Ленинградский ордена Ленина и ордена Трудового Красного Знамени Государственный Университет им. А.А. Жданова

ТЕРЕХОВ Андрей Николаевич

Методы синтеза эффективной рабочей программы

Диссертация
на соискание ученой степени
кандидата физико-математических наук

Научный руководитель:
доктор физико-математических наук
ЦЕЙТИН Г.С.

Ленинград
1976

Введение

Задача синтеза рабочей программы возникла впервые в связи с трансляцией формул. Было установлено, что для трансляции формул удобно переводить формулы в бесскобочную запись (чаще всего в прямую или обратную польскую запись), а затем интерпретировать полученную запись с помощью стека. Позднее выяснилось, что всю программу (а не только формулы) удобно представлять тем же способом; например, `a := b` можно представить `a, b, :=`, причем здесь `a` и `b` могут быть произвольными конструкциями. Для машин, не имеющих аппаратной стековой памяти, оказалось, что эффективность рабочей программы можно существенно улучшить, если вычисление адресов, соответствующих позициям стека, производить во время трансляции. Так появились методы синтеза рабочей программы, основанные на псевдоисполнении программы, на использовании стека периода компиляции и т.д.

Известные подходы к синтезу рабочей программы условно можно разделить на два класса: подходы, ориентированные на получение эффективной рабочей программы, и подходы, ориентированные на быструю трансляцию. Если ставится задача получения эффективной рабочей программы, то обычно используются методы глобальной оптимизации, предусматривающие иногда существенную перестройку синтаксической структуры текста программы. Такая глобальная оптимизация связана со значительными затратами времени на трансляцию. В противоположность этому быстрые методы синтеза основаны на последовательных линейных просмотрах транслируемого текста и используют дополнительно только информацию об исходной синтаксической структуре, получаемую при помощи стека.

Во многих языках программирования (например, в АЛГОЛе 60, PL/I) оптимизация генерируемой программы при использовании синтаксических методов трансляции затруднена тем, что информация о виде обрабатываемого программой значения не отражена полностью в синтаксисе и поэтому в рабочую программу приходится вставлять дополнительные команды для учета этой информации во время счета.

АЛГОЛ 68 с его полным контролем видов и ортогональностью конструкций открыл возможности для оптимизации программы синтаксическими методами.

При разработке транслятора с АЛГОЛа 68 в Ленинградском университете был использован следующий метод (предложенный в начале 1972 года Г.С. Цейтиным). В этом методе способ размещения значения в памяти не фиксируется (в отличие от традиционного метода, предписывающего размещение каждого нового значения на вершине стека), а выбирается отдельно для каждой конструкции исходной программы. Для этого определяется способ обмена информацией между программами транслятора, синтезирующими различные конструкции рабочей программы. Разумеется, программа, транслирующая определенную конструкцию, вызывается по отдельности для каждого вхождения такой конструкции, и фактически обмен информацией идет между вызовами. Информация распространяется в двух направлениях: конструкция, использующая значение, вычисляемое ее подконструкцией, передает ей запрос на значение (указание на возможном способе его

размещения), а подконструкция, в свою очередь, возвращает информацию о фактически выбранном способе.

При трансляции для машин ЕС ЭВМ были приняты следующие способы размещения значений:

1) на регистре;
2) в области памяти, адрес которой может быть задан статистически;
3) в области памяти, адрес которой определяется через один или два регистра;
4) для логических значений – в разрядах кода условия слова состояния программы с указанием маски;
5) значения являются константой и не нуждаются в специальном размещении в памяти.

Были определены следующие типы запросов на значения:

1) V – пустой запрос (значения доставлять не нужно);
2) R – регистровый запрос (разместить значение в регистре, указанном в запросе);
3) A – запрос типа присваивания (разместить значение в области памяти, адрес которой задан в вершине стека периода компиляции или для случая логических значений передать в зависимости от результата управление по этому адресу);
4) F – свободный запрос (конструкция размещает значение так, как ей удобно; информация о размещении выдается на вершину стека периода компиляции).

Предполагается, что самыми частыми запросами будут свободные и что в ответ на свободный запрос после генерации команд, формирующих значение, большинство конструкций не будет порождать команды, специальным образом размещающие это значение. В частности многие конструкции (идентификаторы, константы, а также некоторые описания тождества, выборки, формулы и др.) вообще не будут порождать ни одной команды.

Таким образом, программа генерации каждой конструкции языка должна на основе информации о полученном ею запросе на значение и о способе размещения значений ее подконструкций породить команды рабочей программы, наиболее эффективные в данной ситуации.

При этом для порождения команд удобно использовать процесс макрогенерации. В качестве первого варианта был использован макрогенератор языка ассемблер ЕС ЭВМ. Однако этот макрогенератор оказался не вполне подходящим средством, так как:

a) это язык довольно низкого уровня, не предусматривающий широких возможностей для представления разнообразной информации, необходимой во время трансляции;
b) не вся информация, используемая ассемблером, доступна программисту;
c) макроязык ассемблера позволяет выбирать порождаемые команды в зависимости от предшествующей части программы, но не дает средств для получения информации о последующей части программы.

Из-за этих особенностей макроязыка ассемблера при выборе представления информации и при написании макроопределений приходится применять довольно искусственные приемы.

Более того, были обнаружены ситуации, где однопроходных средств недостаточно для выбора наилучшего способа размещения значения (см. §1).

В данной работе предлагается способ построения и реализации другого языка, специально созданного для фазы синтеза компилятора с АЛГОЛа 68; однако этот язык имеет и самостоятельное значение и может быть использован в других компиляторах. Описываются некоторые обобщения упомянутых выше методов, связанные с двухпросмотровой схемой синтеза и направленные на дальнейшее повышение эффективности рабочей программы.

Порождение рабочей программы при помощи макрогенерации используется во многих разработках, но работ, близких по методам передачи информации и по качеству получающейся рабочей программы, практически нет. Среди известных автору работ заслуживает упоминания схема синтеза П. Бранкара и Ж. Леви [1]. В этой работе также допускаются различные способы представления значений, синтез осуществляется с помощью абстрактного стека периода компиляции (в один просмотр), но вместо запросов на значения каждой конструкции передается информация о названиях всех объемлющих конструкций. Это позволяет получать достаточно хорошую рабочую программу, но существенно увеличивает число сравниваемых вариантов и длину макроопределений.

Методы синтеза рабочей программы, аналогичные описываемому в данной работе двухпроходному методу синтезе с автоматическим сравнением многих вариантов программирования, в литературе не встречается.

§1. Задача учета информации о последующем тексте программы

Как оказалось, метод однопросмотрового синтеза недостаточно эффективен в тех случаях, когда нужно выбирать какое-то представление значения или вариант программирования в зависимости от будущей ситуации. В особенности это касается использования регистров. Ответ на вопрос: "Оставить ли данное значение на регистре или сразу выгрузить его в память?" – зависит не только от наличия свободных регистров в данный момент, но и от потребности в регистрах в ходе дальнейшей работы. Это можно проиллюстрировать на примере программирования последовательных и условных предложений, а также предложений выбора (далее эти конструкции будем называть ветвящимися). Пусть перед началом программирования ветвящейся конструкции было занято несколько анонимных (т.е. содержащих промежуточные результаты вычислений) регистров. Пусть далее, при программировании некоторых ветвей этой конструкции мы вынуждены из-за нехватки регистров выгрузить внешний по отношению к данной конструкции анонимный регистр. В этом случае лучше было бы выгрузить этот регистр перед входом в ветвящуюся конструкцию, так как:

a) конструкция, следующая за ветвящейся, должна иметь фиксированную информацию об этом регистре (во время трансляции не известно, какая ветвь будет исполняться во время счета);
b) появляется возможность уменьшить длину рабочей программы (одна команда выгрузки вместо нескольких в разных ветвях).При этом в тех ветвях, в которых удалось воспользоваться значением, хранящимся в этом регистре, можно пользоваться и после выгрузки (на правах остаточно информации).

Задача эффективного программирования с учетом вариантов, возникающих в дальнейшем, может быть решена с помощью двухпроходной схемы синтеза: во время первого просмотра входного текста порождается рабочая программа, содержащая несколько вариантов для каждого спорного случая; во время второго (обратного) просмотра происходит сравнение вариантов и выбор лучшего из них. В частности, решение поставленной выше задачи о выгрузке анонимного регистра в память может быть следующим:

- в первом просмотре для каждого анонимного значения, которое находится в регистре и не используется сразу же после получения, программируем условную выгрузку этого регистра в память. Далее помечаем каким-либо образом те анонимные значения в регистрах, которые из-за нехватки регистров должны быть выгружены в память до своего использования;
- во втором просмотре исключаем из текста ненужные выгрузки.

Заметим, что решение оказалось столь простым благодаря тому, что к моменту использования анонимного значения в регистре уже известно, что этот регистр не был использован для другой цели. В более сложных задачах, связанных с повышением эффективности рабочей программы, могут возникнуть ситуации, в которых представление значения не выбрано окончательно к моменту его использования. Если при появлении подобной альтернативы мы всегда будем начинать независимое программирование двух (или более) вариантов, его придется снова раздроблять на подварианты и т.д. и в результате число рассматриваемых вариантов быстро выйдет за разумные границы. Однако

поиск наилучшего варианта при учете всех таких альтернатив может быть облегчен за счет следующих двух обстоятельств:

a) две разные альтернативы могут оказаться "ортогональными" друг другу, т.е. изменения в порождаемой программе, обусловленные выбором решения по одной из альтернатив, не будут зависеть от решения по другой альтернативе;
b) каждое принимаемое решение касается программирования определенной конструкции и, вообще говоря, не влияет на программу после окончания этой конструкции.

Предлагаемый язык как раз и рассчитан на то, чтобы описывать порождение рабочей программы с учетом альтернатив, разрешаемых при дальнейшей работе, но без явного разветвления генерируемой программы на много вариантов. В последующем тексте дается описание первого варианта такого языка и реализующей его системы.

§2. Описание языка

1. Многие конструкции в языках высокого уровня имеют весьма широкие области применения и разнообразные варианты исполнения. Так, например, в АЛГОЛе 68 присваивание может работать с простыми переменными, массивами, процедурами и т.п. Более того, при одном и том же виде величин могут порождаться разные команды в зависимости от способа размещения этих величин. Очевидно, что для повышения эффективности рабочей программы необходимо подставить вместо вхождения конструкции языка последовательность машинных команд, кратчайшим образом реализующих данное действие. Для этой цели удобно использовать процесс макрогенерации, в котором роль макроопределений играют программы перевода различных конструкций. Макрогенератор, реализующий такие макроопределения, должен удовлетворять следующим требованиям:

a) язык макрогенератора должен быть языком высокого уровня, т.к. построение эффективной рабочей программы требует использование сложных макроопределений;
b) язык макрогенератора должен включать такие специфические для задачи трансляции средства, как магазины, списки, деревья и т.д.
c) макроопределения, входящие в состав транслятора, используются весьма интенсивно, поэтому макрогенератор должен удовлетворять жестким требованиям эффективности.

Последнее требование можно ослабить, заметив, что работа макрогенератора делится на два этапа: подготовка внутреннего машинного представления макроопределения, которая происходит сравнительно редко (во время генерации транслятора в рамках операционной системы), и работа подготовленных макроопределений над конкретными программами. Ясно, что повышение эффективности первого этапа не так важно по сравнению со вторым.

В данной реализации в качестве основы языка макрогенератора предлагается АЛГОЛ 68. Это новый, чрезвычайно мощный язык программирования, имеющий стройное ортогональное описание, благодаря которому можно добиться эффективной его реализации. Некоторые ограничения на язык, обусловленные описываемой реализацией, будут рассмотрены далее.

Разумеется, практическое применение данного подхода не предполагает, что этот язык будет сразу реализован с полным объемом средств АЛГОЛа 68 (наоборот, он строится до реализации полного АЛГОЛа 68 и с целью ускорить такую реализацию). На первом этапе будет использоваться вариант языка с минимальным набором средств, а полный вариант можно построить методом раскрутки.

2. В тексте на промежуточном языке, который поступает на вход фазы синтеза, каждая конструкция входного языка, допускающая внутренние конструкции, имеет "рамочную" структуру, т.е. образы этих внутренних конструкций вставлены между символами, относящимися собственно к данной конструкции. При использовании макроязыка ассемблера каждый такой символ трактуется как отдельная макрокоманда. Эти макрокоманды в тексте удалены друг от друга, хотя по существу они составляют одно целое и должны

пользоваться общей информацией. Для того, чтобы обеспечить это средствами макроязыка ассемблера, приходится вводить специальные магазины.

В предлагаемом языке каждая конструкция рассматривается как одна процедура. Во время синтеза рабочей программы выполнение этой процедуры начинается при чтении первого символа, относящегося к такой конструкции, и продолжается до ее последнего символа. Чтение внутренних конструкций, находящихся между этими словами, связано с выполнением процедур, относящихся к этим конструкциям. Вызов этих процедур происходит внутри внешней процедуры, для инициирования такого вызова внутри внешней процедуры вставляется специальный символ `inner`.

Эта методически удобная конструкция хорошо реализуется стандартными средствами АЛГОЛа 68. Пусть текст на промежуточном языке (вход синтезирующей части компилятора) записан в последовательном наборе данных и состоит из кодов процедур (макрокоманд) и их параметров. В начале работы получает управление ведущая программа `inner`. Она читает из текста очередной (в данном случае – первый) код и вызывает соответствующую процедуру, которая читает из текста свои параметры и осуществляет синтез рабочей программы. В тех точках, где нужно вставить подконструкцию, вызывается (рекурсивно) процедура `inner`.

3. Информация, используемая только внутри данной процедуры, изображается при помощи идентификаторов, описанных в этой процедуре. Часть информации. Используемой данной процедурой, может быть глобальной, т.е. описанной вне всяких процедур.

В данном языке используется еще один вид описаний. Идентификаторы, вводимые с помощью таких описаний, будем называть сменяемыми. Если в данном блоке нет описания сменяемого идентификатора, то значением этого идентификатора является то же значение, которое было у него во внешнем блоке или в вызвавшей процедуре (если оно было там определено), в противном случае значение определяется его описанием. Таким образом, значение сменяемого идентификатора изменяется при входе в любой блок или процедуру, где имеется его новое описание, а при выходе восстанавливается прежнее значение. Сменяемый идентификатор реализуется следующим образом: такому идентификатору отводится глобальная ячейка памяти; при входе в блок, где есть новое описание этого сменяемого идентификатора, его значение отправляется в магазин, а при выходе из этого блока оно восстанавливается. Очевидно, что глобальные идентификаторы являются частным случаем сменяемых.

Рассмотрим пример:

```
(int a;...(new a:=a+1;...)...)
1          2
```

Переменная `a` имеет статически известную ячейку памяти в блоке 1. Во время работы конструкции `new a:=a+1` (новое описание) вычисляется `a+1`, затем текущее значение `a` отправляется в статическую память блока 2 (предполагается, что новые описания могут встречаться только во вступлениях из описаний), а новое значение посылается в ячейку `a` .При выходе из блока 2 в ячейке `a` восстанавливается старое значение.

В языках типа АЛГОЛ 60 или АЛГОЛ 68 процедура с глобальным идентификатором, вызванная в блоке, где вновь описан такой же идентификатор, использует то определение идентификатора, которое действует в месте описания этой процедуры, а не в месте ее вызова. Если же требуется,

чтобы процедура использовала информацию, относящуюся к месту вызова, то единственным способом в таких случаях является передача параметров, что может оказаться громоздким.

Использование описанных выше сменяемых идентификаторов во многих случаях позволяет передавать процедуре информацию о контексте вызова проще, чем через параметры.

4. Для того, чтобы иметь возможность учитывать информацию о последующем тексте программы, в язык вводятся особого вида логические переменные – предсказатели. Если при программировании некоторой конструкции нужно выбрать один из двух вариантов в зависимости от последующих конструкций, то в транслируемой конструкции вводится новый предсказатель и затем допускаются ветвления по этому предсказателю. Условие, задающее действительное значение предсказателя, не задается явно. Вместо этого все порождаемые команды получают определенную оценку "стоимости". Транслятор выбирает значения предсказателей таким образом, чтобы стоимость порождаемой программы была минимальной.

§3. Условные значения

1. Некоторые операторы в программе могут находиться внутри условных предложений и выполняться или не выполняться в зависимости от значений определенных предсказателей. Среди таких операторов могут оказаться и присваивания, и, таким образом, на выходе условного предложения, управляемого предсказателем, окажется, что значение, именуемое некоторым именем. Зависит от этого предсказателя. Далее эта зависимость может распространяться на результаты операций с этим значением, на условные предложения, управляемые результатом такой операции и т. д. Поэтому считается, что любое значение, над которым в этом языке производится действие, может быть условным, например, иметь вид: `if A then x elif B then y else z fi`.

Однако, учет этих условностей не должен усложнять написание макроопределений, и для автора макроопределений эти значения должны выглядеть как простые. Система, реализующая описываемый способ синтеза (далее будем говорить просто система), использует специальный способ представления этих значений в виде двоичного дерева; каждый узел соответствует некоторому предсказателю, а две дуги, выходящие из этого узла, соответствуют значениям предсказателя `true` и `false` . В висячих вершинах этого дерева находятся конкретные значения. Операции над такими деревьями (арифметические операции, присваивания, сравнения и т.п.) выполняются при помощи специальных процедур.

2. Наиболее сложная задача при реализации такой системы – это уметь в каждый момент определить, какое значение именуется данным именем в результате произведенных до данного момента присваиваний, каждое из которых было выполнено под некоторым условием. В частности, трудно установить, исполнялось ли некоторое присваивание, если в программе есть переходы и метки. Мы будем рассматривать такой язык, в котором метки произвольные переходы запрещены, но зато допускаются условные предложения и циклы. Это ограничение не очень существенно, так как известно, что любая программа с переходами в принципе может быть преобразована к виду, содержащему только условные предложения и циклы.

Для правильного исполнения условных присваиваний вводится специальная переменная, образующая текущее условие. Значением этой переменной является условное логическое выражение, т.е. двоичное дерево описанного выше вида, в висячих вершинах которого находятся логические значения. Перед началом генерации программы эта переменная получает значение `true`. Другие значения она получает в условных предложениях и циклах.

Кроме того, значение текущего условия изменяется, если обнаруживается, что при некотором сочетании значений предсказателей программирование не может продолжаться (например, из-за отсутствия свободных регистров); в этих случаях к текущему условию конъюнктивно присоединяется условие продолжимости генерации.

Значение текущего условия влияет на исполнение присваиваний и генерацию рабочей программы. Если при текущем условии `B` исполняется присваивание `x:=y`, то фактически имени `x` присваивается значение `if B then y else x fi` (технически эта операция несколько сложней, поскольку `B, x, y` могут быть условными). Если же при текущем условии `B` генерируется текст

рабочей программы, то он должен быть снабжен пометкой, что в окончательную программу он войдет, лишь если значение B в конечном счете окажется true. Фактически такие пометки в рабочей программе достаточно ставить лишь в тех точках, где текущее условие изменяется.

3. Рассмотрим условное предложение if A then S fi. Пусть перед входом в это предложение значением текущего условия было B. Тогда S исполняется при значении текущего условия B&A. После выхода из условного предложения для текущего условия можно восстановить значение B (фактически мы поступаем иначе, см. §5, п.6).

Условное предложение if A then S1 else S2 fi в принципе реализуется так же, как

bool AA=A; if AA then S1 fi; if not AA then S2 fi.

Рассмотрим цикл while A do S od. Предположим в начале, что известное целое число N, ограничивающее сверху число повторений цикла. Тогда для данного цикла можно определить эквивалентное выражение S_N следующим образом:

в качестве S_0 берем skip;

в качестве S_{k+1} берем if A then S; S_k fi.

Способ реализации таких предложений уже описан. На самом деле не требуется определять такое N заранее. При выполнении цикла мы будем входить во вложенные условные выражения достаточное число раз, пока текущее условие не окажется тождественным false. Этот процесс может, вообще говоря, привести к бесконечному циклу, но это не будет значить, что такой бесконечный цикл возможен и при некоторых конкретных, а не условных значениях используемых величин, т.е. в этом случае макроопределение и не должно оканчивать работу.

§4. Выбор значений предсказателей

Как уже говорилось, новый предсказатель вводится во время генерации кодов для некоторой конструкции, допускающей альтернативные варианты программирования; далее в процессе генерации возникают ссылки на этот предсказатель, либо непосредственные, либо через условные значения переменных, в которые вошел этот предсказатель. Окончательное значение предсказателя выбирается так, чтобы минимизировать "стоимость" порожденной программы. Это можно было бы сделать в конце программы сразу для всех предсказателей, но для упрощения работы желательно хотя бы для части предсказателей делать это раньше.

В принципе значение предсказателей может быть выбрано без ущерба для минимизации в любой момент, когда кончились все ссылки на этот предсказатель как прямые, так и косвенные, и единственным условным значением, зависящим от этого предсказателя, является текущая стоимость порожденной части программы. Практически в большинстве случаев удобнее выбирать значение предсказателей при окончании той конструкции, для которой он был введен. Прямых ссылок на этот предсказатель после этого момента уже не будет, а потерями, связанными с тем, что при минимизации не будет учтен эффект возможных в будущем косвенных ссылок, мы пренебрегаем. Предсказатели, для которых принят такой режим, будем называть локальными. В принципе можно рассматривать глобальные предсказатели, значение для которых выбираются или в момент, указанный автором макроопределения, или после того, как станут недоступными все условные значения, использующие этот предсказатель (кроме текущей стоимости).

Выбор значения предсказателя `A` состоит из следующих шагов:

a) в текущее условное значение стоимости `C` вместо `A` подставляются поочередно `true` и `false`, что дает условные значения `CT` и `CF` соответственно;

b) вычисляем условное логическое значение `CT<CF` и присваиваем его `A` (таким образом, выбранное значение `A` может зависеть от других предсказателей); информация об этом присваивании выдается в выходной текст (см. §5, п.5);

c) во всех условных значениях, доступных в данный момент и содержащих `A`, вместо `A` подставляем присвоенное ему значение.

Описанный процесс будем называть исключением предсказателя.

В связи с тем, что локальные предсказатели исключаются автоматически при выходе из макроопределения, в котором они введены, для них применяется магазинный способ присвоения номеров. Именно, при входе в макроопределение запоминается максимально занятый номер предсказателя, далее предсказателям присваиваются последовательные номера, а при выходе из макроопределения все предсказатели с номерами, большими начального, исключаются; в последующих конструкциях эти номера могут быть использованы повторно.

§5. Представление условных значений

1. Опишем подробнее представление условных значений в памяти машины и процедуры, работающие с этим значениями. Как уже было сказано, условные значения представляются двоичными деревьями; вид таких деревьев можно описать следующим образом:

```
mode узелX = struct (int предсказатель, значениеX да, нет);
mode значениеX = union (X, ref узелX);
```

такие описания даются для каждого вида X, допускаемого реализацией.

В начале работы системы вся память, отведенная для хранения деревьев, разбивается на ячейки, доступные для хранения значений видов узелX; эти ячейки связываются в список свободных ячеек. Имеется процедура, доставляющая очередную свободную ячейку, а также процедура "сборки мусора", которая вызывается при исчерпании списка свободных ячеек.

Система обрабатывает все идентификаторы простых переменных, а также элементы массивов как значения видов значениеX. Такая замена видов происходит автоматически.

2. Основным способом выполнения различных действий над значениями, представленными в виде двоичных деревьев, является копирование двоичного дерева в свободных ячейках памяти с выполнением соответствующего действия на каждой висячей вершине этого дерева. В результате такого действия вместо висячей вершины может быть, в частности, вставлена копия другого дерева, полученная аналогичным процессом. Например, для того, чтобы получить сумму двух значений, заданных в виде двоичных деревьев, мы можем копировать дерево первого слагаемого, выполняя в каждой его висячей вершине следующее действие: запоминаем число, стоящее в этой вершине, затем копируем второе слагаемое с увеличением числа в каждой его висячей вершине на запомненное число из первого слагаемого, получившееся дерево вставляем на место этого числа в копию первого слагаемого. Фактически такое копирование не обязательно будет копированием полного дерева. Именно, при копировании первого дерева мы запоминаем в магазине путь от его корня к текущей вершине и, таким образом, мы имеем список значений всех предсказателей, по которым производилось ветвление на этом пути. Когда в висячей вершине первого дерева мы начнем копирование второго дерева, то оно будет выполняться уже при определенных значениях предсказателей из первого дерева; поэтому, встречая во втором дереве ветвление по предсказателю, значение которого уже записано в магазине, мы не копируем узел с таким ветвлением, а просто сразу идем по нужной ветви и рассматриваем встретившуюся там вершину вместо данного узла. Такое неполное копирование может применяться и к первому копируемому дереву, если текущее условие конъюнктивно содержит некоторые предсказатели или их отрицания. Список таких предсказателей выделяется из текущего условия посредством специальной операции (см. §5, п.4) и добавляется к началу магазина, используемого для копирования.

Рассмотрим как используется описанная техника при действиях над условными значениями.

Для присваивания переменной значения сначала копируется дерево текущего условия, после чего висячие вершины со значением true заменяется копией дерева источника, а висячие вершины со значением false – копией

дерева значения, именуемого получателем. Полученное дерево объявляется новым значением переменной. Таким образом, при тех значениях предсказателей, при которых текущее условие есть `false`, переменная своего значения не изменяет.

Для получения вырезки (в данной реализации используются только одномерные массивы; в индексных позициях могут стоять только индексы) сначала копируется дерево индекса, после чего каждая висячая вершина со значением `i` заменяется копией дерева `i`-го элемента массива. Полученное дерево объявляется значением вырезки.

3. Особого рассмотрения заслуживает присваивание, в котором значение получателя может быть условным. Пусть, например, присваивание имеет вид: `a[i]:=b`. Здесь `i` и `b` условные значения; кроме того, для каждого конкретного значения `j` идентификатора `i` значение `a[j]` также может быть условным.

Обходим дерево индекса, при этом заносим в список различные значения, которые записаны в висячих вершинах. Затем для каждого значения `j` из этого списка осуществляем следующие действия: копируем дерево индекса, каждую висячую вершину со значением `j` заменяем на копию дерева текущего условия, а его висячие вершины со значениями `true` и `false` заменяем на копии деревьев `b` и `a[i]` соответственно; висячие вершины дерева индекса со значениями, не равными `j`, заменяем на копии дерева `a[j]`. Полученное дерево объявляем новым значением `a[j]`.

Различные операции, соответствующие формулам АЛГОЛа 68 с операциями из стандартного вступления, реализуются аналогично описанной выше операции сложения.

Подстановка условного логического значения вместо предсказателя, входящего в некоторое условное значение, осуществляется следующим образом: происходит обход дерева, в котором нужно выполнить подстановку; если встречается узел с заменяемым предсказателем, то запоминаются условные значения, находившиеся на ветвях "да" и "нет" этого узла, а вместо самого этого узла вставляется копия подставляемого условного логического значения, в которой вместо висячих вершин со значениями `true` и `false` в свою очередь подставляются копии запомненных значений ветвей "да" и "нет" соответственно.

4. В описываемой системе существенно используются условные значения, представленные в виде деревьев. Каждое действие с условными значениями связано с копированием деревьев, подстановкой деревьев вместо висячих вершин и т.д. При этом, если не принять специальных мер, объем памяти, отведенной для хранения деревьев, может оказаться недостаточным даже для сравнительно простых случаев.

Очевидно, что задача упрощения деревьев с произвольными конкретными значениями включает в себя упрощение деревьев с булевскими значениями, т.е. задачу минимизации булевской формулы. Известно, однако, что полная минимизация булевской формулы в общем виде является трудоемкой задачей. Исходя из этого, в данной реализации не предполагается полная оптимизация деревьев. Используются лишь сравнительно несложные алгоритмы, позволяющие заметить следующие ситуации:

a) дерево эквивалентно константе;

b) дерево может быть преобразовано к виду:

`if A then x elif B then x elif...elif произвольное дерево fi`

Здесь x обозначает конкретное значение. A, B,... обозначают предсказатели или их отрицания.

В частности, если в таком виде представлено значение текущего условия, причем x есть false, то для генерируемого в данный момент варианта значения определенных предсказателей зафиксированы таким образом, что A,B,... все имеют значение false. Это используется для упрощения копирования деревьев (см. 2). Эквивалентность константе также используется в этой реализации (см. §3, п.4). Для приведения дерева к виду b) используется следующий алгоритм.

Строится таблица из двух столбцов, каждая строка этой таблицы соответствует одному предсказателю. Сначала таблица заполняется кодом "пусто". Далее обходится дерево, которое нужно оптимизировать. В каждой висячей вершине со значением A для каждой строки делается следующее. Если значение соответствующего показателя определено в магазине обхода дерева, то в нужный столбец, а в противном случае – в оба столбца записывается:

если было "пусто", то A;

если было A, то значение не изменится;

в других случаях пишется код "*".

После окончания обхода дерева просматривается таблица. Если найдется строка, в обоих столбцах которой записаны значения, отличные от кода "*", то результирующее значение будет либо константой (если эти два значения совпадают), либо будут состоять из одного узла, обе ветви которого – конкретные значения, взятые из этой строки таблицы.

Если найдется строка, в одном из столбцов которой записано конкретное значение, то пристраиваем дерево так, чтобы в его корне стояло ветвление по предсказателю, соответствующему этой строке таблицы. Одна из ветвей, исходящих из этого узла будет указанным конкретным значением, на второй ветви расположен "остаток" исходного дерева.

Может оказаться несколько строк таблицы, содержащих конкретное значение в одном из столбцов таблицы, (и тогда, как нетрудно догадаться, это конкретное значение будет всегда одним и тем же). В этом случае в корень дерева помещается узел, построенный для одной из тех строк, в начале остатка дерева становится узел для другой такой строки и т.д.

После того, как все такие строки исчерпаны, окончательный остаток дерева, присоединяемый к последнему из построенных узлов, получается копированием исходного дерева, выполняемым с учетом значений тех предсказателей, которые вошли в начальные узлы.

Заметим, что в данной реализации построение таблицы, используемой в алгоритме оптимизации деревьев, может быть совмещено с выполнением копирования. Эквивалентность дерева константе также выясняется во время копирования: при копировании очередного узла проверяется, являются ли обе ветви, исходящие из этого узла, совпадающими конкретными значениями. При положительном ответе этот узел не копируется, вместо него рассматривается конкретное значение. Таким образом, обнаруживается не только эквивалентность дерева константе, но также и эквивалентность константам его поддеревьев.

В заключение отметим, что этот алгоритм, конечно, не дает полной оптимизации деревьев, например,

а) не будет замечено, что дерево не зависит от некоторого предсказателя, хотя его и содержит;

b) не будут найдены эквивалентные поддеревья.

5. Генерируемые команды рабочей программы помещаются в выходной текст, который рассматривается как набор данных с последовательным доступом. Между командами помещается специальная информация, позволяющая на обратном просмотре выбрать последовательность команд рабочей программы в соответствии с окончательно выбранными значениями предсказателей.

Перечислим возможные составные части выходного текста:

1) записи на языке ассемблера;
2) управляющая информация:
   a) символ "если". Этот символ ставится в начале условного генерируемого участка программы. Во время обратного просмотра этот символ воспринимается как признак конца условной генерации.
   b) символ "илсе". Ставится в конце условно генерируемого участка программы. Так как при обратном просмотре он будет встречен первым, то при нем записывается условие генерации этого участка.
   c) символ "иначе". Разделяет два участка выходного текста, генерируемые при противоположных условиях. Этот символ соответствует символу `else` в тексте макроопределения.
   d) символ "установить". Записывается при исключении предсказателя. При нем указывается сам предсказатель и присваиваемое значение. При обратном просмотре этот символ предшествует всем условным значениям, зависящим от этого предсказателя.

В выходном тексте, как в записях на языке ассемблера, так и в управляющей информации, могут встречаться условные значения. Они представляются в виде линейной развертки соответствующих деревьев, аналогичной обратной польской записи, так что при обратном просмотре сначала читается предсказатель, по которому происходит ветвление, а за ним обе ветки.

Перейдем к описанию алгоритма обратного просмотра. Определяется логический массив `predict` для хранения значений предсказателей и целая переменная `flag` с нулевым начальным значением для подсчета скобок. Эта же переменная используется для определения условия генерации. Выходной текст просматривается с конца, при этом происходит распознавание управляющей информации среди записей на языке ассемблера. Записи на языке ассемблера при `flag = 0` переносятся в окончательный текст с заменой содержащихся в них условных значений на конкретные.

Если встретится символ "установить", то считывается предсказатель и присваиваемое ему значение (значение может быть условным, тогда оно вычисляется); затем значение предсказателя запоминается в массиве `predict`.

Символы "илсе", "иначе" и "если" обрабатываются по-разному, в зависимости от значения переменой `flag`.

Пусть `flag = 0`, тогда:

a) если встретится символ "илсе", то вычисляется значение следующего за ним условия генерации этого участка программы. Если это значение есть `false`, то `flag := 1`;
b) если встретится символ "иначе", то `flag := 1`;
c) символ "если" при `flag = 0` пропускается.

Рассмотрим теперь обработку символов "илсе", "иначе" и "если" при

`flag > 0`.

a) по символу "илсе": `flag := flag + 1`;

b) по символу "иначе": если `flag = 1`, то `flag := 0`;

c) по символу "если": `flag := flag - 1`.

6. Рассмотрим подробнее работу конструкций, влияющих на текущее условие и на значения предсказателей.

Условное предложение `if A then S1 else S2 fi`.

Действия, исполняемые по символу `then`: в выходной текст помещается управляющий символ "если", в магазине запоминается конъюнкция текущего условия и отрицания `A`; текущему условию присваивается новое значение – конъюнкция прежнего значения и `A`.

Действия, выполняемые по символу `else`: в выходной текст помещается управляющий символ "иначе"; текущее условие и вершина магазина обмениваются значениями.

Действия, выполняемые по символу `fi`: текущее условие выдается в выходной текст, затем выдается управляющий символ "илсе"; в качестве текущего условия берется дизъюнкция текущего условия и значения, записанного в вершине магазина; из магазина удаляется последняя запись.

Цикл `while A do S od`.

Действия, исполняемые по символу `while`: в некоторой сменяемой переменной запоминается текущее значение указателя вершины магазина; в магазине запоминается адрес первой команды `A`; в вершину магазина записывается значение `false`.

Действия, исполняемые по символу `do`: в выходной текст помещается управляющий символ "если"; в вершину магазина записыается дизъюнкция старого значения и конъюнкции текущего условия и отрицания `A`; текущему условию присваивается новое значение – конъюнкция прежнего значения и `A`; вычисленное текущее условие запоминается в магазине.

Действия, исполняемые по символу `od`: если текущее значение не тождественно `false`, то управление передается на программу `A` (по адресу, записанному в магазине), в противном случае все текущие условия, запомненные в магазине, выдаются в выходной текст, причем, вслед за каждым из них выдается управляющий символ "илсе"; в качестве текущего условия берется дизъюнкция текущих условий, накопленная в вершине магазина.

Начало и конец макроопределения.

При входе в макроопределение в магазине запоминается максимальный занятый номер локального предсказателя и текущее значение стоимости. Затем текущее значение стоимости устанавливается на 0. При выходе из макроопределения производятся следующие действия:

a) поочередно исключаются все локальные предсказатели, введенные в данном макроопределении;

b) вычисленное значение стоимости данного макроопределения добавляется к значению стоимости, запомненному при входе, и результат становится новым текущим значением стоимости;

c) восстанавливается прежнее значение максимального номера локального предсказателя;

d) из магазина удаляются записи, сделанные при входе.

7. В системе имеется процедура "сборки мусора", которая вызывается, когда список свободных ячеек для записи узлов уже исчерпан, но требуется построить еще один узел. Эта процедура состоит, как обычно, в обходе деревьев

всех доступных условных значений с нанесением метки на каждый пройденный узел, после чего просматривается весь массив ячеек для узлов, непомеченные узлы связываются в новый список свободных ячеек, а с помеченных снимаются метки. "Сборка мусора" может произойти во время копирования некоторого дерева, поэтому просматриваться должны не только "законченные" условные значения, но и частичные результаты копирования. В связи с этим копирование организовано таким образом, что новый узел создается лишь после того, как определены обе выходящие из него ветви (если окажется, что обе ветви совпадают, то есть представляют одно и то же конкретное значение, то узел не строится вовсе, см. п. 4). Если для будущего узла готова лишь одна из ветвей, то она записывается в магазин, где хранится до получения второй ветви. Этот магазин также просматривается во время сборки мусора.

Для перебора всех доступных в данный момент "законченных" условных значений ведется специальный список адресов таких значений; этот же список используется и при исключении предсказателей, когда нужно во всех доступных значениях подставить вместо исключаемого предсказателя выбранное для него значение (см. §4, п. 1). Этот список адресов организован как цепной список, используемый в режиме магазина, то есть новые элементы присоединяются и убираются только в начале списка. В начале работы системы этот список содержит адреса нескольких глобальных системных переменных. При исполнении каждого локального описания идентификатора, используемого в макроопределении, его адрес заносится в указанный список (для массивов заносится пара из начального и конечного адресов). При выполнении нового описания сменяемой переменной в список заносится адрес его прежнего значения. Наконец, адрес начала этого списка сам рассматривается как сменяемая переменная, переописываемая в каждом блоке, содержащем описания; поэтому при выходе из блока вся часть списка, надстроенная внутри этого блока, отбрасывается, и локальные идентификаторы блока становятся, таким образом, объектами для сборки мусора.

§6. Распределение регистров в рабочей программе

1. Для определенности в данной работе будем вести изложение в применении к архитектуре ЕС ЭВМ. Основной особенностью системы команд ЕС ЭВМ, приводящей к появлению альтернативных, но не равноценных вариантов программирования одной и той же последовательности операций, является возможность размещать величины или их адреса либо в оперативной памяти, либо на регистрах, причем при использовании регистров также возможны различные варианты. Поэтому наиболее трудной задачей при построении эффективной рабочей программы является рациональное использование регистров.

Система распределения регистров должна решать, по крайней мере, следующие задачи:

1) выделение базовых регистров для адресации команд текущей программной секции (командные регистры);
2) выделение базовых регистров для адресации статически доступных величин, то есть величин, относительные адреса которых вычисляются во время трансляции (статические регистры);
3) выделение регистров для хранения промежуточных результатов (анонимные регистры);
4) выделение регистров для постоянного использования в определенной роли в пределах некоторого блока (специальные регистры), например, выделение регистров для программирования циклов;
5) обеспечение многократного использования значений в регистрах и возможности использования информации, оставшейся в регистрах после предшествующих действий (остаточная информация).

В соответствии с принятой структурой рабочей программы статические и командные программы выделяются в начале трансляции процедуры, но при этом длина процедуры и ее статической секции еще не известны, поэтому не известно, сколько регистров потребуется для этой цели. В связи с этим каждый статический и каждый командный регистр, кроме первых, выделяется условно, для чего с каждым таким регистром связывается новый предсказатель.

Программа трансляции каждой конструкции оформляется в форме макроопределения на специальном языке, причем язык и реализующая его система выбраны так, чтобы максимально упростить написание макроопределений. При этом многие функции перекладываются на предсказатели, в частности, в этой системе внутренняя конструкция никогда не будет освобождать для себя анонимный регистр, занятый внешней конструкцией, так как одновременно с вариантом, где этот регистр был занят, будет рассматриваться вариант, в котором этот регистр или не был занят, или был освобожден той же конструкцией, которая его заняла.

С другой стороны, из-за того, что занятие и освобождение регистра является условным действием, состояние каждого конкретного регистра, если регистры выделять по конкретным номерам, будет зависеть от многих предсказателей, а условное выражение, определяющее номер свободного регистра, окажется еще сложнее. В связи с этим информацию, используемую для распределения регистров, нужно представить таким образом, чтобы по возможности уменьшить использование условных значений. Одним из средств для этого является использование вместо номеров регистров (которые могут быть и условными) символических имен этих регистров.

Если не стремиться к использованию остаточной информации в регистрах, то для выделения регистров можно использовать следующий простой алгоритм: ведется счетчик (условный), указывающий наибольший номер регистра, при отведении нового регистра значение этого счетчика увеличивается на единицу и выдается регистр с полученным номером, причем этому номеру сразу присваивается символическое обозначение (при помощи команды EQU); при освобождении регистра счетчик уменьшается на единицу. Если при увеличении на единицу счетчик выходит за пределы допустимых номеров регистров, то данный вариант бракуется (ему приписывается бесконечная стоимость). Разумеется, все операции над счетчиком являются условными. Аналогичным образом можно обеспечить выделение двойных регистров для команд типа M, SLDL, BXLE и т.п. Такие регистры можно выделять со стороны больших номеров, уменьшая каждый раз соответствующий счетчик на два до встречи (условной!) с первым счетчиком.

Для обеспечения многократности использования значений в регистрах и использования остаточной информации требуются более сложные приемы. Уточним, какого рода информацию мы предполагаем использовать. Для того чтобы при необходимости помещения на регистр некоторого значения можно было узнать, что такое значение уже находится на каком-либо регистре, нужно использовать какую-то содержательную характеристику этого значения. Предлагается учитывать следующие характеристики:

1) значение, которым обладает некоторый идентификатор;
2) результат разыменования значения типа 1);
3) результат выборки поля значений типов 1), 2);
4) значение является константой.

Будем называть такие значения узнаваемыми. Задача использования остаточной информации состоит из определения "времени жизни" остаточного значения и задания по возможности одного и того же регистра для различных использований значений с одинаковой характеристикой.

Отметим, что время жизни остаточного значения ограничивается следующими обстоятельствами:

1) загрузка в тот же регистр другого значения;
2) встреча с меткой, на которую управление могло быть передано до появления на регистре интересующего нас значения;
3) встреча с меткой, на которую могло быть передано управление из последующего текста, в котором данное остаточное значение прекращает существование;
4) вход в цикл, в теле которого данное остаточное значение прекращает существование;
5) слияние с ветвью программы (при входе из условного предложения или последовательного предложения, содержащего завершители), в которой этот же регистр используется иначе;
6) для значений, характеризуемых как результат разыменования, – присваивание соответствующему имени;
7) для значений, характеризуемых через идентификатор, – выход из блока, где описан этот идентификатор.

Заметим, что передачи управления и перегрузка регистров могут быть скрыты в вызовах процедур, однако, вызовы процедур (за исключением вызовов, реализованных особым образом) восстанавливают содержимое всех регистров. Кроме того, передача управления на метку данного блока из вызова возможна

только в том случае, когда область действия вызываемой процедуры или одного из ее фактических параметров меньше данного блока.

Таким образом, для использования остаточных значений необходимо в каждый момент времени иметь информацию обо всех узнаваемых значениях, находящихся в данный момент в регистрах. Естественно, что в описываемой системе вся эта информация будет условной, так как во-первых, выделение регистров происходит условно, а, во-вторых, для определения времени жизни в некоторых случаях используется информация о последующем тексте.

Далее мы опишем способ представления такой информации и способ выделения регистров, ориентированный на то, чтобы при повторном появлении узнаваемого значения ему присваивался по возможности тот же регистр, что и прежде.

2. Обычным решением задачи отведения регистров и использования остаточной информации (см., например, [2]) является следующее. Определяется таблица, содержащая по записи на каждый регистр (сюда включаются регистры с плавающей запятой, общие регистры и разряды кода условия в слове состояния программы). При генерации команды загрузки в регистр узнаваемого значения в соответствующую запись таблицы заносится характеристика этого значения. По этой таблице можно выбрать свободный для отведения регистр, а также определить, имеется ли интересующее нас узнаваемое значение на регистрах. Специальные приемы (часто весьма изощренные) применяются для определения порядка использования регистров в случае невозможности удовлетворения всех запросов на регистры.

В описываемой реализации ответственность за порядок использования регистров полностью перекладывается на транслирующую систему, однако, для нее не подходит описанная выше таблица использования регистров. Дело в том, что система использует специальное представление значений, а именно, двоичные деревья, узлы которых соответствуют предсказателям, а висячие вершины – конкретным значениям. Поэтому, если для каждого регистра хранить дерево характеристик, то сложно найти нужное узнаваемое значение, а если хранить список узнаваемых значений, то трудно определить номер очередного свободного регистра (придется просматривать деревья всех узнаваемых значений). Удобно иметь сразу две таблицы – таблицу регистров и таблицу узнаваемых значений с перекрестными ссылками.

Очевидно, что с помощью описываемой системы можно обеспечить полный перебор всех вариантов генерации программы в целом. Для этого нужно выбирать значения всех предсказателей в самом конце программы и для всех спорных случаев с помощью предсказателей создавать все возможные варианты. Например, при необходимости пожертвовать одним из узнаваемых значений на регистрах можно попробовать выгрузить все значения по очереди. Однако возникающее при этом огромное число различных вариантов не поддается уже никакому учету.

В данной работе предлагается некоторое компромиссное решение, которое при незначительных (по сравнению с полным перебором) потерях в качестве рабочей программы позволяет резко уменьшить число сравниваемых вариантов. Предлагается выбирать значения предсказателей в конце того макроопределения, в котором они были введены. При этом возможность повторного использования значений в регистрах в подконструкциях той конструкции, которой соответствует макроопределение, явным образом

учитывается системой, а использование остаточных значений управляется значительно более простыми алгоритмами (по таблице узнаваемых значений).

Перейдем к более детальному описанию предлагаемой техники использования регистров. Один из основных приемов повышения эффективности описываемых процедур состоит в возможно более частом использовании безусловных значений, в частности используется специальная операция `tree` для проверки тождественности деревьев константам (в данной реализации конкретные значения могут быть только неотрицательными целыми числами, а отрицательные числа представляют собой адреса условных значений; если формула `tree a` выдает неотрицательный результат, то `a` является безусловным значением). Заметим, что в любой процедуре все действия над локальными в ней значениями до первого условия или цикла, управляемого предсказателем, можно выполнять безусловно.

Определяем следующие два массива:

```
[N] struct (int RU, RB) R,
[M] struct (int NAME, REG) C;
```

Здесь `N` – количество регистров, подлежащих распределению, `M` – количество одновременно существующих на регистрах узнаваемых значений, допускаемых реализацией. Очевидно, что `M` может быть больше, чем `N`, так как в одном регистре могут быть разные узнаваемые значения в зависимости от значений предсказателей. `N` и `M` безусловные значения.

Поле `RU` может иметь следующие значения:

- 1, если соответствующий регистр свободен;
- 2, если регистр свободен, но содержит остаточное узнаваемое значение;
- значение, большее, чем 2, если регистр занят (в этом случае регистр `RU` отражает также кратность использования регистра, то есть количество значений на абстрактном стеке, доступных через данный регистр).

Значением поля `RB` является ссылка на массив `C`, если соответствующий регистр содержит узнаваемое значение, и 0 – в противном случае.

Поле `NAME` – характеристика значения (безусловная). Мы считаем, что характеристики каким-либо способом закодированы целыми числами.

Поле `REG` – номер регистра, в котором хранится данное узнаваемое значение. Если значение поля `REG` тождественно равно 0 (то есть является безусловным значением), то весь элемент массива считается неиспользуемым.

1.
```
proc SEARCHREG = (int j) int:
begin int k:=1; while k≤M and
(NAME of C[k]≠j or tree REG of C[k]=0) do k+:=1 od;
k end;
```

Процедура, состоящая только из безусловных действий, выдающая индекс элемента массива `C` с характеристикой `j` (`M+1`, если такого элемента нет). Предполагается, что в `C` для каждой характеристики всегда будет не более чем один элемент с полем `REG`, не равным тождественно нулю.

2.
```
proc  REPREG = (int k) void:
RU of R[REG of C[k]]+:=1;
```

Обеспечивает повторное использование регистра, содержащего узнаваемое значение с характеристикой `NAME of C[k]`.

3.
```
proc OCPREG1 = (int i) void:
begin ref int r = RB of R[i]; if r≠0 then
REG of C[r]:=0; r:=0 fi; RU of R[i]:=3 end;
```

Занимает регистр i неузнаваемым значением.

4.
```
proc OCPREG2 = (int i,j) void:
begin int k:=1; while tree REG of C[k]≠0 do k+:=1 od;
C[k]:=(j,i);
co Все предыдущие действия можно выполнять безусловно co
R[i]:=(3,k)
end;
```

Занимает регистр i узнаваемым значением с характеристикой j.

5.
```
proc FREEREG = (int i) void:
RU of R[i]:=if RB of R[i]=0 then 1 else R[i]-1 fi;
```

Эта процедура освобождает регистр i.

6.
```
proc GETREG = int:
begin int i:=1;
while (i≤N | RU of R[i]≠1 | false) do i+:=1 od;
if i≤N then i else i:=1;
while (i≤N | RU of R[i]≠2 | false) do i+:=1 od;
if i≤N then i else fail; skip fi
fi end;
```

Эта процедура выдает номер регистра, но оставляет его свободным; не сбрасывается даже информация об остаточном значении.

Если свободных регистров нет, то текущий вариант (то есть данный набор значений предсказателя) бракуется. На самом деле это не означает, что дальнейшее программирование невозможно вообще, так как предполагается, что параллельно с каждым вариантом, использующим регистры, был определен вариант программирования без использования регистров.

§7. Экспериментальная реализация

Для того, чтобы проверить изложенные выше методы синтеза рабочей программы, отработать основные алгоритмы и создать базу для последующей раскрутки транслятора, была предпринята экспериментальная реализация транслятора. В качестве инструментальной машины была выбрана ЭВМ "ОДРА 1204". В стандартное математическое обеспечение этой машины входит транслятор с практически полного языка АЛГОЛ 60 с дополнительными возможностями по обработке "адресной" информации и использованию машинных команд. В Вычислительном центре ЛГУ разработана удобная система для диалоговой отладки и исправления программ на базе этой ЭВМ.

1. Вся оперативная память, доступная для динамического распределения (примерно 5000 ячеек), разделяется на элементы по 4 ячейки в каждом. Перечислим способы использования этих элементов.

1.1. Список свободных элементов. В первой ячейке каждого элемента содержится адрес следующего свободного элемента; адрес первого элемента содержится в переменной `S`. Имеется процедура `free`, которая выдает в переменной `s` адрес очередного свободного элемента и продвигает `S`.

1.2. Список условных значений (деревьев), нужных в данный момент системе. Если в некоторый момент (например, при выходе из блока) условное значение становится недоступным, соответствующее дерево убирается из списка и тем самым становится объектом для сборки мусора. Список нужных условных значений ведется в магазинном режиме.

Имеются процедуры:

a) `inlist` – в список включается переменная, являющаяся фактическим параметром этой процедуры;

b) `arlist` – в список включаются элементы массива, расположенные между 2 переменными с индексами, являющимися параметрами этой процедуры;

c) `fromlist` – из списка исключается последний элемент.

Элементы этого списка имеют следующую структуру: адрес предыдущего элемента списка; адрес первого элемента массива; адрес последнего элемента массива (для простых элементов два последних адреса совпадают). Адрес последнего элемента списка содержится в переменной `idlist`.

Список нужных условных значений используется при сборке мусора и исключении предсказателей.

1.3. Основная часть элементов памяти расходуется для записи условных значений. Условные значения представляются целыми числами, причем неотрицательные числа представляют конкретные значения, не зависящие от предсказателей, а отрицательные числа представляют собой адреса деревьев со знаком "–". Каждый узел дерева представляется одним элементом памяти и содержит:

a) номер предсказателя;

b) условное значение, соответствующее значению предсказателя "истина";

c) условное значение, соответствующее значению предсказателя "ложь";

d) максимальный номер предсказателя в дереве, начинающемся с этого узла.

Последняя ячейка используется при подстановке значений предсказателей в деревья: если номер подставляемого предсказателя больше максимального номера предсказателя в дереве, то дерево не изменяется.

Имеются процедуры копирования деревьев (COPY), обхода деревьев (TREE), подстановки предсказателя (DELETE), минимизации дерева стоимости (MINIM), разметки дерева для сборки мусора (TREEGARB), вывода дерева в результирующий текст (outtree) и оптимизации деревьев (OPT и TAB). Алгоритмы этих процедур описаны в §5.

Например, копирование дерева A с выполнением над каждой висячей вершиной действия P и оптимизация результата записываются следующим образом:

```
OPT (COPY (A, TAB (P (x)))).
```

Процедура OPT определяет таблицу значений (см. §5.4), засылает туда начальные значения и после этого вызывает свой фактический параметр (COPY). Процедура COPY копирует дерево A и по каждой висячей вершине вызывает свой второй фактический параметр. Выполняется действие P, а затем процедура TAB заполняет некоторую строку в таблице значений.

В конце своей работы процедура OPT по построенной таблице значений производит оптимизацию результата.

Интересно отметить, что для эффективной реализации приведенного примера недостаточно средств даже такого мощного языка как АЛГОЛ 68. Дело в том, что при исполнении фактического параметра процедуры OPT может снова встретиться вызов процедуры OPT, поэтому желательно, чтобы каждый вызов этой процедуры отводил новую область памяти для таблицы значений, но тогда неясно, как передать эту область процедуре TAB, которая всегда является фактическим параметром процедуры OPT. Наиболее естественной реализацией таких случаев, видимо, является динамическая идентификация идентификаторов, при которой описание идентификатора определяется не в порядке вложенности описания процедур, а в порядке вызовов.

Заметим, что динамическую идентификацию можно эффективно реализовать с помощью сменяемых идентификаторов, определенных в данной работе (см. §2.3).

2. Одна из основных целей экспериментальной реализации состояла в создании основы для последующего создания компилятора с языка АЛГОЛ 68 для ЕС ЕВМ, работы над которым ведутся в ВЦ ЛГУ. Как уже было сказано, экспериментальная работа по синтезирующей части компилятора проводилась на ЭВМ "ОДРА 1204". Анализирующая часть компилятора также была написана на языке АЛГОЛ 60 и отлаживалась на этой же ЭВМ. Планируется, что в конце работы алгоритмы анализирующей части будут просто переписаны на языке АЛГОЛ 68. Синтезирующую часть компилятора (т.е. макроопределения конструкций языка) с самого начала предлагается писать на АЛГОЛе 68 с расширениями, описанными в данной работе (сменяемые переменные и предсказатели). Затем каждое макроопределение нужно преобразовать к записи на АЛГОЛе 60 (это чисто техническое преобразование и его можно выполнить автоматически), например,

```
if newvar then ... else ... fi
```

преобразуется к

```
IF:=NEWVAR; THEN; ... ELSE; ... FI;
```

```
b:=c<d
```

преобразуется к

```
ASSIGN (b, LT(c,d))
```

и т.д.

После этого на ЭВМ "ОДРА 1204" получился транслятор с языка АЛГОЛ 68 в машинный язык (точнее язык ассемблера) ЕС ЭВМ. Этим транслятором можно оттранслировать самого себя, после чего получится такой же транслятор, работающий на машинах типа ЕС ЭВМ.

На самом деле не предполагается, что первый вариант транслятора будет работать с полным АЛГОЛом 68, во всяком случае – его синтезирующая часть. В первый вариант войдут только те конструкции, которые нужны для раскрутки. Такие сложные конструкции, как предложения, гибкие массивы и некоторые другие будут реализованы на АЛГОЛе 68 в конце работы.

Таким образом, раскрутка предполагается многошаговой, при этом мы существенно опираемся на то, что транслятор создает рабочую программу.

3. В данной работе не представляется возможным подробно описать все макроопределения синтезирующей части компилятора. Для иллюстрации описанного в данной работе метода разберем макроопределение, соответствующее присваиванию.

Опишем вкратце общие соглашения по реализации. Запросы на значения подконструкций передаются в сменяемой переменой `box`. Одновременно с выдачей запроса помечается текущее положение указателя статической памяти (с помощью команды `EQU`). При ответе на внешний запрос каждая конструкция освобождает статическую память (с помощью команды `ORG`), возвращая указатель к тому положению, которое было при входе в конструкцию, оставляется только память для доставляемого значения. Для запоминания положения указателя статической памяти служит сменяемая переменная `mem`. Значения доставляются в ответ на запросы `F1`, `F2`, `FR` (свободные запросы). Информацию о размещении значений конструкций выдает процедура-функция `inner` (для запросов, отличных от свободных, выдается неопределенное значение). Запросы типа присваивания `A0`, `AS`, `AT` сопровождаются информацией об области памяти в сменяемой переменной `anst`.

Процедура `sysid` выдает очередной номер рабочего идентификатора.

Процедура `register` по способу размещения выдает номер регистра, используемого для представления значения и 0, если регистры не используются.

Процедура `identifier` по информации о размещении значения выдает характеристику идентификатора, если значением является идентификатор, и 0 – в противном случае.

В тексте на промежуточном языке присваивание представлено следующим образом: код присваивания, программа получателя, программа источника. Перейдем к описанию макроопределения.

Выдается запрос `FR` на получатель, вызывается процедура `inner` и в а запоминается способ размещения значения получателя. Затем запрос заменяется на `AS`, обрабатывается программа источника. Собственно присваивание выполняется программой источника (в ответ на запрос `AS`), а данное макроопределение должно теперь позаботиться о размещении значения присваивания (оно совпадает со значением получателя). Выходим из блока, в котором были описаны новые значения `box`, `mem`, `anst` и тем самым восстанавливаем их старые значения. Значение присваивания размещается в

соответствии с внешним запросом. Проще всего ответить на запрос V (пустой). Выдается команда ORG, если значение получателя использовало для своего представления регистр, то регистр освобождается.

Затем проверяется, является ли значение получателя идентификатором. В этом случае идентификатор мог бы появиться на регистре во время исполнения источника. Это можно использовать для повышения качества рабочей программы.

Далее обрабатываются внешние запросы F2, FR и F1 и запросы A0, AS, AT. В случае свободных запросов информация о размещении получателя выдается в качестве значения процедуры.

§8. Приложение метода к построению языков искусственного интеллекта.

С середины 50-х годов интенсивно развивается такая область применения ЭВМ, как "искусственный интеллект", включающая задачи поиска решения по дереву вариантов ответов или ситуаций. Типичными примерами в этой области являются программы управления роботами, программы, доказывающие теоремы, играющие в различные игры и т.д. Специфика таких задач стимулировала развитие и специальных языков программирования. Несмотря на различия в выборе базовых языков, методов реализации и т.п., языки искусственного интеллекта обладают рядом общих свойств (см. [3]):

a) наличие одной или несколько подразумеваемых баз данных, содержащих информацию о среде;
b) наличие развитых средств поиска по образцам, причем такой поиск может применяться не только к данным, но и к действиям (неявные вызовы процедур);
c) реализация обхода дерева решений с помощью механизма возвратов (backtracking);
d) дедуктивные свойства, позволяющие теоремы представлять процедурами, что (вместе с неявным вызовом процедур) позволяет автоматически строить цепочки выводов.

Кроме того, можно отметить, что большая часть таких языков построена на основе ЛИСПа и реализована интерпретирующими системами.

В данной работе нас будет интересовать механизм возвратов, применяемый практически во всех языках искусственного интеллекта.

Впервые механизм возвратов был четко сформулирован, по-видимому, в языке PLANNER [4,5]. Мы рассмотрим его, пользуясь "алголоподобными" методами описания.

Пусть обходится дерево решений. Есть средства для фиксации точек разветвления if ok then вариант1 else вариант2 fi и есть оператор, указывающий на возможность продолжения варианта (fail). Система, интерпретирующая текст на таком языке, встретив оператор ветвления, запоминает состояние памяти и запускает первый вариант. Если встретится fail, то система восстанавливает содержимое памяти в момент последней точки ветвления и запускает альтернативный вариант; если в данной точке ветвления вариантов больше нет, то рассматривается предыдущая точка ветвления.

В качестве примера рассмотрим задачу восьми ферзей (установить 8 ферзей на шахматной доске так, чтобы они не били друг друга).

```
[8] int a; for i to 8 do a[i]:=0 od;
proc p= (int i) void:
co устанавливает ферзя в i столбце на такую позицию, в которой
он не бьет 1, ... , i-1 ферзей (последняя занятая позиция в i столбце
фиксируется в a[i]); если такой позиции нет, то a[i]:=0 и выдается
fail co
    for i to 8 do
    m: p(i); if ok then
    if i=8 then print (a) fi else go to m fi od
```

Программа достаточно коротким и наглядным способом задает путь поиска решений, причем автор программы может совершенно не заботиться о технических деталях.

Однако механизм возвратов оказался хотя и удобным для авторов программ, но чрезвычайно неэффективным средством. Многократные запоминания состояний памяти, необходимость повторного вычисления многих участков программы (например:

```
if ok then A1 else A2 fi;
if ok then B1 else B2 fi;
if ok then C1 else C2 fi;
```

здесь операторы B1 и B2 выполняются по 2 раза, операторы C1 и C2 – по 4 раза и т.д.) заставили признать, что PLANNER поощряет "плохую практику программирования" (см. [3]).

Поэтому PLANNER так и не был реализован в полном объеме. G.J.Sussman (один из авторов первой реализации MICRO-PLANNER) сделал попытку преодолеть указанные трудности. Он предложил язык CONNIVER ([6]), в котором многие действия, выполняемые автоматически в языке PLANNER, должен указывать программист. В частности, автор программы определяет, значения каких переменных нужно запоминать в точках ветвления, задает явно контексты для выполнения операторов и т.д. Это позволило резко улучшить эффективность работы, но практически явилось шагом назад; как и в традиционных языках программирования, программист должен сам заботиться о многих технических деталях, что затемняет сущность решения задачи.

Нужно заметить, что G.J.Sussman сделал несколько интересных предложений и не связанных явно с повышением эффективности механизма возвратов.

Оператор fail был заменен на процедуру, вычисляющую текущую стоимость варианта. (Если написать work(S), то текущая стоимость увеличится на S; очевидно, что work(∞) эквивалентно fail).

Левосторонний обход дерева решений был заменен "параллельным" сравнением вариантов (если текущая стоимость варианта превысила некоторый порог, то вариант приостанавливается, а запускается следующий вариант, причем порог постепенно растет).

По-видимому, такой способ просмотра дерева решений лучше строгого левостороннего обхода, так как в большинстве задач нужно найти только первое решение (например, доказательство утверждения) и в момент получения решения малоперспективные решения не будут доведены до конца. По нашему мнению, описанные трудности в реализации языков искусственного интеллекта можно преодолеть с помощью введенного здесь метода синтеза рабочей программы.

Предлагается следующее решение.

1. Язык искусственного интеллекта строится на основе описанного в данной работе языка синтеза рабочей программы, дополненного некоторыми, ставшими уже традиционными, средствами (поиск по образцу, базы данных и т.д.).

2. Точки ветвления можно задавать, например, следующим образом:

   if ok then A1 else A2 fi.

3. Определяется явная функция стоимости варианта (аналогично work в CONNIVER) и точки, в которых должно происходить сравнение вариантов.

4. Никакого запоминания состояния памяти не предполагается. Варианты проверяются все одновременно, что отражается в специальной форме представления значений (описанные в данной работе двоичные деревья). Каждый оператор выполняется столько раз, сколько потребуется по алгоритму

программы; никаких перечислений не будет (разумеется, элементарные операции усложнятся). Можно рассчитывать, что в обрабатываемых значениях будут отражаться далеко не все варианты (ввиду локальности и ортогональности вариантов – см. §1).

Отказ от полного копирования состояния памяти в точках ветвления стал возможен в описываемой системе благодаря структурной организации программ, при которой фиксируются не только точки ветвления, но и точки воссоединения вариантов. (Благодаря отсутствию `go to` варианты, возникшие при разветвлении, сойдутся в статически известной точке.) Таким образом удается совместить наглядность программ, написанных на этом языке, и параллельное изучение вариантов с высокой эффективностью реализации (заметим, что язык ориентирован на реализацию с помощью компиляции, что также повысит эффективность).

Предполагается применить этот подход к построению языка для планирования действий робота.

Приложение 1. Пример макроопределения

```
proc assignation = int: (int a, b, k;
(new box:=FR; new mem:=sysid; EQU(mem);
a:=inner; box:=AS; mem:=sysid; EQU(mem);
new anst:=a; inner);
if box=V then ORG(mem); b:=register(a);
(b ne 0 | FREEREG(b)) else
if b:=identifier(a); b gt 0 then
k:=SEARCHREG(b); (k gt 0 | a:=regrept(k)) fi;
if box=F2 then box:=F1; mem:=sysid; EQU(mem); a
elif box=FR then a
elif ORG(mem); box=F1 then a
else (box=A0 | IMPAO | : box=AS | IMPAS | IMPAT)(a)
fi fi)
```

Приложение 2. Алгоритмы обработки условных значений

```
begin integer ur,tt;
integer if,thenelse,fi,coer,begtree,begtext,ass,nullchar,
GZ,IORG,IEQU,IEQU1,IDS,IRR,IRX,IRX1,t18,t20,t22;
if:=256; thenelse:=257; fi:=258; coer:=259;
begtree:=260; begtext:=261; ass:=262; nullchar:=263;
GZ:=32767; IORG:=264; IEQU:=265; IEQU1:=266; IDS:=267; IRR:=268;
IRX:=269; IRX1:=270; t18:=262144;
T20:=1048576; t22:=4194304;
setinput(0); read(ur,tt);
begin integer ur1,m,n,CURCOND,CURFORM,K,IF,r2,r3,f,min,
x,Y,P,i,j,s,VARNO,MAXVARNO,PLP,COST,RAB,idlist,S;
integer array BF[1:100],M[1:200],R[1:ur],
N[1:250],T[1:tt,1:2];

procedure BUF(a); integer a;
begin if P=100 then begin to drum(100,BF[1]);
P:=1 end else P:=P+1; BF[P]:=a
end;

procedure free;
begin s:=S; if s=ur1 then garbage collection; S:=R[S] end;
procedure inlist(a); integer a;
begin free; R[s]:=idlist; idlist:=s; R[s+1]:=R[s+2]:=ref(a)
end;

procedure arlist(a,b); integer a,b;
begin free; R[s]:=idlist; idlist:=s;
R[s+1]:=ref(a); R[s+2]:=ref(b)
end;

procedure fromlist;
idlist:=R[idlist];
?
```

```
procedure auxcopy;
begin integer mx,AA,BB,; AA:=M[m+1];
mx:=if x lt 0 then R[-x+3] else 0;
if AA lt 0 then begin BB:=R[-AA+3];
if BB gt mx then mx:=BB end;
BB:=R[r2]; if BB gt mx then mx:=BB;
free; R[s]:=BB; R[s+1]:=AA; R[s+2]:=x;
R[s+3]:=mx; x:=-s end;

integer procedure DELETE(b,f,t);
value f,t; integer b,f,t;
begin integer r1; r1:=m;x:=b;
m1: if x lt 0 then
begin if R[-x+3] lt f then go to m2;
if R[-x]=f then begin x:=R[-x+t]; go to m2 end;
m:=m+2; M[m]:=x; M[m+1]:=0; x:=R[-x+1];
go to m1 end;
m2: if m ne r1 then begin r2:=M[m];
if r2 lt 0 then begin M[m+1]:=x; M[m]:=-r2;
x:=R[-r2+2]; go to m1 end;
if x ne M[m+1] then auxcopy;
m:=m-2; go to m2 end else DELETE:=x end;

procedure MINIM(ff); value ff; integer ff;
begin integer i,j,r2,r3;
min:=LT(DELETE(COST,ff,1),DELETE(COST,ff,2));
i:=idlist; f:=ff;
m1: if i=0 then go to m2; r2:=R[i+1]; r3:=R[i+2];
for j:=r2 step 1 until r3 do
begin usak(+j); pska(r2);
r2:=OPT(COPY(r2,TAB(x))); pska(+j) end;
i:=R[i]; go to m1;
m2: outtree(min); BUF(ff); BUF(ass); f:=min:=0
end;
?
```

```
integer procedure COPY(b,d); integer b,d;
begin integer r1;
procedure auxin;
if R[-x+3] lt f then go to m2 else
if R[-x]=f then
begin integer f0,f1,f2;
f0:=f; f:=0; f1:=R[-x+1]; f2:=R[-x+2];
x:=COPY(min,COPY(if x=1 then f1 else f2,x));
f:=f0; go to m2 end;
r1:=m; n:=n+1; N[n]:=x; x:=b;
m1: if x ge 0 then x:=d else
begin if f ne 0 then auxin;
r2:=R[-x]; for i:=r1 step -2 until 2 do
if R[abs(M[i])]=r2 then begin x:=
R[-x+(if M[i] lt 0 then 1 else 2)]; go to m1 end;
m:=m+2; M[m]:=x; M[m+1]:=0; x:=R[-x+1]; go to m1 end;
m2: if m ne r1 then begin r2:=M[m]; if r2 lt 0 then
begin M[m+1]:=x; M[m]:=-r2;
x:=R[-r2+2]; go to m1 end;
if x ne M[m+1] then auxcopy;
m:=m-2; go to m2 end
else begin n:=n-1; COPY:=x end
end;

procedure outtree(a); integer a;
begin integer r1; r1:=m; x:=a; BUF(begtree);
m1: if x ge 0 then BUF(x) else
begin m:=m+1; M[m]:=x; x:=R[-x+1]; go to m1 end;
m2: if m ne r1 then begin r2:=M[m]; if r2 lt 0 then
begin M[m]:=-r2; x:=R[-r2+2]; go to m1 end;
Buf(-R[r2]); m:=m-1; go to m2 end
end;
?
```

```
integer procedure TREE(b,d); integer b,d;
begin integer r1; r1:=m; TREE:=x:=b;
m1: if x ge 0 then x:=d else
begin r2:=R[-x]; for i:=r1 step –2 until 2 do
if R[abs(M[i])]=r2 then begin
x:=R[-x+(if M[i] lt 0 then 1 else 2)]; go to m1 end;
m:=m+2; M[m]:=x; M[m+1]:=0;
x:=R[-x+1]; go to m1 end;
m2: if m ne r1 then begin r2:=M[m];
if r2 lt 0 then begin M[m]:=-r2; x:=R[-r2+2]; go to m1 end;
m:=m-2; go to m2 end
end;

procedure TREEGARB(a); integer a;
begin integer r1; r1:=m; x:=a;
m1: if x lt 0 then begin if R[-x] lt 0 then go to m2;
m:=m+1; M[m]:=x; x:=R[-x+1]; go to m1 end;
m2: if m ne r1 then begin r2:=M[m]; if r2 lt 0 then
begin M[m]:=-r2; x:=R[-r2+2]; go to m1 end;
R[r2]:=-R[r2]; m:=m-1; go to m2 end
end;

procedure PREF;
begin m:=0; x:=CURCOND;
m1: if x lt 0 then begin x:=-x; if R[x+1]=2 then begin i:=2;
m2: m:=m+2; M[m]:=sign(i-1.5)*x; M[m+1]:=0;
x:=R[x+i]; go to m1 end;
if R[x+2]=2 then begin i:=1; go to m2 end end
end;
?
```

```
integer procedure TAB(a); value a; integer a;
begin integer r1,r2; procedure d1;
begin r1:=T[i,r2];
T[i,r2]:=if r1=-1 or r1=a then a else –1 end; TAB:=a;
for i:=1 step 1 until VARNO do begin
for j:=2 step 2 until m do begin r1:=M[j];
r2:=if r1 lt 0 then 1 else 2; r1:=R[abs(r1)];
if i=r1 then begin d1; go to md end end;
for r2:=1,2 do d1; md: end end;

integer procedure OPT(a); integer a;
begin integer r1,r2,r3,r4;
procedure d2(b); value b; integer b;
begin free; R[s]:=i; if b lt 0 then
begin R[s+1]:=-b; R[s+2]:=r4 end else
begin R[s+1]:=r4; R[s+2]:=b end; r4:=-s end;
for i:=1 step 1 until VARNO do T[i,1]:=T[i,2]:=-1;
r1:=a; if r1 ge 0 then OPT:=r1 else
begin r4:=0; inlist(r4); inlist(r1);
for i:=1 step 1 until VARNO do
begin r2:=T[i,1]; r3:=T[i,2];
if r2 ge 0 then begin if r3 ge 0 then
begin free; R[s]:=i; R[s+1]:=r2; R[s+2]:=r3; R[s+3]:=i;
OPT:=-s; go to exit end; d2(-r2) end else
if r3 ge 0 then d2(r3) end;
OPT:=if r4=0 then r1 else COPY(r4,COPY(r1,x));
exit: fromlist; fromlist end
end;
?
```

```
procedure ASSIGN(a,b); value b; integer a,b;
begin integer r; inlist(b);
r:=m; m:=0;
a:=OPT(COPY(CURCOND, COPY(if x=1 then b else a,TAB(x)))); m:=r;
fromlist end;

integer procedure SLICE(a,i); value i; integer i;
integer array a;
begin inlist(i); SLICE:=OPT(COPY(i,COPY(a[x],TAB(x))));
fromlist end;

integer procedure ADD(a,b); value a,b; integer a,b;
begin inlist(a); inlist(b);
ADD:=OPT(COPY(a,COPY(b,TAB(N[n]+x))));
fromlist; fromlist end;

integer procedure SUB(a,b); value a,b; integer a,b;
begin inlist(a); inlist(b);
SUB:=OPT(COPY(a,COPY(b,TAB(N[n]-x))));
fromlist; fromlist end;

integer procedure LT(a,b); value a,b; integer a,b;
begin inlist(a); inlist(b);
LT:=OPT(COPY(a,COPY(b,TAB(if N[n] lt x then 1 else 2))));
fromlist; fromlist end;

integer procedure NEWVAR;
begin VARNO:=VARNO+1; if VARNO gt MAXVARNO then
MAXVARNO:=VARNO; free; R[s]:=R[s+3]:=VARNO;
R[s+1]:=1; R[s+2]:=2; NEWVAR:=-s end;

integer procedure PLUS(a); integer a;
PLUS:=OPT(COPY(COST,if x=GZ then TAB(x) else
COPY(a,if x=GZ then TAB(x) else TAB(N[n]+x*K))));
?
```

```
procedure BEGIN;
begin n:=n+3; N[n-1]:=VARNO; N[n]:=COST;
COST:=0; N[n-2]:=idlist; inlist(N[n]) end;

procedure END;
begin integer i,r1; r1:=N[n-1];
for i:=VARNO step –1 until r1+1 do MINIM(i);
VARNO:=r1; r1:=N[n]; ASSIGN(r1, PLUS(r1));
COST:=r1; idlist:=N[n-2]; n:=n-3 end;

procedure THEN;
begin BUF(if); n:=n+1; m:=0; N[n]:=OPT(COPY(CURCOND,
if x ne 1 then TAB(x) else COPY(IF,TAB(3-x))));
inlist(N[n]); CURCOND:=OPT(COPY(CURCOND,
if x ne 1 then TAB(x) else COPY(IF,TAB(x)))); PREF end;

procedure ELSE;
begin BUF(thenelse); RAB:=N[n]; N[n]:=CURCOND;
CURCOND:=RAB; PREF end;

procedure FI;
begin outtree(CURCOND); BUF(fi); m:=0; IF:=N[n];
CURCOND:=OPT(COPY(CURCOND,if x=1 then TAB(x)
else COPY(IF,TAB(x))));
PREF; fromlist; n:=n-1 end;
?
```

```
procedure WHILE;
begin n:=n+1; N[n]:=PLP; PLP:=n:=n+2;
usak(+Y); pska(RAB); N[n-1]:=RAB;
N[n]:=2; inlist(N[n]) end;

procedure DO;
begin BUF(if); N[PLP]:=OPT(COPY(N[PLP],
if x=1 then TAB(x) else COPY(CURCOND,
if x ne 1 then TAB(x) else COPY(IF,TAB(3-x)))));
m:=0; CURCOND:=OPT(COPY(CURCOND,if x ne 1 then
TAB(x) else COPY(IF,TAB(x)))); PREF end;

procedure OD;
begin n:=n+1; N[n]:=CURCOND; inlist(N[n]);
if CURCOND ne 2 then begin RAB:=N[PLP-1];
usak(RAB); pska(+Y) end else
begin for i:=n step -1 until PLP+1 do
begin outtree(N[i]); BUF(fi); fromlist end;
CURCOND:=N[PLP]; n:=PLP-3; PLP:=N[n+1];
fromlist; PREF end end;

procedure ASSINGSL(a,j,b); value j,b;
integer j,b; integer array a;
if j ge 0 then ASSIGN(a[j],b) else
begin integer r1,r2;
integer procedure aux;
begin aux:=0; for i:=r1+1 step 1 until n do
if x=N[i] then go to m1; n:=n+1; N[n]:=x;
m1: end;
r1:=n; TREE(j,aux); inlist(b);
for i:=r1+1 step 1 until n do
begin r2:=N[i]; a[r2]:=OPT(COPY(j,if x ne r2
then COPY(a[r2],TAB(x)) else COPY(CURCOND,
COPY(if x=1 then b else a[r2],TAB(x))))) end;
fromlist; n:=r1 end;
?
```

```
procedure garbage collection;
begin integer j,r1,r2,r3; i:=idlist;
m1: if i=0 then go to m2; r1:=R[i];
r2:=R[i+1]; r3:=R[i+2];
for j:=r2 step 1 until r3 do
begin usak(+j); pska(r2); TREEGARB(r2) end;
R[i]:=-r1; i:=r1; go to m1;
m2: for i:=3 step 2 until m+1 do TREEGARB(M[i]);
i:=-3; for i:=i+4 while i lt ur do
if R[i] lt 0 then R[i]:=-R[i] else
begin R[i]:=S; S:=i end; s:=S;
if s=ur1 then begin setoutput(0);
print('?There is no free main'); stop end
end;

procedure endtext;
begin K:=101-P; for i:=1 step 1 until K do BUF(nullchar);
N[1]:=MAXVARNO+1; N[2]:=drumplace;
drumplace:=0; todrum(2,N[1]);
format('?1234'); print(COST) end;
?
```

```
procedure bufstr(S); string S;
begin integer W,Y,Z; mok1(+1); usak(+5); pska(Z);
uswk(+Z); sdak(5); usak(0); dsak(3); pska(Y);
BS: usak(+Z); pska(X); BUF(X); dokj(Z);
odkj(Y); skd(BS) end;

procedure OUTR(A); integer A;
begin outtree(A); BUF(coer) end;

procedure ORG(N); integer N;
begin BUF(IORG); bufstr('ORG'); OUTR(N) end;

procedure EQU(N); integer N;
begin BUF(IEQU); OUTR(N) end;

procedure EQU1(N1,N2); integer N1,N2;
begin BUF(IEQU1); OUTR(N1); OUTR(N2) end;

procedure DS(N,K,type); integer N,K,type;
begin BUF(IDS); OUTR(N); OUTR(K); OUTR(type) end;

procedure RR(c,R1,R2); string c; integer R1,R2;
begin BUF(IRR); bufstr(c); OUTR(R1); OUTR(R2);
ASSIGN(COST,PLUS(2)) end;

procedure RX(c,R1,N); string c; integer R1,N;
begin BUF(IRX); bufstr(c); OUTR(R1); OUTR(N);
ASSIGN(COST,PLUS(4)) end;

procedure RX1(c,R1,D,R2,R3); string c; integer R1,D,R2,R3;
begin BUF(IRX1); bufstr(c); OUTR(R1); OUTR(D);
OUTR(R2); OUTR(R3); ASSIGN(COST,PLUS(4)) end;

procedure JUMP(c,N); string c; integer N;
begin BUF(IORG); bufstr(c); OUTR(N);
ASSIGN(COST,PLUS(4)) end;
?
```

```
procedure body(inlist,arlist,ASSIGN,SLICE,ADD,SUB,LT,NEWVAR,
BEGIN,END,IF,THEN,ELSE,FI,WHILE,DO,OD,ASSIGNSL,ORG,EQU,EQU1,
DS,RR,RX,RX1,JUMP,COST,CURCOND,R);
integer IF,COST,CURCOND;
integer array R;
integer procedure SLICE,ADD,SUB,LT,NEWVAR;
procedure inlist,arlist,ASSIGN,BEGIN,END,THEN,ELSE,FI,
WHILE,DO,OD,ASSIGNSL,ORG,EQU,EQU1,DS,RR,RX,RX1,JUMP;
drum;
?
```

```
P:=MAXVARNO:=VARNO:=0; ur1:=ur+1; K:=S:=i:=1;
in1: if i lt ur then begin i:=R[i]:=i+4; go to in1 end;
f:=x:=m:=n:=idlist:=COST:=PLP:=0;
inlist(CURCOND); inlist(COST); inlist(CURFORM);
inlist(x); inlist(min); inlist(IF);
drumplace:=100; BUF(begtext);
usak(3); pck(+1); pska(Y); CURCOND:=1;

body(inlist,arlist,ASSIGN,SLICE,ADD,SUB,LT,NEWVAR,
BEGIN,END,IF,THEN,ELSE,FI,WHILE,DO,OD,ASSIGNSL,
ORG,EQU,EQU1,DS,RR,RX,RX1,JUMP,COST,CURCOND,R);
endtext
end end
?
```

```
begin comment BACKPASS;
integer array R[1:2],BR,BW[1:100];
integer if,thenelse,fi,coer,begtree,begtext,ass,nullchar,
GZ,IORG,IEQU,IEQU1,IDS,IRR,IRX,IRX1,t18,t20,t22;
if:=256; thenelse:=257; fi:=258; coer:=259;
begtree:=260; begtext:=261; ass:=262; nullchar:=263;
GZ:=32767; IORG:=264; IEQU:=265; IEQU1:=266; IDS:=267;
IRR:=268; IRX:=269; IRX:=270;
t18:=262144; t20:=1048576; t22:=4194304;
drumplace:=0; fromdrum(2,R[1]);
begin boolean array predict[1:R[1]);
integer x,flag,pr,pw,dr,dw,c;

integer procedure get;
begin if pr=1 then begin dr:=dr-100;
drumplace:=dr; fromdrum(100,BR[1]);
pr:=100 end else pr:=pr-1; get:=BR[pr] end;

procedure put(x); integer x;
begin if pw=100 then begin drumplace:=dw;
dw:=dw+100; todrum(100,BW[1]);
pw:=1 end else pw:=pw+1; BW[pw]:=x end;

integer procedure comtree;
begin
ct1: x:=get; if x ge 0 then comtree:=x else
if predict[-x] then begin c:=0;
ct2: c:=c-sign(get); go to if c=-1 then ct1 else ct2
end else go to ct1;
for x:=get while x ne begtree do end;
flag:=0; dr:=R[2]; dw:=dr+1; pr:=1; pw:=0; put(begtext);
mk1: x:=get; if x=nullchar then go to mk1;
if x=ass then predict[get]:=comtree=1 else
if x=fi then flag:=if flag gt 0 then flag+1 else
if comtree=2 then 1 else 0 else
if x=thenelse then flag:=if flag=0 then 1 else
if flag=1 then 0 else flag else
if x=if then begin if flag gt 0 then
flag:=flag-- end else if x=begtext then go to exit
else begin if x=coer then x:=comtree;
if flag=0 then put(x) end; go to mk1;
?
```

```
exit: begin procedure outstr;
begin integer array STR[1:4]; integer x;
x:=get; STR[1]:=x; flag:=(x+5) div 3;
for c:=2 step 1 until flag do STR[c]:=get;
outstring(STR[1]) end;

procedure outnum;
begin format('12'); print(get) end;

procedure assname;
begin integer x; x:=get;
outchar(if x lt t18 then 113 else if x lt 1048576
then 88 else if x lt 4194304 then 82 else 70);
format('1234'); usak(x); mlak(262143);
pska(x); print(x) end;
pr:=pw+1; dr:=dw; copy(100,BW[1],BR[1]);
mk2: x:=get; if x=begtext then go to exit2;
line(1); if x=IEQU then begin assname;
print(' EQU *') end else
if x=IEQU1 then begin assname; print(' EQU ');
assname end else
if x=IDS then begin assname; print(' DS ');
outnum; c:=get; outchar(if c=1 then 120 else
if c=2 then 117 else 118) end else
begin space(6); outstr; outchar(64);
if x=IORG then assname else begin outnum; outchar(48);
if x=IRR then outnum else
if x=IRX then assname else
if x=IRX1 then begin outnum; outchar(72);
outnum; outchar(48); outnuml outchar(80)
end end end; go to mk2;
exit2: end end end
?
```

Литература

1. **Branquart P., Cardinael J. P., Lewi J.** *"An optimised translation process and application to ALGOL 68. Part 1: General Principles" //* MBLE Lab. de Rech., Report R209, Bruxelles, September 1972
2. **Д. Грисс** *"Конструирование компиляторов для цифровых вычислительных машин"* // М., "Мир", 1975
3. **Bobrow D. G., Raphael B.** *"New programming languages for AI research" //* Xerox Palo Alto Research Center, August 1973
4. **Hewitt C.** *"Description and theoretical analysis (using schemata) of Planner: A language for proving theorems and manipulating models in a robot"* // AI Memo No. 251, MIT Project MAC, April 1972
5. **Sussman G. J., Winograd T.** *"Micro-Planner reference manual"* // AI Memo No. 203, MIT Project MAC, July 1970
6. **McDermot J., Drew V., Sussman G. J.** *"The Conniver reference manual"* // AI Memo No. 259, MIT Project MAC, May 1972
7. **Терехов А. Н., Цейтин Г. С.** *"Средства эффективного синтеза объектной программы"* // "Программирование", №6, Москва, 1975
8. **Терехов А.Н., Цейтин Г. С.** *"Язык синтеза объектной программы с учетом последующего контекста"* // В сб.: "Труды всесоюзного симпозиума по методам реализации новых алгоритмических языков, часть II", Новосибирск, 1975
9. *"Алгол 68. Методы реализации"* // Коллективная монография под ред. Г. С. Цейтина, Л., ЛГУ, 1976
10. **Терехов А. Н.** *"Распределение регистров в рабочей программе"* // "Программирование", №1, 1977

ОГЛАВЛЕНИЕ