Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование операционных систем
Системное программирование

Асеева Серафима Олеговна

# Создание предметно-ориентированного языка для разработки серверного ПО

Курсовая работа

Научный руководитель:
д. ф.-м. н., профессор Терехов А.Н.

Санкт-Петербург
2020

# Оглавление

# Введение

Системы хранения данных (СХД) — это программно-аппаратные комплексы, обеспечивающие хранение большого объема информации, а также запись данных, их чтение, копирование, создание снимков. СХД Symmetrix компании Dell отличается тем, что большая часть операций чтения, записи и копирования управляется командами с мейнфрейма с операционной системой z/OS [2] или Linux [3], использующий СХД. Поэтому обмен данными между мейнфреймом и СХД очень интенсивный.

Программное обеспечение, предоставляемое компанией Dell EMC для разных СХД, реализовано на низкоуровневых языках программирования. В связи с этим оно сильно платформозависимо, а его разработка оень дорогая. Помимо этого, присутствуют основные недостатки императивного подхода по сравнению с декларативным: сильная зависимость частей программы друг от друга, трудности в отладке, отслеживании изменения значений переменных и обнаружении "мертвого кода". Один и тот же код может дублироваться, посылка системных вызовов никак не оптимизирована — одни и те же запросы посылаются несколько раз, хотя это не всегда необходимо (например, иногда можно хранить информацию о состоянии дисков в кэше, а не запрашивать ее у СХД каждый раз). Это приводит к значительному снижению производительности и становится проблемой в условиях жестко ограниченных RTO (recovery time objective — время восстановления системы после катастрофы) и RPO (recovery point objective — к какой точке во времени необходимо восстановиться).

Отсутствует единый стандарт разработки управляющего программного обеспечения, его крайне сложно обновлять и поддерживать.

Данная курсовая является частью проекта по оптимизации работы с СХД в компании Dell EMC. В рамках этого проекта было придумано два способа избежать некоторых из указанных выше недостатков:

1. Разработка специальной библиотеки на высокоуровневом языке программирования, ориентированной на декларативное описание

правил бизнес-логики управляющего ПО.

2. Создание предметно-ориентированного языка программирования, на котором можно будет декларативно описывать формат команд, транслирующихся в последовательность системных вызовов.

В данной работе был реализован второй способ, в то время как первый способ реализуется в рамках другой курсовой работы.

Прежде всего, бизнес-логику управляющего ПО удобнее описывать на декларативном языке, так как большая часть работы заключается в проверке правил, при которых допустимо выполнение команды. Таким образом, разработчик лишь будет описывать эти правила, тогда как транслятор сможет подбирать подходящие системные вызовы и вызывать их в нужном порядке.

К тому же, высокий уровень и использование терминологии предметной области сделает программу более простой для чтения и нахождения ошибок, а разработка станет дешевле.

В работе планируется проверить, насколько жизнеспособен данный подход и есть ли необходимость в дальнейшей доработке с целью внедрения в индустрию.

# 1. Постановка задачи

В ходе работы планируется:

1. Разработать декларативный язык программирования, который будет транслироваться в запросы, распознаваемые и обрабатываемые системой хранения данных.
2. Протестировать работу транслятора с помощью эмулятора СХД.
3. Провести тестирование среди потенциальных пользователей, чтобы выявить возможные проблемы и определить пути для дальнейшего развития.

# 2. Существующие решения

1. В качестве примеров существующего серверного ПО для сообщения с СХД можно привести:

   (a) Mainframe enablers (Dell EMC) — существующие продукты, в большинстве из которых, как указано выше, используется язык ассемблера.

   (b) HPE Host Explorer — программное обеспечение, написанное на языке Python, однако вышеперечисленные недостатки императивного подхода все так же присутствуют [5].

2. Декларативные предметно-ориентированные языки программирования, с помощью которых удалось решить проблемы, аналогичные поставленным:

   (a) Язык SQL — позволяет описывать запросы к базе данных и правила, по которым ее можно изменять, при этом проверка правил и оптимизация выполнения запросов осуществляется интерпретатором.

   (b) OCL — язык описания ограничений, накладываемых на модели, описываемые на языке UML. Помимо этого, так же, как и SQL, он позволяет посылать запросы к модели [1].

   (c) Языки описания грамматик (например, YACC, ANTLR, Bison) — языки, позволяющие описывать грамматики языков программирования в декларативном стиле. По описанию грамматики автоматически строится процедура разбора входного текста на одном из существующих языков программирования (YACC и Bison — на C/C++, ANTLR поддерживает большое количество языков) [4].

# 3. Описание реализации

СХД способна обрабатывать команды в специальном формате и выдавать ответы - так же в специальном формате.

На вход лексического анализатора принимается текст программы, состоящий из:

1. команды со списком условий, которые нужно проверить, и действий, которые нужно выполнить.

2. описание форматов системных вызовов, проверяющих эти условия.

3. описание кода, исполняемого при невыполнении условий.

Лексический анализатор разбивает текст программы на токены и передает результат парсеру. Парсер строит необходимые таблицы идентификаторов и синтаксические деревья для каждой команды.

Когда пользователь вводит команду в командную строку, интерпретатор обходит дерево, составляет последовательность команд, которые необходимо послать на СХД, анализирует ответы от нее и сообщает пользователю о результатах.

UML-диаграмма последовательности изображена на Рис. 1

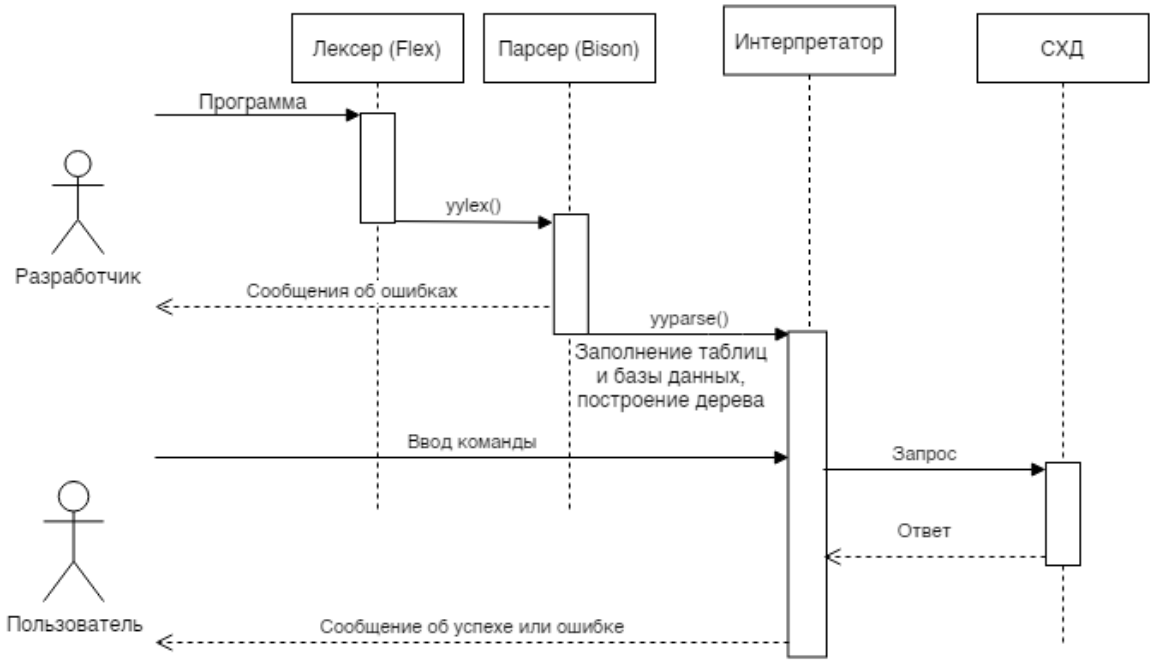


Рис. 1: Диаграмма последовательности

## 3.1. Синтаксис

Описание системного вызова представляет собой описание структуры запроса. Каждое описание, а также некоторые из полей этой структуры, помечается тегами, которые, по идее, должны отображать функцию, выполняемую системным вызовом или группой вызовов.

Теги системных вызовов и их полей при трансляции добавляются в базу данных. Эти теги можно использовать при описании команды. Когда в описании команды встречается тег, транслятор ищет этот тег в базе данных и находит все системные вызовы или поля, имеющие этот тег, модифицирует помеченные поля и формирует очередь запросов из помеченных системных вызовов.

В представленном ниже листинге описан синтаксис языка в форме Бэкуса-Наура.

⟨*syscall description*⟩ ::= 'syscall' '' ⟨*fields list*⟩ '' [⟨*tag*⟩]

⟨*fields list*⟩ ::= ⟨*none*⟩ | ⟨*fields list*⟩ ⟨*field*⟩

⟨*field*⟩ ::= ⟨*identifier*⟩ '['⟨*size*⟩']' ['=' ⟨*hexadecimal number*⟩] ';' [⟨*tag*⟩]
⟨*tag*⟩ ::= #⟨*identifier*⟩
⟨*size*⟩ ::= ⟨*decimal number*⟩

⟨*rules list*⟩ ::= ⟨*none*⟩ | ⟨*rules list*⟩ ⟨*rule*⟩

⟨*rule*⟩ ::= 'DO' ⟨*tag*⟩ [':' ⟨*field tag assignments*⟩] [',' ⟨*errors list*⟩] ';' | 'PRINT' message
⟨*errors list*⟩ ::= ⟨*errors list*⟩ ',' 'ERR' '=' ⟨*return code*⟩',' ⟨*identifier*⟩
⟨*return code*⟩ ::= ⟨*hexadecimal number*⟩
⟨*field tag assignments*⟩ ::= ⟨*field tag assignments*⟩ ',' ⟨*assignment*⟩
⟨*assignment*⟩ ::= ⟨*tag*⟩ '=' ⟨*hexadecimal number*⟩
⟨*message*⟩ ::= " ⟨*string of symbols*⟩ "

⟨*command description*⟩ ::= 'command' ⟨*identifier*⟩ '{'⟨*rules list*⟩'}'

⟨*error description*⟩ ::= 'error' ⟨*identifier*⟩ '{' ⟨*rules list*⟩ '}'

⟨*comment*⟩ ::= '//' ⟨*string of symbols*⟩

⟨*letter*⟩ ::= [a-z,A-Z]
⟨*digit*⟩ ::= [0-9]
⟨*a-f letter*⟩ ::= [a-f, A-F]
⟨*identifier*⟩ ::= '_' | ⟨*letter*⟩ | ⟨*identifier*⟩'_' | ⟨*identifier*⟩⟨*letter*⟩ | ⟨*identifier*⟩⟨*digit*⟩
⟨*decimal number*⟩ ::= ⟨*digit*⟩ | ⟨*decimal number*⟩⟨*digit*⟩
⟨*hexadecimal number*⟩ ::= '0x'⟨*a-f letter*⟩ | '0x'⟨*digit*⟩ | ⟨*hexadecimal number*⟩ ⟨*a-f letter*⟩ | ⟨*hexadecimal number*⟩⟨*digit*⟩
⟨*string of symbols*⟩ ::= 'n'| [-128 - 127]⟨*string of symbols*⟩

## 3.2. Семантика

Команды транслируются в промежуточный код на языке C++ в соответствии с таблицей трансляции (Таблица 1). Полученная программа составляет очередь из строк — запросов к СХД и посылает их с помощью функции send. На рисунке (Рис. 2) схематически изображено выполнение очередного правила команды.

| **Код на новом языке** | **Промежуточный код на C++** |
|---|---|
| `syscall <identifier> {`<br>` <field1> [size1] = xxxx`<br><br>` #tag1 #tag2 ...`<br>` ...`<br>`}` | ...//Добавление идентификатора в таблицу идентификаторов и всех его тегов в<br>базу данных тегов<br>...//Добавление каждого поля в таблицу идентификаторов и заполнение их значений (нулями по умолчанию) |
| `error <identifier> {`<br>` <rules list>`<br>`}` | ...//Добавление ошибки в таблицу идентификаторов |
| `command <identifier> {`<br>` <rules list>`<br>`}` | Интерпретация правил по порядку до символа “}” или до возникновения ошибки<br>Вывод сообщения об успешном выполнении команды |
| `DO`<br>`<tag1>:<field_tag1>=xxxx,`<br>`…, ERR=xxxx,<identifier>,`<br>`…;` | ...//Поиск системных вызовов с тегом tag1 в базе данных<br>...//Поиск полей с тегом field_tagX по базе данных и замена их значений в таблице символов<br>...//Запись всех значений полей каждого системного вызова в одну строку, добавление строки в очередь syscall_queue — очередь запросов к СХД. |

| | |
|---|---|
| | ...//Составление таблицы кодов возврата и идентификаторов ошибок<br>`string answer;`<br>`int retcode = 0;`<br>`while (!syscall_queue.empty() && retcode == 0) {`<br>`answer = send(syscall_queue.front());`<br>// Функция send посылает запрос эмулятору и возвращает его ответ<br>`syscall_queue.pop()`<br>`retcode = stoi(answer.substr(4, 4), 0, 16);` //Код возврата всегда находится в 3 и 4 байтах ответа<br>`}`<br>`if (retcode != 0) {`<br>`error_id = error_table.getid(retcode);`<br>`if (error_id == "0") {`<br>...// Вывод сообщения о ненулевом коде возврата по умолчанию и завершение исполнения программы<br>`}`<br>`else {`<br>...//Поиск идентификатора ошибки в таблице символов, переход к интерпретации ее списка правил<br>`}`<br>`}` |
| `PRINT <message>` | Печать сообщения в командной строке |

Таблица 1: Таблица трансляции

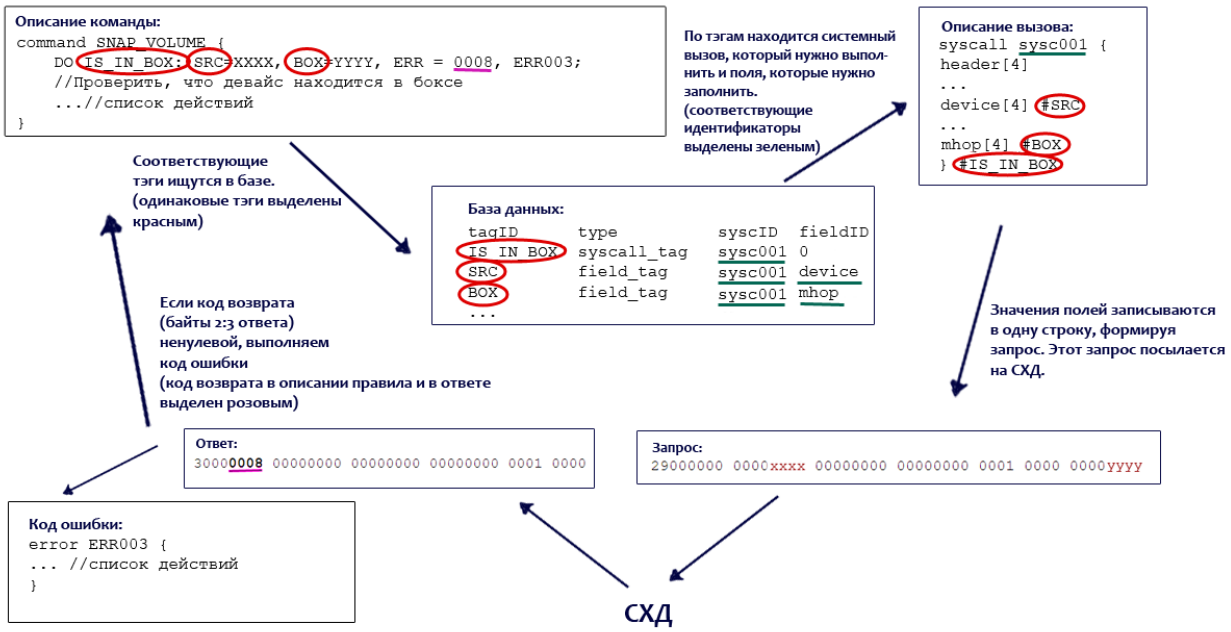


Рис. 2: Пример анализа команды

## 3.3. Инструменты

Для создания языка использовались лексический анализатор Flex и генератор синтаксических анализаторов GNU Bison. Два этих инструмента были выбраны, так как сгенерированный ими код гарантированно запускается на z/OS. Тестирование других аналогичных инструментов на z/OS не выполнялось и не предусмотрено в рамках данной работы, поэтому для создания прототипа языка было решено использовать то, что было проверено ранее.

Язык не является встраиваемым и не может быть использован как часть кода на другом языке. Встаривание частично реализовано в другой курсовой работе, в данной же работе было решено создать зык с нуля, чтобы не ограничивать возможности синтаксиса (в частности, для возможности более простой реализации поиска системных вызовов).

# Заключение

В ходе работы были выполнены следующие задачи:

1. Описаны синтаксис и семантика предметно-ориентированного языка программирования для управляющего ПО.

2. Реализован интерпретатор данного языка.

# Список литературы


[1] Cabot Jordi. Object Constraint Language (OCL) tutorial. — URL: https://modeling-languages.com/ocl-tutorial/ (online; accessed: 08.03.2020).

[2] Dell EMC. Mainframe Enablers SRDF Host Component for z/OSMainframe Enablers Version 8.4 Installation and Customization Guide. — URL: https://www.dellemc.com/en-us/collaterals/unauth/technical-guides-support-information/products/storage/docu95448.pdf (online; accessed: 28.02.2020).

[3] Dell EMC. Solutions Enabler Array Controls and Management. — URL: https://www.dellemc.com/en-us/collaterals/unauth/quick-reference-guides/2017/10/docu84170.pdf (online; accessed: 28.02.2020).

[4] GNU. Bison tutorial. — URL: https://www.gnu.org/software/bison/manual/bison.html (online; accessed: 08.03.2020).

[5] Hewlett Packard. HPE 3PAR Software Development Kit for Ruby // github. — URL: https://github.com/HewlettPackard/hpe3par_ruby_sdk (online; accessed: 27.12.2019).