего ему аргумента. Далее вычисляется форма. Результат вычисления возвращается, как результат вычисления безымянной функции. И, наконец, уничтожаются формальные параметры.

(labels ((<sym> ({<sym>}) <form>) {(<sym> ({<sym>}) <form>)}) <form>) Служит для определения функции, к которому будет доступ только внутри области видимости labels. Нужен для задания рекурсивных функций, которые невозможно реализовать через lambda, так как замыканиия захватывают окружения до их создания.

(error <form>) Позволяет передать в континуацию нужное сообщение об ошибке.

(errset <form>) Передаёт сообщение об ошибке через стандартную континуацию, если оператор, в котором вызван `errset` имеет синтаксическую или арности (количества аргументов).

Авторы выражают благодарность Попову Роману Андреевичу за внимание к этой работе и полезные обсуждения.

Список литературы

[1] Peter Seibel “Practical Common Lisp”, January 2005.

[2] John Allen “Anatomy of Lisp”, January 1978.

[3] Daniel Weinreb David Moon Lisp Machine Manual, 3rd Edition, March 1981

[4] http://www.maclisp.info/pitmanual/error.html

[5] John McCarthy "Recursive Functions of Symbolic Expressions and Their Computation by Machine", 1960.

[6] John McCarthy, Paul W. Abrahams “LISP 1.5 Programmer’s Manual The Computation Center and Research Laboratory of Electronics Massachusetts Institute of Technology”, 1985.