УДК 004.043

ИССЛЕДОВАНИЕ МОДЕЛИ ГЕНЕРАТОРА ОБЪЕКТНОГО КОДА, АССЕМБЛЕРА И БЛОКА ОПТИМИЗАЦИИ ИМПЕРАТИВНОГО ЯЗЫКА НА ЯЗЫКЕ PROLOG

[1]Обломов И.А., [1]Ржавин В.В., [1]Первова В.В., [2]Герасимова А.Г.

[1]ФГБОУ ВО «Чувашский государственный университет имени И.Н. Ульянова», Чебоксары, e-mail: shuvsuoblomov@gmail.com;

[2]ФГБОУ ВО «Чувашский государственный педагогический университет им. И.Я. Яковлева», Чебоксары, e-mail: g.alina2012@yandex.ru

Основная концепция традиционных и повсеместно используемых императивных языков программирования напрямую связана с «фон-неймановской» моделью архитектуры компьютера. Соответственно этой модели все данные и программы компьютера хранятся в одной памяти. Процессор получает из памяти очередную команду, декодирует её, выбирает из памяти указанные в качестве операндов данные, выполняет команду и размещает результат снова в памяти. Предлагается модель генератора объектного кода для процедурного (императивного) языка программирования на примере бинарных арифметических выражений. В качестве инструмента реализации генератора кода выбран декларативный язык Prolog, позволяющий существенно сократить объем программного кода при обработке сложных символьных структур, таблиц и словарей, полученных в результате лексической и синтаксической обработки исходного текста программы. Данная статья является продолжением цикла работ, посвященных теории трансляции языков программирования высокого уровня. В статье был рассмотрен декларативный язык программирования Prolog, так как входные данные, полученные от лексического и синтаксических анализаторов, представляют собой поток символьной информации, представленной в виде деревьев вывода. Рекурсивная структура языка Prolog с помощью нескольких предложений позволяет обрабатывать сложные массивы древовидных данных. Например, поиск идентификаторов, программных меток, сформированных предыдущими анализаторами в двоичные справочники, осуществляется двумя-тремя предложениями языка Prolog.

Ключевые слова: генератор объектного кода, словарь, дерево разбора, выражение, ассемблер

RESEARCH OF THE MODEL OF OBJECT CODE GENERATOR, ASSEMBLE AND OPTIMIZATION BLOCK OF IMPERATIVE LANGUAGE IN PROLOG LANGUAGE

[1]Oblomov I.A., [1]Rzhavin V.V., [1]Pervova V.V., [2]Gerasimova A.G.

[1]Chuvash State University named after I.N. Ulyanov, Cheboksary, e-mail: shuvsuoblomov@gmail.com;

[2]Chuvash State Pedagogical University named after I.Ya. Yakovlev, Cheboksary, e-mail: g.alina2012@yandex.ru

The basic concept of traditional and ubiquitous imperative programming languages is directly related to the «von Neumann» model of computer architecture. According to this model, all data and programs of the computer are stored in one memory. The processor receives the next command from memory, decodes it, selects the data specified as operands from memory, executes the command and places the result back in memory. An object code generator model for a procedural (imperative) programming language is proposed on the example of binary arithmetic expressions. The declarative language Prolog was chosen as a tool for implementing the code generator. This article is a continuation of a series of works devoted to the theory of translation of high-level programming languages. The article considered the declarative programming language Prolog, since the input data received from the lexical and parsers is a stream of symbolic information presented in the form of output trees. The recursive structure of the Prolog language, with the help of several clauses, allows processing complex arrays of tree-like data. For example, the search for identifiers, software labels, generated by previous analyzers into binary directories, is carried out by two or three sentences of the Prolog language.

Keywords: object code generator, dictionary, parse tree, expression, assembler

В настоящее время современный рынок труда испытывает острую нехватку специалистов, способных решать насущные информационные проблемы наиболее эффективными и экономичными методами, в том числе с использованием декларативных или неалгоритмических языков программирования. Следовательно, научные работы, которые направлены на исследование проблем изучения декларативных языков программирования и путей их решения, остаются актуальными [1]. Наиболее актуальны на данный момент методы и инструменты программирования, которые позволяют прогнозировать будущие тенденции развития технологий [2]. Язык программирования Prolog используется при решении большого класса задач, связанных с разработкой систем искусственного интеллекта, который используется для обработки естественного языка и имеет обширные возможности, позволяющие обрабатывать информацию из баз данных [3].

Основное назначение генератора объектного кода – развертывание и обработка последовательности атомов, построенных синтаксическим анализатором в последовательность машинных команд. Многие детали генератора кода зависят от окружения, в котором он работает, а также от машины, для которой порождается объектный код.

Цель работы – выявление наиболее эффективных способов решения задач с применением генератора объектного кода средствами языка логического программирования Prolog.

Материалы и методы исследования

В данной работе применялись теоретический метод (анализ научной, учебно-методической литературы по вопросам исследования), методы обобщения, систематизации знаний, тестирования программы, анализ хода ее выполнения.

Результаты исследования и их обсуждение

Информация для человека играет немаловажную роль, поэтому основной из задач человечества является обработка данной информации и упорядочивание [4]. База знаний логического языка Prolog представляет собой предикатную модель фрагмента предметной области – совокупность правил и фактов, которыми декларируется решение задачи [5]. Мониторинг научно-исследовательских работ позволяет находить и создавать новые решения для упрощения и автоматизации процессов разработки [6].

Структура генератора кода представляется следующими блоками: блок моделирования исполнения, блок распределения памяти, элементы таблиц, процедура генерации кода, блок управления сумматором. В состав генератора входит набор процедур генерации ассемблерных или машинных команд.

Блок моделирования исполнения определяет процедуру вычисления выражений, в частности, для выражения $(A + B) + (X + Y)$, что представляет собой очевидный способ его вычисления, с точки зрения машинного языка представлен на рис.1. Каждому из трех сложений предшествует своя последовательность команд загрузки и записи.

Для создания данного кода генератор кода хранит некоторую информацию о том, что произойдет во время выполнения сгенерированного им кода. Работа по хранению этой информации называется моделированием производительности. Модель, задаваемая на языке ПРОЛОГ, должна быть корректной, иначе получаются неверные следствия, выводы [7].

Блок распределения памяти отвечает за организацию памяти машины во время выполнения скомпилированной программы. Один из способов представления распределения памяти показан на рис. 2.

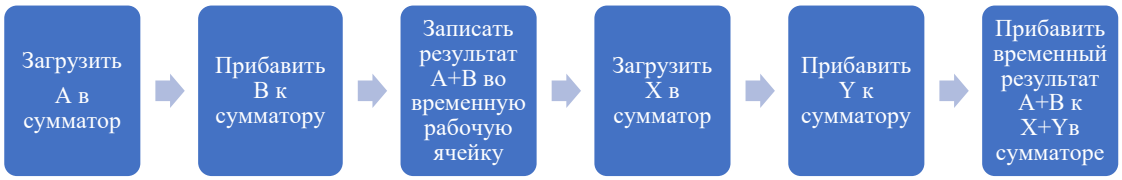

Рис. 1. Процедура вычисления (A + B) + (X + Y)

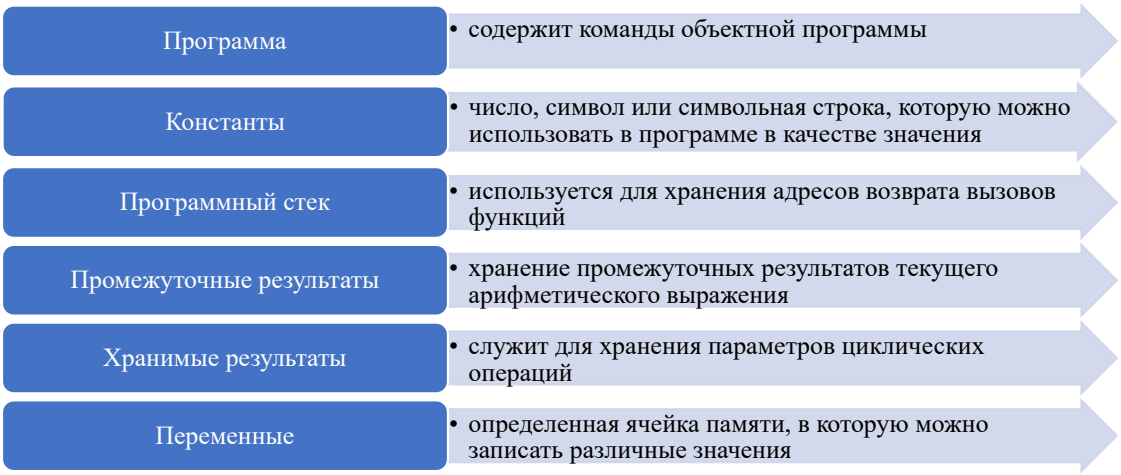

Рис. 2. Способ представления распределения памяти

С каждой из областей связывается переменная периода исполнения, которые являются указателями на очередную свободную ячейку памяти соответствующей ей области. Если значение переменной равно нулю, то переменная в данный момент находится в сумматоре.

Элементы таблиц содержат элементы о параметрах атомов. С целью оптимизации программного кода табличные элементы представляются с помощью двоичных справочников. Табличные элементы формируются на этапе лексического и синтаксического анализа.

Процедура поиска элемента таблиц может быть описана следующими отношениями:

```
lookup(Key,dict(X,Left,Right),Value):- !,
   X=Value.
lookup(Key,dict(X,Left,Right),Value):-
Key<Key_1, lookup(Key,Left,Value);
Key>Key_1, lookup((Key,Right,Value).
```

Поиск осуществляется по ключу Key (поле «Вид»). Значением Value (поле «Значение» или «Адрес») может быть целочисленная или вещественная величина или адрес идентификатора исходной программы, адрес именованной метки. Двоичный справочник dict(Key, X, Left, Right) упорядочен в лексикографическом порядке имен.

Блок управления сумматором следит за содержимым сумматора в период исполнения программы. Для его функционирования используется переменная периода компиляции, которая содержит информацию о текущем состоянии сумматора. Он также используется для генерации команд запоминания, обновляя одновременно информацию о соответствующем табличном элементе. Если сумматор не занят, значение переменной равно нулю. Если сумматор занят, то он содержит указатель на табличный элемент, который соответствует промежуточному результату. Генератор кода взаимодействует с блоком управления сумматором через набор процедур.

Процедуры для атомов. Множество атомов определяется набором ключевых слов и операций исходного языка. В генераторе кода должен присутствовать вектор начального перехода, использующий имя входного атома для перехода к процедуре, обрабатывающий этот атом. Если установлен флаг ошибки, генерация кода прекращается. Управление передается предыдущим анализаторам.

Составление программы для решения простой вычислительной задачи на языке Prolog потребует освоения особого подхода, связанного со спецификой логического программирования [8]. Концепция выполнения программы неразрывно связана с формальным определением программ [9]. В предлагаемой статье рассматриваются атомы для вычисления арифметических операций, например addition (p, q, r) – сложение, subtraction (p, q, r) – вычитание, multiplication (p, q, r) – умножение, division (p, q, r) – деление. Здесь p, q, r – соответственно левый, правый операнды и результат операции.

Обобщенный алгоритм функционирования процедур следующий:

```
if(Address(p) = 0) then
{
   // p левый операнд в сумматоре
   // сумматор использован
   gen(op, q); // op – операция, q – правый
   операнд, результат в сумматоре
   load(q); //  сохранение результата
}
else (Address(q)=0) then
{
   // q правый операнд в сумматоре
   // сумматор использован
  gen_ reverse (op, p); // gen_ reverse – обратная
   операция
   load(p); //  сохранение результата
}
```

Основное отношение, моделирующее генератор объектного кода, описывается выражением encode:

encode(Structure, Dictionary, Code), где:

− Structure – дерево вывода арифметического выражения, построенного синтаксическим анализатором;

− Dictionary – словарь идентификаторов и меток;

− Code – объектный код, который представляет собой процедуру, содержащую ассемблерный код некоторой машины, выполняющей действие, предписанное данной операцией.

При разработке генератора объектного кода набор процедур, выполняющих арифметические действия (суммирование, вычитание и т.д.), операции загрузки, передачи управления, готовится заранее. Эти процедуры должны учитывать семантику своих аргументов. В частности, аргумент может быть целочисленным или вещественным (константой) или же идентификатором. Анализ построенных и апробированных процедур динамической генерации состояний и генерации фактов позволяет говорить о применимости такого решения для определенных прикладных задач [10].

Оптимизация одного атома. Некоторые простые виды оптимизации можно

осуществить, выясняя для каждого атома, образуют ли значения его параметров один из выделенных частных случаев, позволяющих генерировать более эффективный код. Например, можно посмотреть, не окажется ли один из операндов атома сложения константой, равной нулю. Это возможно, когда оба операнда – константы, тогда сложение можно выполнить в период компиляции, и генерировать код также не требуется.

Другой пример – возведение в степень, равную некоторой небольшой константе. Например, для возведений в квадрат проще сгенерировать код умножения вместо генерации вызова функции возведения в степень.

Иногда система команд машины имеет команды, используемые только в специальных случаях, например запись нуля в ячейку памяти.

Оптимизация фиксированной цепочки атомов. В некоторых случаях возможности для оптимизации можно выделить, исследуя некоторое фиксированное число последовательных атомов, называемых «окном». Пусть в грамматике языка имеется правило [6]:

<operator> -> if(<logical_expression>) then <operator>

Соответствующее правило атрибутной транслирующей грамматики, используемой в синтаксическом анализаторе, может иметь вид

<operator> – > if(<logical_expression>)$_{R1}$ then
{ crossing_the_lie $_{R2,L1}$} <operator>
{label $_{L2}$}
(L1,L2) <- NewAtom (R2 <- R1)

Процедура NewAtom дает указатель на новый табличный элемент для метки.

Если за then следует оператор перехода, то такому условному оператору может соответствовать типичная цепочка атомов:

transition_in_default (39,62) transition (92) label (62)

Эту цепочку атомов можно заменить более эффективной цепочкой:

transition_in_truth (39,92).

Оптимизация одного оператора. Переупорядочение выражений. Часто возможно генерировать более эффективный код для вычисления арифметических выражений, если использовать порядок вычислений, отличный от порядков атомов, соответствующий польскому переводу исходного выражения. Допустим, имеется выражение A*B-C*D. Если выполнять команды в соответствии с порядком польской записи, то объектный код будет состоять из восьми команд. Однако если выражение переупорядочить таким образом, чтобы C*D вычислялось раньше, чем A*B, то количество команд сократится до шести.

Вызовы процедур. Имеется возможность сокращения времени выполнения процедур (функций), вызываемых внутри оператора. Во многих императивных языках имеются Inline-функции, подставляемые в точку вызова. Время выполнения существенно снижается, однако растет объем памяти, занимаемый программой.

Комбинирование переходов. Во многих практических приложениях используются условные и безусловные переходы на метку, а первый атом после нее – это переход на некоторую другую метку. В этом случае первый переход можно заменить переходом на вторую метку. Например, имеется выражение

if(x>y) z = q else goto L;

Условный переход в случае x = <y можно адресовать непосредственно метке L, а код для оператора goto L генерировать не нужно.

Оптимизация нескольких операторов. Общие подвыражения. Общим подвыражением называется такое подвыражение, которое встречается в программе более одного раза. Например, (A+B)*C+D/(A+B),

Здесь сложение A+B достаточно выполнить только один раз.

Общие подвыражения часто используются при вычислении адресов индексированных переменных, например, в операторе

arr_1(i,j) = arr_1(i,j)+arr_2(i,j);

адрес переменной arr_1(i,j) достаточно вычислить один раз.

Выполнение константных действий. Операции над константами можно выполнить на этапе компиляции, не откладывая их до периода исполнения программы. Во многих случаях результаты таких операций можно распространить по нескольким операторам. Например,

```
int a = 1;
int b = 2;
int c = a+b;
```

сложение в третьем операторе можно выполнить на этапе компиляции, а сам оператор заменить более эффективным

int c = 3;

Распространение констант очень часто используется при вычислении адресов индексированных переменных, в тех случаях, когда значения индексов можно определить на этапе компиляции.

Оптимизация циклов. Вынесение кода или чистка цикла. Во многих случаях часть кода можно вынести так, чтобы она исполнялась перед выполнением цикла. Например, если в теле цикла используется подвыражение A*B, причем A и B не изменяют своих значений в пределах цикла, то это произведение можно вычислить один раз перед входом в цикл, а в теле цикла использовать результат.

Расчленение циклов. В некоторых случаях цикл можно разбить на два цикла, из которых будет выполняться только один. Это позволит вынести из тела основного цикла многие операторы, существенно сократив время выполнения и машинные ресурсы.

Уменьшение силы операции. Преобразования касаются замены одной операции другой, выполняемой быстрее. Например, в некоторых случаях возможно использование операции сложения вместо операции умножения.

Развертывание цикла. Развертывание цикла означает уменьшение числа повторений цикла за счет выполнения вычислений, соответствующих двум повторениям старого цикла. Данное преобразование уменьшает число проверок переменных цикла.

Линеаризация массивов. Линеаризация n-мерного массива означает, что компилятор генерирует код так, как если бы массив был одномерным. Например, двумерный массив

```
int arr[20][20] array_int;
```

следует трактовать как одномерный

```
int arr[400] array_int;
```

Заключение

Модель исследования научного предмета логического программирования значительно отличается от алгоритмического подхода, который применяется в императивных языках. Логическая программа является базой знаний и не содержит последовательности инструкций, который предполагает порядок выполнения вычислений. Последовательность работы с такой базой подразумевает правильную формулировку вопросов к ней и автоматизированное заключение решений. Механизм логического вывода уже интегрирован в среду программирования, и программист должен только результативно им управлять. На начальном этапе логического программирования программисту сложнее всего отказаться от алгоритмического мышления в пользу декларативного. Однако на настоящий момент развития компьютерной техники полный переход к декларативному программированию невозможен, поскольку сами компьютеры работают последовательно, а результат поиска решений существенно зависит от порядка записи и, следовательно, унификации фактов и правил в логической программе. Логический язык программирования можно использовать наиболее эффективно для решения комбинаторных задач, написания систем поддержки принятия решений, разработки экспертных систем и моделирования интеллектуальной деятельности.

Список литературы

1. Обломов И.А., Ржавин В.В., Краснов В.И., Фадеева К.Н. Проблемы преподавания декларативных языков программирования // Вестник Чувашского государственного педагогического университета им. И.Я. Яковлева. 2020. № 1 (106). С. 204–210. DOI: 10.37972/chgpu.2020.69.53.026.

2. Обломов И.А., Ржавин В.В., Первова Н.В., Герасимова А.Г. Применение модели синтаксически управляемого перевода арифметических выражений в процессе обучения студентов // Вестник Чувашского государственного педагогического университета им. И.Я. Яковлева. 2020. № 1 (106). С. 211–218. DOI: 10.37972/chgpu.2020.28.84.027.

3. Марченков С.А. Автоматизация процессов программирования агентов на основе кодогенерации при построении семантических сервисов интеллектуальных пространств. Ч. 2 // Программная инженерия. 2019. Т. 10. № 11–12. С. 440–450. DOI: 10.17587/prin.10.440-450.

4. Кушнир Н.В., Кушнир А.В., Кабанова А.Л., Гвозденко А.А., Деева И.Ю. Проектирование семантических сетей на языке программирования Visual Prolog // Электронный сетевой политематический журнал «Научные труды КубГТУ». 2015. № 11. С. 461–466.

5. Половикова О.Н., Зенков А.В. Решение некоторого класса логических задач на языке Prolog декларированием генераторов состояний // Компьютерные инструменты в образовании. 2019. № 1. С. 54–67. DOI: 10.32603/2071-2340-2019-1-54-67.

6. Oblomov I.A., Rzhavin V.V., Pervova N.V., Fadeeva K.N., Gerasimova A.G. Computing process as a diagnostic object // IOP Conference Series: Materials Science and Engineering, Krasnoyarsk, 16–18 april 2020). Krasnoyarsk Science and Technology City Hall of the Russian Union of Scientific and Engineering Associations. Krasnoyarsk: Institute of Physics and IOP Publishing Limited. 2020. P. 52023. DOI: 10.1088/1757-899X/862/5/052023.

7. Тюрин С.Ф., Городилов А.Ю. Особенности логического вывода в ПРОЛОГ-программах // Вестник Пермского университета. Математика. Механика. Информатика. 2019. № 3 (46). С. 91–97. DOI: 10.17072/1993-0550-2019-3-91-97.

8. Здор Д.В. Теоретические основы организации ветвлений и повторений в программах на языке логического программирования Пролог // Advanced Engineering Research. 2021. Т. 21. № 2. С. 200–206. DOI: 10.23947/2687-1653-2021-21-2-200-206.

9. Ланец С.А. Программирование на языке PROLOG интеллектуальных задач // Вестник Российского нового университета. Серия: Сложные системы: модели, анализ и управление. 2021. № 2. С. 119–128. DOI: 10.25586/RNU.V9187.21.02.P.119.

10. Половикова О.Н., Ширяев В.В., Оскорбин Н.М., Смолякова Л.Л. Особенности программной реализации логических задач на языке Prolog // Известия Алтайского государственного университета. 2021. № 1 (117). С. 116–120. DOI: 10.14258/izvasu(2021)1-20.