УДК 004.4’422 DOI: 10.17587/it.26.169-176

С. М. Салибекян, канд. техн. наук, e-mail: ssalibekyan@hse.ru,
Национальный исследовательский университет "Высшая школа экономики", Москва

Трансляция арифметико-логического выражения с использованием формата внутреннего представления на базе парадигмы dataflow

Статья посвящена разработке методики разбора, внутреннего представления и трансляции в машинный код инфиксных арифметико-логических выражений. Отличительной чертой разработки является применение нового формата внутреннего представления выражения, основанного на парадигме dataflow (вычисления с управлением потоком данных). Методика может найти применение в компиляторах и интерпретаторах языков программирования высокого уровня.

Ключевые слова: *разбор арифметико-логического выражения, инфиксная форма записи арифметического выражения, компиляция, внутреннее представление арифметического выражения, языки программирования высокого уровня, вычисления с управлением потоком данных*

Введение

Арифметико-логические выражения (АЛВ) являются частью синтаксиса всех языков высокого уровня (ЯВУ). Например, целью создания первого в истории ЯВУ Fortran как раз и была автоматизация разбора арифметических выражений: Fortran — это сокращение от англ. выражения "FORmula TRANslator" ("транслятор формул"). Наиболее удобной для человека является инфиксная форма записи АЛВ. И для перевода формул из инфиксной формы в машинный код довольно часто применяется промежуточное представление АЛВ [1], называемое внутренним представлением, из которого впоследствии генерируется машинный код.

Темой настоящей статьи является разработка нового формата промежуточного представления АЛВ, способов трансляции инфиксной записи АЛВ в это представление и генерация машинного кода. Необходимость разработки методики возникла в ходе создания компилятора объектно-атрибутного (ОА) языка (ОА язык) [2]. Язык создавался для управления параллельными вычислениями в ОА вычислительной системе (ВС), относящейся к классу dataflow (вычисления с управлением потоком данных) [3, 4], а также для описания сложно структурированных динамических информационных конструкций, обрабатываемых ею. Компилятор для ОА языка разрабатывался методом "раскрутки", когда сначала создается простейший компилятор, затем с его использованием пишется программа новой версии компилятора, расширяющего функциональность языка, и т. д. В настоящий момент уже реализована вторая версия ОА языка, где пока нет автоматического распознания АЛВ. И поэтому самой важной задачей при создании третьей версии языка стала ее реализация.

Таким образом, целью исследования является создание и формализация методики анализа АЛВ, представленного в инфиксной форме. Задачами настоящего исследования является разработка:

- формата внутреннего представления арифметических выражений, приспособленного для реализации на базе ОА графа (динамическая информационная конструкция для представления сложно структурированных данных в ОА ВС);
- методики синтеза внутреннего представления инфиксного АЛВ, содержащего бинарные арифметико-логические операции ("+", "–" и т. д.), а также функции с фиксированным и произвольным числом аргументов во внутреннее представление;
- методики выполнения АЛВ во внутреннем представлении или перевода внутреннего представления в машинный код.

Автором был проведен анализ существующих решений в данной области, который пред-

ставлен в следующей главе, однако ни одно из них не оказалось приемлемым решением поставленной задачи. Дело в том, что, во-первых, эти методики ориентированы на разбор только АЛВ с бинарными операторами и не приспособлены, например, для разбора функций с переменным числом операндов. Во-вторых, в ОА ВС применяется универсальный формат представления данных и программы в виде семантической сети специального формата (ОА граф), потому-то и понадобилась разработка новой методики анализа и внутреннего представления АЛВ на его основе. Однако разработанные в ходе исследования формат внутреннего представления АЛВ и метод распознания АЛВ универсальны и могут быть использованы в компиляторе любого другого ЯВУ.

1. Обзор существующих решений

Для выбора наиболее подходящего метода внутреннего представления был выполнен обзор существующих решений. Разбор АЛВ можно разделить на три составные части: метод распознания выражений (перевод из инфиксной формы во внутреннее представление), внутреннее представление и перевод из внутреннего представления в машинный язык. Начнем с обзора форматов внутреннего представления арифметических выражений.

Наиболее известными форматами внутреннего представления АЛВ являются: тетрады, обратная польская запись и арифметический граф (ациклический ориентированный граф) [1].

При представлении в виде четверок (тетрад) каждая АЛ операция представляется в виде общности, состоящей из четырех полей: код арифметической операции, первый и второй операнды (константа или ссылка на переменную), ссылка на переменную для результата. Поле "результат" может хранить либо указатель на переменную, либо идентификатор поля операндов той тетрады операции, куда этот результат передается в качестве операнда. С помощью такого метода можно описывать только унарные или бинарные операции. Также применяется и представление в виде триад: каждая триада имеет поля: операция, операнд 1, операнд 2. В качестве операнда может выступать константа, переменная или ссылка на другую триаду, результат выполнения операции которой является операндом.

Формат "обратная польская запись" получил свое название в честь польского ученого Лукасевича, предложившего его. В нем описание АЛВ представляет собой последовательность констант, переменных и операций над ними. Обозначение операции записывается после обозначения операндов. Например, инфиксное выражение $(3 + x)\cdot(y + 5)$ в обратной польской записи имеет вид: $3\ x + y\ 5 + *$. Недостатком такого способа является то, что возможно только представление операций с фиксированным числом операндов. Например, функцию минимума, где может быть произвольное число операндов, таким способом описать не представляется возможным.

Арифметический граф (или синтаксическое дерево) — еще один способ промежуточного представления, где АЛВ записывается в виде ориентированного графа-дерева: узлы графа ассоциируются с арифметическими операциями, а дуги идентифицируют данные (начало дуги обозначает результат операции, а конец направлен на узел, обозначающий операцию, для которой данные являются операндом). Пример программной реализации синтеза арифметического графа приведен в работе [5]. Этот формат описания имеет преимущество перед тетрадами и польской записью, так как может описывать операции с нефиксированным числом операндов. Арифметический граф можно реализовать, например, с помощью динамических структур: каждый узел представляет собой структуру (в нашем случае имеется в виду конструкция с ключевым словом struct в языке программирования Си), которая включает в себя поле кода АЛ операции и несколько ссылок на другие структуры, описывающие операцию, функцию или операнд. В качестве недостатка такого представления можно отметить сложность перевода из инфиксной записи АЛВ во внутреннее представление.

Одним из наиболее удачных алгоритмов анализа АЛВ является алгоритм Бауэра—Замельзона — его можно использовать как для перевода инфиксной формы АЛВ в обратную польскую форму, так и для непосредственного выполнения вычислений по АЛВ во время интерпретации программы. По этому алгоритму разбор АЛВ ведется с использованием двух стеков: в один из них помещаются операнды, в другой — операции. Приоритет и порядок обработки арифметических операций задается с помощью таблицы функций перехода, описывающей действие компилятора при переходе от одного символа операции к другому. Данный алгоритм был упрощен Э. Дейкстрой (метод стека с приоритетами) [6]. В нем каждой операции присваивается число, отражающее ее приоритет, и поэтому отпадает необходимость применения громоздкой таблицы обработки операций. Вышеприведенные алгоритмы при-

меняются и для перевода инфиксной записи АЛВ в обратную польскую, впрочем, их легко модифицировать для синтеза тетрад или арифметического дерева.

Еще одна интересная методика разбора АЛВ предложена в работе [7]. В ней используется вектор типов операции. Каждый тип подразумевает обозначение операции, ее арность (сколько аргументов имеется у операции) и приоритет. Синтаксис АЛВ задается с помощью контекстно-свободной грамматики, а синтаксический анализ АЛВ осуществляется с помощью LR(1) разбора [8]. После разбора АЛВ представляется в виде арифметического графа, а граф может быть использован, например, для вычисления значения, записанного с помощью АЛВ. Данный способ подходит для разбора математических операций с фиксированным числом аргументов.

Однако все алгоритмы, попавшие в обзор данной статьи, в состоянии анализировать только унарные (с одним аргументом) и бинарные (с числом аргументов, равным двум) операции, а операции с фиксированным числом операндов в состоянии анализировать только алгоритм, предложенный в работе [7]. Таким образом, все вышеперечисленные методики внутреннего представления, синтеза и перевода на машинный язык АЛВ оказались неудобными при реализации на dataflow вычислительной системе. Поэтому и потребовалась разработка собственного формата внутреннего представления арифметических выражений, а также методов синтеза АЛВ и их исполнения или трансляции на машинный язык.

2. Объектно-атрибутный граф как способ представления арифметических выражений

В настоящей работе внутреннее представление арифметических выражений строится на базе ОА подхода к организации вычислений и структур данных. ОА ВС относится к классу ВС, управляемых потоком данных, где акцент делается на описание обмена данными [9], а не последовательности операций, как в классической парадигме с управлением вычислениями посредством потока команд. Еще одной особенностью ОА подхода является работа с динамическими данными, упакованными в информационную конструкцию под названием ОА граф (или ОА сеть, ОА дерево), включающего в себя как данные, так и программный код [3]. Для описания предложенного подхода к обработке АЛВ введем два понятия: АЛ ОА граф, который будет использоваться как реализация арифметического графа, и арифметико-логическое устройство (АЛУ). АЛУ вычисляет значение АЛВ, представленного в виде ОА графа (ОА граф также может найти применение и для перевода АЛВ в машинный код).

"Кирпичиком", из которых строится ОА граф, является милликоманда (МК). Она состоит из двух полей: нагрузка (данные или указатель) и атрибут, идентифицирующий нагрузку. Формально МК — это кортеж $c = \langle a, l \rangle$, где l — множество нагрузок (константы или указатель на ячейку памяти), $a \in A$ — множество атрибутов (A — счетное множество). В частности, атрибут может хранить код операции, которую необходимо проводить над операндом, хранящимся в нагрузке МК.

В состав АЛУ входит регистр под названием "аккумулятор". Один из операндов АЛ операции обязательно должен находиться в аккумуляторе, и результат выполнения операции помещается в аккумулятор — такое решение аналогично аккумуляторной архитектуре процессора [10]. Например, операция сложения описывается с помощью трех МК: первая — запись значения из нагрузки МК в аккумулятор, вторая — сложение нагрузки со значением в аккумуляторе (результат помещается в аккумулятор) и третья — запись результата из аккумулятора в ячейку памяти. Любое АЛВ можно представить как линейную последовательность МК, исполняемых АЛУ.

Для описания последовательности МК будем применять следующую нотацию: нагрузка отделяется от атрибута с помощью знака "#", а несколько последовательно выполняемых МК разделяются знаком пробела. Атрибут МК обозначается с помощью мнемоники, например, Set — установить значение в аккумулятор АЛУ, Add — сложить, Sub — вычесть, Out — выдать результат из аккумулятора. Так, операция сложения в предложенной нотации выглядит следующим образом: Set#2 Add#2 (по первой МК АЛУ запишет 2 в аккумулятор, по второй — прибавит 2 к значению из аккумулятора и полученный результат поместит в аккумулятор). Для выдачи результата сложения можно использовать МК выдачи с мнемоникой "Out", в нагрузке которой находится указатель на ячейку памяти (переменную), куда необходимо поместить результат следующей операции:

$$\text{Set\#x Add\#y Out\#Var}, \qquad (1)$$

где x, y — имена переменных-операндов; Var — имя переменной для записи результата.

Последовательность МК может быть сколь угодно длинной, если приоритет операций будет одинаков. Последовательность МК будем называть информационной капсулой (ИК).

Однако в инфиксной форме АЛВ первым идет указание переменной для записи результата ("y = ..."). И тот факт, что МК записи результата (Out) находится в конце ИК, вызывает небольшие сложности при разборе выражения: необходимо запоминать адрес, а затем записывать его в конец ИК. Поэтому для удобства разбора АЛВ введем в состав АЛУ регистр, хранящий адрес для записи результата вычислений, и расширим функциональность АЛУ атрибутом МК "OutSet", по которой АЛУ выполнит запись адреса из нагрузки МК в этот регистр. В этом случае запись результата АЛУ будет проводиться после того, как оно выполнит все МК из ИК с описанием АЛВ. После того как АЛУ выполнит все МК из ИК, значение из аккумулятора (т. е. результат вычислений) будет записан по адресу, установленному в регистре адреса результата Var:

OutSet=Var Set#x Add#y. (2)

Для реализации записи результата сразу в несколько ячеек памяти, например, как в выражении Var1 = Var2 = x + y, необходимо заменить регистр записи адреса на буфер (вектор) адресов. Тогда ИК с АЛВ примет вид (3), в которой встречаются две МК OutSet. После выполнения последней МК из ИК результат вычислений будет записан по адресам Var1 и Var2:

OutSet#Var1 OutSet#Var2 Set#x Add#y. (3)

Следующей задачей, требующей решения, является представление арифметических выражений с различным приоритетом операций (приоритет задается как приоритетом самой АЛ операции, так и с помощью скобок при записи АЛВ). Наиболее удачным оказалось следующее решение: расположить описание операций более высокого приоритета в отдельной ИК, а ссылку на нее расположить в нагрузке МК, которая использует результат вычисления этого выражения в качестве операнда. Таким образом, описание АЛВ представляет собой ОА граф топологии "дерево". На рис. 1 представлен ОА граф, описывающий выражение: Y = D − − 10 + max(A + B, C − 5, 0). На рис. 1 прямоугольником показан атрибут МК, а овалом — его нагрузка. Номер уровня, где расположена ИК, соответствует уровню приоритета операций. Например, на рис. 1 на первом уровне находятся операции OutSet, Set, "−" и "+". У всех операций, кроме последней, операндами являются константы или переменные, а нагрузка последней МК хранит ссылку на описание АЛВ более высокого приоритета, а именно функцию Max. В нагрузке МК с атрибутом "Max", в свою очередь, находится указатель на ИК с описанием трех операндов для этой функции: первые два операнда — выражения, третий — константа. Первая МК устанавливает в аккумулятор значение, а последующие (с атрибутом "MaxCalc") записывают в аккумулятор максимальное значение аккумулятора и нагрузки.

Для каждого уровня АЛВ ОА графа ИК используется свой отдельный аккумулятор (т. е. применяется стек аккумуляторов): при переходе на более высокий уровень иерархии задействуется следующий аккумулятор из стека; при возврате на предыдущий уровень активным становится предыдущий аккумулятор из стека, а значение из аккумулятора предыдущего уровня используется в качестве операнда для операции на нижележащем уровне. Таким образом, АЛУ может вычислить значение АЛВ путем последовательного выполнения МК из ИК, входящей в состав ОА графа (наподобие левостороннего обхода графа в глубину [11]).

Для формализации выполнения АЛВ необходимо ввести индексацию МК ОА графа (каждая МК имеет свой уникальный индекс). МК дополняется полем *inext*, в которой хранится индекс следующей МК из ИК. Специальный индекс с обозначением *nil* используется в качестве индекса в поле *inext* в конце ИК. С помощью поля *inext* МК можно выстроить в цепочку, заканчивающуюся индексом *nil*, которая формирует ИК. В состав АЛУ также вводится регистр индекса текущей (выполняемой) МК. АЛУ обрабатывает МК, индекс которой находится в этом регистре, далее же в регистр записывается индекс из поля *inext*, и цикл выполнения МК повторяется.

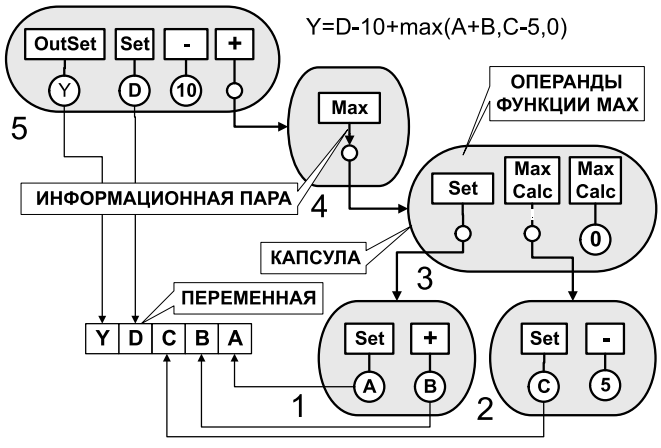

Рис. 1. Внутреннее представление арифметического выражения с многоаргументной функцией

В том случае, когда в регистр записывается *nil*, АЛУ может осуществить две реакции. Если ИК имеет наинизший уровень, то АЛУ записывает результат по адресам, хранящимся в буфере адресов для записи результата, иначе передает значение из аккумулятора текущего уровня в качестве операнда на уровень ниже.

Осуществим формализацию предложенной модели описания и выполнения АЛВ посредством математического аппарата, предложенного в работе [12]. Итак, АЛ ОА граф можно представить в виде тройки:

$$OAG = \{MKA,\ L,\ \bar{C}\},$$

где *MKA* — множество индексов операций, выполняемых АЛУ;

$$L = \{Const \cup I \cup \Omega\},$$

где *Const* — это множество любых констант (например, $Const = \{\mathbb{R} \cup \Sigma\}$, где $\mathbb{R}$ — множество вещественных чисел, Σ — множество цепочек символов ($\Sigma = A^*$, где A — алфавит символов); Ω — множество индексов (адресов) ячеек памяти (*RAM*), где хранятся данные; I — множество адресов МК. *RAM* (память переменных) формализуется как вектор, элементами которого являются константы, т. е. $RAM_k \in Const$ ($|RAM| = |\Omega|$, $\Omega = 1, 2, \ldots |\overline{RAM}|$). Пусть $\bar{C}$ — память МК:

$\bar{C} = <c>^k$ — это вектор кортежей, где каждое *c* представляет собой тройку

$$c = <mka,\ l,\ inext>, \qquad (3)$$

где $mka \in MKA$ — атрибут МК (атрибут идентифицирует операцию, которую необходимо применить к операнду, например, Set, OutSet, Add, Sub и т. д.); *inext* — индекс следующей МК из ИК ($inext \in I$, если МК последняя в ИК, то для нее $inext = nil$); l — нагрузка МК ($l \in L$).

Если нагрузкой МК является индекс МК (т. е. $Load(mk) \in I$, где функция *Load* выделяет из МК нагрузку), то это значит, что операнд для текущей операции получается путем вычисления АЛВ более высокого уровня иерархии (в нагрузке МК хранится индекс первой МК из ИК более высокого приоритета).

Начальное состояние АЛУ можно формализовать в виде четверки:

$$ALU = \{a_0,\ OAG,\ istart,\ \bar{F}\}, \qquad (4)$$

где a_0 — начальное значение аккумулятора ($a \in Const$); $istart \in I$ — индекс МК, с которого начинается обход (выполнение) АЛ ОА графа; $\bar{F}$ — вектор функций, описывающих вычислительные операции; операции F_i: $Const \boxtimes Const \to Const$. Множество индексов элементов вектора $\bar{F}$ совпадает с множеством элементов *MKA*, т. е. $|\bar{F}| = |MKA|$. Иными словами, каждому атрибуту МК соответствует своя функция вычисления результата АЛ операции.

Состояние АЛУ во время вычислительного процесса можно описать с помощью пятерки элементов:

$$S = \{\bar{A},\ Out,\ \overline{ind},\ Level,\ ALU\}, \qquad (5)$$

где *Level* — текущий уровень обрабатываемой ИК из АЛ ОА графа (в самом начале работы АЛУ $Level := 0$);

$\bar{A} = <Const>^{Level}$ — вектор значений аккумулятора (для каждого уровня приоритета АЛВ имеется свой аккумулятор);

$Out = \{\Omega\}^{Level}$ — множество адресов для записи результата вычисления;

$\overline{ind}$, ($ind_i \in I$) — вектор индексов текущей МК (МК, которая распознается); число элементов в векторе эквивалентно приоритету (уровню) разбираемой ИК АЛ ОА графа.

Тогда изменение контекста АЛУ за один такт можно формализовать как:

$$A^{k+1}_{Level} := F_{mka}(A^k_{Level},\ mkl),$$

где A^k_{Level}, A^{k+1}_{Level} — состояние аккумулятора на уровне *Level* на предыдущем и последующем тактах работы АЛУ, где $mka = atr(ind_{Level})$, $mkl = load(ind_{Level})$ — атрибут и нагрузка текущей МК (адрес текущей МК находится в верхушке стека $\overline{ind}$). По окончании выполнения ИК, если $Label \neq 0$ (т. е. распознавание проводится не на первом уровне АЛВ), выполняется запись результата в аккумулятор более низкого уровня $A^{k+1}_{Level-1} := F_{mka}(A^k_{Level},\ mkl)$, $Level := Level - 1$, иначе происходит запись результата вычислений в RAM по индексам из множества *Out* ($RAM_i = A_1$, где $i \in Out$, A_1 — аккумулятор первого уровня), и АЛУ останавливает свою работу.

Следующая МК для выполнения выбирается таким образом. Если $next(mk) \neq nil$, то $ind_{Level} := next(mk)$, где функция *next* выдает поле *inext* из МК. В том же случае, когда $Load(ind_{Level}) \in I$, $Level := Level + 1$. Когда $next(mk) = nil$, то в том случае, если $Level = 1$ (наинизший приоритет ИК), происходит запись значения аккумулятора в ячейки RAM, адреса которых входят в множество *Out*, и сброс множества *Out* (т.е. $Out = \varnothing$); если же $Level \neq 0$, то $A_{Level-1} := F_{mka}(A_{Level-1},\ A_{Level})$, т. е. полученный результат передается на уровень ниже в качестве операнда.

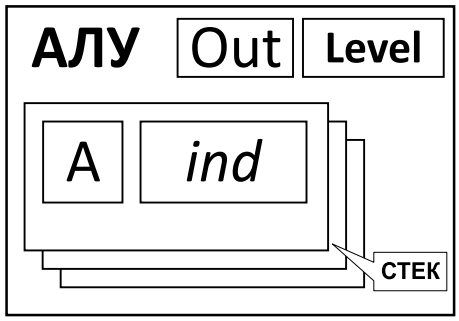

Рис. 2 Схема виртуального устройства АЛУ

Модель функционирования АЛУ можно схематично представить следующим образом. Имеются буфер адресов для записи результата *Out*, ячейка памяти для хранения текущего уровня приоритета разбираемой ИК *Level*, а также стек, который на каждом уровне хранит два элемента: аккумулятор (*A*) и индекс (адрес) анализируемой в настоящее время МК (*ind*). При переходе к анализу ИК более высокого уровня (например, уровень *i*) в стек добавляется новый элемент (т. е. A_i и ind_i), а при окончании разбора ИК элемент из верхушки стека удаляется. Схема АЛУ приведена на рис. 2.

3. Алгоритм преобразования АЛВ, представленного в инфиксной форме, во внутреннее представление и перевод внутреннего представления в машинный код

В ходе исследования выяснилось, что все приведенные в обзоре алгоритмы распознания АЛВ и синтеза внутреннего представления неудобны для реализации на базе dataflow вычислительной системы. Поэтому потребовалась разработка собственного алгоритма, удовлетворяющего заявленным требованиям. На вход алгоритма с лексического анализатора поступает поток описаний лексем, например, представленных в виде токенов [13]. На выходе — АЛ ОА граф, с описанием АЛВ.

Разработанный алгоритм распознания АЛВ достаточно прост, так как в нем применяется только один стек, который используется для хранения адресов ИК, входящих в состав синтезируемого АЛ ОА графа, и приоритета операции каждого уровня графа. Разбор арифметического выражения выполняется слева направо. Пусть S — упорядоченное множество адресов памяти МК ($s \in S$, $s \in I$), а упорядоченное множество *Priority* хранит приоритет последней анализируемой операции АЛВ на данном уровне приоритета АЛВ. Первоначально в S присутствует только один элемент — адрес пустой ИК (в нее будут помещаться МК с описанием операций первого (низшего) уровня приоритета), а уровень *Level* будет равен 1 (индексация элементов S и *Priority* начинается с 1). Пусть функция p возвращает приоритет операции, заданный атрибутом МК. Пусть t — текущая распознаваемая лексема из АЛВ $t \in \{C \cup V \cup O \cup F \cup "(" \cup ")" \cup ","\}$, где C — константы, V — переменные, O — АЛ операции, F — упорядоченное множество применяемых в анализируемой программе функций со знаком "(" в конце, например, "sin(". Пусть *mk* — это текущая синтезируемая МК из ИК, адрес которой хранится в S_{Level}. Пусть первоначально в S хранится один элемент — ссылка на *mk*, причем *atr*(*mk*) = *OutSet*, а в нагрузке находится указатель на переменную, в которую будет записываться результат вычислений (т. е. уже проведен разбор операции присвоения результата: например, "x = "), *Level* = 1, $Priority_1 = 0$ (операция присвоения имеет наинизший приоритет). Все функции из множества F проиндексированы. Далее приведем упрощенный алгоритм синтеза АЛ ОА графа, который работает при условии, что АЛВ написано синтаксически правильно, а в качестве аргументов функции выступают только переменные или константы (но не выражения) и функции, имеющие не менее двух аргументов.

1. Если $t \in \{C \cup V\}$ и $S_{Level} \neq nil$, то *load*(*mk*): = *t*.
2. Если $t \in \{C \cup V\}$ и $S_{Level} = nil$, создать новую МК *mk* = <Set, *t*> и добавить ее в ИК, по адресу из S_{Level}.
3. Если *t* = "(", то создать ИК *IC* (*IC* = *nil*), *load*(*mk*): = *adr*(*IC*), *Level*: = *Level* + 1, S_{Level}: = *adr*(*IC*).
4. Если *t* = ")", то *Level*: = *Level* − 1.
5. Если $t \in O$ и ($Priority_{Level} = p(t)$ или *atr*(*mk*) = = *Set*), то создать новую МК *mk* = <*t*, *nil*>, $Priority_{Level} = p(t)$.
6. Если $t \in O$ и $Priority_{Level} > p(t)$, то *Level*: = = *Level*-1, перейти к началу алгоритма (т. е. заново проанализировать лексему *t*).
7. Если $t \in O$ и $Priority_{Level} < p(t)$, то сформировать новую ИК *IC* = *nil*, сформировать МК *mk* = <*t*, *adr*(*IC*)>, *Level*: = *Level* + 1, S_{Level}: = *adr*(*IC*).
8. Если $t \in F$, то создать новую ИК *IC*, создать новую МК *mk* = < *t*, *adr*(*IC*)>, *Level*: = *Level* + 1, S_{Level}: = *adr*(*IC*), $Priority_{Level}$: = −*t*, добавить в *IC* МК <*Set*, *nil*>.
9. Если *t* = ", ", то считать следующую лексему *t*; создать МК *mk* = < −$Priority_{Level}$, *t*>

где *adr* — функция, возвращающая адрес ИК.

В пункте 8 алгоритма выражение $Priority_{Level}$:= := −*t* происходит сохранение индекса функции (знак минус необходимо для того, чтобы отличать приоритет операции от индекса функции). Данный прием необходим для упрощения алгоритма, чтобы не вводить новый элемент. В пунк-

те 9 $mk = < -Priority_{Level}, t>$ выполняется присвоение индекса функции атрибуту МК. Процесс синтеза АЛ ОА графа проиллюстрирован на рис. 3.

АЛ ОА граф можно использовать и для промежуточного представления АЛВ с последующим преобразованием в машинный код. Для осуществления генерации в машинный код можно использовать АЛУ-транслятор: его функцией является не вычисление АЛВ, а генерация машинного кода по нему. Реализация АЛУ-транслятора входит в планы будущих исследований. Поскольку в машинном коде большинство команд бинарные, то основной задачей АЛУ-транслятора является формирование из ОА графа последовательности бинарных (содержащих два операнда) машинных команд. Процедура трансляции представляет собой рекурсивный алгоритм обхода АЛ ОА графа в глубину. Например, МК Set при трансляции заменяется установкой первого операнда машинной инструкции (рис. 4). Следующая за ней МК с АЛ операцией транслируется в код операции машинной инструкции, а нагрузка транслируется во второй операнд. Если в ИК далее следует МК с операцией, то формируется следующая машинная команда, у которой первый операнд является результатом выполнения предыдущей команды. Если при обходе ОА графа встречается МК со ссылкой на ИК с описанием выражения более высокой иерархии, тогда промежуточный результат запоминается в ячейке памяти, а затем, после того, как алгоритм закончит обход ИК, в качестве операнда машинной инструкции добавляется ссылка на данную переменную. И в качестве адреса для записи результата в последней машинной инструкции указывается адрес ячейки для результата вычисления выражения (на рис. 4 это — переменная *Y*).

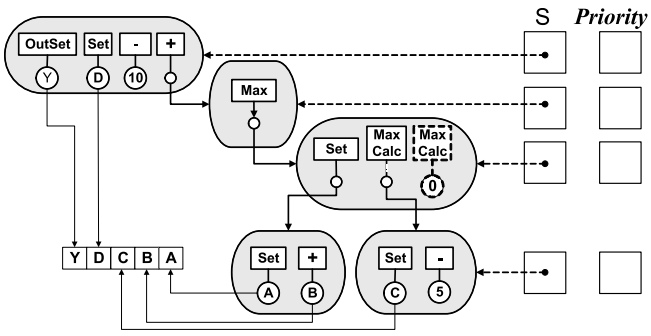

Рис. 3. Разбор АВЛ

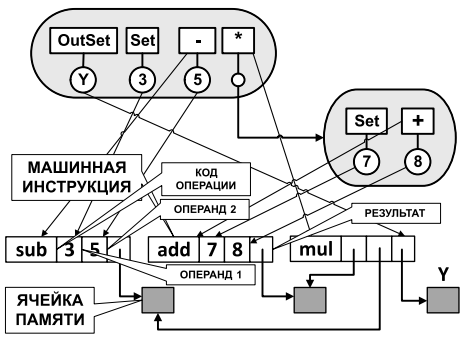

Рис. 4. Формирование машинного кода из АЛ ОА графа

Заключение

АЛУ может быть реализовано в виде виртуального устройства, входящего в состав выполняемой программы. Например, АЛУ было программно реализовано автором и в настоящий момент входит в состав среды имитационного моделирования вычислительного процесса в ОА ВС. Также функциональность компилятора ОА языка была дополнена распознанием АЛВ, представленных в инфиксной форме, и синтезом АЛ ОА графа. На рис. 5 представлена визуализация синтезированного в среде ОА имитационного моделирования АЛ ОА графа (пометка "Var" обозначает, что значение хранится в переменной). В среде программирования и моделирования также было интегрировано разработанное виртуальное устройство АЛУ, функциональность которого заключается в обработке АЛ ОА графа. Таким образом, ОА граф в ОА системе является не промежуточным, а конечным представлением АЛВ.

Разработанный метод представления АЛВ в виде арифметического ОА графа обладает по сравнению со способами, приведенными в обзоре, некоторыми преимуществами. Во-первых, у операндов АЛВ может быть сколь угодно большое (в том числе и переменное) число операндов. Во-вторых, можно задать несколько

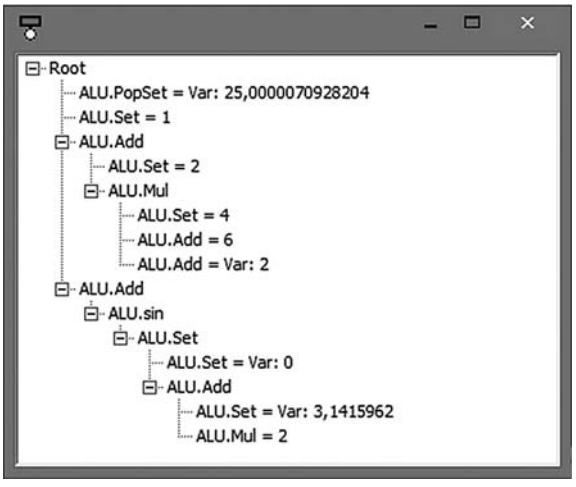

Рис. 5. Визуализация ОА графа арифметического выражения x = 1 + 2*(4 + 6 + x1) + sin(x2 + x3*2)

адресов ячеек памяти, куда будет записываться результат вычисления АЛВ. В-третьих, ОА граф удобен для трансляции в него инфиксной формы АЛВ: трансляция проводится непосредственно в промежуточную форму без использования дополнительных информационных конструкций (применяются только стек ссылок на адреса ИК для каждого яруса ОА графа и приоритет последней АЛ операции).

Следует отметить, что данный способ, в первую очередь, разрабатывался для применения в составе ОА вычислительной системы. В ней АЛ ОА граф используется не в качестве промежуточного, а в качестве конечного представления АЛВ. Однако методика может успешно применяться и для классической архитектуры ВС. Поэтому разработанная методика анализа АЛВ может найти применение в компиляторах ЯВУ на фазе семантического анализа АЛВ.

Список литературы

1. **Ахо А. В., Лам М. С., Сети Р., Ульман Д. Д.** Компиляторы: принципы, технологии и инструментарий. 2-е изд.: Пер. с англ. М.: ООО "И. Д. Вильямс", 2019. 1184 с.

2. **Салибекян С. М., Панфилов П. Б.** Объектно-атрибутная архитектура — новый подход к созданию объектных систем // Информационные технологии. 2012. № 2. С. 8—14.

3. **Data** flow computing: theory and practice / edited by John A. Sharp. Ablex Publishing Corp. Norwood, NJ, USA,1992. 569 с.

4. **Milutinovic V., Trifunovic N., Salom J., Giorgi R.** The guide to dataflow supercomputing. USA: Springer, 2015.

5. **Губаев Т. О., Петрова Н. К.** Разработка синтаксического анализатора арифметических выражений на языке С++ // Вестник Казанского государственного энергетического университета. 2018. Т. 10. № 2 (38). С. 32—40.

6. **Вторников А.** Арифметические выражения: анатомия, разбор, программирование // Системный администратор. 2013. № 10 (131). С. 68—73.

7. **Charles N. Fischer.** On Parsing and Compiling Arithmetic Expressions on Vector Computers ACM // Transactions on Programming Languages and Systems (TOPLAS). 1980. Vol. 2, N. 2. P. 203—224.

8. **Grune D. et al.** Modern Compiler Design. Second Edition. Springer, 2012. 822 p.

9. **Milutinovic V. et al.** The guide to dataflow supercomputing. USA: Springer, 2015.

10. **Корнеев В. В., Киселев А. В.** Современные микропроцессоры. СПб.: БХВ-Петербург, 2003. 448 с.

11. **Скиена С.** Алгоритмы. Руководство по разработке. 2-е изд.: Пер. с англ. СПб.: БХВ-Петербург. 2017. 720 с.

12. **Салибекян С. М., Панфилов П. Б.** Вопросы автоматно-сетевого моделирования вычислительных систем с управлением потоком данных // Информационные технологии и вычислительные системы. 2015. № 1. С. 3—9.

13. **Dick Grune et al.** Modern Compiler Design, Second Edition. Springer, 2012. 822 p.

S. M. Salibekyan, Assistant Professor, e-mail: salibek@yandex.ru,
National Research University Higher School of Economics, Moscow, Russion Federation

Translation of Arithmetic-Logical Expression Using Internal Representation Format Based on Dataflow Paradigm

The paper is devoted to the development of methods of parsing, internal representation and translation of infix arithmetic-logical expressions into machine code. A distinctive feature of the development is the use of a new format for the internal representation of the expression, based on the paradigm dataflow (calculations with data flow control). The technique can be used in high-level language compilers.

Keywords: *analysis of arithmetic-logical expression, infix form of arithmetic expression recording, compilation, internal representation of arithmetic expression, high-level languages, dataflow*

DOI: 10.17587/it.26.169-176

References

1. **Aho Al'fred V., Lam Monika S., Seti Ravi, Ul'man Dzhefri D.** Compilers: principles, technologies and tools, Moscow, I. D. Vil'yams, 2019, 1184 p.

2. **Salibekyan S. M., Panfilov P. B.** *Informacionnye Tekhnologii,* 2012, no. 2, pp. 8—14.

3. **John A. Sharp** ed. Data flow computing: theory and practice, Ablex Publishing Corp. Norwood, NJ, USA, 1992, 569 c.

4. **Milutinovic V., Trifunovic N., Salom J., Giorgi R.** The guide to dataflow supercomputing, USA, Springer; 2015.

5. **Gubaev T. O., Petrova N. K.** *Vestnik Kazanskogo Gosudarstvennogo Energeticheskogo Universiteta,* 2018, vol. 10, no. 2 (38), pp. 32—40.

6. **Vtornikov A.** *Sistemnyj Administrator*, 2013, no. 10 (131), pp. 68—73.

7. **Charles N. Fischer.** On Parsing and Compiling Arithmetic Expressions on Vector Computers, *ACM Transactions on Programming Languages and Systems (TOPLAS),* 2 (2), 1980, p. 203—224.

8. **Grune D. et al.** Modern Compiler Design. Second Edition. Springer, 2012, 822 p.

9. **Milutinovic V. et al.** The guide to dataflow supercomputing, USA, Springer, 2015.

10. **Korneev V. V., Kiselev A. V**. Modern microprocessors, SPb., BHV-Peterburg, 2003, 448 p.

11. **Skiena S.** Algorithms. Development guide, SPb., BHV-Peterburg, 2017, 720 p.

12. **Salibekyan S. M., Panfilov P. B.** *Informacionnye Tekhnologii i Vychislitel'nye Sistemy*, 2015, no. 1, pp. 3—9.

13. **Dick Grune et al.** Modern Compiler Design, Springer, 2012, 822 p.