ФГБОУ ВПО «Сыктывкарский государственный университет»
Институт точных наук и информационных технологий
Кафедра математического моделирования и кибернетики (ММиК)

Допустить к защите
Зав. кафедрой, профессор

_____ Н.А. Беляева
.06.2012

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

Интерпретируемый язык для вычислений с плавающей точкой IEEE 754

Специальность 010501.65 — Прикладная математика и информатика

Научный руководитель _____А.В. Ермоленко
к.ф.-м.н

Исполнитель, студент _____А.Л. Петрунёв
155 гр.

Сыктывкар 2012

АННОТАЦИЯ

В данной работе рассматриваются основные идеи, которые можно использовать для того, чтобы создать транслятор для языка программирования. В частности рассматривается реализация интерпретатора.

Содержание

Введение

Первые компьютеры как известно были не только огромных размеров, но и программировались очень долгим и трудным способом — с помощью нулей и единиц (машинный код). Тогда считалось что время компьютера более значимо чем время программиста. Со временем как размер программ стал расти, стало очевидным, что требуется более надежная (менее подверженная ошибкам) форма записи.

Для облегчения задач программирования были изобретен язык ассемблера. Первоначально ассемблерные языки представляли собой мнемоническое обозначение машинных операций. Позже функциональность ассемблера была расширена за счет появления макросов.

Программирование на ассемблере тесно связанно с архитектурой процессора. Потому как для каждой архитектуры существует свой ассемблер, то процесс написания переносимых программ остается проблемой. Данную проблему были призваны решить языки высоко уровня. Потребность в которых была четко обозначена. И тогда в середине 50-х годов появился язык *Фортран*[1].

Сегодня существует огромное количество языков высокого уровня. Их область применения также огромна, от языков общего назначения (C/C++, Java, Fortran и др.) до специализированных (SQL, AWK, Matlab и др.). Но все они опираются одни и те же базовые принципы, используя которые мы можем создать наш собственный язык.

Так же языки программирования можно разделить на *интерпретируемые* и *компилируемые*. В интерпретируемых языках исходный код программы не преобразуется в машинный код, а исполняется с помощью программы *интерпретатора*. Интерпретаторам и посвящена данная работа.

Посмотрим несколько подробнее на то что представляет собой интерпретатор. Интерпретатором обычно считается компьютерная программа выполняющая инструкции на языке программирования. Примером такой программы может служить *Octave*[2]. В некотором смысле центральный процессор также является интерпретатором, поскольку выполняет машинные инструкции.

Перед тем как выполнить инструкции интерпретатор переводит их в некоторое промежуточное представление.

---

[1] **Фортран** (Fortran) — первый язык высокого уровня, а точнее его первый диалект, который появился в середине 50-х, в последствие значительно эволюционировавший. В те годы наиболее распространенными были программы вычислительного характера. А язык Фортран в свою очередь являлся главным инструментом в научных и инженерно-технических вычислениях. На сегодняшний день существую огромное количество библиотек написанных на Фортране, находящихся в открытом доступе.

[2] **Octave** или **GNU Octave** — свободная система для математических вычислений использующая совместимый с **MATLAB** язык высокого уровня.

Рис. 1. Модель по которой работает интерпретатор

Как уже говорилось, перед тем как выполнить инструкции интерпретатор транслирует эти инструкции в некоторое эквивалентное *промежуточное представление*. Если взглянем более подробно то увидим что трансляция кода в промежуточное представление состоит из двух частей: *анализа* и *синтеза*.

В части анализа программа разбивается на составные части, на которые накладывается грамматическая структура. Затем с использованием этой структуры создается промежуточное представление исходной программы. Если в части анализа обнаруживается, что сходная программа грамматически или семантический неверна, то об этом следует информировать пользователя. Фаза анализа также собирается информацию о программе, помещая ее в специальную структуру — *таблицу символов*, которая передается дальше, в часть синтеза.

В интерпретаторе часть синтеза несколько отлична от аналогичной части в компиляторе. В отличие от компилятора, интерпретатор не создает бинарный файл. В место этого, он выполняет инструкции используя промежуточное представление и таблицу символов.

1. Схема интерпретирования

Процесс интерпретирования или по другому трансляции, делится на фазы:

- Лексический анализ.
- Синтаксический анализ.
- Семантический анализ.
- Промежуточное представление.
- Выполнение

Каждая фаза отображается одно промежуточное представление в другое. Графически это можно изобразить следующим образом:

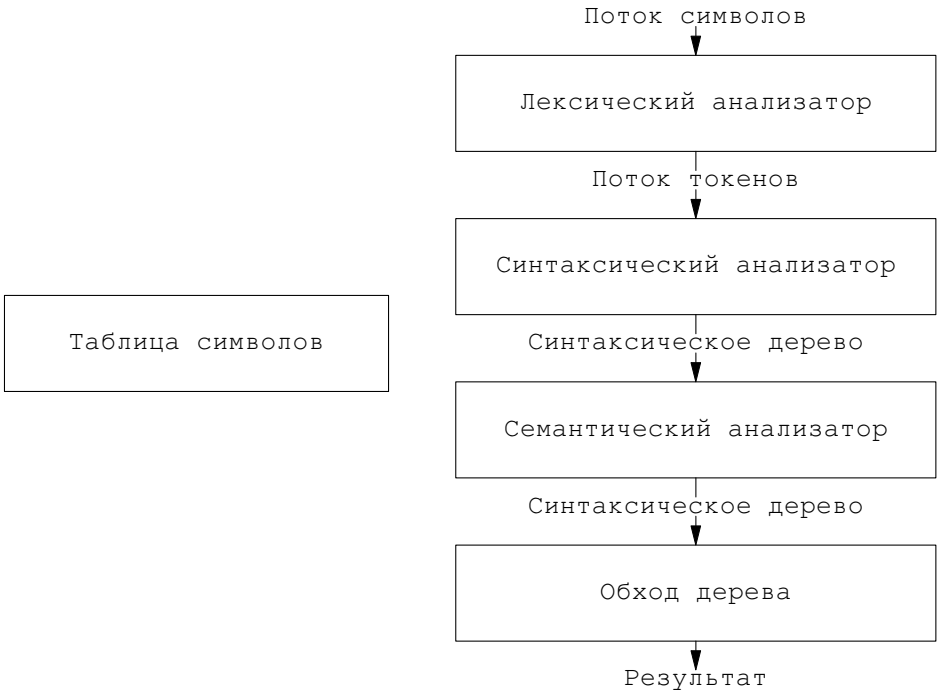

Рис. 2. Схема интерпретирования. Исходная программа переходит из одного промежуточного представления в другое

Как видно из схемы, конечным промежуточным представлением является *синтаксическое дерево*. Оно формируется в фазе синтаксического анализа и передается дальше в семантический анализ, где выполняются некоторые проверки. Далее совершая обход это дерево мы получаем результат.

2. Лексический анализ

Задача лексического анализатора заключается в формировании *лексем*[3] из потока входящих символов. Для каждой *лексемы* затем создается соответствующий *токен*[4], который имеет следующий вид:

```
<имя_токена, значение_атрибута>
```

Иными словами, лексический анализатор видит исходный код программы как последовательность токенов. Для примера рассмотрим выражение:

```
a = b + 10.0
```

Лексический анализатор видит это выражение как:

```
<id, a> <=> <id, b> <+> <10.0>
```

- Лексема `a` отобразится в токен `<id, a>`. Где `id` это абстрактный символ обозначающий `identifier`, тогда как `a` указывает на запись в таблице символов.

- Символ присваивания отображается в токен `<=>`. В данном случаю значение атрибут нам не нужно, поэтому оно опускается.

- Символ `b` отображается в токен `<id, b>`, с соответствующей записью в таблице символов.

- Символ + отображается в токен `<+>`.

- Число `10.0` отображается в токен `<10.0>`.

Пробельные символы разделяющие лексемы лексическим анализатором пропускаются. В контексте нашей программы, поток токенов выглядит как:

```
TOKEN_ID → TOKEN_ASSIGN → TOKEN_ID → TOKEN_PLUS → TOKEN_DIGIT
```

Лексический анализатор реализован как функция и вызывается из синтаксического анализатора, возвращая один токен за один раз. Таким образом если лексический анализатор встретил число `10.0` то синтаксический анализатор получит целое число `TOKEN_DIGIT`, а значение атрибута будет `10.0`. Глобальная переменная которая содержит всю информацию о токене выглядит как:

```
struct lex {
    token_t      token;
    union {
        char    *id;
        double  real;
        char    *string;
    };
```

[3] В контексте лексического анализа, **лексема** представляет собой последовательность символов.

[4] **Токен** в свою очередь это последовательность символов, то есть это та же **лексема**, только уже отнесенная к некоторой категории(Identifier, Number и т.д.) то бишь категоризованная.

```
};
```

В фазе лексического анализа значительно сокращается поток входных символов, путем группирования их в токены. Что упрощает работу следующей фазе.

2.1. Регулярные выражения[5]

Регулярные выражения помогают нам определить токены которые лексический анализатор должен распознать. Для конструирования регулярных выражений нам помогут следующие операции:

- `+` — указывает на один и более предшествующих элементов.
- `?` — указывает на ноль или один предшествующий элемент.
- `*` — указывает на ноль и более предшествующих элементов.

Теперь посмотрим на сами выражения.

- *Пробельные символы* лексическим анализатором пропускаются. То есть для них не генерируются токены. Обозначим их как:

```
ws → (blank|tabs)+
```

- *Цифры*

```
digit → [0-9]
```

- *Числа* целые и с плавающей точкой

```
digits → digit+

number → digits(.digits)?(E[+|-]?digits)?
```

- *Английские буквы* в верхнем и нижнем регистре

```
letter → [A-Za-z_]
```

- *Идентификаторы* могут начинаться с символа подчеркивания или буквы:

```
id → letter_[letter_|digit]*
```

- *Операции*

```
assign_op → =

arith_op → +|-|*|/|%
```

[5] **Регулярные выражения** дают удобный способ определения строк в тексте: символы, слова, и т.д.

```
rel_op → <|>|<=|>=|==|!=

not_op → !

logic_op  →|| | &&
```

- *Основные символы*

```
basic_sym → { | ( | [ | } | ) | ] | . | , | ; | &
```

- *Ключевые слова*

```
if → if

else → else

for → for
```

2.2. Ключевые слова

Каждый язык программирования имеет набор ключевых слов. Ключевое слово, является таким же идентификатором `id`. Когда лексический анализатор находит идентификатор, он просматривает таблицу, если данный `id` является ключевым словом, то лексический анализатор возвращает для него токен, в противном случае возвращается `TOKEN_ID` с именем идентификатора в качестве значения атрибута.

Таким образом каждое ключевое слово имеет свой токен:

```
for → TOKEN_FOR

if → TOKEN_IF

else → TOKEN_ELSE

while → TOKEN_WHILE
```

2.3. Таблица ключевых слов

Как уже говорилось когда лексический анализатор находит идентификатор, ему нужно узнать является ли он ключевым словом или нет. Для хранения данных таблица ключевых слов использует хэш(hash). Перед тем как начать работу таблица инициализируется ключевыми словами.

Для представления ключевого слова используется простая структура. Где `name` — имя ключевого слова, а `token_t` тип токена.

```
struct keyword {
    char    *name;
    token_t  type;
```

```
};
```

Следующая часть кода из лексического анализатора показывает как происходит обработка ключевых слов:

```
keyword = keyword_table_lookup(id);

if (keyword != NULL) {
    lex.token = keyword->token;
    return lex.token;
}

lex.token = TOKEN_ID;
lex.id    = ustrdup(id);
return TOKEN_ID;
```

Код был упрощен, чтобы показать суть. `id` здесь это проверяемая строка. Функция делающая поиск по таблице, является "обверткой" над функцией поиска для хэш-таблицы. Если мы находим совпадение в таблице ключевых слов, то возвращается тип токена из поля `type`. В противном случае мы возвращаем `TOKEN_ID` , а значение атрибута помещаем найденную строку.

3. Синтаксический анализ

Синтаксический анализатор или *парсер* знает обо всех конструкциях языка. Как уже говорилось ранее парсер получает токены от лексического анализатора. Его задача построить однозначное дерево. То есть после работы парсера код программы отображается в древовидную форму.

3.1. Контекстно свободные грамматики

Работа парсера построена вокруг правил. Данные правила описывают конструкции языка. То есть является его формальным описанием. Для записи эти правил мы используем *контекстно свободные грамматики*.

Грамматики состоят из следующих компонентов:

- терминалов (или токенов);
- нетерминалов (или синтаксических переменных);
- продукций (правила разбора);
- один из нетерминалов должен быть *стартовым символом*;

Терминалами называются основные символы языка. Нетерминалы состоят из терминалов, которые организуются некоторым образом, что бы описать синтаксическую конструкцию. Продукции имеют следующую структуру: нетерминал с левой стороны продукции, называемый *левой частью*, и *телом проудкции* с правой, представляющим собой набор из нетерминалов. Обе части разделяются знаком →, в правой части мы можем использовать символ |*(or)*, чтобы описать альтернативный вариант.

Для того чтобы немного прояснить ситуацию, рассмотрим случай. Предположим, что нам нужна грамматика для описания выражения `9 + 5 −3`. Правила описывающие данную грамматику выглядят так:

```
list  → list + digit | list – digit
digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Каждая из этих грамматик является продукцией. С слева стороны находится левая часть или заголовок, а справа тело продукции. `list` и `digit` являются нетерминалами. Где `list` является стартовым символом, потому что его продукция дана первой. Согласно нашей договоренности символы `+ − 0 1 2 3 4 5 6 7 8 9` являются терминалами.

Далее наша задача с помощью имеющейся грамматики "породить" приведенный выше пример. Грамматика порождает строку, начиная со стартового символа, повторно заменяя нетерминалы телами продукций для этих нетерминалов. То есть строка терминалов может быть порождена начиная со стартового символа формирующего язык, который определен с помощью грамматик. Собственно это и входит в задачу парсера, узнать возможно ли породить строку терминалов из имеющихся правил грамматики. Если это не возможно, то следует оповестить пользователя об ошибке.

3.2. Ассоциативность операций

Следующий момент о котором стоит упомянуть является ассоциативность операций. Ассоциативность возникает тогда, когда мы имеем операции одинакового приоритета и нам нужно решить в каком порядке вычислять выражение. Согласно правилам выражение `9 + 5 + 3` эквивалентно `(9 + 5) + 3`, аналогично `9 − 5 − 3` эквивалентно `(9 − 5) − 3`. Когда операнд подобно `5` имеет операторы с двух сторон, необходимо некоторое соглашение что бы решить какой оператор относится к данному операнду. Для человека данный момент очевиден. Но его следует сделать понятным также и для машины. Мы говорим что оператор + лево-ассоциативный, потому что операнд имеющий по обе стороны знак плюс, принадлежит оператору который находится слева.

Другим примером служат право-ассоциативные операторы. Например операция присваивания. То есть выражение `a = b = c` эквивалентно `a = (b = c)`.

3.3. Приоритет операций

Рассмотрим выражение `9 + 5 * 2`. Существует две возможные интерпретации этого выражения: `(9 + 5)*2` или `9 + (5 * 2)`. Очевидно что необходимы некоторые правила, которые бы разрешали не однозначность.

Мы говорим что операция `*` имеет больший приоритет чем +, если она получает свои аргументы раньше. Таким образом деление и умножение имею более высокий приоритет, чем вычитание и сложение. То есть операнд `5` относится к `*`.

Посмотрим на грамматику для арифметических выражений. На данный момент мы имеем `4` арифметических операции, каждая из которых является лево-ассоциативной. Операции `*` и `/` имеют приоритет выше чем `+` и `−`. Мы также

используем скобки, что бы показать, что приоритет выражения в скобках выше.

```
expr → expr + term | expr – term | term
term → term * factor | term / factor | factor
factor → digit | ( expr ) | E
digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

С помощью символа `E` мы обозначаем пустую продукцию.

3.4. Парсер рекурсивного спуска

Данный вид парсера обрабатывает грамматику в один проход слева на право. Парсер получает токены от лексического анализатора, сканируя файл токен за токеном. Синтаксический анализатор сверяет текущий токен, с его правилами грамматики. Если есть совпадения, то он двигается дальше, проверяя приходящие токены на соответствие грамматике. Это делается до тех пор пока не будет достигнут конец, либо найдена ошибка. Данный вид парсера также называется предикативным. Это название он получил в силу того, что следуя выбранному правилу грамматики, он предполагает что следующий приходящий токен, должен быть такой же как и в выбранном правиле. То есть если текущий токен не входит в выбранное правило, это определенно ошибка пользователя.

Рассмотрим простой пример, что бы прояснить работы предсказывающего парсера:

```
stmt → for_expr | if_expr | ... | expr
for_expr → for (optexpt; optexpr ; optexpr) stmt
optexpr → expr | E
expr → описывает присваивание, арифметические и условные выражения
```

Процесс синтаксического анализа происходит следующим образом:

```
for (i = 0; i < 10 ; i = i + 1) k = k + 1
```

Как уже говорилось ранее. Парсер получает токены от лексического анализатора. Работа парсера начинается со стартового символа грамматики. В данном примере, первый токен который придет от лексического анализатора будет `TOKEN_FOR`, обозначающий ключевое слово `for`. Теперь следуя правилу для конструкции `for`, парсер предполагает что следующий токен должен быть `'('` `(TOKEN_LPARNT)`. Если следует совпадение, то продолжаем следовать правилу. Теперь он будет проверять правило для `optexpr`, в данном случае идет присваивание, которая разрешена грамматикой. Дальше мы смотрим на наличие токена `';'` `(TOKEN_SEMICOLON)` и т.д. В конце, после обработки всего выражения, построится узел дерева для цикла `for`.

Следует также упомянуть, что то что мы описали называется синтаксический управляемая трансляцией. Если взглянуть на исходный файл парсера, но можно заметить что функции которые обрабатывают выражение для for имеют точно такие же имена, что записанные в грамматике, так же как и порядок их вызова.

Функция `match()` сравнивает предполагаемый токен с текущим. Следующий фрагмент псевдокода проясняет суть:

```
match('for'); match('(');
optexpr(); match(';');
optexpr(); match(';');
optexpr(); match(')');
stmt();
```

3.5. Устранение левой рекурсии

Для парсера рекурсивного существует возможность зацикливания. Проблемы возникают с так называемой *левой рекурсией*

Взглянем на одну из продукций описанных выше:

```
mult_expr → mult_expr * term | term
```

Проблема заключается в том, что мы имеем в теле продукции терминал с таким именем что и в правой части. Как уже говорилось мы используем синтаксически управляемую трансляцию, то есть элементы продукций отображаются в код, как имена функций. Для нас это означает что первой функцией которую парсер должен вызвать из функции `mult_expr()`, является `mult_expr()`. Таким образом парсер попадает в бесконечный цикл.

Чтобы решить данную проблему нам нужно немного переписать продукции данного вида. Рассмотри нетерминал A с двумя продукциями:

```
A → Aa | B
```

Где a и B являются последовательностью терминалов и нетерминалов, которые не начинаются с A. Продукции данного вида называются леворекурсивными. Для устранения рекурсии перепишем данную продукцию следующим образом:

```
A → BR
R → aR | E
```

Где нетерминал `R` и его продукция является *право-рекурсивными*. Теперь посмотрим на наш пример:

```
A <—> mult_expr
a <—> *term
B <—> term
```

Используя данное правило мы можем переписать грамматику следующим образом:

```
mult_expr → term rest_mult
```

```
rest_mult → * term rest_mult | E
```

Где `A` это `mult_expr`, `a` это `*term`, `B` является `term`, `R` соответствует `rest_expr`.

Теперь перепишем правила приведенные выше для операций `+, -, *, /` . Напомним как выглядели первоначальные грамматики:

```
summ_expr → summ_expr + mult_expr
          | summ_expr - mult_expr
          | mult_expr
mult_expr → mult_expr * term
          | mult_expr / term
          | term
term → digit | (summ_expr) | E
digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Преобразованная грамматика:

```
summ_expr → mult_expr rest_summ
rest_summ → + mult_expr rest_summ
          | - mult_expr rest_summ | E
mult_expr → term rest_mult
rest_mult → * term rest_mult
          | / term rest_mult | E
term → digit | (summ_expr)
digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

В преобразованной форме грамматика уже не является очевидной. Но без этого преобразования мы бы не смогли написать парсер рекурсивного спуска.

3.6. Используемые правила грамматики

```
programme → stmts
stmts → stmt stmts_rest
stmts_rest → stmt stmts_rest | E
stmt → scope_expr
     | if_expr
     | for_expr
     | while_expr
     | break_expr
     | continue_expr
     | return_expr
     | declaration_expr
     | other_expr
     | unknown_expr
     declaration_expr → process_function
```

```
      | process_matrix
      | process_variable
or_expr → and_expr rest_or
rest_or → '||' rest_or
and_expr → rel_expr rest_and
rest_and → '&&' rest_and
expr → id '=' expr | or_expr
summ_expr → mult_expr rest_summ
rest_summ → '+' mult_expr rest_summ
          | '-' mult_expr rest_summ | E
mult_expr → term_expr rest_mult
rest_mult → '*' term_expr rest_mult
          | '/' term_expr rest_mult | E
term_expr → term | '(' expr ')'
term → integer | real | id
```

3.7. Аннотированное синтаксическое дерево

Как мы знаем, задача парсера состоит в построении дерева. Главная функция `programme()` возвращает корень дерева, листья которого представляют терминалы, а внутренние узлы нетерминалы.

Так же нам известно что дерево должно быть однозначным, то есть каждое выражение может иметь только единственное представление в виде дерева.

В качестве примера посмотрим как строится узел для цикла `while`:

```
struct ast_node_while*
ast_node_while(struct ast_node *expr,
               struct ast_node *stmt)
{
    struct ast_node_while *while_node;

    return_val_if_fail(expr != NULL, NULL);

    while_node = ast_node_new(sizeof(*while_node));

    while_node->expr = expr;
    while_node->stmt = stmt;

    AST_NODE(while_node)->type   = NODE_TYPE_WHILE;
    AST_NODE(while_node)->child  = expr;
    AST_NODE(while_node)->destructor = ast_node_while_free;

    expr->parent = AST_NODE(while_node);
    stmt->parent = AST_NODE(while_node);

    return while_node;
}
```

Функция проверяет открывающую скобку, условие и закрывающую скобку. Затем анализируется тело цикла, которое не может быт пустым. Функция `op_expr`, в правилах грамматики служит стартовым символом для грамматики описывающей выражения использующие логический и арифметические операции. Аналогично и для остальных узлов.

4. Таблица символов

В контексте нашей программы таблица символов очень проста. И служит лишь для хранения переменных. Так как интерпретатор это гибкий инструмент, как таковых типов у нас нет. Любая переменная может менять свое значение на ходу.

Каждая программа имеет по крайне мере одну таблицу символов.

```
c = 0

function my_function(a, b) {
    local a1, a2

    a1 = a + 1
    a2 = a + b + c

    return a1 + a2
}
```

Грубо говоря для каждой области видимости переменных, создается своя таблица. То есть переменные из старых таблиц перекрываются переменными из новой. Переменная с находится в глобальной таблице. Локальные переменные `a1, a2` находятся в новой таблице символов.

Таблица символов создается парсером. Все таблицы держатся в односвязном списке, где последний элементом является текущая таблица. А в самом начале находится глобальная таблица.

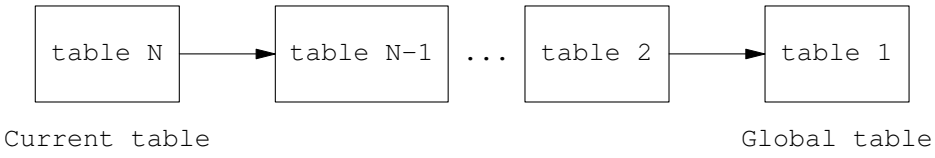

Рис. 3. Для каждого нового пространства создается своя таблица символов

Следующая часть кода показывает обработку области видимости:

```
static struct ast_node*
process_scope(void *opaque)
{
    struct ast_node *stmt;
    struct ast_node_stub *stub_node;
    struct scope_ctx *helper;
```

```
        stmt = stmts(opaque);

        return stmt;
    }
```

Код был значительно упрощен. Когда парсер находит `'{'` он вызывает `scope_scope()` чтобы обработать область видимости. В данном случае мы решили не создавать новую таблицу символов для каждого нового пространства. Таким образ здесь только собираются инструкции которые находятся между `'{'` и `'}'`. Все переменные будут помещены в глобальную таблицу символов.

Основная структура хранящая информацию о переменной:

```
    struct symbol {
            value_t                 v_type;
            char                    *name;
            union {
                    double          digit;
                    char            *string;
                    gsl_vector      *vector;
                    gsl_matrix      *matrix;
            };

            release_t               destructor;
    };
```

4.1. Поиск символов

Таблица символов использует хэш в качестве бэкенда. Поэтому когда мы ищем символ в таблице символов, мы используем хэш функции.

Функции используемый для поиска символов:

```
struct symbol* symbol_table_lookup_top(char *name);
struct symbol* symbol_table_lookup_all(char *name);
```

Первая ищет символ только в текущей таблице. А вторая просматривает все.

5. Таблица функций

В нашей программе функции хранятся в отдельной таблице, вместе с областью видимости своих локальных переменных. Функции можно разделить на пользовательские и библиотечные. Оба типа функций содержатся в одной таблице. При запуске программы таблица функций инициализируется библиотечными функциями. У библиотечных функций отсутствует тело функции.

Структура содержащая информацию о функции:

```
struct function {
```

```
        char                    *name;
        unsigned int            is_lib;
        unsigned int            nargs;
        struct symbol           **args;
        struct symbol_table     *scope;
        struct ast_node         *body;
        lib_handler_type_t      handler;
};
```

API[6] для работы с функциями и символами похожи. Таблица функций также использует хэш. Когда пользователь вызывает функцию мы ищет ее в таблице. Если ее там нет, то сообщаем об ошибке. Для того что бы пояснить как происходит обработка функций, нам придется забежать немного вперед. Посмотрим как обрабатывается узел представляющий функцию при обходе дерева.

```
static void
traverse_func_call(struct ast_node *node)
{
    struct function *function;
    struct ast_node_func_call *func_node;

    return_if_fail(node != NULL);

    func_node = (struct ast_node_func_call *)node;

    function = function_table_lookup(func_node->name);

    if (function->is_lib) {

        perform_lib_function(function, func_node->args);

    } else {
        symbol_table_set_scope(function->scope);

        perform_custom_function(function,func_node->args);

        symbol_table_pop();
    }
}
```

Сначала мы достаем функцию из таблицы. Затем смотрим является ли она библиотечной или пользовательской. Как уже говорилось пользовательские функции содержат в себе таблицу символов для локальных переменных. Поэтому мы сперва устанавливаем эту таблицу символов, чтобы наши локальные переменные

[6] **API** (application programming interface) — набор готовых классов, процедур, функций, структур и констант, предоставляемых приложением (библиотекой, сервисом) для использования во внешних программных продуктах.

стали видны. Нам также необходимо инициализировать аргументы функции. Поле обработки функции мы выталкиваем таблицу для локальных переменных. Теперь текущей таблицей символов остается глобальная таблица.

```
static void
perform_lib_function(struct function *func,
                     struct ast_node **args)
{
    struct eval *eval;
    struct symbol *sym;
    value_t v_type;
    void *result;
    int ok;

    perform_init_args(func, args);

    func->handler(func, &v_type, &result);

    eval = eval_new(TAG_CONST, v_type, result);

    push(eval);
}
```

Данная функция показывает обработку библиотечных функций. Функция была немного упрощена. Сперва инициализируем аргументы, затем вызываем обработчик. Дальше все как обычно, полученное значение помещается в стэк. Так выглядит `handler` для фнкции `sin` в немного упрощенном варианте:

```
int
libcall_sin(struct function *func,
            value_t *v_type, void **result)
{
    double *dg;

    dg  = umalloc(sizeof(double));
    *dg = sin(func->args[0]->digit);

    *v_type = VALUE_TYPE_DIGIT;
    *result = dg;

    return TRUE;
}
```

Внутри обработчика мы вызываем функцию `sin` из стандартной библиотек языка Си.

6. Обход дерева

Мы подошли к последней фазе. Здесь мы имеет синтаксическое дерево, полученное от парсера. Которое является эквивалентом исходной программы. Теперь имея данное представление программы в виде дерева, мы можем выполнять инструкции.

Обход дерева совершаемый нами с целью получения результат называется обход в глубину. То есть мы начиная с корня с пускаемся в дочерние узлы пока это возможно. Дойдя до листьев дерева, мы выполняем необходимы вычисления и поднимаемся обратно.

Пример функции для обхода дерева:

```
static void
traverse_op(struct ast_node *node)
{
    struct ast_node_op *op;
    struct eval *a,*b, *c;
    struct symbol *sym;

    op = (struct ast_node_op *)node;

    traversal(op->left);
    traversal(op->right);

    b = pop();
    a = pop();

    c = eval_add_op(a, b, op->opcode);

    push(c);
}
```

Для каждого узла дерева существует своя функция обхода. Данная функция вызывается когда мы попадаем в узел соответствующий арифметическим или логическим операциям. Сперва мы посещаем дочерние узлы(в данном случае в порядке слева на право, но в целом не важно). Вернувшись обратно, выталкиваем значение из "стека" и производим вычисления. Далее результат вычислений помещается опять в "стек". В итоге, то что осталось в стеке печатается как результат.

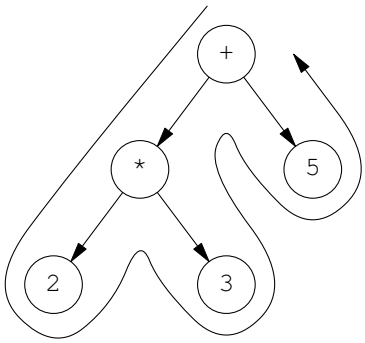

Рис. 4. Схема обхода дерева

Мы чуть не забыли упомянуть о *семантическом анализе*. Задача семантического анализа состоит в проверке синтаксического дерева. Основной задачей является проверка типов переменных. Поскольку у нас интерпретатор, то мы можем не проверять типы переменных как это делается например для языка C. Поэтому обход дерева и семантический анализ объединяются в одни проход.

7. Библиотека GSL

Для работы с матрицами и векторами мы используем библиотеку *GSL*. GSL написана на языке C, для численных вычислений в прикладной математики и науке. То есть представляет собой широкий набор функций, которые разбиты на библиотеки. В частности мы используем из всего этого множества библиотеку *BLAS*[7]. BLAS дает высокоуровневый интерфейс для операций над матрицами и векторами.

Существуют 3 уровня операций:

Level 1
Операции над векторами `y = \alpha x + y`

Level 2
Операции матрица-вектор , т.е `y = \alpha A x + \beta y`

Level 3
Операции над векторами `C = \alpha A B + C`

В нашей программе мы используем данную библиотеку следующим образом:

```
gsl_vector*
libm_vector_matrix_mult_op(gsl_vector *a,
                           gsl_matrix *b,
                           opcode_type_t op)
{
    gsl_permutation *pm;
    gsl_matrix *inv, *c, *mx;
    gsl_vector *vc;
    int err, ok, signum;

    switch(op) {
```

[7] **BLAS** — Basic Linear Algebra Subprograms

```
        case OPCODE_MULT:
            vc  = gsl_vector_alloc(b->size1);
            gsl_blas_dgemv(CblasTrans, 1.0, b, a, 0.0, vc);
            break;
        case OPCODE_DIV:
            pm = gsl_permutation_alloc(b->size1);
            vc = gsl_vector_alloc(b->size1);

            matrix_init(&mx, b);

            gsl_linalg_LU_decomp(mx, pm, &signum);
            gsl_linalg_LU_solve(mx, pm, a, vc);

            gsl_permutation_free(pm);
            gsl_matrix_free(mx);

            break;
        default:
            error(1, "nonconformant operation");
        }

        return vc;
    }
```

Функция была упрощена. Данная функция принимает в качестве параметров матрицу и вектор и производит вычисления.

8. IEEE[8] 754

Это технический стандарт для вычислений с плавающей точкой принятый в 1985 году Институтом инженеров по электронике и электротехнике. Кратко опишем что данный стандарт определяет:

- *Арифметические форматы*: представления двоичном и десятичной форме данных с плавающей точкой, то есть конечные числа(включая также значащие нули, денормализованные числа), бесконечности и NaN'ы.
- *Правила округления*: свойства которые должны быть удовлетворены во время вычислений и преобразований.
- *Операции*: арифметические и другие.
- *Обработка исключений*: деление на ноль, переполнение и др.
- Методы, которые используются для преобразования числа в процессе математических операций.

[8] **IEEE** — Institute of Electrical and Electronics Engineers.

Представление числа с плавающей точкой:

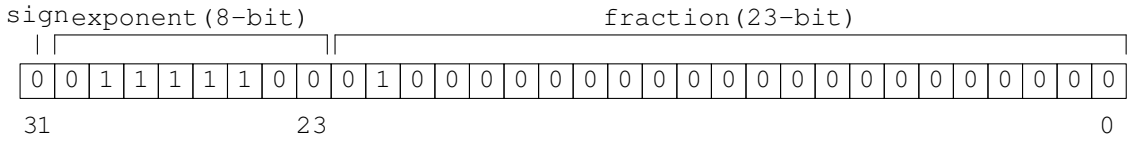

Рис. 5. Представление числа 0.15625 в виде числа с плавающей точкой IEEE 754

Следует также отметить что сопроцессор Intel 8087 выпущенный компанией Intel в 1980 году стал основой для стандарта IEEE 754. Данный сопроцессор обеспечивал два основных формата для чисел с плавающей точкой — 32 и 64 битные. А также формат расширенной точности — 80 бит.

9. Описание языка

Напомним что наш язык работает с числами с плавающей точкой. Есть некоторые сходства с языком C. И так посмотрим что есть.

КОНСТАНТЫ

Внутри числа представлены в виде чисел с плавающей точкой имеющие двойную точность. То есть целые числа которые пользователь может ввести скажем число `256` отобразится в `256.000000`. Константами могут быть также строка, матрица и вектор. Строки начинаются и заканчиваются двойным апострофом ‘`"`’. Пример вектора `[1,4,2]`. Матрица как и вектор начинается с открывающей скобки, затем через запятую перечисляются элементы. Единственно отличие это размерность, матрицы имеют две размерности. Пример матрицы `[1,2;8,6]`.

ПЕРЕМЕННЫЕ

Переменные начинаются с буквы или символа подчеркивания, с последующим произвольным числом символов подчеркивания, цифр и букв. Есть также специальная переменная `ans` которая содержит результат последнего вычисления. Изначально `ans` содержит путь к рабочей директории.

КОММЕНТАРИИ

Комментарии начинаются с символа `#`. Символы после `#` пропускаются вплоть до конца строки. Символ конца строки не является частью комментария.

ВЫРАЖЕНИЯ

Самым простым выражением в нашем языке является константа. В остальном имеются сходства с другими языками. Далее `expr` ссылается на полное выражение, а `var` на переменную. В частности переменная это всего лишь `id` — идентификатор. Для того что бы обратится к элементам вектора достаточно ввести `id[idx]`, для матрицы `id[idx1][idx2]`.

`expr + expr`
: Результатом выражения является сумма двух выражений.

`expr - expr`
: Результатом выражения является разность двух выражений.

`expr * expr`
: Результатом выражения является произведение двух выражений.

`expr / expr`
: Результатом выражения является частным двух выражений.

`expr ^ expr`
: Результатом выражения является первое выражение возведенное во второе. Значение второго должно быть целым. Если первое выражения является матрицей, то она должны быт квадратной.

`var = expr`
: Переменной присваивается выражение.

`expr1 < expr2`
: Результатом является `1` если `expr1` строго меньше чем `expr2`.

`expr1 <= expr2`
: Результатом является `1` если `expr1` меньше либо равно `expr2`.

`expr1 > expr2`
: Результатом является `1` если `expr1` строго больше `expr2`.

`expr1 >= expr2`
: Результатом является `1` если `expr1` больше либо равно`expr2`.

`expr1 == expr2`
: Результатом является `1` если `expr1` равно `expr2`.

`expr1 != expr2`
: Результатом является `1` если `expr1` не равно `expr2`.

`expr && expr`
: Результатом является `1` если оба выражения являются `1`.

`expr || expr`
: Результатом является `1` если одно из выражения не нулевое.

Старшинство операций от имеющих наименьший приоритет к наибольшему:

`||` — лево-ассоциативная

`&&` — лево-ассоциативная

Операции отношения — лево-ассоциативные

Оператор присваивания — право-ассоциативный

`+` и `-` — лево-ассоциативные

`*` и `/` — лево-ассоциативные

`^` — право-ассоциативные

ИНСТРУКЦИИ

Инструкции упорядочивают вычисление выражений. Инструкции запускаются когда встречается символ перевода строки, который обозначает конец инструкции.

`expression`

Инструкции данного типа ведут себя следующим образом. Если инструкции начинается с `<var><=> ...` то она рассматривается как инструкция присваивания. В противном случае выражение вычисляется и результат печатается на выход.

`if ( expression ) statement1 else statement2`

Эта инструкция сначала вычисляет `expression`, затем в зависимости от результата вычисляет либо `statement1` либо `statement2` (если присутствует). То есть если выражение в скобках дает не нулевой результат то выполняется `statement1`, в противном случае `statement2`.

`while ( expression ) statement`

Инструкция `while` будет запускать `statement` пока значение `expression` не нулевое. Цикл прекращается если `expression` обращается в ноль, либо выполняется инструкция `break`.

`for ( optexpr1 ; optexpr2 ; optexpr3 ) statement`

Цикл `for` контролирует запуск `statement` следующим образом. Сперва вычисляется `optexpr1`. Затем вычисляется `optexpr2` если его значение не нулевое вычисляется `statement` В противном случае цикл не выполняется. После запуска `statement` вычисляется `optexpr3`. Если `optexr1` или `optexpr3` пропущены то на их месте не производится никаких вычислений. Если пропущено `optexpr2` то это эквивалентно замене `optexpr2` на `1`. Следующий код является эквивалентом цикла `for`:

```
optexpr1;
while (optexpr2) {
    statement;
    optexpr3;
}
```

`break`
: Инструкция прерывающая цикл.

`continue`
: Начинает новую итерацию в цикле.

`return`
: Инструкция выхода из функции. Для возвращения значени из функции используется `return expr`.

ФУНКЦИИ

Функции нужны для определения вычислений которые используются позже. Функции могут возвращать или не возвращать значения. Определяются функции следующим образом:

```
function fname(args) { \newline
    local vars

    statement_list
}
```

Аргументы функции перечисляются через запятую. То есть `arg1, arg2,...,argN`. Список локальных переменных является необязательным. Объявляются они как `local var1, var2,...,varN`. Тело функции представляет собой список инструкций. Есть также так называемые библиотечные функции: `sin(), cos(), ln(), exp()`.

ПРИМЕРЫ

Решение дифференциальных уравнений методом Рунге-Кутты:

```
# RK4

t0 = 0.0
tn = 0.2
h  = 0.005
y0 = 1.0

function f(t,y) {
    return exp(t)
}

function deriv(t) {
```

```
    return exp(t)
}

len = (tn - t0) / h

y = vector(len)
dy = vector(len)
err = vector(len)

i = 0
dy[i] = y0
y[i] = deriv(t0)
err[i] = dy[i] - y[i]

i = i + 1

while (i < len) {

        k1 = h * f(t0, y0)
        k2 = h * f(t0 + 0.5 * h, y0 + 0.5 * k1)
        k3 = h * f(t0 + 0.5 * h, y0 + 0.5 * k2)
        k4 = h * f(t0 + h, y0 + k3)

        yn = y0 + 1/6 * (k1 + 2*k2 + 2*k3 + k4)

        d = deriv(t0)

        t0 = t0 + h
        y0 = yn

        dy[i] = yn
        y[i] = d
        err[i] = dy[i] - y[i]

        i = i + 1
}

"Vector dy:"
dy

"Vector y:"
y

"Vector err:"
err
```

Заключение

В работе были использованы базовые принципы так называемой теории компиляторов, для создания интерпретируемого языка. На данный момент с помощью данного языка решена задача нахождения решения обычного дифференциального уравнения методом Рунге-Кутты. Код находится в примерах в разделе описания языка.

Литература

[1]
Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman Compilers: Principles, Techniques, and Tools: Addison Wesley, 2006

[2]
Michael L. Scott Programming language pragmatics: Morgan Kaufman Pablishers

[3]
LitePAC [Электронный ресурс]: — Режим доступа: http://www.litepac.org/ — Дата посещ. 23.06.2012

[4]
Bc [Электронный ресурс]:— Режим доступа: http://en.wikipedia.org/wiki/Bc_programming_language — Дата посещ. 23.06.2012

[5]
Tiny C Compiler [Электронный ресурс]:— Режим доступа: http://bellard.org/tcc/ — Дата посещ. 23.06.2012